Alternative Techniques for Built-In Self-Test of Field Programmable Gate Arrays

Except where reference is made to the work of others, the work described in this thesis is my own or was done in collaboration with my advisory committee. This thesis does not include proprietary or classified information.

Aditya Newalkar

Certificate of Approval:

Victor P. Nelson
Professor
Electrical and Computer Engineering

Charles E. Stroud, Chair,
Professor
Electrical and Computer Engineering

Foster Dai
Associate Professor
Electrical and Computer Engineering

Stephen L. McFarland
Acting Dean, Graduate School

Alternative Techniques for Built-In Self-Test of Field Programmable Gate Arrays

Aditya Newalkar

A Thesis

Submitted to

the Graduate Faculty of

Auburn University

in Partial Fulfillment of the

Requirements for the

Degree of

Master of Science

Auburn, Alabama
August 8, 2005

Alternative Techniques for Built-In Self-Test of Field Programmable Gate Arrays

Aditya Newalkar

Permission is granted to Auburn University to make copies of this thesis at its discretion, upon the request of individuals or institutions and at their expense. The author reserves all publication rights.

Signature of Author

Date

Copy sent to:

Name	Date

Vita

Aditya Newalkar, son of Anil and Sugandha Newalkar was born on April 17, 1979, in Mumbai, India. He graduated with Bachelor of Engineering degree in Electronics Engineering from Mumbai University in December 2000. What started as a student project at Indian Institute of Technology (IIT), Powai, Mumbai in year 1999 grew into valuable two year research experience for him after graduating from Mumbai University. While in pursuit of his Master of Science degree at Auburn University, he received guidance of Dr. Charles Stroud in the Electrical and Computer Engineering department. He worked as an intern in Medtronic Navigation in Louisville, CO.

Thesis Abstract

Alternative Techniques for Built-In Self-Test of Field Programmable Gate Arrays

Aditya Newalkar

Master of Science, August 8, 2005
(B.E., Mumbai University, 2000)

174 Typed Pages

Directed by Charles E. Stroud

In the Built-In Self-Test method of testing the logic and interconnect resources of the Field Programmable Gate Arrays (FPGAs), configuration time and time to retrieve of the test results dominates the duration of the test. The techniques presented in this thesis offer reduction in the configuration time and result retrieval time for the Built-In Self-Test using partial reconfiguration and partial configuration memory readback. Though the work has been done targeting Xilinx Virtex-I and Spartan-II FPGAs, the method is general enough to be applied on any FPGA featuring Partial Run Time Reconfiguration (PRTR). We also evaluate the Computer Aided Design (CAD) tools that are mainly used for partial reconfiguration, for their usefulness in generating test configurations for the programmable interconnect and logic resources of an FPGA using the Built-In Self-Test method.

Acknowledgments

The author would like to thank Dr. Charles Stroud for giving him insight on the subject of Built-In Self-Test for FPGAs. The author admires his relentless pursuit for quality and is thankful for his patience. Special thanks to author's family, Anil, Sugandha, Bhau Newalkar and Siddharth Tambe for their unconditional love, support and continuous encouragement. The author thanks all friends from Auburn for making his time enjoyable. Finally, principles by which Mohandas Gandhi lived his life give author strength and inspiration.

Style manual or journal used Journal of Approximation Theory (together with the style known as "aums"). Bibliograpy follows van Leunen's *A Handbook for Scholars*.

Computer software used The document preparation package TEX (specifically LATEX) together with the departmental style-file `aums.sty`.

Table of Contents

List of Tables xi

List of Figures xiii

1 Introduction 1
1.1 FPGA Architecture . . . 1
1.1.1 Programmable Logic Blocks . . . 1
1.1.2 Programmable Interconnection Network . . . 3
1.1.3 Programmable I/O Cells . . . 4
1.2 Flow of Design with FPGAs . . . 5
1.3 Advantages of FPGAs . . . 5
1.4 Reconfigurable Computing . . . 6
1.4.1 Dynamic Reconfiguration . . . 7
1.4.2 Static Reconfiguration . . . 8
1.4.3 Partial Reconfiguration . . . 8
1.5 Testing of FPGAs . . . 9
1.6 Built-In Self Test . . . 10
1.6.1 BIST for FPGAs . . . 10
1.7 Thesis Statement . . . 12

2 Review of Partial Reconfiguration and BIST 14
2.1 Architecture of Virtex-I and Spartan-II FPGAs . . . 14
2.1.1 PLB Architecture . . . 14
2.1.2 Interconnect Architecture . . . 17
2.1.3 Block RAMs . . . 20
2.2 Configuration of the FPGA . . . 20
2.2.1 SelectMAP Mode . . . 21
2.2.2 Boundary Scan Mode . . . 21
2.2.3 Start-up Sequence . . . 23
2.3 Configuration Memory Architecture of Virtex-I and Spartan-II FPGAs . . . 23
2.3.1 Addressing . . . 24
2.3.2 Frame Organization . . . 27
2.3.3 Configuration Registers . . . 28
2.3.4 Full Reconfiguration Bitstream . . . 32
2.4 Readback . . . 33
2.4.1 Readback Verification . . . 33
2.4.2 Readback Capture . . . 33
2.4.3 Readback Operations . . . 34

2.5 Partial Reconfiguration 35
2.5.1 Partial Reconfiguration without Shutdown Sequence 35
2.5.2 Partial Reconfiguration with Shutdown Sequence 36
2.5.3 BitGen 36
2.5.4 JBits 37
2.6 BIST for FPGAs 48
2.6.1 Logic BIST 48
2.6.2 Interconnect BIST 50
2.6.3 BIST for Xilinx FPGAs 54
2.6.4 Using JBits API to Generate Interconnect BIST Configurations . . 57
2.7 Thesis Statement 59

3 Partial Reconfiguration and Readback for Logic BIST 61
3.1 Floorplan of Logic BIST to Aid Partial Reconfiguration 62
3.2 Generating Partial Reconfiguration Files 63
3.2.1 Using BitGen 64
3.3 Generating a Test Plan for Logic BIST 66
3.4 Experimental Results for Logic BIST 73
3.5 Partial Configuration Memory Readback to Retrieve the BIST Results . . 76
3.5.1 Commands for Partial Configuration Memory Readback 80
3.6 Summary 81

4 Generating Routing BIST Configurations using JBits 83
4.1 Overview of Routing BIST Architecture 84
4.1.1 Testing the Interconnects in Parallel 86
4.2 The Routing BIST RTPCores 87
4.2.1 Configuring the TPG 91
4.2.2 Configuring the ORA 92
4.2.3 Routing the WUTs 93
4.2.4 Populating the PLB Array 94
4.2.5 Generating the XDL File 98
4.3 Experimental Results of Routing BIST 98
4.3.1 Partial Reconfiguration and Routing BIST 102
4.3.2 Test Phase Sequence 104
4.4 Calculation of the Total Number of Interconnect BIST Configurations Required 106
4.4.1 Hex Interconnects 107
4.4.2 Single Interconnects 108
4.4.3 Switch Box CIPs 108
4.4.4 MUX CIPs 117
4.5 Generating Configurations for Switch-Box CIPs 120
4.6 Conclusion 122

5 Summery and Future Work 123

Bibliography 127

Appendices 132

A Steps in Writing Parent RTPCore 133

B Steps in Writing Child RTPCores 138

C Complete Program Source 140

D Complete List of Connections Between the Mux CIPs 158

List of Tables

2.1 Virtex TAP Controller Pins . . . 22

2.2 Constants Used in the Address Calculation [Xil03d] . . . 26

2.3 Variables Used for Address Calculation [Xil03d] . . . 26

2.4 Calculating the Location of the LUT RAM Bit in Virtex-I Bitstream [Xil03d] . . . 27

2.5 Equations for Calculating PLB FF Location in the Bitstream [Xil03d] . . . 27

2.6 PLB Column Frame Organization . . . 27

2.7 Configuration Registers [Xil03d] . . . 29

2.8 Command Header Format [Xil02d] . . . 29

2.9 Configuration Commands and their Usage [Xil03d] [Xil04] . . . 30

2.10 Readback Commands Required to Perform Readback on PLB Configuration . . . 34

2.11 Classes Used for Bit Level Manipulation of PLB Elements [Xil01d] . . . 42

2.12 Classes Used for Bit Level Manipulation of Switch Box CIPs[Xil01d] . . . 43

2.13 Classes Used for Bit Level Manipulation of Output MUX CIPs[Xil01d] . . . 44

2.14 Classes Used for Bit Level Manipulation of input MUX CIPs[Xil01d] . . . 44

2.15 Model of Interconnect Resources in the Package com.xilinx.JRoute2.Virtex.ResourceDB [Xil01d] . . . 45

3.1 Command Listing for Scenario 1 . . . 70

3.2 Command Listing for Scenario 2 . . . 71

3.3 Command Listing for Scenario 3 . . . 72

3.4 Command Listing for Scenario 4 72

3.5 Partial Frames 73

3.6 Partial Frames 73

3.7 Sizes of Partial Bitstreams vs. Full Bitstreams 75

3.8 Comparison of Boundary Scan Access Method and Partial Configuration Memory Readback 79

3.9 Bitstream for Partial Configuration Memory Readback 81

4.1 Connectivity between Output Multiplexers and Interconnects 87

4.2 Input and Output ports of TPGCounterCore 91

4.3 Input and Output Ports of ORACore 93

4.4 Command Line Arguments Available for the JBits Program 99

4.5 Possible Values of the Command Line Arguments 100

4.6 Command Listing for Generating Partial Bitstreams for Routing BIST . . 103

4.7 Sizes of Partial Bitstreams vs. Full Bitstreams 104

4.8 Routing BIST and Test Phase Sequence 106

4.9 Mapping of the CIPs in Various JBits Classes [Xil01d] 111

4.10 MUX CIPs in Virtex-I Architecture and their Functions [Xil01d] 119

4.11 MUX CIP Groups Tested in Parallel 119

4.12 Testing MUX CIPs for Stuck-On and Stuck-Off Faults [SWHA98] 120

C.1 Input and Output ports of LUT5 140

D.1 Mux CIPs Mux28to1 and Connecting Single Interconnects 158

D.2 MUX CIPs Mux16to1 and Connecting Interconnects 159

List of Figures

1.1 General Architecture of FPGA 2

1.2 Typical Architecture of PLB [Str02] 3

1.3 Typical CIP Structure 4

1.4 Spatial Vs. Temporal Computing [DeH00] 7

1.5 BIST for FPGA [AS01] 11

2.1 Internal Architecture of Virtex-I Slice [Xil01b] 16

2.2 Different Types of CIPs Found in FPGA [SWHA98] [FH03] 18

2.3 Switch Box CIP and Xilinx Interconnect Architecture 19

2.4 Block RAM in Virtex-I and Spartan-II FPGAs 20

2.5 Xilinx Virtex-I and Spartan-II Addressing Scheme [Xil03d] 25

2.6 Design Flow of the Application with JBits 38

2.7 JBits Program for Manual Routing 47

2.8 BIST for FPGA Interconnect Resources [SWHA98] 51

2.9 FPGA Floorplan for Online Interconnect Testing [AES01] 53

2.10 FPGA Floorplan with "Galaxy" BIST [SNLA02] 54

2.11 Complete Testing of Switch Boxes [SWHA98] 56

2.12 Scan Cell Interfacing with IEEE 1149.1 [HGWS99] 58

3.1 Floorplan for BIST Test Session 63

3.2 Four Different Test Plans for Testing Two Slices 69

4.1 Horizontal and Vertical Interconnect Resources Tested for Shorts and Opens 85

4.2 Hex and Single Wires Tested for Shorts and Opens 88

4.3 Configuration of Comparator-Based ORA 94

4.4 JBits Program for Routing the WUTs 95

4.5 Boundary Condition for Populating PLB Array in Vertical Direction . . . 97

4.6 Switch Box CIP and Xilinx Interconnect Architecture 107

4.7 Sections of Switch Box CIP 110

4.8 Test Configurations Needed to Completely Test Switch Box CIPs 113

4.9 Test Configurations Continued... 114

4.10 Routing BIST Configuration for Testing Mux CIPs 115

4.11 Testing MUX CIPs in Parallel [RPFZ99] 118

4.12 Problem of Undetected Faults Due to Invisible Logic in MUX [AS01] . . . 118

A.1 JBits Program for Instantiating a Counter core 136

A.2 JBits Program Continued... 137

C.1 Configuration of LUT5 RTPCore 141

C.2 JBits Program for User Interaction and Populating the PLB Array 143

C.3 JBits Program Continued... 144

C.4 JBits Program Continued... 145

C.5 JBits Program Continued... 146

C.6 JBits Program Continued... 147

C.7 JBits Program Continued... 148

C.8 JBits Program Continued... 149

C.9 JBits Program Continued... 150

C.10 JBits Program Continued... 151

C.11 JBits Program Continued... 152
C.12 JBits Program Continued... 153
C.13 JBits Program Continued... 154
C.14 JBits Program Continued... 155
C.15 JBits Program Continued... 156
C.16 JBits Program Continued... 157

Chapter 1

Introduction

Field programmable gate arrays (FPGAs) have evolved from simple *programmable logic devices* (PLDs) like *programmable array logic* (PALs) and *programmable logic arrays* (PLAs). These early devices were used in digital design as glue logic. As the need of the digital system designers grew from simple decoders to more complicated designs like protocol resolvers, multiple PLDs were connected through a programmable routing architecture to form the FPGA [BR96]. This architecture gives the user the ability to program the interconnects to realize various types of complex digital designs. Due to their size and programmability, testing modern FPGAs has become a complex and time consuming task.

1.1 FPGA Architecture

The Figure 1.1 shows the general architecture of a typical FPGA. The FPGA consists of uncommitted resources of an NxM array of *programmable logic blocks* (PLBs), programmable *input and output* (I/O) cells, a programmable interconnection network, and a configuration memory to program the device.

1.1.1 Programmable Logic Blocks

The PLBs of most FPGAs contain multiplexers, *look up tables* (LUTs) and flip-flops. An important characteristic of a PLB is its *functionality*, defined as the number of different boolean functions that it can implement [BFRV92]. The elements in the

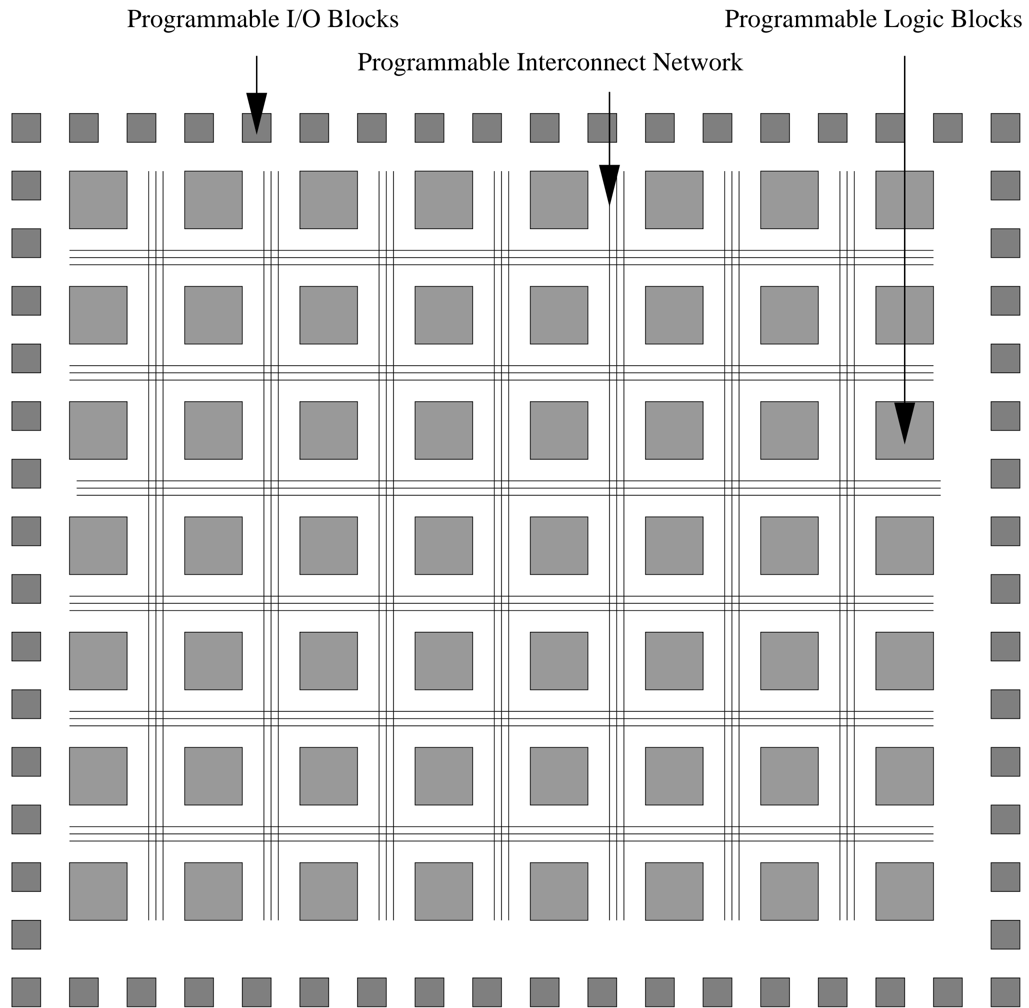


Figure 1.1: General Architecture of FPGA

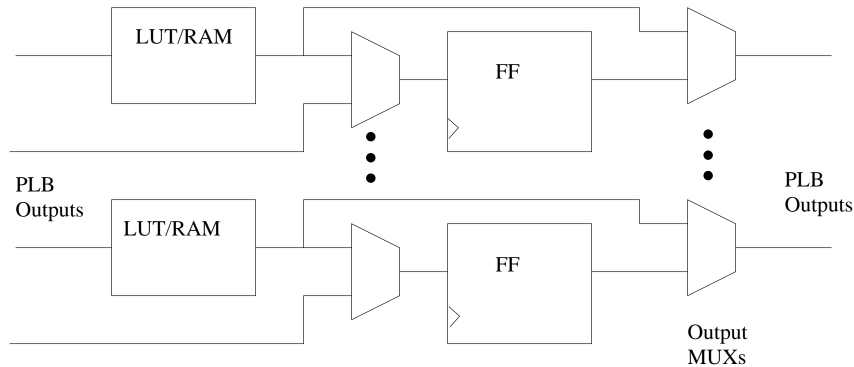


Figure 1.2: Typical Architecture of PLB [Str02]

PLB architecture can be programmed to function in different modes of operation. The LUT can be used either in a LUT mode of operation or *random access memory* (RAM) mode of operation. In the LUT mode, the element can implement combinational logic functions of multiple inputs (typically 3 to 4). In the RAM mode of operation, the PLB can logically be configured to behave either as a synchronous or asynchronous, single port or dual port RAM. The flip-flops can be configured in latch mode or edge-triggered mode, with asynchronous or synchronous preset/clear, and programmable clock enable. The multiplexers can be selected to connect the LUT outputs to the flip-flops or to bypass the flip-flops [Str02]. Thus the PLB contains the functionality to implement any combinational or sequential logic functions using the logic resources in the architecture as shown in the Figure 1.2. The mode of operation for each element is selected when the device is programmed or *configured*.

1.1.2 Programmable Interconnection Network

The programmable interconnect network in an FPGA, also called its routing architecture [BFRV92], consists of segments of wires of various lengths and programmable

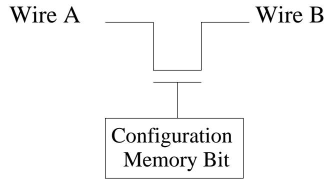


Figure 1.3: Typical CIP Structure

switches. There are global routing resources and local routing resources. Global routing resources facilitate the routing of the signals between the PLBs that are separated by other PLBs. Local routing resources facilitate the routing between PLBs that are next to each other in the array [SNLA02]. The connections are made via *configurable interconnect points* (CIPs), also referred to as *programmable interconnect points* (PIPs). A CIP consists of a transmission gate controlled by configuration memory bit. As shown in the Figure 1.3, the connection between the wire segments A and B is made or broken depending upon the logic value of the configuration memory bit [Str02].

1.1.3 Programmable I/O Cells

Most of the I/O pins of the FPGA can be configured in input, output or bi-directional mode of operation. The I/O cells can also be programmed as registered or latched I/Os depending on the design. The I/O cells support TTL as well as CMOS I/O standards thereby eliminating need for the voltage shifters for interfacing [Lat02] [Xil01b].

1.2 Flow of Design with FPGAs

The circuit designer typically implements the design in a *hardware description language* (HDL) and synthesizes the circuit description with the help of one or more *computer aided design* (CAD) tools. The CAD tools generate a *bitstream* file that contains programming instructions and data to establish the application specific system functionality of the various programmable resources of the device like PLBs, routing architecture and I/O blocks. This bitstream file is then loaded into the FPGA chip using one of the configuration interfaces provided for the FPGA [Jay01]. The process of loading a design-specific bitstream into one or more FPGAs to define the functional operation of the PLBs, the interconnect resources and I/O blocks is known as *configuring* or downloading the bitstream to the device [Xil99]. The significance of the FPGA design lies in the fact that *static random access memory* (SRAM) based FPGAs can be reconfigured an unlimited number of times, implementing a different design each time. In order to implement a different design simply requires overwriting the previous configuration loaded into the SRAM with a new bitstream through the configuration interfaces provided by the FPGA manufacturer [Xil99].

1.3 Advantages of FPGAs

FPGAs provide a low cost solution to low volume products where user programmability is needed at the deployment time. *Application specific integrated circuits* (ASICs) edge out FPGAs in high volume products in terms of unit costs and performance parameters, which are significantly better than FPGAs. The reason for significant difference in performance parameters is that the flexibility provided by the programmability in

FPGAs is at the cost of substantial signal delays and area overhead introduced by the programming circuitry [AR94]. However, FPGAs offer some advantages over ASICs including:

- Low cost solution for low volume applications,
- Low *non-recurring engineering* (NRE) costs, and
- Rapid prototyping [Mil94].

1.4 Reconfigurable Computing

Traditionally, software is considered to be a component that is flexible, relatively slow and inefficient compared to hardware. The hardware is perceived to be customized to the problem and faster compared to the software [DW99]. Hardware can be designed to execute functions concurrently. Therefore, at any given time there are multiple computing elements actively performing their functions. This is referred to as spatial computing. In a conventional processor, instructions are executed serially using memory or registers to store the program variables. This is called temporal computing. Figure 1.4 shows examples of spatial and temporal computing. A conventional digital signal processor (DSP) would take multiple instruction cycles to execute a filter algorithm (Figure 1.4(b)), while the spatial implementation of the same filter in an FPGA gives a new result every cycle in a pipelined fashion, thus higher throughput is observed (Figure 1.4(a)) [DeH00]. The idea of reconfigurable computing tries to bring together best of both worlds. Reconfigurable computing relies on devices, like FPGAs, that are user programmable an unlimited number of times. The more generalized resources or structures like LUTs, flip-flops, and SRAMs, that are provided in an FPGA, can be configured to

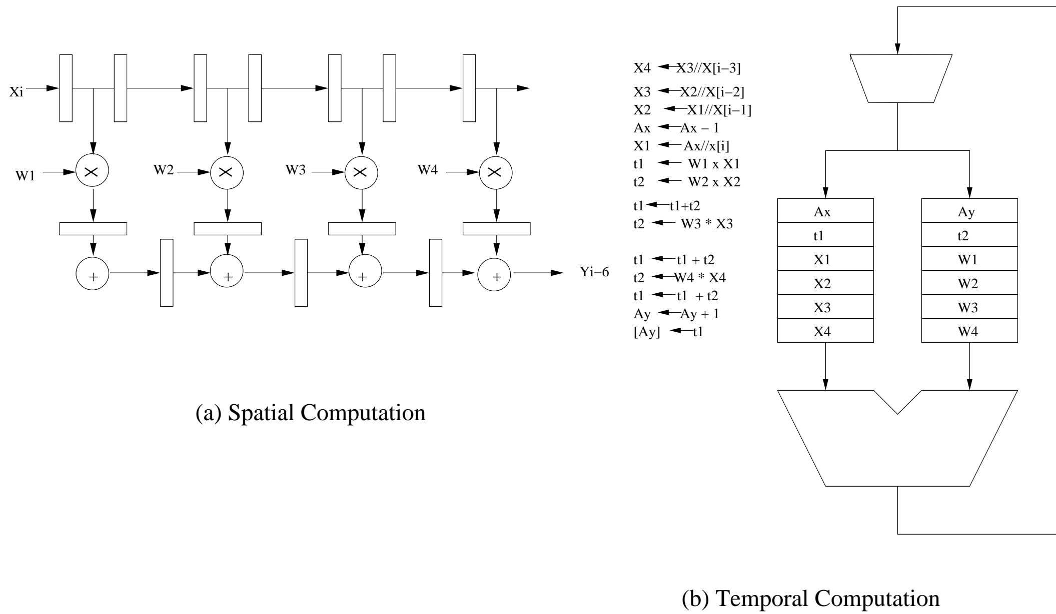


Figure 1.4: Spatial Vs. Temporal Computing [DeH00]

execute the functions spatially. The process of reconfiguration gives the ability to load different functions in the FPGA serially in time, taking advantage of temporal computing [DeH00] [DW99]. Because of this inherent parallelism in the FPGA architecture, these devices frequently show an order of magnitude higher performance than a general purpose processor [DeH00][CHW00][GSB$^+$00] [DW99].

1.4.1 Dynamic Reconfiguration

Dynamic or runtime reconfiguration is a process where the reconfigurable unit is configured without interrupting the configured system function [ESSA00]. The reason for such an arrangement might be that the design is partitioned into many small parts, either too large or too many to fit in the FPGA simultaneously. These partitions of the original design can be loaded into the FPGA without interrupting the function of other

partition(s) already loaded into the FPGA. An external agent like a microprocessor may be used to control which partition(s) is(are) loaded and in what order [ESSA00].

1.4.2 Static Reconfiguration

Static or compile time reconfiguration is an idea that can be defined as an inverse of dynamic reconfiguration, where the reconfigurable unit is configured while it is idle or inactive. Most FPGAs are capable of reading the configuration data from an *electrically erasable programmable read-only memory* (EEPROM) when the power is turned on [Xil99]. This is referred to as power-on configuration. This is an example of static reconfiguration.

1.4.3 Partial Reconfiguration

Complete reconfiguration of an FPGA chip can be an onerous process [HLS98]. The configuration time varies depending on the size of the bitstream to be loaded into the FPGA. This delay could be unacceptable in high performance systems, which are expected to be reconfigured many times to execute the system function. If the circuit implemented in one configuration is not significantly different from the one implemented in the next configuration, the configuration time can be reduced if the next configuration bitstream were to contain only the programming instructions and data for the programmable resources of the FPGA that are configured differently from the previous configuration. The size of the partial reconfiguration bitstream is now reduced as it contains only the difference between the prior full configuration and the subsequent reconfiguration [HLS98]. The problem in implementing such a scheme is that the FPGAs should have architectural support for configuring only part of their programmable resources,

referred to as partial reconfiguration. For the applications that need to reconfigure only part of their logic depending on the circumstances, this raises an exciting possibility of gaining significant time advantage. Fault-tolerant applications are one example of such applications that benefit from partially reconfigurable architectures [ESSA00].

1.5 Testing of FPGAs

Commercially available FPGAs have reached gate counts of 8 million, feature banks of RAMs, hundreds of user I/Os and are capable of running at clock speeds of 400 MHz [Xil02e]. Such high performance FPGA-based systems, when subjected to aging and environment (temperature, humidity, vibration, cosmic rays and α-rays) are vulnerable to faults [ESSA00]. Therefore having a good FPGA testing method is ever more essential.

Testing of FPGAs poses a different set of challenges than ASICs. The challenge is to test PLBs as well as interconnect resources in all possible modes of operation. It is an important consideration for safety-critical applications because if the test methodology tests only the normal mode of operation for a given system function, when the FPGA is reconfigured to implement different system function, the latent faults may take over and hamper the system function [AS01] [SS99]. Testing the PLBs and interconnect in all possible modes of operation is advantageous for the fault-tolerant applications to identify if any particular mode of PLB operation is faulty so it can be used in one of the other fault-free modes. The testing method is required to detect single and multiple faults in PLBs and interconnects. Meeting this requisite entails selection of a method that is capable of in-system testing. Ideally, the testing method should not introduce any overhead of area and delay penalties [AS01]. Diagnostics provided by the test method

should enable the user to identify and locate the defective module for fault-tolerant applications [SNLA02].

1.6 Built-In Self Test

Built-In Self Test (BIST) is a *design-for-testability* (DFT) approach in which testing (test pattern generation, application and output response analysis) is accomplished through built-in hardware features [AKS93]. In other words, the BIST circuitry is part of the hardware that it tests. One of the advantages of BIST includes the capability of in-system testing without the need of external test equipment.

For ASIC testing, BIST has area overhead and delay penalties. However, when viewed in the context of FPGAs, BIST offers a unique advantage over the external or internal dedicated BIST circuitry: SRAM based FPGAs are reconfigurable and are capable of implementing any given design. While testing, the FPGA may be configured as a BIST circuit, the tests are run and the results are obtained. If the device passes the test, it can be reconfigured to implement the desired system function. If one or more faults are detected and identified, the system function can be reconfigured to avoid the fault(s) for fault-tolerant applications. Thus, the BIST circuit would "disappear" in the reconfiguration process after testing of the device is complete. Therefore, it can be said that testability is achieved without any area or performance penalties [AS01].

1.6.1 BIST for FPGAs

An example of the structure of BIST for FPGAs is as follows: a group of PLBs are configured as *test pattern generators* (TPGs), *blocks under test* (BUTs) and *output*

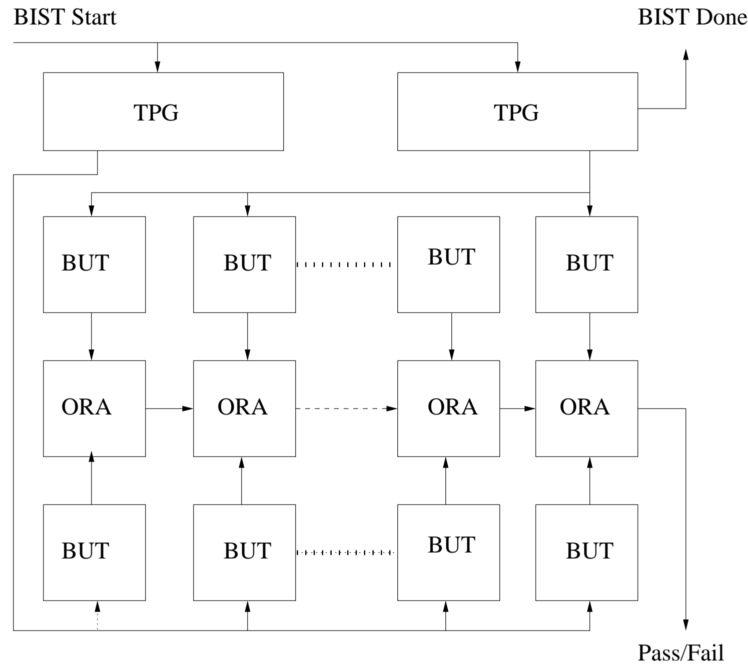


Figure 1.5: BIST for FPGA [AS01]

response analyzers (ORAs) as illustrated in Figure 1.5. The TPGs generate test patterns that are applied as inputs to the BUTs. All BUTs are configured and tested in identical modes of operation. The outputs of the identically programmed BUTs for a set of test patterns generated by TPGs are compared by the *output response analyzers* (ORAs) giving a single Pass/Fail indication at the end of the BIST sequence depending on whether a mismatch in the BUT outputs was observed [SCKA96] [SKCA96] [AES01] [SNLA02] [AS01]. The structure of TPGs can be as simple as a N-bit counter or a *linear feedback shift register* (LFSR) [Str02]. ORAs consist of comparators with a latch to retain any mismatch observed by the comparison.

The problem with BIST in regards to PLB and interconnect testing is that the FPGA needs to be reconfigured many times, each time testing a different mode of operation. For example, to completely test programmable logic in Xilinx 4000 and Spartan

series FPGAs, 24 BIST configurations are needed, while to completely test interconnects it takes 206 configurations [SLS03]. Therefore, with BIST for FPGAs, testing all modes of operation of the FPGA requires a large number of time consuming reconfigurations and thus poses a problem for high performance systems which cannot afford to spend system down time in lengthy BIST configuration.

1.7 Thesis Statement

Since some FPGA architectures are capable of partial reconfiguration to speed-up the reconfiguration process, the work presented in this thesis focuses on optimizing the BIST method for FPGA testing using partial reconfiguration. Chapter 2 presents more details about the full reconfiguration and partial reconfiguration facility offered by the new Xilinx FPGA families like Spartan-II, Spartan-III, Virtex and Virtex-II. In Chapter 2 reviews CAD tools used for partial reconfiguration. It presents details about the boundary scan interface, used to partially reconfigure these FPGAs. It provides details about the current state-of-art BIST method for testing PLBs as well as interconnect resources in FPGAs. Chapter 3 describes how the partial reconfiguration can be used for logic BIST of Xilinx Spartan-II, Spartan-III, Virtex and Virtex-II FPGAs and presents results from actual BIST of PLBs in Spartan-II and Virtex devices to illustrate the improvements obtained with partial reconfiguration. Chapter 4 explores a technique to generate interconnect BIST configurations using Java application programmer's interface library, JBits. The chapter also describes the experiments performed to generate partial interconnect BIST configurations and the effects of partial reconfiguration on the size of the routing BIST configurations. Chapter 4 also includes estimates of the number of BIST configurations required to test the routing resources in Virtex I and Spartan

II FPGAs and considers partial reconfiguration for routing BIST. Finally, Chapter 5 presents the summary and conclusions as well as suggestions for the future research and development. While this thesis will focus on Xilinx FPGAs, it is important to emphasize that these techniques can be applied to any FPGA that supports partial reconfiguration.

