Built-In Self-Test of Programmable Resources in Microcontroller Based System-on-Chips

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.

____

John Sunwoo

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

____

Stephen L. McFarland
Acting Dean
Graduate School

BUILT-IN SELF-TEST OF PROGRAMMABLE RESOURCES IN MICROCONTROLLER BASED SYSTEM-ON-CHIPS

John Sunwoo

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
Dec 16, 2005

BUILT-IN SELF-TEST OF PROGRAMMABLE RESOURCES IN

MICROCONTROLLER BASED SYSTEM-ON-CHIPS

John Sunwoo

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

VITA

John Sunwoo, son of Changshin and Jungjean Sunwoo, was born on November 14, 1980 in Gwangju, Korea. He graduated from High School Attached to Chosun University in 1999. He graduated with a Bachelor of Science degree in Electrical Engineering with a major in Computer Engineering at Auburn University in May 2003. After completion of his undergraduate degree, he entered the graduate program in Electrical and Computer Engineering at the same institute in August 2003. While in pursuit of his Master of Science degree at Auburn University, he worked under the guidance of Dr. Charles E. Stroud as a graduate student research assistant in the Auburn University Built-In Self-Test (AUBIST) laboratory.

THESIS ABSTRACT

BUILT-IN SELF-TEST OF PROGRAMMABLE RESOURCES IN MICROCONTROLLER BASED SYSTEM-ON-CHIPS

John Sunwoo

Master of Science, Dec 16, 2005
(B.S.E.E, Auburn University, Alabama, 2003)
85 Typed Pages

Directed by Dr. Charles E. Stroud

System-on-Chip (SoC) implementations typically incorporate embedded Field Programmable Gate Array (FPGA) cores to take advantage of the programmable logic and routing resources provided by FPGAs. Testing the FPGA core typically requires numerous configuration downloads to completely test the various modes of operation of the programmable logic resources and the size of each configuration download file is large due to large amount of programmable resources. However, the ability to perform dynamic partial reconfiguration of the FPGA core from embedded processor core opens new opportunities for testing the FPGA using Built-In Self-Test (BIST). This thesis discusses the implementation of BIST for FPGA cores using partial dynamic reconfiguration from the embedded processor. As a result, all external configuration downloads are eliminated and replaced by one single processor program that programs

the FPGA core for BIST, executes the BIST sequence, retrieves the BIST results, and executes diagnostic procedures to locate and identify faults detected by the BIST. Total testing time is improved by as much as a factor of 45 and a configuration memory storage requirement by as much as a factor of 83 by using dynamic partial reconfiguration compared to the traditional approach that requires BIST configuration downloads for every mode of operation of the programmable logic resources in the FPGA core of the Atmel AT94K series SoCs.

ACKNOWLEDGMENTS

I am greatly indebted to Dr. Charles E. Stroud for his guidance and support during this study. He helped me becoming a better engineer not only with his technical assistance, but also with his moral support. Also, I would like to express sincere appreciation to my fellow research colleagues Srinivas, Jonathan, Sudheer, Sachin and Adam for their advice, concern and friendship. I reserve special thanks to Jinsung who has given so much of her time for the completion of this thesis. Additional appreciation is extended to my committee members Dr. Victor Nelson and Dr. Thaddeus Roppel for their critical reading of the thesis and their helpful suggestions.

Finally, I would like to express my deepest gratitude to my parents, and brothers (Jin and Nelson). Through their support, encouragement and love I found the strength to pursue my goals.

Style manual or journal used: IEEE (Institute of Electrical and Electronic Engineers) Journal style

Computer software used: Microsoft Office Word 2003, Microsoft Office Visio 2003

TABLE OF CONTENTS

LIST OF FIGURES xi

LIST OF TABLES xiii

CHAPTER ONE 1

1.1 Overview of SoCs 2

1.2 Overview of FPGA Core 3

1.3 SoC Testing 5

1.3.1 Overview of Built-In Self-Test 5

1.3.2 Merits of BIST 6

1.4 BIST for FPGAs 6

1.5 Thesis Statement 7

CHAPTER TWO 11

2.1 Architecture of Atmel AT94K Series FPSLIC SoCs 11

2.1.1 FPGA Core Architecture 12

2.1.2 AVR Microcontroller Architecture 16

2.1.3 RAM Architecture 18

2.1.4 FPGA-RAM-AVR Interface 19

2.2 Special Features in the AT94K 20

2.2.1 Cache Logic Mode 20

2.2.2 Use of Macro Generation Language (MGL) 22

2.3 Overview of BIST for Embedded FPGA Core in the Atmel FPSLIC 23

2.4 Thesis Restatement 27

CHAPTER THREE ……………………………………………..………………………29

3.1 Implementation of ORA and Shift Register …………………………………….. 30

3.2 Implementing Shift Register Reconfiguration for Logic BIST …………………. 33

3.3 Dynamic AVR Reconfiguration of BUTs and ORAs for BIST …………………. 37

3.4 A Better Logic BIST Sequence ……………………………………………….. 42

CHAPTER FOUR …………………………………….…………………………..47

4.1 Development of C Program for Logic BIST Generation ………………………...47

4.1.1 Implementation Issues and Considerations …………………………………48

4.1.2 Efficient Sequence of On-chip Dynamic Configuration of FPGA BIST from AVR ……………………………………………………………………………..50

4.2 Debugging Technique for Developing Logic BIST from Scratch ………………. 56

4.3 Experimental Results ……………………………………………………………. 57

CHAPTER FIVE ………………………………….………………………………... 62

5.1. Summary …………………………………………………………………..…….. 62

5.2. Improvements in Total Test Time and Configuration Memory Requirements …. 64

5.3. Main Contribution ……………………………………………………………….66

5.4. Future Research ………………………………………………………………… 67

REFERENCES ………………………………..…………………………………….. 69

LIST OF FIGURES

Figure 1.1 Basic Structure of Microcontroller Based SoC with FPGA Core 2

Figure 1.2 General FPGA Structure and Configurable Interconnect Points 4

Figure 1.3 Basic BIST Architecture 5

Figure 1.4 Basic Logic BIST Structure 7

Figure 2.1 Symmetrical FPGA Core Surrounded by I/O 12

Figure 2.2 Cell-to-Cell Connections & PLB Cell 13

Figure 2.3 Cell-to-Bus Connections 13

Figure 2.4 Configurable Interconnect Point Structure and Types [12] 14

Figure 2.5 Basic Modes of Horizontal Repeater 15

Figure 2.6 Banked Clock & Set/Reset for One Column of PLB Cells 16

Figure 2.7 AVR Core Architecture 17

Figure 2.8 FPGA-RAM-AVR Interface 19

Figure 2.9 AVR-FPGA Dynamic Cache Logic 21

Figure 2.10 Cell Reconfiguration Method 21

Figure 2.11 Basic Bist Structure- Logic Bist 23

Figure 2.12 Basic Comparison Based ORA Structure 24

Figure 2.13 FPGA BIST Structure for Complete Test 25

Figure 2.14 Diagnosis PLBs from Analyzing Comparison Based ORA Results 26

Figure 2.15 Logic BIST Architecture of AT94K Series SoCs 27

Figure 3.1 ORA Structure for Logic BIST 30

Figure 3.2 Two-PLB ORA 31

Figure 3.3 High Level Structure of ORA and After the Reconfiguration 31

Figure 3.4 Comparison ORA and the ORA After Reconfiguration 32

Figure 3.5 Shift Register Layout 33

Figure 3.6 AVR Code of ORA Reconfiguration to Shift Register 35

Figure 3.7 Four Layouts for Logic BIST [33] 39

Figure 3.8 AVR Code of BUT Reconfiguration and ORA Initialization ……………….. 40
Figure 3.9 Four BIST Phases in One Session for AT94K SoCs ………………………... 43
Figure 4.1 Illustration of Configuration Byte Shared by More then One Resources ….. 50
Figure 4.2 FIGARO Illustration of How AVR Connects to a Global Clock Buffer ……. 53
Figure 4.3 AVR Code of Generating N Clock Cycles to 'FPGAIOWE' ……………….. 54
Figure 4.4 ADINO Pad connected from Scan Chain …………………………………… 55
Figure 4.5 Use of MGL to Verify AVR Routines ……………………………………… 57

