Built-In Self Test of Configurable Memory Resources in 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.

Daniel Milton

Certificate of Approval:

Victor P. Nelson
Professor
Electrical and Computer Engineering

Charles E. Stroud, Chair
Professor
Electrical and Computer Engineering

Thaddeus A. Roppel
Associate Professor
Electrical and Computer Engineering

George T. Flowers
Interim Dean
Graduate School

Built-In Self Test of Configurable Memory Resources in Field Programmable Gate Arrays

Daniel Milton

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
December 17, 2007

Built-In Self Test of Configurable Memory Resources in Field Programmable Gate Arrays

Daniel Milton

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 of Graduation

Vita

Daniel Milton, son of Thomas and Diane Milton, was born in Birmingham, Alabama on April 25, 1983. In the Fall of 2005, he graduated Summa cum Laude with a Bachelor of Electrical Engineering majoring in Computer Engineering. Upon graduation he immediately began working on his Master of Science degree at Auburn University under the advisement of Dr. Charles E. Stroud.

Thesis Abstract

Built-In Self Test of Configurable Memory Resources in Field Programmable Gate Arrays

Daniel Milton

Master of Science, December 17, 2007
(B.E.E., Auburn University, 2005)

151 Typed Pages

Directed by Charles E. Stroud

Testing embedded memory resources in Field Programmable Gate Arrays (FPGAs) is difficult because the collective signal fan-in and fan-out is much greater than the available external I/O. A testing approach is needed that can test all of the memory resources in parallel without out being limited to external I/O. Built-in Self Test (BIST) is a testing method that incorporates test circuitry around the devices under test (DUT). The programmable nature of FPGAs allows the BIST circuitry to have no performance and size overhead because the BIST circuitry can be downloaded to the FPGA while the system is offline. Once offline, resources inside the FPGA can be tested and the results retrieved. If the FPGA is found to be fault-free then the system function can be downloaded again and brought back online.

BIST for embedded memory resources in Virtex 4 FPGAs is developed and test configurations are generated for all Virtex 4 devices. Twenty-five total BIST configurations are developed to test memories operating in RAM, FIFO, ECC, and cascade modes. To test each operating mode, a hardware design language (HDL) based test pattern generator

(TPG) is developed and then incorporated into an algorithmically placed BIST template that contains two TPGs, DUTs, and output response analyzers (ORAs) to observe DUT outputs. Partial reconfiguration is used to reduce both configuration bitstream storage and test time. A total speed-up factor of 12 is observed when utilizing partial reconfiguration.

Acknowledgments

I would like to thank Dr. Stroud for his support and advise during my tenure at Auburn University during both my undergraduate and graduate studies. I would also like to thank Dr. Nelson and Dr. Roppel for their contribution to this thesis by serving on my graduate committee. To my research colleagues, Sachin, Sudheer, Bobby, Lee, Mustafa, Brad, David, and Noah, I am grateful for all of your help and assistance throughout my research. Lastly, I would like to acknowledge my parents, as their ever present support has always inspired my to fulfill my potential.

Style manual or journal used Journal of Approximation Theory (together with the style known as "aums"). Bibliography follows IEEE Transactions.

Computer software used The document preparation package TEX (specifically LATEX) together with the departmental style-file `aums.sty`. Plots were generated using *Microsoft Excel* and figures were drawn in *Microsoft Visio*.

Table of Contents

List of Figures xi

List of Tables xiv

1 Introduction 1
1.1 Built-In Self Test (BIST) 2
1.2 FPGAs 3
1.3 Embedded Memory Resources in FPGAs 5
1.4 Thesis Statement 6

2 Background 8
2.1 Introduction to FPGAs 8
2.2 Virtex 4 Architecture 12
2.2.1 Virtex 4 PLBs 14
2.2.2 Virtex 4 BRAMs 15
2.2.3 Virtex 4 FIFOs 22
2.2.4 Virtex 4 CAD Tools 27
2.2.5 Virtex 4 Boundary Scan 27
2.3 SRAM Testing 30
2.3.1 SRAM Fault Models 31
2.3.2 March Tests for Single-Port Memories 32
2.3.3 March Tests for Dual-Port Memories 35
2.4 Overview of BIST for FPGAs 37
2.4.1 BIST for BRAMs 40
2.5 Thesis Restatement 42

3 Virtex 4 Block RAM BIST Implementation 44
3.1 Virtex 4 BRAM BIST Architecture 44
3.2 TPG Development 48
3.3 BRAM BIST Configurations 51
3.4 Running BIST Configurations 61
3.5 ORA Results Retrieval 62
3.6 BIST Results 63
3.7 BRAM BIST Summary 68

4 Virtex 4 FIFO BIST Implementation 69
4.1 Virtex 4 FIFO BIST Architecture 69
4.2 FIFO TPG Development 69
4.3 FIFO BIST Configuration Development 74
4.4 Running FIFO BIST Configurations 77
4.5 FIFO BIST Results 78

5 Virtex 4 ECC and Cascade BIST Implementation 82
5.1 ECC and Cascade BIST Architecture 82
5.2 ECC BRAM BIST Development 85
5.3 Cascade TPG Development 89
5.4 BIST Configurations 91
5.5 Running BIST Configurations 91
5.6 BIST Results 93

6 Summary and Conclusion 98
6.1 Summary of Virtex 4 BIST Results 98
6.2 Application to Virtex 5 100

Bibliography 101

Appendices 104

A MMTPG VHDL Source code 105

B FIFOTPG VHDL Source code 123

C ECCTPG VHDL Source code 127

D CASTPG VHDL Source code 131

E List of Acronyms 135

List of Figures

1.1 General BIST Architecture 3

1.2 FPGA Architecture 4

1.3 Basic PLB Architecture 4

2.1 Configuration Bits in FPGAs 10

2.2 A Five Transistor Configuration SRAM Cell [1] 11

2.3 A Six Transistor Configuration SRAM Cell [1] 11

2.4 Basic Virtex 4 Architecture 14

2.5 Virtex 4 PLB [2] 15

2.6 Simplified Virtex 4 Slice Diagram [2] 16

2.7 Virtex 4 BRAM [2] 18

2.8 ECC BRAM Architecture [2] 22

2.9 BRAM Cascade Operational Diagram [2] 23

2.10 Virtex 4 FIFO Implementation [2] 25

2.11 Virtex 4 FIFO [2] 25

2.12 TAP Controller State Diagram [3] 29

2.13 SRAM Memory Functional Model 30

2.14 Structural Model of a two-port SRAM cell 31

2.15 March LR with 4-bit BDS [4] 35

2.16 March s2pf [5] 36

2.17 March d2pf [5] 37
2.18 A General Comparison Based BIST Architecture 38
2.19 A Circular Comparison Based BIST Architecture 39
2.20 Comparison Based ORA with Shift Chain 40
2.21 Comparison Based ORA without a Shift Chain 40
3.1 BRAM BIST Architecture 45
3.2 BRAM ORA Orientation 46
3.3 ORA Placement and Comparison in SX devices 47
3.4 ORA Placement and Comparison in FX devices 48
3.5 MMTPG Control Shift Register 51
3.6 FX12 BRAM BIST 55
3.7 LX25 BRAM BIST 56
3.8 V4BRAMTPG Syntax 59
3.9 V4BRAMBIST Syntax 59
3.10 V4BRAMMOD Syntax 59
3.11 Example BIST program execution 59
3.12 Partial BRAM BIST in LX60 60
3.13 LX60 BRAM BIST Speed-up factors 64
3.14 Timing Analysis (Slowest / Fastest) 66
3.15 Timing Analysis per BRAM BIST Configuration 67
4.1 FIFO ORA Placement 70
4.2 FULL Flag Transition Timing 73

4.3 FIFOTPG Control Register 74

4.4 LX60 FIFO BIST Configuration 75

4.5 FIFO BIST Timing Analysis 79

4.6 LX60 FIFO Speed-up Factors 81

4.7 Routed FX12 FIFO BIST Configuration 81

5.1 ECC and Cascade BIST Architecture 84

5.2 Expected Cascade ORA Failure Locations 85

5.3 Parity Tree TPG [6] 87

5.4 Cascade BRAM Operational Diagram 90

5.5 FX12 ECC BIST 92

5.6 FX12 Cascade BIST 92

5.7 LX60 ECC BIST Speed-up Factors 95

5.8 LX60 CAS BIST Speed-up Factors 95

5.9 ECC BIST Timing Analysis 96

5.10 Cascade BIST Timing Analysis 97

6.1 BIST Speed-up for LX60 99

List of Tables

2.1 Overview of Resources in Virtex 4 Family Devices [2] 13

2.2 Summary of Virtex 2 and 4 BRAM Aspect Ratios 17

2.3 BRAM Signal Descriptions [2] 19

2.4 BRAM Configuration Options [2] 20

2.5 ECC Status Description 22

2.6 FIFO Configuration Options [2] 24

2.7 Virtex 4 Status Flag Clock Cycle Latency [2] 26

2.8 FIFO Port Signal Descriptions [2] 26

2.9 Summary of Xilinx Design Tools 28

2.10 Virtex 4 BSCAN Module Access Commands [3] 29

2.11 Common SRAM Fault Types 32

2.12 March Test Notation Descriptions 33

2.13 Common March Tests for Single Port Memories 34

2.14 Background Data Sequence for 8-bits 35

2.15 Virtex 2 BRAM Summary 41

3.1 Virtex 4 TPG March Test Algorithms 50

3.2 BRAM BIST Configuration Detail 52

3.3 BRAM Initialization Values 53

3.4 XDL Argument Summary 58

3.5 BRAM BIST Execution Detail 62

3.6 Summary of LX60 BRAM BIST Download Size and Test Times 65

4.1 Summary of Virtex 4 FIFO Configurations 76

4.2 Summary of LX60 FIFO BIST Download Size and Test Times 80

5.1 ECC BRAM BIST Configuration Settings 88

5.2 Summary of Cascade BIST Configuration Settings 91

5.3 Summary of LX60 CAS BIST Download Size and Test Times 94

5.4 Summary of LX60 ECC BIST Download Size and Test Times 94

Chapter 1

Introduction

Moore's Law states that the complexity of Integrated Circuits (ICs) tends to double every 24 months [7]. While this empirical observation was first observed in 1965, more recently, the International Technology Roadmap for Semiconductors has predicted Moore's Law will persist at least until 2016 based upon industry data and forecasts [8]. One of the earliest microprocessors, the Intel 4004, had approximately 2300 transistors [9]. For contrast, recent state-of-the-art Field Programmable Gate Arrays (FPGAs) may contain more than one billion transistors [10]. Clearly, one can see the exponential growth in transistor count over the last four decades. The continuing problem with these higher density ICs is that developing tests for such complex devices is becoming progressively more difficult with each generation of new ICs. Recent IC fabrication technology has led to larger chip sizes with smaller feature sizes, but also has introduced new types of defects and subsequently increased the probability of defects [6].

Developing good tests for ICs is becoming a major factor in the cost of producing working silicon ICs. One of the reasons for high costs is the disparity between the number of Input/Output (I/O) pins in packaged ICs verses the number of transistors in the package. Using external 'bed-of-nails' test equipment, the costs of testing are generally attributed to the fixed cost of the test equipment and the speed at which the actual tests can be performed. IC manufactures generally employ tests that minimize the cost of the test equipment while minimizing the device test time [11]. Improvements in Design for Test (DFT) methodology have produced scan design and Built-in Self Test (BIST) [6][11].

1.1 Built-In Self Test (BIST)

DFT is a common practice in the VLSI design process. Traditionally, DFT for ICs has involved using scan flip-flops. Scan flip-flops reduce the amount of time needed to generate tests for sequential circuits because they can operate as a shift register to shift in a test vector from an external Automated Test Equipment (ATE). When shifting a new test vector into a scan chain, the output response from the previous vector is shifted out, which is then can be compared to an expected value. Scan design eliminates the possibility of being unable to initialize a flip-flop to a desired value. However, for many large VLSI circuits, the number of test vectors that must be applied to achieve a necessary fault coverage percentage has also increased. Coupled with a growing amount of test vectors and ATE not being able to test at speed for newer ICs, a different approach for applying test vectors was introduced [11].

BIST is a DFT technique that allows the Device Under Test (DUT) or a portion of the DUT to tell the tester if it is fully functional. BIST implementations include a Test Pattern Generator (TPG) which drives a DUT or many DUTs in parallel. The outputs of the DUTs are then analyzed by an Output Response Analyzer (ORA) which determines the correctness of the DUT. The general BIST architecture can be seen in Figure 1.1. BIST solves two of the major issues with ATE based testing. First, the BIST circuitry is implemented in the chip itself, and therefore it can perform at speed. Secondly, since the BIST circuitry generates test vectors, the external tester merely needs to tell the device to perform BIST and then report whether the device is faulty or not [6].

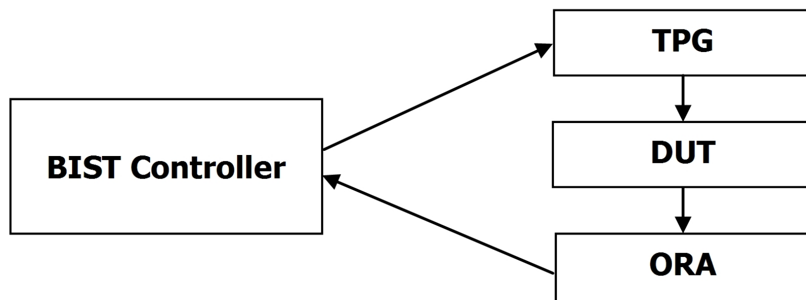

Figure 1.1: General BIST Architecture

Traditionally, BIST has been used to test logic and memory resources in VLSI circuits [6]. One caveat of BIST is that it usually implies an overhead in terms of increased chip area which may in turn reduce the yield of the chip [6]. One may wonder if it is possible to implement BIST with no overhead. BIST implementations proposed in [12][13][14][15][16][17][18][19][20][21][22] suggest BIST for FPGAs can incur no overhead penalty in terms of speed and area.

1.2 FPGAs

An FPGA can be described as "an array of logic blocks that can be programmably interconnected to realize different designs" [23]. A general FPGA architecture, as seen in Figure 1.2, contains I/O cells that facilitate signals entering and exiting the device. Programmable Logic Blocks (PLBs) perform the necessary digital logic functions and the programmable routing resources direct signals both between PLBs and the overall signal path from inputs to outputs.

PLBs vary among manufactures, but a simple PLB can contain Look-up Tables (LUTs), flip-flops/latches, and multiplexers. A simple PLB architecture is illustrated in Figure

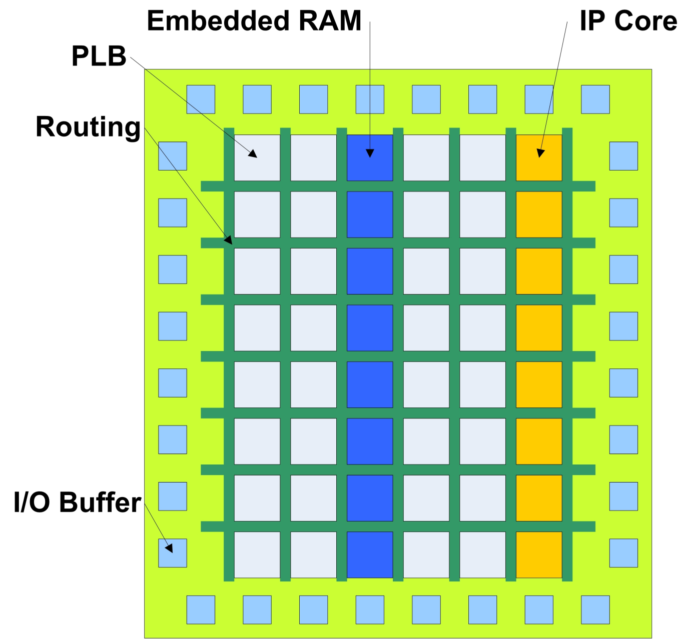

Figure 1.2: FPGA Architecture

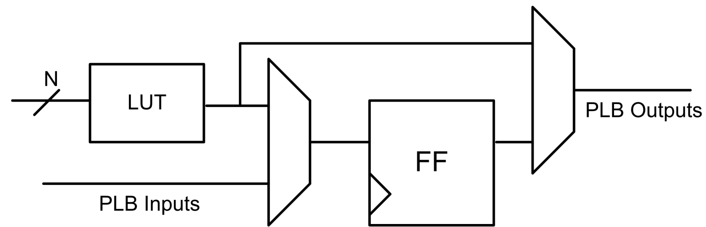

Figure 1.3: Basic PLB Architecture

1.3. LUTs can be configured as the truth table for a given logical function and they can implement small distributed random access memories (RAMs). The interconnection or routing between PLBs is realized by configuring Programmable Interconnect Points (PIPs) to create signal paths from wire segments inside the FPGA [23].

FPGA programming techniques vary among manufacturers, and include static RAM (SRAM), fuse/anti-ifuse, and floating gate methods. SRAM is the most popular programming technique for advanced FPGAs because SRAM based designs allow high density and fast configuration [23]. SRAM based FPGAs contain a configuration memory that, when written to, specifies the operation of PLBS, I/O cells, and routing resources. Besides logic blocks and routing resources, the configuration memory may configure other embedded resources in the FPGA such as embedded RAMs, multipliers, and digital signal processors (DSP). The inclusion of additional embedded resources allows for higher PLB utilization because the embedded resources can offload much of the functionality of digital systems.

1.3 Embedded Memory Resources in FPGAs

FPGA manufacturers have incorporated embedded memories for many product generations [10][24]. Previously, storing large quantities of data internally required an appreciable amount of PLB resources to implement a memory resource. While most designers frequently utilize the memory resources as regular RAM modules, several FPGAs now allow system designers to configure memory resources as multi-port RAMs and First-In-First-Out (FIFO) modules. The advent of dedicated memory resources extends the potential of FPGAs to act as a programmable System-on-Chip (SoC). However, as advantageous as memory resources

are, there are testability concerns that must be addressed. Traditionally, testing RAMs requires applying test patterns that read and write data in such an order that within a set of faults being tested, any such fault will be sensitized if present. These testing algorithms are generally known as march tests. There have been many march tests developed for detecting certain types of faults within memory resources [25][26]. Unfortunately, applying march tests to embedded memories is complicated. ATE is generally used to verify production ICs. However, ATEs usually provide test patterns to the external I/O pins of an IC. A better testing approach is needed because most FPGAs contain many memory resources whose collective fan-in and fan-out are much greater than the available I/O pins [10][24]. BIST is an ideal solution for testing embedded resources in FPGAs. A BIST approach offers more flexibility for testing than an external ATE because test pattern generation is not limited by the external I/O availability. Previous implementations have shown that this approach is quite feasible and efficient for testing embedded memories [27][17][28].

1.4 Thesis Statement

The goal of this thesis is to develop a BIST architecture and BIST configurations for testing embedded memories in Xilinx Virtex 4 FPGAs. This BIST architecture will address minimizing both the test time required and the memory required to store the BIST configurations. This minimization is achieved by designing the BIST architecture to efficiently utilize an FPGA configuration technique known as partial reconfiguration. The remainder of this thesis is organized as follows: In Chapter 2, additional background information on BIST for FPGAs will be given along with details toward testing SRAMs in general and a detailed overview of the Virtex 4 FPGA architecture. In Chapter 3, a

BIST architecture for Virtex 4 embedded memories will be presented with applications to memory resources configured to operate in a basic RAM mode of operation. Chapters 4 and 5 will apply the presented BIST architecture to memory resources configured to operate as FIFOs and Error-Correcting Code (ECC) RAMs, respectively. Chapter 6 will summarize the work presented in this thesis and include ideas for future research in this field.

Chapter 2

Background

This chapter presents an overview of different BIST techniques for FPGAs found in the literature. The architecture of the Xilinx Virtex 4 FPGA will also be presented with an emphasis on the dedicated memory resources referred to as block RAMs (BRAMs). Background on SRAM testing will be presented that predominately focuses on march tests and the associated fault models for which they are designed.

2.1 Introduction to FPGAs

FPGAs differ from Application Specific Integrated Circuits (ASICs) because they are made of logic and routing resources capable of implementing most digital systems while not being explicitly fabricated for a specific task. For example, one might buy a microprocessor, an ASIC, as part of a digital system while another designer might use an FPGA to implement a microprocessor and other supporting functions all in the same IC. Furthermore, another designer might use the exact same FPGA in a DSP application. Clearly, the flexibility of FPGAs is its main advantage. Another advantage is the reduced non-recurring engineering costs by eliminating the need to design and fabricate custom ASICs. The main disadvantage, however, of FPGAs is higher chip area, higher power consumption, and lower operational speeds as compared to a custom ASIC which is due to extra programming circuitry overhead. Also, FPGAs typically are more expensive than traditional ASICs so they are usually relegated to low volume or prototype designs [23].

In order for an FPGA to realize a digital system, it must be programmed [23]. Several programming technologies exist, but SRAM based FPGAs are currently the most common and popular. Other older programming technologies exist such as fuse or anti-fuse based and mask-programmable FPGAs. However, these programming technologies only allow for one-time programmability, either at the factory in the case of mask-programmed FPGAs or in the field as is the case with fuse or anti-fuse based FPGAs. While SRAM based FPGAs are advantageous in their capabilities of being programmed more than once, they also must be reprogrammed each time the chip is power cycled. This is necessary because SRAM is inherently a volatile storage medium [1].

When a FPGA is configured, the configuration data, also known as a bitstream, is downloaded to the device. The bitstream contains all of the configuration bits that, when downloaded, implement a desired logic function. Configuration bits can control many resources inside the FPGA such as a LUTs' contents, routing resources, and the operational mode of embedded intellectual property (IP) cores such as BRAMs and DSP modules. Configuration bits also can determine the initialization values of flip-flops in PLBs and data values stored in BRAMs [29][2]. Figure 2.1 illustrates the use of configuration bits. In Figure 2.1a, a configuration bit is used to block or pass a signal for implementing programmable routing resources. In Figure 2.1b, configuration bits are used to implement a LUT such that a 2-input truth table can be realized for a logical function. Configuration bits can also initialize a BRAM as seen in Figure 2.1c.

As described in [1], configuration memory SRAM cells do not require fast read and write performance, which allows FPGA designers to use a five transistor SRAM as shown

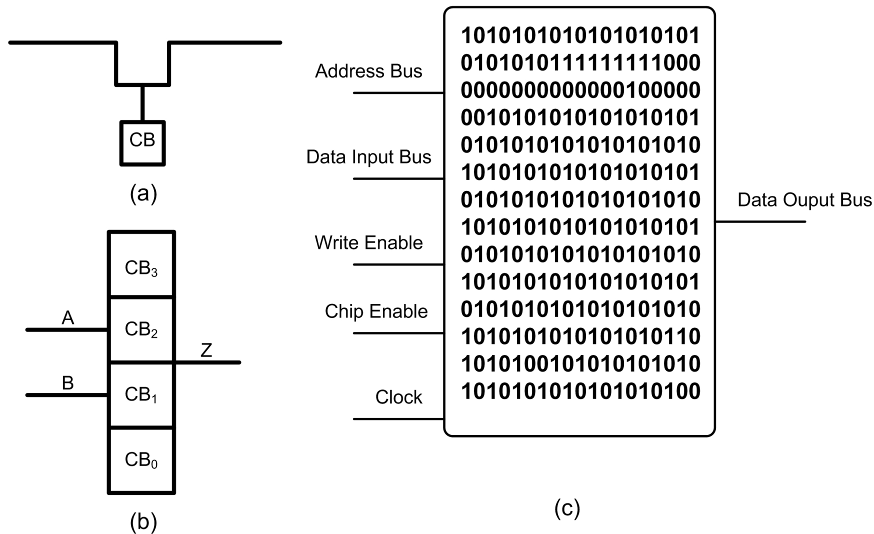

Figure 2.1: Configuration Bits in FPGAs

in Figure 2.2 instead of a more standard six transistor SRAM cell that provides both the complemented and uncomplemented form as seen in Figure 2.3.

Another important attribute concerning FPGA configuration is the ability for FPGAs to be partially reconfigured. Most modern FPGAs like the Xilinx Virtex series support partial reconfiguration [3][29]. Partial reconfiguration allows for a reduction in bitstream size by only storing the difference between a previous full download and the changes needed to implement the modified system function. Partial reconfiguration allows a designer to extract more functionality from a smaller FPGA for usage scenarios where a subset of implemented system functions can be swapped in and out of the FPGA without compromising critical performance parameters. In terms of testing FPGAs, partial reconfiguration is a valuable tool that allows the test configuration download time to be reduced.

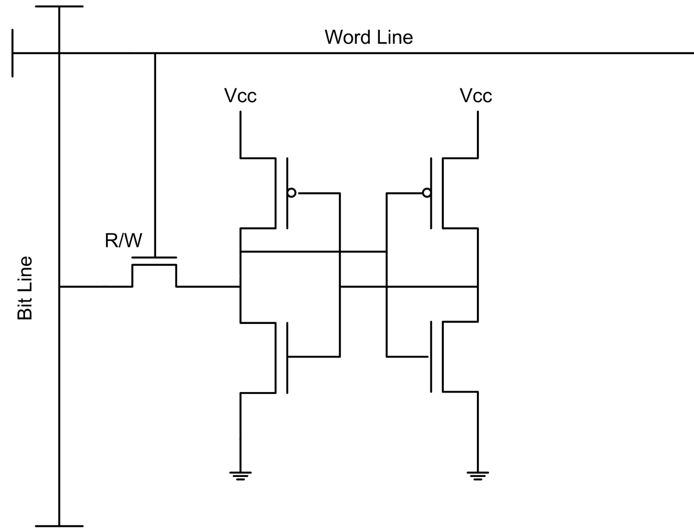

Figure 2.2: A Five Transistor Configuration SRAM Cell [1]

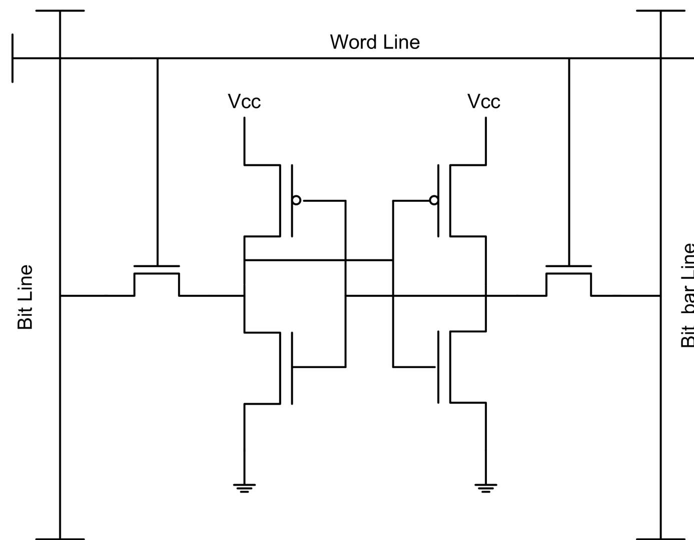

Figure 2.3: A Six Transistor Configuration SRAM Cell [1]

2.2 Virtex 4 Architecture

Xilinx released the Virtex 4 FPGA family in 2004 and, at the time of its introduction, it was one of the most complex FPGAs available on the market. For its production, Xilinx used a 1.2V, 90nm, triple-oxide process and it is only available in a flip-chip BGA package [10]. The Virtex 4 PLB contains four slices and each slice features two 4-input LUTs and two flip-flops. Dedicated memory storage is provided via a programmable 18K-bit SRAM called a block RAM. Previously in Virtex 2, an 18 x 18 multiplier module was available [29]. In Virtex 4, the multiplier has grown into a complete DSP module that incorporates a 48-bit accumulator attached to the 18 x 18 multiplier. The accumulator can also perform addition and subtraction on two data words without first being processed by the multiplier. Virtex 4 is available in three distinct product families: LX, SX, and FX as summarized in Table 2.1. The LX family contains eight devices that have been tailored toward design implementations that have high PLB usage. The SX family consists of three devices and is targeted toward DSP-oriented designs due to higher number of DSP modules. The FX family is a blend of the SX and LX family and provides more specialized modules which include PowerPC (PPC) and high performance I/O Serial/Deserial (SERDES) modules.

Virtex 4 employs a column based architecture. Each device has a center column which divides the FPGA into two halves. Beginning from the center column and moving in either direction, first there are columns of PLBs. The width of the first columns of PLBs differs by device size and family. Moving outward again, the next resource will either be a column of DSP or BRAM modules as illustrated in Figure 2.4. In FX family devices, either one or two PPC modules will be positioned to the left of the center column.

Table 2.1: Overview of Resources in Virtex 4 Family Devices [2]

Device	Row x Col	Slices	DSPs	BRAMs	PPCs	I/O
XC4VLX15	64 x 24	6,144	32	48	-	320
XC4VLX25	96 x 28	10,752	48	72	-	448
XC4VLX40	128 x 36	18,432	64	96	-	640
XC4VLX60	128 x 52	26,624	64	160	-	640
XC4VLX80	160 x 56	35,840	80	200	-	768
XC4VLX100	192 x 64	49,152	96	240	-	960
XC4VLX160	192 x 88	67,584	96	288	-	960
XC4VLX200	192 x 116	89,088	96	336	-	960
XC4VSX25	64 x 40	10,240	128	128	-	320
XC4VSX35	96 x 40	15,360	192	192	-	448
XC4VSX55	128 x 48	24,576	512	320	-	640
XC4VFX12	64 x 24	5,472	32	36	1	320
XC4VFX20	64 x 36	8,544	32	68	1	320
XC4VFX40	96 x 52	18,624	48	144	2	448
XC4VFX60	128 x 52	25,280	128	232	2	576
XC4VFX100	160 x 68	42,176	160	376	2	768
XC4VFX140	192 x 84	63,168	192	552	2	896

The smallest addressable unit of configuration memory in Virtex 4 is referred to as a *frame*, and each *frame* consists of 41 32-bit words. During the creation of a configuration bitstream, successive frame data along with frame addresses are written to the configuration bitstream. Designs utilizing regular structures, such as multiple identically configured BRAMs or replicated system logic in fault tolerant designs, can take further advantages of multi-frame write capabilities. Multi-frame write capabilities allow the configuration bitstream to specify multiple frame addresses to be given the same frame data. This can allow substantial reductions in configuration bitstream sizes and can be beneficial to reducing the time to perform BIST as discussed in Chapter 3.

All or parts of the configuration memory can also be read back. Configuration memory readback can be performed for various reasons such as configuration download verification,

but for BIST, readback is used to retrieve ORA results. In order to capture the contents of flip-flops, the *CAPTURE* module must be instantiated in the design. The *CAPTURE* module permits the current contents of flip-flops to overwrite their initial specified state stored in the configuration memory. Once the current flip-flop values have been captured, the ORA results can be retrieved via configuration memory readback.

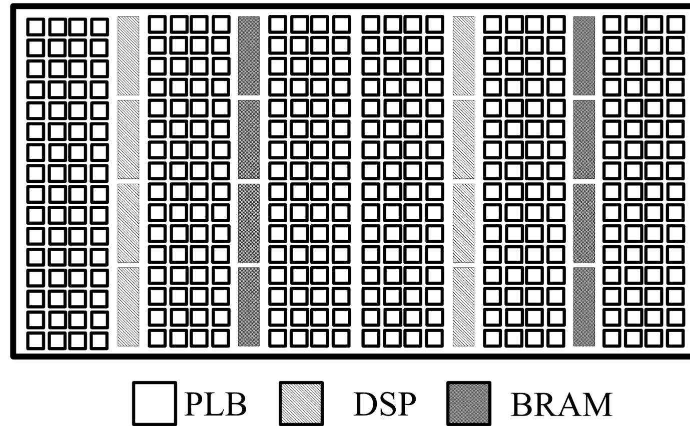

Figure 2.4: Basic Virtex 4 Architecture

2.2.1 Virtex 4 PLBs

Virtex 4 PLBs each consist of 2 SLICEMs and 2 SLICELs as pictured in Figure 2.5. A simplified representation of a SLICE is shown in Figure 2.6. A SLICEL contains 2 LUTs and 2 flip-flops plus multiplexers and logic gates to allow implementation of complex switching functions that span multiple SLICELs and even multiple PLBs. SLICEMs are a superset of SLICELs and have additional specialized features that enable them to be used as small distributed memories or function as a fast shift register. This thesis does not focus on testing PLBs and the slices within them. PLBs will be used in the work presented in this

thesis, however their use will only be to implement TPG and ORA components to facilitate testing BRAMs.