Chapter 2

Review of Partial Reconfiguration and BIST

This chapter begins with a review of the operation and architecture of Xilinx FPGAs. The configuration memory architectures of Virtex-I and Spartan-II families are then reviewed. These two FPGAs have nearly identical architectures and will be the target of this research. The differences between partial reconfiguration and full reconfiguration are discussed along with the CAD tools used to generate partial reconfiguration viz. BitGen and JBits. Finally, a description of how BIST methods are applied to test programmable logic and interconnect of FPGAs is given. In this section, we review the prior work done using JBits to automatically generate interconnect BIST configurations.

2.1 Architecture of Virtex-I and Spartan-II FPGAs

The PLB and interconnect architecture of Xilinx Virtex-I and Spartan-II is similar. Therefore, unless specified, whenever reference is made towards Virtex-I architecture it is assumed that it applies to Spartan-II.

2.1.1 PLB Architecture

The unit logic cell in the PLB, consists of a 4-input LUT, a flip-flop, and additional dedicated logic. A *slice* consists of two of these unit logic cells. There are two identical slices in a PLB [Xil01b]. The internals of a single Virtex-I slice are shown in Figure 2.1.

Look-Up Table

Each logic cell in the PLB features a 4-input LUT. The LUT can be used in the LUT mode, in a RAM mode or in a shift register mode of operation. In the LUT mode of operation, it acts as a 4-input combinational logic function generator. In the RAM mode of operation, Virtex-I and Spartan-II contain support for implementing 16x1 synchronous RAM or combining two LUTs in a slice to implement 32x1 or 16x2 synchronous RAM. The LUT can also implement up to a 16-bit shift register in the shift register mode of operation [Xil01b].

Flip-Flops

The flip-flops can be configured as edge-triggered flip-flops or level-sensitive latches, can be set or reset synchronously or asynchronously, and contain a clock enable signal. The signals used to set or reset the flip-flops are shared by both the flip-flops within the slice. A global reset signal initializes the storage elements [Xil01b]. The value with which the flip-flops are initialized is specified by a specific bit in the configuration bitstream. The input signals can be applied at the input of the flip-flop through the LUT or directly, bypassing the LUT.

Additional Logic

The dedicated carry logic is used to implement the carry chains found in wide adders and counters. In order to realize the wide adders and counters, the carry logic takes carry input from the previous stages [Xil01b]. A dedicated multiplexer CY, illustrated in Figure 2.1, is utilized to implement wide arithmetic logic functions [Xil03c].

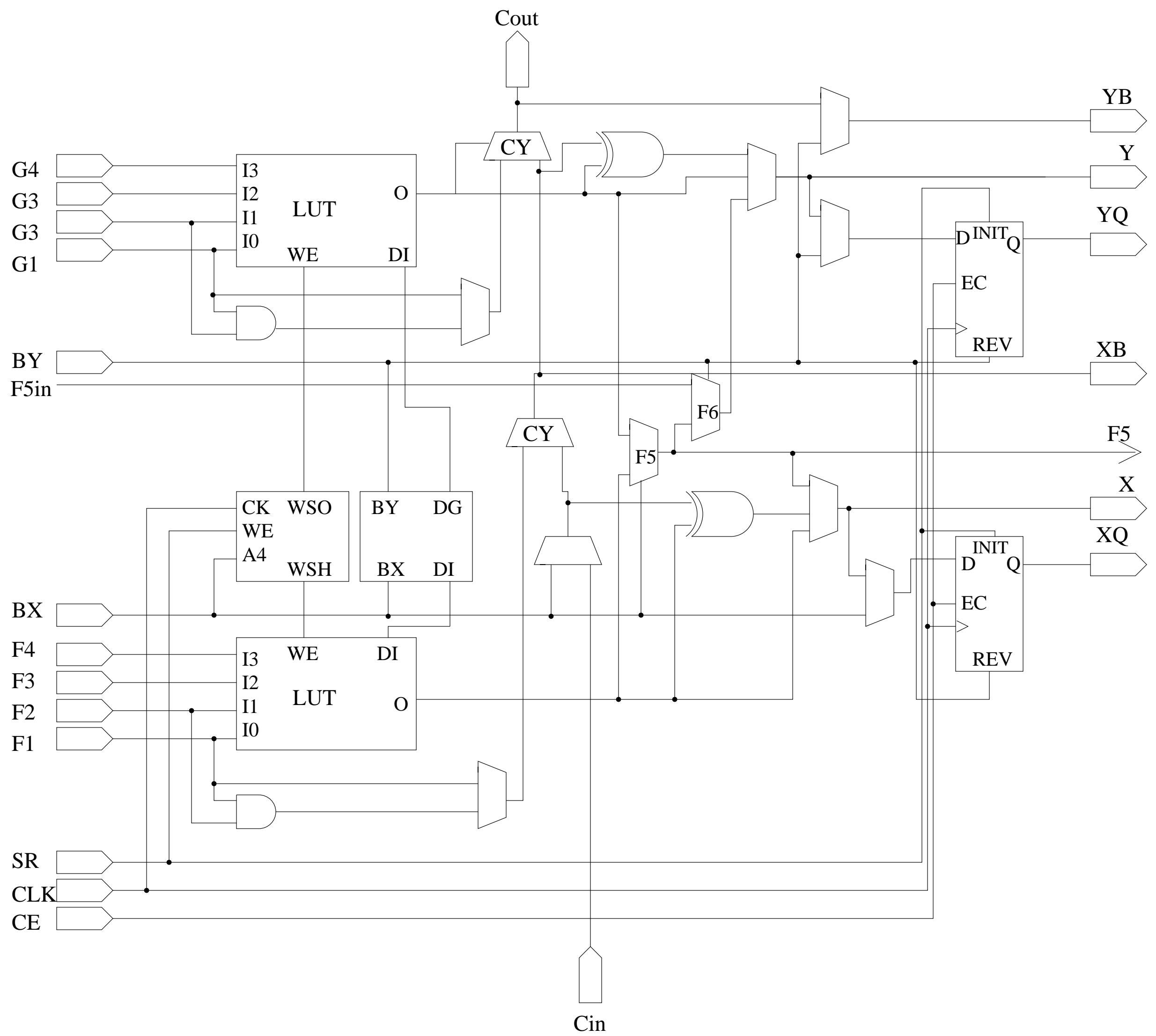


Figure 2.1: Internal Architecture of Virtex-I Slice [Xil01b]

Using multiplexer F5 in Figure 2.1, either of the outputs of the LUTs can be selected. Thus implementing a 5-input function generator, a 4:1 multiplexer, or selected functions of up to nine inputs [Xil01b]. The F6 multiplexer, on the other hand, facilitates implementation of any 6-input function, an 8:1 multiplexer, or selected functions of up to 19 inputs [Xil01b].

2.1.2 Interconnect Architecture

CIPs are the programmable switches used to make connections in the global and local routing resources and the PLBs. There are three basic types of CIPs that can be found in the FPGA interconnect network: cross-point CIP, break-point CIP and multiplexer CIP, as illustrated in Figure 2.2. The cross-point CIP connects or disconnects the connection between a wire segment in the horizontal plane and a wire segment in the vertical plane, depending on the value loaded in the configuration memory bit. The state of the memory bit controlling the break-point CIP determines if the two segments in the same plane would be connected [SWHA98]. The Xilinx FPGAs feature a switch box CIP or *global routing matrix* as referred in the literature from Xilinx [Xil01b]. The switch box CIP comprises an array of break-point CIPs that can be programmed to provide a variety of connections in horizontal and vertical routing resources as well as the PLB inputs and outputs as shown in Figure 2.2(d) [Xil01b]. The multiplexer or MUX CIP controls connection to the common interconnect from one of the k possible connections. There are k configuration memory bits associated with a MUX CIP. The complete set of switch box CIPs has 24 Single wires emerging from its four sides that allow connection between the four neighboring switch box CIPs. The Single lines or x1 lines are part of the local interconnects and provide signal connectivity between the

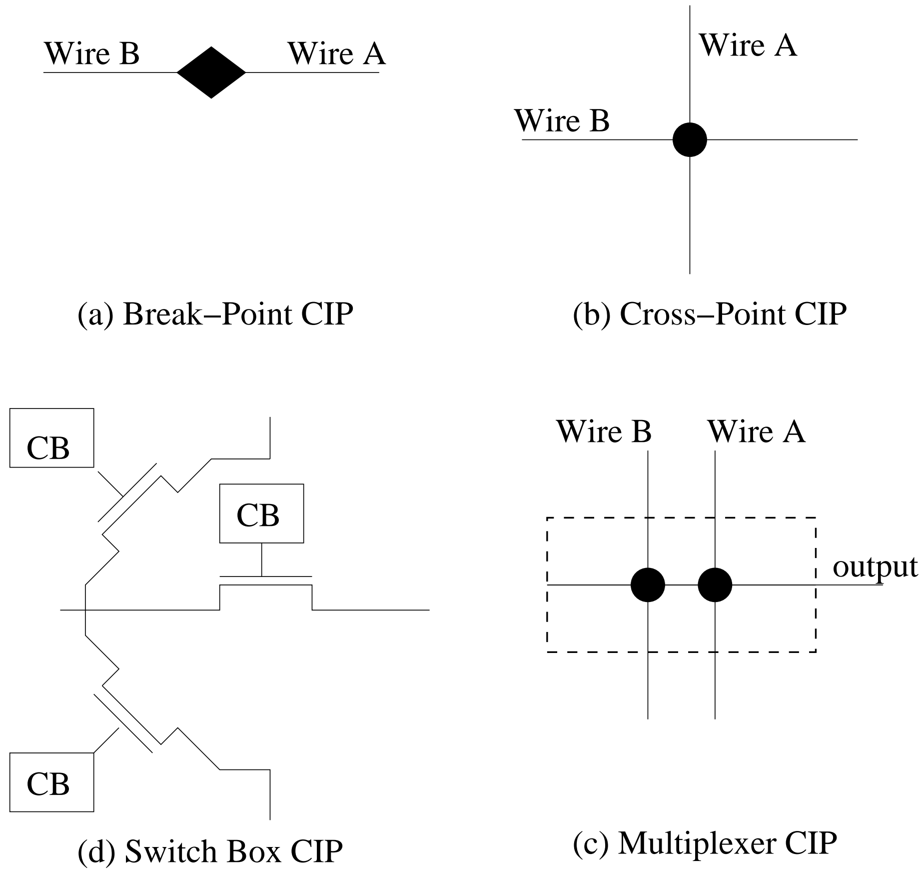


Figure 2.2: Different Types of CIPs Found in FPGA [SWHA98] [FH03]

adjacent PLBs. A total of 12 buffered Hex wires at each of the four sides drive the signal to switch box CIP that is six PLBs away. The Hex lines or the x6 lines span between the PLBs separated by five PLBs and are part of global interconnect resources. A total of 12 Long wire segments provide connectivity across the horizontal width and vertical length of the chip [Xil01b]. The Long wires often carry signals to multiple PLBs and span all the PLBs in horizontal or vertical direction. The switch box featured in Virtex-I and Spartan-II architecture along with the Hex and Single interconnects, is shown in the Figure 2.3.

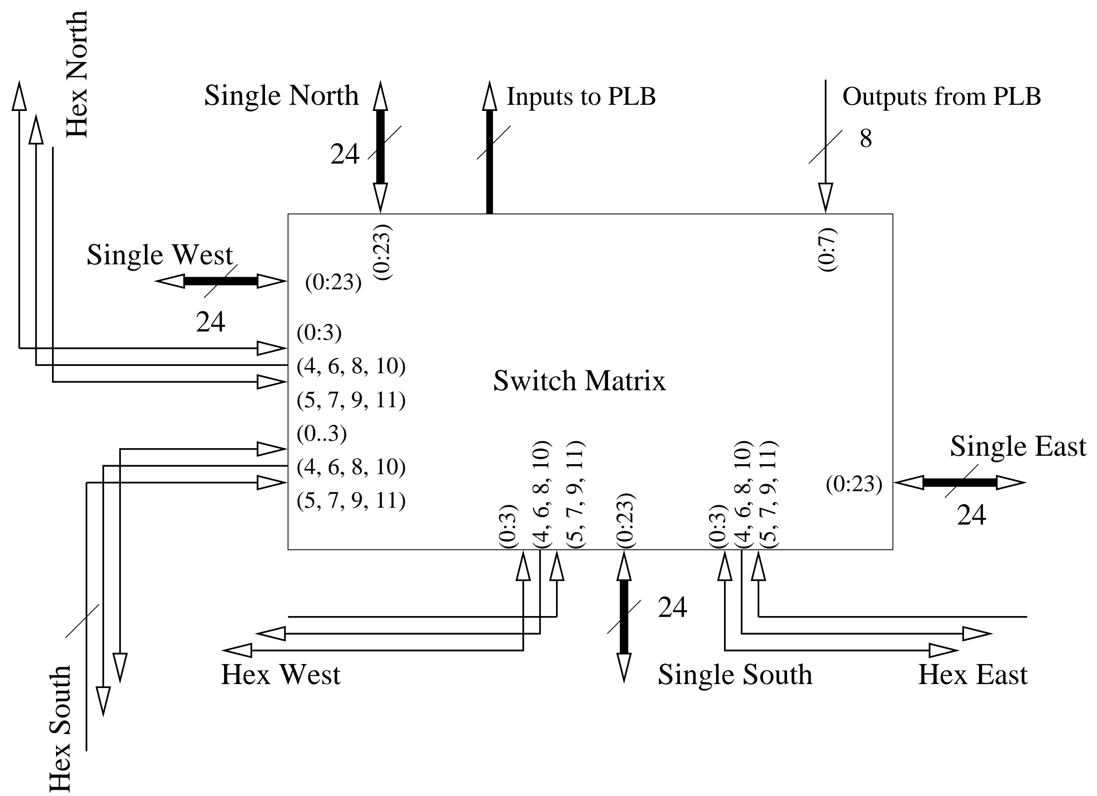


Figure 2.3: Switch Box CIP and Xilinx Interconnect Architecture

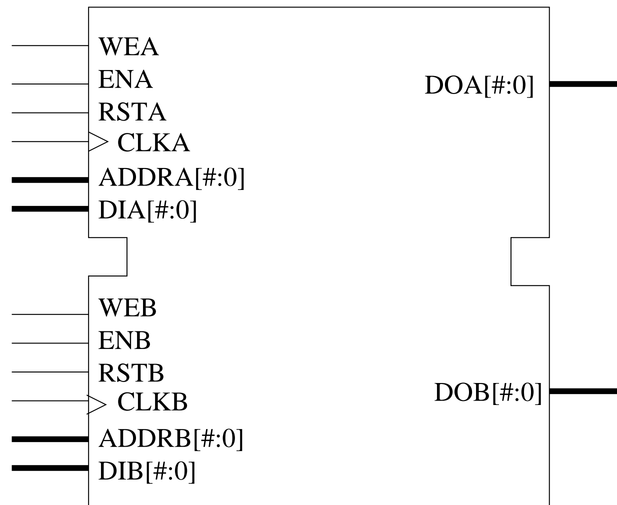


Figure 2.4: Block RAM in Virtex-I and Spartan-II FPGAs

2.1.3 Block RAMs

Large memory blocks are provided in the architecture and are referred to as block RAMs. These block RAMs are located along the two outside columns of the PLB array and each memory block RAM occupies the same height as that of 4 PLBs. Thus a PLB array 64 PLBs high will have 16 memory block RAMs in each outer column and thus 32 total block RAMs. The block RAM is, as illustrated in Figure 2.4, a synchronous, dual port memory and has a total capacity of 4096 bits. The width of data and address bus is configurable and can be set as per the design requirement.

2.2 Configuration of the FPGA

Typically the bitstream can be downloaded bit serially or byte-wide, i.e. one bit or one byte of configuration data is written into the configuration memory each clock cycle. How the bitstream is loaded into the FPGA depends upon the *configuration mode*. The configuration mode can be selected by setting particular logic levels at the mode-select

pins of the FPGA. Different configuration modes offer different capabilities e.g. speed of configuration, partially reconfiguration etc. Different sequences of events may also take place in different configuration modes. Thus selecting the mode of configuration is an important design decision. Virtex-I and Spartan-II FPGAs support eight different modes of configuration. The partial reconfiguration support is available in *selectMAP* and *boundary scan* modes of configuration [Xil02c].

2.2.1 SelectMAP Mode

In the selectMAP mode, one byte of the bitstream is written every clock cycle into the configuration data bus interface (pins D[0:7]). To load a given configuration, selectMAP takes the least time among all the configuration modes available [Xil01b]. First the configuration memory is cleared. The configuration control circuitry senses the mode pins and the mode of configuration is determined to be selectMAP. The bitstream is them loaded byte-by-byte on every rising edge of the configuration clock. To ensure the veracity of the bitstream, a *cyclic redundancy check* (CRC) check is performed at the end. If the CRC checksum loaded is different from the internally calculated CRC, the configuration sequence is aborted. Otherwise the normal Startup-Sequence is commenced as will be discussed in subsection 2.2.3 [Xil02d].

2.2.2 Boundary Scan Mode

In the boundary scan mode, one bit of the bitstream is written into the *test access port* (TAP) of the FPGA each clock cycle. The IEEE 1149.1 test access port and boundary scan architecture is an IEEE standard for in-system testing [IEE90]. The boundary scan has a four wire interface as shown in Table 2.1. All FPGAs from Xilinx support

boundary scan mode of configuration and contain all the mandatory elements in the IEEE 1149.1 standard: the TAP controller, the instruction register, the instruction decoder, the boundary scan register, and the bypass register [Xil01b] [Xil02e] [Xil03b] [Xil03a].

Table 2.1: Virtex TAP Controller Pins

TDI	Test Data In
TDO	Test Data Out
TMS	Test Mode Select
TCK	Test Clock

The TAP controller is a 16-state finite state machine. The logic value of the TMS pin at the rising edge of TCK determines the next state of the TAP controller. The data can be shifted into the data registers by selecting the data register scan sequence or the instruction register by selecting the instruction scan sequence.

The Virtex-I and Spartan-II devices implement all the mandatory commands as well as additional commands to the IEEE 1149.1 standard. These additional commands allow read and write access to the configuration memory. The boundary scan interfaces provide two user defined serial interfaces to the core of the FPGA. In order to use them, the interfaces must be incorporated in the design. The user defined serial interfaces are active after the configuration is completed and may be accessed using special instructions to the TAP controller [Xil02b] [Xil02a] [Xil].

The configuration control circuitry senses the mode pins and mode of configuration is determined to be boundary scan. The CFG_IN instruction is loaded into the instruction register to allow write access to the configuration memory. The bitstream is then loaded bit serially using the boundary scan interface. If the CRC is determined to be correct then the JSTART instruction is loaded in the instruction register which will initiate the Start-up Sequence [Xil02b] [Xil02a].

2.2.3 Start-up Sequence

After the bitstream is completely and successfully written into the configuration memory, the *Start-up Sequencer* state machine in the FPGA initiates the *Start-up Sequence*. Start-up is the transition from the configuration mode to normal operational mode of the FPGA [Xil02d]. The Start-up Sequence includes activation of global reset for initialization of the device. Xilinx provides a CAD tool, called BitGen, to control the Start-up Sequence according to the options set by the user. The subsection 2.5.3 gives an overview of this tool.

2.3 Configuration Memory Architecture of Virtex-I and Spartan-II FPGAs

The configuration memory of Xilinx Virtex-I FPGAs is divided into sections called *frames*. A frame contains configuration data for each section of the device, extending vertically from top to the bottom of the device. Multiple configuration frames clubbed together form a *column* [Xil02d]. The columns can belong to one of the following types:

Center: The center column contains the configuration for the four global clock pins and routing in the center of the device.

Configurable Logic Blocks: The PLBs are sometimes also referred to as *configurable logic blocks* (CLBs). This type of column contains the configuration for all the PLBs and routing in that column, along with two *I/O blocks* (IOBs) at the top and bottom of the column.

IOB: The IOB columns contain the configuration for all the IOBs on the left and right edges of the device.

Block RAM Interconnect: These columns contain the configuration for all interconnect of the block RAMs of the device.

Block RAM Content: These columns contain the initial data contents with which the block RAMs will be pre-loaded during configuration [Xil03d].

A frame is the smallest unit of reconfiguration. The least data that needs to be written into the configuration memory, in order to configure a portion of FPGA, is one frame. The length of the frame increases with the dimensions of PLB array to account for the increase in programmable logic and routing resources in the array. The length of the frame is written into a dedicated internal register in the full configuration process. As the FPGA is fully configured at least once before the partial reconfiguration, it is not necessary to write frame length for the partial reconfiguration [Xil02d].

2.3.1 Addressing

The configuration memory address space is divided into RAM blocks and PLB blocks. The RAM block contains the block RAM content columns. The PLB blocks include the Center, PLB, IOB and block RAM interconnect columns. These blocks are then further divided into major and minor addresses where each configuration column has a unique major address and each frame has a unique minor address within its column [Xil03d]. For the Virtex-I family, the following addressing scheme is in place for the configuration memory as shown in Figure 2.5 (which also includes the number of frames in each column):

- the address '0' is assigned to the center column,
- the even major addresses of PLB column are on the left side of the device,

Figure 2.5: Xilinx Virtex-I and Spartan-II Addressing Scheme [Xil03d]

- the higher even major addresses are assigned to left IOB columns,
- the left block RAM interconnect columns mark the end of even major addresses,
- the address '1' is assigned to the PLB column at the right side of the center column
- the odd major addresses of PLB column are on the right side of the device,
- the higher odd major addresses are assigned to right IOB columns, and
- the right block RAM interconnect columns mark the end of the PLB address block.

The major and minor addresses for any slice or LUT bit in any row/column can be easily calculated, by inserting the values of the constants in the Table 2.2 and Table 2.3, in the formulae given in Table 2.4 and Table 2.5.

Table 2.2: Constants Used in the Address Calculation [Xil03d]

Term	Definition
Chip_Cols	Number of PLB columns on the Virtex device.
Chip_Rows	Number of PLB rows on the Virtex-I device.
Chip_Rams	Number of block RAM columns on the Virtex-I device RAM Space Spacing of block RAM columns (in terms of PLB columns).
FL	Number of 32-bit words in the frame.
RW	1 for Read, 0 for Write.
CLB_Col	Column number of the desired PLB.
CLB_Row	Row number of the desired PLB.
Slice	0 or 1.
FG	0 for the F-LUT, 1 for the G-LUT.
lut_bit	The desired bit from the given LUT. Bits in the LUT are indexed from 0 to 15.
XY	0 for the X Flip-Flop, 1 for the Y Flip-Flop.
RAM_Col	Column number of the desired block RAM.
RAM_Row	Row number of the desired block RAM.
ram_bit	The desired bit from the given block RAM. Bits are indexed from 0 to 4095.

Table 2.3: Variables Used for Address Calculation [Xil03d]

MJA	Frame Major Address.
MNA	Frame Minor Address.
fm_st_wd	The index of the word within a full configuration segment that corresponds to the starting word of the desired frame. A full configuration segment is defined as the following: 1) for PLB/IOB, all PLB, IOB, and RAM interconnect frames beginning at MJA=0, MNA=0 and 2) for block RAM, all RAM content frames for the given RAM column. Words are numbered starting at 0.
fm_wd	The index of the 32-bit word within a frame that contains the desired bit. Words in a frame are numbered starting at 0.
fm_wd_bit_idx	The bit index of the desired bit within frame word fm_wd. Words are indexed in big-endian style, with bit 31 on the left and bit 0 on the right.
fm_bit_idx	Bit index within a frame of the desired bit. Numbered starting with 0 as the left-most (first) bit. Bit numbering within a frame continues across all the words in the frame.

Table 2.4: Calculating the Location of the LUT RAM Bit in Virtex-I Bitstream [Xil03d]

MJA	if (CLB_Col $\leq$ Chip_Cols $\div$ 2), then Chip_Cols - CLB_Col $\times$ 2 + 2 else 2 $\times$ CLB_Col - Chip_Cols - 1
MNA	lut_bit + 32 - Slice $\times$ (2 $\times$ lut_bit + 17)
fm_bit_idx	3 + 18 $\times$ CLB_Row - FG + RW $\times$ 32
fm_st_wd	FL $\times$ (8 + (MJA - 1) $\times$ 48 + MNA) + RW $\times$ FL
fm_wd	floor(fm_bit_idx $\div$ 32)
fm_wd_bit_idx	31 + 32 $\times$ fm_wd - fm_bit_idx

Table 2.5: Equations for Calculating PLB FF Location in the Bitstream [Xil03d]

MJA	if (CLB_Col $\leq$ Chip_Cols $\div$2) then Chip_Cols - CLB_Col$\times$ 2 + 2 else 2 $\times$ CLB_Col - Chip_Cols - 1
MNA	Slice $\times$ (12 $\times$ XY - 43) - 6 $\times$ XY + 45
fm_bit_idx	(18 $\times$ CLB_Row) + 1 + (32 $\times$ RW)
fm_st_wd	FL $\times$ (8 + (MJA - 1)$\times$ 48 + MNA) + RW $\times$ FL
fm_wd	floor(fm_bit_idx $\div$ 32)
fm_wd_bit_idx	31 + 32 $\times$ fm_wd - fm_bit_idx

2.3.2 Frame Organization

The frame can be viewed as being vertically superimposed on the device, with the beginning of the frame at the top of the device. As shown in Table 2.6, the first 18 bits control the two IOBs at the top of the column. The subsequent groups of 18 bits are allocated for each PLB row. Finally the last 18 bits control the two IOBs at the bottom of the PLB column. The frame data is then padded with '0's to make it an integral multiple of 32-bit words [Xil03d].

Table 2.6: PLB Column Frame Organization

Top 2 IOBs	PLB R1	PLB R2	...	PLB Rn	Bottom 2 IOBs
18	18	18	...	18	18

2.3.3 Configuration Registers

The Virtex-I FPGAs provide *configuration registers* to control the configuration process. The configuration architecture defines eleven 32-bit configuration registers, summarized in Table 2.7. To configure the FPGA, commands are written into these configuration registers followed by the data frames containing the configuration data, which are then loaded into the configuration memory of the FPGA [Xil02d].

The configuration register where the commands are to be written is selected by a 32-bit word called *command header format* (Table 2.8) or *Type-I header*. The field *word count* in the command header format, gives the number of words to be written in the subsequent write sequence. With the command header alone, 2048 32-bit words can be written. The configuration architecture also defines the *large block count header extension format* also known as *Type-II header format*, that supports larger write sequences [Xil02d].

Command Register (CMD)

The state of the configuration state machine, the Frame Data Register (FDR), and some of the global signals are determined by the command loaded in the command register. The commands are executed each time a new value is loaded into the Frame Address Register (FAR) [Xil03d]. The commands and their functions are summarized in Table 2.9.

Configuration Option Register (COR)

The bits in the Configuration Option Register (COR) determine the behavior of specific signals used during configuration and the Start-up Sequence. The fifteenth bit

Table 2.7: Configuration Registers [Xil03d]

Register Name	R/W	Function
Command (CMD)	R/W	Controls the operation of the configuration state machine.
Configuration Option (COR)	R/W	Sets various options for events that take place in the configuration and behavior of the device after configuration.
Control (CTL)	R/W	Sets the preferences for the behavior of the device after the configuration.
Cyclic Redundancy Check (CRC)	R/W	Used while configuring the device to load CRC checksum that is verified against the internally counted one.
Frame Address (FAR)	R/W	Used to load the frame address of the ensuing frame data. For PLB data, this is automatically incremented after a complete frame is loaded. For block RAM data the frame address has to be incremented manually.
Frame Data Input (FDRI)	W	Writing the configuration data into the configuration memory.
Frame Data Output (FDRO)	R	Reading the configuration data and states of registers, flip-flops and LUTs from the configuration memory.
Frame Length (FLR)	R/W	Determines the size of the frame in 32-bit words.
Legacy Output (LOUT)	W	For daisy chaining the bitstream of legacy devices.
Mask (MASK)	R/W	Mask register for writes to CTL register.
Status (STAT)	R	Loaded with current values of various control and status signals.

Table 2.8: Command Header Format [Xil02d]

Type	Write/Read	Destination Register Address	Byte Address	Word Count 32-bit Words
31:29	28:27	26:13	12:11	10:0
001	10/01	xxxxxxxxxxxxxx	xx	xxxxxxxxxxx

Table 2.9: Configuration Commands and their Usage [Xil03d] [Xil04]

Command	Code	Description
WCFG	1	Write Configuration Data: Used prior to writing configuration data to the FDRI. It takes the internal configuration state machine through a sequence of states that control the shifting of the FDR and the writing of the configuration memory.
LFRM	3	Last Frame: This command is loaded prior to writing the last (pad) data frame if the GHIGH_B signal was asserted. This command is not necessary if the GHIGH_B signal was not asserted. This allows overlap of the last frame write with the release of the GHIGH_B signal.
RCFG	4	Read Configuration Data: Used prior to reading frame data from the Frame Data Output (FDRO). Similar to the WCFG command in its effect on the Frame Data Register (FDR).
START	5	Begin Start-up Sequence: Starts the Start-up Sequence. This command is also used to start a shutdown sequence prior to partial reconfiguration. The Start-up Sequence begins with the next successful CRC check.
RCAP	6	Reset Capture: Used when performing capture in single-shot mode. This command must be used to reset the capture signal if single-shot capture has been selected.
RCRC	7	Reset CRC: Used to reset CRC register.
AGHIGH	8	Assert GHIGH_B Signal: Used prior to reconfiguration to prevent contention while writing new configuration data. All PLB outputs and signals are forced to a one.
SWITCH	9	Switch CCLK Frequency: Used to change the frequency of the Master CCLK.

of this register is reset by the "SHUTDOWN" command, used for shutting down the FPGA for partial reconfiguration. The "START" command sets this bit to value '1', initiating the Start-up Sequence [Xil03d]. The BitGen tool, which is used to generate the configuration file for the device, provides options to set/reset bits in this register.

Cyclic Redundancy Check (CRC)

The Cyclic Redundancy Check (CRC) register provides a means of checking for transmission errors in the bitstream. A 16-bit CRC checksum is calculated every time data is written into specific registers using the following polynomial:

CRC-16 $= X^{16} + X^{15} + X^2 + 1$ [Xil02d]

The CRC register is used to store the checksum. In complete reconfiguration, CRC check is performed twice by loading a pre-calculated CRC block-check value. The second CRC checksum is calculated with the data of the last frame. A non-zero resulting value indicates error in transmission, therefore, configuration is aborted.

Frame Address Register (FAR)

The Frame Address Register contains the address of the frame being loaded. The address is partitioned into block type (PLB or RAM block), major address, and minor address. The minor address is auto-incremented each time a complete data frame is loaded and major address is auto-incremented if the last frame for the PLB column is completely loaded. For RAM blocks the major address needs to be loaded separately [Xil03d].

Frame Data Input Register (FDRI)

The Frame Data Input Register is used to specify the size of the configuration data in words that would be written to the configuration memory. Type-I or Type-II headers are used depending on how large the data is. The FDRI is used to hold this header information [Xil03d].

Frame Length Register (FLR)

The length of the frame without the pad word is set in terms of 32-bit words in this register. As the devices grow in the array size, the frame length increases to incorporate the configuration data of increased routing and logic resources [Xil03d].

2.3.4 Full Reconfiguration Bitstream

The commands in the bitstream for full reconfiguration of a Virtex-I device can be divided into 3 command sets. The first command set initializes the internal configuration logic for loading the data frames. A default value is assigned to the CRC register. The frame length is set in Frame Length Register (FLR). The Configuration Option Register (COR) is loaded with the value that would specify the desired behavior of the device after the configuration. The SWITCH command is loaded into the CMD register to change the configuration clock frequency to the clock frequency specified in the COR.

The second command set writes the configuration data frames. The command, WCFG (Write Configuration), is loaded into the CMD register. This, among other things, activates the circuitry that writes the data loaded into the FDRI into the configuration memory cells. The data word count is specified in command word of Type 1 or if the data word count is too large, the command word of Type 2 follows command word of Type 1. Typically three large frame sets are loaded containing the PLB configuration, the block RAM configuration and the last frame data. At the end of the third frame set, the CRC checksum is loaded into the CRC register. The Last Frame command (LFRM) is loaded into the CMD register indicating to the configuration circuitry that the following frame set will be the last frame.

The third command set triggers the Start-up Sequence with the START command and completes the CRC checking and activates the FPGA.

2.4 Readback

Readback is the process of reading data from the configuration memory [Xil02d]. Readback can be utilized to compare the stored configuration against the actual bitstream, as well as to read the current state of all internal PLB and IOB registers, LUTs operating in RAM mode and block RAM values. The former is known as *readback verification* and latter is referred to as *readback capture*. Both verification and capture can be done in one readback sequence.

2.4.1 Readback Verification

Readback verification can be obtained without any changes in the configuration memory, through selectMAP and boundary scan mode [Xil02d]. This readback data can then be verified against a *bitmap file* generated by the BitGen tool when run with "readback" option enabled for each design.

2.4.2 Readback Capture

In order to examine the state of the internal logic resources, the readback capture capability must be enabled. An additional readback capture option allows a single capture or multiple captures after the device is configured. When asserted, the register states are captured in unused space in the configuration memory on the rising edge of the clock signal [Xil] [Xil02d].