LIST OF TABLES

Table 1.1 Advantages vs Disadvantages BIST [17] ……………………………………6
Table 1.2 List of Acronyms Used ……………………………………………………... 9
Table 3.1 Logic BIST Reconfiguration …………………………………………………41
Table 3.2 Total Memory Reduction …………………………………………………….. 41
Table 3.3 Total Test Time and Speed Up ……………………………………………….. 42
Table 3.4 Logic BIST Reconfiguration Improvement ………………………………….. 45
Table 3.5 Total Memory Reduction …………………………………………………….. 46
Table 3.6 Total Test Time and Speed Up ……………………………………………….. 46
Table 4.1 Total Configuration Routine Analysis ……………………………………...58
Table 4.2 Actual Download File Size (Kbytes) ……………………………………… 60
Table 4.3 Total Memory Reduction …………………………………………………….. 60
Table 4.4 Total Test Time and Speed Up ……………………………………………… 60
Table 5.1 Logic BIST Reconfiguration Comparison …………………………………… 65
Table 5.2 Total Configuration Memory Reduction …………………………………….. 65
Table 5.3 Total Test Time and Speed Up ……………………………………………… 65

CHAPTER ONE
INTRODUCTION

Developments of System-on-Chips (SoCs) which integrate high performance processors, programmable logic and interconnect resources and a considerable amount of memory in a single chip have recently become a popular trend. An SoC is also referred to as "System Large Scale Integration (System LSI)" or "System Integrated Circuit (System IC) [6]." SoC technology is the packaging of various kinds of digital system components on a single IC, where systems could only be implemented on Printed Circuit Boards (PCBs) in the past. SoC technology is making rapid progress because it is essential to realizing inevitable trends in modern electronic devices such as miniaturization, low-power, low-cost, high-speed and high-reliability [6]. Some SoC devices have more processing ability than a typical 10 year-old desktop computer.

As IC technology advances, it not only makes the design and manufacturing process more costly but also makes the testing process after manufacturing even costlier [1]. As a result, the increase in testing cost is much higher compared to the increase in the integration ratio [1]. The architecture of a typical SoC facilitates interaction between the on-chip microcontroller and the Field Programmable Gate Array (FPGA) that contains the programmable logic and interconnection resources. This interaction can assist in the development of fault detection tests as well as fault recovery strategies [5]. Therefore, Built-In Self-Test (BIST) for SoCs is a very attractive solution not only for

systems but also for designers and manufacturers.

1.1 Overview of SoCs

Typical microcontroller based SoCs include an FPGA core, Random Access Memory (RAM), a microcontroller, and peripheral input/output logic [7]. Figure 1.1 shows the typical SoC structure.

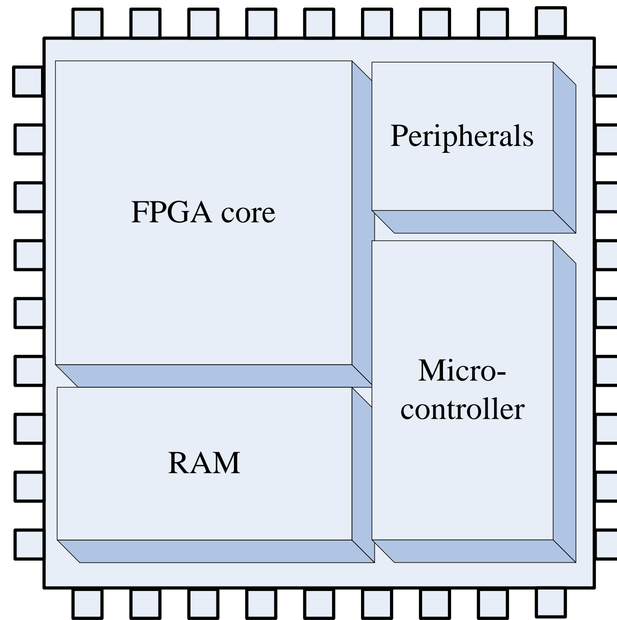


Figure 1.1 Basic structure of microcontroller based SoC with FPGA core

Current FPGAs are capable of higher logic capacity than the earlier programmable logic devices [22]. FPGA cores provide the reconfigurable resources within most microcontroller based SoCs. A detailed FPGA core structure is presented in Section 1.2.

RAM, in general, is a storage media where data can be stored or accessed [8]. RAM in SoCs sometimes interfaces with both FPGA core and microcontroller core. It enables data interaction between the microcontroller core and the FPGA core [7]. In particular, Static Random Access Memory (SRAM) is commonly used in SoCs [9]. The

SRAM is important for speed and efficiency of SoCs because it interacts with both FPGA core and microcontroller simultaneously. Program memory is another type of storage media; it stores the programs to be executed by the microcontroller.

A microcontroller is a type of processor that is intended to operate in an embedded system on a single IC. General purpose registers are fixed memory spaces that help the microcontroller to process data faster and more efficiently [7]. Programming of the microcontroller is implemented using assembly or C programming language. This makes the use of microcontrollers in SoCs convenient without time-consuming design and synthesis processes [7].

All components in a SoC are usually linked to each other closely for maximum performance [7]. Internal or external interrupts allow interaction with the microcontroller to initiate execution of certain tasks. Therefore, the FPGA core can generate internal interrupts to the microcontroller.

1.2 Overview of FPGA Core

An FPGA consists of reconfigurable logic blocks, where the logic can be programmed multiple times after it is manufactured [8][10]. Unlike standard ICs, FPGAs can have flexible functionality while having a general structure [11]. As illustrated in Figure 1.2-a, an FPGA consists of an array of programmable logic blocks (PLBs) (usually an M x N array), containing gates, look-up table RAMs, flip-flops, and programmable interconnect wiring. All FPGAs are reprogrammable, since their logic functions and interconnect are defined by the contents of a configuration memory [8].

The PLBs are functional logic units which can be programmed for different modes

of operation such as RAM-based look-up tables (LUTs) for combinational logic functions, flip-flops or latches for sequential logic functions, arithmetic operations, memory functions, etc [12][13]. Usually one PLB consists of multiplexers (MUXs), LUTs, flip-flops, and routing resources.