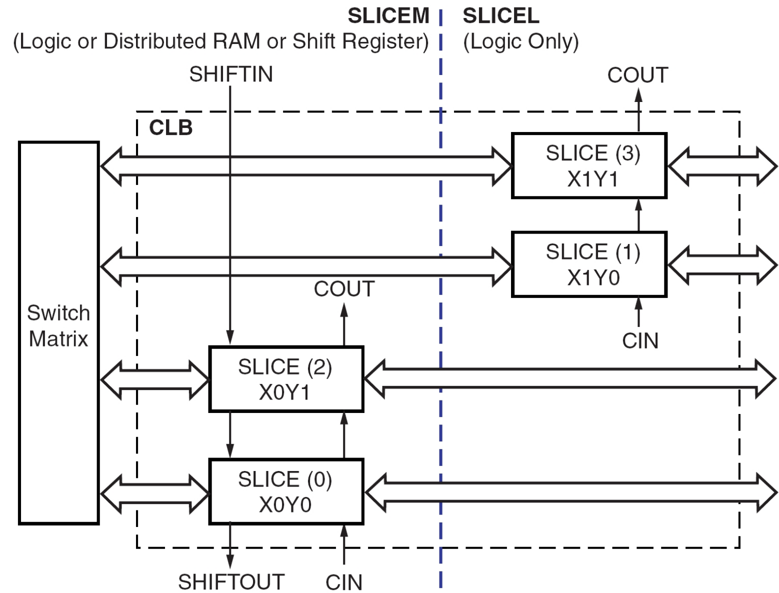

Figure 2.5: Virtex 4 PLB [2]

2.2.2 Virtex 4 BRAMs

This thesis concentrates on the development of BIST for BRAMs in Virtex 4 FPGAs. In [14], BIST was implemented for distributed RAMs, BRAMs, and multipliers in Xilinx's Virtex 2 series FPGAs. Distributed RAMs are memory resources created by using PLBs to create small memories. The BRAMs available in both Virtex 2 and 4 share much of the same functionality. The BRAM features present in Virtex 2 can be viewed as a subset of those present in Virtex 4. The Virtex 4 BRAM, as seen in Figure 2.7, is a true dual-port RAM meaning that is has dual address and data input and output buses. The memory array can be configured in multiple aspect ratios that utilize up to 18K of addressable memory as

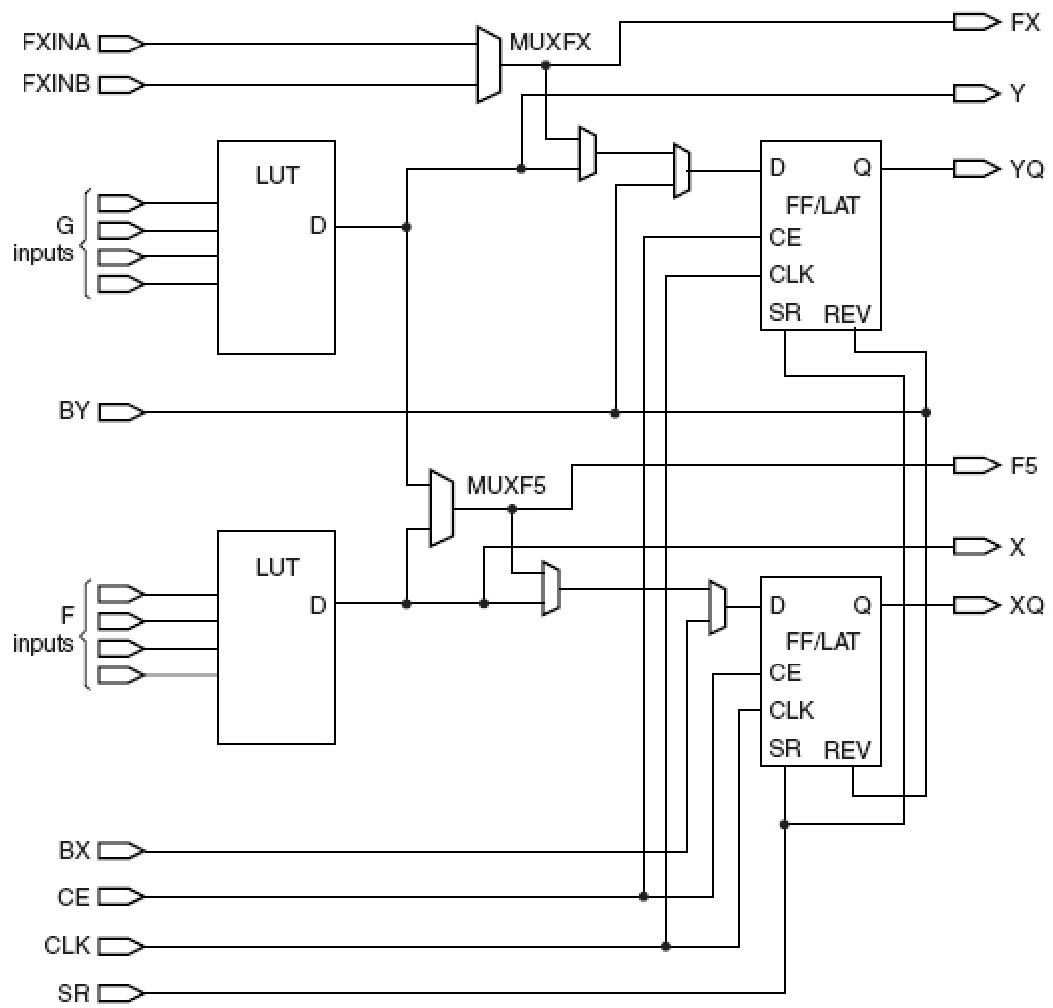

Figure 2.6: Simplified Virtex 4 Slice Diagram [2]

outlined in Table 2.2. For word sizes of eight bits and larger there are additional parity bits associated with each word. It is important to note, however, that the user is responsible for writing data to the parity bit locations since parity over the written data word is not calculated automatically. As for BRAM performance, all BRAM read and write operations can be completed in a single clock period unless using a registered output mode.

Table 2.2: Summary of Virtex 2 and 4 BRAM Aspect Ratios

Word Depth	Word Width	Parity Width
512	32	4
1K	16	2
2K	8	1
4K	4	-
8K	2	-
16K	1	-

Table 2.3 lists all of the input and output signal names and functionality found in a Virtex 4 BRAM. It should be noted that the Virtex 2 BRAM uses the same signal names except for the cascade input and outputs. Unlike Virtex 2, Virtex 4 BRAMs support cascading two 16K x 1-bit configured BRAMs to create a 32K x 1-bit memory without utilizing additional PLBs. All of the inputs are captured at either the rising or falling edge of the input clock depending if the BRAM clock input is programmed to be rising or falling edge triggered. Also, the following signals are programmable in their active levels: WE, EN, SSR, and REGCE. The WE signal is actually a 4-bit bus which enables the ability to write a single byte to a BRAM when configured to have a data word wider than a byte. This feature is most commonly used in conjunction with combining a PPC module and several BRAMs such that they implement program and data storage for the processor.

Table 2.4 summarizes additional configuration parameters for each BRAM. The SAVE-DATA configuration option determines if a partial reconfiguration will overwrite the present

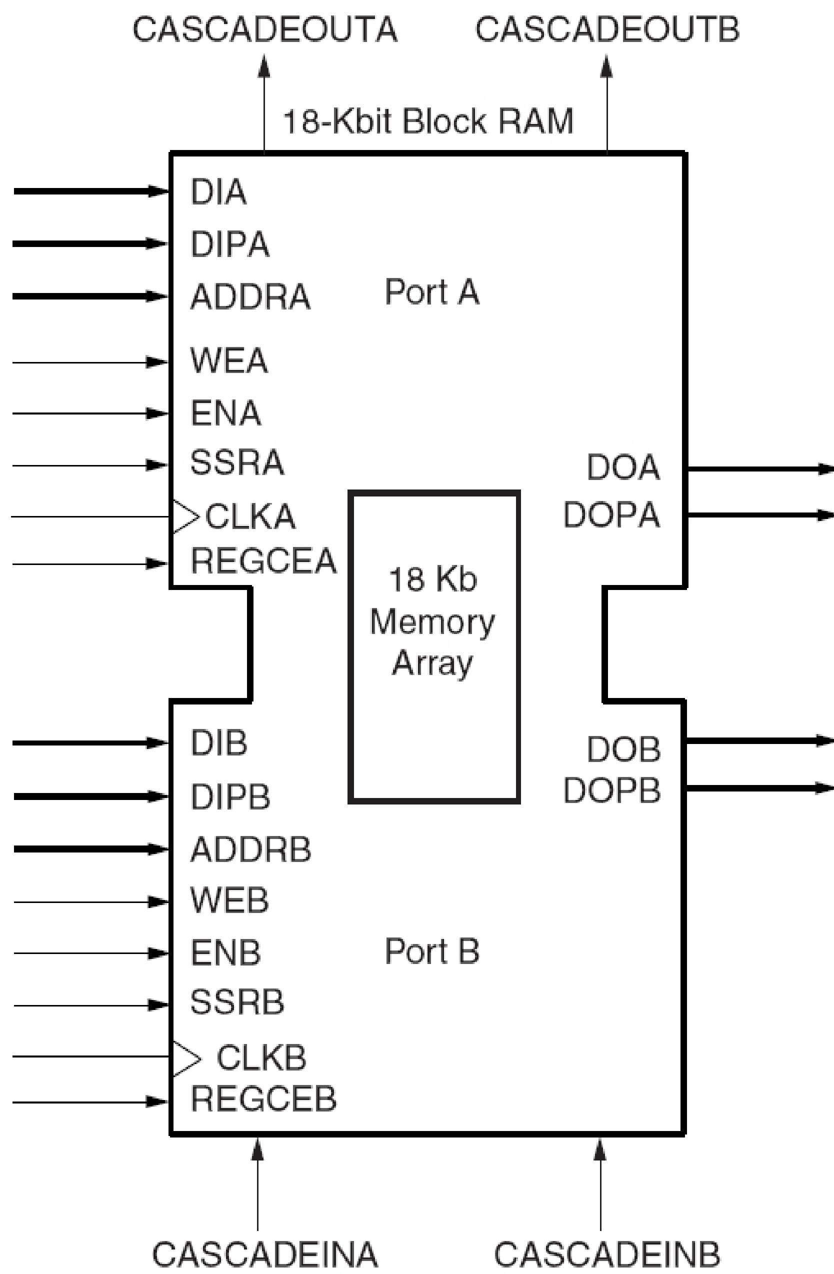

Figure 2.7: Virtex 4 BRAM [2]

Table 2.3: BRAM Signal Descriptions [2]

Port Name	Description
DI[A,B]	Data Input Bus
DIP[A,B]	Data Input Parity Bus
ADDR[A,B]	Address Bus
WE[A,B]	Write Enable
EN[A,B]	Port Enable
SSR[A,B]	Set/Reset
CLK[A,B]	Clock Input
DO[A,B]	Data Output Bus
DOP[A,B]	Data Output Parity Bus
REGCE[A,B]	Output Register Clock Enable
CASCADEIN[A,B]	Cascade input pin for 32K x 1 mode
CASCADEOUT[A,B]	Cascade output pin for 32K x 1 mode

contents of a BRAM. This option is useful if the reconfiguration is targeted at changing the system function that operates on data stored in the BRAM. The RAM_EXTENSION parameter for each port determines which BRAM is the UPPER or LOWER BRAM when configured in the cascade mode of operation. There are no restrictions on whether a particular BRAM can be UPPER or LOWER. This designation is decided by either the designer or the constraints associated with the available resources during resource placement. It is important to point out that the BRAMs located on a bottom row or directly above a PPC modules do not have CASCADEIN[A,B] inputs. Likewise, BRAMs located on the top row or directly below a PPC module do not have CASCADEOUT[A,B] ports. The DO_REG parameter determines if the output data bus is either latched or sent through an extra flip-flop. The READ_WIDTH and WRITE_WIDTH parameters determines the selected BRAM aspect ratio per port. The WRITE_MODE parameter selects one of three supported write modes. READ_FIRST brings the current contents of the addressed word to the output during a write operation. Likewise, WRITE_FRIST writes the input data to the addressed

location while it also forces the input data to the output of the BRAM. The NO_CHANGE write mode prevents the data output from changing during a write operation. Only a read operation will change the output data.

Table 2.4: BRAM Configuration Options [2]

Configuration Attribute	Parameters
SAVEDATA	TRUE, FALSE
RAM_EXTENSION_[A,B]	LOWER, UPPER, NONE
DO_[A,B]_REG	0,1
INVERT_CLK_DO[A,B]_REG	TRUE, FALSE
READ_WIDTH_[A,B]	36,18,9,4,2,1,0
WRITE_WIDTH_[A,B]	36,18,9,4,2,1,0
WRITE_MODE_[A,B]	READ_FIRST, WRITE_FIRST, NO_CHANGE

BRAM Error Checking Code (ECC) and Cascade Operational Modes

A pair of BRAMs can be configured to either implement a 512 x 72-bit ECC RAM or a 32K x 1-bit RAM. However, there are several restrictions for these additional modes concerning the actual placement during design implementation. An ECC BRAM can be placed in a Virtex 4 so long as the bottom BRAM is located on an even row. For clarity, the numbering convention Xilinx uses sets the bottom most row to be row zero. If an FX devices is being used, an ECC BRAM can not use the BRAM directly below or above the PowerPC core. Restrictions on the placement for cascaded BRAMs are somewhat less strict. Any pair of BRAMs in a column can be cascaded so long as the two BRAMs are physically adjacent and the pair does not span a PPC module when using FX family devices [2].

An ECC RAM is commonly used in systems that are designed to be fault tolerant. In Virtex 4, when a data word is written to an ECC RAM, a Hamming code is generated for the written data word and stored alongside the data. A Hamming code is a form of an

error checking code that inserts Hamming bits throughout a data word. Each Hamming bit contains parity over a subset of the data word. The contents of all the Hamming bits for a given data word is what is referred to as the Hamming code associated with a data word. The Hamming code is designed such that any single bit error in the Hamming code or in the data word will be able to indicate the presence of a single-bit error and indicate which bit is incorrect. If a Hamming bit is allocated to generate parity over the entire Hamming code and the data, then it is possible to provide single-bit error correction and double-bit error detection [30].

Virtex 4 uses what is termed a (72,64) Hamming code, meaning the Hamming codeword is 72 bits with 64 of those bits being the actual data. Seven of the eight Hamming bits are used to provide single-bit correction and the eighth bit is used to provide parity over the entire Hamming codeword which enables double-bit error detection [30]. Figure 2.8 illustrates the architecture of an ECC BRAM. An ECC BRAM also generates status bits that indicate if a single bit error was corrected or a double bit error was detected. Table 2.5 gives a description of the three valid ECC status words. It is important to note, however, that when a single bit error is corrected in the Virtex 4 ECC implementation, only the data at the output registers of the ECC BRAM are corrected. The contents stored in the BRAM are not corrected automatically [2].

The cascade mode of operation is implemented by using the MSB of the ADDR[A,B] bus, a single configuration bit, and dedicated routing to send the lower BRAM data output to the upper BRAM in the cascade pair. Figure 2.9 illustrates two BRAMs configured as a cascade pair. Data written to a cascaded pair is routed to either the lower or upper BRAM by the MSB of the address bus. In addition, the MSB of the address bus selects the data

from either the lower or upper BRAM during read operations. The lower BRAM outputs its data to the dedicated routing between a cascade pair and is then output via a multiplexer that is selected by the MSB of the address bus. Both lower and upper BRAMs in a cascade pair output data during every read operation but the output of the upper BRAM is the only valid output for a cascade configured BRAM pair [2].

Table 2.5: ECC Status Description

Status[1:0]	**Condition**
00	No error
01	Single Bit error corrected
10	Double Bit error detected
11	Invalid status condition

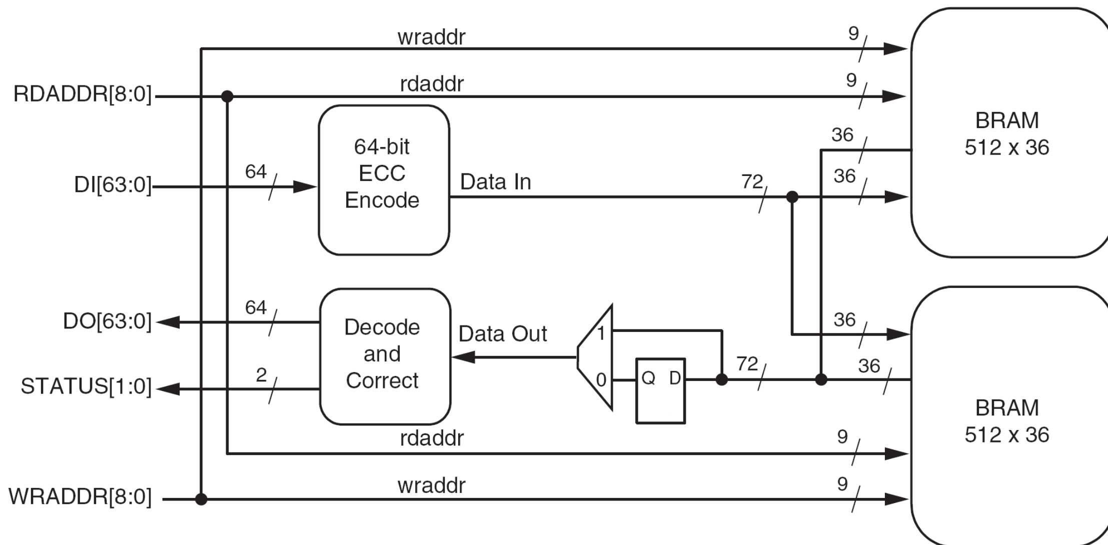

Figure 2.8: ECC BRAM Architecture [2]

2.2.3 Virtex 4 FIFOs

A First In, First Out (FIFO) memory is commonly used in digital systems to handle data flow control and buffering. A FIFO holds data in a queue such that the first data stored

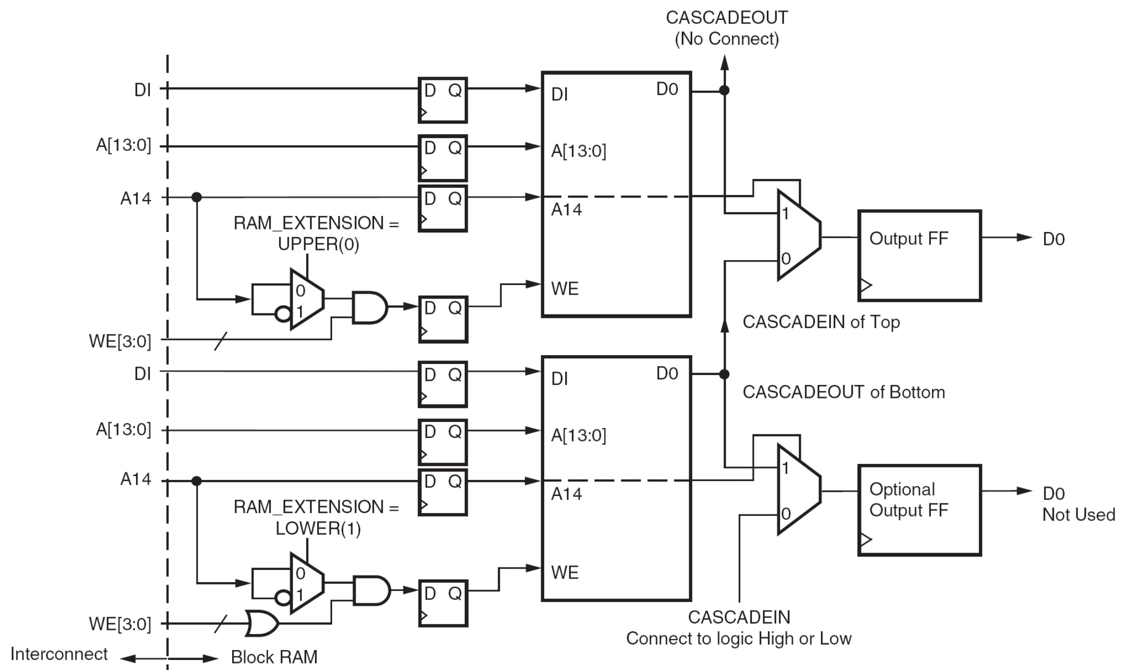

Figure 2.9: BRAM Cascade Operational Diagram [2]

is also the first data able to be retrieved. In Virtex 4 each BRAM can be configured as a FIFO without utilizing any surrounding PLBs. The FIFO implementation, as illustrated in Figures 2.10 and 2.11, generates both read and write pointers used to retrieve and store data from the BRAM [2]. A description of each FIFO input and output is given in Table 2.8. Also, status flags are generated to determine the state of the FIFO. In earlier FPGAs, such as Virtex 2, implementing a FIFO required using a substantial number of PLBs for this supporting logic.

Each Virtex 4 FIFO can be independently configured to four different depths as summarized in Table 2.6. Like BRAMs, the FIFO control signals RDCLK, WRCLK, RDEN, WREN, and RST also have programmable active levels. The first-word fall-through (FWFT) operational mode extends the depth of a FIFO by one data word. In this mode, the first

data item written to the FIFO is not stored in the BRAM, but is immediately available at the registered outputs. The remaining configuration options pertain to the programmable Almost Full/Empty flag limits. These programmable flags are set by the designer to help coordinate the status of the FIFO with surrounding logic.

Table 2.6: FIFO Configuration Options [2]

Configuration	ALMOST_EMPTY_OFFSET		ALMOST_FULL_OFFSET
	Standard	FWFT	
4k x 4	5 to 4092	6 to 4093	4 to 4091
2k x 9	5 to 2044	6 to 2045	4 to 2043
1k x 18	5 to 1020	6 to 1021	4 to 1019
512 x 36	5 to 508	6 to 509	4 to 507

The timing characteristics of the various FIFO status flags are given in Table 2.7. The most interesting attribute in this table is the assertion of the FULL flag. The FULL flag is asserted one clock cycle after the last possible data entry is written and deasserts 3 to 4 clock cycles later depending on whether the standard or FWFT operational modes is used. This latency means that one could accidentally write to a FIFO when in actuality it is already full. To remedy this problem, Xilinx recommends using the ALMOST FULL flag to signal the FIFO is full [2]. In terms of testing, the FIFO's status flag timing creates several problems in terms of the ability to create test algorithms that fully test the device in all modes of operation. This issue will be discussed in detail in Chapter 4.

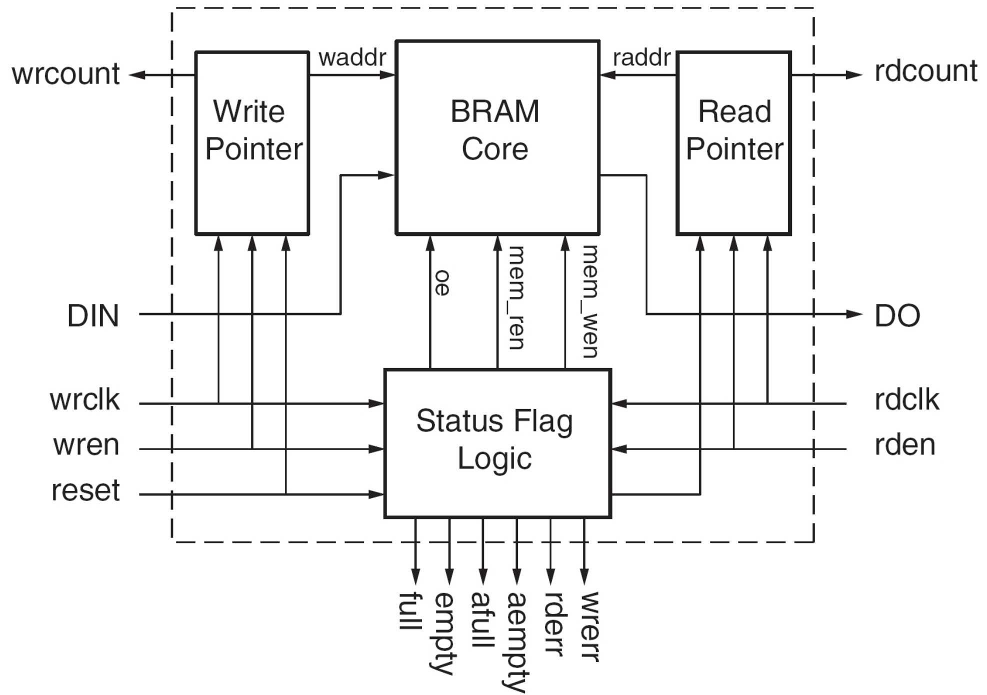

Figure 2.10: Virtex 4 FIFO Implementation [2]

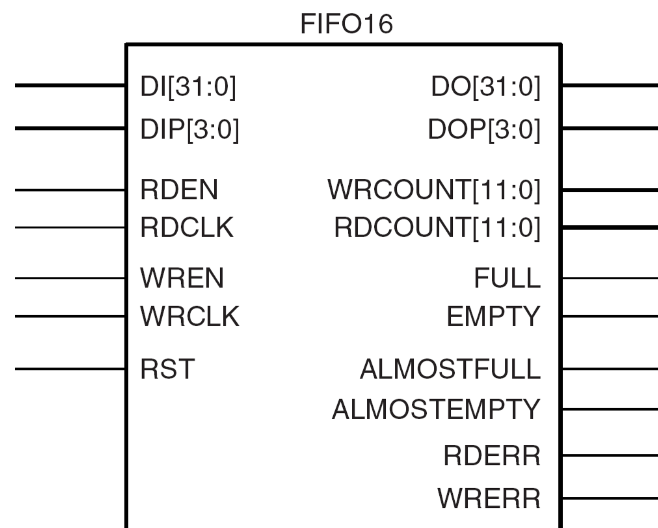

Figure 2.11: Virtex 4 FIFO [2]

Table 2.7: Virtex 4 Status Flag Clock Cycle Latency [2]

Clock Cycle Latency	Assertion		Deassertion	
	Standard	FWFT	Standard	FWFT
EMPTY	0	0	3	4
FULL	1	1	3	3
ALMOST EMPTY	1	1	3	3
ALMOST FULL	1	1	3	3
READ ERROR	0	0	0	0
WRITE ERROR	0	0	0	0

Table 2.8: FIFO Port Signal Descriptions [2]

Port Name	Direction	Description
DI	Input	Data input.
DIP	Input	Parity-bit input.
WREN	Input	Write enable, Active high or low.
WRCLK	Input	Clock for write domain operation. Rising or falling edge triggered.
RDEN	Input	Read enable, Active high or low.
RDCLK	Input	Clock for read domain operation. Rising or falling edge triggered.
RESET	Input	Asynchronous reset of all FIFO functions, flags, and pointers. Active high or low.
DO	Output	Data output.
DOP	Output	Parity-bit output.
FULL	Output	All entries in FIFO memory are filled. No additional write enable is performed.
ALMOSTFULL	Output	Almost all entries in FIFO memory have been filled.
EMPTY	Output	FIFO is empty. No additional read can be performed.
ALMOSTEMPTY	Output	Almost all valid entries in FIFO have been read.
RDCOUNT	Output	The FIFO data read pointer.
WRCOUNT	Output	The FIFO data write pointer.
WRERR	Output	When the FIFO is full, any additional write operation generates an error flag.
RDERR	Output	When the FIFO is empty, any additional read operation generates an error flag.

2.2.4 Virtex 4 CAD Tools

Xilinx provides a complete set of computer-aided design (CAD) tools that enable a designer to implement digital systems using a high-level design methodology that supports schematic entry, hardware description language (HDL) synthesis, and IP core integration. A designer can either use a graphical user-interface (GUI), *Project Navigator*, to implement designs, or command-line tools can be used. By using command-line tools and batch files, one can automate the BIST configuration generation process. Table 2.9 summarizes several Xilinx tools that are used to implement the BIST configurations presented in this thesis. *BITGEN* has several options that can be utilized in BIST configuration generation. *BITGEN* supports the creation of partial configuration bitstream from a set of regular full configuration bit-files. During the creation of partial bit-files *BITGEN* compares two NCD design files and generates a bit-file containing the difference between the two. The *XDL* command-line tool allows the conversion between an NCD file and an XDL file. An XDL file type contains a human-readable netlist description of a FPGA configuration. *XDL* can convert an XDL file to an NCD file type which describes a given FPGA configuration, but this file type is a binary file which is not human-readable [31].

2.2.5 Virtex 4 Boundary Scan

Virtex 4 implements boundary scan, also called JTAG, that meets IEEE Standard 1149.1-2001 [3]. Boundary scan was originally intended as a mechanism for testing interconnect between multiple IC chips on a system board. A boundary scan implementation includes the following four signals: Test Clock (TCK), Test Mode Select (TMS), Test Data In (TDI), and Test Data Out (TDO). By asserting TMS to either a logic high or low during

Table 2.9: Summary of Xilinx Design Tools

Application	Input File Type	Output File Type	Description
XST	VHDL, Verilog	NGC	Synthesis Tool - Compiles HDL and generates a design netlist compatible with Xilinx devices
NGDbuild	NGC	NGD	Compiles designs to common format
MAP	NGD	NCD	Translates post-synthesis design to a device specific implementation
PAR	NCD	NCD	Places and routes device specific designs
BITGEN	NCD	.BIT, .RBT	Creates bitstream configuration files for download
XDL	XDL, NCD	NCD, XDL	Converts in between XDL and NCD design formats
TRCE	NCD	TWR	TRACE - Generates configuration timing analysis report

a rising transition on TCK, one can navigate a state machine termed the test access port (TAP) controller as illustrated in Figure 2.12 [11]. However, the technology has evolved and now most FPGAs allow for device programming via boundary scan [29][3]. Xilinx's Virtex series of FPGAs also allow connecting boundary scan signals to internal FPGA logic. This connection is made via what Xilinx calls boundary scan (BSCAN) modules. In Virtex 4, four BSCAN modules are available for use and each module is selected by shifting a specific data word into TAP's instruction register as shown in Table 2.10. The convention that is used when shifting data either into the data register when in state Shift-DR or the instruction register when in state Shift-IR is to assert TMS to a '1' on the last data bit transmitted over TDI.