Table 2.10: Readback Commands Required to Perform Readback on PLB Configuration

Synchronization word	AA99 5566h
Packet Header: Write to FAR Register	3000 2001h
Packet Data: Starting frame Address	0000 0000h
Packet Header: Write to CMD Register	3000 8001h
Packet Data: RCFG	0000 0004h
Packet Header: Read from FDRO	2800 6000h
Packet Header Type 2: Data Words	48– —-h

The logic allocation file provided by BitGen indicates the absolute position of the flip-flop bits in the complete readback file. This information will prove to be important while retrieving BIST results, as will be discussed in Chapter 3.

2.4.3 Readback Operations

Readback is performed by reading a data packet from the Frame Data Output Register (FDRO) register. There are three types of data packets to be read (one for PLB configuration with capture data and two for block RAMs). The commands needed to be given in order to accomplish this are summarized in Table 2.10.

The complete configuration memory readback is initiated by writing the starting frame address of (0000 0000)h in the FAR register. The number of words to be read for full readback capture is function of the size of the device, e.g. for an XCV100 the number of 32-bit readback words would be 22,554 [Xil02d].

The bits in the readback bitstream indicate three types of information 1) configuration data, 2) captured data and 3) pad bytes. The pad bytes align the frame data to a 32-bit word boundary. It can be noted that readback bitstream does not contain any CRC check information.

2.5 Partial Reconfiguration

The sequence of events taking place while partially reconfiguring the device is considerably different than that of full reconfiguration. For partial reconfiguration, it is required that device be fully configured once and that the configuration interface be active after the full configuration is complete. That is to say, the I/O pins used for selectMAP mode of reconfiguration would retain their configuration function [Xil02d]. The boundary scan mode is a permanent interface and is always present [Xil03d].

The bitstream performing partial reconfiguration of any logic resource of the FPGA, henceforth simply referred to as a *partial bitstream*, contains the major and minor addresses of the frame containing the configuration data for that resource. The major and minor addresses of the frame are calculated using formulae given in Table 2.4 and Table 2.5 [Xil03d]. The partial bitstream contains instructions to write the address to the FAR register. The ensuing instructions load the FDRI register with the number of words to be written into the configuration memory. After these instructions, the frame data follows. There are two ways in which the FPGA can be partially reconfigured: with or without shutdown [Xil02c].

2.5.1 Partial Reconfiguration without Shutdown Sequence

If the device is not shut down, the functions implemented in the parts of the FPGA not affected by partial reconfiguration may continue to work without interruption. The logic changes in the PLB or routing take place once the corresponding frame gets completely written into the device. This mode would be used for operations such as online testing and fault-tolerance [AES01].

2.5.2 Partial Reconfiguration with Shutdown Sequence

If the device is shut down, then parts of the FPGA not affected by partial reconfiguration would stop executing the configured function. At the start of the Shutdown Sequence, the dummy word (FFFF FFFF)h and the synchronization word (AA99 5566)h are written. The dummy word provides the clock cycles necessary to initialize the configuration logic. The synchronization word is used to align the bitstream on the 32-bit-word boundary [Xil04]. The Shutdown bit is set in the COR register. The START command is then loaded into the CMD register to start the Shutdown Sequence. The CRC value is reset. As the Shutdown Sequence requires all the other logic in the device to be disabled, the clock to all sequential logic is disabled. The AGHIGH command is then loaded into the CMD register to prevent contention on the internal signals while writing the new data. As the GHIGH_B signal is asserted due to the AGHIGH command, the LFRM command is written into the CMD register. This allows writing the Last Frame packet as GHIGH_B signal is released [Xil03d].

2.5.3 BitGen

BitGen is a command line tool that converts the netlist file in Xilinx native format (.ncd file) into a configuration bitstream file. The FPGA can then be configured with this bitstream. This tool gives a number of options to control the tasks, in and after the Start-up Sequence, for the design implemented. These tasks include: the timing of the start-up signals, clock rate to be used for the configuration, signal assignment of some of the I/O pins used during configuration once the configuration is over, etc. The BitGen options are set as per the design being implemented. BitGen also gives options for generating a partial reconfiguration bitstream that contains only the difference between

the .ncd file, and the old bitstream. The use of BitGen for partial reconfiguration during BIST will be discussed in more detail in Chapter 3.

2.5.4 JBits

JBits is a set of Java classes which provide an Application Program Interface (API) into the Xilinx XC4000, Virtex-I, Virtex-II series FPGA bitstreams [GLS99]. The JBits API facilitates writing applications that would modify the bitstream on-the-fly, configure and readback from the FPGA configuration memories [Xil01a]. The API gives the programmer gate level access to the FPGA and the Java programming language allows a programmer to create many layers of abstraction. JBits can therefore be used to write custom CAD tools featuring dynamic partial reconfiguration or traditional CAD tools to produce place-and-route for the FPGA families supported [Xil01a].

The simplest of the applications that can be built with JBits API would take the bitstream generated by BitGen as input and configure an FPGA board with it. More advanced applications contain circuit designs specified with the JBits API calls and generate the output in Xilinx Design Language (XDL) which defines the netlist in symbolic format along with the bitstream. The design flow with JBits API is illustrated in Figure 2.6.

In order to use JBits API, the programmer writes a Java program that utilizes JBits API calls (henceforth simply referred to as *JBits program*) containing calls to configure the PLB and routing resources of the FPGA. Upon execution, the JBits program generates the desired configuration of logic as well as interconnect resources of the FPGA. Thus, the programmer specifies the design of the circuit using JBits API calls embedded

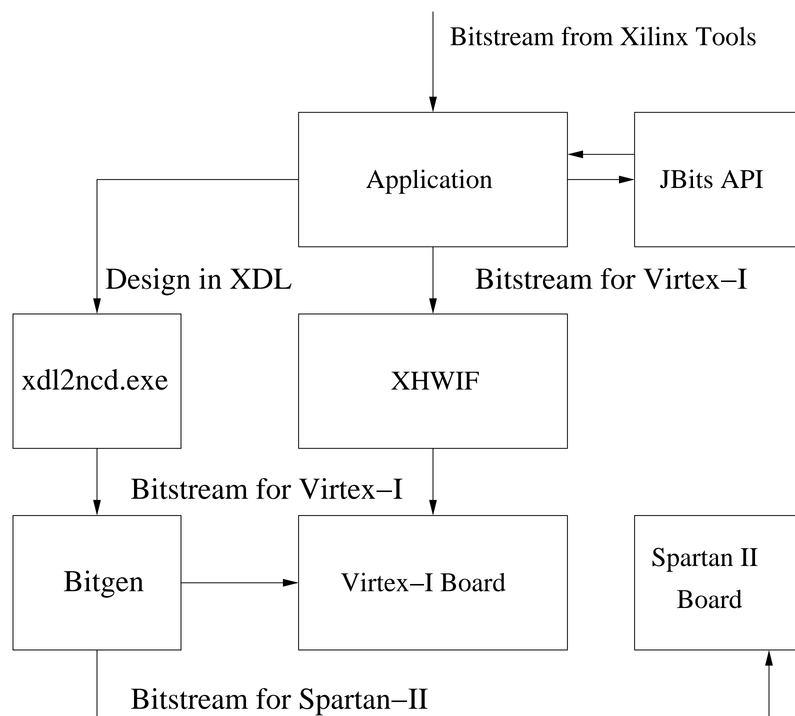


Figure 2.6: Design Flow of the Application with JBits

into a regular Java language program [Xil01a]. The JBits program can then be compiled with the Java compiler (javac). The JBits program runs under the Java Virtual Machine (JVM) environment. The output of the program can be a regular bitstream, or core files (.ctf files), that may run from the hardware simulator described in [Xil01c], or an XDL file containing the design description which can then be viewed and processed using conventional Xilinx tools like FPGA editor or BitGen [Xil01a]. Therefore, the application program does not rely on conventional *place-and-route* (PAR) tools for automatic routing. The JBits API thus combines the flexibility of an HDL as well as the functionality of a PAR tool. JBits API provides a configuration memory readback API which has a facility to read back the state of the logic elements in PLB [Xil01a]. Therefore, tools can be written to interface with the FPGAs. Graphical tools, such as BoardScope, demonstrate the use of this API for tracing the logic values of flip-flops, LUTs and internal signals [LG98].

For the bitstream produced by the JBits program to work correctly, when an FPGA is configured with it, there are a number of structures, internal to JBits API, that need to be initialized in a specific sequence. For the simplest applications that can be developed with JBits API, these internal structures can be initialized with a relative ease as their number is small and the process of initialization is well documented in the documentation included with the JBits software [Xil01a]. However, for the advanced applications a large number of structures internal to JBits need to be initialized. The sequence in which these internal structures need to be initialized is not as well documented in [Xil01a] [Xil01d]. Therefore, the documentation provided with JBits software is insufficient for a test engineer to write advanced applications. In this thesis, we have developed a sequence of

steps that correctly initializes these internal structures. A JBits program that implements this sequence, is observed to generate a bitstream that correctly configures the FPGA to produce the desired results and textual representation of the design in XDL.

Definitions Used

As JBits API is completely written in Java [Xil01d], therefore we find it convenient to follow the terminology of object oriented languages wherever possible.

class "A class is a blueprint or prototype that defines the variables and the methods common to all objects of a certain kind" [Jav04].

object "An object is a software bundle of related variables and methods" [Jav04]. The object is an instantiation of a class. For example, in JBits API, a physical pin of an FPGA is modeled in class "Pin". When a reference to particular pin is to be made, an instance of the class "Pin" is created.

method "A function defined in a class" [Jav04]. The methods are invoked from the main program to change the state or to retrieve the state of the object(s).

package "A package is a collection of related classes and interfaces providing access protection and namespace management" [Jav04]. Packages are the mechanism created so that names of the identifiers declared in one class defined in one package do not collide with another class declared in a separate package and create confusion for the compiler. The fully qualified name of the class is the class name prefixed by the name of the package that it belongs to. In this thesis, whenever referring to a class in JBits API, we will identify the class by its fully qualified name.

Device Model

There are two models provided by JBits API to access and manipulate the FPGA resources. In the first model, each physical FPGA resource (like a logic element in the PLB, IOB and block RAM) is mapped into one of the classes in the package com.xilinx.JBits.Virtex.Bits [Xil01d]. Each class has static 2-dimensional array(s) of integers representing configuration memory bit(s) associated with the resource(s). The applications that need to set and reset configuration memory bits, that modify bitstream on-the-fly may use this model [Xil01a]. This model is not suitable for designing circuits and specifying routing between the PLBs.

There exists another model which is more suitable for the advanced applications that specify the design in the JBits program. This model provides a logical abstraction of the underlying FPGA by modeling every logic element in the PLB, IOB or block RAM as a *runtime parameterized core (RTPCore)* [GL99]. The inputs and outputs of a PLB, IOB or block RAM, Single, Hex and Long wires are modeled as *pins*. The RTPCores contain *ports* and *signals* which are the HDL like features of JBits API. The logical connections between RTPCores are specified through ports and signals.

If the programmer simply specifies the logic elements within the PLB that are used in the design and the logical connections between them, then in this model, the synthesis, placement and routing of these cores is completely automatic and is done by another Java program called *JRoute* [Xil01d]. The automatic synthesis, placement or routing can be a serious impediment when it does not happen the way the designer intended it. JBits API gives the programmer the ability to specify the placement and routing of the RTPCores. The programmer assigns pins to the ports of an RTPCore that needs to be

placed. Note that a port may still have unrouted nets and buses attached to it and the router will continue to place them.

Model of PLBs

As stated above, in the first model, for every element in the PLB like the flip-flop, LUT, multiplexer, and combinational logic element, there exists a Java class. The class has a static 2-dimensional integer array as a data member [Xil01d]. The bits and their positions in the array correspond to the state and location of the configuration memory bit associated with that element. Table 2.11 provides the mapping of classes for the elements contained in a Virtex-I PLB.

There exists another model for the PLB where the flip-flops, LUTs, multiplexers and logic gates are viewed as RTPCores and the inputs and outputs of these elements are viewed as pins. In order to configure the PLB, these RTPCores are instantiated in the program and pins are connected to the ports of the RTPCores. It is sufficient to specify the logical connections between the RTPCores. The router appropriately places the partially routed cores.

Table 2.11: Classes Used for Bit Level Manipulation of PLB Elements [Xil01d]

CIPs related to slice 0	com.xilinx.JBits.Virtex.Bits.S0Control
CIPs related to slice 1	com.xilinx.JBits.Virtex.Bits.S1Control
Logic Value of slice 0 FF	com.xilinx.JBits.Virtex.Bits.CLB
Logic Value of slice 1 FF	com.xilinx.JBits.Virtex.Bits.CLB
Logic Values of slice 0 LUT	com.xilinx.JBits.Virtex.Bits.LUT
Logic Values slice 1 LUT	com.xilinx.JBits.Virtex.Bits.LUT

Table 2.12: Classes Used for Bit Level Manipulation of Switch Box CIPs[Xil01d]

com.xilinx.JBits.Virtex.Bits.BiHexToSingle
com.xilinx.JBits.Virtex.Bits.UniHexToSingle
com.xilinx.JBits.Virtex.Bits.SingleToSingle
com.xilinx.JBits.Virtex.Bits.OutMuxToSingle

Routing Model

There are two models for the routing resources. The configuration memory bits corresponding to the switch box CIPs are modeled by the static 2-dimensional integer arrays split into four classes shown in the Table 2.12 [Xil01d]. The configuration memory bits corresponding to the output MUX CIPs are modeled by the static 2-dimensional integer arrays in eight classes shown in the Table 2.13 [Xil01d]. The configuration memory bits corresponding to the input MUX CIPs are modeled by the static 2-dimensional integer arrays split in twenty eight classes shown in the Table 2.14 [Xil01d]. These classes are useful when explicitly specifying if the CIP is to be turned on or off, or when explicitly specifying the connection between the routing resources.

A separate model exists for the applications where the routing between two PLBs, PLBs and IOBs, and PLBs and block RAMs is specified. In this model, interconnect resources are modeled as static one-dimensional integer arrays as depicted in Table 2.15. These integer arrays do not represent the CIPs but instead they model the actual interconnect resource. Therefore, in order to use the interconnect resources in conjunction with the HDL-like features in the JBits program, an object is created with appropriate static integer array passed as an argument.

Table 2.13: Classes Used for Bit Level Manipulation of Output MUX CIPs[Xil01d]

com.xilinx.JBits.Virtex.Bits.OUT0	1st Output of the Out Mux
com.xilinx.JBits.Virtex.Bits.OUT1	2nd Output of the Out Mux
com.xilinx.JBits.Virtex.Bits.OUT2	3rd Output of the Out Mux
com.xilinx.JBits.Virtex.Bits.OUT3	4th Output of the Out Mux
com.xilinx.JBits.Virtex.Bits.OUT4	5th Output of the Out Mux
com.xilinx.JBits.Virtex.Bits.OUT5	6th Output of the Out Mux
com.xilinx.JBits.Virtex.Bits.OUT6	7th Output of the Out Mux
com.xilinx.JBits.Virtex.Bits.OUT7	8th Output of the Out Mux

Table 2.14: Classes Used for Bit Level Manipulation of input MUX CIPs[Xil01d]

com.xilinx.JBits.Virtex.Bits.S0BX	BX input of the slice 0
com.xilinx.JBits.Virtex.Bits.S0BY	BY input of the slice 0
com.xilinx.JBits.Virtex.Bits.S0CE	CE input of the slice 0
com.xilinx.JBits.Virtex.Bits.S0Clk	Clk input of the slice 0
com.xilinx.JBits.Virtex.Bits.S0SR	SR input of the slice 0
com.xilinx.JBits.Virtex.Bits.S1BX	BX input of the slice 1
com.xilinx.JBits.Virtex.Bits.S1BY	BY input of the slice 1
com.xilinx.JBits.Virtex.Bits.S1CE	CE input of the slice 1
com.xilinx.JBits.Virtex.Bits.S1Clk	Clk input of the slice 1
com.xilinx.JBits.Virtex.Bits.S1SR	SR input of the slice 1
com.xilinx.JBits.Virtex.Bits.TS0	TS0 input of the slice 0
com.xilinx.JBits.Virtex.Bits.TS1	TS1 input of the slice 0
com.xilinx.JBits.Virtex.Bits.S0F1	1th input of F LUT in the slice 0
com.xilinx.JBits.Virtex.Bits.S0F2	2th input of F LUT in the slice 0
com.xilinx.JBits.Virtex.Bits.S0F3	3th input of F LUT in the slice 0
com.xilinx.JBits.Virtex.Bits.S0F4	4th input of F LUT in the slice 0
com.xilinx.JBits.Virtex.Bits.S0G1	1th input of G LUT in the slice 0
com.xilinx.JBits.Virtex.Bits.S0G2	2th input of G LUT in the slice 0
com.xilinx.JBits.Virtex.Bits.S0G3	3th input of G LUT in the slice 0
com.xilinx.JBits.Virtex.Bits.S0G4	4th input of G LUT in the slice 0
com.xilinx.JBits.Virtex.Bits.S1F1	1th input of F LUT in the slice 1
com.xilinx.JBits.Virtex.Bits.S1F2	2th input of F LUT in the slice 1
com.xilinx.JBits.Virtex.Bits.S1F3	3th input of F LUT in the slice 1
com.xilinx.JBits.Virtex.Bits.S1F4	4th input of F LUT in the slice 1
com.xilinx.JBits.Virtex.Bits.S1G1	1th input of G LUT in the slice 1
com.xilinx.JBits.Virtex.Bits.S1G2	2th input of G LUT in the slice 1
com.xilinx.JBits.Virtex.Bits.S1G3	3th input of G LUT in the slice 1
com.xilinx.JBits.Virtex.Bits.S1G4	4th input of G LUT in the slice 1

Table 2.15: Model of Interconnect Resources in the Package com.xilinx.JRoute2.Virtex.ResourceDB [Xil01d]

Interconnect Type	Class
Horizontal Long	Long_Horiz
Vertical Long	Vert_Horiz
Horizontal Hex	Hex_Horiz_East, Hex_Horiz_West.
Vertical Hex	Hex_Vert_North, Hex_Vert_South.
Single	Single_East, Single_West, Single_North, Single_South.

Routing

To route the Virtex-I and Virtex-II family of FPGAs, JBits API provides a Java program known as *JRoute*. The capabilities of JRoute include routing between two PLB pins, a PLB pin and an IOB pin or a PLB pin and a block RAM pin. JRoute also features the capability of connecting a single source pin to multiple sink pins. The programmer has a choice between automatic routing, template based routing and manual routing of the FPGA [Kel00].

Automatic Routing

The programmer defines the source pin and the sink pin and leaves the task of routing to the automatic router. The call is described as follows [Kel00] [Xil01d]:

```
route(source, sink);
```

Template Based Routing

A template contains the direction and the type of a wire. The template does not identify the wire in itself. The static integers in the class com.xilinx.JRoute2.Virtex. ResourceDB.CenterWires denote the direction and type. For example: SINGLE_EAST represents any Single wire in the east direction. Similarly, HEX_SOUTH is any Hex wire in the south direction. Therefore, in turn, it represents general guidelines to the router.

```
route (source, sink, Template t);
```

The source and the sink are objects of class com.xilinx.JBits.CoreTemplate.Pin, and thus model the physical resources of the FPGA. The template is an array of paths existing between the source and the sink. Consider the following example, which instructs the router to take the specified path while routing pins slice 0, X flip-flop output (S0_XQ) in (row,col)=(11,12) to the input of slice 0, F LUT (S0_F1) in (row,col)=(18,5) [Xil01a]:

```
int[] template = { TemplateRouter.OUTMUX,
                   TemplateRouter.HEX_WEST,
                   TemplateRouter.HEX_NORTH,
                TemplateRouter.SINGLE_NORTH,
                TemplateRouter.SINGLE_WEST,
                TemplateRouter.INPUT   };
Pin source = new Pin(Pin.CLB, row, col, CenterWires.S0_XQ);
jroute.route(source, CenterWires.S0_F1, template);
```

The template router has a limitation that it can only be used to route between the two PLBs [Kel00] [Xil01d].

```
JBits jbits = super.getJBits();

ResourceFactory rf = ResourceFactory.getResourceFactory(jbits);

row = 1;
col = 3;


//The sink resource is the output of the second multiplexer
Pin outmux = new Pin(Pin.CLB, row, col, CenterWires.OUT[2]);
Segment seg = rf.getSegment(outmux);

//First step: Mark the resource as saved
seg.save();

Pin src7  = new Pin(Pin.CLB, row, col, CenterWires.S0_X);
Pin sink7 = new Pin(Pin.CLB, row, col, CenterWires.OUT[2]);

//Second step: Connect the input of the slice 0 'X' flip–flop to the second output multiplexer
JBitsConnector.makeConnection(jbits, src7, sink7, ps);
```

Figure 2.7: JBits Program for Manual Routing

Manual Routing

The programmer can specify each interconnect resource to be used for the routing. It is a two step process as shown in the Figure 2.7. First, the required resource is reserved using calls provided in the class com.xilinx.JBits.JRoute2.Virtex.ResourceFactory. This class keeps track of the utilization of the interconnect resources. The resource can be either saved for the immediate use or simply marked as "used" for later use. In the second step, a call to an appropriate method in com.xilinx.JBits.JRoute2.Virtex.JBitsConnector is performed to complete the manual routing [Kel00] [Xil01d]. The fourth argument to the call to *makeConnection* method is given to display the output of the routing to the console.

2.6 BIST for FPGAs

In this section, we explore BIST for FPGAs in more detail. The key points to be considered to understand BIST for FPGAs are: how BIST does in-system testing of the FPGAs, how complete fault coverage is achieved, the diagnostic resolution that the method provides, and the scalability of the approach. Some of the notable attempts to implement BIST for FPGAs are [RZ00] [RPFZ99] [HGWS99] [SWHA98] [SXCT00] [AES01] [AS01] [SKCA97] [SNLA02] [SLS03] [TM03].

BIST for FPGAs is typically divided into *logic BIST* and *interconnect BIST* according to the FPGA resources that it tests. BIST can also be considered according to the state of operation of the FPGA at the time of testing. If part of FPGA is tested without affecting the normal operation of other part of FPGA, it is referred to as *on-line* BIST and, if FPGA device needs to be shut down prior to testing, then it is referred to as *off-line* BIST [AES01] [SNLA02].

2.6.1 Logic BIST

The general idea of off-line logic BIST, as proposed in [SKCA96] [SCKA96], is to configure groups of PLBs as TPGs and ORAs that test the BUTs. As the PLBs can be configured in different modes of operation, the logic BIST must test all of these modes. The process of testing the PLB in a given mode of operation is referred to as a *test phase*. To completely test the PLB in all of its modes of operation requires set of test phases referred to as a *test session*. Each test phase consists of reconfiguring the FPGA with the BIST circuitry, initiating the BIST sequence, and reading the BIST results from the ORAs [HGWS99]. For in-system testing, BIST configurations would

be stored in a system memory and a system controller would be responsible for loading each configuration into the configuration memory of the FPGA, initializing the TPGs, BUTs, and ORAs via a *Global Reset*, and reading the *Pass/Fail* results in the ORAs at the end of BIST sequence. Generating the test patterns and analyzing the output responses are performed concurrently by the BIST circuitry in the device as shown in the Figure 1.5 [AS01]. All the BUTs are configured identically. As the same test stimulus is applied to these functionally and architecturally equivalent BUTs by two or more TPGs, the output responses received from fault-free BUTs should be the same. Therefore a comparator-based ORA compares results from different BUTs and produces a Pass/Fail result by latching any mismatch observed during the BIST sequence.

In the next test session the roles of the PLBs are changed from BUTs to ORA or TPGs and vice versa. This approach requires only two test sessions to completely test the PLB in the FPGA as long as at least half of the PLBs are BUTs in each test session [AS01]. The cases of combinations of faults that cannot be detected by this method have negligible chance of occurrence.

The requirement of two logic BIST test sessions to completely test the logic resources, is independent of the device size, provided that number of PLBs required to implement TPGs ($2^{*}N_{TPG}$) for the logic resources do not exceed the number of PLBs in the array (N). If $N_{TPG} > \frac{N}{2}$, then more than two BIST test sessions are required to test the logic resources of the FPGA [SLS03]. Therefore, there is a potential penalty of additional test sessions for smaller FPGAs. For FPGAs featuring larger PLB arrays, excessive loading of TPGs might pose a problem depending on the architecture. Here the solution is to divide the PLB array into quadrants with each quadrant configured with an independent BIST architecture executing in parallel. There is no delay associated

with this and, therefore, this BIST architecture scales linearly for the large PLB arrays [SLS03] [AS01].

This BIST approach has been implemented on commercially available FPGAs such as XC4000 and Spartan from Xilinx [SLS03], ORCA 2C from Lucent Technologies [AS01] and Altera Flex8000 [SCKA96] for programmable logic resources. In order to extend this approach to any FPGA architecture, 1) the PLB should be capable of implementing the functionality of LFSR-based or counter-based TPG and an ORA/scan cell, 2) the routing architecture should feature global and local routing elements, and 3) the FPGA should be capable of in-system reconfiguration, as the PLBs need to be reconfigured several times for complete testing. As all these capabilities can be either implemented or are available on most of the commercially available FPGAs, this approach is architecture independent.

2.6.2 Interconnect BIST

The interconnect BIST tests a group of wire segments controlled by CIPs, known as *wires under test (WUTs)*. The WUTs are tested for the fault models described below. Counter-based TPGs are suitable for generating the test patterns. All possible 2^n combinations of test vectors are applied to the WUTs, provided n is not large [SWHA98]. The mismatch between a good and faulty WUT is latched by a comparison-based ORA. The general architecture of interconnect BIST is shown in Figure 2.8.

The comparison-based ORA has a limitation that a fault will go undetected if the WUTs being compared have equivalent faults, thereby giving equivalent responses for the test patterns applied. Another problem associated with this arrangement is that the diagnostic resolution obtained may be insufficient for fault-tolerant applications.

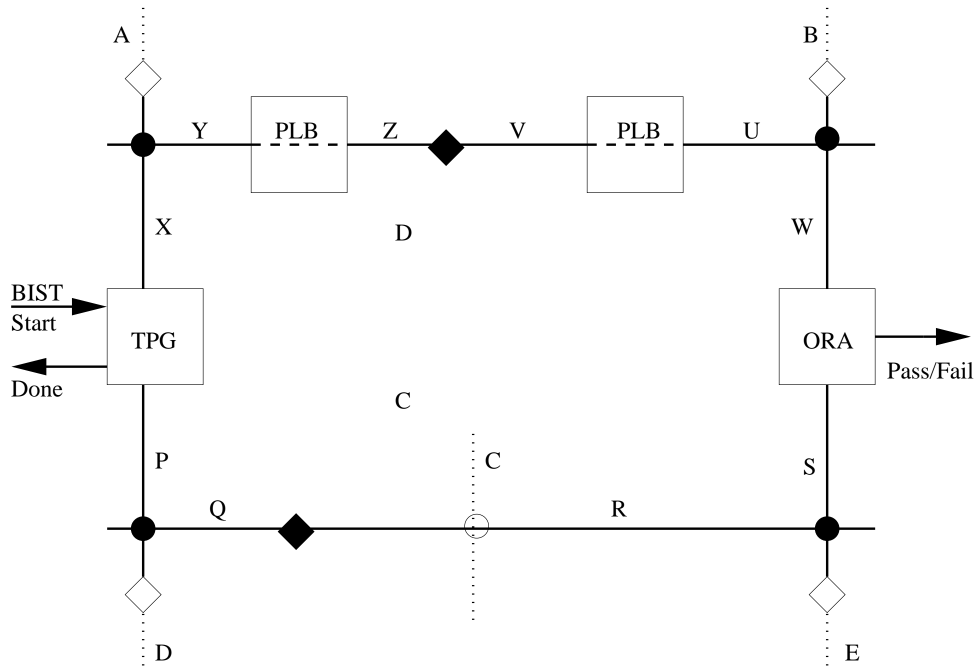


Figure 2.8: BIST for FPGA Interconnect Resources [SWHA98]

[SNLA02] addresses the fault masking problem associated with the comparison-based ORA, through *two-testing analysis*. This ensures that every WUT is tested at least twice with a different group of wires tested by different TPG and ORA. The fault and diagnostics concerns are alleviated by employing a combination of strategies like replacing the comparison-based ORA by a scan based ORA, comparing the fault signature against a fault dictionary, isolating the faulty wire using the progressive deletion, interchanging the roles of TPG and ORA, and divide-and-conquer for locating the faults in the various segments of the wire. The approach described in [SXCT00] implements a parity-based ORA that checks parity generated by TPG along with the response of WUTs.

Fault Models

The following fault models are considered in BIST for interconnect resources:

- Bridging faults between the wires,
- Opens in the wires,
- CIPs stuck-on (stuck-closed),
- CIPs stuck-off (stuck-open),
- Wires stuck-at-0,
- Wires stuck-at-1.

The bridging faults are observed between the wires that run parallel, where there is a likelihood of having a short. The parallel wires affected with bridging faults, only when subjected to opposite logic values, fail to transmit correct logic value at the other end. Therefore, applying opposite logic values sensitizes the bridging fault between the parallel running wires [Str02]. As the counter-based TPG applies exhaustive test patterns to the WUTs, the opposite logic is present on each wire segment with both wire segments monitored by an ORA.

A stuck-off CIP would prevent the transmission of the signal between the wire segments that it connects. If the test patterns applied along the set of WUTs are received correctly by the ORA, the CIP of the set of WUTs, cannot be stuck-off [SNLA02].

A stuck-on CIP would be unable to break the connection between the wire segments that it connects, similar to a bridging fault. In order to detect the CIP stuck-on fault, the opposite logic values are applied at the two ends of the CIP and each end of the wire

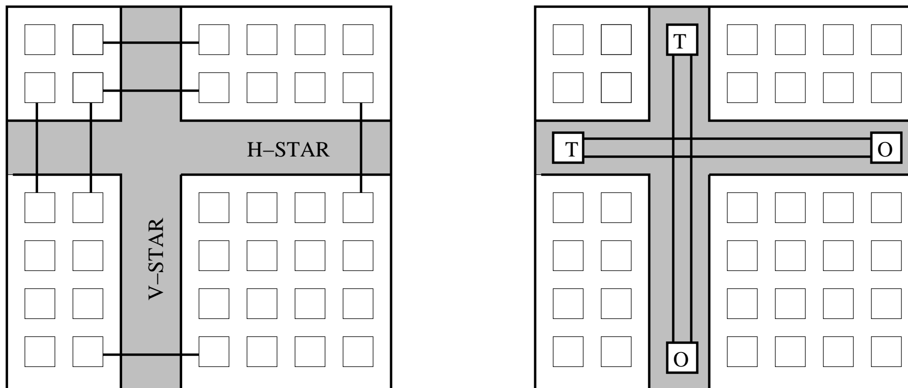


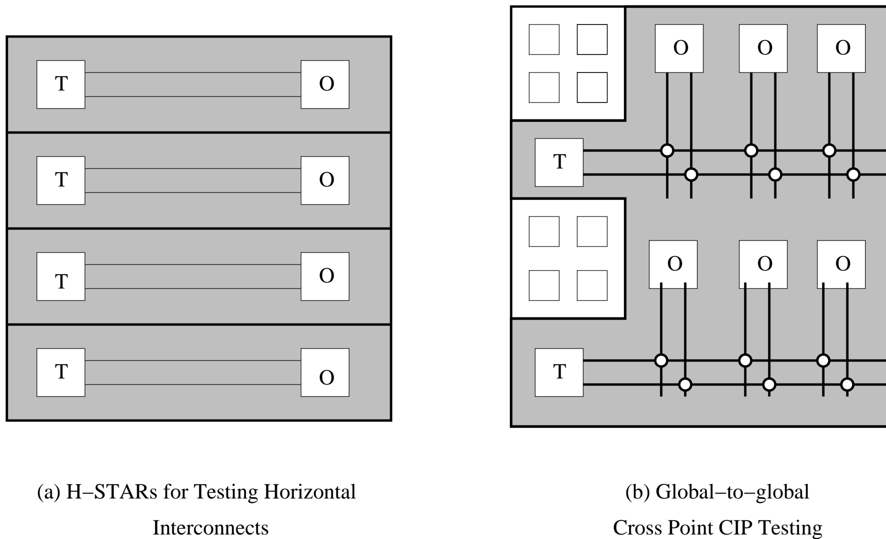
Figure 2.9: FPGA Floorplan for Online Interconnect Testing [AES01]

segment controlled by the CIP is monitored by ORAs. The fault is detected when an incorrect value is read from one of the ends of the CIP [SNLA02].

The interconnects shorted to VDD or GND would cause the wire stuck-at 1 or 0 faults. The broken continuity in the metal wires are referred to as opens in the wires. The test to detect the wire stuck-at faults and opens in the wire is to determine the ability of the wire to transmit both 0 and 1 between the ends.

Previous Implementations of Interconnect BIST

For on-line BIST, horizontal and vertical *self test areas* (STARs) are configured in the FPGA without disturbing the system function configured in other part of the FPGA. The STARs then rove across the FPGA such that different routing resources of FPGA are brought under test. Once a fault is located, the faulty routing resources can be excluded from being used again or used as long as it does not form contention with other resources. The TPG, WUT and ORA configuration for roving STARs is shown in Figure 2.9.



(a) H–STARs for Testing Horizontal Interconnects

(b) Global–to–global Cross Point CIP Testing

Figure 2.10: FPGA Floorplan with "Galaxy" BIST [SNLA02]

In the off-line BIST, the system function is not configured while the FPGA is being tested. Therefore, FPGA interconnects are configured with parallel horizontal-STARs and vertical-STARs, also referred to as *galaxy* BIST phases (Figure 2.10). Thus the test patterns are applied to the global and local interconnect resources under test in different test phases. While the number of test phases remains the same, the off-line BIST reduces the number of times the FPGA needs to be reconfigured in order to test the routing resources, as compared to the on-line BIST approach [SNLA02].