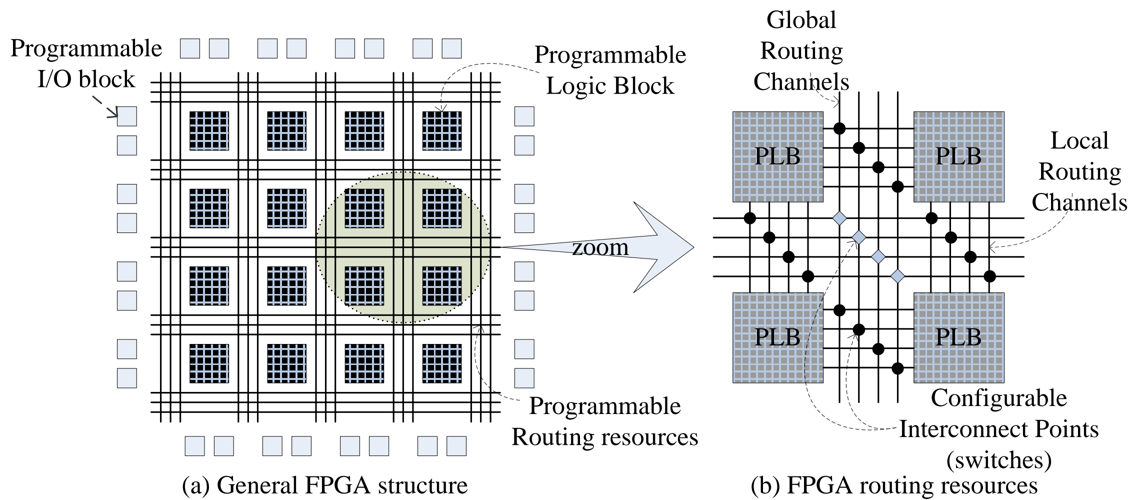


Figure 1.2 General FPGA Structure and Configurable Interconnect Points

There are additional programmable routing resources outside the PLBs. A large number of programmable switches, known as Configurable Interconnect Points (CIPs) or Programmable Interconnect Points (PIPs), are built-in into cross sections of the routing resources. These CIPs enable the internal circuitry of an FPGA to be connected in various network structures [14][15]. Thus, configuring programmable routing resources determines the connectivity between PLBs and other components in the chip. As illustrated in Figure 1.2-b, local routing resources determine connectivity of a PLB to its neighboring PLB and to global routing resources, while global routing resources determine connectivity of a given PLB to non-neighboring PLBs, programmable I/O blocks, or other components in the SoC. [12].

1.3 SoC Testing

Testing of embedded cores in SoCs is a challenging problem as they are deeply embedded in the SoC with a limited number of Input/Output (I/O) pins. As a result, it may not be possible to test all the embedded cores in a SoC using test patterns from external sources [24]. In some companies, more than 30% of the total production cost is due to testing [16]. BIST could be a better approach for testing SoCs as it does not require any external test equipment and test patterns are generated internally by the embedded core itself, thus eliminating the problem of core access [24]. By eliminating external test equipment, the BIST approach reduces the testing time and cost [24].

1.3.1 Overview of Built-In Self-Test

The most fundamental definition of BIST is: 'To design a circuit so that the circuit can test itself and determine whether it is "good" or "bad"' [17]. As shown in Figure 1.3, the Test Pattern Generator (TPG), Output Response Analyzer (ORA), and Test Controller (TC) are the primary components in BIST technology.

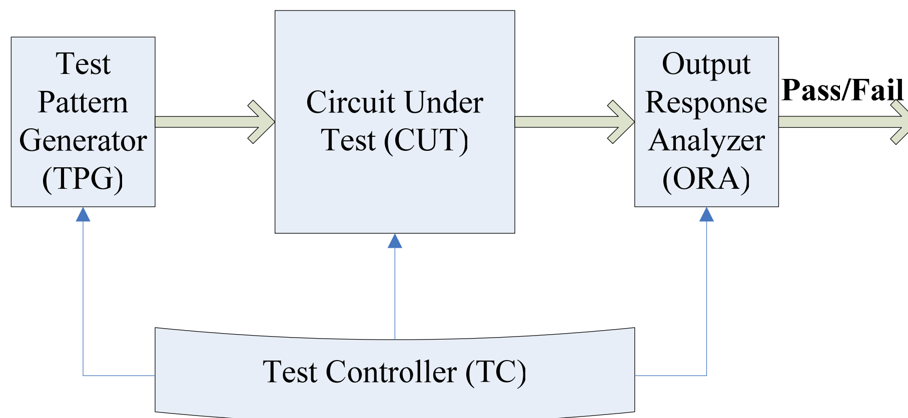


Figure 1.3 Basic BIST architecture

Sets of test vectors generated by the TPG are applied to the Circuit Under Test

(CUT) while the ORA monitors test responses from the CUT in order to determine whether the CUT is good (fault-free) or bad (faulty). The test controller starts the BIST sequence by initializing the target circuit, and it also controls the BIST sequence [12][17].

1.3.2 Merits of BIST

Various chip testing techniques are currently being widely studied. Among them, the BIST approach has excellent advantages compared to its disadvantages, as shown in Table 1. [17]. Eliminating the need for external test equipment as well as reducing manufacturing test time and cost are the main merits of BIST. BIST fits nicely in modern SoC testing because it has good internal access to individual embedded cores which, in most cases, are difficult to access through external I/O pins [9]. For configurable components such as FPGAs, the disadvantages shown in Table 1, such as the area overhead and performance penalties, are no longer a consideration, as will be discussed in the following section.

Table 1.1 Advantages vs Disadvantages BIST [17]

Advantages	Disadvantages
+vertical testability (wafer to system)	-area overhead
+high diagnostic resolution	-performance penalties
+at speed testing	-additional design time & effort
+reduced need for external test equipment	-additional risk to project
+reduced development time & effort	
+more economical burn-in testing	
+reduced manufacturing test time & cost	
+reduced time-to-market	

1.4 BIST for FPGAs

Traditional BIST approaches introduce area overhead and performance penalties [17]. However, BIST for FPGAs removes these associated problems by using the re-programmability of the FPGAs. Initially the FPGA is configured to perform the BIST

operation and, after the test is complete, the chip is reconfigured for its normal system operation [10].

In the general FPGA BIST structure, groups of PLBs in the FPGA are configured to be TPGs, Blocks Under Test (BUTs), and ORAs as shown in Figure 1.4. During each BIST sequence, the BUTs receive identical test patterns from the TPGs and the BUT outputs are compared by the ORAs [18]. The BUTs are reconfigured in a different mode of operation after each BIST sequence until all modes of operation are tested. After all the BIST configurations have been run, a test session is completed [12]. After the first test session is over, the FPGA is configured reversely: BUTs become ORAs and TPGs, and vice versa [19][20][21]. In this way, all PLBs in the FPGA are tested completely.

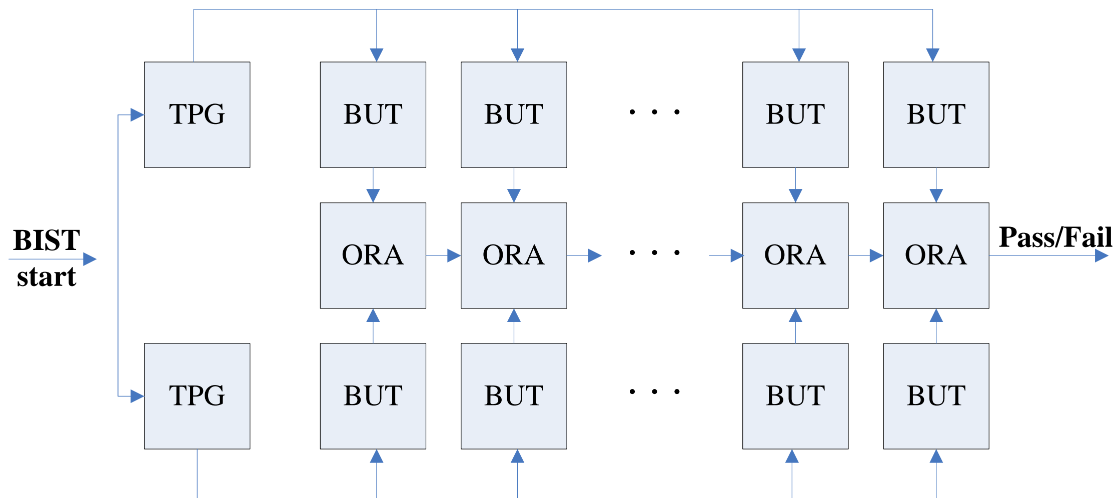


Figure 1.4 Basic Logic BIST Structure

1.5 Thesis Statement

One of the important goals of BIST is to minimize the testing time and cost [18]. For most FPGA BIST approaches, however, reconfiguration time for each BIST

configuration consumes most of the testing time. This is because of the use of external configuration control (such as a PC) and the large number and size of the BIST configuration files that need to be stored in memory and downloaded into the FPGA. For example, the Xilinx 4000XL has 230 BIST configurations [11]. This means the FPGA has to be reconfigured 230 different times to completely test it, and a significant amount of external memory space is needed to store the 230 BIST configurations and a significant amount of time is required to download the configuration data into the device. Moreover, there are additional time requirements to retrieve ORA results [18].