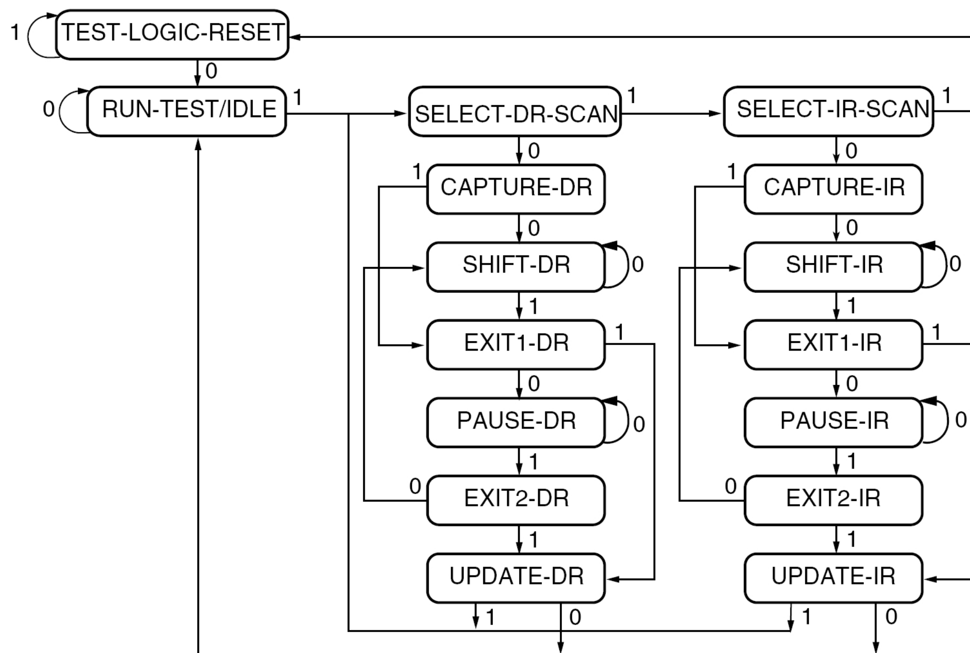

Figure 2.12: TAP Controller State Diagram [3]

Table 2.10: Virtex 4 BSCAN Module Access Commands [3]

Boundary Scan Command	Binary Code (9:0)	Description
USER1	1111000010	Access user-defined register 1
USER2	1111000011	Access user-defined register 2
USER3	1111100010	Access user-defined register 3
USER4	1111100011	Access user-defined register 4

2.3 SRAM Testing

Most digital systems incorporate some type of memory element whether it is an embedded SRAM in a microprocessor's cache or a standalone memory in a digital system. Applications that utilize memories are quite abundant and thus there is a need to ensure that memory devices such as SRAMs are fault-free. Figure 2.13 illustrates a functional model of a common SRAM. As seen in this figure, an array of memory cells and supporting functions such as decoders, sense amplifiers, and storage registers are all components integral to a SRAM. Figure 2.14 shows how a two-port memory can be made by additional word and bit lines that connect the the cross-coupled inverters.

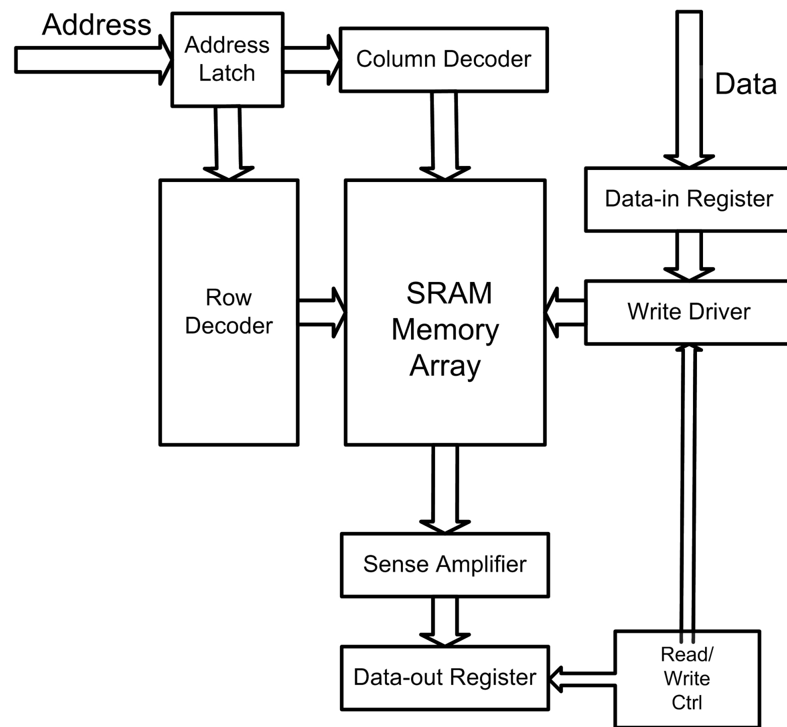

Figure 2.13: SRAM Memory Functional Model

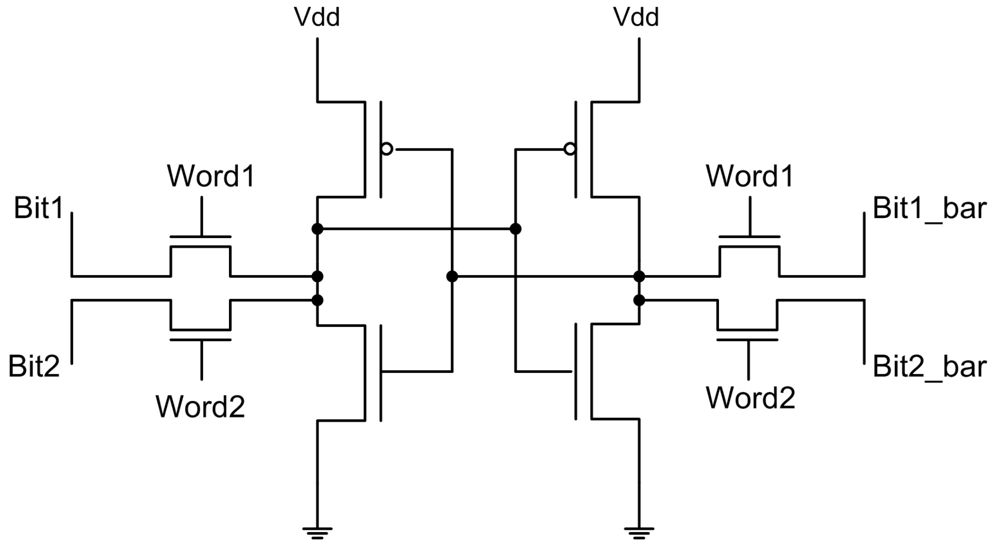

Figure 2.14: Structural Model of a two-port SRAM cell

2.3.1 SRAM Fault Models

In [26], van de Goor shows that a simple stuck-at fault (SAF) model is not sufficient for modeling faults found in memory devices. Table 2.11 lists common types of faults for SRAMs and are sometimes referred to as simple faults. Coupling faults can be further classified into four subtypes: inversion coupling faults (CFin), idempotent coupling faults (CFid), state coupling faults (CFst), and disturb coupling fault (CFdst). A CFin refers to a ↑ and/or ↓ write operation in a coupling cell that causes an inversion in the coupled cell. For example, if *cell A* and *cell B* are both '1' and a '0' is written to *cell A*, a CFin fault in *cell B* would invert its value to a '0' as well. In this case *cell A* is considered the coupling cell and *cell B* is deemed the coupled cell. A CFid is similar to a CFin except that the coupled cell is forced to either a 1 or 0 instead of an inversion. A CFst is slightly more complicated as the coupled cell is only affected if the coupling cell is in a certain state. For example, a '1' in *cell A* could force *cell B* to either a '1' or '0', but if *cell A* is a '0', no fault

effect would be observed. Finally, a CFdst refers to a fault where the coupled cell undergoes a transition due to a read or write operation to the coupling cell [32]. When coupling faults are considered as part of a memory's fault model, often the occurrence of multiple faults is considered. These multiple faults are said to be linked and are referred to as linked faults. The term linked is used because in the presence of multiple faults, each fault can possibly influence the effect of the other faults [11].

Table 2.11: Common SRAM Fault Types

Fault Type	Description
Stuck-At (SAF)	A logic value of a memory cell always being 1 or 0
Transition Faults (TF)	Memory cell not able to make a 0 to 1 or 1 to 0 transition
Coupling Faults (CF)	A change in one memory cell, the coupling cell, causes another cell , the coupled cell, to change its value
Address Decoder Faults (AF)	The expected memory cell is not selected or another cell is selected
Data Retention Faults (DRF)	The expected stored data is corrupted
Pattern Sensitive Faults (PSF)	The contents of a memory cell changes the value of another memory cell

2.3.2 March Tests for Single-Port Memories

Memory devices are traditionally tested with march tests. March tests apply defined test patterns consisting of writing and reading varying patterns of 1s and 0s to and from a memory device. Table 2.12 lists common notations used to describe march tests while Table 2.13 lists several march tests of varying complexity. For example, when running the MATS+ march test on a memory device, a '0' is written to each memory location in either a descending or ascending traversal. Next, starting at the lowest addressed memory location and traversing upward, a '0' is read and a '1' is written to each memory location. The next

Table 2.12: March Test Notation Descriptions

Notation	Description
r	A read operation
w	A write operation
r0	Read a 0
w0	Write a 0
w1	Write a 1
$\uparrow$	Traverse upward through memory addresses
$\downarrow$	Traverse downward through memory addresses
$\updownarrow$	Traverse any direction through memory addresses

sequence begins at the highest memory location and traverses downward while reading a '1' and then finally writing a '0' to each memory location.

In terms of fault detection, MATS+ is able to detect all SAF and all AF [11]. More complex tests such as March Y can detect additional faults, those being TF and CFin and some linked faults. In addition, March C- can detect most all types of CFs. The general trend for march test fault detection is that longer, more complex tests offer higher fault detection. While high fault detection is obviously desired, some tests like those used specifically to target pattern sensitive faults can be unpractical, especially if the DUT is of any sufficient size.

In [32], van de Goor presents March LR as "a test for simple faults and realistic linked faults." The set of realistic linked faults further reduces the universe of possible linked faults by removing combinations that are not likely to occur in actual devices. Realistic linked faults do not include linked faults containing one or more CFins and linked faults containing two CFids or two CFdsts are also removed. Van de Goor shows that March LR is superior to March C- as it can also detect some neighborhood PSFs (NPSFs)[32].

Table 2.13: Common March Tests for Single Port Memories

March Test	Description	Test time
MATS+	$\{\updownarrow(w0);\uparrow(r0,w1);\downarrow(r1,w0)\}$	5N
March C-	$\{\updownarrow(w0);\uparrow(r0,w1);\uparrow(r1,w0);\downarrow(r0,w1);\downarrow(r1,w0);\updownarrow(r0)\}$	10N
March Y	$\{\updownarrow(w0);\uparrow(r0,w1,r1);\downarrow(r1,w0,r0);\updownarrow(r0)\}$	8N
March LR	$\{\updownarrow(w0);\downarrow(r0,wl);\uparrow(r1,w0,r0,wl);\uparrow(r1,w0);\uparrow(r0,w1,r1,w0);\updownarrow(r0)\}$	14N

Note: N = Number of address locations

All of the march tests listed in Table 2.13 are designed for bit-oriented memories (BOMs), meaning each memory word is a single bit. However, many memories, including the Virtex 4 BRAM, perform as a word-oriented memory (WOM), meaning each word is more than a single bit. Executing BOM-based march tests on WOM involves extending the bit operation to the entire word. For example, if a *w0* operation is to be applied to a BOM of 4-bits it could be interpreted as *w0000*, meaning write all zeros to each bit in the data word. Writing and reading either all zeros or ones in a WOM does not sufficiently detect CF between cells in a data word. In [4], van de Goor et al. develop methods for converting BOM march tests, specifically March LR, to WOM march tests by using background data sequences (BDS). Instead of writing either all zeros or ones, BDS involve writing binary patterns consisting of alternating ones and zeros and alternating sets of ones and zeros. Table 2.14 list all possible BDS for an eight bit word. The number of BDS for a M-bit word is given by Equation 2.1. Figure 2.15 illustrates converting March LR to March LR with 4-bit BDS. The test length increases from 14N to 30N, where N is number of words in the memory. In general, Equation 2.2 gives the expected test length for a WOM march test derived from a BOM march test. The variable 'M' in Equation 2.2 refers to the number of bits in a data word. Converting March LR to incorporate BDS requires running

{[↕(w0000);↓(r0000,wl111);↑(r1111,w0000,r0000,wl111);
↑(r1111,w0000);
↑(r0000,w1111,r1111,w0000);↕(r0000)]

[↑(r0000,w1111,r1111);↓(r1111,w0000,r0000);
↑(r0000,w0101,w1010,r1010);↓(r1010,w0101,r0101);
↑(r0101,w0011,w1100,r1100);↓(r1100,w0011,r0011);↑(r0011)]}

Figure 2.15: March LR with 4-bit BDS [4]

Table 2.14: Background Data Sequence for 8-bits

#	Normal	#	Inverse
0	00000000	1	11111111
2	01010101	3	10101010
4	00110011	5	11001100
6	00001111	7	11110000

the original March LR test as enclosed in the first set of square brackets in Figure 2.15 and then addition additional marches that incorporate BDS.

$$\#BDS = \lceil \log_2 (M) \rceil + 1 \tag{2.1}$$

$$Test\ Length = (16 + 7 * \lceil \log_2 (M) \rceil \tag{2.2}$$

2.3.3 March Tests for Dual-Port Memories

In [5], Hamddioui and van de Goor describe two march tests for dual-port memories. Specific march tests are required for both ports in order to detect specific faults in dual-port memories. Dual-port memories generally support the following operations by the two ports[5]:

- Simultaneous read and write operations to different addresses.

- Simultaneous read and write operations to the same address. For this case, however, the write operation is assumed to have higher priority over the read operation.

- Two simultaneous reads to either the same or different addresses.

- Two simultaneous write operations to different addresses.

The only operation not allowed is simultaneous write operations to the same address location. The Virtex 4 BRAM supports the same operations and limitations discussed above. The faults associated with dual-port memories are classified as 2PF1 and 2PF2[5]. Faults classified as 2PF1 are sensitized by two simultaneous reads or both a read and write operation. Two types of faults are associated with two simultaneous reads. In one case, the correct data value is read through the sense amplifier in the SRAM cell, but the actual data stored in the cell will flip. The other case is when sense amplifier reads an incorrect value and the actual data stored is also flipped. Simultaneous read and write operations can also cause the intended write operation to not occur. March s2pf, as seen in Figure 2.16, detects all 2PF1 type faults. The ':' symbol used in Figure 2.16 separates the operation on each port. For example, 'r1:-' would indicate a read operation on one port and any allowed operation on the second port.

$$\{ \underset{M_0}{\Updownarrow (w0 : n)} \; ; \; \underset{M_1}{\Uparrow (r0 : r0, r0 : -, w1 : r0)} \; ; \; \underset{M_2}{\Uparrow (r1 : r1, r1 : -, w0 : r1)} \; ; \; \underset{M_3}{\Downarrow (r0 : r0, r0 : -, w1 : r0)} \; ; \; \underset{M_4}{\Downarrow (r1 : r1, r1 : -, w0 : r1)} \; ; \; \underset{M_5}{\Downarrow (r0 : -)} \; \}$$

Figure 2.16: March s2pf [5]

2PF2 faults are similar to the 2PF1 type faults except the affected cell is a neighboring cell. For example, a fault may flip a neighboring $cell_i$ if two read operations occur on $cell_j$. Another type of fault occurs when a read occurs at $cell_i$ and a write to $cell_j$. During this scenario, the fault will cause the read to return the wrong value [5]. March d2pf, as seen in Figure 2.17, is able to detect all 2PF2 faults. The the variables 'c' and 'r' correspond to the memory cell column and row location, respectively. The variables 'C' and 'R' represent the number of memory cell columns and rows, respectively. The widest word size, 36-bits, is assumed to be the memory array column size. When this march test is applied to BRAMs in Virtex 4, 'C' is considered to be zero and R is '512'.

$$
\begin{array}{l}
\{ \underbrace{\Updownarrow (w0:n)}_{M_0} \; ; \; \underbrace{\Uparrow_{c=0}^{C-1} (\Uparrow_{r=0}^{R-1} (w1_{r,c} : r0_{r+1,c}))}_{M_1} \; ; \; \underbrace{\Uparrow_{c=0}^{C-1} (\Uparrow_{r=0}^{R-1} (w1_{r,c} : r1_{r+1,c}))}_{M_2} \; ; \\
\quad \underbrace{\Uparrow_{c=0}^{C-1} (\Uparrow_{r=0}^{R-1} (w0_{r,c} : r1_{r+1,c}))}_{M_3} \; ; \; \underbrace{\Uparrow_{c=0}^{C-1} (\Uparrow_{r=0}^{R-1} (w0_{r,c} : r0_{r+1,c}))}_{M_4} \; ; \\
\quad \underbrace{\Uparrow_{c=0}^{C-1} (\Uparrow_{r=0}^{R-1} (w1_{r,c} : r0_{r,c+1}))}_{M_5} \; ; \; \underbrace{\Uparrow_{c=0}^{C-1} (\Uparrow_{r=0}^{R-1} (w1_{r,c} : r1_{r,c+1}))}_{M_6} \; ; \\
\quad \underbrace{\Uparrow_{c=0}^{C-1} (\Uparrow_{r=0}^{R-1} (w0_{r,c} : r1_{r,c+1}))}_{M_7} \; ; \; \underbrace{\Uparrow_{c=0}^{C-1} (\Uparrow_{r=0}^{R-1} (w0_{r,c} : r0_{r,c+1}))}_{M_8} \; \}
\end{array}
$$

Figure 2.17: March d2pf [5]

2.4 Overview of BIST for FPGAs

Developing BIST for FPGAs consists of designing test configurations that fully test all components within a FPGA. Traditionally, these components have been grouped into the following types of tests: BIST for PLBs, BIST for I/O Buffers, BIST for programmable routing resources, and BIST for specialized embedded cores such as large SRAMs [14][18][33]. The work presented in this thesis concentrates on testing Virtex 4 BRAMs which fall into the category of testing specialized embedded cores. BIST for PLBs is also called Logic BIST

and this is the most common BIST found in the literature. Logic BIST usually requires repeatedly configuring PLBs in different modes of operations and applying test patterns the PLBs under test. A general BIST architecture is illustrated in Figure 2.18. Two identical TPGs drive alternating columns of blocks under tests (BUTs) whose outputs are observed by two adjacent ORAs. For Logic BIST, all of the BIST components are implemented using PLBs. In certain implementations such as in [34], the TPGs can be implemented using other available specialized cores. In order to test all PLBs in a FPGA, the entire BIST architecture must be flipped such that the PLBs acting as BUTs are now ORAs and the PLBs previously implementing ORAs are now configured to be BUTs [34].

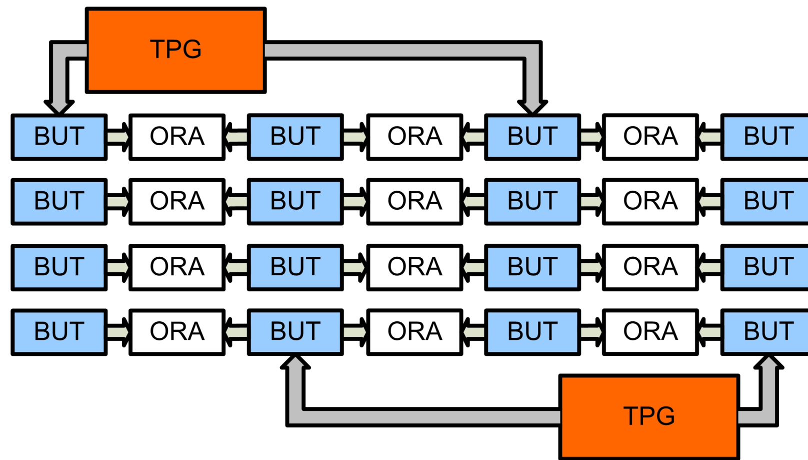

Figure 2.18: A General Comparison Based BIST Architecture

Several different types of ORAs are used in BIST for FPGAs. A comparison-based ORA design is the most common. Figures 2.20 and 2.21 illustrate two types of comparison-based ORAs: one with a shift chain and one without a shift chain. Using an ORA design without shift chains reduces the size of an ORA and can allow for more ORAs to be used which increases diagnostic resolution [21]. In both designs, any mismatch from the BUTs will cause the flip-flop to latch to a '1' until the end of the test. The addition of the

shift chain allows the results to be shifted out of the device at the end of testing [14]. A comparison-based ORA with no shift chain can be used when the FPGA supports read back of the contents of the flip-flops in the each PLB via configuration memory readback. Modern FPGAs such as the Virtex family from Xilinx support this readback operation [29][3].

Figure 2.18 also presents an example of a general comparison-based ORA BIST architecture. This type of comparison has limitations in terms of diagnostic resolution for BUTs located at the outer columns since they are only compared by a single ORA. In [14], cases where a fault could escape detection are discussed. A circular-comparison based BIST architecture, as seen in Figure 2.19, eliminates the loss of fault detection around outer columns. Circular-comparison based BIST architectures require a minimum of three BUTs for fault detection and four BUTs for fault diagnosis [28]. The only condition for a fault to escape detection is for all BUTs in a circular comparison chain to have identical equivalent faults. If the comparison chain is sufficiently long, then the probability of multiple equivalent fault occurring is negligible.

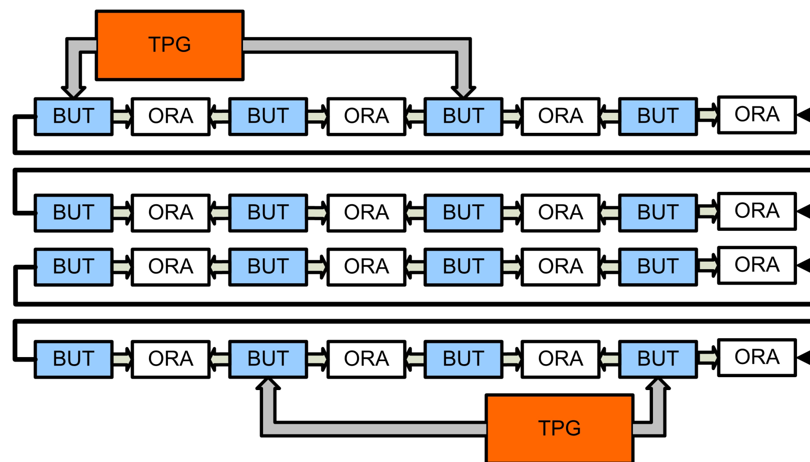

Figure 2.19: A Circular Comparison Based BIST Architecture

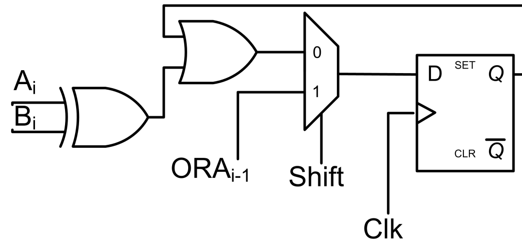

Figure 2.20: Comparison Based ORA with Shift Chain

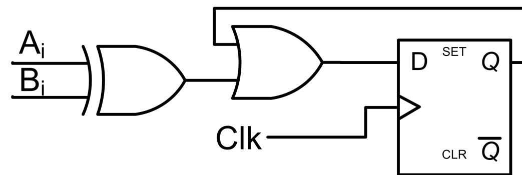

Figure 2.21: Comparison Based ORA without a Shift Chain

2.4.1 BIST for BRAMs

In [17], Garimella developed BIST configurations to test BRAMs in Virtex 2 FPGAs. The work presented in this thesis borrows many of the same concepts and applies them to BIST for BRAMs in Virtex 4. BRAMs in Virtex 2 only operate as a dual-port memory. There are no built-in FIFOs or ECC or cascade modes of operation. Garimella's approach was to create a portable BIST architecture that could be adapted to different FPGAs. A HDL was used to model, synthesize, and implement the entire BIST architecture. Garimella used the comparison-based ORA design that includes the shift chain. The end of the shift chain was connected to the boundary scan port TDO. The boundary scan pin TDI was connected to each ORA such that a '1' on TDI enabled all of the ORAs to become a shift register. Likewise, a '0' on TDI would disable the shift operation and the ORAs would

Table 2.15: Virtex 2 BRAM Summary

BIST Configuration	Test Algorithm	Address Locations (A)	Data Width (D)	Clock Cycles
1	March LR w/ BDS	512	36	$58A$
2	March LR	1K	18	$14A$
3		2K	9	$14A$
4		4K	4	$14A$
5		8K	2	$14A$
6		16K	1	$14A$
7	March s2pf	512	36	$14A$
8	March d2pf	512	36	$9A$

TOTAL BIST CLOCKS= 485,888

continue to compare BUT responses on each clock cycle. The system clock for the ORAs, BUTs, and TPG was sent through the TCK port on the Boundary Scan port.

Table 2.15 summarizes the eight BIST configurations generated by Garimella for Virtex 2 BRAMs. March LR was implemented as a TPG and used for each of the programmable aspect ratios. March LR was chosen primarily because of its relatively low complexity and high fault coverage. The 512 x 36 addressing mode used March LR modified to generate BDS so that fault detection of intra-word coupling faults was maximized. March s2pf and d2pf were also used to test dual-port functionality.

The advantages of Garimella's approach was that the BIST architecture was described in a HDL which allowed for very rapid development. There are, however, some important disadvantages. BIST approaches such as in [34] take advantage of partial reconfiguration FPGA techniques in order to reduce test time and the amount of configuration data needed to download to a device during multiple BIST configurations. In order for several FPGA configurations to use partial reconfiguration efficiently, there needs to be only small regular changes between each configuration. In terms of BIST for PLBs, the changes made between

each configuration is only to change the operational mode of the BUT and keep the rest of the BIST circuitry static [34]. This approach yielded substantial test time and configuration storage reductions when applied to BIST for PLBs in Virtex 2 and Virtex 4.

Garimella's approach is not compatible with partial reconfiguration. The use of a HDL to develop BIST configurations removes the control over the placement of BIST circuitry. The CAD tools that transform a HDL to a configuration ready for download are not able to take advantage of the regularity of BIST architectures. The result is that the TPG and ORA portions of the BIST circuitry are intermingled amongst the available logic resources surrounding BRAMs. CAD tools also do not obtain identical results in repeated implementations. This severely limits the use of partial reconfiguration because at no time are subsequent BIST configurations guaranteed to be similar to previous configurations. For this reason, Garimella used full configuration downloads for each BIST configuration. Another disadvantage with the HDL approach is that developing BIST at a high level reduces the controllability over the resource being tested. For example, in BIST configurations that used active low signals, the synthesis tool inverted the signals connected to the BRAM instead of configuring the BRAM ports to the opposite level. This behavior was observed when determining the portability of Garimella's BIST approach to Virtex 4. As a result, configuration bits and logic inverting BRAM control signals are not tested by Garimella's approach.

2.5 Thesis Restatement

The work by Garimella in developing BIST configurations for Virtex 2 BRAMs is a basis for the work presented in this thesis. The main disadvantage of Garimella's approach

is not being able to take advantage of partial reconfiguration techniques between each BIST configuration following the initial full BIST configuration download. Downloading BIST configurations is the most time expensive portion of the entire time required for BIST. The time spent applying BIST clock cycles is nominal compared to the configuration time required. This thesis presents the development of BIST configurations for Virtex 4 BRAMs compatible with partial reconfiguration. Using partial reconfiguration, a large portion of the time needed to perform BIST can be reduced. Unlike Garimella, the BIST architecture presented in this thesis does not rely on a HDL to describe the overall BIST architecture. Instead, this approach uses custom created BIST programs to implement BIST configurations which enables much greater control of each BIST configuration as compared to the HDL approach by Garimella. This improvement allows for testing BRAMs in all of their configuration options gives higher fault coverage. Chapter 3 will introduce the general BIST architecture for testing BRAMs. Also in Chapter 3, BIST configurations testing BRAMs in single and dual-port modes are presented. BIST configurations for FIFO operational modes are developed in Chapter 4 while ECC and cascade mode BIST configurations are presented in Chapter 5.

Chapter 3

Virtex 4 Block RAM BIST Implementation

A BIST approach developed for BRAMs in Virtex 4 FPGAs is presented in this chapter. The BIST architecture will be discussed as well as a TPG which is able to generate multiple march tests. Finally, results from applying BIST to Virtex 4 devices are given and compared to results obtained by Garimella in [17].

3.1 Virtex 4 BRAM BIST Architecture

In all Virtex 4 devices, BRAMs are located along columns that span the entire device. In between columns of BRAMs there are at least 4 columns of PLBs. In order to achieve high fault coverage and diagnostic resolution, circular comparison-based ORAs are used in conjunction with two identical TPGs which drive alternating rows of BRAMs as seen in Figure 3.1. Each BRAM output is observed by ORAs immediately adjacent and directly above each BRAM. The topmost BRAM outputs connect to adjacent ORAs as well, however, there are no ORAs above the topmost BRAM. To enable all BRAMs to have each output compared by two ORAs, the topmost BRAM's outputs are compared by the ORAs adjacent to the bottommost BRAM in each column. This arrangement implements a circular comparison chain per column of BRAMs.

Each BRAM has 72 outputs, 36 per port. A Virtex 4 slice contains two 4-input LUTS and two flip-flops. With these resources, two comparison-based ORAs can be implemented in each slice. Given that there are four slices per PLB, it would require nine PLBs to implement all the ORAs needed to compare the outputs of two BRAMs. The Virtex 4

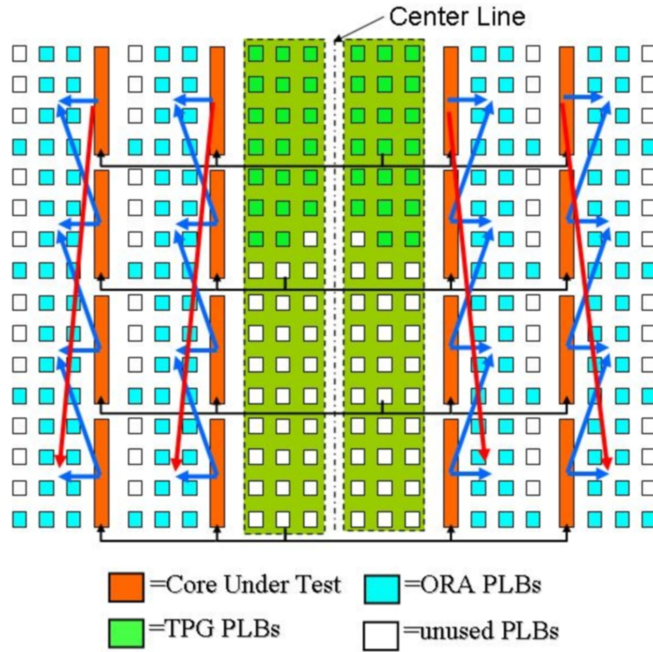

Figure 3.1: BRAM BIST Architecture

architecture places four rows of PLBs per BRAM, which in turn yields 16 PLBs total since there are four columns between each BRAM. From these 16 PLBs, nine PLBs are used to implement ORAs beside each BRAM. Figure 3.2 illustrates the BRAM to ORA connections. The first column of four PLBs compare DOA[31:0] while the second column of 4 PLBs compares DOB[31:0]. A single PLB from a third column is used to compare the parity bits DOPA[3:0] and DOPB[3:0]. The fourth column of 4 PLBs is unused. Locating ORAs in an algorithmic method also facilitates the use of configuration memory readback to retrieve the ORA results at the end of a test. The ORA results are stored in frame data and the bit-locations of the ORAs within each frame can be obtained by generating a logic allocation file using the '-l' argument in *BITGEN*.