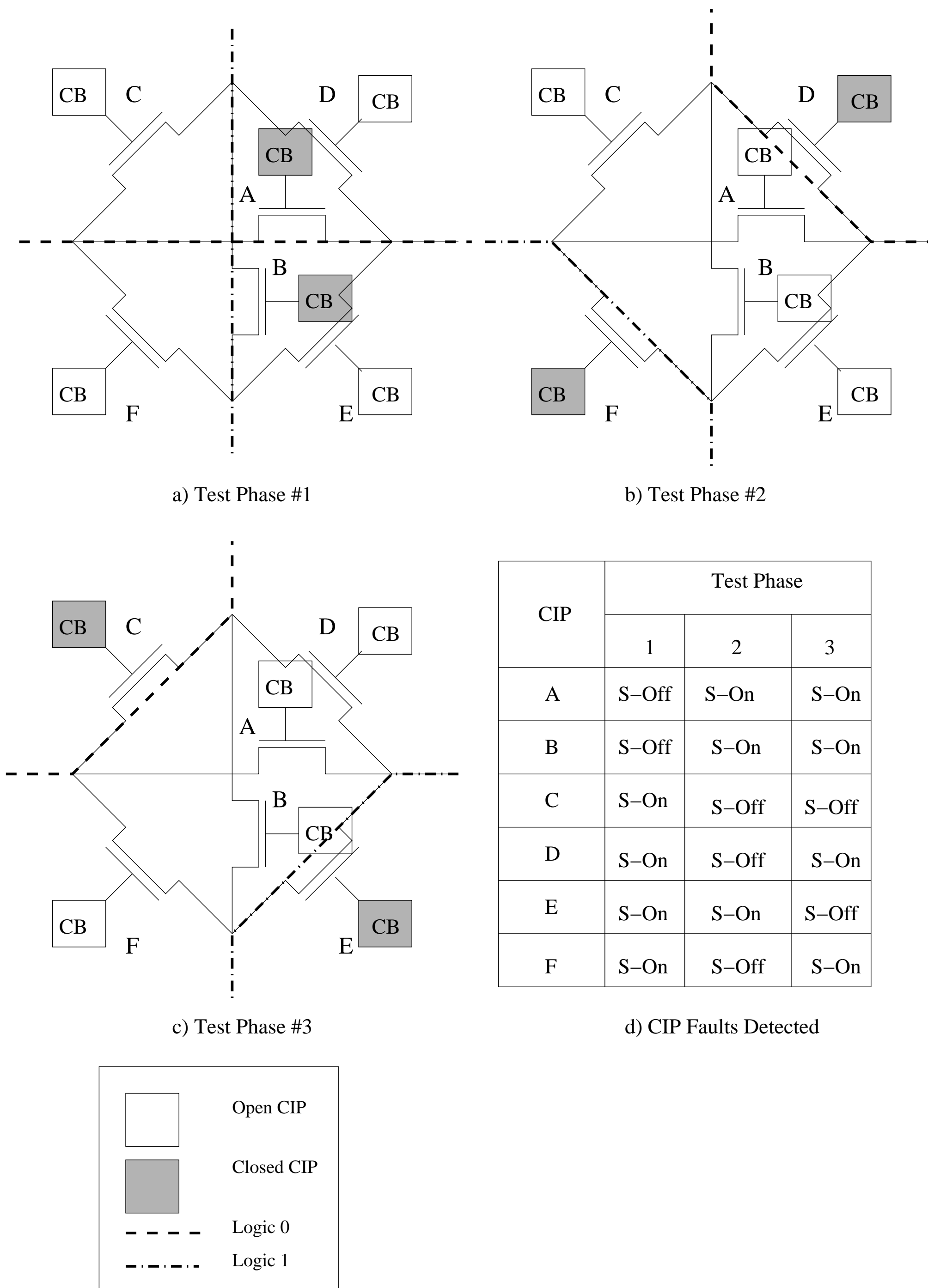
2.6.3 BIST for Xilinx FPGAs

The work related to BIST for Xilinx FPGAs can be found in [SCKA96] [RFZ97] [RPFZ99] [SXCT00] [SLS03] [RZ00]. All of them target the Xilinx 4000 series FPGA family. Xilinx 4000 and Spartan-I family of FPGAs do not support partial reconfiguration.

[RFZ97] and [SXCT00] target Single interconnects that span between the adjacent PLBs. The methodology described in [RFZ97] is one of the first attempts to target the switch-box CIPs. The paper discusses testing of switch-box CIPs for stuck-on and stuck-off faults in three test phases. This is elaborated in Figure 2.11. The approach presented in [SXCT00] generates a parity bit for the WUTs consisting of single lines. It seeks to eliminate the fault masking inherent in comparison-based ORAs, where both sets of WUTs have equivalent faults. Another advantage of this approach is that the TPG has to control only a single set of WUTs instead of two as in the case shown in Figure 2.8. The Single lines tested by this approach comprise only about 10% of the total routing resources of Xilinx 4000 series FPGA. These are simpler to test than MUX CIPs and global routing resources that crisscross the complete PLB array [SLS03]. The design of the TPG is also more complicated because of the overhead of calculating parity as opposed to simple counter-based TPG used in [SLS03].

[SLS03] uses comparison-based ORAs for testing all the logic as well as interconnect resources available in Xilinx 4000 and Spartan-I. The BIST strategy implemented is derived from the earlier works such as [AS01] [HGWS99] [SNLA02]. For logic BIST, two identical TPGs drive the test patterns to the BUTs. The structure of the TPG is different depending on the mode of operation being tested. The output of each BUT is compared by two comparison-based ORAs on both the sides. The BIST test sessions are column oriented due to the presence of more vertical routing resources than the horizontal ones and the presence of dedicated carry chain routing that is vertically oriented.

For interconnect BIST, comparison-based ORAs and 2-bit counter-based TPGs are used. The 2-bit counter-based TPG is implemented within a single PLB of Xilinx 4000 family for generating 4-bit test patterns using the current and next state of the counter.



CIP	Test Phase		
	1	2	3
A	S–Off	S–On	S–On
B	S–Off	S–On	S–On
C	S–On	S–Off	S–Off
D	S–On	S–Off	S–On
E	S–On	S–On	S–Off
F	S–On	S–Off	S–On

Figure 2.11: Complete Testing of Switch Boxes [SWHA98]

A 0,1 and 1,0 combination exists in any pair of the four bits of the test pattern, at least once in the four test pattern sequences [SLS03].

A minimum of 12 test phases are required to completely test the PLBs, including the dedicated carry-logic and 206 configurations are required to test the programmable interconnect resources [SLS03]. A third of the test configurations of programmable interconnect are attributed to the presence of wider MUX CIPs that require one configuration for each of its inputs [SLS03] [SNLA02]. Other reasons cited for the higher number of test configurations than reported in the earlier works are: reduced observability of dedicated carry logic for testing, sharing of routing resources with the adjacent PLBs, the local routing resources disallowing inputs to and outputs from the PLBs to come from and go to buses on any side of the PLB [SLS03].

Usually a scan chain mechanism would be used to for retrieving the ORA results as shown in Figure 2.12. As illustrated in [HGWS99], scan chain interfaced with IEEE 1149.1 boundary scan interface improves diagnostic resolution. However, due to the lack of architectural features in the Xilinx 4000 and Spartan series PLBs, retrieving BIST results through a scan chain connected to an IEEE 1149.1 boundary scan interface requires additional test phases. Thus the Pass/Fail results from the ORAs are obtained through complete configuration memory readback.

2.6.4 Using JBits API to Generate Interconnect BIST Configurations

Testing logic resources of an FPGA using JBits API is claimed in [SMG01]. The approach described does not configure the FPGA with logic or routing BIST architecture. The JBits program described in this approach does functional level testing of the configuration memory of the FPGA and the LUTs. The JBits program writes an FPGA

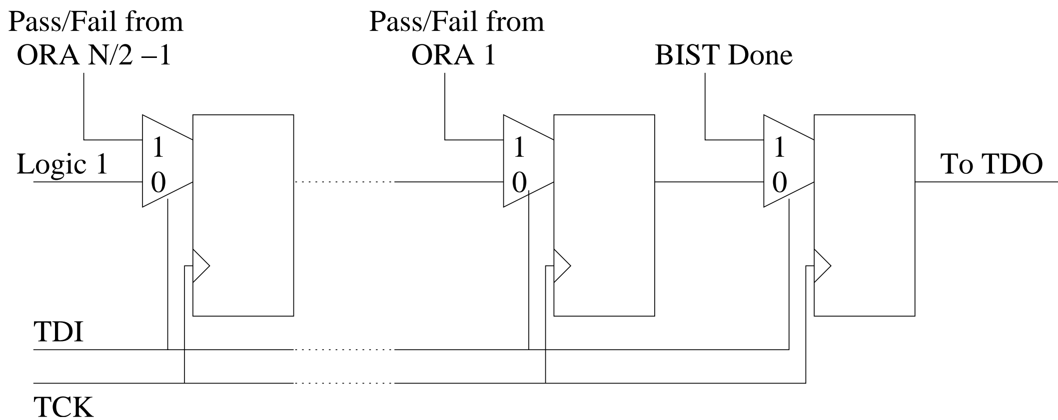


Figure 2.12: Scan Cell Interfacing with IEEE 1149.1 [HGWS99]

configuration into the configuration memory and then reads the configuration data back. While testing the LUTs, the JBits program writes a '0' value into the LUTs and reads back the data from the LUTs. Any mismatch between the two, is signaled on the user input/output terminal. The method described for interconnect testing, configures two LUTs with 16-bit shift registers. The first shift register drives logic patterns on the WUT. The second shift register records the outputs. The contents of the second LUT are then read back using configuration memory readback. The approach does not consider any of the fault models for the interconnect test. Therefore, fault coverage is an issue. The approach considers the MUX in the PLB that drives the wire as a part of the wire. Therefore, either the MUX or the wire may be faulty if there is a mismatch between the data pattern being driven and the data pattern recorded.

[FH03] explores testing interconnect resources by implementing routing BIST using JBits. The counter-based TPG used is 11 PLBs tall and 2 PLBs wide. The comparison-based ORA used has height of 20 PLBs. Of the Single, Hex and Long wires, only Single wires are tested. The approach only tests switch box CIPs. As the Hex wires are not

tested, the CIPs at the either end of the Hex wires also remain untested. The CIPs are not tested for stuck-on (stuck-closed) faults. Out of 960 switch boxes in a Virtex XCV150 chip that features a 24×36 array of PLBs, 776 switch boxes are tested. In this approach, it is not possible to determine the location of the fault. For the approach to work, an FPGA should feature a PLB array with at least 20 rows. This raises concerns about the scalability of the approach. Finally, the memory required to store the configuration, is not optimized through partial reconfiguration.

2.7 Thesis Statement

Improvements in performance of BIST for FPGAs can be obtained by minimizing the size of the configuration data required for each test phase and minimizing the amount of readback data that needs to be read in order to extract the BIST results. With partial reconfiguration, regularity across the test phases can be exploited to generate the partial bitstreams that are much smaller in size than the full configuration bitstreams. With the partial configuration memory readback, it is possible to read only the contents of the ORA columns instead of the complete configuration memory. This thesis examines the improvements that can be obtained with partial reconfiguration and readback. It also examines implications of the design and generation of logic and routing BIST configurations as a result of partial reconfiguration and readback. While this thesis focuses on Xilinx Virtex-I and Spartan-II FPGAs the proposed methods can be used in any FPGA that supports partial reconfiguration and readback.

The JBits API contains complete functionality to generate logic BIST configurations for Virtex-I and Virtex-II FPGAs. In order to generate logic BIST configurations for Virtex-I and Virtex-II family of FPGAs using JBits, RTPCores can be designed for

counter-based TPGs, BUTs configured in different modes of operation and comparator-based ORAs. The RTPCores can be routed by a parent core. In this thesis, we identify a semi-automatic routing method using JRoute that is suitable for generating routing BIST configurations. The response of the ORAs can be retrieved using the partial configuration memory readback facility offered by the Virtex architecture. Strictly speaking, JBits API is a collection of user abstractions built over the configuration memory map of the FPGA. Therefore, porting JBits API to other FPGAs requires knowledge of the configuration memory map of the FPGA, which FPGA manufacturers are reluctant to share with their customers. However, methods exist to identify the correspondence between the bitstream offsets and the resources controlled. Therefore, this method of generating interconnect configurations using JBits may be ported to other FPGAs as long as the contract with JBits is fully implemented for the target FPGA.

Chapter 3

Partial Reconfiguration and Readback for Logic BIST

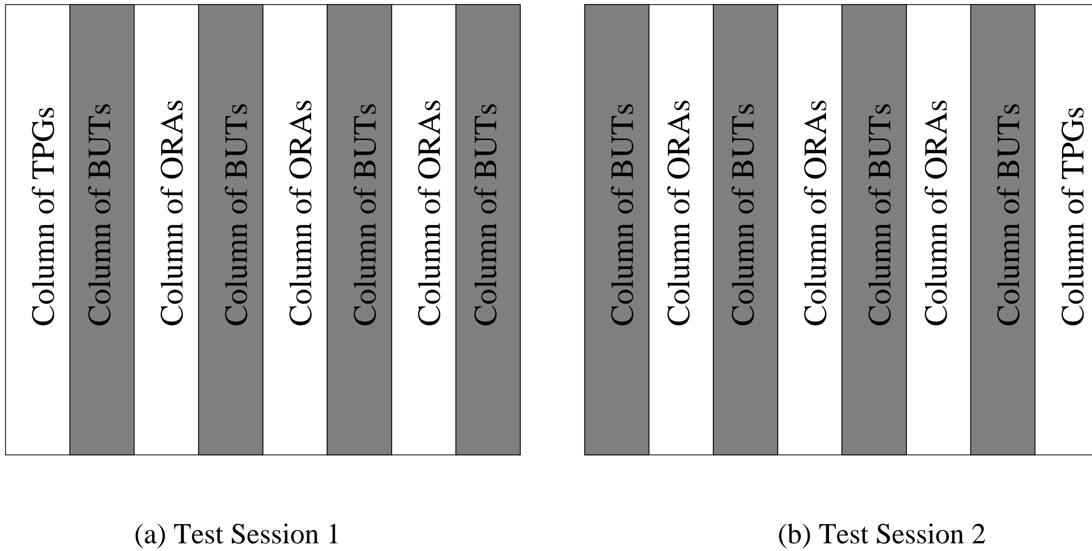
The performance of BIST depends on the number of test phases and test sessions, the total time to download the test phases, the size of the memory required to store all BIST configurations needed to guarantee 100% stuck-at fault coverage and the time required to retrieve the ORA *Pass/Fail* results. Partial reconfiguration can be effectively used as a strategy to reduce download time and to minimize the size of the memory required to store all logic and routing BIST configurations. In this chapter we present the experiments performed to quantify the performance improvement in terms of reduced time required while loading the logic BIST configuration by partial reconfiguration and reading the results by partial configuration memory readback. In the next chapter we discuss the minimizing configuration time required to load the routing BIST configurations using partial reconfiguration.

In each logic BIST test phase, PLBs configured as BUTs are tested in one of their modes of operation. In the next test phase, the mode of operation of the BUTs is changed. The difference between the configuration bitstreams for the two test phases, is expected to be small as now only the BUTs are configured in the different mode of operation, while the TPGs, ORAs and routing usually maintain the same functionality and configuration. Partial readback of the configuration memory would reduce the time required to retrieve the ORA results compared to reading the entire configuration memory.

3.1 Floorplan of Logic BIST to Aid Partial Reconfiguration

The frame organization of the Virtex-I and Spartan-II architecture is column based. If the BUTs are also aligned in the columns then all the changes in the configuration bitstream are limited to the frames associated with the columns of BUTs. This would reduce the amount of configuration data needed to be stored and downloaded; as after the first full configuration, only data that needs to be saved in the memory and downloaded for the next BIST configuration are the frames that have changed from the configuration already in the place. However, if the floorplan is row oriented, there would be a change in the configuration of BUTs which are now row-based. This would result in a change in the configuration of every configuration memory column that holds the PLB configuration. Hence, there would be a change in the frame-data corresponding to those columns. Thus the number of frames that need to be loaded is twice the number of frames that need be loaded if the floorplan of the BIST test phase were column oriented, since half as many PLB columns are under test in a given test session.

Another important effect of having a column-oriented test session is observed while performing partial configuration memory readback to retrieve the ORA Pass/Fail results at the end of each BIST sequence. In the column-oriented BIST test phases, the ORA results would be confined to $(\frac{N}{2} - 1) \times 2$ frames, as there are two ORAs per column. In order to retrieve the ORA Pass/Fail results, only the frames that map the flip-flops in the PLBs configured as ORAs need to be read. The flip-flops containing the Pass/Fail results of the test phase lie in different frames and therefore have different minor addresses. As a result, the number of ORA Pass/Fail results is the number of ORA columns multiplied by two. However, in the row-oriented BIST test phase, it is necessary to read back N $\times$



(a) Test Session 1

(b) Test Session 2

Figure 3.1: Floorplan for BIST Test Session

2 frames to retrieve all the ORA Pass/Fail results as there would be $(\frac{N}{2} - 1) \times 2$ ORA Pass/Fail results in every PLB column.

Therefore the floorplan of a BIST test session i.e. TPG, BUT and ORA, should be column-based to aid partial reconfiguration and readback as shown in the Figure 3.1.

3.2 Generating Partial Reconfiguration Files

In this section we present the process of generating the partial reconfiguration files for logic and interconnect BIST. A JBits program or a C program may be written to produce the XDL files containing the netlist of logic or interconnect BIST circuits for Virtex-I and Spartan-II families of FPGAs. We used a C program that automatically generates XDL files for the logic BIST test phases, developed by Dr. Charles Stroud. This program is run with specific command line arguments to directly generate the XDL

files for the target chip. The XDL files are then converted into NCD files using the program "xdl.exe" as follows:

```
xdl.exe -xdl2ncd outfile.xdl
```

The logic BIST configuration bitstreams for the target chip are generated from the NCD files using the BitGen program. The partial bitstreams can be generated following the process described in the next subsection.

3.2.1 Using BitGen

In order to generate the partial reconfiguration files, BitGen is run from the command line with the design netlist file (.ncd) passed as an argument and the command line option "ActiveReconfig" is disabled by default (ActiveReconfig:No). This means that the generated bitstream would contain the shut down instruction (Shutdown and AGHIGH commands) and the GSR signal would get activated after the partial reconfiguration is done [Xil02c] [Xil02d]. After the partial reconfiguration is complete, the GSR signal clears the ORA flip-flops and initializes the TPGs and BUTs for the next test phase.

The following is an example usage of BitGen for generating partial BIST configuration files:

```
bitgen -g ActiveReconfig:No -r \
bist_phase00.bit bist_phase01.ncd bist_phase01.bit
```

Here *-r* option is used to create the partial bitstream (bist_phase01.bit). This file contains frames that are different between the full configuration bitstream of the new design (bist_phase01.ncd) and the currently-loaded full configuration bitstream (bist_phase00.bit). The "\" simply indicates the continuation of the command on the next line.

BitGen Command Line Options

The following command line options are useful when generating partial reconfiguration bitstream, readback bitstream, interconnect or logic BIST:

-b When this option is specified on the command line, BitGen produces an ASCII version of the bitstream (extension .rbt). The ASCII file represents the bitstream in human readable format. The ASCII file can be used for debugging when the bitstream is modified for the fault-injection emulation.

-g Persist When this option is specified on the command line, the readback circuit is configured in the output bitstream and contents of the configuration memory can be retrieved. This option should be enabled for retrieving the ORA Pass/Fail results using the configuration memory readback.

-g Readback When this option is specified on the command line, BitGen produces an ASCII file that contains readback commands (extension .rba) and its binary version (extension .rbb) as well as an ASCII file with readback data (extension .rbd) that can be used to verify if the configuration was loaded properly.

-l When this option is specified on the command line, BitGen produces an ASCII report file (extension .ll) that enumerates all the components in the design that can be read back or captured. The file contains information about the location of the bits

in the readback bitstream, frame address, frame offset, type of logic resource and name of the component. After the ORA frames are retrieved, the location of the ORA Pass/Fail results in the frame are indicated by the logic allocation file.

3.3 Generating a Test Plan for Logic BIST

As the partial bitstream records only the changes in the configuration memory between the two configurations, a partial reconfiguration bitstream can be downloaded only after its reference configuration precedes it. Therefore, in general, the sequence in which the full and partial bitstreams are downloaded is important. The sequence of the partial bitstreams, and consequently logic or interconnect BIST test phases, should be such that a minimum number of resources change their configuration from one phase to the next. If only a small number of logic resources change their configuration, the number of frames that change from one BIST test phase to the next is small. This results in a smaller size of the partial reconfiguration bitstream containing the BIST test phase and faster download times.

Limitations in the Virtex-I architecture prevent testing two slices in a PLB at the same time. For example, there are only eight output muxes to drive the signals from the two slices to the routing resources. If the implemented design contains any feedback loops or drives multiple outputs of the multiplexer, the number of signal outputs from the BUT that can be compared in a single BIST test phase is further reduced. The BUT has twelve output signals that need to be compared by two different ORAs. The ORA implemented in a single slice is capable of comparing three pairs of output signals from two identically configured BUTs. Due to the limited number of output multiplexers to carry the signals from BUT to ORA, only a single slice can be configured as a BUT.

Hence, in a single test phase, one can test a single slice. This doubles the number of test configurations as well as having an impact on generating a test plan for partial configurations.

In order to study the effect of partial reconfiguration on logic BIST, we generated different logic BIST configurations to bring individual slices under test and configured them in different modes of operation. There are four options available for configuring and testing the two slices.

- The slice not under test maintains the same operation mode from the previous test phase while the slice under test changes its mode of operation (Scenario 1),
- Both slices change the operation mode simultaneously (Scenario 2),
- The slice under test alternates from slice 0 and slice 1 in the successive test phases and the slice not under test maintains its mode of operation from the previous configuration (Scenario 3),
- The mode of operation is changed keeping the same slice under test and the slice not under test maintains its mode of operation from the first configuration (Scenario 4).

The C program is capable of generating all four scenarios described above for two consecutive test phases. This program demonstrates the improvement in time required to load test configurations and is sufficient to draw conclusions about the test plan that would benefit the most from the partial reconfiguration. The size of the bitstream containing full configuration of a test phase and consequently, the configuration time

required to load the bitstream, does not vary over the test phases. Therefore, we are able to generalize our results for all logic BIST test phases.

Initially, the FPGA is fully configured with the bitstream containing both slices configured in test phase 1, and slice 0 is tested. The subsequent configurations are partial configurations. The slice under test is indicated by shaded region in Figure 3.2. Scenario 1 (Figure 3.2(a)) illustrates the case when one of the slices is partially reconfigured with the second test configuration (from test phase 1 to test phase 2), while the other slice maintains its previous configuration. In Scenario 2 (Figure 3.2(b)), the case under investigation is when both slices change their test configurations (from test phase 1 to test phase 2) at the same time. As noted before, only one slice can be tested. In Scenario 3 (Figure 3.2(c)), slice 1 is brought under test while both slice 0 and slice 1 remain in test phase 1. Finally, in the Scenario 4 (Figure 3.2(d)), the slice under test is first tested in the test phase 1, then tested in test phase 2 while other slice maintains its configuration.

The Table 3.1 shows the command list executed to prepare the full and partial bitstream files for executing the test plan in the Scenario 1. The first four lines of the command listing shown in the Table 3.1 generate the corresponding NCD files from the XDL files. On line 5, the full configuration bitstream is generated from the initial configuration, i.e. where the slice 0 and 1 are both configured with test phase 1. The line 6 lists the command line for generating a partial bitstream containing only the frames differing between the full configuration (LogicBIST_WestP1S0O0.bit) and the new configuration (LogicBIST_WestP2S0O1.ncd). The result is a partial bitstream (LogicBIST_Part_West_P12S00O01.bit). We compile a full configuration using the command on line 7. This is the full configuration with which the FPGA is configured, after loading the full configuration bitstream, followed by the partial bitstream generated by

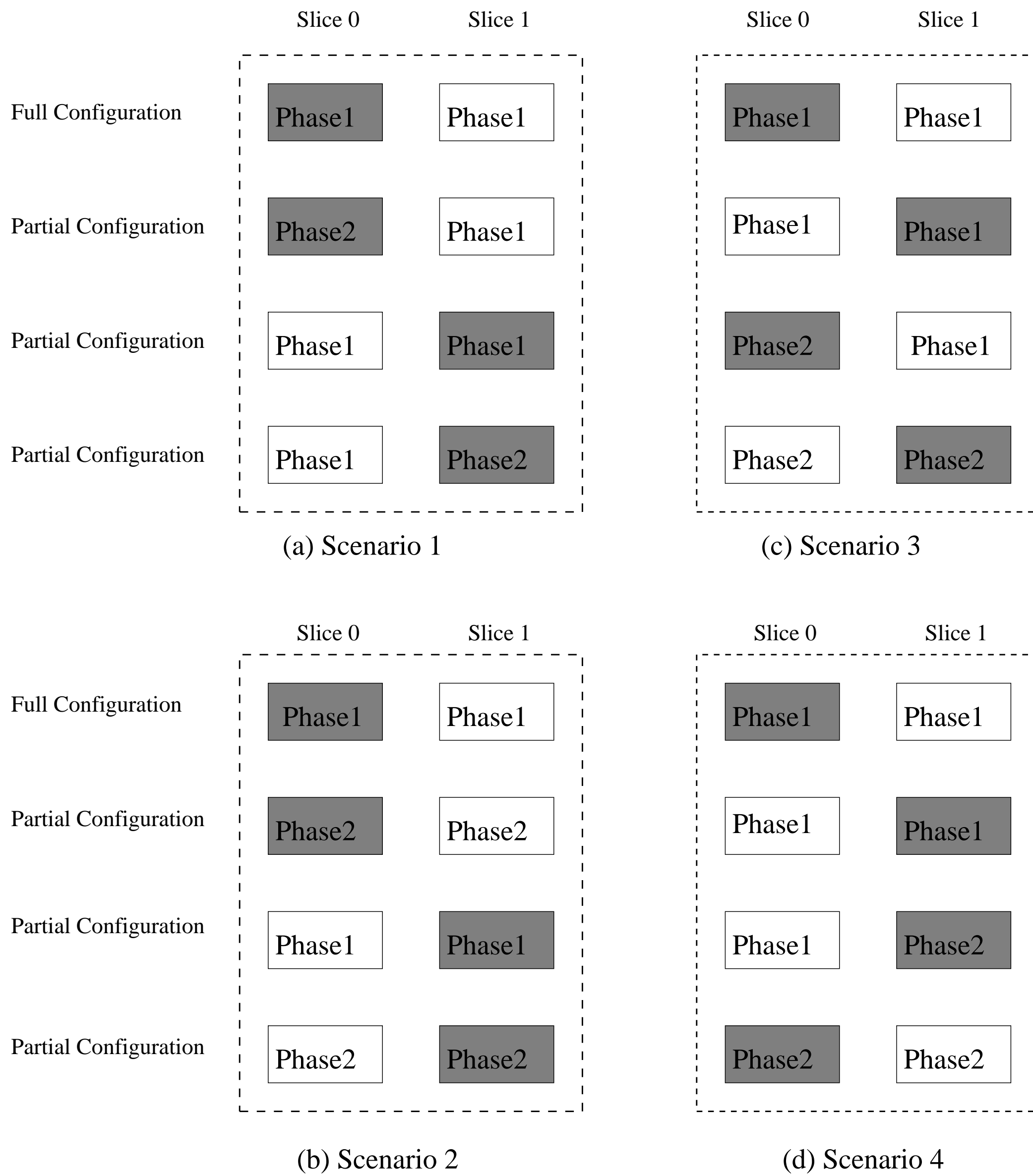


Figure 3.2: Four Different Test Plans for Testing Two Slices

line 6. The full configuration is needed while generating the next partial configuration. The lines 7 and 9 are exactly same as line 5 except that the bitstreams are generated for different test phases. The lines 8 and 10 are exactly same as line 6 except that partial bitstreams contain the difference between LogicBIST_WestP2S0O1.bit and LogicBIST_WestP1S1O0.ncd (line 8) and LogicBIST_WestP1S1O0.bit and LogicBIST_WestP2S1O1.ncd (line 10).

Table 3.1: Command Listing for Scenario 1

Line	Command
1.	xdl.exe -xdl2ncd LogicBIST_WestP1S0O0.xdl
2.	xdl.exe -xdl2ncd LogicBIST_WestP2S0O1.xdl
3.	xdl.exe -xdl2ncd LogicBIST_WestP1S1O0.xdl
4.	xdl.exe -xdl2ncd LogicBIST_WestP2S1O1.xdl
5.	bitgen -d -l -b LogicBIST_WestP1S0O0.ncd
6.	bitgen -d -l -b -w -g Persist:Yes -g ActiveReconfig:No -r LogicBIST_WestP1S0O0.bit LogicBIST_WestP2S0O1.ncd LogicBIST_Part_West_P12S00O01.bit
7.	bitgen -d -l -b -w -g Persist:Yes LogicBIST_WestP2S0O1.ncd
8.	bitgen -d -l -b -w -g Persist:Yes -g ActiveReconfig:No -g Readback -r LogicBIST_WestP2S0O1.bit LogicBIST_WestP1S1O0.ncd LogicBIST_Part_WestP21S01O01.bit
9.	bitgen -d -l -b -w -g Persist:Yes LogicBIST_WestP1S1O0.ncd
10.	bitgen -d -l -b -w -g Persist:Yes -g ActiveReconfig:No -g Readback -r LogicBIST_WestP1S1O0.bit LogicBIST_WestP2S1O1.ncd LogicBIST_Part_WestP12S11O01.bit

In order to generate the logic BIST configurations for Scenario 2, we use the sequence of commands given in Table 3.2. The flow of commands is essentially the same as that of Scenario 1.

Table 3.2: Command Listing for Scenario 2

Line	Command
1.	xdl.exe -xdl2ncd LogicBIST_WestP2S0O0.xdl
2.	xdl.exe -xdl2ncd LogicBIST_WestP2S1O0.xdl
3.	bitgen -d -l -b LogicBIST_WestP1S0O0.ncd
4.	bitgen -d -l -b -w -g Persist:Yes -g ActiveReconfig:No -r LogicBIST_WestP1S0O0.bit LogicBIST_WestP2S0O0.ncd LogicBIST_Part_WestP12S00O00.bit
5.	bitgen -d -l -b -w -g Persist:Yes LogicBIST_WestP2S0O0.ncd
6.	bitgen -d -l -b -w -g Persist:Yes -g ActiveReconfig:No -r LogicBIST_WestP2S0O0.bit LogicBIST_WestP1S1O0.ncd LogicBIST_Part_WestP21S01O00.bit
7.	bitgen -d -l -b -w -g Persist:Yes LogicBIST_WestP1S1O0.ncd
8.	bitgen -d -l -b -w -g Persist:Yes -g ActiveReconfig:No -r LogicBIST_WestP1S1O0.bit LogicBIST_WestP2S1O0.ncd LogicBIST_Part_WestP12S11O00.bit

In order to generate the logic BIST configurations for Scenario 3, we use the sequence of commands given in Table 3.3. The flow of commands is essentially the same as that of Scenario 1 and 2. We do not repeat the process of converting XDL to NCD files, as they would be available to us from the Scenario 1 and 2.

Table 3.3: Command Listing for Scenario 3

Line	Command
1.	bitgen -d -l -b -w -g Persist:Yes -g ActiveReconfig:No -r LogicBIST_WestP1S0O0.bit LogicBIST_WestP1S1O0.ncd LogicBIST_Part_WestP11S01O00.bit
2.	bitgen -d -l -b -w -g Persist:Yes LogicBIST_WestP1S1O0.ncd
3.	bitgen -d -l -b -w -g Persist:Yes -g ActiveReconfig:No -r LogicBIST_WestP1S1O0.bit LogicBIST_WestP2S0O0.ncd LogicBIST_Part_WestP12S10O00.bit
4.	bitgen -d -l -b -w -g Persist:Yes LogicBIST_WestP2S0O0.ncd
5.	bitgen -d -l -b -w -g Persist:Yes -g ActiveReconfig:No -r LogicBIST_WestP2S0O0.bit LogicBIST_WestP2S1O0.ncd LogicBIST_Part_WestP22S01O00.bit

In order to generate the logic BIST configurations for Scenario 4, we use the sequence of commands given in Table 3.4. The flow of commands is essentially the same as that of Scenarios 1, 2 and 3. The partial bitstream (LogicBIST_Part_WestP11S01O00.bit) and the full configuration as a resultant of that (LogicBIST_WestP1S1O0.bit) are common between Scenario 3 and Scenario 4, as the sequence of first two configurations is the same.