The objective of this research and thesis is to improve FPGA BIST time efficiency on SoCs by utilizing the microcontroller core embedded in the SoC. This thesis focuses on overcoming BIST time and memory storage penalty factors due to the large number of BIST configurations. Unlike traditional FPGA BIST approaches, the computing power of the embedded microcontroller in SoCs can be used to dynamically reconfigure and test the FPGA cores within the SoC boundary, with improved configuration time and memory storage requirements [6].

As a result, there is no need for BIST configurations to be downloaded from the external configuration storage into the FPGA. Only an initial download is done to the program and data memories for the microcontroller. The microcontroller then reconfigures the FPGA core, executes the BIST sequence, and retrieves the ORA results. Only one initial download to the program memory of the SoC is needed, and thus only one configuration needs to be stored in external memory. Alternatively, the BIST configuration program can reside in the program memory for on-demand executions of BIST if the BIST configuration program is sufficiently small.

The proposed BIST approach has been implemented on the Atmel AT94K series FPSLIC (Field Programmable System Level Integrated Circuit). Further details on the embedded microcontroller and FPGA core as well as their interactions in the FPSLIC are described in Chapter 2. Chapter 3 discusses how the microcontroller assists the BIST of the embedded FPGA core to improve the BIST performance. Chapter 4 extends the idea to use the microcontroller as the main BIST component which configures the FPGA for BIST, executes the BIST sequence, retrieves the BIST results and diagnoses faulty PLBs without the need of external configuration downloads. Experimental results for the implementation and application in actual SoCs, along with possible improvements, will also be discussed in each chapter. Finally, Chapter 5 summarizes this research and its significance, along with possible directions for future research and development. A list of acronyms used in this thesis is in shown Table 1.2.

Table 1.2 List of Acronyms Used

ADIN	AVR Data In
ALU	Arithmetic Logic Unit
AVR	Advanced Virtual RISC
BIST	Built-In Self-Test
BUT	Block Under Test
CAD	Computer Automated Design
CIP	Configurable Interconnect Point
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
CUT	Circuit Under Test
DSP	Digital Signal Processing
FF	Flip-Flop
FPGA	Field Programmable Gate Array
FPGAIOWE	FPGA I/O Write Enable

FPGAIORE	FPGA I/O Read Enable
FPSLIC	Field Programmable System Level Integrated Circuit
HDL	Hardware Description Language
IC	Integrated Circuit
IDS	Integrated Development System
I/O	Input/Output
LFSR	Linear Shift Feedback Register
LSI	Large Scale Integration
LUT	Look-Up Table
MIPS	Million Instructions Per Second
MGL	Macro Generation Language
MUX	Multiplexer
ORA	Output Response Analyzer
PC	Personal Computer or Program Counter
PCB	Printed Circuit Board
PIP	Programmable Interconnect Point
PLB	Programmable Logic Block
PWM	Pulse Width Modulation
RISC	Reduced Instruction Set Computer
SoC	System-on-Chip
SRAM	Static Random Access Memory
TC	Test Controller
TPG	Test Pattern Generator
UART	Universal Asynchronous Receiver-Transmitter
VLSI	Very Large Scale Integration
WUT	Wire Under Test
XDL	Xilinx Design Language
XOR	Exclusive OR-Gate

CHAPTER TWO
BACKGROUND

SoCs consist of multiple cores integrated within the same chip boundary. A study described in [25] introduced the method of using the embedded processor to test other cores in the SoC [25]. However, it did not address testing embedded FPGA cores. Proposals such as [26] and [27] suggested using the embedded FPGA core as the main test resource for SoCs. However, a case study of these proposals showed that the FPGA core's limited access to the other cores prevented the thorough test of an SoC [28]. The test limitations due to the architecture of SoCs are the main concern for BIST. In this chapter, the architectural features of Atmel's AT94K series Field Programmable System Level Integrated Circuit (FPSLIC) are described, followed by the features that affect the BIST approaches. An overview of previous work in BIST for the embedded FPGA core in the Atmel AT94K series SoCs is then presented. This chapter concludes with the restatement of this thesis motivation.

2.1 Architecture of Atmel AT94K Series FPSLIC SoCs

The Atmel AT94K series SoC architecture consists of an FPGA core, RAM cores, and an 8-bit Advanced Virtual RISC (Reduced Instruction Set Computer) processor core, denoted as AVR [7]. The individual components have different features for operation in unique modes as well as in mutual aid modes within a system.

2.1.1 FPGA Core Architecture

As illustrated in Figure 2.1, the Atmel FPGA core comprises a symmetrical *NxN* array of identical PLBs, where $N = 48$ for the largest AT94K series device, the AT94K40. The FPGA core is based on a fine-grain architecture that has a large number of small PLBs, each of which is about the one-fourth size of the Xilinx Virtex/Spartan II series PLB [28] [30].

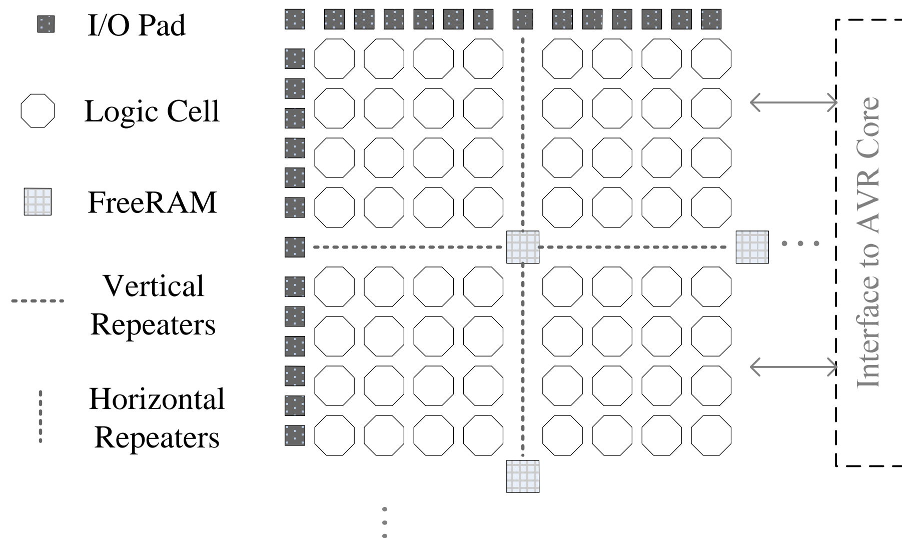


Figure 2.1 Symmetrical FPGA Core Surrounded by I/O

As illustrated in Figure 2.2, each PLB contains two 3-input LUTs, a D Flip-Flop (FF) with asynchronous set/reset, and a number of multiplexers that provide a variety of functions including several modes of operation such as sequential mode, arithmetic mode, DSP/multiplier mode, counter mode, tri-state/multiplexer (MUX) mode [7]. The logical value produced by each PLB can be held in the D Flip-Flop (FF) present in the PLB. As shown in Figure 2.2, the X and Y outputs of each PLB connect diagonally and orthogonally to its neighboring cells, respectively [7], and these resources are considered as local routing resources. As illustrated in Figure 2.3, five vertical and five horizontal

busing planes are associated with each PLB as *x8* and *x4* lines respect to repeater boundaries. The *x8* and *x4* lines are considered as global routing resources that span eight and four PLBs, respectively, with repeaters separating the groups of PLBs as shown in Figure 2.3. Four inputs to the PLB or one output from the PLB can access any of five *x4* lines in the busing planes adjacent to the PLB through Configurable Interconnect Points (CIPs).