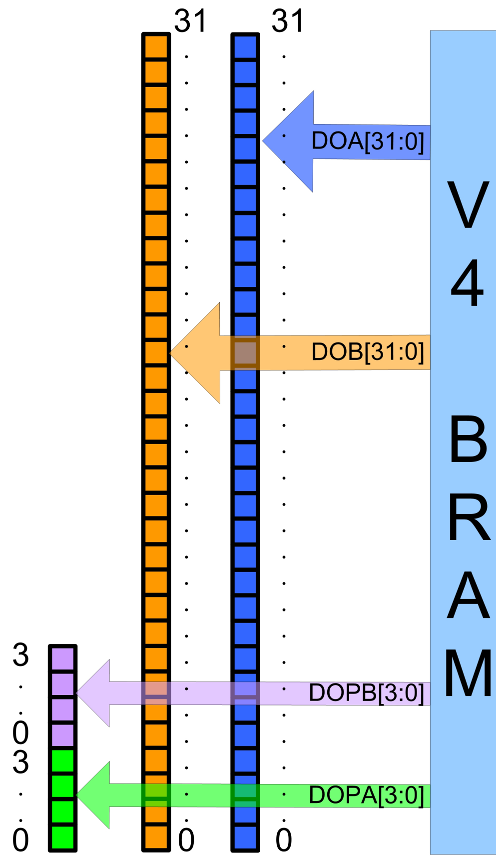

Figure 3.2: BRAM ORA Orientation

The FX and SX family of Virtex 4 devices require special ORA placements. In all SX devices there are several columns of BRAMs that do not have four consecutive columns of PLBs. In these devices a row of DSP modules bisects the 4 columns of PLBs leaving two columns on each side. As illustrated in Figure 3.3, this case requires that the ORAs comparing the parity bit outputs of each BRAM be shifted by one column. In FX family devices there are two exceptions that must be handled. The first problem is the presence of either one or two PPC modules. For BRAMs located in columns with PPC modules, these BRAMs must correctly send their outputs to their directly adjacent ORAs and the

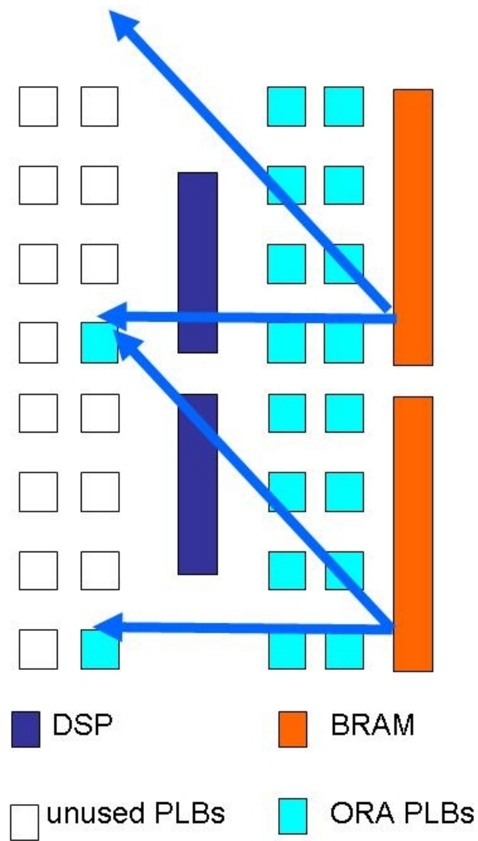

Figure 3.3: ORA Placement and Comparison in SX devices

ORAs in the same column above the PPC module. The other exception is only applicable to FX40, FX100, and FX140 devices wherein a column of BRAMs directly to the right of a PPC module has only a single column of PLBs in between the BRAM column and the PPC module. Instead of straddling ORAs across the PPC, the entire set of ORAs is shifted to the left of the PPC module. Figure 3.4 illustrates the ORA placement and circular comparison modification for BRAM BIST.

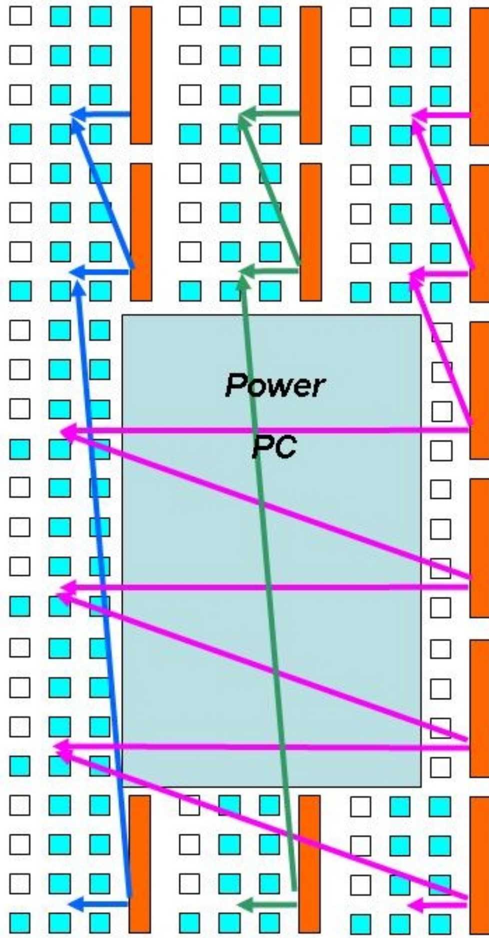

Figure 3.4: ORA Placement and Comparison in FX devices

3.2 TPG Development

Unlike, the approach used by Garimella in [17], the BIST approach for Virtex 4 can take advantage of partial reconfiguration. By being able to algorithmically place and route all BRAMs and ORAs identically for each BIST configuration, the only change that must be made between BIST configurations is to modify the BRAMs' configuration. In order for each BIST configuration to differ only by changes to the BRAMs, the TPG must be static

throughout all BIST configurations. These two modifications are the most different aspects of BIST developed for Virtex 2 by Garimella and the BIST presented in this thesis.

Garimella's BIST architecture used different TPGs, with each TPG implementing a single march test in each BIST configuration [17]. This approach was a logical choice since each BIST configuration was developed exclusively using VHDL. The BIST architecture developed for Virtex 4, however, requires a much more complicated TPG. In order for the TPG to remain static between all BIST configurations, it needs to be able to generate all of the required test patterns for all configurations. Garimella used March LR for all single-port configured BRAMs. March s2pf and d2pf were used to test dual-port functionality. March LR was also converted to incorporate BDS and applied to BRAMs configured in 512 x 36 mode of operation. Table 3.1 summarizes the march test selection for testing BRAMs in this thesis. March LR with BDS ensures that most faults are detected in the memory array. March LR with BDS is run on a BRAM configured in its widest aspect ratio in order to maximize the detection of intra-word faults with BDS and to minimize the number of BIST clock cycles required to run the test. The regular BOM-based March LR cannot be applied to a BRAM configured in a 16K x 1-bit mode because this mode does not utilize 2K of the parity memory space. March s2pf and d2pf are also used to detect the specific dual-port faults discussed in the previous chapter. MATS+ is used to test the remaining BRAM size configurations to detect faults within the programmable row and column decoders.

All four march tests were incorporated into a single multi-march TPG (MMTPG) and implemented in VHDL. The MMTPG VHDL source code is given in Appendix A. Since the Virtex 4 BRAM has programmable active levels for each of its control inputs, the MMTPG must be able to allow inverting active levels of the march tests driving the BRAMs. In order

Table 3.1: Virtex 4 TPG March Test Algorithms

March Test	Address Locations (A)	Data Width	Clock Cycles	MMTPG Op Mode[3:0]
March LR with BDS	512	36	2*58*A	000
s2pf	512	36	14*A	011
d2pf	512	36	9*A	100
MATS+	16K	1	2*5*A	010
	8K	2	2*5*A	001
	4K	4	2*5*A	110
	2K	9	2*5*A	111
	512	36	2*5*A	101

to control the march test selection and appropriate active levels, a control shift register was added to the MMTPG. The control shift register was connected to one of four boundary scan modules in a Virtex 4. Figure 3.5 summarizes the control function of each bit in the control register. For example, if "000000" was shifted into the control register, this control string would direct the MMTPG to execute March LR with BDS assuming the BRAM was configured for active low WE, SSR, REGCE, and EN control signals. Similarly, if "101010" was shifted into the shift register (beginning with the zero), that would select MATS+ with the BRAM configured to have an active high WE, EN, and SSR and an active low REGCE. For the single-port march tests, March LR with BDS and MATS+, each of these march tests run twice; once through port A and then again through port B, after which the TPG repeats the sequence. As seen in Table 3.1, this doubles the required number of clock cycles to run each of the march tests. By designing the MMTPG to generate several march tests along with programmable active levels, the MMTPG is able to apply appropriate march tests to any size configured Virtex 4 BRAM.

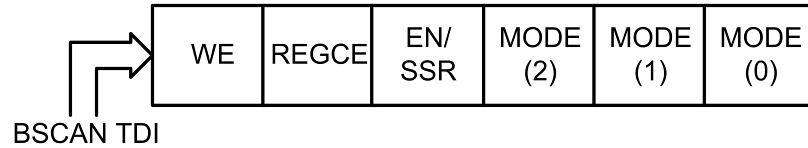

Figure 3.5: MMTPG Control Shift Register

3.3 BRAM BIST Configurations

Table 3.2 summarizes all of the BRAM BIST configurations. By the end of the last BIST configuration, all active levels and additional configuration options have been applied and tested. Two March LR configurations are listed, each with different initialization values. March LR (Init A) initializes the BRAMs contents to all alternating ones and zeros beginning with a zero in the list significant bit of BRAM. March LR (Init B) initializes the BRAMs to the opposite values. Each ports' SR values are configured to have values opposite of the BRAMs' contents such that when the SSR signal is asserted, the output response makes a transition. To test initialization values, a BRAM must be configured as READ_FIRST which means an addressed memory cell outputs its contents before being overwritten. Table 3.3 gives the initialization values for each BIST configuration. After each March LR configuration, the initialization values remain constant such that when generating partial configurations, the number of configuration bits is reduced. Dual-port testing using march s2pf and d2pf is accomplished with one configuration; however, to select between the two march tests, a separate MMTPG control register value is applied. BRAM BIST does not test BRAMs configured in either the 4K x 4-bit or the 2K x 9-bit memory aspect ratios. While the MMTPG has the ability to generate tests for these memory sizes, the FIFO mode of operation also can be configured in these two memory aspect ratios.

Table 3.2: BRAM BIST Configuration Detail

Config Num.	March Test	Read_Width [A,B]	WRITE_WIDTH [A,B]	DOA_REG	Invert DO Register CLK	RAM_EXTENSION [A,B]
1	March LR (Init A)	36	36	0	FALSE	NONE
2	March LR (Init B)	36	36	0	FALSE	NONE
3	s2pf / d2pf	36	36	1	FALSE	NONE
4	MATS+	1	1	1	FALSE	NONE
5	MATS+	8	8	1	TRUE	NONE
6	MATS+	512	512	1	FALSE	NONE
Config Num.	Write Mode	REGCE	SSR	WE	EN	CLK
1	READ_FIRST	High	High	High	High	High
2	READ_FIRST	High	High	High	High	High
3	READ_FIRST	High	High	High	High	High
4	READ_FIRST	High	High	High	High	High
5	NO_CHANGE	Low	Low	Low	Low	High
6	WRITE_FIRST	Low	Low	Low	Low	Low

Table 3.3: BRAM Initialization Values

Config Num.	March Test	Memory Init Value	Srval	Output Latch Init
1	March LR (Init A)	1010	0101	1010
2	March LR (Init B)	0101	1010	0101
3	s2pf / d2pf	0101	1010	0101
4	MATS+	0101	1010	0101
5	MATS+	0101	1010	0101
6	MATS+	0101	1010	0101

BIST Configuration Development

Developing BIST configurations that maximize the use of partial reconfiguration requires that only the BUTs' configuration changes between subsequent configurations. BRAM BIST for Virtex achieves this by algorithmically placing ORAs and setting aside dedicated logic to implement the MMTPG. However, implementing the required architecture violates many of the design rules defined by Xilinx. All inputs and outputs of a BRAM must be connected at all times such that all possible BRAM modes can be tested without adding modifying signals and altering the routing. Normally, Xilinx's CAD tools tie any unused input to BRAM to a global logic '1'. In order to meet this convention, the MMTPG complies with this design rule by driving any unused input to a '1' for a given configuration. For example, if the BRAM is configured in a 16K x 1 mode, then the MMTPG drives both DI ports with a '1' except for the least significant bit.

The design flow of BIST configurations for PLBs presented by Dhingra in [34] can be modified to generate BRAM BIST configurations. Instead of the high-level HDL design methodology used by Garimella in [17], Dhingra uses a low-level vendor specific design language called Xilinx Design Language (XDL). XDL is a human readable description of the physical placement and routing of a design in a Xilinx FPGA. XDL allows one to have

the utmost control over all aspects of a FPGA design, especially when developing BIST configurations. Dhingra used XDL to describe the placement of ORAs, BUTs, and TPGs used in testing PLBs. This design process must be modified to allow for the creation of BRAM BIST configurations. For a TPG, Dhingra used a pseudo-random pattern generator built from a Virtex 4 DSP module that continuously accumulated a prime number [34]. Dhingra's TPG only required the XDL instantiation of a DSP module and then made logical routing connections to the PLBs under test. Compared to Dhingra's TPG, the MMTPG is much larger, requiring 531 slices. Dhingra inserted the XDL TPG instantiation into a program that generated BIST configurations. However, inserting a 531-slice MMTPG instantiation into a similar tool for BRAM BIST is impractical because any slight change of the MMTPG during development would require modifying the program source code.

To overcome this problem, a TPG parsing tool, *V4BRAMTPG*, was developed that accepts an XDL version of the HDL synthesized MMTPG. The parsing program removes all instantiated components except the slices implementing the MMTPG. In order for the MMTPG to synthesize without failing design rule checks by Xilinx's CAD tools the TPG was connected to a dummy BRAM, which the parsing tool also removes. The next program created is called *V4BRAMBIST*. This program instantiates BRAMs, ORAs, and the parsed XDL MMTPG. *V4BRAMBIST* also generates all of the logical routing for the entire BIST configuration. When the program processes the MMTPG, it duplicates the TPG logic in order to implement the two required MMTPGs. During synthesis and implementation, the MMTPG is constrained in a Virtex 4 FX12 to fit in the first four columns of PLBs to the left of the center line. This placement allows the XDL version of the MMTPG to be compatible with all Virtex 4 devices. *V4BRAMBIST* shifts the slice coordinate of each MMTPG slice

such that it is always aligned to the four columns of PLBs directly to the left and right of the center line. Figure 3.6 and 3.7 show a logically connected (unrouted) BRAM BIST configuration in an FX12 and LX25 device, respectively.

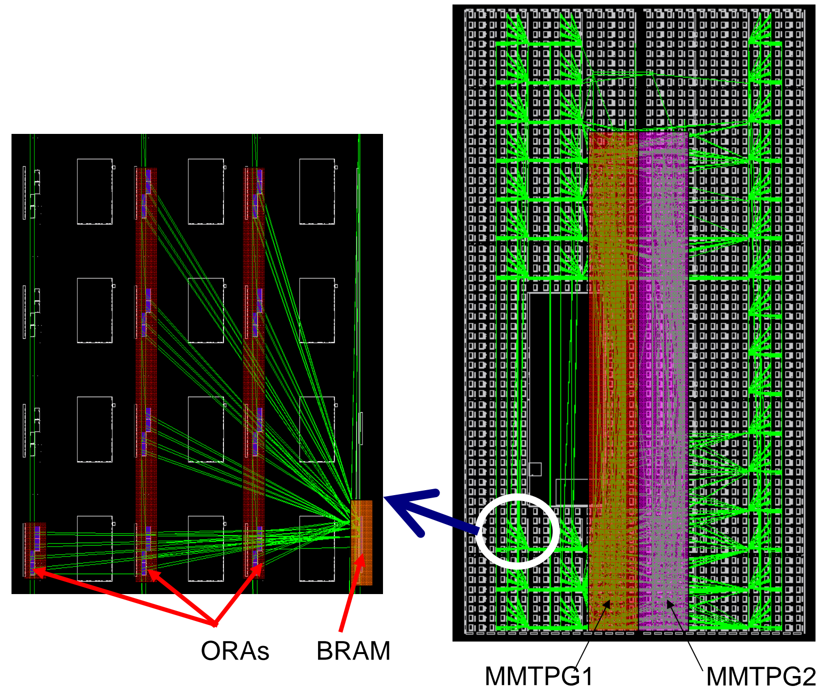

Figure 3.6: FX12 BRAM BIST

To control each BIST configuration, V4BRAMBIST instantiates two BSCAN modules: one configured as USER1 and the other as USER2. The USER1 BSCAN module uses TDI to reset the TPG and TCK to apply clock cycles to BRAMs, ORAs, and the two MMTPGS. The USER2 BSCAN module uses its TDI and TCK to serial shift control data into each of the MMTPG control registers. Two different BSCAN modules are used such that when changing the MMTPG mode, the rest of the BIST circuity is clock inhibited. This prevents

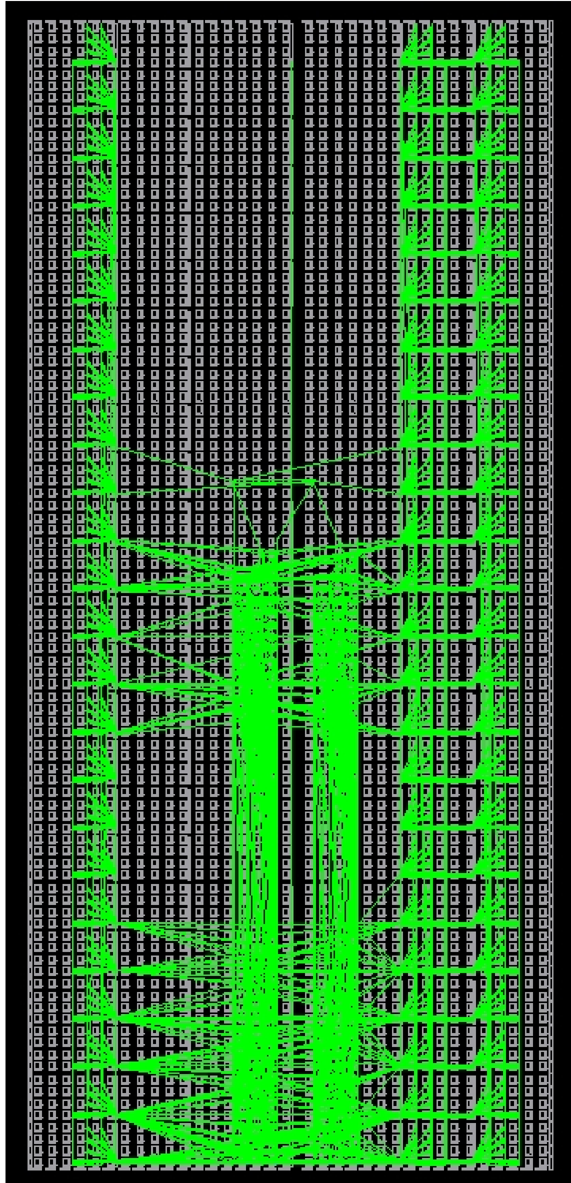

Figure 3.7: LX25 BRAM BIST

the BIST circuitry from going in into an unknown state that could occur as the control string is shifted into the control register.

The output of *V4BRAMBIST* is a logically connected (unrouted) XDL file that is a template for all BRAM configurations. This XDL file must be converted to an NCD file type such that Xilinx's place and route tool (PAR) can operate on the file. The template design must then be modified to configure all instantiated BRAMs such that their configuration corresponds to the settings shown in Table 3.2. These modifications are performed automatically by another program called *V4BRAMMOD*. *V4BRAMMOD* retains the current physical routing between configurations which is necessary to reduce the amount of partial configuration bits. With the combination of Xilinx's CAD tools, *V4BRAMTPG*,*V4BRAMBIST*, and *V4BRAMMOD* all six BIST configurations can be generated. The following procedure outlines the process of generating a set of BRAM BIST configurations:

1. Synthesize VHDL TPG
2. Place and Route TPG in FX12 with TPG constrained to the first four columns of PLBs to the left of the center column
3. Convert TPG to XDL format using *XDL* with the -ncd2xdl -nopips -nocom -cfg_brief arguments (See Table 3.4 for a summary of XDL arguments).
4. Run *V4BRAMTPG* to parse and extract TPG
5. Build the BRAM BIST template with *V4BRAMBIST*.
6. Convert BRAM BIST template to NCD format using *XDL* with the -xdl2ncd -force -nodrc arguments.

Table 3.4: XDL Argument Summary

XDL argument	Description
-ncd2xdl	Selects conversion from NCD to XDL
-xdl2ncd	Selects conversion from XDL to NCD
-nopips	Routing is removed from NCD when converted to XDL
-nocom	XDL file will not contain comment blocks
-cfg_brief	Unused configuration options are not listed during NCD to XDL conversion
-force	Force conversion of XDL to NCD despite design rule errors
-nodrc	Disables design rule checking during XDL to NCD conversion

7. Fully route the BIST template using *PAR*.

8. Convert BIST template to XDL format using *XDL* with the -ncd2xdl -nocom -cfg_brief arguments.

9. Run V4BRAMMOD for each of the six BRAM BIST configurations.

10. Convert each BIST configuration to NCD format using *XDL* with the -xdl2ncd -force -nodrc arguments.

11. Generate configuration bit-files for each BIST configuration using *BITGEN*.

Figures 3.8, 3.9, and 3.10 show the command-line options available in V4BRAMTPG, V4BRAMBIST, and V4BRAMMOD, respectively. Each BRAM BIST configuration can be constrained to test a subset of BRAMs in a device as long as there are four BRAMs in each circular comparison ORA chain. Figure 3.11 demonstrates the use of these tools to create the BIST configuration shown in Figure 3.12. Testing a subset of BRAMs in a device can be used to increase the timing performance of a configuration and also can lower the BIST power consumption.

Three different types of configuration bitstreams can be generated using *BITGEN*: full, compress, and partial. Full configuration bit-files contain frame data for every addressable

```
V4BRAMTPG.exe
V4BRAMTPG processes XDL from synthesis for use in XDL generation program\\
command line format:\\
V4BRAMTPG <XDLin_file> <XDLout\_file>}
notes: assumes input XDL file generated with 'xdl -ncd2xdl -cfg_brief -nocom'}
```

Figure 3.8: V4BRAMTPG Syntax

```
V4BRAMBIST.exe
V4BRAMBIST - generates template file for BRAM BIST config in any Virtex 4
command line format:
V4BRAMBIST <xdlfile> <startrow> <startcol> <endrow> <endcol> <dev> <part>
 <tpgXDLfile>
   dev part rows cols   dev part rows cols   dev part rows cols
    lx   15   64   31    sx   25   64   55    fx   12   64   31
    lx   25   96   35    sx   35   96   55    fx   20   64   47
    lx   40  128   43    sx   55  128   69    fx   40   96   65
    lx   60  128   61                         fx   60  128   67
    lx   80  160   65                         fx  100  160   85
    lx  100  192   73                         fx  140  192  103
    lx  160  192   98
    lx  200  192  127
```

Figure 3.9: V4BRAMBIST Syntax

```
V4BRAMMOD.exe
V4BRAMMOD - modifies routed XDL for BRAM BIST
command line format:
V4BRAMMOD <xdl_in> <xdl_out> <phase>
   where phase = MLRA,MLRB,DUALP,8K,16K,MATS512}
```

Figure 3.10: V4BRAMMOD Syntax

```
V4BRAMTPG TPG.xdl parsedTPG.xdl
V4BRAMBIST BRAM\_lx60 64 32 128 61 lx 60 parsedTPG.xdl
V4BRAMMOD BRAM_lx60.xdl BRAM_lx60_DUALP.xdl dualp
```

Figure 3.11: Example BIST program execution

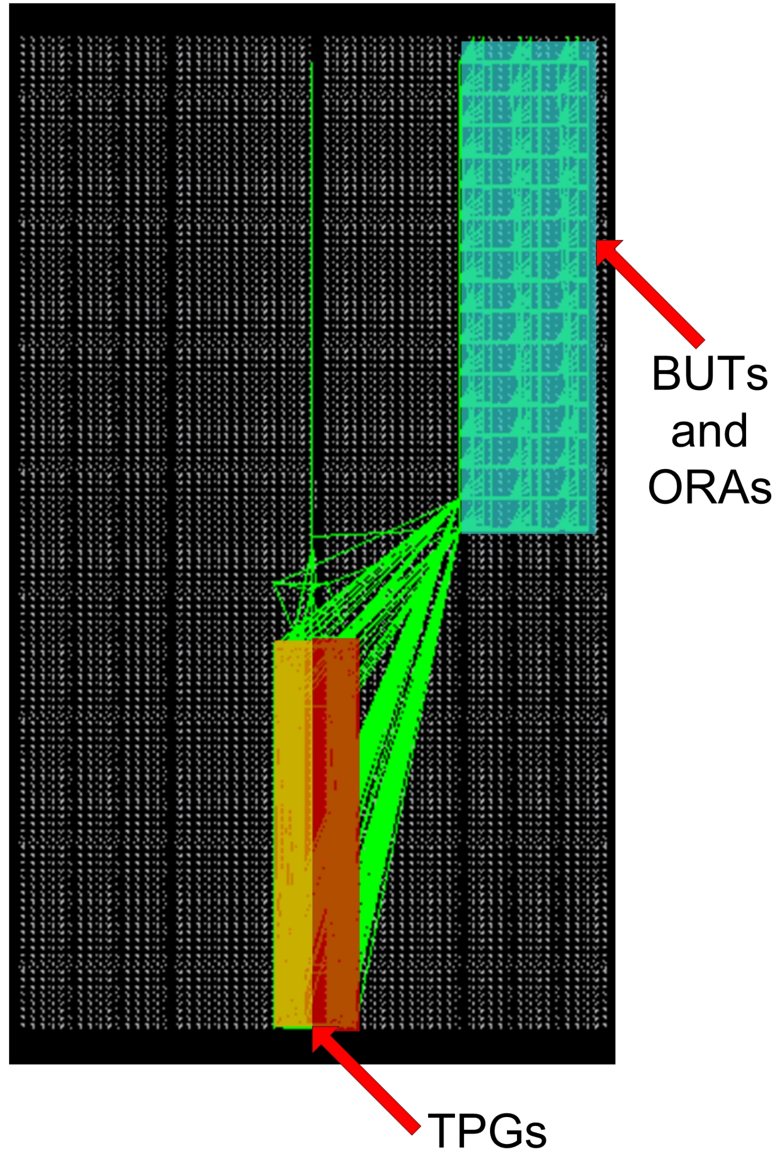

Figure 3.12: Partial BRAM BIST in LX60

frame in a given device. Full configurations are the default configuration bit-file type for *BITGEN*. Executing *BITGEN* with the '-g Compress' argument allows *BITGEN* to take advantage of the multi-frame write capabilities that are usually used when generating partial configuration files. To create a partial configuration bit-file, the '-r previousconfig.bit' argument must be used. In the previous argument, 'previousconfig.bit' is used as a reference for generating the partial configuration bit-file.

One of the most time expensive portions of generating BIST configurations is routing BIST configurations. Garimella's approach for generating BRAM BIST configurations also required the synthesis of the entire BIST architecture as well as the placement and routing. These three processes were executed for each configuration. For Virtex 4, however, only the synthesis of the TPG is required and only one configuration must be routed. As mentioned earlier, each subsequent configuration retains the same routing.

3.4 Running BIST Configurations

Once all of the configurations have been generated using the aforementioned BIST tools, the configurations can be downloaded to the device via boundary scan. Table 3.5 lists the six BRAM BIST configurations and the MMTPG control string needed to configure the TPG along the the number of BIST clock cycles needed to run each march test to completion. The three least significant bits of each control string define the march test for the MMTPG as also shown in Table 3.1. A procedure for running BRAM BIST configurations is as follows:

1. Download BRAM BIST configuration to device.

2. Goto USER2 access register.

Table 3.5: BRAM BIST Execution Detail

Config Num.	March Test	MMTPG Control String [MSB:LSB]	BIST Clock Cycles
1	March LR (Init A)	111000	60,000
2	March LR (Init B)	111000	550
3a	s2pf	111011	7,200
3b	d2pf	111100	5,000
4	MATS+ (16K)	111010	165,000
5	MATS+ (8K)	000001	82,000
6	MATS+ (512)	000101	5,500

Total BRAM BIST Clock Cycles = 325,250

3. Clock in MMTPG control string LSB first and assert TMS on the MSB.
4. Goto USER1 access register.
5. Toggle TDI to reset MMTPG (Active high asynchronous reset).
6. Apply BIST clock cycles.
7. Retrieve ORA results via configuration memory readback.
8. Repeat steps 1-7 for each addition configuration.

Performing a configuration memory readback at the end of each BIST configuration can indicate the mode of failure for a BUT. Delaying the configuration memory readback until the last BIST configuration shortens the time required to perform BIST at the expense of diagnostic resolution due to an uncertainty in the mode of failure recorded.

3.5 ORA Results Retrieval

The *CAPTURE* module is instantiated to transfer ORA flip-flop contents to the configuration memory for configuration memory readback. Xilinx's Virtex 4 Configuration

Guide [2] provides a procedure for reading specific frames of configuration memory through boundary scan. It is important to point out that the readback flip-flop data is inverted. During configuration bit-file creation, a the '-l' *BITGEN* argument creates a logic allocation file that reports the configuration memory frame bit associate with each ORA. Retrieving ORA results is efficient since all of the flip-flop contents for a column of 16 PLBs are located within a single frame. Since four BRAMs span the height of 16 PLBs and each set of ORAs for a single BRAM are contained in three PLB columns, only 3 frames are read for each column of four BRAMs.

3.6 BIST Results

Previously in [17], Garimella calculated the total number of BIST clock cycles for Virtex 2 BRAMs to be 485,888. From Table 3.5, the total BIST clock cycles needed for Virtex 4 BRAMs are 325,250 which represents an overall savings of 160,638 clock cycles. Table 3.6 summarizes the configuration time required to test an LX60 device. In Table 3.6 the configuration bit-file sizes and associated download and test times are given for an LX60. The test clock frequency of 50 MHz is used because it is the maximum BSCAN clock frequency supported in all Virtex 4 devices [3].

Table 3.6 also compares three different BIST download techniques: full, compressed, and partial reconfigurations. In full configurations, all addresses are written with frame data while the compressed technique allows for a reduction in configuration file size by using the multi-frame write capabilities. The partial configurations are generated from a set of either full or compressed configurations and further reduces the configuration bit-file size by only writing frame data that differs between two given configurations. Figure 3.13 illustrates the

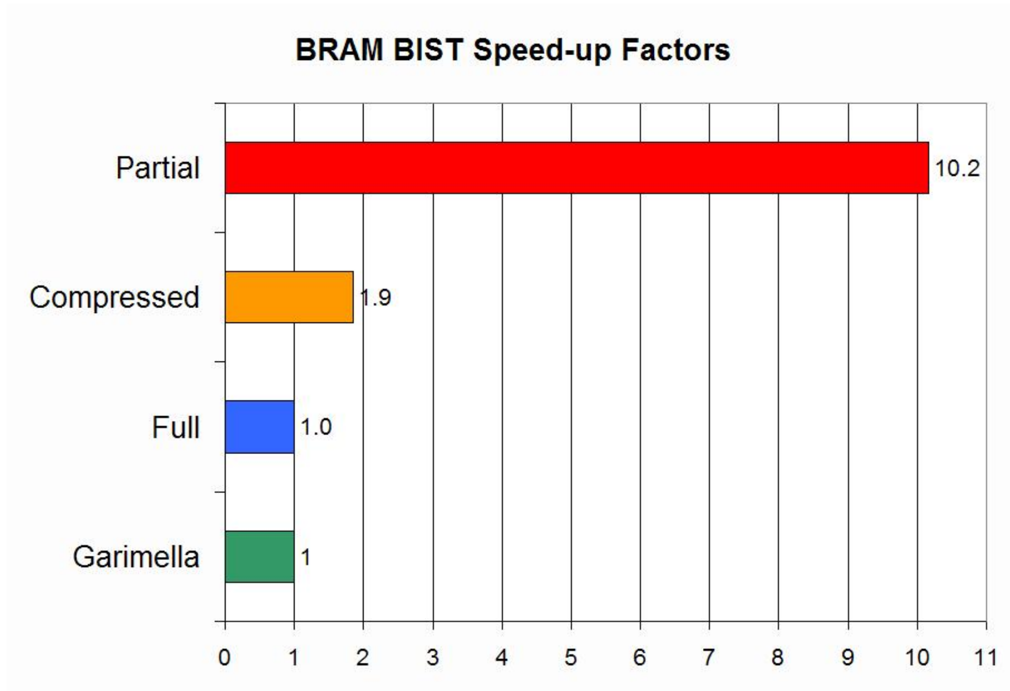

Figure 3.13: LX60 BRAM BIST Speed-up factors

BIST speed increase associated with using compress and partial configuration techniques over full configurations. Garimella's BIST configurations for Virtex 2 BRAMs can only be compared to full BRAM BIST configurations in Virtex 4 as also seen in Figure 3.13. It is clear that the BIST architecture for Virtex 4 is superior to that used in Virtex 2.