Table 3.4: Command Listing for Scenario 4

1.	bitgen -d -l -b -w -g Persist:Yes -g ActiveReconfig:No -r LogicBIST_WestP1S1O0.bit LogicBIST_WestP2S1O0.ncd LogicBIST_Part_WestP12S11O00.bit
2.	bitgen -d -l -b -w -g Persist:Yes LogicBIST_WestP2S1O0.ncd
3.	bitgen -d -l -b -w -g Persist:Yes -g ActiveReconfig:No -r LogicBIST_WestP2S1O0.bit LogicBIST_WestP2S0O0.ncd LogicBIST_Part_WestP22S10O00.bit

Table 3.5: Partial Frames

Scenario 1				Scenario 2			
Config	Slice 0 Phase	Slice 1 Phase	Frames	Config	Slice 0 Phase	Slice 1 Phase	Frames
Full	1	1		Full	1	1	
Partial	2	1	138	Partial	2	2	264
Partial	1	1	195	Partial	1	1	274
Partial	1	2	138	Partial	2	2	264

Table 3.6: Partial Frames

Scenario 3				Scenario 4			
Config	Slice 0 Phase	Slice 1 Phase	Frames	Config	Slice 0 Phase	Slice 1 Phase	Frames
Full	1	1		Full	1	1	
Partial	1	1	92	Partial	1	1	92
Partial	2	1	274	Partial	1	2	264
Partial	2	2	92	Partial	2	2	92

3.4 Experimental Results for Logic BIST

The partial bitstream is composed of the frames that are different between the configuration with which the FPGA is currently configured and the new configuration. Table 3.5 and Table 3.6, give the mode of operation for each slice and the number of configuration data frames of the current configuration that differ from the previous configuration. The first column in the tables indicates that the first configuration is a full configuration while the subsequent configurations are partial. In Scenario 4, the total number of frames of the bitstreams required to execute the test phases 1 and 2 would be least. However, in the Scenario 4, the configuration of both slices is changed at the same time. As more and more test phases are considered, as opposed to two here, the number of frames changed while switching the phase would be a dominant factor. The Scenario 1 shows a lower number of frames differing when changing the test phase. In this Scenario,

the slice under test is configured in one test phase while the other slice maintains its configuration to test phase one. As a result, when all the test phases are considered, the Scenario 1 may be expected produce fewer changed frames while switching the test phases. Therefore, Scenario 1 depicts a test plan that benefits the most from the partial reconfiguration.

The configuration time is the time to load the FPGA with the specific design. The configuration time is directly proportional to the size of the bitstream. If all the other factors, like the rate at which the configuration bitstream is loaded, and the mode in which the FPGA is configured, remain the same, the gain in the configuration time is a function of the ratio of the size of the partial bitstream in bytes to the size of the full bitstream in bytes. The "Ratio" field in the Table 3.7 indicates the ratio of the size of the partial bitstream in bytes and the size of the corresponding full bitstream in bytes. From the Table 3.7 it is clear that as the devices get bigger (XC2S15 has PLB array of 8x12 while XCV50 and XC2S50 have PLB matrix of 16x24), the ratio of the size of the full bitstream to the size of the partial bitstream increases. Not only that, the ratio is highest when the partial bitstream comprises two logic BIST phases with only a few differences in configuration. Thus the benefits of partial reconfiguration are more pronounced for the logic BIST test configurations for larger devices. The reason is that full reconfiguration becomes a more and more expensive process as the PLB array grows and that high architectural regularity in a logic BIST test phases translates to lower numbers of frames changing their configurations with the next logic BIST phase.

Table 3.7: Sizes of Partial Bitstreams vs. Full Bitstreams

XC2S15							
Full Bitstream				Partial Bitstream			
Direction	Phase	Slice	Size (bytes)	Size (bytes)	Ratio	Previous Phase	
						Phase	Slice
West	1	0	24797	NA	NA	NA	NA
West	1	1	24797	6301	3.94	1	0
West	2	1	24797	8597	2.88	1	1
West	2	0	24797	6301	3.94	2	1
West	2	0	24797	8717	2.84	1	0
West	1	1	24797	8597	2.88	2	0
West	2	1	24797	8717	2.84	1	1
XC2S50							
Full Bitstream				Partial Bitstream			
Direction	Phase	Slice	Size (bytes)	Size (bytes)	Ratio	Previous Phase	
						Phase	Slice
West	1	0	69985	NA	NA	NA	NA
West	1	1	69985	6829	10.25	1	0
West	2	1	69985	17179	4.07	1	1
West	2	0	69985	6829	10.25	2	1
West	2	0	69985	17805	3.93	1	0
West	1	1	69985	18561	3.77	2	0
West	2	1	69985	17717	3.95	1	1
XCV50							
Full Bitstream				Partial Bitstream			
Direction	Phase	Slice	Size (bytes)	Size (bytes)	Ratio	Previous Phase	
						Phase	Slice
West	1	0	69984	NA	NA	NA	NA
West	1	1	69984	6828	10.25	1	0
West	2	1	69984	17716	4.07	1	1
West	2	0	69984	6828	10.25	2	1
West	2	0	69984	17804	3.93	1	0
West	1	1	69984	18472	3.77	2	0
West	2	1	69984	17716	3.95	1	1
XC2S50							
Full Bitstream				Partial Bitstream			
Direction	Phase	Slice	Size (bytes)	Size (bytes)	Ratio	Previous Phase	
						Phase	Slice
East	1	0	69985	NA	NA	NA	NA
East	1	1	69985	6389	10.95	1	0

East	2	1	69985	16489	4.24	1	1
East	2	0	69985	6389	10.95	2	1
East	2	0	69985	16577	3.93	1	0
East	1	1	69985	17245	4.22	2	0
East	2	1	69985	16489	4.24	1	1

3.5 Partial Configuration Memory Readback to Retrieve the BIST Results

After the completion of a BIST test phase, the flip-flops in each of the ORAs contain the Pass/Fail result for that test phase. Using the boundary scan interface, the results can then be shifted to the TDO pin and subsequently read by the system controller to determine faulty/fault free states of the FPGA. A diagnostic algorithm can be performed on the BIST results to determine the location of the faulty PLBs [AS01]. Using the boundary scan interface, the results can be retrieved using four methods: 1) using existing boundary scan registers, 2) user defined internal scan registers, 3) configuration memory readback, 4) integrated ORA and scan register [HGWS99]. The third and fourth methods have proved to be the most valuable in actual BIST implementations [SLS03] [SNLA02] [AS01].

The Virtex series devices allow partial configuration memory readback of the configuration memory. The sequence of commands given to accomplish the partial configuration memory readback is given in section 2.4. The FAR is set to the major and minor address of the frame containing the ORA flip-flop that contains the Pass/Fail results. As all the logic resources in a single column share a common major address, all the flip-flops in one column have a common major address. The minor addresses of the flip-flops, however, depend on the slice that flip-flops belong to. The flip-flops in the same slice will have common minor address irrespective of the row or the column containing the

slice. Therefore, all the flip-flops lying in a single column of the PLB array and in a single slice have common major and minor addresses. As there are two slices in each PLB, each having two flip-flops, regardless of the number of PLBs in a column, flip-flops in a PLB column are mapped into four different configuration frames. Thus, the Pass/Fail results from all the ORAs in a single column can be retrieved by reading only four frames. The column-based logic BIST floorplan for XCV100 would contain $\frac{M}{2}$-1=14 ORA columns, where M equals the number of columns of PLBs. Therefore, all the Pass/Fail results from all ORAs in the FPGA would be contained in 56 frames. In the selectMAP mode, the readback data frame of XCV100 contains thirteen 32-bit words [Xil02d].

In order to perform the partial configuration memory readback on 56 frames we must first write $56\times6+1$ command words into the FPGA as the partial configuration memory readback is iterative in nature [MG00] [Xil02d]. Therefore, the total number of command bytes written for the partial configuration memory readback of all the ORAs in a BIST test phase for an XCV100 is 1124. The total number of clock cycles required would depend on the configuration mode the FPGA is set in. For SelectMAP mode of configuration, one byte can be read from/written to the configuration memory in each clock cycle. Therefore, the total number of bytes in the readback to retrieve the Pass/Fail results from all the ORAs in a BIST test phase on XCV100 would be $56\times13\times4 = 2912$. The total number of clock cycles to perform partial configuration memory readback for the ORA Pass/Fail results would be $2912+1348 = 4260$. The number of clock cycles to perform partial configuration memory readback would see an eight-fold increase in the boundary scan mode which is a serial mode. Therefore, the total number of clock cycles required to perform partial configuration memory readback in boundary scan mode would be $4260\times8=34080$.

The full configuration memory readback requires 90216 bytes of readback bitstream to be read and 28 bytes of command words to be written. The number of clock cycles required for full configuration memory readback is 21.2 times the number of clock cycles required for partial configuration memory readback.

For the integrated ORA and scan register, the results of a test phase are latched in each ORA. There are ($\frac{N*M}{2}$-N)×4 ORA Pass/Fail results that need to be scanned out. For XCV100, the number of ORA Pass/Fail results that need to be scanned out would be 1120, which is 20 times faster than partial memory readback through boundary scan interface. The Virtex architecture does not contain a dedicated multiplexer for utilizing the internal scan chains. Therefore, implementing this approach on Virtex-I or Spartan-II FPGAs presents a logic overhead of one multiplexer per ORA. Also, the flip-flops in the scan chain and the boundary scan signals need to be routed. The lack of dedicated routing and logic resources to implement user defined scan chains manifests itself in the increase in the number of test configurations. The number of BUT inputs that can be compared in ORA reduce from six to four, as two inputs need to be reserved for scan-in from the previous stage and shift control input. Therefore, to test twelve BUT outputs it takes three instead of two configurations.

From the above calculations, it can be deduced that the number of clock cycles required for partial configuration memory readback using the SelectMAP mode is $4\times\frac{M}{2}$-$1\times ff_n$[6+FLR] + 4, where M is the number of columns in the PLB array, ff_n is the number of flip-flops in a PLB and FLR is the length of the configuration memory frame. Therefore, the total number of clock cycles required for partial configuration memory readback is proportional to the product of the number of columns in the PLB array (M), the number of flip-flops in a PLB (ff_n) and the length of the configuration memory

frame (FLR). While the number of clock cycles required for integrated ORA and scan register is proportional to the product of the number of columns in the PLB array (M) and the number of rows in the PLB array (N). As N would be always smaller than the product of FLR and the number of flip flops in a PLB, the integrated ORA scan chain would fair better than partial configuration memory readback despite one extra partial configuration.

The comparison of implementing three approaches is given in the Table 3.8. The table depicts the number of bytes that need to be retrieved with each of the methods, as the PLB array size increases from XCV100 to XCV150 to XCV1000. The full configuration memory readback for XCV150 requires 121536 bytes to be read and 28 command bytes to be written. Similarly, full configuration memory readback for XCV1000 requires 745524 to be read and 28 command bytes to be written. For partial configuration memory readback, XCV150 requires 72 frames, each of 60 bytes, to be retrieved. Therefore, it requires 4320 bytes of data to be read and 1732 command bytes to be written. For partial configuration memory readback, XCV1000 requires 192 frames, each consisting of 152 bytes, to be retrieved. Therefore, it requires 29184 bytes of data to be read and 4612 command bytes to be written.

Table 3.8: Comparison of Boundary Scan Access Method and Partial Configuration Memory Readback

XCV100		
Method	Additional Logic	Number of clock cycles
Full Memory Readback (Boundary Scan)	None	721728
Partial Memory Readback	None	4260

(SelectMAP)		
Partial Memory Readback (Boundary Scan)	None	34080
Integrated ORA and Scan Chain	1 MUX/PLB + routing	1120
XCV150		
Full Memory Readback (Boundary Scan)	None	972512
Partial Memory Readback (SelectMAP)	None	6052
Partial Memory Readback (Boundary Scan)	None	48416
Integrated ORA and Scan Chain	1 MUX/PLB + routing	1632
XCV1000		
Full Memory Readback (Boundary Scan)	None	5964416
Partial Memory Readback (SelectMAP)	None	33796
Partial Memory Readback (Boundary Scan)	None	270368
Integrated ORA and Scan Chain	1 MUX/PLB + routing	12032

3.5.1 Commands for Partial Configuration Memory Readback

The command set for retrieving the BIST results from Virtex FPGAs using partial configuration memory readback is given in the Table 3.9. The FAR register is loaded with the start frame address. The starting frame address comprises the major address and the minor address of the frame of data to be read back. The number of 32-bit words to be read back is calculated from the frame length of the device multiplied by the number of frames to be read back. This value is indicated by the bits (10:0) of FDRO register. In this example, the column number 11 in the XCV50 is to be read back. The major address of this column is 2. The packet data to be loaded into the FAR register is as follows: The bits (26:25) are set to $(00)_2$ to signify that the address lies in the one of the PLB columns, the bits (24:17) in FAR register indicate the major address, bits (16:9) indicate the minor address. Therefore, the FAR register is written with the

address as (00040000)h. The column consists of 48 frames, each of 44 bytes. The read back data is preceded by the a 32-bit pad data word. One pad frame is also included while calculating the number of words to be read back. Therefore, the total number of 32-bit words to be readback would be 49×48=588.

Table 3.9: Bitstream for Partial Configuration Memory Readback

FFFF FFFF	
AA99 5566	Synchronization Word
3000 2001	Packet Header: Write to FAR register
0004 0000	Packet Data: Starting frame address
3000 8001	Packet Header: Write to CMD register
0000 0004	Packet Data: RCFG
2800 624C	Packet Header: Read from FDRO
0000 0000	Flush the pipeline

3.6 Summary

The floorplan of the logic BIST should be column-based to aid partial reconfiguration and partial configuration memory readback. The plan that benefits the most from partial reconfiguration is identified as Scenario 4 where except for the first configuration, a slice under test changes its mode of operation while the other slice maintains its configuration. The configuration time for loading the logic BIST test phase is significantly reduced using the partial configurations instead of the full configurations. As the devices get bigger, the ratio of the size of equivalent full configuration bitstreams to the size of partial bitstreams increases. Thus, less time is required to load the test phases. The regularity in the architecture of logic BIST across the test phases is advantageous for the partial reconfiguration. Partial configuration memory readback is a preferred over the full configuration memory readback to retrieve the ORA Pass/Fail results, however

partial configuration memory readback still lags behind the integrated ORA and scan chain method of retrieving the ORA Pass/Fail results.

Chapter 4

Generating Routing BIST Configurations using JBits

In this chapter we explore how JBits API can be used to generate test configurations for testing interconnect resources. Though it is possible to generate logic as well as routing BIST configurations using JBits, in this thesis, a method has been implemented for generating routing BIST configurations for Virtex-I and Spartan-II FPGAs using JBits API. For implementing routing BIST using JBits API, one approach is to use the RTPCores to configure the PLBs as counter-based TPGs or comparator-based ORAs. These RTPCores are referred to as *BIST RTPCores*. The routing BIST architecture implemented by the BIST RTPCores and the fault models targeted by the routing BIST architecture, are presented in section 4.1.

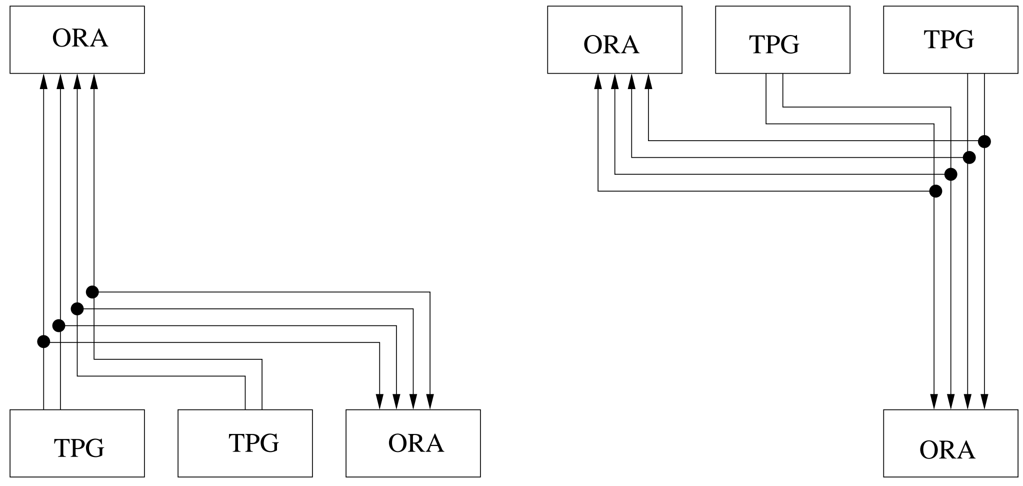
The JBits program developed for this thesis automatically generates bitstreams and XDL files containing routing BIST configurations using the JBits API for Virtex-I. The BIST RTPCores used as TPGs and ORAs and the RTPCores responsible for routing the WUTs between the TPGs and ORAs, as well as populating the PLB array, are described in the section 4.2. The JBits program outputs a bitstream for Virtex-I FPGAs and an XDL file that enumerates the design specified in the program. The header of the XDL file denotes the chip, package and speed grade information [Xil00]. As the architecture of the Virtex-I and Spartan-II FPGAs is the same, the header of the XDL is modified to reflect the target Spartan-II chip, package and speed grade information. The modified XDL file is further processed to generate a bitstream that tests the interconnects of Spartan-II FPGAs. The details about command line options and results obtained by

running the program are given in the section 4.3. This section also presents an estimate of the number of routing BIST configurations required to test all routing resources of Virtex-I and Spartan-II FPGA.

In the subsection 4.4.3, we present a set of four BIST configurations to completely test switch box CIPs encountered in Virtex-I and Spartan-II, for CIP stuck-on and CIP stuck-off faults. We also propose the desired modifications in order to generate the configurations to test the switch box CIPs using JBits. The section 4.5 proposes the desired modifications in the current implementation of the JBits program in order to generate the configurations to test the switch box CIPs using JBits.

4.1 Overview of Routing BIST Architecture

The BIST RTPCores instantiate the architecture for routing BIST: Two counter-based TPGs generate two-bit exhaustive test patterns over eight parallel WUTs. Two comparison-based ORAs compare the WUTs from different TPGs for a mismatch. The TPGs, WUTs and ORAs are shown in the Figure 4.1. The BIST RTPCores configure two slices with *TPGCounterCores* and two slices with *ORACores* which are comparison-based ORA that compare the WUTs from different TPGs for a mismatch. The BIST RTPCores inherit the functionality of an RTPCore. This enables the BIST RTPCores to take the advantage of HDL-like features and let the router do the most of the routing. When the specific resource under test, like a Hex wire or Single wire, is to be specified, the BIST RTPCores directly refer the physical resources of the FPGA into the design. In order to ensure the fault detection offered by this architecture, we consider the fault models described in the Chapter 2. The routing BIST configurations generated by the program, test for bridging faults, wire stuck-at faults, CIPs stuck-off and CIPs stuck-on



(a)H–STAR and V–STAR while Testing Horizontal Hex Interconnects going E–W or Vertical Hex Interconnects going S–N

(b)H–STAR and V–STAR while Testing Horizontal Hex Interconnects going W–E or Vertical Hex Interconnects going N–S

Figure 4.1: Horizontal and Vertical Interconnect Resources Tested for Shorts and Opens

faults affecting Single as well as Hex wires. To sensitize the bridging faults between adjacent wires, both logic levels, logic-0 and 1, need to be presented at least once in the test phase on the parallel running Hex and Single wires. To identify the wires that run parallel, and thus are susceptible to bridging faults, we must have the physical layout of the FPGA. In the absence of this we rely on the graphical layout presented by the FPGA Editor. As the exhaustive test patterns would contain both combinations of opposite logic values (0,1 and 1,0) at least once, the bridging faults between the parallel wires should be detected. If any of the CIPs along the WUTs is affected by stuck-open faults, then the comparison-based ORA would record a mismatch between the outputs of the two identically configured TPGs driving the WUTs. In order to detect the CIPs stuck-closed fault, the TPG controls both the segments associated with the open CIP and applies opposite logic patterns to each segment. Thus both combinations of logic, 01 and 10, need to be tested.

4.1.1 Testing the Interconnects in Parallel

Multiple Hex and Single interconnects can be driven by an output mux. The TPG drives exhaustive test patterns to the fan-outs of the output mux and tests them in parallel. The number of Hex and Single resources that can be reached from an output multiplexer are listed in the Table 4.1. A set of interconnects routable from one output multiplexer does not overlap with another set of interconnects reachable from the other output multiplexer. While identifying the Hex and Single interconnects that may be tested in parallel, we notice that only the following types of Hex interconnects may be grouped together: Hex East and Hex South, or Hex East and Hex West, or Hex North and Hex South, or Hex North and Hex West. Testing East and South interconnects and West and North interconnects together is simpler compared to testing East and West and North and South interconnects. When testing the Hex wires, in the first interconnect BIST phase, two identically configured TPGs drive two-bit exhaustive test patterns on four Hex interconnects going South-North and four Hex interconnects going East-West. Two comparison-based ORAs detect any mismatch that results from a fault. In the second interconnect BIST phase, four Hex interconnects going North-South and four Hex interconnects going West-East are tested. The TPGs and the ORAs are arranged in the PLB array as shown in Figure 4.2(a). When testing the Single interconnects, the TPGs and the ORAs are arranged in the PLB array as shown in Figure 4.2(b).

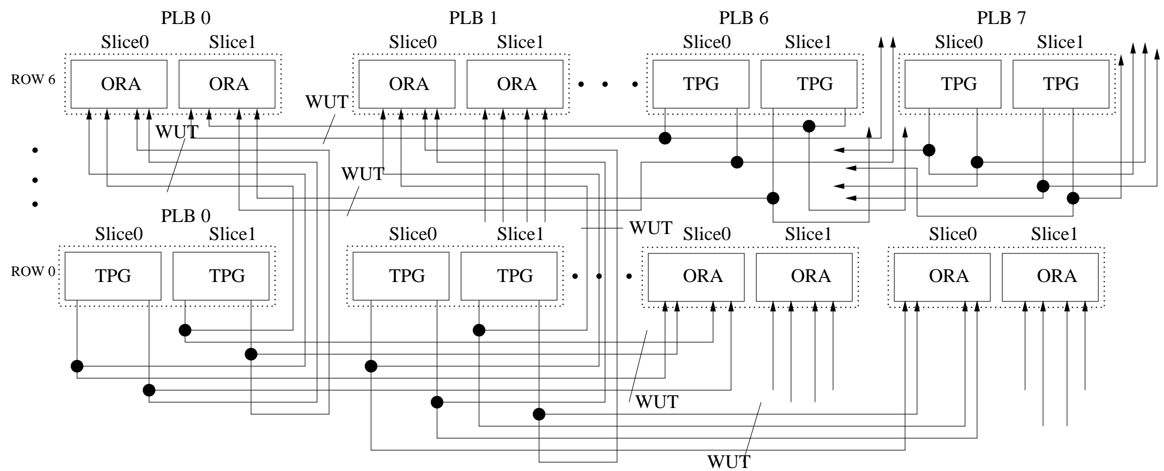
Table 4.1: Connectivity between Output Multiplexers and Interconnects

Interconnect	Output Multiplexer							
	0	1	2	3	4	5	6	7
Hex_Horiz_East	1	1	1	1	1	1	1	1
Hex_Horiz_West	1	1	1	1	1	1	1	1
Hex_Vert_South	1	1	1	1	1	1	1	1
Hex_Vert_North	1	1	1	1	1	1	1	1
Single_East	1	2	1	2	1	2	1	2
Single_West	1	2	1	2	1	2	1	2
Single_South	2	1	2	1	2	1	2	1
Single_North	2	1	2	1	2	1	2	1

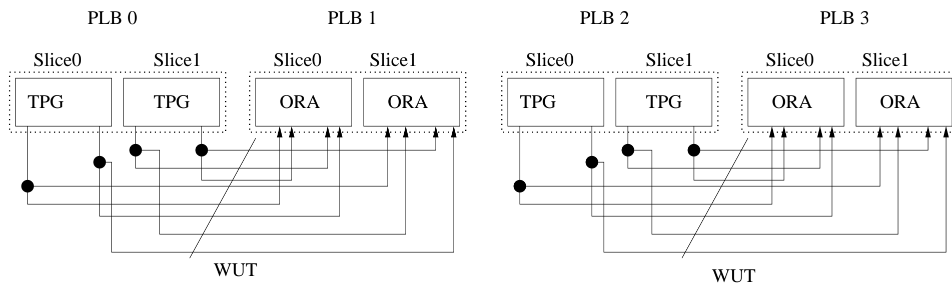
4.2 The Routing BIST RTPCores

The components of a JBits program observe a certain hierarchy. The components at a certain level in the hierarchy are designed to perform certain tasks. While designing the BIST RTPCores, the hierarchy should be taken into consideration. This enables the RTPCores to make optimum use of the functionality offered by the JBits API. Observing these restrictions also results in maintainable code. Some of the important design decisions are:

- The parameters that define and modify the behavior of the RTPCore,
- Which RTPCores in the hierarchy should configure PLBs,
- Which RTPCores in the hierarchy should configure the routing,
- How the bitstream and XDL file is generated,
- How does the programmer control routing, and
- Which logical nets and buses are reflected in the XDL file and which are ignored.



(a) Testing Hex Lines for Bridging Faults and Opens



(b) Testing Single Lines for Bridging Faults and Opens

Figure 4.2: Hex and Single Wires Tested for Shorts and Opens

As these considerations are generic and pertain to every JBits application, discussion on all of the above considerations is beyond the scope of this thesis. In this thesis we restrict ourselves to the discussion on how these considerations come into play while developing BIST RTPCores. The steps described in the Appendix A and Appendix B adhere to hierarchy in JBits API and are flexible enough to let the programmer control the routing. The program developed for this thesis is given in Appendix C and contains four classes – SimpleRouteBISTApp, SimpleRouteBIST, TPGCounterCore and ORACore.

SimpleRouteBISTApp is the entry point of the application. Therefore, it takes care of user interaction and command line arguments. Many useful functions pertaining to the user interaction are contained in the class com.xilinx.util.JBitsCommandLineApp. Therefore, SimpleRouteBISTApp derives its functionality from this class. SimpleRouteBISTApp performs following functions:

1. Parses the command line arguments.

2. Populates entire PLB array with the routing BIST RTPCore generated by SimpleRouteBIST. In order to accomplish this, the parameters to the RTPCore SimpleRouteBIST are varied, as will be explained in subsection 4.2.4.

3. Generates the XDL file, CTF file and the bitstream for the target FPGA.

The class SimpleRouteBIST is a parent RTPCore. The RTPCore derives its functionality from the class com.xilinx.JBits.CoreTemplate.RTPCore, which contains useful functions to define logical nets and ports. The behavior of SimpleRouteBIST depends on two parameters – i and j, which identify row and column numbers in the PLB array,

respectively, where the TPGs should be placed. When testing the Hex lines, the ORA is always six PLB blocks away from the PLB configured as a TPG. Therefore, when testing Hex lines, SimpleRouteBIST class configures a slice of the PLB six blocks away as an ORA. When testing Single lines, SimpleRouteBIST class configures the ORA in a slice of the PLB adjacent to the one configured as a TPG. SimpleRouteBIST class then calls a method *internalRoute* with parameters *width*, *row* and *col*. The parameter *width* tells the method the size of the WUT group. The parameter *width* depends on the number of flip-flops contained in a slice, which is two in case of Virtex-I. The parameters *row* and *col* specify the row and column number of the TPG. The method then routes the WUT group of size $width \times 2$ originating from the PLB(*row*,*col*) to the PLB(*row*,*col+6*) when generating routing BIST configuration testing Hex wires running East-West. When WUTs are Hex wires running West-East, the co-ordinates of the PLB configured as ORA change to (*row*,*col-6*). Similarly, when testing the vertical Hex wires going from North-South, the ORA would be placed in the PLB (*row-6*,*col*). Finally, when the WUTs are vertical Hex wires with direction South-North, the coordinates of the PLB configured as an ORA would be (*row+6*,*col*). SimpleRouteBIST performs following functions:

1. Assigns placement constraints to the child cores depending on the parameters,
2. Configures two slices of a PLB as identical counter-based TPGs,
3. Configures a slice of another PLB as an ORA comparing the outputs of the identical TPGs, and
4. Configures routing between the two PLBs with the specified WUTs.

The classes TPGCounterCore and ORACore are child RTPCores. The child RTPCores are spawned by the parent RTPCore SimpleRouteBIST. Therefore the placement constraints are defined by the parent RTPCore. The behavior of these RTPCores depends on the parameter *width*. The value of this parameter determines how many slices the RTPCore occupies in the PLB array. The function of specifying the routing between the child RTPCores is also performed by SimpleRouteBIST. The architecture of each of these RTPCores is elaborated in the subsection 4.2.1 and 4.2.2. The bitstream manipulation is the responsibility of the top level RTPCore: SimpleRouteBISTApp.

4.2.1 Configuring the TPG

The program uses a counter-based TPG to generate exhaustive test patterns for the WUTs. Virtex-I and Spartan-II PLBs contain four flip-flops, two in each slice. Therefore, a Virtex-I or Spartan-II slice can be configured as a 2-bit counter, generating four test vectors. The logical buses and ports defined in the class *TPGCounterCore* are shown in the Table 4.2. The RTPCore *SimpleRouteBIST* defines upper level logical buses. The ports connect the upper level logical buses to internally defined ones. The TPG flip-flops must be configured to be reset during the start-up sequence in ensure that the two TPGs being compared by the ORA are synchronized to produce identical test patterns during the BIST sequence.

Table 4.2: Input and Output ports of TPGCounterCore

Port	Width (in Bits)	Direction	Function
clk	1	IN	Counter Clock
dout	2	OUT	Counter Output
ce	1	IN	Counter Enable/Disable
sr	1	IN	Counter Set/Reset

4.2.2 Configuring the ORA

The functionality of the comparator-based ORA is implemented in the class *ORA-Core*. The logical input and output ports of the ORACore are shown in the Table 4.3. A Virtex-I slice configured as an ORA is shown in the Figure 4.3. As stated earlier, each of the two identical TPGs drives 2 WUTs. Four WUTs are connected to the *addr* input port of the ORACore. This input port, when the design is placed and routed, corresponds to the A1, A2, A3, and A4 inputs of the G LUT. Therefore, when the design is placed and routed, the inputs of the G LUT are connected to the WUTs. The G LUT needs to be configured to output a logic 1 if there is any mismatch between the logic values of the WUTs driving identical test patterns. Therefore, the LUT is configured with the expression *(A1 XOR A2) OR (A3 XOR A4)*.

The Pass/Fail result is latched by the flip-flop. Since the 4 inputs of G LUT have been exhausted, we cannot use the Y flip-flop to form a feedback path to the input of the G LUT. Therefore, the F LUT and the X flip-flop is used. The F LUT ORs the output of the G LUT and X flip-flop, which contains the Pass/Fail result of the test phase. Thus, the output of the G LUT needs to be connected to the Y output of the PLB to form a feedback loop to the A1 input of the F LUT. Here we utilize the facility given by JBits API to allocate FPGA physical resources, such as the Y output pin of the PLB, to any of the logical ports of the RTPCore. Using this facility, port *lutOut* is assigned to the Y output pin. The F LUT is configured to implement logic expression (A1 OR A2). Finally, the output of the X flip-flop (XQ) is connected to the A2 input of the F LUT. The X flip-flop must be configured to be reset during the start-up sequence prior to execution of the BIST sequence.

Table 4.3: Input and Output Ports of ORACore

Port	Width (in Bits)	Direction	Function
clk	1	IN	Clock Input
addr	4	IN	Input to the G LUT
lutOut	1	IN-OUT	Feedback to the Latch
oraOut	1	IN-OUT	Pass/Fail Output of the ORA
sr	1	IN	Set/Reset Input

4.2.3 Routing the WUTs

JBits API gives flexibility to the programmer to specify the physical routing after the RTPCores have been placed. The programmer can skip this step and let JRoute do the completely automatic logical-to-physical mapping at the expense of the control over the routing. In order to generate routing BIST configurations, we must maintain control over the process of routing the interconnect resources to be tested. This is the rationale behind method *internalRoute(int i, int j)*. This method may be called anywhere in the JBits program before the call to static method *connect(Bus)* in the class com.xilinx.JBits.CoreTemplate.RTPCore.Bitstream is made. This method first defines the physical sources and sink pins along with the WUTs. The output pins of the flip-flops of the two TPGs are defined as source pins. The input pins of the G LUT of the ORA are the sink pins. The static integers corresponding to the WUTs are selected from the class com.xilinx.JRoute2.Virtex.ResourceDB.CenterWires.