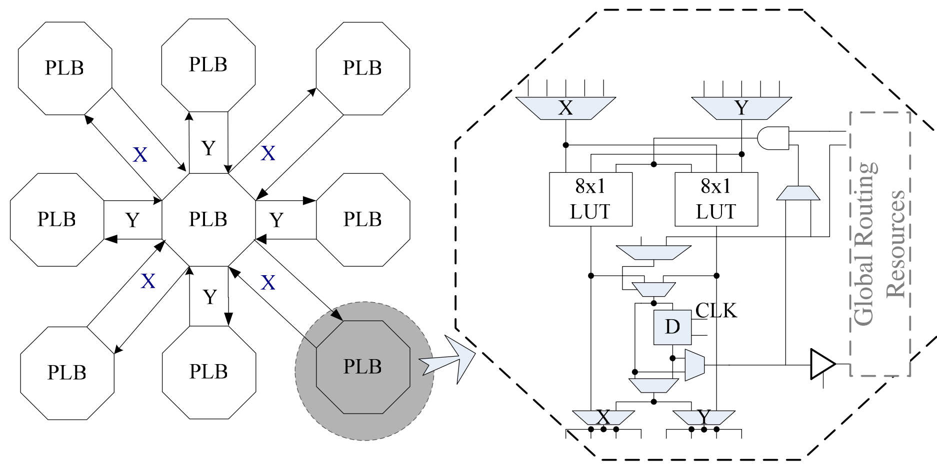


Figure 2.2 Cell-to-Cell Connections & PLB Cell

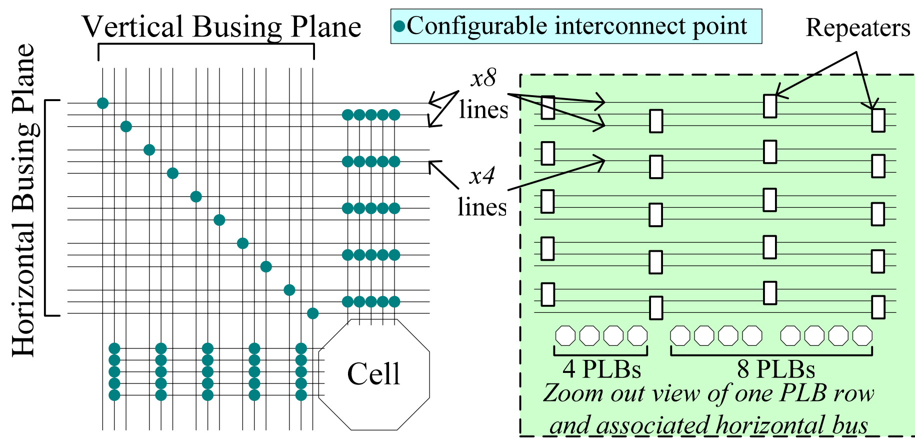


Figure 2.3 Cell-to-Bus Connections

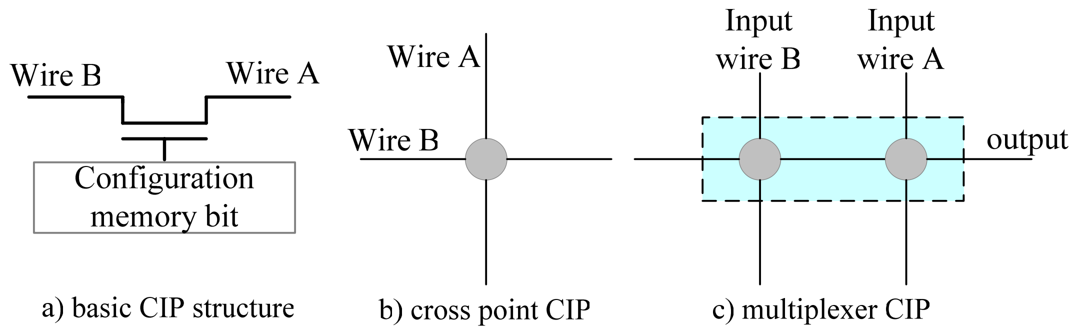


Figure 2.4 Configurable Interconnect Point Structure and Types [12]

The basic structure of a CIP is shown in Figure 2.4a and consists of a pass transistor controlled by a configuration memory bit [12]. When the configuration memory bit is programmed to logic “1”, wire segments A and B are connected [12]. Cross-point CIPs and Multiplexer CIPs constitute most of the routing resources of the embedded FPGAs present in AT94K FPSLIC SoCs [33]. Cross-point CIPs enable the connection between the two wires. As illustrated in Figure 2.4b, the vertical wire A will be connected to wire B when the cross point CIP is turned “on”, meaning that the configuration memory bit controlling the CIP is a logic “1”. The cross-point CIP is used when the signal needs to turn from one direction to a perpendicular direction [12]. A MUX CIP, shown in Figure 2.4c, enables the connection between a single input wire from a group of wires to a single output wire [12].

As shown in Figure 2.1, vertical and horizontal bus repeaters, placed within the global routing resources for every 4x4 array of PLBs, prevent signal degradation in the process of sending signals on distant or heavily loaded nets [33]. Each repeater consists of four MUX CIPs. The repeater can be configured in the modes illustrated in Figure 2.5 and one repeater block can have a combination of the modes if there are no conflicts in the directions of different signal paths. For instance, a repeater can be configured to have

the modes shown in Figure 2.5a, 2.5i, and 2.5c with no conflicts. A conflict of the signals will occur when the repeater is configured to have modes shown in Figure 2.5a, 2.5i, and 2.5k because the modes shown in 2.5i and 2.5k conflict, since two MUX CIPs are driving the same x8 line. All the repeater signals are buffered through the MUX CIPs except the mode shown in Figure 2.5e, which consists of a transmission gate and is used for bi-directional signals. Vertical repeaters are configured in the same way as the horizontal repeaters and the repeater models shown in Figure 2.5 should be rotated by 90 degrees for visualization of the vertical repeater configuration modes.