Figures 3.14 and Figure 3.15 summarize the timing analysis of each BIST configuration for several Virtex 4 devices. The last configuration "512" in Figure 3.15 refers to the MATS+ 512 x 36-bit configuration and is approximately one-half as fast as the rest of the BRAM BIST configurations. This is due to the BUTs' inverted clock input which acts to halve the available propagation delay. However, this configuration is necessary in order to test BRAMs configured for falling-edge triggered operation.

Table 3.6: Summary of LX60 BRAM BIST Download Size and Test Times

BIST Configuration	Download Size (Bits)			Configuration Time @ 50 Mhz (seconds)		
	Full	Compressed	Partial	Full	Compressed	Partial
March LR (Init A)	17,717,632	9,516,672	9,516,672	0.354	0.190	0.190
March LR (Init B)	17,717,632	9,516,672	446,688	0.354	0.190	0.00893
s2pf/d2pf	17,717,632	9,516,672	24,032	0.354	0.190	0.000481
MATS+ (16K)	17,717,632	9,516,672	24,032	0.354	0.190	0.000481
MATS+ (8K)	17,717,632	9,516,672	122,688	0.354	0.190	0.00245
MATS+ (512)	17,717,632	9,516,672	24,032	0.354	0.190	0.000481
TOTAL	106,305,792	57,100,032	10,158,144	2.13	1.14	0.203

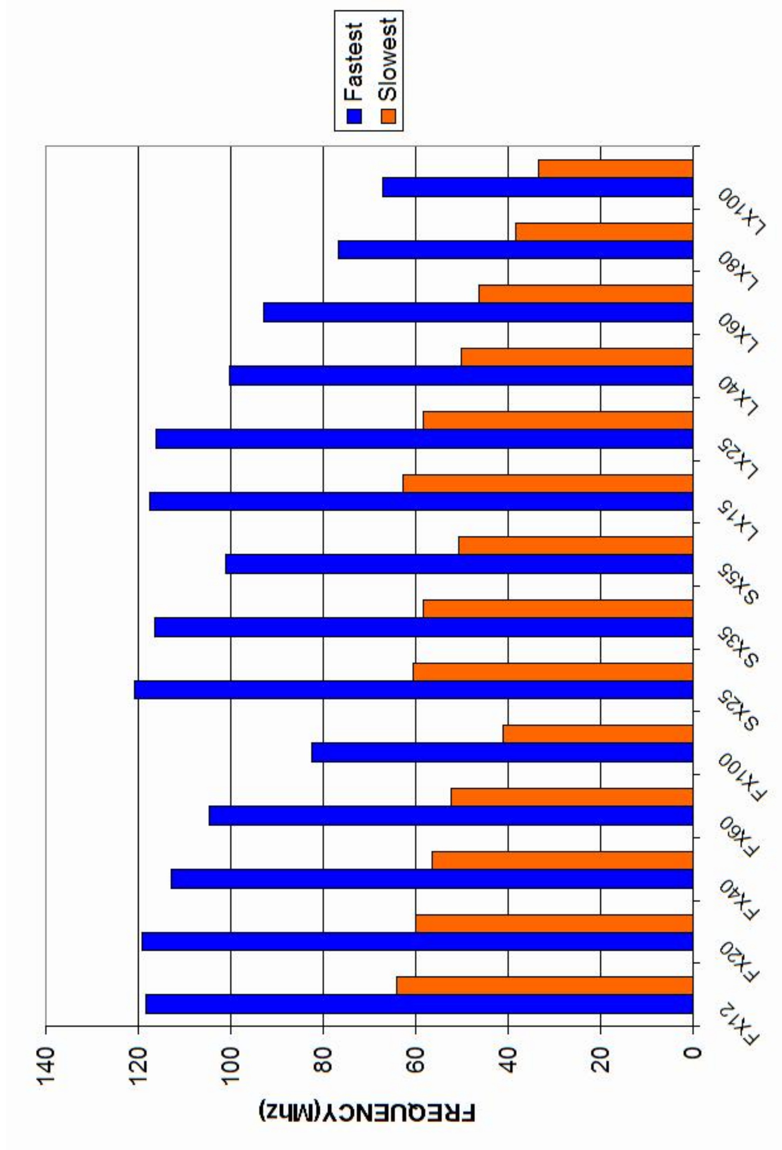

Figure 3.14: Timing Analysis (Slowest / Fastest)

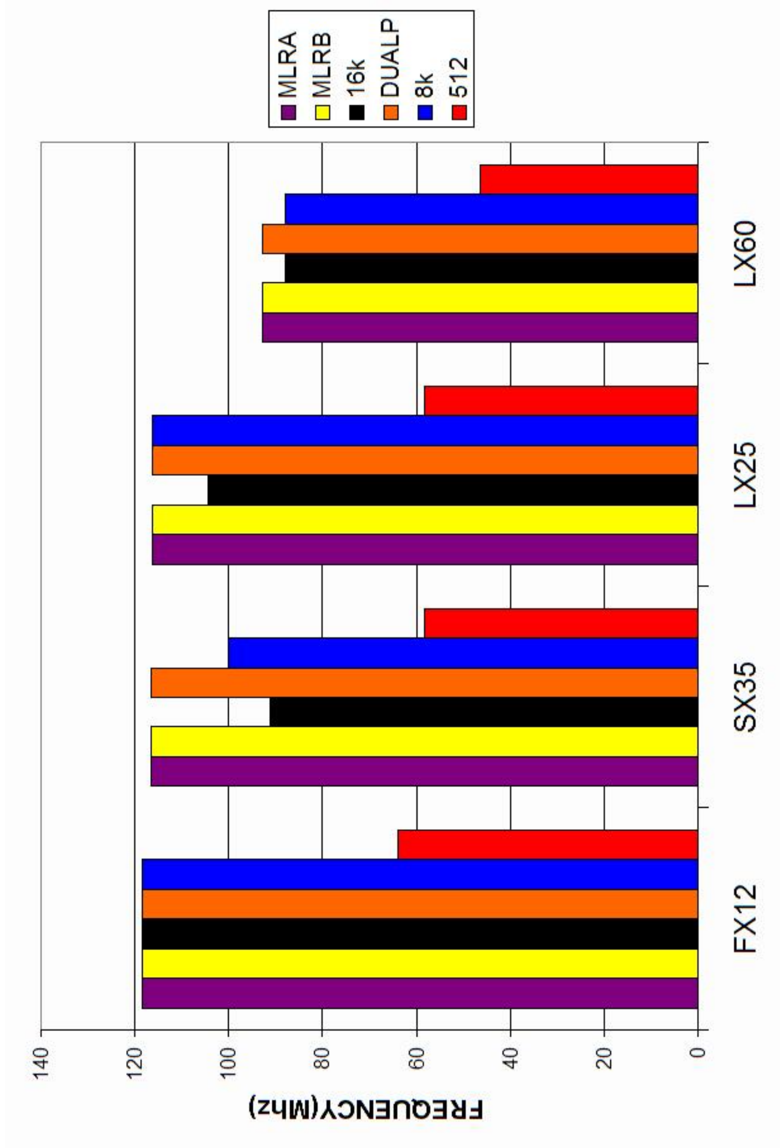

Figure 3.15: Timing Analysis per BRAM BIST Configuration

3.7 BRAM BIST Summary

This chapter has described a BIST architecture for testing BRAMs in Virtex 4 FPGAs. The architecture consists of BRAMs tested by a two TPGs driving alternating rows of BRAMs. A circular comparison based ORA architecture was implemented such that each column of BRAMs formed a separate circular comparison chain. Each TPG in a BRAM BIST configuration can apply multiple march tests depending on a TPG control register that communicates the current configuration of each BRAM such that an appropriate march test is applied.

To implement the BRAM BIST architecture, several BIST programs were also developed to facilitate generating BRAM BIST configurations for any Virtex 4 device. As shown in the following chapters, these programs will be modified to support BIST for BRAMs operating in FIFO, ECC, and cascade modes of operation.

Chapter 4

Virtex 4 FIFO BIST Implementation

A BIST approach developed for BRAMs configured in a FIFO mode of operation in Virtex 4 FPGAs is presented in this chapter. The BIST architecture will be discussed as well as a TPG for FIFO testing. Finally, results from applying BIST to Virtex 4 devices are given.

4.1 Virtex 4 FIFO BIST Architecture

Each Virtex 4 BRAM can operate in a FIFO mode of operation which allows for the same number of BUTs in both FIFO and BRAM BIST architecture. Also, the overall BRAM BIST architecture developed in the previous chapter can be applied to FIFO BIST. The logical connections from TPG to BUT and BUT to ORA must be modified slightly because each FIFO has its own dedicated inputs and outputs. While there are 72 outputs per BRAM, each FIFO has only 66 outputs. Fewer BUT outputs translates into fewer ORAs per BUT. Figure 4.1 illustrates the location of the BUT output signal comparisons by the ORAs. As with BRAM BIST, 9 PLBs are still required for FIFO BIST ORAs, but in the ninth PLB, only a single slice is used. In addition, the ORA placement exceptions for SX and FX device families discussed in Chapter 3 also are present in FIFO BIST configurations.

4.2 FIFO TPG Development

In [35], a test algorithm is described for Atmel FIFOs without programmable *ALMOSTFULL* and *ALMOSTEMPTY* status flags. The test algorithm of length 6N (where

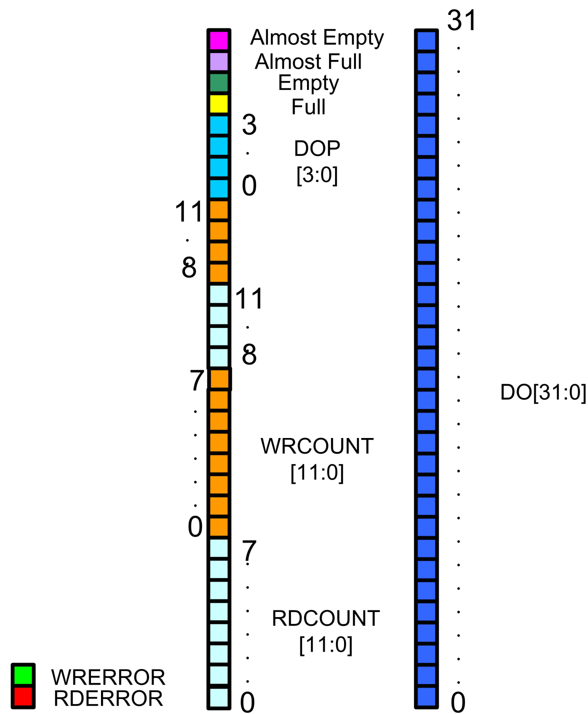

Figure 4.1: FIFO ORA Placement

N is the number of address locations) is given below.

Atmel FIFO Test Algorithm (Test Length = 6N) [35]:

Step 1: Reset the FIFO

Step 2: Repeat N times: Write a word with all zeros and observe the *FULL* flag assertion after N writes and the *EMPTY* flag deasserts after the first word written.

Step 3: Repeat N times: Read a word with all zeros and then write a word with all ones. The *FULL* should toggle in between each read and write operation.

Step 4: Repeat N times: Read a word with all ones and then write a word with all zeros. The *FULL* should toggle in between each read and write operation.

Step 5: Repeat N times: Read a word expecting all zeros and observe the *EMPTY* flag assertion after N reads and the *FULL* disasserts after the first read word.

Steps 2 and 3 repeatedly read and write to the FIFO which fully test the FIFO read and write pointers by walking each pointer through the entire memory space. A comparison of read and write pointers along with the previous operation determines if the *EMPTY* or *FULL* flag asserts. The above test also ensures opposite logic values are written and read to each memory location.

In [36], the Atmel FIFO algorithm was generalized to test FIFOs with programmable *ALMOSTFULL* and *ALMOSTEMPTY* status flags. These additional status flags are tested in Step 2 by repeatedly reconfiguring the *ALMOSTFULL* flag from its minimum to its maximum value while continuing to write and read data and observing the *ALMOSTFULL* flag toggle after each partial reconfiguration. The *ALMOSTEMPTY* flag can be tested in Step 5, during which the *ALMOSTEMPTY* flag is repeatedly reconfigured from its maximum to its minimum allowed value while the FIFO is emptied with N reads. Also during Step 5, the *ALMOSTEMPTY* flag will toggle after each reconfiguration.

For testing Virtex 4 FIFOs, it is not critical that opposite logic values are both written and read to and from each FIFO data word. March LR with BDS applied during BRAM BIST ensures the memory array is fault-free which allows the Virtex 4 FIFO test algorithm to concentrate on testing the status logic and read and write pointer logic. In Chapter 2, Table 2.7 summarizes the timing for FIFO status flag assertion and deassertion. The deassertion period for *FULL* and *EMPTY* flags is at most 4 clock cycles when in the FWFT mode. This latency becomes problematic for FIFO test algorithms. Steps 3 and 4 from the Atmel FIFO test algorithm are designed to test the *FULL* flag generation logic,

but if these steps were applied to a Virtex 4 FIFO, the *FULL* flag would not deassert in time for the next read and write operations. Fault coverage is reduced because the reassertion of the *FULL* flag would be masked since the *FULL* flag does not deassert until several clock cycles later.

In order to test the *FULL* and *EMPTY* flags in Virtex 4 FIFOs several clock cycles where no operation is performed are inserted between read and write operations. These no operation (NO-OP) clock cycles allow the *FULL* to deassert before the read and write sequence is repeated. The test algorithm for Virtex 4 FIFO testing is given below.

Virtex 4 FIFO Test Algorithm (Test Length = 8N):

Step 1: Reset the FIFO

Step 2: Repeat N times: Write a word with all zeros and observe the *FULL* flag assertion after N+2 writes and the *EMPTY* flag deassertion after the first word written.

Step 3: Repeat N times: Read a word with all zeros

NO-OP

NO-OP

NO-OP

Write a word with all ones

Write a word with all ones

Step 4: Repeat N times: Read a word expecting all ones

By inserting the 3 NO-OP clock cycles, the FULL flag deasserts before the write and then read sequence. The repeated write in in Step 3 asserts the write error (WRERR) flag for one clock cycle to test the logic associated with this error indication. Figure 4.2 shows

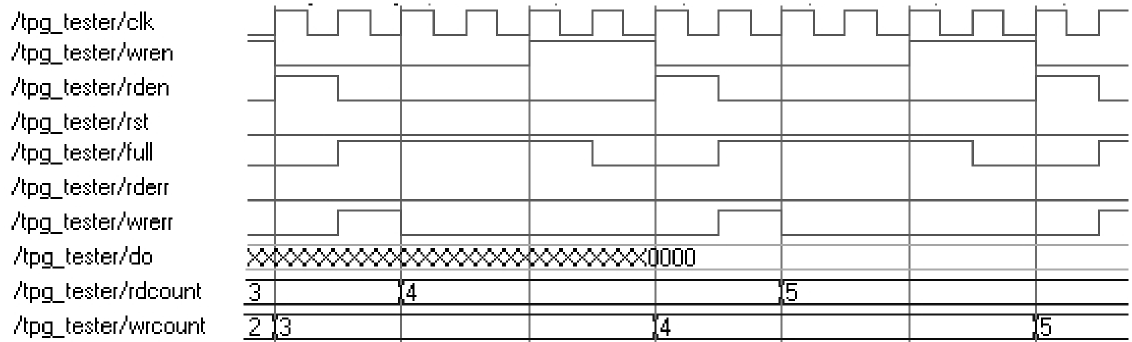

Figure 4.2: FULL Flag Transition Timing

a timing diagram to that describes the FIFO response to Step 3 in the Virtex 4 FIFO Test Algorithm.

A TPG implementing the above Virtex 4 FIFO test algorithm was implemented in VHDL and required 96 slices when implemented in a FX12 device. The TPG, named FIFOTPG, is able to generate the above test for the four different FIFO depth configurations. The VHDL source for the FIFOTPG is given in Appendix B. Like the BRAM MMTPG, the FIFOTPG is also able to invert the active level of the FIFO control signals so that any possible FIFO configuration can be tested with the FIFOTPG. In order to program the FIFOTPG for the current FIFO configuration, a TPG control register is used to communicate with the FIFOTPG. Figure 4.3 indicates the function of each bit in the shift register. The three most significant bits control the active levels of the the FIFOTPG control signals RDEN, WREN, and RST while the two least significant bits determine the operational mode that corresponds to one of the four configurable FIFO word depths. When writing to the control register, the control string value is shifted in, LSB first. The control string values for each FIFO BIST configuration is summarized in Table 4.1.

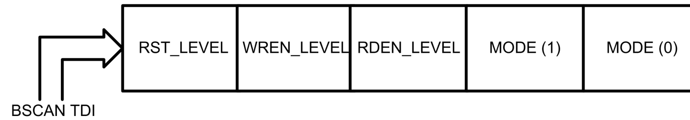

Figure 4.3: FIFOTPG Control Register

4.3 FIFO BIST Configuration Development

Three custom BIST generation tools were developed to enable FIFO BIST configuration generation for all Virtex 4 devices using the same procedure discussed in Chapter 3 Section 3. *V4FIFOTPG* parses an XDL version of the FIFOTPG developed in the previous section. *V4FIFOBIST* places all of the BIST circuitry into a specified device. Figure 4.4 shows a logically connected (unrouted) FIFO BIST configuration in a LX60 device. The third program, *V4FIFOMOD* modifies each FIFO BIST configuration such that the configuration options match those listed in Table 4.1. During each FIFO BIST configuration the FIFOTPG to FIFO routing and the FIFO to ORA routing is not changed. The FIFOTPG also remains static throughout all of the configurations. The only portion of the BIST configuration changed during each modification is the FIFO configuration options. The command-line options for these three programs are the same those shown for BRAMs in Figures 3.8 3.9 3.10.

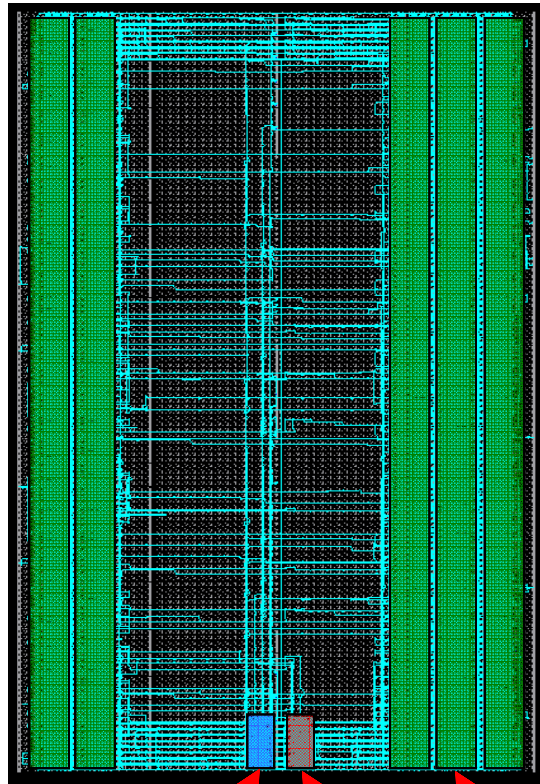

Figure 4.4: LX60 FIFO BIST Configuration

Table 4.1: Summary of Virtex 4 FIFO Configurations

Config Num.	FIFO MODE	RST, RDEN, WREN ACTIVE LEVEL	FWFT	ALMOST EMPTY LEVEL	ALMOST FULL LEVEL	FIFOTPG Control String [MSB:LSB]	BIST Clock Cycles
1	2K x 9-bits	LOW	TRUE	15	2,043	00001	16,384
2	512 x 36-bits	HIGH	FALSE	15	496	11111	4,096
3	1K x 18-bits	HIGH	FALSE	5	507	11110	8,192
4	4K x 4-bits	HIGH	FALSE	5	5	11100	7
5	4K x 4-bits	HIGH	FALSE	6	7	11100	8
6	4K x 4-bits	HIGH	FALSE	8	8	11100	10
7	4K x 4-bits	HIGH	FALSE	16	16	11100	18
8	4K x 4-bits	HIGH	FALSE	32	32	11100	34
9	4K x 4-bits	HIGH	FALSE	64	64	11100	66
10	4K x 4-bits	HIGH	FALSE	128	128	11100	130
11	4K x 4-bits	HIGH	FALSE	256	256	11100	258
12	4K x 4-bits	HIGH	FALSE	512	512	11100	514
13	4K x 4-bits	HIGH	FALSE	1,024	1,024	11100	1,026
14	4K x 4-bits	HIGH	FALSE	2,048	2,048	11100	2,050
15	4K x 4-bits	HIGH	FALSE	4,092	4,092	11100	32,768

TOTAL BIST CLOCK CYCLES = 65,561

The *ALMOST FULL* and *ALMOST EMPTY* flag values each are specified by 12 bits of configuration memory. The 4K x 4-bit operation mode is used in testing the *ALMOST FULL* and *ALMOST EMPTY* flags because it requires 12 bits to specify values up to 4K-bits. In order to test these configuration bits, FIFO BIST configurations 4 through 15 move both the *ALMOST FULL* and *ALMOST EMPTY* flags higher such that each of the 12 configuration bits undergo a transition from '1' to a '0'. Also, the number of BIST configurations is minimized by configuring the *ALMOST FULL* and *ALMOST EMPTY* flags to transition from the minimum allowed value in configuration 4 to the maximum allowed configuration in 15. This minimization is achieved because configurations 4-14 are designed to only test the *ALMOST FULL* and *ALMOST EMPTY* flags and for those configurations, only the number of clock cycles needed to reach the configured *ALMOST* values are applied. In configuration 15, 32,768 clocks cycles are applied to fully execute the FIFO test algorithm and to test the final *ALMOST FULL* and *ALMOST EMPTY* flags values.

4.4 Running FIFO BIST Configurations

Running each of the FIFO BIST configurations is similar to the procedure described for BRAM BIST in Chapter 3. A minor difference is that the FIFOTPG contains a 5-bit control register compared to the 6-bit control register in each MMTPG. The following procedure summarizes running each FIFO configuration:

1. Download FIFO BIST configuration to device.

2. Goto USER2 access register.

3. Clock in FIFOTPG control string, LSB first, and assert TMS on the MSB.
4. Goto USER1 access register.
5. Toggle TDI to reset MMTPG (Active high asynchronous reset).
6. Apply BIST clock cycles.
7. Retrieve ORA results via configuration memory readback.
8. Repeat 1-7 for each addition configuration.

4.5 FIFO BIST Results

Table 4.2 provides a summary of all fifteen FIFO configurations for a LX60. Due to the larger number of configurations, FIFO BIST benefits more from partial reconfiguration than BRAM BIST. While BRAM BIST gained a ten times speed-up factor over full configurations, FIFO BIST gained over a 32 times speed-up factor as seen in Figure 4.6. The number of BIST clock cycles for both BRAM and FIFO BIST is 390,811. This total still represents 95,077 less clock cycles than Garimella required for only testing Virtex 2 BRAMs. A FX12 FIFO BIST is shown in Figure 4.7. The two FIFOTPGs and their associated routing have been highlighted. The FIFOTPG to the left of the center column is highlighted green while the FIFOTPG to the right is highlighted in yellow. The FIFO to ORA routing is also highlighted in blue. By highlighting the FIFOTPG routing, one can see that each FIFOTPG is driving alternating rows of Virtex 4 FIFO modules. The timing analysis for each FIFO BIST configuration is shown in Figure 4.5. The 4K x 4-bit FIFO mode is only listed once because each of the 12 configurations have the same timing analysis result due to each FIFO being identically configured.

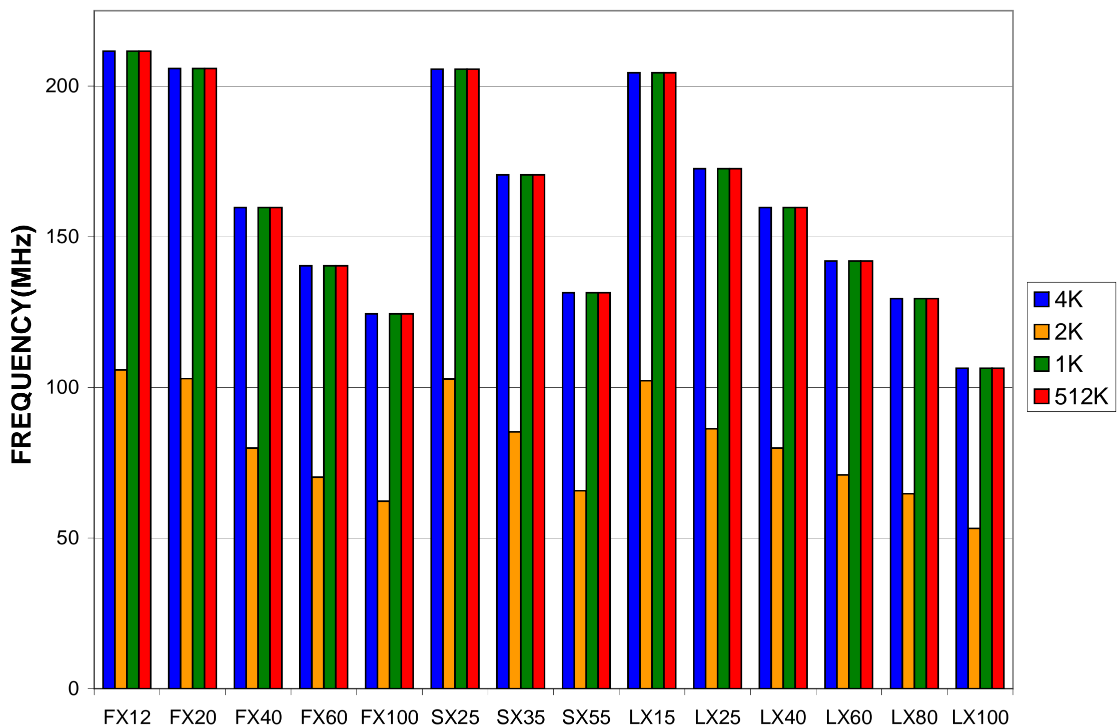

Figure 4.5: FIFO BIST Timing Analysis

Table 4.2: Summary of LX60 FIFO BIST Download Size and Test Times

BIST Configuration	Download Size (Bits)			Configuration Time @ 50 MHz (seconds)		
	FULL	Compress	Partial	FULL	Compress	Partial
1	17,717,632	7,777,728	7,777,728	0.354	0.156	0.1556
2	17,717,632	7,777,728	121,856	0.354	0.156	0.000464
3	17,717,632	7,777,728	23,200	0.354	0.156	0.000464
4	17,717,632	7,777,728	23,200	0.354	0.156	0.000464
5	17,717,632	7,777,728	23,200	0.354	0.156	0.000464
6	17,717,632	7,777,728	23,200	0.354	0.156	0.000464
7	17,717,632	7,777,728	23,200	0.354	0.156	0.000464
8	17,717,632	7,777,728	23,200	0.354	0.156	0.000464
9	17,717,632	7,777,728	23,200	0.354	0.156	0.000464
10	17,717,632	7,777,728	23,200	0.354	0.156	0.000464
11	17,717,632	7,777,728	23,200	0.354	0.156	0.000464
12	17,717,632	7,777,728	23,200	0.354	0.156	0.000464
13	17,717,632	7,777,728	23,200	0.354	0.156	0.000464
14	17,717,632	7,777,728	23,200	0.354	0.156	0.000464
15	17,717,632	7,777,728	23,200	0.354	0.156	0.000464
TOTAL	265,764,480	116,665,920	8,201,184	5.315	2.333	0.162

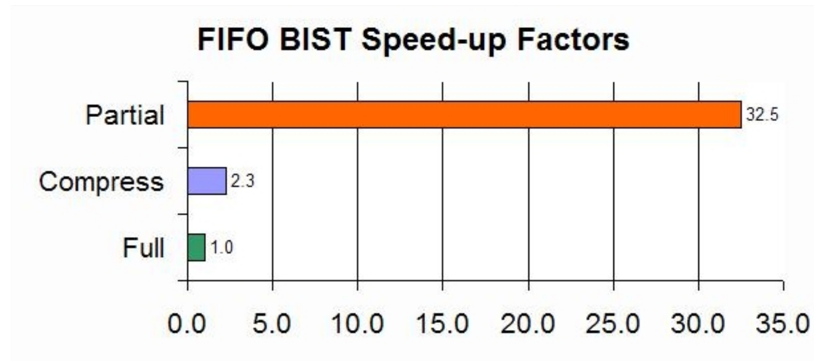

Figure 4.6: LX60 FIFO Speed-up Factors

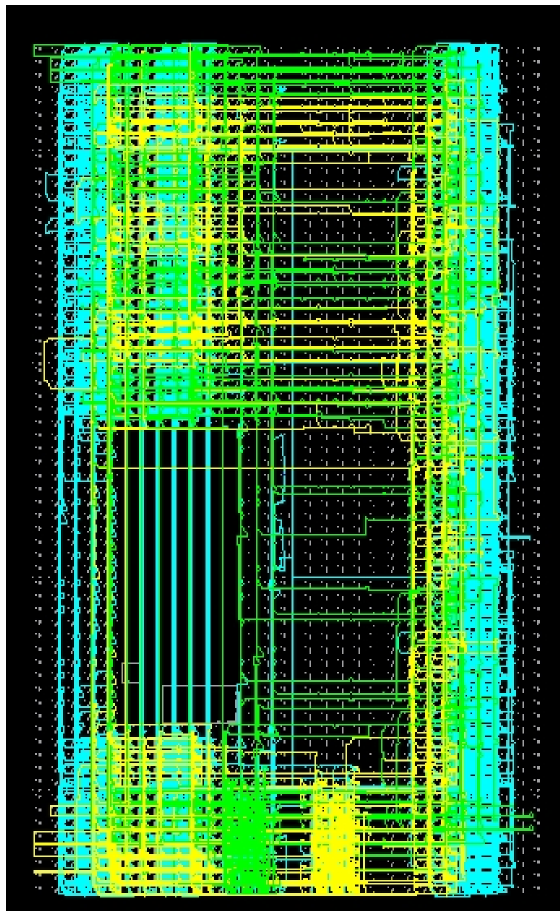

Figure 4.7: Routed FX12 FIFO BIST Configuration

Chapter 5

Virtex 4 ECC and Cascade BIST Implementation

A BIST approach developed for BRAMs configured in both ECC and cascade modes of operation in Virtex 4 FPGAs is presented in this chapter. The BIST architecture will be discussed as well as TPGs for ECC and cascade testing. Finally, results from applying BIST to Virtex 4 devices are given.

5.1 ECC and Cascade BIST Architecture

The BIST architectures presented in previous chapters considered a BUT to be a single BRAM. For ECC and cascade BIST configurations, the BUT is enlarged to encompass two adjacent BRAMs because both ECC and cascade modes require a pair of adjacent BRAMs. In order for the BUT to be a pair of BRAMs, the BIST architecture developed for BRAM and FIFO BIST is modified such that instead of TPGs driving alternating rows of BRAMs, the TPGs drive alternating pairs of BRAMs. BRAM to ORA routing is also modified such that the outputs of the lower BRAM pair is compared with the outputs of the next lower BRAM pair. The ORA comparison per BRAM output is identical to BRAM BIST. Figure 5.1 illustrates the configuration for the TPG to BUT and BUT to ORA connections.

All Virtex 4 devices contain an even number of BRAMs per column which allows for complete ECC and cascade BRAM pairs per column. The only exception to this rule is the in FX devices. For ECC BIST configurations, the BRAMs directly above and below each PPC module cannot operate in an ECC BRAM pair. As shown in Figure 5.1, ECC BRAM

pairs in Virtex 4 devices are fixed and the even numbered rows are always configured as the *LOWER* ECC BRAM.