These physical resources are then routed using the automatic routing method. Routing RTPCores using this semi-automating method has several advantages:

1. The routing resources to be tested can be specified under user control,

2. It is not necessary to specify routing of the other elements not under test in the path,

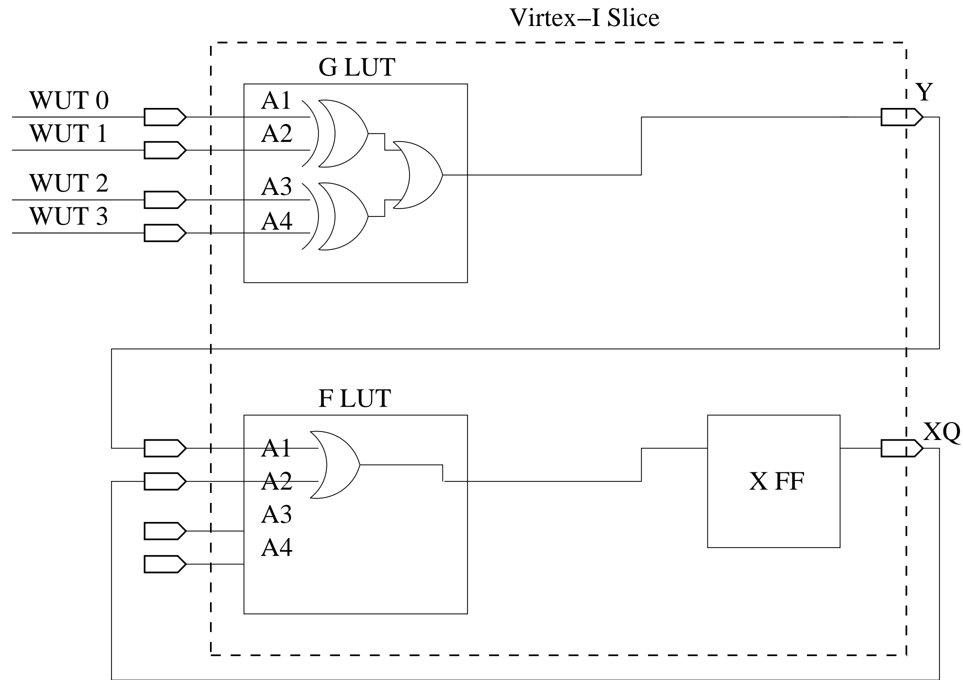


Figure 4.3: Configuration of Comparator-Based ORA

3. The complexity of routing the switch box CIPs is avoided except while testing CIPs stuck-off faults.

4.2.4 Populating the PLB Array

The class SimpleRouteBISTApp takes care of populating the PLB array with the instances of RTPCore SimpleRouteBIST. The command line arguments that are parsed in the main routine, specify the Virtex-I chip and the package being used. The number of columns and rows in the PLB array of the chip is obtained with the static methods defined in the class com.xilinx.JBits.Virtex.Devices: *getClbCols(device_name)* and *getClbRows(device_name)*, respectively. It is important to keep in mind that JBits API defines the lower left corner of the PLB array as the origin and calculates the X and

```
private void internalRoute(int w, int row, int col)
                              throws RouteException, ConfigurationException  {
        CoutSrcPin = new Pin[w];
        OraPin = new Pin[w];
        WutPin = new Pin[w];
        JRoute jroute = Bitstream.getVirtexRouter();


         CoutSrcPin[0] = new Pin(Pin.CLB, row, col, CenterWires.Slice_XQ[slice]);
         CoutSrcPin[1] = new Pin(Pin.CLB, row, col, CenterWires.Slice_XQ[slice+1]);
         CoutSrcPin[2] = new Pin(Pin.CLB, row, col, CenterWires.Slice_YQ[slice]);
         CoutSrcPin[3] = new Pin(Pin.CLB, row, col, CenterWires.Slice_YQ[slice+1]);

         OraPin[0] = new Pin(Pin.CLB, row, col+6, CenterWires.SliceG1[slice]);
         OraPin[1] = new Pin(Pin.CLB, row, col+6, CenterWires.SliceG2[slice]);
         OraPin[2] = new Pin(Pin.CLB, row, col+6, CenterWires.SliceG3[slice]);
         OraPin[3] = new Pin(Pin.CLB, row, col+6, CenterWires.SliceG4[slice]);

         WutPin[0] = new Pin(Pin.CLB, row, col, CenterWires.Hex_Horiz_East[0]);
         WutPin[1] = new Pin(Pin.CLB, row, col, CenterWires.Hex_Horiz_East[1]);
         WutPin[2] = new Pin(Pin.CLB, row, col, CenterWires.Hex_Horiz_East[2]);
         WutPin[3] = new Pin(Pin.CLB, row, col, CenterWires.Hex_Horiz_East[3]);

         for (int i=0;i<w;i++) {
                jroute.route(CoutSrcPin[i], WutPin[i]);
                jroute.route(WutPin[i], OraPin[i]);
         }
}
```

Figure 4.4: JBits Program for Routing the WUTs

Y coordinates of the PLB accordingly. The location of the origin is different from the one followed by the FPGA Editor and Xilinx data manuals, where the PLB located at the upper left corner is regarded as the origin. This is important while determining the location of the ORA relative to the location of the TPG.

Suppose the device contains *MAX_COL* columns and *MAX_ROW* rows in the PLB array. The program that tests the horizontal Hex interconnects loops over the parameter i, which is varied from 0 to *MAX_ROW* as the outer loop and the parameter j, which is varied from 0 to *MAX_COL-6*, as the inner loop, to populate the entire PLB array with the routing BIST test phase. Once the value of j reaches *MAX_COL-6*, it can no longer be incremented as this would place the ORA beyond the maximum number of columns, which triggers an exception condition in the JBits program. All Virtex-I devices have a *MAX_COL* value that is a multiple of six. Therefore, the edges of the horizontal BIST RTPCore align with the PLB columns in all the Virtex-I devices. However, *MAX_ROW* is not a multiple of six. Therefore, when populating the PLB array with the vertical BIST RTPCore, the TPG and ORA need to be routed through the top IOB cell. This condition is shown in the Figure 4.5. Care should be taken that the corresponding IO pin is set in the tristate mode.

A special condition exists because the Hex lines end at the PLB six blocks over. Because of this, there is an ORA configured every six blocks. Therefore when the inner loop iterates five times, the PLB is already configured as an ORA. Therefore, any more increment in j causes the program to throw an exception condition as it tries to overwrite the PLB configured as an ORA. This condition is detected and j is incremented by six instead of one.

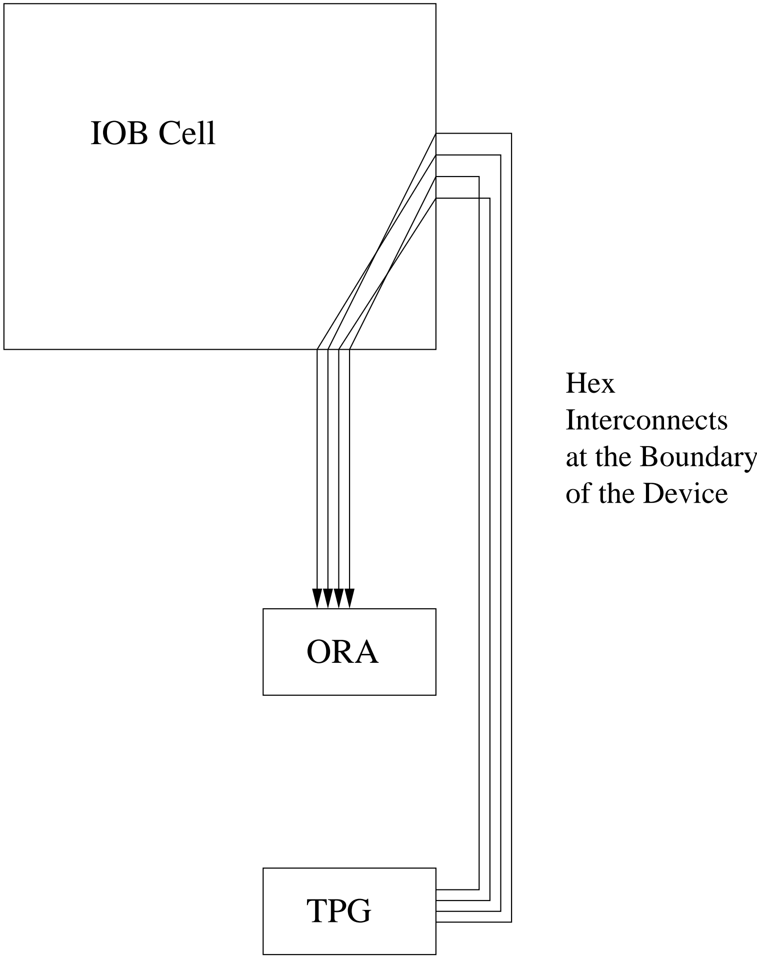


Figure 4.5: Boundary Condition for Populating PLB Array in Vertical Direction

4.2.5 Generating the XDL File

The JBits API gives allows the programmer to specify the output file format of the application. The output file format of the JBits program can be XDL and/or bitstream. The choice of the output file format is specified by calling the static methods, *generateXDL()* and/or *generateBitstream()* from class CoreOutput in the package com.xilinx.JBits.CoreTemplate.RTPCore. These method calls are placed in the *run()* method of class SimpleRouteBISTApp. The XDL file can be later converted into an NCD file using the "xdl.exe" program. The default XDL file generated by the JBits program does not contain the package and speed grade of the device and therefore does not conform to the XDL version 1.6 specifications [Xil00]. Therefore, the default XDL file generated by the JBits program cannot be converted to an NCD file without modifications. The class SimpleRouteBISTApp takes Virtex-I device name, package and speed grade as the command line parameters. The SimpleRouteBISTApp modifies the XDL file generated to reflect the package and the speed grade information in the header so that the XDL file can be converted into an NCD file and viewed in the Xilinx ISE tool suite.

4.3 Experimental Results of Routing BIST

The program takes command line arguments as listed in Table 4.4 to generate the routing BIST configurations for the target FPGA. Table 4.5 indicates all the possible values of some of the command line arguments.

Table 4.4: Command Line Arguments Available for the JBits Program

Command Line Arg	Function
Chip	Name of the Target Chip
infile	One of the null bitstreams provided by Xilinx
outfile	Name of the Output bitstream
Package	Package Identifier for the Target Chip
SpeedGrade	Speedgrade of the Target Chip
Rec_Under_Test	WUT Groups To Test

Table 4.5: Possible Values of the Command Line Arguments

<table>
<tr><td>Chip</td><td>xcv50
xcv300
xcv800
xcv1000
xc2s15
xc2s30
xc2s50
xc2s100
xc2s150
xc2s200</td></tr>
<tr><td>Rec_Under_Test</td><td>Hex_Horiz_East_0_1_2_3
Hex_Horiz_West_4_6_8_10
Hex_Horiz_East_5_7_9_11
Hex_Vert_North_0_1_2_3
Hex_Vert_South_4_6_8_10
Hex_Vert_North_5_7_9_11
Single_East_0_1_2_3
Single_East_4_5_6_7
Single_East_5_9_10_11
Single_East_12_13_14_15
Single_East_16_17_18_19
Single_East_20_21_22_23
Single_West_0_1_2_3
Single_West_4_5_6_7
Single_West_8_9_10_11
Single_West_12_13_14_15
Single_West_16_17_18_19
Single_West_20_21_22_23
Single_North_0_1_2_3
Single_North_4_5_6_7
Single_North_8_9_10_11
Single_North_12_13_14_15
Single_North_16_17_18_19
Single_North_20_21_22_23
Single_South_0_1_2_3
Single_South_4_5_6_7
Single_South_8_9_10_11
Single_South_12_13_14_15
Single_South_16_17_18_19
Single_South_20_21_22_23</td></tr>
</table>

The Virtex FPGAs can be directly configured with the bitstream generated by the program developed for this thesis. The bitstream of Virtex-I FPGA is not compatible with Spartan-II. Therefore, the bitstream generated at the output of the program cannot be used on Spartan-II FPGAs. An XDL file contains symbolic names of the nets used in the design and the configuration of the routing and PLB resources. As Virtex-I and Spartan-II have equivalent routing and PLB architectures, the XDL file generated for one architecture can be used for the other one, with a little modification. Thus, the XDL files are more portable across the architectures than the bitstream. Hence, an XDL file containing a description of a routing BIST configuration for a Virtex-I FPGA, may be used with a little modification to generate the same routing BIST configuration for a Spartan-II FPGA. In order to generate the routing BIST configuration for the Spartan-II, the command line arguments supplied to the program indicate the name and the package of the target Spartan-II chip. The name of the target chip defines the total number of rows and columns in the PLB array. This information is used to set the maximum limit on the values of parameters i and j.

The program for generating the BIST configuration for testing four Hex horizontal resources of Spartan-II FPGA, is run as follows:

```
java %PATH_TO_PROGRAM%/RouteBIST.SimpleRouteBISTApp \
-xc2s50 %JBits%/data/Bitstream/XCV50/null50GCLK1.bit \
%OUTPUT_DIR%/SpartanRBIST.bit Hex_Horiz_West_4_6_8_10
```

The *%PATH_TO_PROGRAM%* is an operating system variable which is set to the directory where the java package resides. The *%JBits%* is an operating system variable indicates the installation directory of JBits 2.8. The operating system variable

%OUTPUT_DIR% is the destination directory for writing the resultant bitstream and the XDL files. The "\" indicates the continuation of the command line. Finally, the command line argument *Hex_Horiz_West_4_6_8_10* instructs the program to generate routing BIST configuration for Hex horizontal resources indexed 4, 6, 8 and 10.

4.3.1 Partial Reconfiguration and Routing BIST

Partial reconfigurations can be generated from the full configurations for routing BIST using BitGen. For the BitGen tool to generate the partial bitstream containing the frames that are different between the two full configurations, it needs the design netlist (NCD) file. The XDL files generated by the JBits program are converted into design netlist files using the "xdl.exe" program. The partial bitstream is generated using the command list shown in the Table 4.6. The commands on the line 1, 3, 5 and 7 generate the bitstream and the XDL file containing full configuration for testing horizontal Hex interconnects 0, 1, 2, 3 (line 1), horizontal Hex interconnects 4, 6, 8, 10 (line 2), horizontal Hex interconnects 5, 7, 9, 11 (line 3) and vertical Hex interconnects 0, 1, 2, 3 (line 4). The XDL files generated corresponding to the lines 1, 2, 3,and 4 are: simpleRBIST_P0.xdl, simpleRBIST_P1.xdl, simpleRBIST_P2.xdl, simpleRBIST_P3.xdl, respectively. The lines number 5, 6, 7 and 8 convert the XDL files into the NCD files. The line 9 generates the full configuration bitstream with which the FPGA would be first configured. The line 10 generates the partial reconfiguration bitstream (simpleRBIST_Part_P01.bit) from the bitstream with which the FPGA is currently configured (simpleBIST_P0.bit) and the design netlist file (simpleRBIST_P1.ncd). The lines 11 and 13 are exactly same as that of line number 9 except the full bitstream is generated for different test phase. After loading the full configuration (simpleRBIST_P0.bit)

followed by the partial configuration (simpleRBIST_Part_P01.bit), the effective configuration would be simpleRBIST_P1.bit (line 11). Therefore the full configuration for Phase 1 (simpleRBIST_P1.bit) acts as the reference configuration for the subsequent partial reconfiguration (simpleRBIST_Part_P12.bit) as generated on line 12.

Table 4.6: Command Listing for Generating Partial Bitstreams for Routing BIST

Line	Command
1.	java RouteBIST.SimpleRouteBISTApp -xcv50 "%JBits%\data\Bitstream\XCV50\null50GCLK0.bit" simpleRBIST_P0.bit Hex_Horiz_East_0_1_2_3
2.	java RouteBIST.SimpleRouteBISTApp -xcv50 "%JBits%\data\Bitstream\XCV50 \null50GCLK0.bit" simpleRBIST_P1.bit Hex_Horiz_West_4_6_8_10
3.	java RouteBIST.SimpleRouteBISTApp -xcv50 "%JBits%\data\Bitstream\XCV50\null50GCLK0.bit" simpleRBIST_P2.bit Hex_Horiz_East_5_7_9_11
4.	java RouteBIST.SimpleRouteBISTApp -xcv50 "%JBits%\data\Bitstream\XCV50\null50GCLK0.bit" simpleRBIST_P3.bit Hex_Vert_North_0_1_2_3
5.	xdl -xdl2ncd simpleRBIST_P0.xdl
6.	xdl -xdl2ncd simpleRBIST_P1.xdl
7.	xdl -xdl2ncd simpleRBIST_P2.xdl
8.	xdl -xdl2ncd simpleRBIST_P3.xdl
9.	bitgen -d -l -b -w -g Persist:Yes simpleRBIST_P0.ncd
10.	bitgen -d -l -b -w -g Persist:Yes -g ActiveReconfig:No -r simpleRBIST_P0.bit simpleRBIST_P1.ncd simpleRBIST_Part_P01.bit
11.	bitgen -d -l -b -w -g Persist:Yes simpleRBIST_P1.ncd
12.	bitgen -d -l -b -w -g Persist:Yes -g ActiveReconfig:No -r simpleRBIST_P1.bit simpleRBIST_P2.ncd simpleRBIST_Part_P12.bit
13.	bitgen -d -l -b -w -g Persist:Yes simpleRBIST_P2.ncd
14.	bitgen -d -l -b -w -g Persist:Yes -g ActiveReconfig:No -r simpleRBIST_P2.bit simpleRBIST_P3.ncd simpleRBIST_Part_P23.bit

The Table 4.7 lists the size of the full bitstream in bytes, size of the partial bitstream in bytes and the ratio of the size of the full bitstream vs. size of the partial bitstream. As the size of the partial reconfiguration depends on the reference full configuration that precedes it, the phase of the reference configuration is listed in the Previous Phase column.

Table 4.7: Sizes of Partial Bitstreams vs. Full Bitstreams

XCV50					
Full Bitstream			Previous Phase		Ratio
Direction	Phase	Size (bytes)	Phase	Size (bytes)	
West	1	69968	0	33043	2.11
West	2	69968	1	29823	2.34
West	3	69968	2	30123	2.32
XC2S50					
Full Bitstream			Previous Phase		Ratio
Direction	Phase	Size (bytes)	Phase	Size (bytes)	
West	1	69978	0	33043	2.11
West	2	69978	1	29823	2.34
West	3	69978	2	30123	2.32
XCV300					
Full Bitstream			Previous Phase		Ratio
Direction	Phase	Size (bytes)	Phase	Size (bytes)	
West	1	219055	0	59283	3.7
West	2	219055	1	55407	4.0
West	3	219055	2	57155	3.8

4.3.2 Test Phase Sequence

To determine the order in which the routing resources should be tested, we experimented with the routing BIST configurations for Single and Hex resources. It is expected that smaller number of frames would differ between the two test phases testing resources

of same type (Single or Hex) as compared to the number of frames differing between the two test phases testing resources of different types. The Table 4.8 gives the size of the full configuration bitstream in bytes, size of partial configuration bitstream in bytes and the ratio of the size of full configuration bitstream in bytes to size of the partial configuration bitstream in bytes. Each test phase is identified by the type of resources tested.

The speedup due to partial reconfiguration is more observable in the case of the FPGA featuring a large PLB array (XCV300) than the FPGA featuring a small PLB array (XCV50). When two consecutive test phases test either the Hex or Single interconnects, a small number of frames differ between the two. The size of the partial bitstream in bytes is larger when the two phases involved test different routing resources.

Therefore, to reap the benefits of partial reconfiguration, for larger FPGAs, all Single interconnects should be tested in consecutive test phases and all Hex interconnects should be tested in consecutive test phases. It is observed that when a test phase testing Hex interconnects follows a test phase testing Single interconnects, a smaller number of frames differ between the two consecutive test phases. There would be only one occurrence of this event in the test sequence if all Single interconnects are tested in consecutive test phases and all the Hex resources are tested in consecutive test phases. This enables us to choose the best case scenario for the partial configuration. Therefore, the Single interconnects should be tested first.

Table 4.8: Routing BIST and Test Phase Sequence

XCV50					
Full Bitstream			Previous Phase		Ratio
Direction	Phase	Size (bytes)	Phase	Size (bytes)	
West	Hex	69978	Hex	32238	2.17
West	Single	69978	Single	19414	3.6
West	Single	69978	Hex	25122	2.7
West	Single	69978	Hex	28586	2.44
West	Hex	69978	Single	25854	2.7
West	Hex	69978	Single	27298	2.6
XCV300					
Full Bitstream			Previous Phase		Ratio
Direction	Phase	Size (bytes)	Phase	Size (bytes)	
West	Hex	219055	Hex	59283	3.69
West	Single	219055	Single	46099	4.75
West	Single	219055	Hex	63603	3.44
West	Single	219055	Hex	63419	3.45
West	Hex	219055	Single	71667	3.06
West	Hex	219055	Single	49767	4.4

4.4 Calculation of the Total Number of Interconnect BIST Configurations Required

We refer to the Virtex-I switch box shown in Figure 4.6. We envision four BIST test sessions for obtaining 100% stuck-at fault coverage for interconnect testing in Virtex-I and Spartan-II FPGAs. In the first test session, half of the Hex, Single and Long interconnect resources and the associated CIPs are tested for wire stuck-at, bridging and CIPs stuck-open faults. In the second test session, the placement of TPGs and ORAs is interchanged and other half of the Hex, Single and Long interconnects are tested. In the third test session the switch-box CIPs are tested for CIPs stuck-on and stuck-off faults.

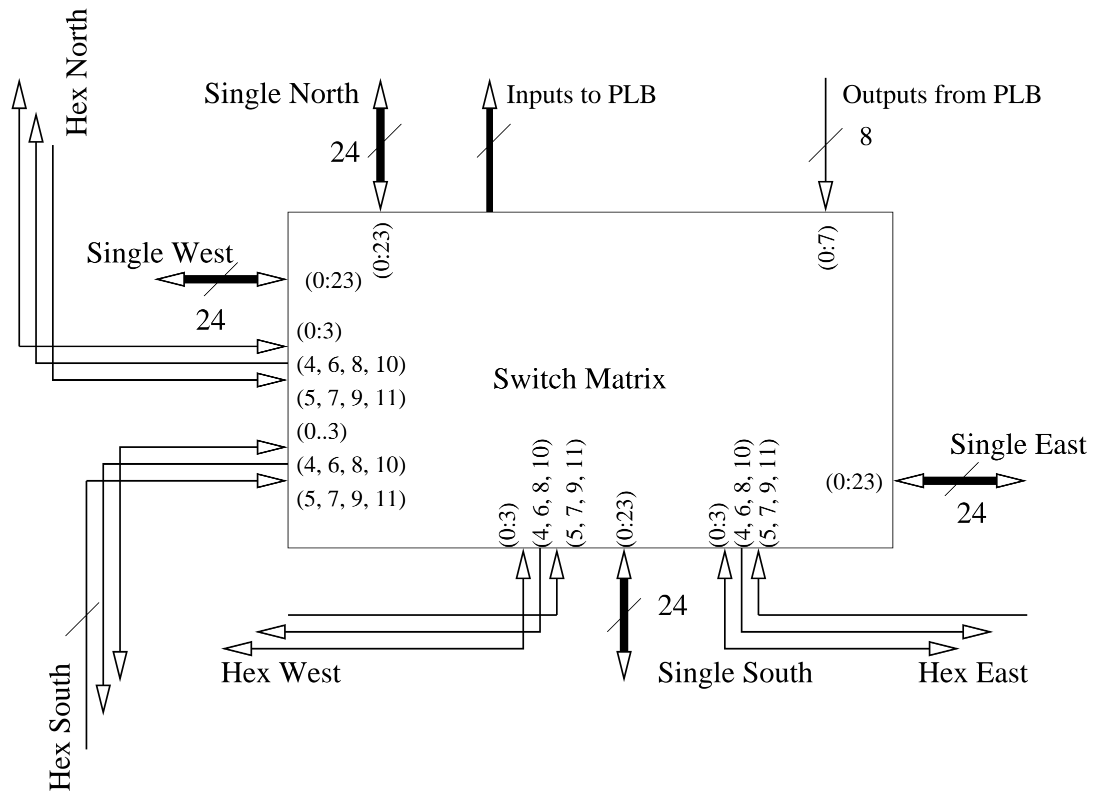


Figure 4.6: Switch Box CIP and Xilinx Interconnect Architecture

In the fourth and final test session the MUX CIPs are tested for stuck-on and stuck-off faults.

4.4.1 Hex Interconnects

Each TPG implemented in a slice is capable of driving four Hex wires in a routing BIST test phase. As there are 48 Hex wires emerging from the switch box, first six interconnect test configurations are required to detect wires stuck-at and bridging faults in half of the Hex lines. Another six interconnect test configurations which are generated by interchanging the positions of TPGs and ORAs, brings the other half of the Hex lines

under test. These test phases also test the CIPs associated with the wires for CIP stuck-open faults. These CIPs are tested for stuck-open and stuck-closed faults in the second test session. Therefore, a total of twelve configurations are generated.

4.4.2 Single Interconnects

Each TPG implemented in a slice is capable of driving four Single wires in a routing BIST test phase. A switch box provides connection to 96 Single wires. Therefore, to test the Single wires for the wire-stuck-at and bridging faults and the associated CIPs for stuck-open faults, we need a total of 24 interconnect BIST configurations. First twelve routing BIST configurations test half of the Single wires in the PLB array. Other twelve configurations test remaining half of the Single wires by interchanging position of TPGs and ORAs. The CIPs associated with Single interconnects are tested for stuck-open and stuck-closed faults in the second test session by applying opposite logic at both ends of the CIP while observing the two wire segments by two different ORAs.

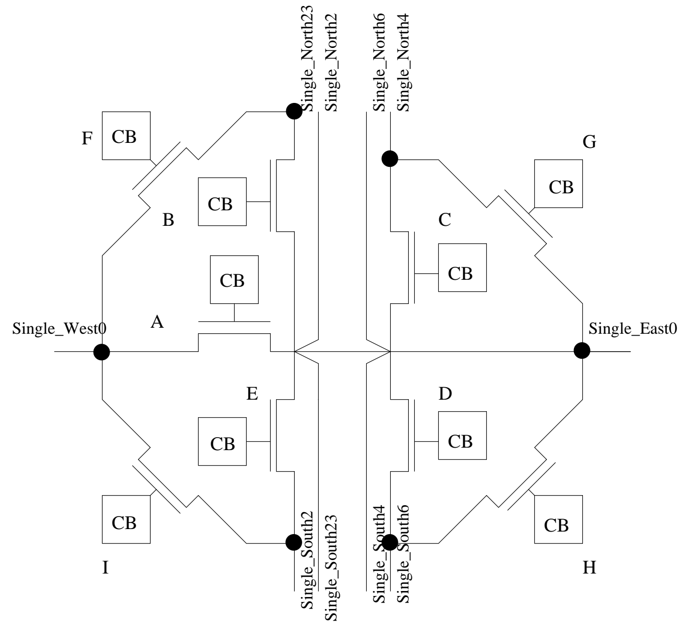
4.4.3 Switch Box CIPs

To test the switch-box CIPs for stuck-closed faults, the TPG applies opposite logic patterns on two wire segments connected by the CIP. In the approach described in [RFZ97], three test configurations were all that were required to completely test the switch box CIPs for the CIP stuck-open as well as CIP stuck-closed faults. These three configurations were suitable for testing the switch box CIPs in Xilinx XC4000 FPGAs. The architecture of the switch box CIP as encountered in a Virtex-I FPGA is different from that of its predecessors. We have deduced this architecture from JBits 2.8 documentation as provided by Xilinx and verified it against report files generated by the XDL

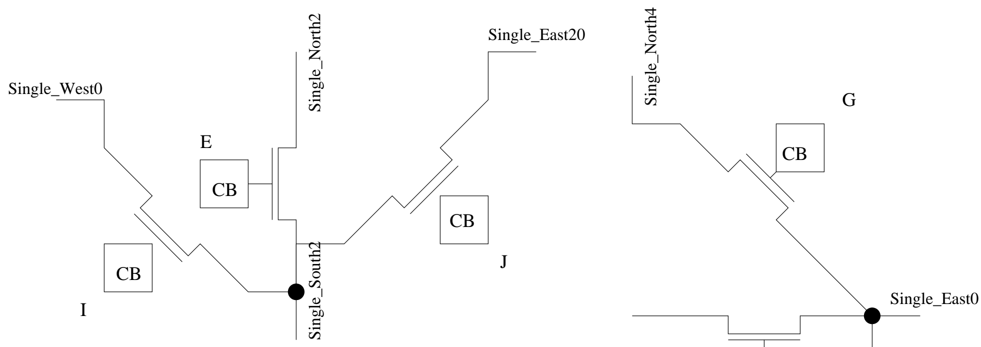
program that lists all CIPs in the Virtex-I architecture. The structure of switch box CIP found in Virtex-I devices is depicted in Figure 4.7(a). All Single North interconnects provide connection to Single South interconnects with the same index and all Single East interconnects can be connected to Single West interconnects bearing the same index e.g. interconnect Single_South23 can be connected to Single_North23 by CIP labeled *B*. Any Single West interconnect is connected to exactly one Single North interconnect and exactly one Single South interconnect, e.g. interconnect Single_West0 can be connected to Single_North23 by CIP labeled *F* and to Single_South2 by CIP labeled *I*. Similarly, Single East interconnect is connected to exactly one Single North interconnect and exactly one Single South interconnect. The possible connections for Single_South2 and Single_East0 are shown in Figures 4.7(b) and 4.7(c), respectively. The effect of this interconnect architecture is that we may connect Single_West0 to Single_North23 but there is no way to connect to Single_North23 and Single_East0 because Single_East0 provides one connection to Single North interconnect: Single_North4. In the previous architectures, like XC4000, we could have bypassed the connection Single_West0 to Single_East0 by connecting Single_West0 to Single_North23 and Single_North23 to Single_East0. The disadvantage of the switch box CIP shown in the Figure 4.7 is that it requires more test configurations to test for stuck-off and stuck-on faults than those required to test switch box CIP in XC4000.

The classes that model the CIPs and the number of unique CIPs modeled per PLB (third column) are as tabulated in Table 4.9.

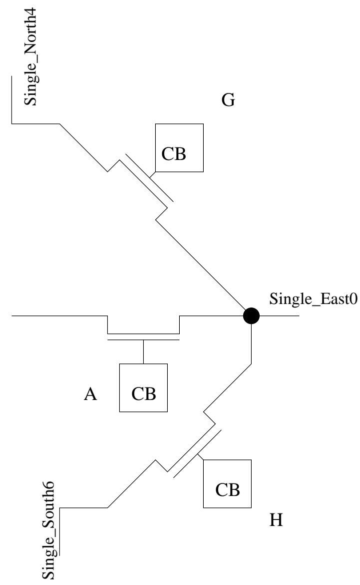
Among the four categories of CIPs, *SingleToSingle* are the switch box CIPs. The rest of them are either the break-point CIPs or cross-point CIPs. In order to test the *SingleToSingle* switch box CIPs, we propose a set of four test configurations as shown



(a) Architecture of Virtex−I Switch Box CIP



(b) Interconnects and CIPs Accessible from Single_South2



(c) CIPs and Interconnects Accessible from Single_East0

Figure 4.7: Sections of Switch Box CIP

Table 4.9: Mapping of the CIPs in Various JBits Classes [Xil01d]

Interconnect Resources Connected	Class	#
Bi-Directional Hex Lines and Single Lines	BiHexToSingle	24
Uni-Directional Hex Lines and Single Lines	UniHexToSingle	32
Single Lines and Single Lines	SingleToSingle	144
Output Muxes and Single Lines	OutMuxToSingle	48

in the Figures 4.8 and 4.9 that would completely test the CIPs for the CIP stuck-open and CIP stuck-closed faults. The CIPs tested in each test configuration are as noted in Figures 4.8(c) and 4.9(c). Thus after four test configurations, the CIPs A, F, H, E, I, J are completely tested for stuck-on and stuck-off faults.

Thus, after four configurations six unique CIPs are completely tested for stuck-at faults. As there is no overlap between the CIPs tested in say configuration 1 and configuration 5, the total number of configurations required to test all 144 switch box CIPs would be

$$\frac{144}{Number_of_Unique_CIPs_Tested_Per_Configuration} \times Number_of_Configurations$$

As it can be seen, this number reduces as CIPs are tested in parallel.

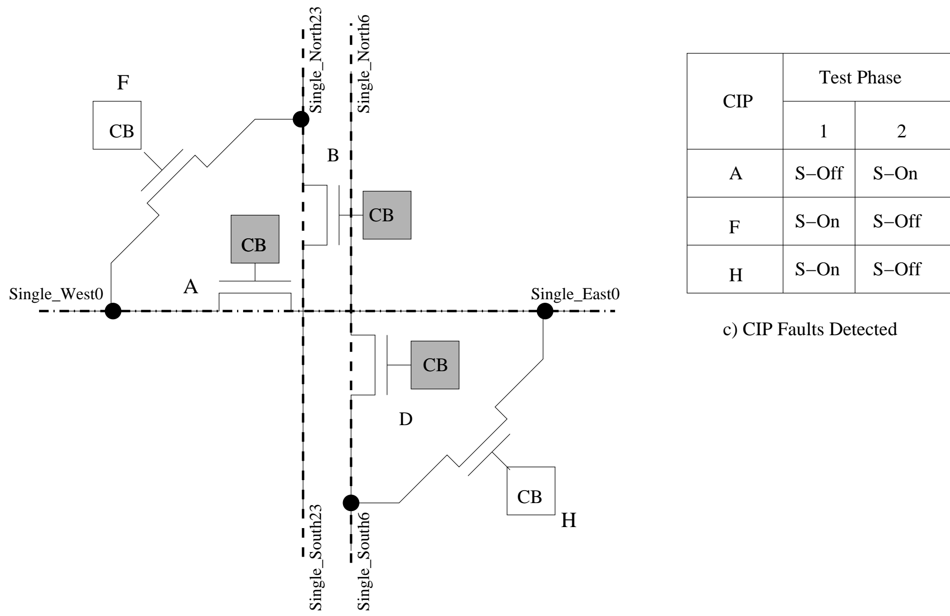
To prove that six unique CIPs are brought under test by a set of four configurations, consider the configuration shown in the Figure 4.8(a) for detecting stuck-open faults in the CIP that connects Single_West0 and Single_East0 interconnect resources. The Single_North23 and Single_South6 cannot be accessed from any other Single_West or Single_East interconnect resource.

Now consider the configuration shown in the Figure 4.8(a) for detecting stuck-open faults in the CIP between Single_West1 and Single_East1 interconnects. The West-North and East-South Single interconnects connected to Single_West1 and Single_East1 interconnects are different from Single_North23 and Single_South6 and cannot be accessed

from any other Single_West or Single_East interconnect resource. The configuration 5 is exactly the same as that of configuration 1 except the CIPs now under test are: CIP between Single_West1-Single_East1 for stuck-off, CIP between Single_West1-Single_North20 for stuck-on, and Single_East1-Single_South3 for stuck-on. Similarly, the configuration 6 is exactly the same as that of configuration 2 except the CIPs now under test are: CIP between Single_West1-Single_East1 for stuck-on, CIP between Single_West1-Single_North20 for stuck-off, and Single_East1-Single_South3 for stuck-off. We observe that these CIPs are different from the ones tested in any of the previous configurations or are tested for different stuck-at faults.