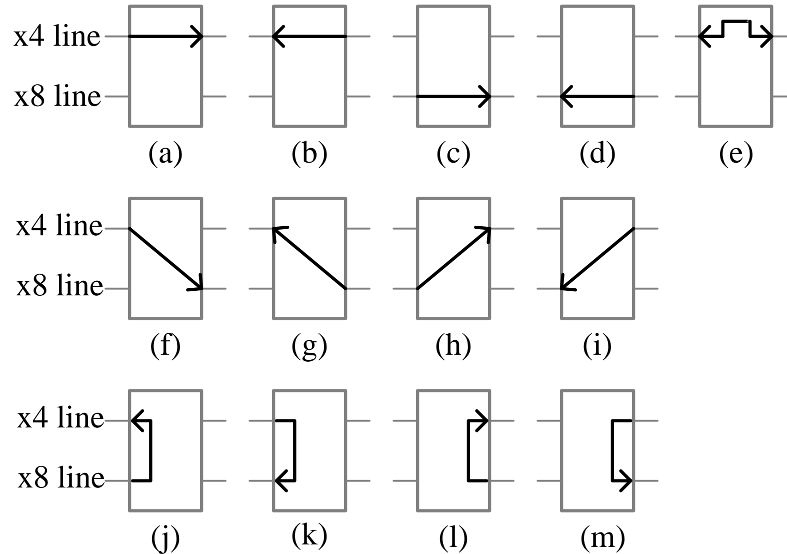


Figure 2.5 Basic Modes of Horizontal Repeater

Banked clock and set/reset lines run to the groups of four PLB cells in a single column within repeater boundaries. As shown in Figure 2.6, eight global clock buses are connected to the column clock MUX which routes one of the eight clocks to all PLBs in the column. Any FPGA internal signal can be routed to one of the global clocks or it can be routed directly to the clock input for any set of four PLBs. Set/reset lines have a similar architecture and the difference is the direction of the signal flow, set/reset goes up

through the PLBs while clock goes down. Both the clock and set/reset signal can be inverted by choosing an inverting path on the MUX which is present before the signal reaches the set of four PLBs.

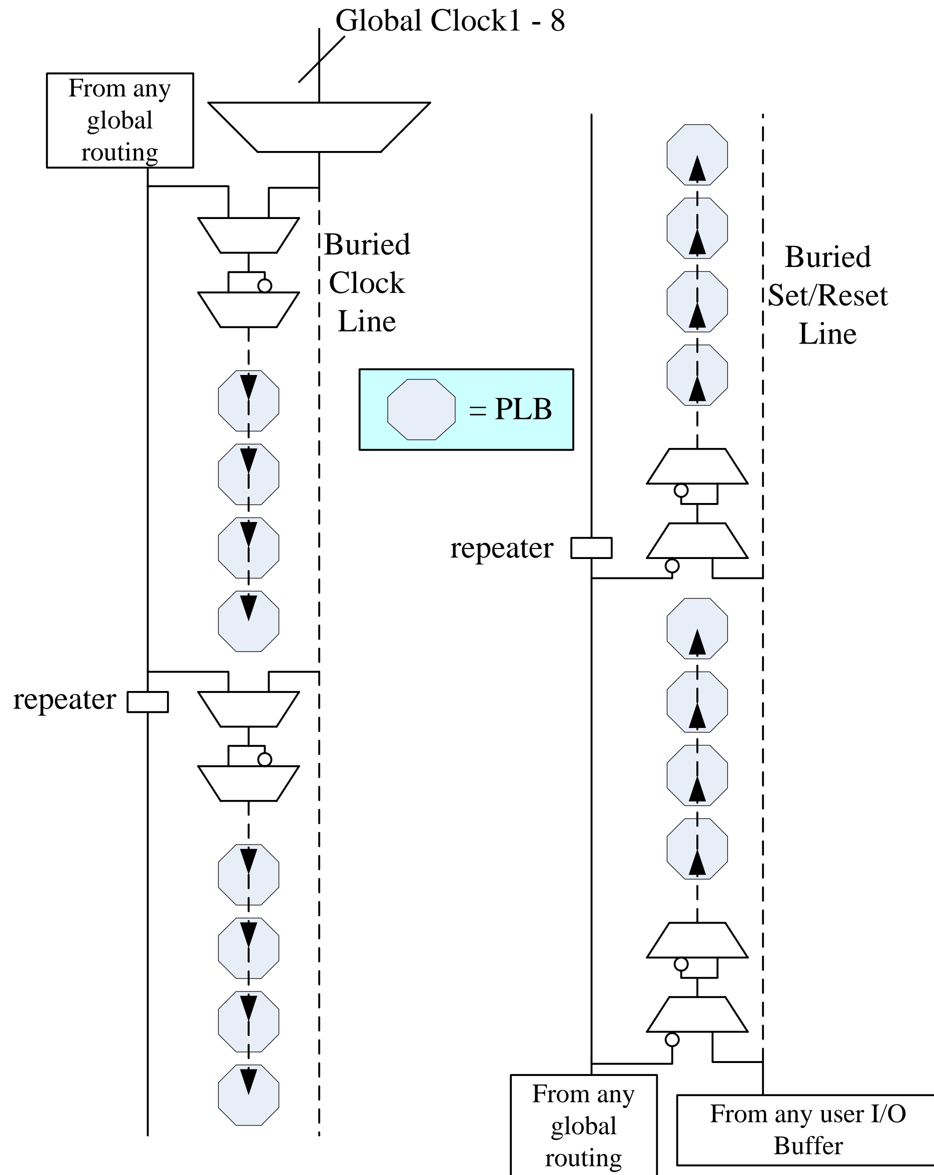


Figure 2.6 Banked Clock & Set/Reset for One Column of PLB Cells

2.1.2 AVR Microcontroller Architecture

The microcontroller from Atmel is called the AVR (Advanced Virtual RISC, and also known as Alf Vergard RISC: named after the founders Alf Bogen and Vergard

Wollan) [29]. The AVR is based on an 8-bit RISC architecture, meaning 1 byte wide working registers are used when instructions are fetched and executed. As shown in Figure 2.7, all 32x8 general-purpose registers are tied to the Arithmetic Logic Unit (ALU) so that two independent registers can be accessed in only one clock cycle, allowing most of the AVR instructions to be executed in a single clock cycle [7].

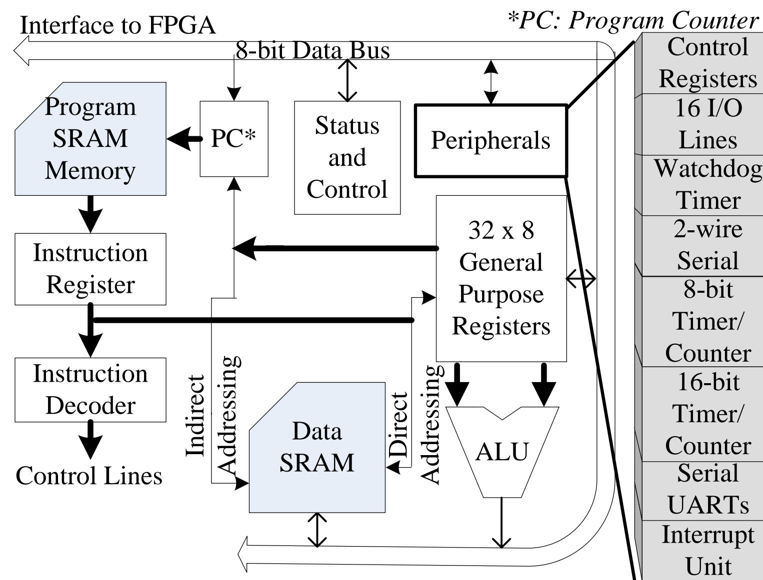


Figure 2.7 AVR Core Architecture

The AVR core has a Harvard architecture, which has the ability to execute an instruction while accessing memory space at the same time [30]. With its architectural advantage, the AVR has up to ten times faster throughput than the CISC (Complex Instruction Set Computer) developed by Intel [7]. The AVR can achieve a throughput of 1 MIPS (Million Instructions Per Second) per MHz [7]. In addition, there are two 8-bit bi-directional general purpose Input/Output (I/O) ports called PORTD and PORTE [7]. There are peripherals such as 8-bit or 16-bit timer/counter with Pulse Width Modulation (PWM), Universal Asynchronous Receiver-Transmitter (UART), 16 I/Os, and 2-wire serial port located within the AVR core. Peripherals attached to the AVR core can be

programmed in assembly language or C language. Interrupt sources internal and external to the SoC allow the AVR to be operated in more interactive ways. The return address of the program counter (PC) is stored on the stack when interrupts and subroutine calls occur and the stack is allocated in the Data SRAM [7].