For cascade configurations, problems arise when the cascade pair is separated by a PPC module. Cascaded BRAMs can be either cascaded with the BRAM directly above or below. Testing this attribute requires instantiating the bottom BRAM as a *LOWER* BRAM and continuing to alternate between *UPPER* and *LOWER* configured BRAM as illustrated in Figure 5.1. A second configuration is required to test BRAMs that were configured as an *UPPER* BRAM in the *LOWER* cascade mode of operation and vice versa. Due to the Virtex 4 BRAM cascade routing architecture, each cascade BIST configuration is expected to have ORA failures in a fault-free device. At the top of each BRAM column and directly below each PPC module, there are no cascade routes that wrap around to the bottom BRAM in a column or route through a PPC module to the next BRAM. This causes the bottom BRAM in a cascade pair located at the bottom of a BRAM column and also directly above a PPC to generate incorrect results when compared to the other BRAMs in a given configuration. The incorrect results stem from the cascade implementation as shown in Chapter 2, Figure 2.9. In this implementation, the bottom BRAM outputs data irrespective if the latched address targets the upper BRAM. The MSB of the address bus acts to enable writing to the appropriate BRAM and it also selects the corresponding output by selecting a multiplexer at the output of the upper BRAM in the cascade pair. In [2], Xilinx recommends leaving the outputs of a lower BRAM in a cascade pair unconnected. However, the BIST configurations for both ECC and Cascade compare all of the BRAMs outputs. For BRAMs not expected to generate ORA failures, this enhances the BIST diagnostic resolution by enhancing the observability of cascade pair outputs.

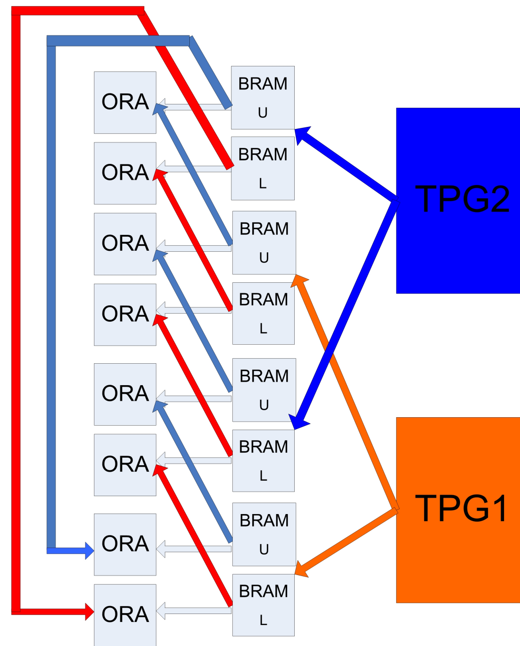

Figure 5.1: ECC and Cascade BIST Architecture

Figure 5.2 illustrates the expected ORA cascade failures in the LSB of each port's data outputs. In general, the number of expected failures can be calculated by Equation 5.1. Four failures are expected per BRAM column because each data ouput is observed by two ORAs. Eight failures are expected per PPC module because the width of the PPC module spans two BRAM columns.

$$\#\ Expected\ ORA\ Failures = 4 * (\#\ BRAM\ Columns\ ) + 8 * (\#\ PPC) \tag{5.1}$$

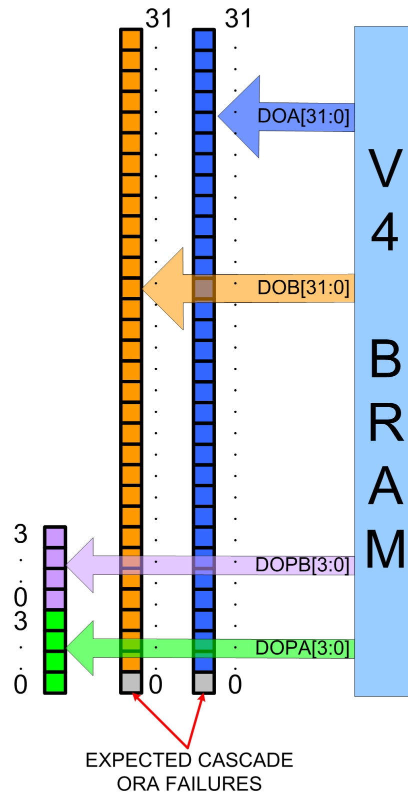

Figure 5.2: Expected Cascade ORA Failure Locations

5.2 ECC BRAM BIST Development

In [36], Stroud discusses a general testing methodology for ECC RAMs. ECC RAMs are typically implemented with an ECC encode logic which generates Hamming bits for written data and ECC decode logic which regenerates the Hamming bits when a data word is read and compares the regenerated Hamming bits to the stored Hamming bits. The problem with testing ECC memories is that they are inherently fault tolerant. For example, in order to test if a ECC BRAM can detect Hamming bit errors, actual Hamming bit errors would have to be introduced. Fortunately, Virtex 4 has two configuration bits, *EN_ECC_WRITE* and *EN_ECC_READ*, which can either be configured to *TRUE* or

FALSE. When *EN_ECC_WRITE* is *TRUE*, the ECC encode logic is enabled and when *EN_ECC_READ* is *TRUE*, the ECC decode logic is enabled.

The remaining issue with testing ECC BRAMs is how to test the ECC encode logic. Out of the 72 data bits, 64 are data and 8 are Hamming bits. Generating all 2^{64} possible inputs to the ECC encode circuity is infeasible. The ECC encode logic consists of an XOR parity tree and testing a parity tree can be completed with four test vectors if the structure of the parity tree is known [37]. However, the structure of the parity tree in the ECC encode circuit is not given in any Xilinx documentation so a more generic parity tree test is needed that will yield high fault coverage irrespective of the parity tree structure. In [36] Stroud shows that the following test vectors will achieve 100% fault coverage for any parity tree implementation:

Generic Parity Tree Test Vectors:

- All zeros
- All combinations of a 1 in a field of zeros
- All combinations of two 1s in a field of zeros

In [6], Stroud implements a circuit that generates all of the above test vectors and is shown in Figure 5.3 with modifications for use with Virtex 4 ECC BRAMs. Given the above circuit, a TPG, ECCTPG, was implemented in VHDL and required 192 slices when implemented in a FX12. The source code is available in Appendix C.

In order to test the ECC decode and correction circuitry, all possible 2^8 Hamming bit values must be read from an ECC BRAM. Without knowing the parity connections for each Hamming bit, it is not feasible to write data to the ECC BRAM to generate all

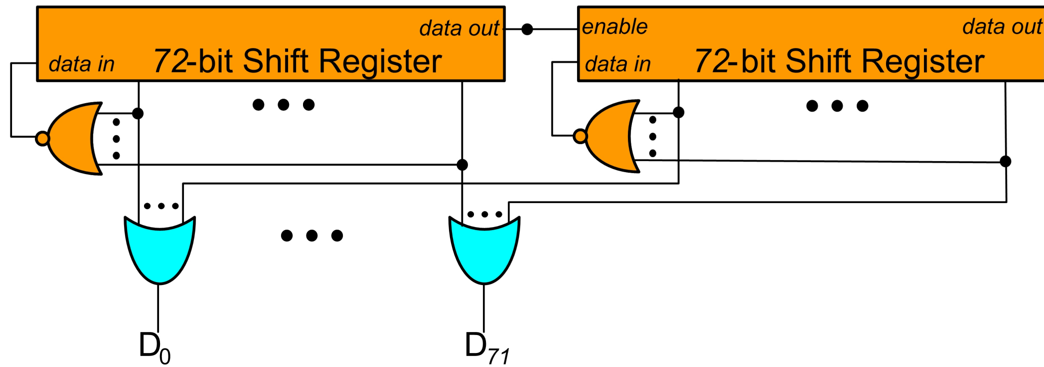

Figure 5.3: Parity Tree TPG [6]

256 Hamming bit values. Fortunately, an ECC BRAM can be initialized to contain all 256 Hamming bit values. These preloaded Hamming bit values will cause the ECC BRAM to indicate and correct single-bit errors or indicate double-bit errors when each memory address is read.

In order generate the parity tree test vectors and read the preloaded Hamming bit values, a TPG with two test phases was developed. The ECC BRAM test algorithm is as follows:

ECC BRAM Test Algorithm:

Phase 1: Read each address and observe a single-bit or double-bit read error along with correcting single-bit erros when detected.

Phase 2: Write to and then read from a memory address the vectors listed for a generic parity tree. Observe single-bit or double bit read error if *EN_ECC_WRITE* is *FALSE*.

The first phase tests the error detection and correction circuitry in the ECC decode circuity by applying all possible Hamming bit combinations to the ECC decode circuity by initializing all ECC BRAMs with all possible 256 Hamming bit combinations and then

Table 5.1: ECC BRAM BIST Configuration Settings

Config Num.	ECC_WRITE_EN	ECC_READ_EN	Phase 1 Clock Cycles	Phase 2 Clock Cycles
1	TRUE	TRUE	512	5184
2	FALSE	TRUE	512	5184

Total ECC BIST Clock Cycles = 11,392

reading the stored patterns. During this phase, a single-bit or double-bit error condition is expected to occur during the read traversal through memory. The second phase further tests the ECC decode parity tree and is also able to test the ECC encode circuitry depending on the configuration of the ECC BRAM as discussed below.

Virtex 4 ECC BRAM BIST consists of two configurations. Table 5.1 summarizes the configuration settings for the two configurations. During the first configuration, the first phase of the TPG causes the ECC BRAM to generate single-bit and double-bit read errors, while the second phase tests the ECC encode circuitry and does not cause read errors because *ECC_WRITE_EN* is generating Hamming for each written data word. In the second configuration, *ECC_WRITE_EN* is set to *FALSE* and tests the ECC decode parity tree because in this mode all 72-bits of a data word are written. Phase 1 of the ECCTPG algorithm is not needed for this second configuration, but this phase only requires 512 clock cycles and it is applied during both configurations so that the same ECCTPG can be used. Applying the phase 2 test vectors causes single-bit and double-bit read errors because in this configuration mode, the TPG is writing directly to the Hamming bit locations instead of the ECC encode circuitry.

5.3 Cascade TPG Development

Since the BRAM memory cell array is tested using March LR with BDS during BRAM BIST, only address decoding faults need to be testing in cascade BIST configurations. While applying MATS+ in BRAM BIST was used to detect all AFs, applying MATS+ to 32K x 1-bit memory requires 327,680 clock cycles which represents almost the total number of clock cycles for all other BRAM BIST configurations.

A cascaded BRAM is two 16K x 1-bit BRAMs configured such that data in the bottom half of the 32K-bit memory space is in the lower BRAM and the upper half is in the upper BRAM. As seen in Figure 5.4, the MSB of the address bus and an inverter selects the write enable for each BRAM. The MSB of the address bus also selects which cascade BRAM data to output. Since all AF faults are tested in BRAM BIST, cascade BIST only needs to test that opposite logic values can be read and written to the upper and lower cascaded BRAMs. The MATS+ march test can still be used, however, the number of address locations is reduced to two: one location in the upper BRAM and one location in the lower BRAM. This simplification allows all but the MSB of the address bus to be grounded (set to logic zero) as also seen in Figure 5.4. Applying MATS+ to both A and B BRAM ports of a cascaded BRAM only requires 20 clock cycles. The MATS+ portion of the MMTPG was modified to create the cascade TPG, CASTPG. The CASTPG implementation required 15 slices and was constrained to four PLB columns to the left of the FX12 center column. The VHDL source code for the CASTPG is included in Appendix D.

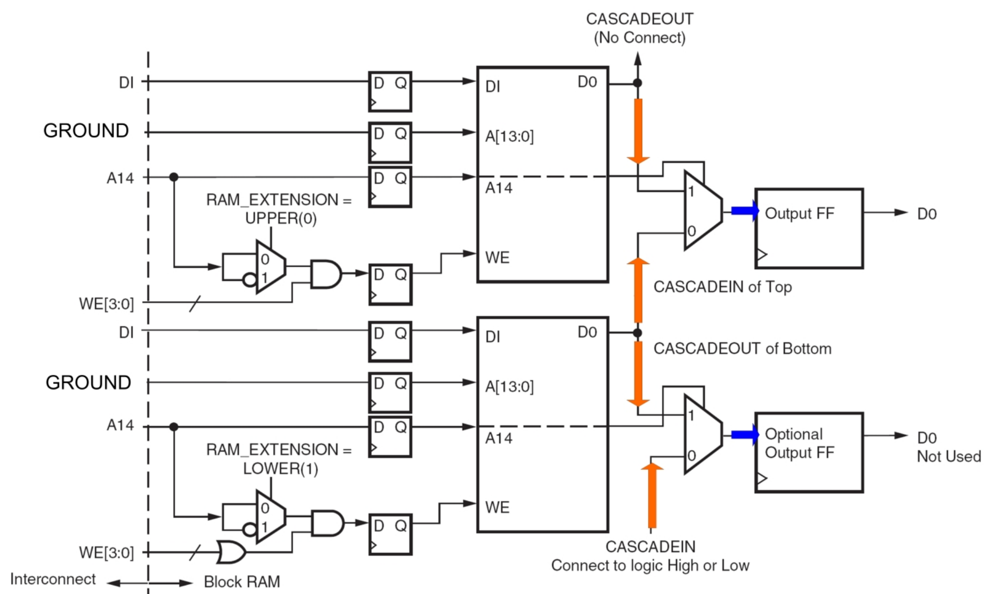

Figure 5.4: Cascade BRAM Operational Diagram

Table 5.2: Summary of Cascade BIST Configuration Settings

Config	Upper BRAM	Lower BRAM	BIST
Num.	RAM_EXTENTION[A,B]	RAM_EXTENTION[A,B]	Clock Cycles
1	UPPER	LOWER	20
2	LOWER	UPPER	20

5.4 BIST Configurations

Two sets of BIST generation programs were developed. *V4ECCTPG*, *V4ECCBIST*, and *V4ECCMOD* facilitate ECC BIST configuration generation while *V4CASTPG*, *V4CASBIST*, and *V4CASMOD* generate CAS BIST configurations. Each program in the the two sets follows the same procedure outlined in BRAM BIST and FIFO BIST. Unlike BRAM and FIFO BIST, ECC and cascade BIST configurations do not require a TPG control register. Table 5.1 summarizes the BRAM configuration settings for ECC BIST, and Table 5.2 outlines the BRAM configuration settings for cascade BIST. Figures 5.5 and 5.6 show unrouted ECC and cascade BIST configurations in a FX12, respectively.

5.5 Running BIST Configurations

ECC and cascade BIST configurations use a single BSCAN module to apply BIST clock cylcles and reset the TPG. Previously, BRAM and FIFO BIST configurations used a second BSCAN module to shift in a control string. This feature is not needed for ECC and cascade configurations since the same TPG algorithm is applied in each of the two configurations. Specifying if the BUT is a upper or lower BRAM, or a portion of the ECC circuitry is enabled or disabled does not necessitate a change in TPG outputs. The testing procedure for ECC and cascade configurations is given below:

1. Download BIST configuration to device

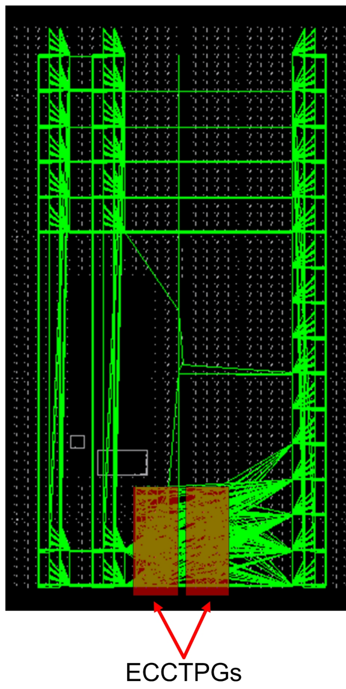

Figure 5.5: FX12 ECC BIST

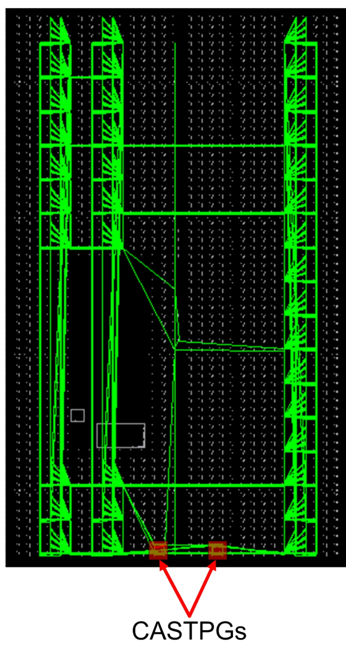

Figure 5.6: FX12 Cascade BIST

2. Goto USER1 access register.
3. Toggle TDI to reset TPG (Active high asynchronous reset).
4. Apply BIST clock cycles.
5. Retrieve ORA results via configuration memory readback.
6. Repeat Steps 1-5 for each addition configuration.

5.6 BIST Results

For cascade BIST, the expected ORA failures discussed previously were observed when ORA results were read back for both cascade configurations generated for LX60 and FX12 devices. The observed failures for this device were 20, which is expected since the LX60 contains five columns of BRAMs. FX12 devices contain three columns of BRAM and a single PPC which caused 20 expected failures.

Since both ECC and cascade BIST have only two configurations, the advantage of partial reconfiguration is minimized because even in partial reconfiguration, the first configuration is a compressed full configuration. Tables 5.3 and 5.4 summarizes the download size and test times for both cascade and ECC BIST configurations. Figures 5.8 and 5.7 illustrate the speed-up factors attained by using both compressed and partial reconfiguration techniques. ECC and cascade BIST configuration timing analysis for several Virtex 4 devices are shown Figure 5.9 and Figure 5.10, respectively. For all devices, the the slowest clock frequency is greater than the 50 MHz maximum boundary scan clock frequency.

Table 5.3: Summary of LX60 CAS BIST Download Size and Test Times

BIST Configuration	Download Size (Bits)			Configuration Time @ 50 Mhz (seconds)		
	FULL	Compress	Partial	FULL	Compress	Partial
1	17,717,632	7,808,352	7,808,352	0.354	0.156	0.156
2	17,717,632	7,808,352	36,256	0.354	0.156	0.001
TOTAL	35,435,264	15,616,704	7,844,608	0.709	0.312	0.157

Table 5.4: Summary of LX60 ECC BIST Download Size and Test Times

BIST Configuration	Download Size (Bits)			Configuration Time @ 50 Mhz (seconds)		
	FULL	Compress	Partial	FULL	Compress	Partial
1	17,717,632	10,121,376	10,121,376	0.354	0.202	0.202
2	17,717,632	10,121,376	113,920	0.354	0.202	0.002
TOTAL	35,435,264	20,242,752	10,235,296	0.709	0.405	0.205

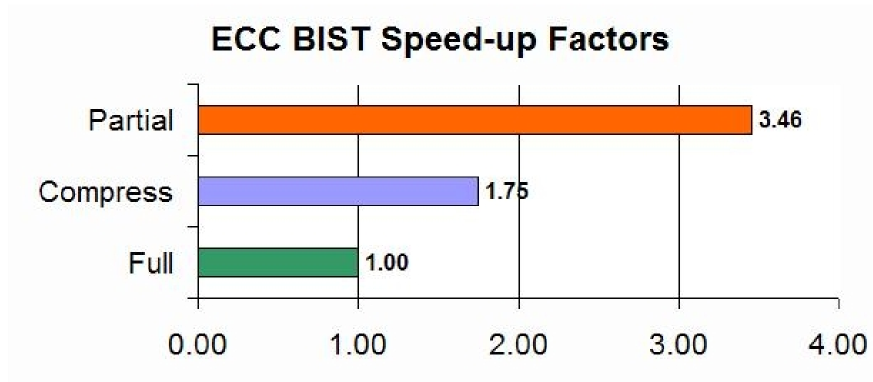

Figure 5.7: LX60 ECC BIST Speed-up Factors

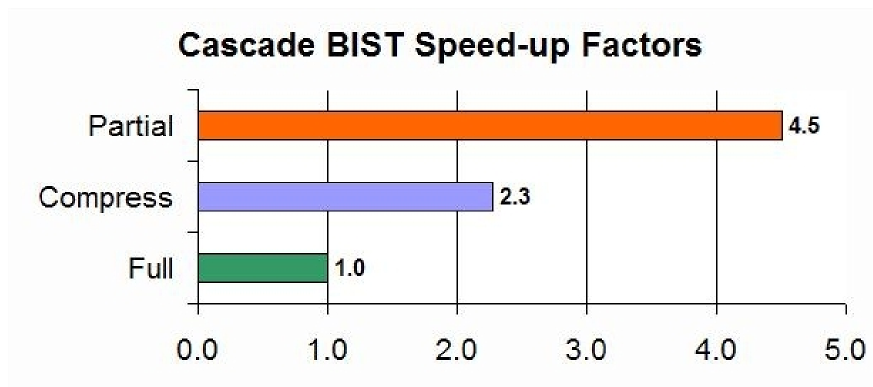

Figure 5.8: LX60 CAS BIST Speed-up Factors

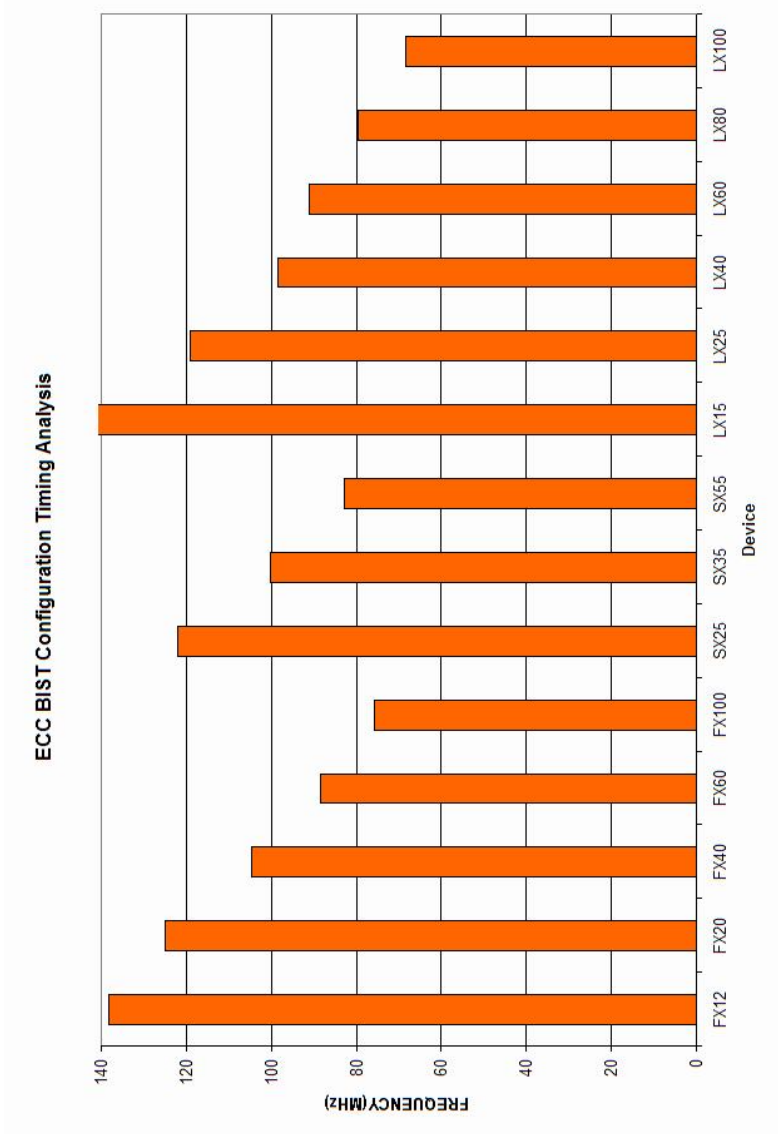

Figure 5.9: ECC BIST Timing Analysis

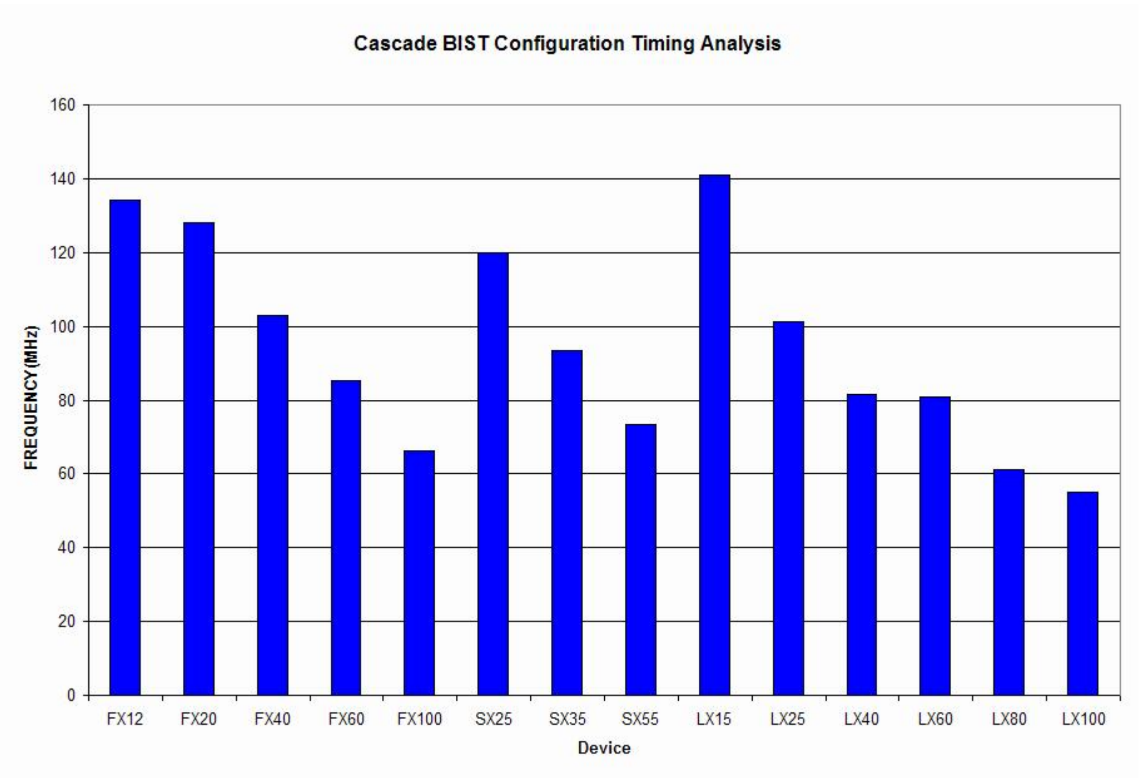

Figure 5.10: Cascade BIST Timing Analysis

Chapter 6

Summary and Conclusion

BIST results for all generated BIST configurations are summarized along with a comparison to the final results attained by Garimella [17]. The BIST architecture detailed in this thesis can also be applied to more recent FPGAs such as the Xilinx Virtex 5. In Virtex 5, Xilinx has introduced several important testability improvements for BRAMs. A potential BIST architecture for Virtex 5 BRAMs will outline future work in this field.

6.1 Summary of Virtex 4 BIST Results

The work contained in this thesis developed a BIST architecture for Virtex 4 BRAMs in all modes of operations. Six BIST configurations were generated to test BRAMs configured to operate as a regular RAM. Fifteen BIST configurations were needed to test BRAMs configured to operate as a FIFO. Two BIST configurations were generated to test both ECC and Cascade BRAM operational modes. The 25 total BIST configurations were generated and downloaded to a LX60, SX35, and FX12 devices. Several faulty LX60 devices were tested using these BIST configurations and in one of the devices, an ORA failure indicated that a single BRAM's DOA[23] output was faulty.

The eight BIST configurations developed by Garimella for Virtex 2 required a total of 485,888 clock cycles [17]. The 25 BIST configurations presented in this thesis require 402,243 clock cycles, a savings of 83,645 clock cycles. This BIST approach was able to reduce the amount BIST clock cycles while testing BRAMs in more modes of operations. Selecting MATS+ instead of March LR for testing additional memory sizes greatly reduced

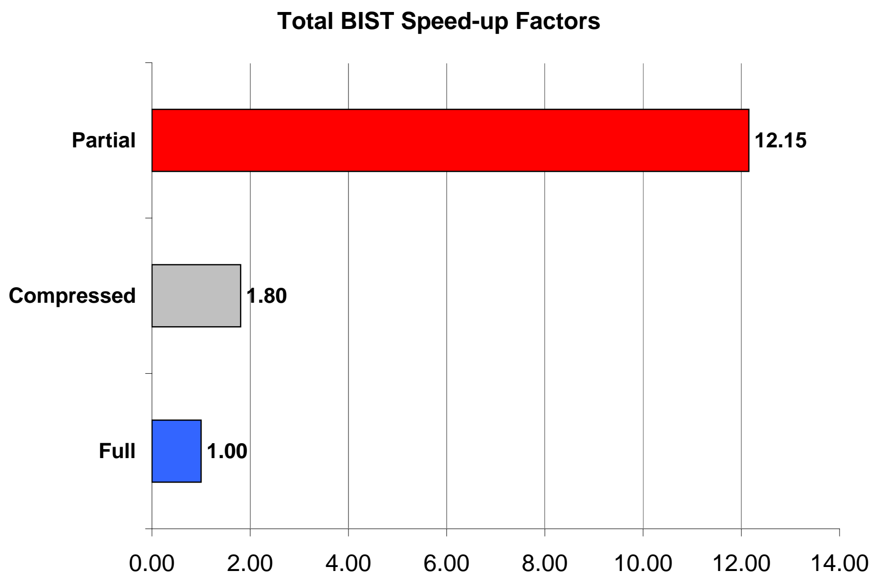

Figure 6.1: BIST Speed-up for LX60

the number of clock cycles. Moreover, the use of partial reconfiguration during each set of BIST configurations allowed for a considerable reduction in the total number of downloaded configuration bits. As shown in the previous chapters, the number of configuration bits is vastly greater than the number of BIST clock cycles. The Virtex 4 BIST configurations were able to take advantage of partial reconfiguration by keeping the TPG-to-BUT and BUT-to-ORA routing static. TPGs capable of adapting to different BRAM configurations were developed and their physical placement was not modified during each set of BIST configurations. The speed-up factors for each configuration technique are shown in Figure 6.1. The speed-up factors are normalized to FULL configurations. Clearly, utilizing partial reconfiguration allows for significant gains in terms of test time and also BIST configuration memory storage.

6.2 Application to Virtex 5

In 2006, Xilnx released the successor to Virtex 4, the Virtex 5. The main distinction between in the two device families is the transition from a 4-input LUT to a 6-input LUT [38]. BRAMs for Virtex 5 have also been modified. Each Virtex 5 BRAM consists of two Virtex 4 BRAMs and each Virtex 5 BRAM can also be cascaded to form a 64K x 1-bit RAM. In Virtex 4, FIFO and BRAM output connections were in different locations, but in Virtex 5, all FIFO and BRAM connections are located together. Having all BRAM and FIFO outputs together allows for the MMTPG and FIFOTPG developed for Virtex 4 to be combined to form an even larger TPG. A BIST architecture could be developed that contains an initial full or compressed configuration and every configuration thereafter could be done through partial reconfigurations. This architecture would require two sets of ORAs: one set of ORAs to compare BRAM, FIFO, and ECC modes, and the second set to compare the cascade BRAM outputs. Another improvement in Virtex 5 is that the ECC encode and decode logic can be tested separately. The encode and decode logic can be configured such that it's outputs bypass the BRAM memory. In addition, the Hamming bits generated for each data word are also available at the BRAM outputs. Virtex 4 did not output Hamming bits.

Virtex 5 represents an excellent platform to develop BIST for BRAM due to the BIST-friendly architectural improvements over Virtex 4. Using partial configuration, Virtex 5 BIST could potentially be more efficient.

Bibliography

[1] S. Trimberger, D. McCarty, and T. Whitney, *Field Programmable Gate Array Technology*, S. Trimberger, Ed. Kluwer, 1994.