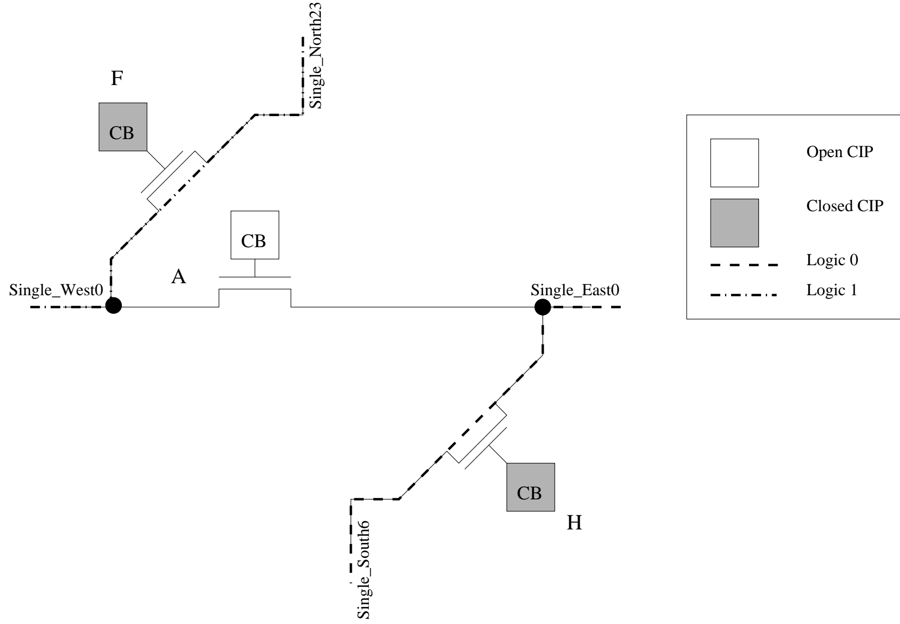
The TPGs and the ORAs along with the routing required to set up test configuration 4 (Figure 4.9(d)) are shown in the Figure 4.10. For the sake of clarity, consider the net WUT0. The net WUT0 is routed through the switch boxes in a zigzag pattern. It comprises CIPs Single_North-Single_South, Single_South wire segments in every switch box in Row 1, Row 2 and Row 0. WUT0 passes through PLBs configured with an identity function (not shown in the figure) to cross the rows. The output multiplexers of the PLBs configured as identity functions, connect WUT0 to Single wire segments across the rows (shown by the broken lines in Row 2 and Row 0). WUT0 makes use of PLB input mux CIPs connecting Single_West wire segments to the input of the ORA. Therefore, none of the CIPs which are under test for stuck-off faults are used for routing.

The number of CIPs that can be tested in a configuration is also affected by the Virtex-I architecture. Consider, there are four outputs from two counter-based TPGs implemented in a PLB. Therefore, each output of the TPG can connect to a maximum of two output muxes. According to Table 4.1, each TPG output connected to two output muxes can drive only three East-West or North-South or East-South or West-North or



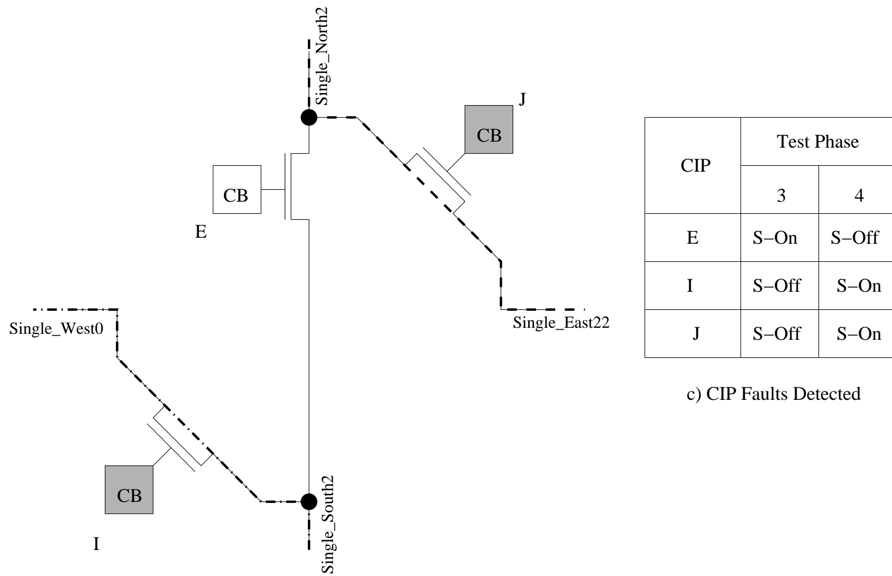
CIP	Test Phase	
	1	2
A	S−Off	S−On
F	S−On	S−Off
H	S−On	S−Off

(a) Test Configuration 1



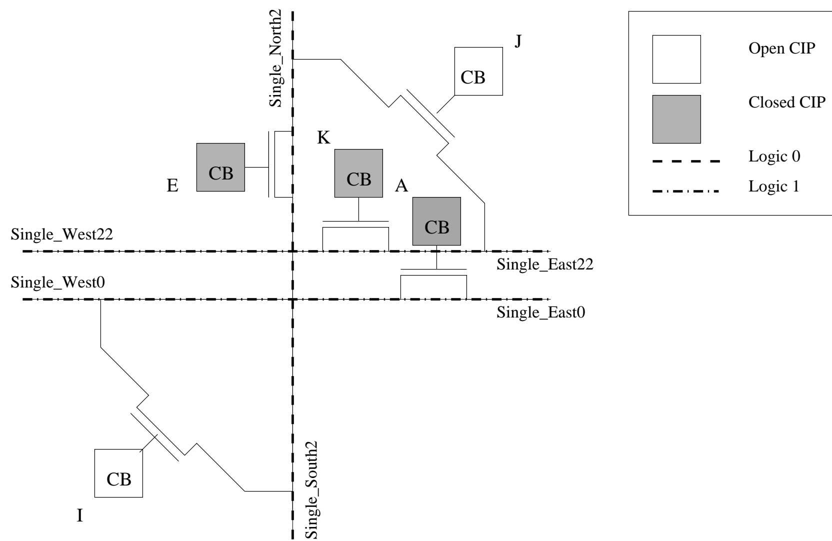
(b) Test Configuration 2

Figure 4.8: Test Configurations Needed to Completely Test Switch Box CIPs



CIP	Test Phase	
	3	4
E	S−On	S−Off
I	S−Off	S−On
J	S−Off	S−On

(c) Test Configuration 3



(d) Test Configuration 4

Figure 4.9: Test Configurations Continued...

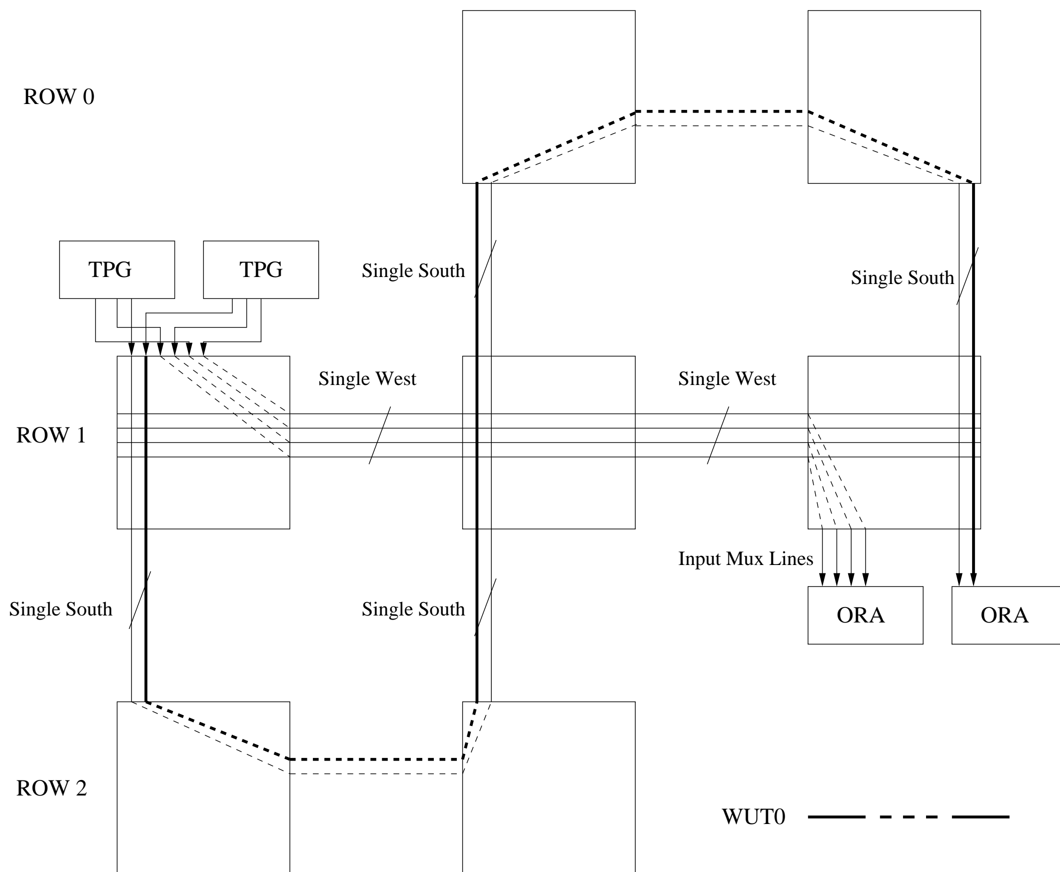


Figure 4.10: Routing BIST Configuration for Testing Mux CIPs

West-South or East-North lines. Thus, six structures shown in the Figure 4.8(b) and Figure 4.9(c) can be tested in parallel by two TPGs. Two comparison-based ORAs are capable of comparing six pairs of inputs and there are a considerable number of input muxes. So ORAs do not pose a problem. However, suppose each TPG output is driving two Single_West lines when testing the group of CIPs in parallel as shown in the Figure 4.8(a). It has to drive four Single_North lines in parallel. As two output muxes may drive only three Single_North lines (Table 4.1), four TPG outputs may test only two structures shown in the Figure 4.8(a) in parallel, as shown in the Figure 4.10. This has an impact on the total number of configurations required to completely test the switch-box CIPs.

There are 24 CIPs of each type: A, F, H, E, I and J. As each TPG output may drive three interconnects, eight configurations of type 2 and type 3 are sufficient to test all 24 CIPs of type A and E for stuck-on and F, I, H, J for stuck-off faults. However, we would need twelve configurations of type 1 and 4 to test all 24 CIPs of type A and E for stuck-off and F, I, H, J for stuck-on faults because of the forementioned limitation on routability of TPG outputs. Therefore, a total of 20 configurations is required to completely test switch-box CIPs for stuck-at faults. If it were not for the peculiar output multiplexers, a Virtex-I PLB containing two slices could generate eight-bit test patterns and two CIPs providing connection between the horizontal Single interconnects could be tested in a configuration. Under these circumstances only half the number of test configurations are needed to fully test the switch box CIP.

4.4.4 MUX CIPs

There are 70 non-decoded MUX CIPs per switch box in the Virtex-I architecture. The MUX CIPs are grouped by the resources they control or the number of inputs as indicated by Table 4.10. The *UniHexMux1* and *UniHexMux2* consist of sixteen 6:1 muxes per PLB that control the connections to the uni-directional Hex lines. *BiHexMux1* and *BiHexMux2* consist of sixteen 4:1 muxes that control the connections to the bi-directional Hex lines. *Mux13to1* consists of eight 13:1 output muxes available to connect outputs of the PLB to the routing resources. *Mux16to1* consists of 16:1 muxes that control the connectivity of BX, BY, CE, Clk and SR inputs of the slice (therefore, ten per PLB) and two tristate buffers in the PLB, thus totaling twelve of them. *Mux28to1* consist of muxes that control the inputs of the LUT. There are sixteen LUT inputs per PLB. The MUX CIPs of type *Mux28to1* and *Mux16to1* provide connectivity to a total of twenty four and sixteen Single interconnects, respectively. The architecture of *Mux28to1* and *Mux16to1* can be represented by the Figure 4.11. As the *Mux28to1* and *Mux16to1* share the Single interconnects, the MUX CIPs belonging to these types can be tested in parallel as shown in the Figure 4.11. The group of MUX CIPs that can be tested in parallel are given in the Table 4.11 and the complete connectivity is listed in Appendix D.

Testing a group of MUX CIPs in parallel requires a total number of configurations equal to the width of the largest MUX CIP [RPFZ99]. However, this approach relies on applying test patterns only through the selected MUX CIP inputs. This method ignores the problem of what is referred to as "invisible logic" in [SKCA96] and thus complete stuck-at fault coverage is not achieved. This condition is depicted in the Figure 4.12.

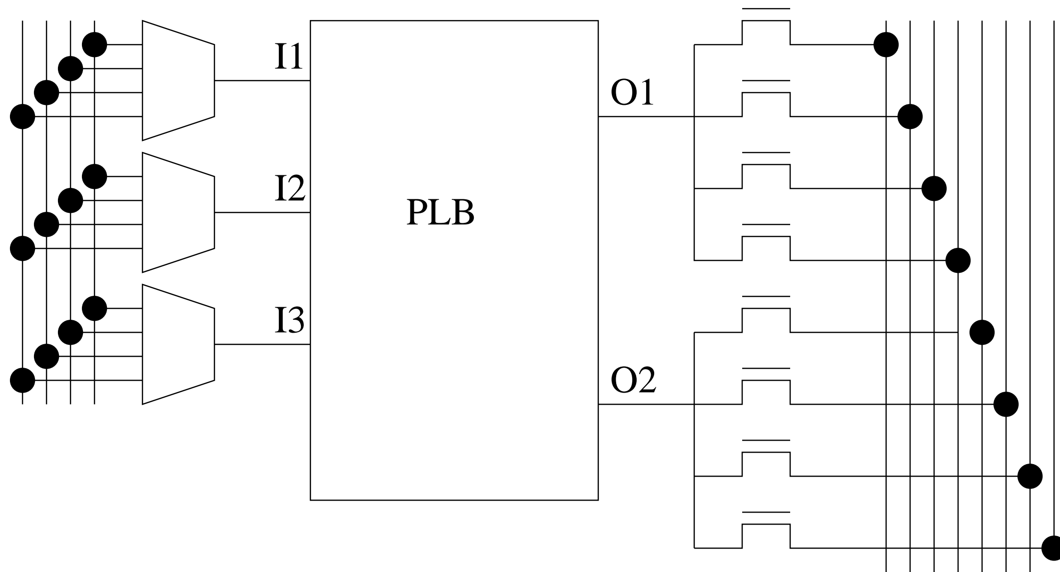


Figure 4.11: Testing MUX CIPs in Parallel [RPFZ99]

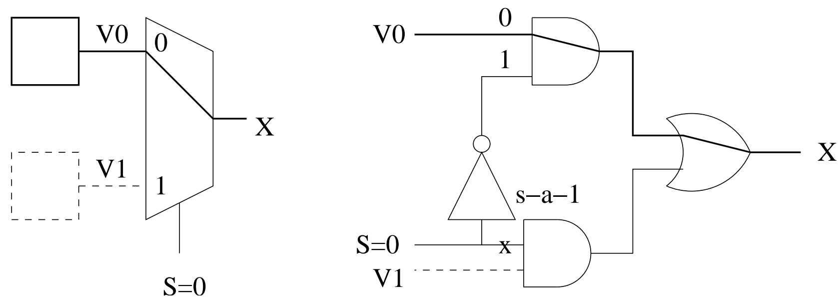


Figure 4.12: Problem of Undetected Faults Due to Invisible Logic in MUX [AS01]

Table 4.10: MUX CIPs in Virtex-I Architecture and their Functions [Xil01d]

Name	Input	MUX CIPs/PLB	Function
UniHexMux1 UniHexMux2	6	16	Used to drive the uni-directional Hex lines
BiHexMux1 BiHexMux2	4	16	Used to drive the bi-directional Hex lines
Mux13to1	13	8	Used to produce the output bus in the CLB
Mux16to1	16	12	Used as the inputs to various control circuits in the CLB
Mux28to1	24	16	Used as the inputs to the LUTs
UniMux8to1	8	2	Used to select the input to the tristate buffers

Table 4.11: MUX CIP Groups Tested in Parallel

Type	MUX CIPs
Mux16to1	S0BX, S0BY S1BX, S1BY S0CE, S1CE S0SR, S1SR S0Clk, S1Clk TS0, TS1
Mux28to1	S0F1, S0G1, S1F4, S1G4 S0F2, S0G2, S1F3, S1G3 S0F3, S0G3, S1F2, S1G2 S0F4, S0G4, S1F1, S1G1

The approach described in [SKCA96] and [AS01] obtains complete stuck-at fault coverage for the MUX CIPs. The invisible logic due to unselected inputs of the MUX CIP is not ignored. The stuck-off faults in the selected inputs are detected by applying both, 0 and 1, to the selected input. At the same time, the unselected logic is configured to apply an opposite logic values to the unselected inputs. In case of non-decoded mulitiplexers, at least one unselected input needs to be configured to apply opposite logic value as shown in Table 4.12.

Table 4.12: Testing MUX CIPs for Stuck-On and Stuck-Off Faults [SWHA98]

Inputs				Configuration Bits			
I0	I1	I2	I3	C3	C2	C1	C0
0	1	0	0	0	0	0	1
1	0	0	0				
1	*0*	0	0	0	0	1	0
0	*1*	0	0				
0	0	*0*	1	0	1	0	0
0	0	*1*	0				
0	0	1	*0*	1	0	0	0
0	0	0	*1*				

If tested in parallel, the four groups of *Mux28to1* with twenty four inputs would take 96 configurations. Complete testing of the five groups of *Mux16to1* with sixteen inputs would require 80 configurations. The 16:1 MUX CIPs controlling the tristate buffers, *UniMux8to1*, would take 16 configurations. *Mux13to1* contain the eight output MUX that would require 13 configurations to cover all the stuck-at faults as they may be tested in parallel. The MUX CIPs belonging to types*BiHexMux1, BiHexMux2, UniHexMux1* and *UniHexMux2* do not share routing resources and therefore cannot be tested together. In order to test BiHexMux1, BiHexMux2 would require a total of 8 configurations and UniHexMux1, UniHexMux2 would require a total of 12 configurations. Thus, a total of 96+80+16+12+13+8=225 configurations is required.

4.5 Generating Configurations for Switch-Box CIPs

The *internalRoute* method is modified to generate the configurations to test the switch box CIPs as described above. The nets such as *WUT0* in the Figure 4.10 have

a source and a destination. The specific connections such as Single_West[0] to Single_East[0] are specified by declaring the corresponding routing resources as the intermediate pins in the *internalRoute* method. The intermediate pins are then routed using automatic routing.

Creating single-point nets with JBits is a non-trivial task. The *set()* call in the class com.xilinx.JBits.Virtex.JBits is utilized. This allows us to create a net without specifying the source pin or the destination pin.

```
/*Header files*/
import com.xilinx.JBits.Virtex.Bits.*;
void internalRoute(int plbrow, int plbcol, int width)
     throws ConfigurationException, RouteException {
jbits.set(plbrow, plbcol,
SingleToSingle.SINGLE_WEST0_TO_SINGLE_EAST0,
SingleToSingle.ON);
jbits.set(plbrow, plbcol,
SingleToSingle.SINGLE_NORTH23_TO_SINGLE_SOUTH23,
SingleToSingle.ON);
jbits.set(plbrow, plbcol,
SingleToSingle.SINGLE_SOUTH6_TO_SINGLE_NORTH6,
SingleToSingle.ON)
}
```

When the TPG and ORA cores are defined in the JBits program and when the router places the design, it automatically includes the CIPs set or reset in the *internalRoute*

method in the appropriate partially placed nets. In this case, these CIPs would become part of the output net of the TPG.

4.6 Conclusion

Generating the logic BIST and interconnect BIST configurations for Virtex-I and Spartan-II requires pursuing different tools from that used while testing its predecessors like XC4000 and Spartan series FPGAs. The previous attempts at targeting interconnect resources of Virtex-I FPGAs using JBits have either ignored the problem of testing all the interconnect resources and switch boxes [FH03] or have incorrectly addressed it [SMG01]. The current implementation of the BIST RTPCores generates six BIST configurations required to test all of the Hex resources, twenty-four BIST configurations required to test all the Single resources for the wire stuck-at faults and the associated CIPs for CIP stuck-open fault. The RTPCores can be extended for testing the switch-box CIPs for CIP stuck-closed faults. The Virtex-I FPGA can be directly configured with the BIST configuration bitstreams. The bitstreams targeting Spartan-II and Virtex-E FPGAs may be compiled from the XDL files after modifying the headers that reflect the target chip. The mechanism provided in the Virtex-I routing architecture for connecting two interconnects or transporting signals in and out of the PLB is either using the MUX CIPs or the switch box CIPs. Complete testing of switch box CIPs requires 20 configurations. Testing the MUX CIPs requires one configuration per input to guarantee that invisible logic is also tested for stuck-on faults. Therefore, MUX CIPs alone would require 224 test configurations. To test the interconnect resources for wire stuck-at, open wire faults and pair-wise shorts, requires another 36 configurations.

Chapter 5

Summery and Future Work

The reduction in the size of the memory required to store the partial logic BIST configurations as well as routing BIST configurations was discussed. Smaller size BIST configurations leads to faster configuration times. The advantage of partial configuration memory readback over full configuration memory readback is predicted. Partial configuration memory readback may still lag behind the integrated ORA scan-chain method of retrieving the ORA Pass/Fail results. However, there are no additional configurations required and there are no additional logic and routing resources needed for partial configuration memory readback on Virtex-I and Spartan-II series FPGAs. A program was developed to automatically generate routing BIST configuration bitstreams using JBits API for Virtex-I FPGAs. The program outputs XDL files enumerating connections between the logic and interconnect resources for the routing BIST configurations. The XDL files may be further processed to generate partial routing BIST configurations for Spartan-II FPGAs. The program implements BIST RTPCores to generate the interconnect BIST configurations using JBits API. The interconnect BIST configuration bitstreams test Single and Hex interconnects for wire stuck-at faults and associated CIPs for stuck-open faults. The WUT pairs falling at the boundary of adjacent WUT groups e.g. WUT pairs indexed (3,4), (7,8) etc., are not tested for bridging faults. A set of four test sessions was identified to completely test switch box CIPs for stuck-at faults in the Virtex-I and Spartan-II architecture. A method was proposed to define single point nets in the JBits program. This method would be useful while algorithmically generating test

configurations that sensitize stuck-on faults in the multiplexer CIPs, break-point CIPs and cross-point CIPs.

The FPGA architecture has an impact on the number of logic and interconnect BIST configurations required to achieve 100% stuck-at fault coverage. The Virtex-I PLB contains eight output muxes to connect twelve PLB signals to the routing resources, thereby preventing us from routing signals to two comparison-based ORAs from two slices in the PLB configured as BUTs, at the same time in the logic BIST test phase. This reduces the number of outputs from the BUTs that can be compared by an ORA. As only one slice can be tested in a test phase, the total number of logic BIST configurations required increases.

To develop interconnect and logic BIST configurations for FPGAs using CAD tools poses difficulties because the conventional CAD tools are written with the designer in mind. Therefore, test engineers resort to finding new ways to use the same tool that is available to the designers. The XDL intermediate representation has been successfully demonstrated in the past as a useful tool to generate interconnect and logic test configurations for XC4000 and Spartan-I FPGAs. The attractiveness of this approach lies in the simplicity and efficiency, especially for the logic and interconnect BIST for FPGAs where the structure shows high regularity. XDL represents the BIST circuitry in human readable form, converts easily into Xilinx netlist format so that it may be viewed in the graphical viewers such as FPGA Editor, and reliably generates bitstreams for the target chip with tools such as BitGen. The disadvantage of this approach, as felt by the author, is the maintainability and rigidness of the code that generates the XDL, which is actually inherent among with the programs manipulating the textual files. Even this difficulty is dwarfed as the logic and interconnect BIST structure is very regular e.g.,

in routing BIST, iterating XDL circuit description for H-STAR or V-STAR testing one row or column over the entire PLB array, generates XDL circuit description for H-STAR or V-STAR instantiated over entire PLB array. The more serious problem is increasing difficulty in managing the symbolic names in XDL. The reporting facility offered by XDL enlists all the symbolic names of the CIPs and the logic resources. The report files typically run in megabytes even for a small size Virtex-I FPGA. The symbolic names of the logic resources and CIPs are manageable once they are put into a database or a spread-sheet.

The designers and test engineers frequently need to use the utilities such as design rule checks, delay calculations and checking the veracity of the design specified in higher level languages, as offered by FPGA Editor. Therefore, FPGA Editor is an essential tool while developing the logic and interconnect test configurations. The FPGA Editor currently accepts design input only in Xilinx NCD format, which is interconvertible with XDL. It is felt that XDL support should be a criterion while evaluating CAD tools for generating logic and interconnect BIST for Xilinx FPGAs. The JBits API can be effectively utilized if it is viewed as a tool for specifying designs in high level languages and generating XDL. However, the new version of JBits API for Virtex-II does not have built-in support for XDL like its previous versions. This is not to say that one cannot be built. The XDL can be viewed simply as another output format supported by the JBits application along with bitstream. The FPGA place and route tools like VPR have been modified to take advantage of JBits API to generate design in XDL format [BR97] [XRT03]. JBits API for Virtex-I is capable of enumerating the circuit specified in higher level languages in XDL format. The functionality of the new version

of JBits API needs to be evaluated as to how it may be used to generate the output of the design specified in high level languages in XDL format.

Routing capabilities of the CAD tool is another important aspect to consider, while evaluating the tool for generating logic and interconnect BIST. While generating interconnect and routing BIST configurations for an FPGA, the test engineer must maintain control over the routing. The primary consideration for a CAD tool would be if it would allow the test engineer to specify the exact routing resources to use while routing source and sink. For the test engineer to achieve 100% stuck-at fault coverage of an FPGA, the CAD tool must also allow creation of single-point nets. This facility is hard to find among conventional design entry and place-and-route tools, but it might be useful for the test engineers. Therefore, the routing capabilities of the new version of JBits API for Virtex-II should be probed for the satisfaction of above mentioned requirements.

Bibliography

[AES01] M. Abramovici, J. E. Emmert, and C. Stroud. Roving STARs: An Integrated Approach to On-Line Testing, Diagnosis and Fault Tolerance for FPGAs in Adaptive Computing Systems. In *Proceedings of Third NASA/DoD Workshop on Evolvable Hardware*, pages 73–92, July 2001.

[AKS93] V. D. Agrawal, C. R. Kime, and K. K. Saluja. A Tutorial on Built-In Self Test-Part 1 Principles. In *IEEE Design and Test of Computers*, 10(1):73–82, March 1993.

[AR94] M. J. Alexander and G. Robins. High-performance Routing for Field Programmable Gate Arrays. In *Proceedings of International ASIC Conference and Exhibit*, pages 138–141, September 1994.

[AS01] M. Abramovici and C. Stroud. BIST-Based Test and Diagnosis of FPGA Logic Blocks. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(1):159–172, February 2001.

[BFRV92] S. Brown, R. Francis, J. Rose, and Z. Vranesic. *Field-Programmable Gate Arrays.* Kluwer Academic Publishers, 1992.

[BR96] S. Brown and J. Rose. FPGA and CPLD Architectures: A Tutorial. In *IEEE Design & Test of Computers*, 13(2):42–57, Summer 1996.

[BR97] V. Betz and J. Rose. VPR: A New Packing, Placement, and Routing Tool for FPGA Research. In *International Workshop on Field-Programmable Logic and Applications*, pages 213–222, September 1997.

[CHW00] T. J. Callahan, J. R. Hauser, and J. Wawrzynek. The Garp Architecture and C Compiler. In *IEEE Computer*, 33(4):62–69, April 2000.

[DeH00] A. DeHon. The Density Advantage of Configurable Computing. In *IEEE Computer*, 33(4):41–49, April 2000.

[DW99] A. DeHon and J. Wawrzynek. Reconfigurable Computing: What, Why, and Implications for Design Automation. In *Proceedings of IEEE Design Automation Conference*, pages 610–615, June 1999.

[ESSA00] J. Emmert, C. Stroud, B. Skaggs, and M. Abramovici. Dynamic Fault Tolerance in FPGAs via Partial Reconfiguration. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 165–174, April 2000.

[FH03] D. A. Fernandes and I. G. Harris. Application of Built in Self-Test for Interconnect Testing of FPGAs. In *Proceedings of IEEE International Test Conference*, pages 1248–1257, October 2003.

[GL99] S. Guccione and D. Levi. Run-time parameterizable cores. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 252–259, February 1999.

[GLS99] S. Guccione, D. Levi, and P. Sundararajan. JBits: Java Based Interface for Reconfigurable Computing. In *Proceedings of Military and Aerospace Applications of Programmable Logic Devices (MAPLD) International Conference*, September 1999.

[GSB+00] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor. Piperench: A Reconfigurable Architecture and Compiler. *In IEEE Computer*, 33(4):70–77, April 2000.

[HGWS99] C. Hamilton, G. Gibson, S. Wijesuriya, and C. Stroud. Enhanced BIST-Based Diagnosis of FPGA via Boundary Scan Access. In *Proceedings of IEEE VLSI Test Symposium*, pages 413–418, April 1999.

[HLS98] S. Hauck, Z. Li, and E. Schwabe. Configuration Compression for Xilinx 6200 FPGA. In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 138–146, April 1998.

[IEE90] Test Access Port and Boundary Scan Architecture IEEE Standard P1149.1-1990, February 1990.

[Jav04] The Java(tm) Tutorial. URL: http://java.sun.com/docs/books/tutorial/ java/ concepts/ index.html, 2004.

[Jay01] R. Jayaraman. Physical Design for FPGAs. In *Proceedings of ACM International Symposium on Physical Design*, pages 214–221, April 2001.

[Kel00] E. Keller. JRoute: A Run-Time Routing API for FPGA Hardware. In *Parallel and Distributed Processing*, pages 874–881, May 2000.

[Lat02] Lattice Semiconductor. *ORCA Series 3 FPGAs Programmable I/O Cell (PIC): Logic, Clocking, Routing, and External Device Interface, (AP99-042FPGA)*, January 2002.

[LG98] D. Levi and S. A. Guccione. BoardScope: A Debug Tool for Reconfigurable Systems. In *Proceedings of SPIE Configurable Computing: Technology and Applications*, pages 239–246, November 1998.

[MG00] S. McMillan and S. A. Guccione. Partial Run-Time Reconfiguration Using JRTR. In *Proceedings of the 10th International Workshop on Field-Programmable Logic and Applications*, pages 352–360, August 2000.

[Mil94] W. Miller. Real World Applications for Field Programmable Gate Array Devices-An Overview. In *Proceedings of IEEE West Coast Electronics Show and Convention (WESCON)*, pages 27–29, September 1994.

[RFZ97] M. Renovell, J Figueras, and Y. Zorian. Test of RAM-Based fpga: Methodology and Application to the Interconnect. In *Proceedings of IEEE VLSI Test Symposium*, pages 230–237, April 1997.

[RPFZ99] M. Renovell, J. M. Portal, J. Figueras, and Y. Zorian. Testing the Configurable Interconnect/Logic interface of SRAM-based FPGA's. In *Proceedings of IEEE International Conference on Design Automation and Test in Europe*, pages 618–622, March 1999.

[RZ00] M. Renovell and Y. Zorian. Different Experiments in Test Generation for Xilinx FPGAs. In *Proceedings of IEEE International Test Conference*, pages 854–862, October 2000.

[SCKA96] C. Stroud, P. Chen, S. Konala, and M. Abramovici. Evaluation of FPGA Resources for Built-In Self-Test of Programmable Logic Blocks. In *Proceedings of ACM International Symposium on Field-Programmable Gate Arrays*, pages 107–113, February 1996.

[SHGS04] C. Stroud, J. Harris, S. Garimella, and J. Sunwoo. Built-In Self-Test Configurations for Atmel FPGAs Using Macro Generation Language. In *Proceedings of IEEE North Atlantic Test Workshop*, pages 88–90, May 2004.

[SKCA96] C. Stroud, S. Konala, P. Chen, and M. Abramovici. Built-In Self-Test for Programmable Logic Blocks in FPGAs(Finally, A Free Lunch: BIST Without Overhead!). In *Proceedings of IEEE VLSI Test Symposium*, pages 387–392, April 1996.

[SKCA97] C. Stroud, S. Konala, P. Chen, and M. Abramovici. BIST-based Diagnostics for FPGA Logic Blocks. In *Proceedings of IEEE International Test Conference*, pages 539–547, October 1997.

[SLS03] C. Stroud, K. N. Leach, and T. A. Slaughter. BIST for Xilinx 4000 and Spartan Series FPGAs: A Case Study. In *Proceedings of IEEE International Test Conference*, pages 1258–1267, October 2003.

[SMG01] P. Sundararajan, S. McMillan, and S. A. Guccione. Testing FPGA Devices Using JBits. In *Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, September 2001.

[SNLA02] C. Stroud, J. Nall, M. Lashinsky, and M. Abramovici. BIST-Based Diagnosis of FPGA Interconnect. In *Proceedings of IEEE International Test Conference*, pages 618–627, October 2002.

[SS99] A. Steininger and C. Scherrer. On the Necessity of On-line BIST in Safety-Critical Applications. In *Proceedings of Fault-Tolerant Computing Symposium*, pages 208–215, June 1999.

[SSGH04] C. Stroud, J. Sunwoo, S. Garimella, and J. Harris. Built-In Self-Test for System-on-Chip: A Case Study. In *Proceedings of IEEE International Test Conference*, October 2004.

[Str02] C. Stroud. *A Designer's Guide to Built-In Self-Test*. Kluwer Academic Publishers, 2002.

[SWHA98] C. Stroud, S. Wijesuriya, C. Hamilton, and M. Abramovici. Built-In Self-Test of FPGA Interconnect. In *Proceedings of IEEE International Test Conference*, pages 404–411, October 1998.