2.1.3 RAM Architecture

There are two types of SRAMs present in AT94K series devices. One type of SRAM is evenly distributed through the FPGA core and the other type of SRAM is placed outside of the FPGA core, shared by other cores such as the AVR and its peripherals.

The SRAMs distributed through the FPGA core are 32x4-bit memory blocks with one RAM placed in every 4x4 array of PLBs as illustrated in Figure 2.1. This dedicated SRAM, denoted as *freeRAM* by Atmel, can be accessed through the global routing resources by PLBs [7]. Each *freeRAM* can operate in single port or dual port mode [7].

The other type of SRAM resides outside the FPGA core. Both the FPGA core and the AVR core share the embedded Data SRAM and, thus, it is designed with a bigger size than the *freeRAM* [7]. The Data SRAM is used by the AVR and FPGA for general-purpose data storage [7]. Depending on the design, the SRAM can be configured in various modes and can also be flexibly partitioned. In the AT94K40 series SoCs, a maximum size of 36 Kbyte SRAMs are supported, which can be partitioned into different sizes of Data SRAM and program memory blocks. Both the AVR and FPGA are connected to the Data SRAM, which can be partitioned in size from 4 Kbytes to 16 Kbytes. It stores data from the FPGA and AVR, and provides register space for the AVR. The program memory is used to store AVR programs and it can be partitioned in size

from 20 Kbytes to 32 Kbytes. The program memory provides the space from which the AVR fetches instructions and runs programs, and it cannot be accessed from the embedded FPGA core [7].

2.1.4 FPGA-RAM-AVR Interface

As illustrated in Figure 2.8, the Data SRAM resides between the FPGA core and AVR core, enabling smooth data sharing and/or exchange between the AVR and FPGA cores. To access the Data SRAM, a 16-bit address is required from the FPGA or AVR core. Data to be accessed or stored pass through the bi-directional 8-bit data bus across the FPGA core, the Data SRAM, and the AVR core. The Write/Read Enable signals along with clock signal provide control over access of the Data SRAM.

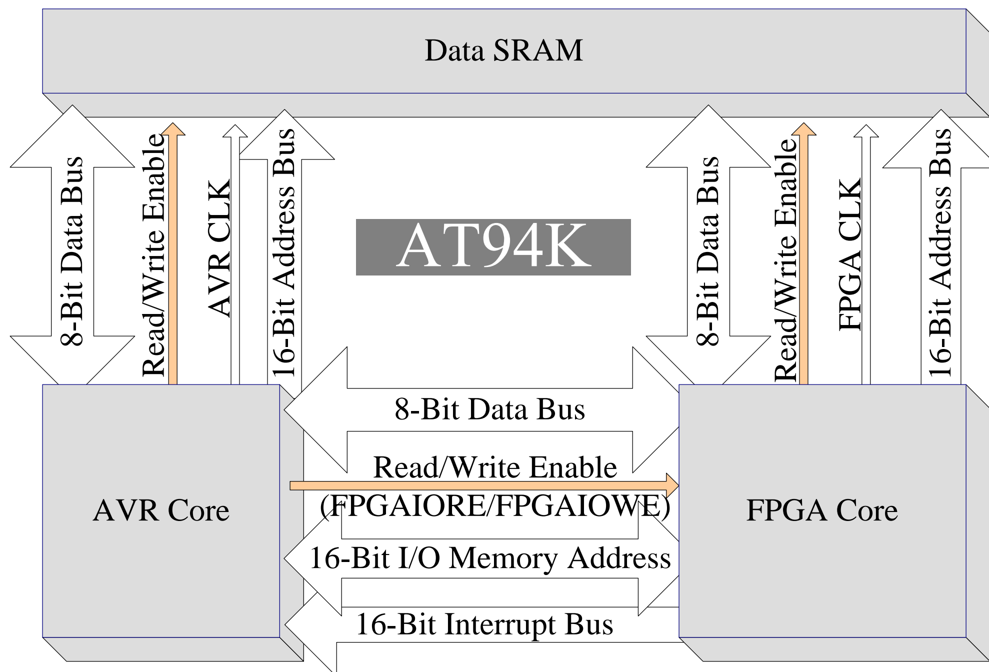


Figure 2.8 FPGA-RAM-AVR Interface

The FPGA core can be directly accessed by the AVR core, as shown in Figure 2.8. There is an 8-bit data bus between the FPGA core and the AVR that allows them to

communicate interactively under the control of the AVR. FPGAIOWE (FPGAIORE) is a strobe line that is activated when the AVR writes to (reads from) the 8-bit bi-directional data bus. There are 16 decoded address lines supplied from the AVR to the FPGA. Also, a maximum of 16 interrupts are available from the FPGA to the AVR with various priority levels to make the operations of the AVR efficient [7].

2.2 Special Features in the AT94K

In this section, some of the unique features of the AT94K series device are described. These features have a direct impact on the development and execution of BIST in this thesis.

2.2.1 Cache Logic Mode

In the AVR Cache Logic mode, the configuration memory of the FPGA core can be dynamically reconfigured by the AVR during system operation, without re-downloading the configuration data externally. This can be done without affecting the contents of the flip-flops, known as dynamic partial reconfiguration. As illustrated in Figure 2.9, due to its PLB addressable structure, FPGAX, FPGAY, and FPGAZ hold the address of the target configuration memory byte of the FPGA to be reconfigured, where FPGAX corresponds to the horizontal PLB location, FPGAY corresponds to the vertical PLB location, and FPGAZ corresponds to specific configurable logic and/or routing resources within the specified PLB. A 32-bit configuration word cache waits until the FPGAD register receives new data to be written into the FPGA configuration memory [7]. Any writes into FPGAD result in a configuration clock cycle to the FPGA configuration memory [7]. Thus, instead of downloading a full configuration each test phase, the AVR

can partially reconfigure the locations where a change is needed. The basic routine to program the AVR to reconfigure the FPGA core is shown in Figure 2.10.

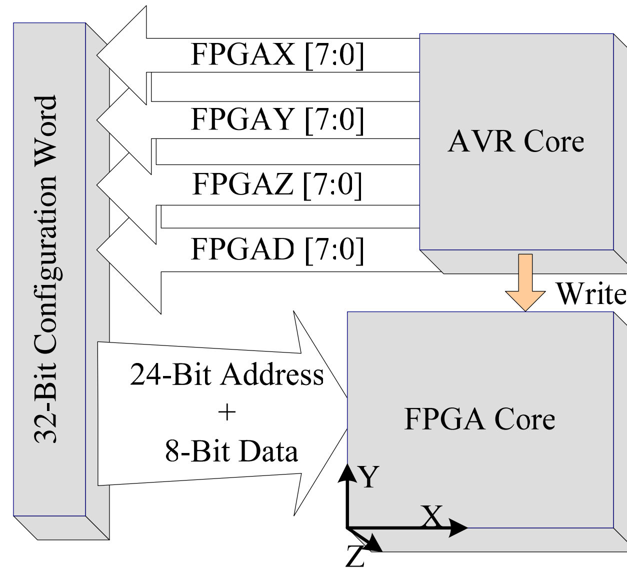


Figure 2.9 AVR-FPGA Dynamic Cache Logic

```
ldi     rTemp, (Column# - 1)          ; PLB Horizontal Coordinate
out     FPGAX, rTemp

ldi     rTemp, (Row# - 1)             ; PLB Vertical Coordinate
out     FPGAY, rTemp

ldi     rTemp, 0bttttzzzz             ; TagZ coordinate (Page#[7:4]+Byte#[3:0])
out     FPGAZ, rTemp

ldi     rTemp, 0bxxxxxxxx             ; New PLB "Byte" Contents
out     FPGAD, rTemp
```

Figure 2.10 PLB Reconfiguration Method

Since the AVR can specify the X (horizontal) and Y (vertical) PLB coordinates, it can be programmed in such way that the AVR can algorithmically generate configurations and reconfigure the FPGA core. The fine-grained architecture of the FPGA core is the major advantage when using X (FPGAX) and Y (FPGAY) PLB

coordinates because of its regular and repeatable structure [24][30]. In other words, a coarse-grained architecture would make it difficult to algorithmically reconfigure the FPGA because of its irregular structure. In addition, the FPGA's symmetrical architecture enables simple and predictable reconfiguration [7]. One major drawback with AT94K series SoCs is that the FPGA configuration memory contents cannot be read using the AVR [24].