[2] Virtex-4 User Guide, User Guide UG070 (v1.4), Xilinx, Inc., 2005 (avaiable at www.xilinx.com).

[3] Virtex-4 Configuration Guide , Configuration Guide UG071, Xilinx, Inc., (available at www.xilinx.com).

[4] A. Van De Goor, I. Tlili, and S. Hamdioui, "Converting march tests for bit-oriented memories into tests for word-oriented memories," in *Memory Technology, Design and Testing, 1998. Proceedings. International Workshop on*, 24-25 Aug. 1998, pp. 46–52.

[5] S. Hamdioui and A. van de Goor, "Efficient tests for realistic faults in dual-port SRAMs," *Computers, IEEE Transactions on*, vol. 51, no. 5, pp. 460–473, May 2002.

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

[7] D. Patterson and J. Hennessy, *Computer Organization and Design: The Hardware / Software Infterface*. Elsevier, New York, 2005.

[8] International Technology Roadmap for Semiconductors 2001, http://public.itrs.net.

[9] Intel Corp., www.intel.com/products.

[10] Xilinx Corp., www.xilinx.com/products.

[11] M. Bushnell and V. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Kluwer Academic Publishers, 2000.

[12] M. Abramovici, J. Emmert, and C. Stroud, "Roving STARS: an integrated approach to on-line testing, diagnosis, and fault tolerance for FPGAs in adaptive computing systems," in *Evolvable Hardware, 2001. Proceedings. The Third NASA/DoD Workshop on*, 12-14 July 2001, pp. 73–92.

[13] M. Abramovici and C. Stroud, "BIST-based delay-fault testing in FPGAs," in *On-Line Testing Workshop, 2002. Proceedings of the Eighth IEEE International*, 8-10 July 2002, pp. 131–134.

[14] ——, "BIST-based test and diagnosis of FPGA logic blocks," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 9, no. 1, pp. 159–172, Feb. 2001.

[15] ——, "BIST-based detection and diagnosis of multiple faults in FPGAs," in *Test Conference, 2000. Proceedings. International*, 3-5 Oct. 2000, pp. 785–794.

[16] M. Abramovici, C. Stroud, and J. Emmert, "Online BIST and BIST-based diagnosis of FPGA logic blocks," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 12, no. 12, pp. 1284–1294, Dec 2004.

[17] S. Garimella, "Built-in self test for regular structured embedded cores in system-on-chip," Master's thesis, Auburn Unversity, 2005.

[18] S. Garimella and C. Stroud, "A system for automated built-in self-test of embedded memory cores in system-on-chip," in *System Theory, 2005. SSST '05. Proceedings of the Thirty-Seventh Southeastern Symposium on*, 20-22 March 2005, pp. 50–54.

[19] C. Stroud, S. Wijesuriya, C. Hamilton, and M. Abramovici, "Built-in self-test of FPGA interconnect," in *Test Conference, 1998. Proceedings. International*, 18-23 Oct. 1998, pp. 404–411.

[20] C. Stroud, M. Lashinsky, J. Nall, J. Emmert, and M. Abramovici, "On-line BIST and diagnosis of FPGA interconnect using roving STARS," in *On-Line Testing Workshop, 2001. Proceedings. Seventh International*, 9-11 July 2001, pp. 27–33.

[21] C. Stroud, K. Leach, and T. Slaughter, "BIST for Xilinx 4000 and Spartan series FPGAs: a case study," in *Test Conference, 2003. Proceedings. ITC 2003. International*, vol. 1, Sept. 30-Oct. 2, 2003, pp. 1258–1267.

[22] C. Stroud, E. Lee, and M. Abramovici, "BIST-based diagnostics of FPGA logic blocks," in *Test Conference, 1997. Proceedings., International*, 1-6 Nov. 1997, pp. 539–547.

[23] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli, "Architecture of field-programmable gate arrays," *Proceedings of the IEEE*, vol. 81, no. 7, pp. 1013–1029, July 1993.

[24] Altera Corp., www.altera/products.

[25] S. Hamdioui, *Testing Static Random Access Memories Defects, Fault Models and Test Patterns.* Kluwer Academic Publishers, 2004.

[26] A. Van de Goor, *Testing Semiconductor Memories: Theory and Practice.* John Wiley & Sons, 1991.

[27] S. Jain and C. Stroud, "Built-in self testing of embedded memories," *Design & Test of Computers, IEEE*, vol. 3, no. 5, pp. 27–37, 1986.

[28] C. Stroud and S. Garimella, "Built-in self-test and diagnosis of multiple embedded cores in socs," in *Proc. International Conference on Embedded Systems and Applications*, 2005.

[29] Virtex-II Pro / Virtex II Pro X Complete Data Sheet, Data Sheet DS083 (v4.5), Xilinx, Inc., 2005 (available at www.xilinx.com).

[30] *Single Error Correction and Double Error Detection, Xilinx Application Note XAPP645, 2006 (available at www.xilinx.com).*

[31] Development System Reference Guide (v8.2i), Xilinx, Inc., 2005 (available at www.xilinx.com).

[32] A. van de Goor, G. Gaydadjiev, V. Mikitjuk, and V. Yarmolik, "March LR: a test for realistic linked faults," in *VLSI Test Symposium, 1996., Proceedings of 14th*, 28 April-1 May 1996, pp. 272–280.

[33] C. Stroud, J. Sunwoo, S. Garimella, and J. Harris, "Built-in self-test for system-on-chip: a case study," in *Test Conference, 2004. Proceedings. ITC 2004. International*, 2004, pp. 837–846.

[34] S. Dhingra, "Built-in self-test of logic resources in field programmable gate arrays using partial reconfiguration," Master's thesis, Auburn University, 2006.

[35] Atmel Corp. Combined Megacell Testing, Application Note AN0696C, 1999.

[36] L. Wang, C. Stroud, and N. Touba, *System-on-Chip Test Architectures: Nanometer Design for Testability.* Elsevier, 2007.

[37] S. Mourad and E. McCluskey, "Testability of parity checkers," *Industrial Electronics, IEEE Transactions on*, vol. 36, no. 2, pp. 254–262, May 1989.

[38] Virtex-5 User Guide UG190 (v3.0), Xilinx, Inc., 2007 (available at www.xilinx.com).

Appendices

Appendix A

MMTPG VHDL Source code

```
library ieee;
   use ieee.std_logic_1164.all;
   use ieee.std_logic_unsigned.all;
 entity fsm is
   port (
     Reset : in std_logic;
     TDI,DRCK,UPDATE,SHIFT: in std_logic;
     Clk : in std_logic;
     WEA : buffer std_logic;
     WEB: buffer std_logic;
     --OEN : out std_logic;
     DIA : out std_logic_vector(35 downto 0);
     DIB: out std_logic_vector(35 downto 0);
     ADDRA : out std_logic_vector(13 downto 0);
     ADDRB: out std_logic_vector(13 downto 0);
     EnA: out std_logic;
     EnB : out std_logic;
     SSR : out std_logic; --follows EN_LEVEL signal
     REGCE: out std_logic);
     --FINISHTEST: out std_logic);

   end fsm;
 architecture  BEHAVIORAL of fsm is
   type testmodes is (MarchLR,MATS,March_s2pf,March_d2pf);
   type phases is (Init,dummy,Phase1,phase2,phase3,phase4,phase5,phase6,phase7,phase8,p
hase9,phase10,phase11,phase12,phase13,phase14,phase15,phase16);
   type elements is (ele1,ele2,ele3,ele4,ele5);
   signal testmode: testmodes :=MarchLR;
   signal phase : phases := dummy;
   signal Element : elements := ele1;
   signal Address : std_logic_vector (13 downto 0);
   signal AddressB: std_logic_vector(13 downto 0);
   signal MAXADDRESS : std_logic_vector ( 13 downto 0);
   constant MINADDRESS : std_logic_vector ( 13 downto 0) := (others => '0');
   signal portBtested: std_logic:='0';
   signal tempdata: std_logic_vector(35 downto 0); --to comply with design consideratio
ns by Xilinx
   signal tempdataB: std_logic_vector(35 downto 0); --to comply with design considerati
ons by Xilinx
   signal ENATEMP,ENBTEMP,SSRTEMP,WEAtemp,WEBtemp,EN_LEVEL, WEN_ACTIVE, REGCE_ACTIVE: s
td_logic;
   signal MODE: std_logic_vector(2 downto 0);
   signal SR, PDO : std_logic_vector(5 downto 0);

   begin
     bsync:
   process (DRCK, UPDATE,SHIFT) begin -- sync circuitry on BSCAN clock
       if (DRCK'event and DRCK = '1') then
         if (SHIFT = '1') then               -- shift
           for I in 0 to 4 loop
             SR(I) <= SR(I+1);
           end loop;
           SR(5) <= TDI;
         -- TDO <= SR(0);
         end if;
       end if;
       if (UPDATE = '1') then PDO <= SR;   -- update
       end if;
     end process bsync;
     EN_LEVEL <= PDO(3);
     WEN_ACTIVE <=PDO(5);
     REGCE_ACTIVE <= PDO(4);
     MODE(0)<=PDO(0);
     MODE(1)<= PDO(1);
     MODE(2) <= PDO(2);

   --begin
     p0:
    Process(clk,Reset,MODE,MAXADDRESS,tempdata,tempdataB,Address,AddressB,WEA)
```

```
begin
  if ( Reset = '1' ) then
    ADDRB <= (others => '1');
    ADDRA <= (others => '1');
    DIA <= (others => '0');
    DIB <= (others => '0');
    REGCE <= REGCE_ACTIVE;
    EnA <= not EN_LEVEL;
    EnB <= not EN_LEVEL;
    WEA <= not WEN_ACTIVE;
    WEB <= not WEN_ACTIVE;
    SSR <= not EN_LEVEL;

  elsif (Clk = '1' and Clk'Event) then

    case MODE is
      when "000" =>
        MAXADDRESS<="00000111111111"; --512 X 36 with BDS
        DIA<=tempdata;
        DIB<=tempdata;
        ADDRA<=Address(8 downto 0)&"11111";
        ADDRB<=Address(8 downto 0)&"11111";
        REGCE <=(REGCE_ACTIVE);
        WEA<=WEAtemp;
        WEB<=WEBtemp;
        SSR<=SSRtemp;
        EnA<=EnAtemp;
        EnB<=EnBtemp;
        testmode <= MarchLR;
      when "001" =>
        MAXADDRESS<="01111111111111"; --8k X 2
        DIA<="1111111111111111111111111111111111"&tempdata(1 downto 0);
        DIB<="1111111111111111111111111111111111"&tempdata(1 downto 0);
        ADDRA<=Address(12 downto 0)&'1';
        ADDRB<=Address(12 downto 0)&'1';
        WEA<=WEAtemp;
        WEB<=WEBtemp;
        SSR<=SSRtemp;
        EnA<=EnAtemp;
        EnB<=EnBtemp;
        Testmode <= MATS;
        REGCE <= REGCE_ACTIVE;

      when "010" =>
        MAXADDRESS<="11111111111111"; --16k X 1
        DIA<="11111111111111111111111111111111111"&tempdata(0);
        DIB<="11111111111111111111111111111111111"&tempdata(0);
        ADDRA<=Address( 13 downto 0);
        ADDRB<=Address( 13 downto 0);
        REGCE <= REGCE_ACTIVE;
        WEA<=WEAtemp;
        WEB<=WEBtemp;
        SSR<=SSRtemp;
        EnA<=EnAtemp;
        EnB<=EnBtemp;
        Testmode <= MATS;
      when "101" =>
        MAXADDRESS<="00000111111111"; --512 X 36
        DIA<=tempdata;
        DIB<=tempdata;
        ADDRA<=Address(8 downto 0)&"11111";
        ADDRB<=Address(8 downto 0)&"11111";
        REGCE <= (REGCE_ACTIVE);
        Testmode <= MATS;
        WEA<=WEAtemp;
        WEB<=WEBtemp;
        SSR<=SSRtemp;
        EnA<=EnAtemp;
        EnB<=EnBtemp;
```

```
when "011" =>
    MAXADDRESS<="00000111111111"; --s2pf
    DIA<=tempdata;
    DIB<=tempdataB;
    ADDRB<=Address(8 downto 0)&"11111";
    ADDRA<=Address(8 downto 0)&"11111";
    REGCE <= (REGCE_ACTIVE);
    testmode <= March_s2pf;
    WEA<=WEAtemp;
    WEB<=WEBtemp;
    SSR<=SSRtemp;
    EnA<=EnAtemp;
    EnB<=EnBtemp;

when "100" =>
    MAXADDRESS<="00000111111111"; --d2pf
    DIA<=tempdata;
    DIB<=tempdataB;
    ADDRB<=AddressB(8 downto 0)&"11111";
    ADDRA<=Address(8 downto 0)&"11111";
    REGCE <= (REGCE_ACTIVE);
    testmode <= March_d2pf;
    WEA<=WEAtemp;
    WEB<=WEBtemp;
    SSR<=SSRtemp;
    EnA<=EnAtemp;
    EnB<=EnBtemp;

when "110" =>
    MAXADDRESS<="00111111111111"; --4k x 4
    DIA<="1111111111111111111111111111"&tempdata(3 downto 0);
    DIB<="1111111111111111111111111111"&tempdata(3 downto 0);
    ADDRB<=AddressB(11 downto 0)&"11";
    ADDRA<=Address(11 downto 0)&"11";
    REGCE <= (REGCE_ACTIVE);
    testmode <= MATS;
    WEA<=WEAtemp;
    WEB<=WEBtemp;
    SSR<=SSRtemp;
    EnA<=EnAtemp;
    EnB<=EnBtemp;

when "111" =>
    MAXADDRESS<="00011111111111"; --2k x 9
    DIA<="11111111111111111111111"&tempdata(8 downto 0);
    DIB<="11111111111111111111111"&tempdata(8 downto 0);
    ADDRB<=AddressB(10 downto 0)&"111";
    ADDRA<=Address(10 downto 0)&"111";
    REGCE <= (REGCE_ACTIVE);
    testmode <= MATS;
    WEA<=WEAtemp;
    WEB<=WEBtemp;
    SSR<=SSRtemp;
    EnA<=EnAtemp;
    EnB<=EnBtemp;
when others =>
end case;
end if;
--  end if;
end process;

p1:
Process(Clk)
begin
if (Reset ='1') then
    tempdata <= (others => '0');
    tempdataB <= (others => '0');
    AddressB <= (others => '0');
```

```
Address <= (others => '0');
Element <= ele1;
Phase <= dummy;
WEAtemp <= not(WEN_ACTIVE);
WEBtemp <= not(WEN_ACTIVE);
SSRtemp <= not EN_LEVEL;
EnAtemp <= not EN_LEVEL;
EnBtemp <= not EN_LEVEL;
--
elsif (Clk = '1' and Clk'Event) then

  case testmode is

   -----------------------------MARCH LR------------------------------
   when MarchLR=>

     case Phase is
      when dummy =>
       EnAtemp <= EN_LEVEL;
       EnBtemp <= EN_LEVEL;
       SSRtemp <= EN_LEVEL;
       Phase <= Init;
      when Init =>
       SSRtemp<= not(EN_LEVEL);
       Address <= MINADDRESS;
       WEAtemp <= WEN_ACTIVE;
       WEBtemp <= WEN_ACTIVE;
       tempdata <= (others => '0');
       Element <= ele1;
       Phase <= Phase1;
       EnAtemp <= EN_LEVEL;
       EnBtemp <= not(EN_LEVEL);
      when phase1 => --  U w 00000000000000000000000000000000000
       if ( Address /= MAXADDRESS ) then
        Address <= Address + '1';
        WEAtemp <= WEN_ACTIVE;
        WEBtemp <= WEN_ACTIVE;
        Element <= ele1;
        tempdata <= (others => '0');
       else -- D r 00000000000000000000000000000000000
        Address <= MAXADDRESS;
        WEAtemp <= not(WEN_ACTIVE);
        WEBtemp <= not(WEN_ACTIVE);
        tempdata <= (others => '0');
        Phase <= Phase2;

        Element <= ele2;
       - end if;
      when phase2 => --  D r 00000000000000000000000000000000000 w 1111111
11111111111111111111111111111
       case Element is
        when ele2 =>
         WEAtemp <= WEN_ACTIVE;
         WEBtemp <= WEN_ACTIVE;
         tempdata <= (others => '1');
         Element <= ele1;
        when ele1 =>
         if ( Address /= MINADDRESS ) then
          Address <= Address - '1';
          WEAtemp <= not(WEN_ACTIVE);
          WEBtemp <= not(WEN_ACTIVE);
          Element <= ele2;
          tempdata <= (others => '0');
         else -- U r 111111111111111111111111111111111111
          Address <= MINADDRESS;
          WEAtemp <= not(WEN_ACTIVE);
          WEBtemp <= not(WEN_ACTIVE);
          tempdata <= (others => '1');
          Phase <= Phase3;
```

```
                Element <= ele2;
            - end if;
        when others =>
        end case;
    when phase3 => --  U r 111111111111111111111111111111111111 w 0000000
00000000000000000000000000000 r 000000000000000000000000000000000000 r 0000000000000000000
00000000000000000 w 111111111111111111111111111111111111
        case Element is
        when ele2 =>
            WEAtemp <= WEN_ACTIVE;
            WEBtemp <= WEN_ACTIVE;
            tempdata <= (others => '0');
            Element <= ele3;
        when ele3 =>
            WEAtemp <= not(WEN_ACTIVE);
            WEBtemp <= not(WEN_ACTIVE);
            tempdata <= (others => '0');
            Element <= ele4;
        when ele4 =>
            WEAtemp <= not(WEN_ACTIVE);
            WEBtemp <= not(WEN_ACTIVE);
            tempdata <= (others => '0');
            Element <= ele5;
        when ele5 =>
            WEAtemp <= WEN_ACTIVE;
            WEBtemp <= WEN_ACTIVE;
            tempdata <= (others => '1');
            Element <= ele1;
        when ele1 =>
            if ( Address /= MAXADDRESS ) then
                Address <= Address + '1';
                WEAtemp <= not(WEN_ACTIVE);
                WEBtemp <=not WEN_ACTIVE;
                Element <= ele2;
                tempdata <= (others => '1');
            else -- U r 111111111111111111111111111111111111
                Address <= MINADDRESS;
                WEAtemp <= not(WEN_ACTIVE);
                WEBtemp <= not(WEN_ACTIVE);
                tempdata <= (others => '1');
                Phase <= Phase4;

                Element <= ele2;
            - end if;
        when others =>
        end case;
    when phase4 => --  U r 111111111111111111111111111111111111 w 0000000
00000000000000000000000000000
        case Element is
        when ele2 =>
            WEAtemp <= WEN_ACTIVE;
            WEBtemp <= WEN_ACTIVE;
            tempdata <= (others => '0');
            Element <= ele1;
        when ele1 =>
            if ( Address /= MAXADDRESS ) then
                Address <= Address + '1'; --was -
                WEAtemp <= not(WEN_ACTIVE);
                WEBtemp <= not(WEN_ACTIVE);
                Element <= ele2;
                tempdata <= (others => '1');
            else -- U r 000000000000000000000000000000000000
                Address <= MINADDRESS;
                WEAtemp <= not(WEN_ACTIVE);
                WEBtemp <= not(WEN_ACTIVE);
                tempdata <= (others => '0');
                Phase <= Phase5;
```

```
                Element <= ele2;
              end if;
            when others =>
          end case;
        when phase5 => --  U r 000000000000000000000000000000000000 w 1111111
11111111111111111111111111111 r 111111111111111111111111111111111111 r 1111111111111111111
11111111111111111 w 000000000000000000000000000000000000
          case Element is
            when ele2 =>
              WEAtemp <= WEN_ACTIVE;
              WEBtemp <= WEN_ACTIVE;
              tempdata <= (others => '1');
              Element <= ele3;
            when ele3 =>
              WEAtemp <= not(WEN_ACTIVE);
              WEBtemp <= not(WEN_ACTIVE);
              tempdata <= (others => '1');
              Element <= ele4;
            when ele4 =>
              WEAtemp <= not(WEN_ACTIVE);
              WEBtemp <= not(WEN_ACTIVE);
              tempdata <= (others => '1');
              Element <= ele5;
            when ele5 =>
              WEAtemp <= WEN_ACTIVE;
              WEBtemp <= WEN_ACTIVE;
              tempdata <= (others => '0');
              Element <= ele1;
            when ele1 =>
              if ( Address /= MAXADDRESS ) then
                Address <= Address + '1';
                WEAtemp <= not(WEN_ACTIVE);
                WEBtemp <= not(WEN_ACTIVE);
                Element <= ele2;
                tempdata <= (others => '0');
              else -- D r 000000000000000000000000000000000000
                Address <= MAXADDRESS;
                WEAtemp <= not(WEN_ACTIVE);
                WEBtemp <= not(WEN_ACTIVE);
                tempdata <= (others => '0');
                Phase <= Phase6;

                Element <= ele2;
              end if;
            when others =>
          end case;
        when phase6 => --  D r 000000000000000000000000000000000000 w 0101010
10101010101010101010101010101 w 101010101010101010101010101010101010 r 1010101010101010101
01010101010101010
          case Element is
            when ele2 =>
              WEAtemp <= WEN_ACTIVE;
              WEBtemp <= WEN_ACTIVE;
              tempdata <= "010101010101010101010101010101010101";
              Element <= ele3;
            when ele3 =>
              WEAtemp <= WEN_ACTIVE;
              WEBtemp <= WEN_ACTIVE;
              tempdata <= "101010101010101010101010101010101010";
              Element <= ele4;
            when ele4 =>
              WEAtemp <= not(WEN_ACTIVE);
              WEBtemp <= not(WEN_ACTIVE);
              tempdata <= "101010101010101010101010101010101010";
              Element <= ele1;
            when ele1 =>
              if ( Address /= MINADDRESS ) then
                Address <= Address - '1';
                WEAtemp <= not(WEN_ACTIVE);
```

```
            WEBtemp <= not(WEN_ACTIVE);
            Element <= ele2;
            tempdata <= (others => '0');
          else -- U r 10101010101010101010101010101010
            Address <= MINADDRESS;
            WEAtemp <= not(WEN_ACTIVE);
            WEBtemp <= not(WEN_ACTIVE);
            tempdata <= "10101010101010101010101010101010";
            Phase <= Phase7;

            Element <= ele2;
          end if;
        when others =>
      end case;
    when phase7 => --  U r 10101010101010101010101010101010 w 0101010
1010101010101010101010101010101 r 01010101010101010101010101010101
      case Element is
        when ele2 =>
          WEAtemp <= WEN_ACTIVE;
          WEBtemp <= WEN_ACTIVE;
          tempdata <= "01010101010101010101010101010101";
          Element <= ele3;
        when ele3 =>
          WEAtemp <= not(WEN_ACTIVE);
          WEBtemp <= not(WEN_ACTIVE);
          tempdata <= "01010101010101010101010101010101";
          Element <= ele1;
        when ele1 =>
          if ( Address /= MAXADDRESS ) then
            Address <= Address + '1';
            WEAtemp <= not(WEN_ACTIVE);
            WEBtemp <= not(WEN_ACTIVE);
            Element <= ele2;
            tempdata <= "10101010101010101010101010101010";
          else -- D r 01010101010101010101010101010101
            Address <= MAXADDRESS;
            WEAtemp <= not(WEN_ACTIVE);
            WEBtemp <= not(WEN_ACTIVE);
            tempdata <= "01010101010101010101010101010101";
            Phase <= Phase8;

            Element <= ele2;
          end if;
        when others =>
      end case;
    when phase8 => --  D r 01010101010101010101010101010101 w 0011001
1001100110011001100110011 w 11001100110011001100110011001100 r 110011001100110
01100110011001100
      case Element is
        when ele2 =>
          WEAtemp <= WEN_ACTIVE;
          WEBtemp <= WEN_ACTIVE;
          tempdata <= "00110011001100110011001100110011";
          Element <= ele3;
        when ele3 =>
          WEAtemp <= WEN_ACTIVE;
          WEBtemp <= WEN_ACTIVE;
          tempdata <= "11001100110011001100110011001100";
          Element <= ele4;
        when ele4 =>
          WEAtemp <= not(WEN_ACTIVE);
          WEBtemp <= not(WEN_ACTIVE);
          tempdata <= "11001100110011001100110011001100";
          Element <= ele1;
        when ele1 =>
          if ( Address /= MINADDRESS ) then
            Address <= Address - '1';
            WEAtemp <= not(WEN_ACTIVE);
            WEBtemp <= not(WEN_ACTIVE);
```

```
Element <= ele2;
tempdata <= "01010101010101010101010101010101";
else -- U r 11001100110011001100110011001100
Address <= MINADDRESS;
WEAtemp <= not(WEN_ACTIVE);
WEBtemp <= not(WEN_ACTIVE);
tempdata <= "11001100110011001100110011001100";
Phase <= Phase9;

Element <= ele2;
- end if;
when others =>
end case;
when phase9 => --  U r 11001100110011001100110011001100 w 0011001
10011001100110011001100110011 r 00110011001100110000110011001100
case Element is
when ele2 =>
WEAtemp <= WEN_ACTIVE;
WEBtemp <= WEN_ACTIVE;
tempdata <= "00110011001100110011001100110011";
Element <= ele3;
when ele3 =>
WEAtemp <= not(WEN_ACTIVE);
WEBtemp <= not(WEN_ACTIVE);
tempdata <= "00110011001100110000110011001100";
Element <= ele1;
when ele1 =>
if ( Address /= MAXADDRESS ) then
Address <= Address + '1';
WEAtemp <= not(WEN_ACTIVE);
WEBtemp <= not(WEN_ACTIVE);
Element <= ele2;
tempdata <= "11001100110011001100110011001100";
else -- D r 00110011001100110011001100110011
Address <= MAXADDRESS;
WEAtemp <= not(WEN_ACTIVE);
WEBtemp <= not(WEN_ACTIVE);
tempdata <= "00110011001100110011001100110011";
Phase <= Phase10;

Element <= ele2;
- end if;
when others =>
end case;
when phase10 => --  D r 00110011001100110011001100110011 w 000011
11000011110000111100001111110000 w 11110000111100001111000011110000 1111 r 111100001111000011
110000111100001111
case Element is
when ele2 =>
WEAtemp <= WEN_ACTIVE;
WEBtemp <= WEN_ACTIVE;
tempdata <= "00001111000011110000111100001111 0000";
Element <= ele3;
when ele3 =>
WEAtemp <= WEN_ACTIVE;
WEBtemp <= WEN_ACTIVE;
tempdata <= "11110000111100001111000011110000 1111";
Element <= ele4;
when ele4 =>
WEAtemp <= not(WEN_ACTIVE);
WEBtemp <= not(WEN_ACTIVE);
tempdata <= "11110000111100001111000011110000 1111";
Element <= ele1;
when ele1 =>
if ( Address /= MINADDRESS ) then
Address <= Address - '1';
WEAtemp <= not(WEN_ACTIVE);
WEBtemp <= not(WEN_ACTIVE);
Element <= ele2;
```

```
tempdata <= "001100110011001100110011001100110011";
else -- U r 111100001111000011110000111100001111
Address <= MINADDRESS;
WEAtemp <= not(WEN_ACTIVE);
WEBtemp <= not(WEN_ACTIVE);
tempdata <= "111100001111000011110000111100001111";
Phase <= Phase11;

Element <= ele2;
- end if;
when others =>
end case;
when phase11 => -- U r 111100001111000011110000111100001111 w 000011
110000111100001111000011110000 r 000011110000111100001111000011110000
case Element is
when ele2 =>
WEAtemp <= WEN_ACTIVE;
WEBtemp <= WEN_ACTIVE;

tempdata <= "000011110000111100001111000011110000";
Element <= ele3;
when ele3 =>
WEAtemp <= not(WEN_ACTIVE);
WEBtemp <= not(WEN_ACTIVE);
tempdata <= "000011110000111100001111000011110000";
Element <= ele1;
when ele1 =>
if ( Address /= MAXADDRESS ) then
Address <= Address + '1';
WEAtemp <= not(WEN_ACTIVE);
WEBtemp <= not(WEN_ACTIVE);
Element <= ele2;
tempdata <= "111100001111000011110000111100001111";
else -- D r 000011110000111100001111000011110000
Address <= MAXADDRESS;
WEAtemp <= not(WEN_ACTIVE);
WEBtemp <= not(WEN_ACTIVE);
tempdata <= "000011110000111100001111000011110000";
Phase <= Phase12;

Element <= ele2;
- end if;
when others =>
end case;
when phase12 => -- D r 000011110000111100001111000011110000 w 000000
001111111110000000001111111110000 w 111111111000000000111111111000000001111 r 11111111100000000011
11111110000000001111
case Element is
when ele2 =>
WEAtemp <= WEN_ACTIVE;
WEBtemp <= WEN_ACTIVE;
tempdata <= "0000000001111111110000000001111111110000";
Element <= ele3;
when ele3 =>
WEAtemp <= WEN_ACTIVE;
WEBtemp <= WEN_ACTIVE;
tempdata <= "1111111110000000001111111110000000001111";
Element <= ele4;
when ele4 =>
WEAtemp <= not(WEN_ACTIVE);
WEBtemp <= not(WEN_ACTIVE);
tempdata <= "1111111110000000001111111110000000001111";
Element <= ele1;
when ele1 =>
if ( Address /= MINADDRESS ) then
Address <= Address - '1';
WEAtemp <= not(WEN_ACTIVE);
WEBtemp <= not(WEN_ACTIVE);
Element <= ele2;
```

```
                    tempdata <= "00001111000011110000111100001111 0000";
                else -- U r 11111111000000000111111110000000001111
                    Address <= MINADDRESS;
                    WEAtemp <= not(WEN_ACTIVE);
                    WEBtemp <= not(WEN_ACTIVE);
                    tempdata <= "11111111000000000111111110000000001111";
                    Phase <= Phase13;

                    Element <= ele2;
                end if;
            when others =>
        end case;
    when phase13 => --  U r 11111111000000000111111110000000001111 w 000000
0011111111100000000111111110000 r 000000001111111100000000111111110000
        case Element is
            when ele2 =>
                WEAtemp <= WEN_ACTIVE;
                WEBtemp <= WEN_ACTIVE;

                tempdata <= "000000001111111100000000111111110000";
                Element <= ele3;
            when ele3 =>
                WEAtemp <= not(WEN_ACTIVE);
                WEBtemp <= not(WEN_ACTIVE);
                tempdata <= "000000001111111100000000111111110000";
                Element <= ele1;
            when ele1 =>
                if ( Address /= MAXADDRESS ) then
                    Address <= Address + '1';
                    WEAtemp <= not(WEN_ACTIVE);
                    WEBtemp <= not(WEN_ACTIVE);
                    Element <= ele2;
                    tempdata <= "11111111000000000111111110000000001111";
                else -- D r 000000001111111100000000111111110000
                    Address <= MAXADDRESS;
                    WEAtemp <= not(WEN_ACTIVE);
                    WEBtemp <= not(WEN_ACTIVE);
                    tempdata <= "000000001111111100000000111111110000";
                    Phase <= Phase14;

                    Element <= ele2;
                end if;
            when others =>
        end case;
    when phase14 => --  D r 000000001111111100000000111111110000 w 000000
00000000000011111111111111100 w 11111111111111110000000000000000001111 r 1111111111111111100
00000000000001111
        case Element is
            when ele2 =>
                WEAtemp <= WEN_ACTIVE;
                WEBtemp <= WEN_ACTIVE;

                tempdata <= "000000000000000000111111111111111100";
                Element <= ele3;
            when ele3 =>
                WEAtemp <= WEN_ACTIVE;
                WEBtemp <= WEN_ACTIVE;

                tempdata <= "111111111111111100000000000000001111";
                Element <= ele4;
            when ele4 =>
                WEAtemp <= not(WEN_ACTIVE);
                WEBtemp <= not WEN_ACTIVE;
                tempdata <= "111111111111111100000000000000001111";
                Element <= ele1;
            when ele1 =>
                if ( Address /= MINADDRESS ) then
                    Address <= Address - '1';
                    WEAtemp <= not(WEN_ACTIVE);
```

```
        WEBtemp <= not(WEN_ACTIVE);
        Element <= ele2;
        tempdata <= "000000001111111100000000111111110000";
      else -- U r 111111111111111100000000000000001111
        Address <= MINADDRESS;
        WEAtemp <= not(WEN_ACTIVE);
        WEBtemp <= not(WEN_ACTIVE);
        tempdata <= "111111111111111100000000000000001111";
        Phase <= Phase15;

        Element <= ele2;
      end if;
    when others =>
    end case;
  when phase15 => --  U r 111111111111111100000000000000001111 w 000000
000000000000111111111111111100 r 000000000000000000111111111111111100
    case Element is
      when ele2 =>
        WEAtemp <= WEN_ACTIVE;
        WEBtemp <= WEN_ACTIVE;
        tempdata <= "000000000000000000111111111111111100";
        Element <= ele3;
      when ele3 =>
        WEAtemp <= not(WEN_ACTIVE);
        WEBtemp <= not(WEN_ACTIVE);
        tempdata <= "000000000000000000111111111111111100";
        Element <= ele1;
      when ele1 =>
        if ( Address /= MAXADDRESS ) then
          Address <= Address + '1';
          WEAtemp <= not(WEN_ACTIVE);
          WEBtemp <= not(WEN_ACTIVE);
          Element <= ele2;
          tempdata <= "111111111111111100000000000000001111";
        else -- D r 000000000000000000111111111111111100
          Address <= MAXADDRESS;
          WEAtemp <= not(WEN_ACTIVE);
          WEBtemp <= not(WEN_ACTIVE);
          tempdata <= "000000000000000000111111111111111100";
          Phase <= Phase16;

          Element <= ele1;
        end if;
      when others =>
    end case;
  when phase16 => --  D r 000000000000000000111111111111111100
    if ( Address /= MINADDRESS ) then
      Address <= Address - '1';
      WEAtemp <= not(WEN_ACTIVE);
      WEBtemp <= not(WEN_ACTIVE);
      Element <= ele1;
      tempdata <= "000000000000000000111111111111111100";
    else -- U w 000000000000000000000000000000000000
      Address <= MINADDRESS;
      WEAtemp <= WEN_ACTIVE;
      WEBtemp <= WEN_ACTIVE;
      tempdata <= (others => '0');
      if ( portBtested = '1' ) then -- change testing mode
        portBtested<='0';
        phase<=Init;
        Element<=ele1;
      else
        portBtested<='1';
        Phase <=Phase1;
        EnBtemp <= EN_LEVEL;
        EnAtemp <= not(EN_LEVEL);
        Element <= ele1;
      end if;
    end if;
```

```
end case;

-----------------------------MATS-------DTM---------------------
when MATS=>
  case Phase is
    when dummy =>
      EnAtemp <= EN_LEVEL;
      EnBtemp <= EN_LEVEL;
      SSRtemp <= EN_LEVEL;
      Phase <= Init;
      portbtested <='0';

    when Init =>
      Address <= MINADDRESS;
      WEAtemp <= WEN_ACTIVE; --changed from not
      WEBtemp <= WEN_ACTIVE;
      tempdata <= (others => '0');
      Phase <= Phase1;
      if (portbtested ='1') then
        EnAtemp <= not EN_LEVEL;
        EnBtemp <= (EN_LEVEL);
        portbtested<='0';
      else
        EnAtemp <= EN_LEVEL;
        EnBtemp <= not(EN_LEVEL);
      end if;
      SSRtemp <=not(EN_LEVEL);

    when phase1=>
      if ( Address /= MAXADDRESS ) then
        WEAtemp <= WEN_ACTIVE;
        WEBtemp <= WEN_ACTIVE;
        tempdata <= (others => '0');
        Address <= Address+'1';
      elsif (Address = MAXADDRESS) then

        tempdata <= (others => '0');
        WEAtemp <=not WEN_ACTIVE;
        WEBtemp <=not WEN_ACTIVE;
        Phase <=Phase2;
        Address <= MINADDRESS;
      end if;


    when phase2 => --  up R0 , W1
      if ( Address /= MAXADDRESS ) then
        case element is
          when ele1 => --read zeros
            WEAtemp <= WEN_ACTIVE;
            WEBtemp <= WEN_ACTIVE;
            tempdata <= (others => '1');
            element <=ele2;

          when ele2 => --write ones
            WEAtemp <=  not WEN_ACTIVE;
            WEBtemp <=  not WEN_ACTIVE;
            tempdata <= (others => '0');
            Address <= Address +'1';
            element <=ele1;
          when others =>
        end case;
      elsif (Address = MAXADDRESS) then
        case element is
          when ele1 => --read zeros
            WEAtemp <=  WEN_ACTIVE;
            WEBtemp <=  WEN_ACTIVE;
            tempdata <= (others => '1');
            element <=ele2;
```

```
when ele2 => --write ones
    WEAtemp <=   not WEN_ACTIVE;
    WEBtemp <=   not WEN_ACTIVE;
    --Address <= MAXADDRESS ; already at MAX!!!
    tempdata <= (others => '1');
    element <=ele1;
    Phase <= Phase3;
    Address <= MAXADDRESS ;
    tempdata <= (others => '1');
when others =>
end case;

end if;

when phase3 => --  Down R1,WO

if ( Address /= MINADDRESS ) then
  case element is
  when ele1 => --read ones
      WEAtemp <=  WEN_ACTIVE;
      WEBtemp <=  WEN_ACTIVE;
      tempdata <= (others => '0');
      element <=ele2;

  when ele2 => --write zeros
      WEAtemp <= not WEN_ACTIVE;
      WEBtemp <= not WEN_ACTIVE;
      tempdata <= (others => '0');
      Address <= Address -'1';
      element <=ele1;
  when others =>
  end case;
elsif (Address = MINADDRESS) then
  case element is
  when ele1 => --read 1s
      WEAtemp <= WEN_ACTIVE;
      WEBtemp <=WEN_ACTIVE;
      tempdata <= (others => '0');
      element <=ele2;

  when ele2 => --write ones
      WEAtemp <= not WEN_ACTIVE;
      WEBtemp <= not WEN_ACTIVE;
      Phase <=phase4;
      element <=ele1;
  when others =>
  end case;
end if;
when phase4 =>
if (portBtested='1') then
  portBtested<='0';
else
  portBtested<='1';
  Phase <= init;
end if;
when others =>
end case;



---------------------------MARCH S2PF-------DTM----------------------
when March_s2pf =>
  case Phase is
  when dummy =>
    EnAtemp <= EN_LEVEL;
    EnBtemp <= EN_LEVEL;
    SSRtemp <= EN_LEVEL;
```

```
    Phase <= Init;
when Init =>
    element<=ele1;
    Address <= MINADDRESS;
    WEAtemp <= WEN_ACTIVE;
    WEBtemp <= not(WEN_ACTIVE);
    SSRtemp <= not EN_LEVEL;
    tempdata <= (others => '0');
    tempdatab <= (others => '0');
    Phase <= Phase1;
    EnAtemp <= EN_LEVEL;
    EnBtemp <= EN_LEVEL;
when phase1=>--  M0 up write 0s
    if ( Address /= MAXADDRESS ) then
        Address <= Address + '1';
        WEAtemp <= WEN_ACTIVE;
        WEBtemp <= not(WEN_ACTIVE);
        tempdata <= (others => '0');

    else
        Address <= MINADDRESS;
        WEAtemp <= not(WEN_ACTIVE);
        tempdata <= (others => '0');
        Phase <=Phase2;
    - end if;
when Phase2=>
    case element is
        when ele1 => --2
            tempdata<=(others => '0');
            element<=ele2;
        when ele2=>--3
            element<=ele3;
        when ele3=>   --4
            WEAtemp<=(WEN_ACTIVE);
            tempdata <= (others => '1');
            tempdataB <= (others => '0');
            --port B output should have zero's on it still.
            element<=ele4;
        when ele4 =>  --goes to next RAM address or next march
            if (Address /=MAXADDRESS) then
                Address <=Address +'1';
                WEAtemp<=not(WEN_ACTIVE);
                element <=ele1;
            else -- done with March M1
                Address <= MINADDRESS;
                WEAtemp <= not(WEN_ACTIVE);
                tempdata <= (others => '0');
                element<=ele1;
                Phase <=Phase3;
            - end if;
        when others =>
    end case;
when Phase3 => --M2
    case element is
        when ele1 => --5
            element<=ele2; --reading 1s from each port and then reading
again.
        when ele2=> --6
            element<=ele3; --done reading same address twice
        when ele3=>   --7
            WEAtemp<=(WEN_ACTIVE);
            tempdata <= (others => '0');
            tempdataB <= (others => '1');
                --port B output should have 1s on it still.
            element<=ele4;
        when ele4 =>
            if (Address /=MAXADDRESS) then
                Address <=Address +'1';
                WEAtemp<=not(WEN_ACTIVE);
```

```
        element <=ele1;
      else -- done with March M2
        Address <= MAXADDRESS;
        WEAtemp <= not(WEN_ACTIVE);
        --OEN <= OEN_ACTIVE;
        tempdata <= (others => '0');
        Phase <=Phase4;
        element <=ele1;
      end if;
    when others =>
  end case;
when Phase4 => --M3
  case element is
    when ele1 => --8
      element<=ele2; --reading zeros from each port and then readi
ng again.
    when ele2=>  --9
      element<=ele3; --done reading same address twice
    when ele3=>  --10
      WEAtemp <= (WEN_ACTIVE);
      tempdata <= (others => '1');
      tempdataB <= (others => '0');
        --port B output should have zero's on it still.
      element<=ele4;
    when ele4 =>
      if (Address /=MINADDRESS) then
        Address <=Address - '1';
        WEAtemp <= not(WEN_ACTIVE);
        element <=ele1;
      else -- done with March M3
        Address <= MAXADDRESS;
        WEAtemp <= not(WEN_ACTIVE);
        tempdata <= (others => '0');
        element<=ele1;
        Phase <=Phase5;
      end if;
    when others =>
  end case;

when Phase5 =>
  case element is
    when ele1 => --11
      element<=ele2; --reading zeros from each port and then readi
ng again.
    when ele2=>  --12
      element<=ele3; --done reading same address twice
    when ele3=>  --13
      WEAtemp <= (WEN_ACTIVE);
      tempdata <= (others => '0');
      tempdataB <= (others => '1');
        --port B output should have zero's on it still.
      element<=ele4;
    when ele4 =>
      if (Address /=MINADDRESS) then
        Address <=Address - '1';
        WEAtemp <=not(WEN_ACTIVE);
        element <=ele1;
      else -- done with March M4
        Address <= MAXADDRESS;
        WEAtemp <= not(WEN_ACTIVE);
        --OEN <= OEN_ACTIVE;
        tempdata <= (others => '0');
        element<=ele1;
        Phase <=Phase6;
      end if;
    when others =>
  end case;
when Phase6 => --14
  if ( Address /= MINADDRESS ) then
```

```
            Address <= Address - '1';
            WEAtemp <= not(WEN_ACTIVE);
            tempdata <= (others => '0');
          else
            Element <= ele1;
            Phase <=init;
          --    testmode <= March_s2pf; --ADDRESS already at LOWER BOUND for
next test session
          - end if;
        when others =>
      end case;

    ------------------------------MARCH D2PF------------------------------

    when March_d2pf =>
      case Phase is
        when dummy =>
          phase <= init;
          EnAtemp <= EN_LEVEL;
          EnBtemp <= EN_LEVEL;
          Phase <= Init;
          SSRtemp <= EN_LEVEL;
        when init =>
          SSRtemp <= not(EN_LEVEL);
          phase <= Phase1;
          Address <= MAXADDRESS;
          AddressB <= MAXADDRESS;
          WEAtemp <= WEN_ACTIVE;
          WEBtemp <= not(WEN_ACTIVE);
          tempdata <= (others => '0');
          tempdataB <= (others => '0');
          element <= ele2;
        when  Phase1 =>
          if ( Address = MINADDRESS) then
            Phase <= Phase2;
            AddressB <= MINADDRESS + '1';
            WEAtemp <= WEN_ACTIVE;
            WEBtemp <= not(WEN_ACTIVE);
            tempdata <= (others=>'1');
            tempdataB <= (others=>'0');
            element <= ele1;
          elsif (Address /= MINADDRESS and element /= ele2) then
            Address <= Address - '1';
            AddressB <= MAXADDRESS;
            WEAtemp <= WEN_ACTIVE;
            WEBtemp <= not(WEN_ACTIVE);
            tempdata <= (others=>'0');
            tempdataB <= (others=>'0');
          elsif ( Address /= MINADDRESS and element = ele2 ) then
            AddressB <= MAXADDRESS;
            WEAtemp <= WEN_ACTIVE;
            WEBtemp <= not(WEN_ACTIVE);
            tempdata <= (others=>'0');
            tempdataB <= (others=>'0');
            element <= ele1;
          end if;
        when Phase2 =>
          if ( element = ele1 ) then
            if ( Address = MINADDRESS) then
              AddressB <= MAXADDRESS; -- (r1_r:r0_MAX) for low address
              WEBtemp <= not(WEN_ACTIVE);
              tempdataB <= (others=>'0');
            elsif ( Address /= MINADDRESS) then
              AddressB  <= Address - '1';
              WEBtemp <= WEN_ACTIVE;
              tempdataB <= (others=>'1');
            end if;-- r1_r : w1_r-1
            WEAtemp <= not WEN_ACTIVE;
            tempdata <= (others=>'1');
```

```
--tempdataB <= (others=>'1');
element <= ele2;
elsif ( element = ele2 ) then  -- w0_r: r1_r-1
  if ( Address = MINADDRESS) then
    AddressB <= MAXADDRESS; -- (w0_r:r0_MAX) for low address

    tempdataB <= (others=>'0');
  elsif ( Address /= MINADDRESS) then
    AddressB  <= Address - '1';
    tempdataB <= (others=>'1');
  end if;
  WEBtemp <= not(WEN_ACTIVE);
  WEAtemp <= WEN_ACTIVE;
  tempdata <= (others=>'0');
  element <= ele3;

elsif (element = ele3 ) then   -- r0_r:w0_r+1
  AddressB <= Address + '1';
  WEAtemp <= not(WEN_ACTIVE);
  WEBtemp <= WEN_ACTIVE;
  tempdata <= (others=>'0');
  tempdataB <= (others=>'0');
  element <= ele4;
elsif ( element = ele4) then

  if ( Address = MAXADDRESS - '1') then
    Phase <= init;
    Address <= MAXADDRESS;
    AddressB <= MAXADDRESS;
    WEAtemp <= WEN_ACTIVE;
    WEBtemp <= not(WEN_ACTIVE);
    tempdata <= (others=>'0');
    tempdataB <= (others=>'0');
    element <= ele1;

  elsif ( Address /= MAXADDRESS - '1') then
    Address <= Address + '1';
    AddressB <= Address + "10";
    WEAtemp <= WEN_ACTIVE;
    WEBtemp <= not(WEN_ACTIVE);
    tempdata <= (others=>'1');
    tempdataB <= (others=>'0');
    element <= ele1;
  end if;
end if;
when others =>

end case;
when others =>
end case;
end if;
end process;
end;
```