[SXCT00] X. Sun, J. Xu, B. Chan, and P. Trouborst. Novel Technique for Built-In Self-Test of FPGA Interconnects. In *Proceedings of IEEE International Test Conference*, pages 795–803, October 2000.

[TM03] M. B. Tahoori and S. Mitra. Automatic Configuration Generation of FPGA Interconnect Testing. In *Proceedings of IEEE VLSI Test Symposium*, pages 134–139, April 2003.

[Xil] Xilinx Inc. *Libraries Guide*. ISE 6.1i.

[Xil99] Xilinx Inc. *XC4000E and XC4000X Series Field Programmable Gate Arrays Product Specification, (v1.6)*, May 1999.

[Xil00] Xilinx Inc. *Xilinx Design Language Reference (v1.6)*, July 2000.

[Xil01a] Xilinx Inc. *JBits Tutorial*, June 2001.

[Xil01b] Xilinx Inc. *Virtex 2.5 V Field Programmable Gate Arrays, (DS003-1 v2.5)*, April 2001.

[Xil01c] Xilinx Inc. *Virtex Device Simulator (VirtexDS)*, June 2001.

[Xil01d] Xilinx Inc. *Xilinx JBits SDK Version 2.8 for Virtex*, June 2001.

[Xil02a] Xilinx Inc. *Configuration and Readback of Spartan-II FPGAs Using Boundary Scan, (XAPP188 v2.1)*, March 2002.

[Xil02b] Xilinx Inc. *Configuration and Readback of Virtex FPGAs Using (JTAG) Boundary Scan, (XAPP139 v1.4)*, April 2002.

[Xil02c] Xilinx Inc. *Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations, XAPP290 (v1.0)*, May 2002.

[Xil02d] Xilinx Inc. *Virtex FPGA Series Configuration and Readback, (XAPP138 v2.7)*, July 2002.

[Xil02e] Xilinx Inc. *Virtex-II Pro Platform FPGAs: Introduction and Overview, (DS031-1 v1.9)*, September 2002.

[Xil03a] Xilinx Inc. *Spartan-3 1.2V FPGA Family:Complete Data Sheet, (DS099 —)*, October 2003.

[Xil03b] Xilinx Inc. *Spartan-II 2.5V FPGA Family: Complete Data Sheet, (DS001-1 v2.3)*, September 2003.

[Xil03c] Xilinx Inc. *Virtex-II Platform FPGAs: Introduction and Overview, (DS031-1 v2.0)*, August 2003.

[Xil03d] Xilinx Inc. *Virtex Series Configuration Architecture Users Guide, (XAPP151 v1.6)*, March 2003.

[Xil04] Xilinx Inc. *Virtex-II Platform FPGA User Guide, (UG002 v1.7)*, February 2004.

[XRT03] W. Xu, R. Ramanarayanan, and R. Tessier. Adaptive Fault Recovery for Networked Reconfigurable Systems. In *Field-Programmable Custom Computing Machines*, pages 143–152, April 2003.

Appendices

Appendix A

Steps in Writing Parent RTPCore

1. Create a class that inherits its functionality from class RTPCore in the package com.xilinx.JBits.CoreTemplate,

2. Define the methods to calculate the width and height of the RTPCore in terms of PLBs and/or slices. The methods are declared *static*, therefore the object does not have to be created in order to determine the width and height of the RTPCore. The following methods help while placing the RTPCore in the PLB array:

   - *calcHeight()*: This method, when called, returns the height of the RTPCore in the multiple of the granularity set by *calcHeightGran()* method. The granularity can be a Logic Element (such as flip-flop, LUT), a slice or a PLB.
   - *calcWidth()*: This method, when called, returns the width of the RTPCore in the multiple of the granularity set by *calcWidthGran()*. The granularity can be a Logic Element (such as flip-flop, LUT), a slice or a PLB.
   - *calcHeightGran()*: Sets the granularity or the unit of height of the RTPCore in terms of Logic Element (such as flip-flop, LUT), slice or PLB.
   - *calcWidthGran()*: Sets the granularity or the unit of width of the RTPCore in terms of Logic Element (such as flip-flop, LUT), slice or PLB.

3. Initialize the data members inherited from the parent class and this class. This is achieved by calling following methods in the parent class RTPCore:

   - *setHeight()*

- *setWidth()*
- *setHeightGran()*
- *setWidthGran()*

4. Define the *implement()* method. The *implement()* method of the parent RTPCore has the following steps:

   (a) Define the logical nets and buses required to connect the child RTPCore. The methods *newNet(NetName)* or *newBus(BusName, width)* provided in the class com.xilinx.JBits.CoreTemplate.RTPCore define the logical nets and buses, respectively. The nets and buses thus defined, are passed as parameters to the constructors of the child RTPCore class. It is the responsibility of the *implement()* method of the child RTPCore classes to define the mapping between these logical nets and buses defined at the upper level of hierarchy to the physical resources of FPGA.

   (b) Instantiate the child RTPCore by calling the constructor of its class and pass the logical nets and buses as parameters.

   (c) Call the *addChild(Child_Core)* method from parent class RTPCore in the package com.xilinx.CoreTemplate. This call establishes parent-child relationship.

   (d) Specify the placement of the child RTPCore in the PLB array. This is accomplished either by passing a static integer defined in class Offset from package com.xilinx.JBits.CoreTemplate.RTPCore to the *addChild()* method or assigning absolute offsets to the child RTPCore before calling the *implement()* method of the child RTPCore class.

(e) Call the *implement()* method of the child RTPCore class.

(f) Specify routing between the child RTPCores if necessary.

(g) Call the static method *connect(Bus)* in the class Bitstream belonging to package com.xilinx.JBits.CoreTemplate.RTPCore. The logical buses are mapped into physical wires with the call to this method. This call can only be made once the RTPCore is implemented i.e. complete routing and placement of the RTPCore have been specified. Unless this call is successful, the logical buses specified in the JBits program will not map into the physical wires and will not reflect in the XDL file generated at the output of the JBits program.

The most important function of the *implement()* method in the class SimpleRouteBIST is to configure the routes between the TPGCounterCore and ORACore. This is done using following steps:

1. Create instance of class TPGCounterCore by calling the constructor. The constructor expects object of class CounterProperties. Therefore, we assume this object is created and all the logical buses are appropriately assigned before the constructor is called. Assume the object is identified by *instance_of_class_TPGCounterCore*.

2. Call the *addChild(string, instance_of_class_TPGCounterCore)* method inherited from the parent class com.xilinx.JBits.CoreTemplate.RTPCore. Thus parent core-child core hierarchy is established.

```
public final void implement() throws CoreException,
                ConfigurationException, RouteException {
      /* Set App vars */
      int WIDTH = 4;

      /* Create Top level net and bus */
      Net clk = newNet("clk");
      Bus coutSrc = newBus("coutSrc",WIDTH/2);
      Bus coutSrc1 = newBus("coutSrc1",WIDTH/2);

      Clock clock = new Clock("clock", clk);
      clock.implement(1); // Use GCLK1

      TPGProperties cp = new TPGProperties();
      cp.setIn_clk(clk);
      cp.setIn_ce(Net.NoConnect);
      cp.setIn_rst(Net.NoConnect);
      cp.setOut_dout(coutSrc);

      /* Create Counter Core */
      TPGCounter tpg = new TPGCounter("counter",cp);
      addChild(tpg);
```

Figure A.1: JBits Program for Instantiating a Counter core

3. Obtain an object of class com.xilinx.JBits.CoreTemplate.RTPCore.Offset using *getRelativeOffset()* method of *instance_of_class_TPGCounterCore*. Let's say the object is identified by *offset_of_TPGCounterCore*. Set the horizontal and vertical offsets according to arguments i and j, using methods *setHorOffset()* and *setVerOffset()* respectively of the object *offset_of_TPGCounterCore*.

4. Call the *implement()* method of the *instance_of_class_TPGCounterCore*.

This process is depicted in the Figure A.1 and A.2.

```
        Offset CntrOffset = tpg.getRelativeOffset();
        CntrOffset.setHorOffset(Gran.SLICE, clbCol*2+clbSlice);
        CntrOffset.setVerOffset(Gran.CLB, clbRow);

        tpg.implement();

        TPGProperties cp1 = new TPGProperties();
        cp1.setIn_clk(clk);
        cp1.setIn_ce(Net.NoConnect);
        cp1.setIn_rst(Net.NoConnect);
        cp1.setOut_dout(coutSrc1);

        /* Create Second Counter Core */
        TPGCounter tpg1 = new TPGCounter("counter1",cp1);
        addChild(tpg1);

        Offset CntrOffset1 = tpg1.getRelativeOffset();
        CntrOffset1.setHorOffset(Gran.SLICE, clbCol*2+clbSlice+1);
        CntrOffset1.setVerOffset(Gran.CLB, clbRow);

        tpg1.implement();

        /* Connect Top level nets */
        Bitstream.connect(clk);
        Bitstream.connect(coutSrc);
        Bitstream.connect(coutSrc1);
}
```

Figure A.2: JBits Program Continued...

Appendix B

Steps in Writing Child RTPCores

1. Create a class that inherits its functionality from class RTPCore. Define input and output ports of the RTPCore using methods *newInputPort (Port_Name, Signal_Type)* and *newOutputPort (Port_Name, Signal_Type).*

2. Steps 2 and 3 are the same as that of the parent RTPCore.

3. The *implement()* method of a child RTPCore has following steps:

   (a) Define a logical internal signal for every input and output port.

   (b) Assign the logical internal signal to the corresponding port using *setIntSig (Signal_Type)* method defined in com.xilinx.JBits.CoreTemplate.RTPCore.Port class.

   (c) Define a logical external signal for every input and output port.

   (d) Initialize the logical external signals by *getExtSig(Signal_Type)* method defined in the class com.xilinx.JBits.CoreTemplate.RTPCore.Port. At this point the input or output port defined by the child RTPCore is completely initialized. All the changes made to the internal signal can be guaranteed to reflect on the external signal.

   (e) Write the user code using the logical internal signals.

   (f) If necessary, use the *setPin(Pin_Type)* method defined in the class Port defined in package com.xilinx.JBit.CoreTemplate.RTPCore to explicitly set a particular PLB resource to the port.

(g) Call static method *Bitstream.connect(Signal_Type)* for the internal signals. If this call is successful then the logical internal signal and the routing would be reflected in the XDL files generated at the output of the application. If this call is unsuccessful, the router throws an exception condition and bails out of the JBits program indicating it is unable to route the particular signal.

Appendix C

Complete Program Source

The TPGCounterCore is exactly same as that of two-bit counter found in the class Counter found in the package com.xilinx.JBits.Virtex.RTPCore.Basic. The source code is provided with the JBits software. The source code for the ORACore is derived from the source code of the class LUT5 found in the package com.xilinx.JBits.Virtex.RTPCore.Basic. We cannot publish the modified source code in this thesis because the original source code is copyrighted by Xilinx. The primary purpose of the class LUT5 is to implement combinational logic function with five variables. The input and output ports defined to the LUT5 are as listed in Table C.1. The four nets comprising of the four variables of the combinational logic function are connected to both, *addr0* and *addr1* input ports. The net representing the fifth variable of the combinational logic function is connected to *muxsel* input net of the core. The two LUTs in a slice are configured with the logic expression given to the constructor of the class.

Table C.1: Input and Output ports of LUT5

Port	Width (in Bits)	Direction	Function
clk	1	IN	Clock
addr0	4	IN	Input Net to G-LUT
addr1	4	IN	Input Net to F-LUT
muxsel	1	IN	Input Net to Select the Output of F-LUT or G-LUT
dout	1	OUT	Unregistered Output
dout_reg	1	OUT	registered Output
ce	1	IN	Enable/Disable

In order to derive the functionality of comparison-based ORA, we completely disabled the mux select input, prevented the connection between the output of the F5 mux

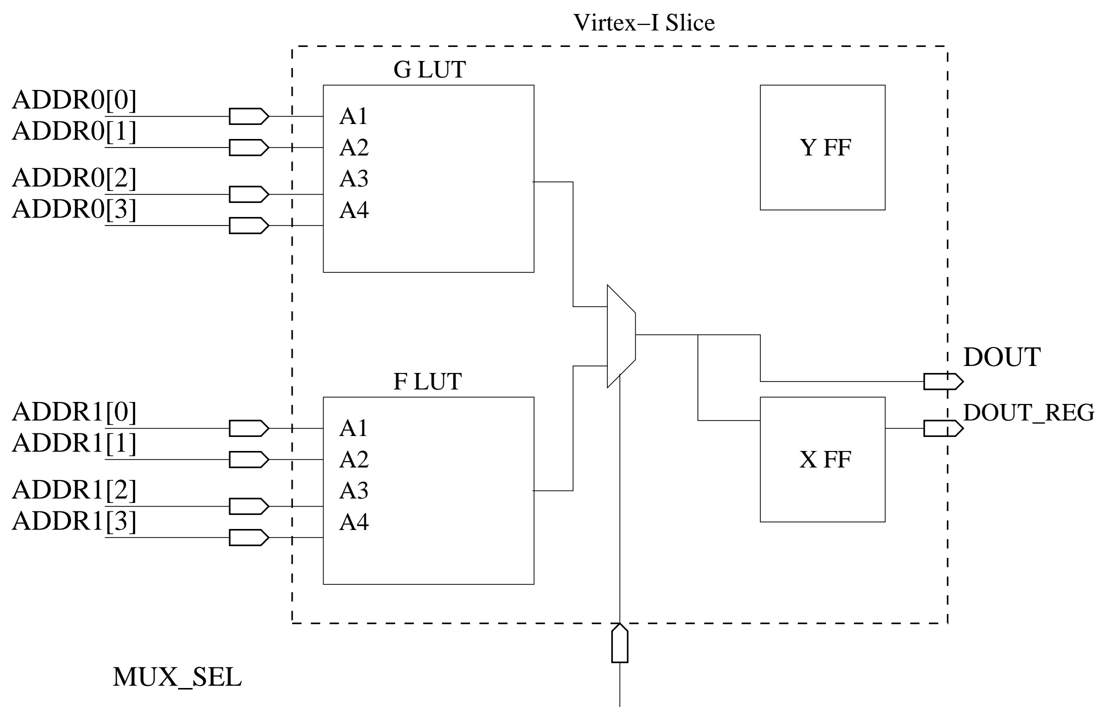


Figure C.1: Configuration of LUT5 RTPCore

and X flip-flop and connected output of G LUT to output Y of the slice as shown in the Figure 4.3. This enabled us to feedback the output of the G LUT to the input F LUT. The actual nets connected to the *addr0* and *addr1* port of the ORACore, are defined by the parent RTPCore, SimpleRouteBIST.

```
/*
 * SimpleRouteBISTApp.java
 *
 * Created on March 5, 2004, 2:16 AM
 */

/**
 *
 * @author  newalaa
 */
/* Simple application for experimenting with routing BIST and JBits JRoute2 API..
 * essentially acts as a wrapper to instantiation of SimpleRouteBIST core.
 *
 */

package RouteBIST;

import com.xilinx.util.JBitsCommandLineApp;
import java.io.*;
import com.xilinx.JBits.CoreTemplate.*;
import com.xilinx.JBits.Virtex.Devices;
import com.xilinx.JRoute2.Virtex.ResourceDB.CenterWires;
import com.xilinx.JBits.Virtex.JBits;
import com.xilinx.Netlist.SYM.*;
import com.xilinx.Netlist.XDL.*;
import com.xilinx.JRoute2.Virtex.JRoute;
import com.xilinx.JRoute2.Virtex.RouteException;
import com.xilinx.JBits.Virtex.ConfigurationException;

public class SimpleRouteBISTApp extends JBitsCommandLineApp {
   /** Creates a new instance of SimpleRouteBIST */
   public SimpleRouteBISTApp() {   }
```

Figure C.2: JBits Program for User Interaction and Populating the PLB Array

```
public void run() {
   try {
      jbits = getJBits();
      jroute = new JRoute(jbits);
      Bitstream.setVirtex(jbits, jroute);
      CoreOutput.generateBitstream(true);
      CoreOutput.generateSYM(true);
      CoreOutput.generateXDL(true);

      /* Generate Outputs */
      FileOutputStream fOut=new FileOutputStream("simpleRBIST.ctf");
      ObjectOutput out = new ObjectOutputStream(fOut);

      int row = 0;
      int col = 0;
      int MAX_COL = Devices.getClbColumns(com.xilinx.JBits.Virtex.Devices.XCV50);
      int MAX_ROW = Devices.getClbRows(com.xilinx.JBits.Virtex.Devices.XCV50);
      int slice = 0;
      System.out.println(" The maximum columns "+MAX_COL);
      System.out.println(" The maximum rows "+MAX_ROW);

      for (int i=0; i<MAX_ROW;i++) {
         for (int j=0; j<MAX_COL;j++) {
            if (j%6==0) {//j has reached where there is new ORA
                j+=6;
            }
            SimpleRouteBIST simpleBIST =
                          new SimpleRouteBIST("simpleBIST"+i+"_"+j,i,j,slice);
            /* Define orgin */
            Offset offset = simpleBIST.getRelativeOffset();
            offset.setVerOffset(Gran.CLB,0);
            offset.setHorOffset(Gran.SLICE,0);
            /* Implement this design */

            simpleBIST.implement();
            out.writeObject(simpleBIST);
         }
      }
      CoreOutput.writeXDLFile("simpleRBIST.xdl");
      CoreOutput.writeSymFile("simpleRBIST.sym");
      out.flush();
      out.close();

   }
```


Figure C.3: JBits Program Continued...

```
        } catch (Exception e) {
        e.printStackTrace();
        System.exit(-1);
        }

   } /* end main() */

   protected JRoute jroute;
   protected JBits jbits;
      public String Speed;
  public String Package;
}
```

Figure C.4: JBits Program Continued...

```
/*
 * ORAProperties.java
 *
 * Created on September 14, 2004, 7:24 PM
 */

package RouteBIST;
import com.xilinx.JBits.CoreTemplate.*;
/**
 *
 * @author  Administrator
 */
public class ORAProperties extends CoreParameters {

/* Inputs. */
private Net clk = null;
private Bus addr0 = null;
private Bus addr1 = null;
private Net lutOut = null;
private Net ce = null;

/* Outputs */
private Net dout = null;
private Net doutReg = null;

/*
*************************************************
* Accessor Methods
*************************************************
*/

/* Inputs. */
public final Net getIn_clk() {
  return clk;
}
```

Figure C.5: JBits Program Continued...

```
public final void setIn_clk(Net net) {
  clk = net;
}

public final Bus getIn_addr0() {
  return addr0;
}

public final void setIn_addr0(Bus bus) {
  addr0 = bus;
}

public final Bus getIn_addr1() {
  return addr1;
}

public final void setIn_addr1(Bus bus) {
  addr1 = bus;
}

public final Net getIn_ce() {
  return ce;
}

public final void setIn_ce(Net net) {
  ce = net;
}
/**********************************************************/
/**********************************************************/

/* Outputs. */
public final Net getOut_dout() {
  return dout;
}

public final void setOut_dout(Net net) {
  dout = net;
}

public final Net getOut_doutReg() {
  return doutReg;
}
```

Figure C.6: JBits Program Continued...

```
public final void setOut_doutReg(Net net) {
  doutReg = net;
}

public final void setOut_lutOut(Net net) {
   lutOut = net;
}

public final Net getOut_lutOut() {
   return lutOut;
}

/**
**  This method checks to make sure that
**  the parameters that were set are valid
**  for the specified core.
**
**  @param core  The RTPCore these parameters will be used with.
**
**  @exception com.xilinx.JBits.CoreTemplate.CoreParameterException
*/
public void checkParameters(RTPCore core) throws CoreParameterException {

  String ret = "";

  if (clk == null) {
    ret = ret.concat(": No clock connection specified");
  }

  if (addr0 == null) {
    ret = ret.concat(": No addr0 connection specified");
  }

  if (addr1 == null) {
    ret = ret.concat(": No addr1 connection specified");
  }

  if (ce == null) {
    ret = ret.concat(": No ce connection specified");
  }

  if (dout == null) {
    ret = ret.concat(": No dout connection specified");
  }
```


Figure C.7: JBits Program Continued...

```
  if (doutReg == null) {
    ret = ret.concat(": No doutReg connection specified");
  }

  if (addr0.getWidth() != 4) {
    ret = ret.concat(": addr0 length equal to " + addr0.getWidth() +
      ", should be equal to 4");
  }

  if (addr1.getWidth() != 2) {
    ret = ret.concat(": addr1 length equal to " + addr1.getWidth() +
      ", should be equal to 4");
  }

  if (!ret.equals("")) {
    throw new CoreParameterException(core, ret);
  }


} /* end checkParameters() */

}
```

Figure C.8: JBits Program Continued...

```
/*
 * SimpleRouteBIST.java
 *
 * Created on March 14, 2004, 10:30 PM
 */

/**
 *
 * @author  newalaa
 */
/**
  2 slices are configured with 2–bit counter cores, wires are connected using manual route,
  ORA core for pass/fail results.  For the time being results are observed through BoardScope...
 *
   Tests four Hex Lines ––
   CenterWires.Hex_Horiz_East[0],
 *  CenterWires.Hex_Horiz_East[1],
   CenterWires.Hex_Horiz_East[2],
 *  CenterWires.Hex_Horiz_East[3]   Tested with indices 0,5,7,9 also
 *
 *  Special Note:  This version uses TPG that occupies 2 slices.  The XQ, YQ outputs
 *  of the slices are used to drive the exhaustive test patterns on the wires.  The outputs are
 *  then compared with the ORA.

 *
 *  The route specified in this program (method internalroute() ) has been observed in
 *  FPGA Editor.
 *
 */
package RouteBIST;
```

Figure C.9: JBits Program Continued...

```
import com.xilinx.JBits.CoreTemplate.*;

import com.xilinx.JRoute2.Virtex.JRoute;
import com.xilinx.JBits.Virtex.JBits;
import com.xilinx.JRoute2.Virtex.ResourceDB.CenterWires;
import com.xilinx.JRoute2.Virtex.ResourceFactory;
import com.xilinx.JRoute2.Virtex.Segment;
import com.xilinx.JRoute2.Virtex.JBitsConnector;
import com.xilinx.Netlist.XDL.*;
import com.xilinx.Netlist.XDL.XDLexception;
import  com.xilinx.JBits.Virtex.ConfigurationException;
import  com.xilinx.JRoute2.Virtex.RouteException;
import com.xilinx.JBits.Virtex.RTPCore.Basic.Clock;
import com.xilinx.JBits.Virtex.RTPCore.Basic.Counter;
import com.xilinx.JBits.Virtex.RTPCore.Basic.CounterProperties;
import com.xilinx.JBits.Virtex.RTPCore.Basic.LUT5;
import com.xilinx.JBits.Virtex.RTPCore.Basic.LUT5Properties;
import com.xilinx.JBits.Virtex.RTPCore.TestGeneration.TestInputVector;
import com.xilinx.JBits.Virtex.RTPCore.Basic.Register;

import com.xilinx.JBits.Virtex.Expr;
import com.xilinx.JBits.Virtex.Util;
import com.xilinx.JBits.Virtex.Bits.*;
import com.xilinx.DeviceSimulator.Virtex.RouteTracer;
import com.xilinx.DeviceSimulator.Virtex.RouteTree;
import com.xilinx.DeviceSimulator.Virtex.SimulationException;
```

Figure C.10: JBits Program Continued...

```
  /** Creates a new instance of SimpleRouteBIST */
public SimpleRouteBIST(String name, int row, int col, int slice)
  throws CoreException {

   super(name);
     // Set up a print stream object.

   clbRow = row;
   clbCol = col;
   clbSlice = slice;
   setHeight(calcHeight());
   setWidth(calcWidth());
   setHeightGran(calcHeightGran());
   setWidthGran(calcWidthGran());
}/* end constructor */
/*
****************************************************
* Required Methods
****************************************************
*/

/**
 * Return the vertical granularity of this core.
 */
public static int calcHeightGran() {
  return Gran.CLB;

  } /* end getHeightGran() */

/**
 * Return the horizontal granularity of this core.
 */
public static int calcWidthGran() {
  return Gran.SLICE;

  } /* end getWidthGran() */
```

Figure C.11: JBits Program Continued...

```
/*
***************************************************
* Convention Methods
***************************************************
*/

/**
 * Calculates height for this core.  This method is
 * static so an object does not have to be created
 * in order to figure out the height of this object
 * with the given properties.
 *
 *
 */
public static int calcHeight() {

  return 1;

  } /* end calcHeight() */

/**
 * Calculates width for this core.  This method is
 * static so an object does not have to be created
 * in order to figure out the width of this object
 * with the given properties.
 *
 */
public static int calcWidth() {

  return 2;

  } /* end calcWidth() */
```

Figure C.12: JBits Program Continued...

```
private void internalRoute(int w, int row, int col)
                throws RouteException, ConfigurationException{
  int i;
  CoutSrcPin = new Pin[w];
  OraPin = new Pin[2*w];
  WutPin = new Pin[w];
  OutMuxPin = new Pin[w];
  int slice = 0;
  JRoute jroute = Bitstream.getVirtexRouter();
  for (i=0;i+3<w;i++) {
    CoutSrcPin[0] = new Pin(Pin.CLB, row, col, CenterWires.Slice_XQ[slice]);
    CoutSrcPin[1] = new Pin(Pin.CLB, row, col, CenterWires.Slice_XQ[slice+1]);
    CoutSrcPin[2] = new Pin(Pin.CLB, row, col, CenterWires.Slice_YQ[slice]);
    CoutSrcPin[3] = new Pin(Pin.CLB, row, col, CenterWires.Slice_YQ[slice+1]);

    OraPin[0] = new Pin(Pin.CLB, row, col+6, CenterWires.SliceG1[slice]);
    OraPin[1] = new Pin(Pin.CLB, row, col+6, CenterWires.SliceG2[slice]);
    OraPin[2] = new Pin(Pin.CLB, row, col+6, CenterWires.SliceG3[slice]);
    OraPin[3] = new Pin(Pin.CLB, row, col+6, CenterWires.SliceG4[slice]);


    WutPin[0] = new Pin(Pin.CLB, row, col, CenterWires.Hex_Horiz_East[0]);
    WutPin[1] = new Pin(Pin.CLB, row, col, CenterWires.Hex_Horiz_East[1]);
    WutPin[2] = new Pin(Pin.CLB, row, col, CenterWires.Hex_Horiz_East[2]);
    WutPin[3] = new Pin(Pin.CLB, row, col, CenterWires.Hex_Horiz_East[3]);
  }
  for (i=0;i<w;i++) {
    System.out.println(" Now Routing ...."+CoutSrcPin[i]+"... and "+WutPin[i]);
    jroute.route(CoutSrcPin[i],WutPin[i]);
    jroute.route(WutPin[i], OraPin[i]);
    //jroute.route(WutPin[i], OraPin[i+4]);
  }
```

Figure C.13: JBits Program Continued...

```
}
/**
 *  This method instantiates the core.
 */
public final void implemen()
            throws CoreException, ConfigurationException,
            RouteException {

    //java.io.PrintStream ps = System.out;
    //ResourceFactory rf = ResourceFactory.getResourceFactory(Bitstream.getVirtex() );

    /* Set App vars */
    int WIDTH = 4;
    int CE_WIDTH = 1;
    int DEPTH = 16;
    /* Create Top level net and bus */
    Net clk = newNet("clk");
    Bus coutSrc = newBus("coutSrc",WIDTH/2);
    Bus ffd = newBus("ffd", WIDTH/2);
    Bus coutSrc1 = newBus("coutSrc1",WIDTH/2);
    Bus ffd1 = newBus("ffd1", WIDTH/2);
    Bus oraSink = newBus("oraSink",WIDTH);


    Clock clock = new Clock("clock", clk);
    clock.implement(1); // Use GCLK1

    TPGProperties cp = new TPGProperties();
    cp.setIn_clk(clk);
    cp.setIn_ce(Net.NoConnect);
    cp.setIn_rst(Net.NoConnect);
    cp.setOut_dout(coutSrc);
    cp.setOut_ffd(ffd);
```

Figure C.14: JBits Program Continued...

```
tpg.implement();

TPGProperties cp1 = new TPGProperties();
cp1.setIn_clk(clk);
cp1.setIn_ce(Net.NoConnect);
cp1.setIn_rst(Net.NoConnect);
cp1.setOut_dout(coutSrc1);
cp1.setOut_ffd(ffd1);


/* Create Second Counter Core */
TPGCounter tpg1 = new TPGCounter("counter1",cp1);
addChild(tpg1);
Offset CntrOffset1 = tpg1.getRelativeOffset();

CntrOffset1.setHorOffset(Gran.SLICE, clbCol*2+clbSlice+1);
CntrOffset1.setVerOffset(Gran.CLB, clbRow);

tpg1.implement();

/*
 for (int i=0;i<WIDTH;i++) {
    if (i<WIDTH/2){
       oraSink.setNet(i, ffd.getNet(i));
    }
    if (i>=WIDTH/2){
       oraSink.setNet(i, ffd1.getNet(i-(WIDTH/2)));
    }

}
*/
```

Figure C.15: JBits Program Continued...

```
    //Bus OraSink = newBus("oraSink",WIDTH);
    Net passFailOut = newNet("passFailOut");        //Pass/Fail output from X flip−flop
    Net lutOut = newNet("lutOut");      //Output of the G LUT
    Net passFailIn = newNet("passFailIn");          //Logical input of X flip−flop
    Bus oraFB = newBus("OraFB",2);
    oraFB.setNet(0, passFailOut);
    oraFB.setNet(1, lutOut);
    ORAProperties op = new ORAProperties();
    op.setIn_addr0(oraSink);        //addr port of G LUT
    op.setIn_addr1(oraFB);          //addr port of F LUT
    op.setOut_lutOut(lutOut);      //Y output of Slice
    op.setIn_clk(clk);
    op.setIn_ce(Net.NoConnect);
    op.setOut_dout(passFailIn);
    op.setOut_doutReg(passFailOut);

    ORACore BISTORACore= new ORACore("ORA00",op);
    addChild(BISTORACore);

    Offset ORAOffset = BISTORACore.getRelativeOffset();

    ORAOffset.setHorOffset(Gran.SLICE, (clbCol+6)*2+clbSlice);
    ORAOffset.setVerOffset(Gran.CLB, clbRow);
    BISTORACore.implement(Expr.G_LUT("(G1 ^ G2) | (G3 ^ G4)"),
                                       Expr.F_LUT("~( ((~F1)&F2) | (F1&(~F2)) )"));
    internalRoute(WIDTH, clbRow, clbCol);
    /* Connect Top level nets */
    Bitstream.connect(clk);
    Bitstream.connect(coutSrc);
    Bitstream.connect(ffd);
    Bitstream.connect(coutSrc1);
    Bitstream.connect(ffd1);
    Bitstream.connect(oraFB);
} /* end implement() */
  private int clbRow, clbCol, clbSlice;

  private Pin [] CoutSrcPin;
  private Pin [] WutPin;
  private Pin [] OraPin;
  private Pin [] OutMuxPin;
};  /* end class SimpleRouteBIST*/
```

Figure C.16: JBits Program Continued...

Appendix D

Complete List of Connections Between the Mux CIPs

Table D.1: Mux CIPs Mux28to1 and Connecting Single Interconnects

Wires	Number of Distinct Wires Connected			
	S0F1, S0G1, S1F4, S1G4	S0F3, S0G3, S1F2, S1G2	S0F4, S0G4, S1F1, S1G1	S0F2, S0G2, S1F3, S1G3
SINGLE_EAST	4	7	7	6
SINGLE_WEST	7	6	5	6
SINGLE_SOUTH	6	5	6	7
SINGLE_NORTH	7	6	6	5

Table D.2: MUX CIPs Mux16to1 and Connecting Interconnects

Wires	Number of Distinct Wires Connected					
	S0BX, S0BY	S0CE, S1CE	S1BX, S1BY	S0SR, S1SR	S0Clk, S1Clk	TS0, TS1
HEX_HORIZ_EAST	0	0	0	0	0	0
HEX_HORIZ_WEST	0	0	0	0	0	0
HEX_VERT_NORTH	0	1	0	1	1	0
HEX_VERT_SOUTH	0	0	0	0	0	0
SINGLE_EAST	4	2	4	2	1	0
SINGLE_WEST	4	2	4	2	1	0
SINGLE_SOUTH	4	3	4	3	2	0
SINGLE_NORTH	4	3	4	3	2	0
HEX_VERT_A0	0	0	0	0	0	1
HEX_VERT_B0	0	0	0	0	0	1
HEX_VERT_C0	0	0	0	0	0	1
HEX_VERT_D0	0	0	0	0	0	1
HEX_VERT_M0	0	0	0	0	0	1
HEX_VERT_A1	0	0	0	1	0	0
HEX_VERT_B1	0	0	0	1	0	0
HEX_VERT_C1	0	0	0	1	0	0
HEX_VERT_D1	0	0	0	1	0	0
HEX_VERT_M1	0	0	0	1	0	0
HEX_VERT_A2	0	0	0	0	1	0
HEX_VERT_B2	0	0	0	0	1	0
HEX_VERT_C2	0	0	0	0	1	0
HEX_VERT_D2	0	0	0	0	1	0
HEX_VERT_M2	0	0	0	0	1	0
HEX_VERT_A3	0	1	0	0	0	0
HEX_VERT_B3	0	1	0	0	0	0
HEX_VERT_C3	0	1	0	0	0	0
HEX_VERT_D3	0	1	0	0	0	0
HEX_VERT_M3	0	1	0	0	0	0
GCLK0	0	0	0	0	1	0
GCLK1	0	0	0	0	1	0
GCLK2	0	0	0	0	1	0
GCLK3	0	0	0	0	1	0