Appendix B

FIFOTPG VHDL Source code

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
   use IEEE.STD_LOGIC_ARITH.ALL;
   use IEEE.STD_LOGIC_UNSIGNED.ALL;

 entity FIFO_TPG is
   Port ( CLK,DRCK,UPDATE,SHIFT,TDI : in  STD_LOGIC;
      Reset: in STD_LOGIC;
      DI: out STD_LOGIC_VECTOR(31 downto 0);
      DIP: out STD_LOGIC_VECTOR(3 downto 0);
      RST,WREN,RDEN: out STD_LOGIC);
 end FIFO_TPG;

 architecture Behavioral of FIFO_TPG is
   type phases is (RESET_FIFO,Phase1,phase2,phase4);
   type elements is(ele1,ele2,ele3,ele4,ele5,ele6);
   signal element : elements :=ele1;
   signal phase : phases := reset_fifo;
   signal MAXCOUNT: STD_LOGIC_VECTOR(12 downto 0);
   signal MINCOUNT: STD_LOGIC_VECTOR(12 downto 0);
   signal COUNT:    STD_LOGIC_VECTOR(12 downto 0);
   signal tempdata: STD_LOGIC_VECTOR(35 downto 0);
   signal SR, PDO : STD_LOGIC_VECTOR(4 downto 0);
   signal MODE: std_logic_vector(1 downto 0);
   signal RDENtemp,WRENtemp,RDEN_level,WREN_level, RST_level: std_logic;
 begin
   MINCOUNT <= (others =>'0');
 bsync:
   process (DRCK, UPDATE,SHIFT,SR) begin
     if (DRCK'event and DRCK = '1') then
       if (SHIFT = '1') then
         for I in 0 to 3 loop
           SR(I) <= SR(I+1);
         end loop;
         SR(4) <= TDI;
       end if;
     end if;
     if (UPDATE = '1') then PDO <= SR;
     end if;
   end process bsync;
   RDEN_level <= PDO(2);
   WREN_level <=PDO(3);
   RST_level <= PDO(4);
   MODE(0)<=PDO(0);
   MODE(1)<= PDO(1);

   Process(MODE,tempdata,WRENtemp,RDENtemp)
   begin
     if (MODE="00") then
       MAXCOUNT <= "1000000000000"; --4096
       DI       <= X"0000000"&tempdata(3 downto 0);
       DIP      <= "0000";
       WREN <= WRENtemp;
       RDEN <= RDENtemp;
     elsif (MODE="01") then
       MAXCOUNT <= "0100000000000"; --2049
       DI       <= X"000000"&tempdata(7 downto 0);
       DIP      <= "000"&tempdata(35);
       WREN <= WRENtemp;
       RDEN <= RDENtemp;
     elsif (MODE="10") then
       MAXCOUNT <= "0010000000000"; --1025
       DI       <=X"0000"&tempdata(15 downto 0);
       DIP      <= "00"&tempdata(35 downto 34);
       WREN <= WRENtemp;
       RDEN <= RDENtemp;
     elsif (MODE="11") then
       MAXCOUNT <= "0001000000000"; --513
       DI       <= tempdata(31 downto 0);
       DIP      <= tempdata(35 downto 32);
```

```
WREN <= WRENtemp;
RDEN <= RDENtemp;
end if;
end process;
Process(Reset, Clk,RDEN_level,WREN_level,RST_level )
begin
if ( Reset = '1' ) then
tempdata<=(others =>'0');
COUNT <= MINCOUNT;
phase <= RESET_FIFO;
rst<=RST_level;
RDENtemp<=not(RDEN_level);
WRENtemp<=not(WREN_level);
elsif (Clk = '1' and Clk'Event) then
case Phase is
when RESET_FIFO =>
case element is
when ele1 =>
rst <=RST_level;
element <= ele2;
when ele2 =>
rst <= RST_level;
element <=ele3;
when ele3 =>
element <= ele4;
when ele4 =>
element <=ele5;
when ele5 =>
phase <= phase1;
element <=ele1;
count <= MINCOUNT;
RDENtemp <= not RDEN_level;
WRENtemp <= WREN_level;
rst <= not RST_level;
when others =>
end case;
when Phase1=>
if ( COUNT <= MAXCOUNT) then
COUNT <= COUNT + '1';
WRENtemp <= WREN_level;
tempdata <= (others => '0');
else
COUNT <=MINCOUNT;
Phase <=Phase2;
WRENtemp <= not (WREN_level);
RDENtemp <= RDEN_level;
tempdata <=(others =>'0');
end if;
when phase2 =>
case Element is
when ele1 =>
if ( COUNT <=MAXCOUNT) then
RDENtemp <=not RDEN_level;
WRENtemp <=not(WREN_level);
tempdata <=(others =>'1');
Element <= ele2;
end if;
when ele2 =>
RDENtemp <=not RDEN_level;
WRENtemp <=not(WREN_level);
tempdata <=(others =>'1');
Element <= ele3;
when ele3 =>
RDENtemp <=not RDEN_level;
WRENtemp <=not(WREN_level);
tempdata <=(others =>'1');
Element <= ele4;
when ele4 =>
RDENtemp <=not RDEN_level;
```

```
WRENtemp <=(WREN_level);
Element <= ele5;
when ele5 =>
  RDENtemp <= not RDEN_level;
  WRENtemp <=(WREN_level);
  element <= ele6;
when ele6=>
  RDENtemp <=(RDEN_level);
  WRENtemp <=not WREN_level;
  tempdata <=(others =>'0');
  COUNT <= COUNT + '1';
  if ( COUNT=MAXCOUNT) then
    Phase <= phase4;
    COUNT<=MINCOUNT;
    RDENtemp <=(RDEN_level);
    WRENtemp <=not WREN_level;
    tempdata <= (others => '1');
  end if;
  Element <= ele1;
when others =>
end case;
when phase4 =>  --read 001s from FIFO
  if ( COUNT <= MAXCOUNT ) then
    RDENtemp <=RDEN_level;
    WRENtemp <=not(WREN_level);
    tempdata <=(others =>'1');
    COUNT <= COUNT + '1';
  else
    COUNT<=MINCOUNT;
    WRENtemp <= not WREN_level;
    RDENtemp <= not RDEN_level;
    tempdata <= (others => '0');
    Phase <=reset_fifo;
  end if;
end case;
end if;
end process;
end Behavioral;
```

Appendix C

ECCTPG VHDL Source code

```
library IEEE;
   use IEEE.STD_LOGIC_1164.ALL;
   use IEEE.STD_LOGIC_ARITH.ALL;
   use IEEE.STD_LOGIC_UNSIGNED.ALL;


 entity ECC_TPG is
   Port (CLK : in  STD_LOGIC;
         RESET : in  STD_LOGIC;
         Data : out std_logic_vector(71 downto 0);
         ADDRB : out std_logic_vector(8 downto 0);
         WREN,RDEN: out std_logic);

 end ECC_TPG;

 architecture Behavioral of ECC_TPG is


   signal CLK_EN: std_logic;
   type phases is (Init_RAMS,Write_RAMS) ;
   signal phase : phases := Init_RAMS ;
   type elements is (ele1,ele2,ele3) ;
   signal element : elements :=ele1 ;
   signal Address : std_logic_vector ( 8 downto 0):= (others => '0') ;
   signal MAXADDRESS : std_logic_vector ( 8  downto 0):= (others =>'1') ;

   component ECC_fsm is
      port (
      CLK: in std_logic;
      CLK_EN: in std_logic;
      RESET: in std_logic;
      DATA_OUT: out std_logic_vector(71 downto 0));

   end component;
 begin

   ECC_fsm_inst: ECC_fsm port map(clk,CLK_EN,reset,data) ;


   Process( Reset, Clk )
   begin
      if ( Reset = '1' ) then
         Address <= "111111110";
         CLK_EN <='1';
         RDEN <= '0';
         WREN <= '1';
         Phase <= Init_RAMs;
         element <= ele1;
      elsif (Clk = '1' and Clk'Event) then

         case Phase is
            when Init_RAMS=>
               if (Address < MAXADDRESS) then
                  case element is
                     when ele1 =>
                        Address <= "111111110";
                        element<=ele2;
                     when ele2 =>
                        element <= ele3;
                        CLK_EN <='0';
                        WREN <='0';
                        RDEN <='1';
                        Address <= (others =>'0');
                     when ele3 =>
                        Address <=Address + '1';
                  end case;
               else
                  RDEN <='0';
                  WREN <= '1';
```

```
            Phase <= Write_RAMs;
            element <=ele1;
          end if;

        when Write_RAMS =>
          Address <= (others => '0');
          case Element is
            when ele1 =>
              CLK_EN <= '1';
              RDEN <= '0';
              WREN <= '1';
              element <= ele2;
            when ele2 =>
              CLK_EN <= '0';
              WREN <= '0';
              RDEN <='1';
              element <= ele1;
            when others =>
          end case;
        end case;
      end if;
    end process;
    ADDRB <= Address;
end Behavioral;


entity ECC_fsm is
  port (
    CLK: in std_logic;
    CLK_EN: in std_logic;
    RESET: in std_logic;
    DATA_OUT: out std_logic_vector(71 downto 0));

end ECC_fsm;

architecture Behavioral of ECC_fsm is
  component shifter is
    Port (
      CLK_EN: in std_logic;
      CLK : in std_logic;
      RESET: in std_logic;
      ENABLE: in std_logic;
      BIT_IN: in std_logic;
      BIT_OUT: out std_logic;
      DOUT: out std_logic_vector(71 downto 0));

    end component;
  signal DONE_check,BIT_IN1,BIT_IN2,BIT_OUT1: std_logic;
  signal DOUT1, DOUT2 : std_logic_vector(71 downto 0);

begin

  SR1: shifter port map(
        CLK_EN => CLK_EN,
        CLK => CLK,
        RESET => RESET,
        ENABLE => '1',
        BIT_IN => BIT_IN1,
        BIT_OUT => BIT_OUT1,
        DOUT => DOUT1
        );

  SR2: shifter port map(
        CLK_EN => CLK_EN,
        CLK => CLK,
        RESET => RESET,
        ENABLE => BIT_OUT1,
        BIT_IN => BIT_IN2,
        DOUT => DOUT2
```

```
        );

    process(DOUT1, DOUT2,BIT_IN1,BIT_IN2)
      variable temp2:std_logic;
      variable temp1:std_logic;
    begin
      temp1:='0';
      temp2:='0';
      for i in 0 to 71 loop
        temp1:=temp1 OR DOUT1(i);
      end loop;

      for i in 0 to 71 loop
        temp2:=temp2 OR DOUT2(i);
      end loop;

      BIT_IN1<=not temp1;
      BIT_IN2<=not temp2;
    end process;
    DATA_OUT <= DOUT1 OR DOUT2;

end Behavioral;

entity shifter is
    Port (
    CLK_EN: in std_logic;
    CLK : in std_logic;
    RESET: in std_logic;
    ENABLE: in std_logic;
    BIT_IN: in std_logic;
    BIT_OUT: out std_logic;
    DOUT: out std_logic_vector(71 downto 0));

end shifter;

architecture Behavioral of shifter is

    signal DATA: std_logic_vector(71 downto 0);

begin
    process(CLK,RESET)
      variable temp:std_logic :='0';
    begin
      temp:=Data(63);
      if (RESET='1') then
        DATA <=X"000000000000000000";

      elsif (CLK='1' and CLK'event)  then
        if (enable='1' and CLK_EN ='1') then
          for i in 0 to 70 loop
            DATA(i+1)<=Data(i);
            Data(0)<=BIT_IN;

          end loop;
        end if;
      end if;
      BIT_OUT<=temp;
    end process;
    DOUT <=DATA;
end Behavioral;
```

Appendix D

CASTPG VHDL Source code

```
library ieee;
   use ieee.std_logic_1164.all;
   use ieee.std_logic_unsigned.all;
 entity fsm is
   generic(
     EN_Level: std_logic := '1';
     WEN_ACTIVE: std_logic := '1');
   port (
     Reset : in std_logic;
     Clk : in std_logic;
     WEA : buffer std_logic;
     DIA : out std_logic_vector( 0 downto 0);
     ADDRA : out std_logic;
     EnA: out std_logic;
     EnB : out std_logic);

 end fsm;
 architecture  BEHAVIORAL of fsm is
   type phases is (Init,dummy,Phase1,phase2,phase3,phase4,phase5,phase6);
   type elements is (ele1,ele2);
   signal phase : phases := dummy;
   signal Element : elements := ele1;
   signal Address : std_logic;
   signal portBtested: std_logic:='0';
   signal tempdata: std_logic;
   signal ENATEMP,ENBTEMP,WEAtemp: std_logic;


   begin

      p0:
      Process(clk,Reset,tempdata,Address,WEA)
      begin
        if ( Reset = '1' ) then
           ADDRA <= '0';
           DIA(0) <= '0';
           EnA <= not EN_LEVEL;
           EnB <= not EN_LEVEL;
           WEA <= not WEN_ACTIVE;

        elsif (Clk = '1' and Clk'Event) then


           DIA(0)<=tempdata;
           ADDRA<=Address;
           WEA<=WEAtemp;
           EnA<=EnAtemp;
           EnB<=EnBtemp;

        end if;
      end process;

   p1:
      Process(Clk)
      begin
        if (Reset ='1') then
           tempdata <= '0';
           Address <= '0';
           Element <= ele1;
           Phase <= dummy;
           WEAtemp <= not(WEN_ACTIVE);
           EnAtemp <= not EN_LEVEL;
           EnBtemp <= not EN_LEVEL;
        --
        elsif (Clk = '1' and Clk'Event) then


           case Phase is
              when dummy =>
```

```
EnAtemp <= not EN_LEVEL;
EnBtemp <= not EN_LEVEL;
Phase <= Init;
portbtested <='0';

when Init =>
Address <= '0';
WEAtemp <= WEN_ACTIVE;
tempdata <= '0';
Phase <= Phase1;
element <= ele1;
if (portbtested ='1') then
EnAtemp <= not EN_LEVEL;
EnBtemp <= (EN_LEVEL);
portbtested<='0';
else
EnAtemp <= EN_LEVEL;
EnBtemp <= not(EN_LEVEL);
end if;

when phase1=>--
case element is
when ele1 =>
Address <= '1';
WEAtemp <= WEN_ACTIVE;
element <= ele2;
when ele2 =>
Address <='0';
WEAtemp <=not WEN_ACTIVE;
tempdata <= '1';
Phase <= Phase2;
element <= ele1;
end case;

when phase2=>--
case element is
when ele1 =>
Address <= '0';
WEAtemp <= WEN_ACTIVE;
element <= ele2;
when ele2 =>
Address <='1';
WEAtemp <=not WEN_ACTIVE;
Phase <= Phase3;
element <= ele1;
end case;

when phase3=>--
case element is
when ele1 =>
Address <= '1';
WEAtemp <= WEN_ACTIVE;
element <= ele2;
when ele2 =>
Address <='0';
WEAtemp <=not WEN_ACTIVE;
tempdata <= '0';
Phase <= Phase4;
element <= ele1;
end case;

when phase4=>
case element is
when ele1 =>
Address <= '0';
WEAtemp <= WEN_ACTIVE;
element <= ele2;
when ele2 =>
Address <='1';
```

```
            WEAtemp <=not WEN_ACTIVE;
            tempdata <= '0';
            Phase <= Phase5;
            element <= ele1;
        end case;

    when phase5=>--
        case element is
          when ele1 =>
            Address <= '1';
            WEAtemp <= WEN_ACTIVE;
            element <= ele1;
            Phase <= Phase6;
          when others =>
        end case;
    when phase6 =>

        if (portBtested='1') then
          portBtested<='0';
        else
          portBtested<='1';
          Phase <= init;
        end if;
    when others =>
    end case;
  end if;
end process;
end;
```

Appendix E

List of Acronyms

ATE - Automatic Test Equipment

BIST - Built-in Self Test

BRAM- Block RAM

BSCAN - Boundary Scan

BUT - Block under Test

CAD - Computer-aided Design

CUT - Circuit under Test

DFT - Design for Test

DSP - Digital Signal Processor

DUT - Device under Test

ECC - Error Correcting Code

FF - Flip-flop

FIFO - First-in First-out

FPGA - Field Programmable Gate Array

FWFT - First-Word-Fall-Through

GUI - Graphical User Interface

HDL - Hardware Description Language

I/O - Input / Output

IC - Integrated Circuit

IP - Intellectual Property

LUT - Look-up Table

LSB - Least Significant Bit

MMTPG - Multi-march Test Pattern Generator

MSB - Most Significant Bit

ORA - Output Response Analyzer

PIP - Programmable Interconnect Point

PLB - Programmable Logic Block

PowerPC - PPC

RAM - Random Access Memory

SERDES - Serial / Deserial

SoC - System-on-Chip

SRAM - Static Random Access Memory

TCK - Test Clock

TDI - Test Data In

TDO - Test Data Out

TMS - Test Mode Select

TPG - Test Pattern Generator

VLSI - Very Large Scale Integration