2.2.2 Use of Macro Generation Language (MGL)

Atmel provides a specially designed programming language called Macro Generation Language (MGL) [31]. The language is utilized through Figaro; one of Atmel's Integrated Development System (IDS) tools [31]. It is used to instantiate designs in the FPGA and to produce a downloadable bitstream [33]. The main advantage of using macro designs made by the MGL is its capability to implement parameterized designs that can be constructed in any size FPGA array. MGL defines the layout and routing of the FPGA core [31]. Furthermore, unlike the Xilinx Design Language (XDL), MGL supports hierarchical designs by calling pre-defined or user-defined macros into newer macros which could reduce program size. Designs described in MGL can be edited, debugged, and executed in Figaro IDS software [31]. When configuring a PLB, MGL based on either predefined macros (gates, multiplexers, flip-flops, etc) or dynamic macros can be used [32]. Dynamic macros give flexibility in defining the PLB function. However, the user can only control the PLBs by the dynamic macros and no further control is provided by MGL [32]. In order to achieve maximum fault coverage from BIST for FPGAs, complete control over the configuration of the logic and routing resources is required [33].

2.3 Overview of BIST for Embedded FPGA Core in the Atmel FPSLIC

The BIST architecture for testing the PLB resources in an FPGA, shown in Figure 2.11, configures a column of PLBs to function as two or more identical TPGs that drive test patterns to alternating columns of identically configured BUTs. The outputs of BUTs are monitored by comparison-based ORAs located in adjacent columns between the BUTs [17].

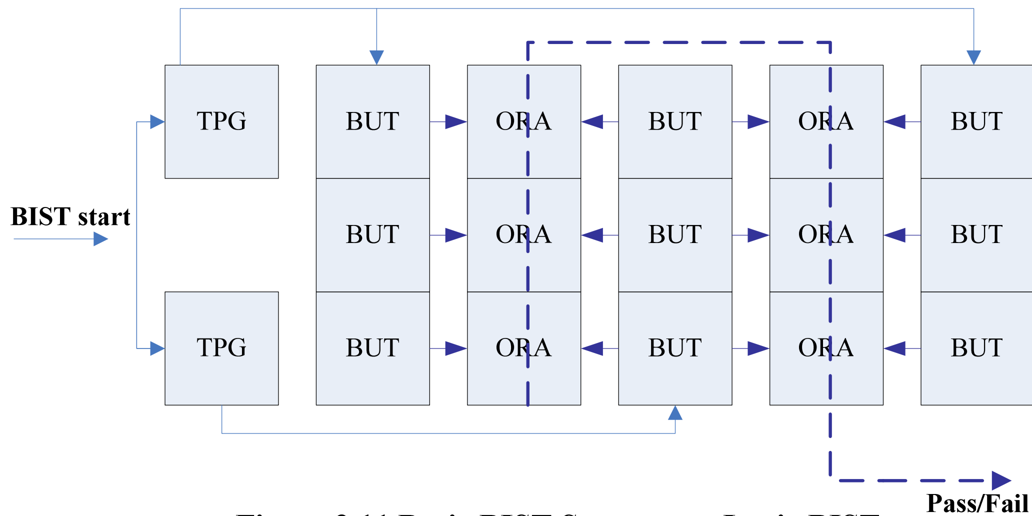


Figure 2.11 Basic BIST Structure – Logic BIST

For applying test patterns to the BUTs, there are a number of TPG types that can be used [17]. The most basic TPG type is the N-bit binary counter since it generates exhaustive 2^N binary test patterns. Another well known type is the Linear Shift Feedback Register (LFSR) which generates pseudo-random test patterns. If the LFSR has a primitive polynomial function then it will generate all possible 2^N-1 patterns excluding the all-zero pattern [17]. The all-zero pattern can be achieved in an LFSR by adding circuitry, however, it would cost additional area in the FPGA to be programmed. Thus, for testing the PLB blocks, if the number of inputs to the BUTs is small, using the binary counter as a TPG is the most economical and efficient method [33].

The basic design of the ORA is shown in Figure 2.12 where two identical BUT outputs are compared by the exclusive OR-gate (XOR-gate) [17]. When a mismatch occurs between the two BUTs, the input of the Flip-Flop (FF) will see a logic “1” from the output of the XOR-gate. The logic value “1” is latched in the flip-flop via the OR gate and held throughout the BIST sequence. At the end of the BIST sequence, the content of the flip-flop indicates whether the ORA saw a mismatch of the two BUT outputs or not. Typically there is more than one ORA to be read at the end of test. ORA results can be read either individually by reading the configuration memory of the ORA block directly or they can be scanned out serially using a shift register as illustrated in Figure 2.11 by the dotted line [17].

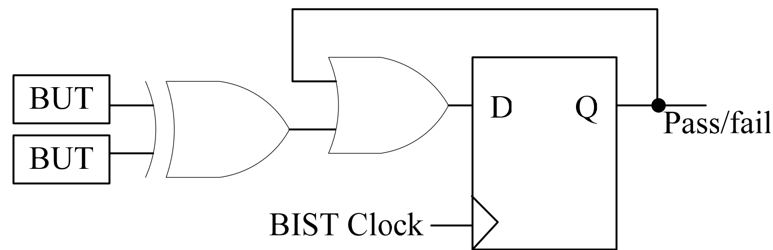


Figure 2.12 Basic Comparison Based ORA structure

The BUTs are reconfigured in various modes of operation until they are completely tested [10]. The number of modes in which the BUTs are to be configured is determined by the complexity of the PLB. The more programmable logic resources the PLB has, the more BUT configurations are typically needed to test all the resources in the PLB. The BIST architecture is then flipped about the vertical axis (Figure 2.13) to test the PLBs that were previously TPGs and ORAs for the complete test of all PLBs as BUTs [28].

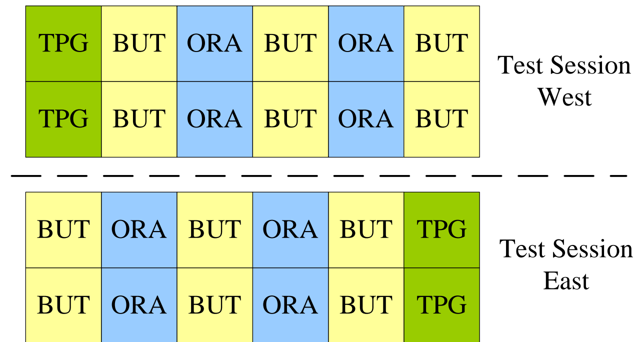


Figure 2.13 FPGA BIST Structure for Complete Test

The basic sequence for FPGA BIST consists of the following steps:

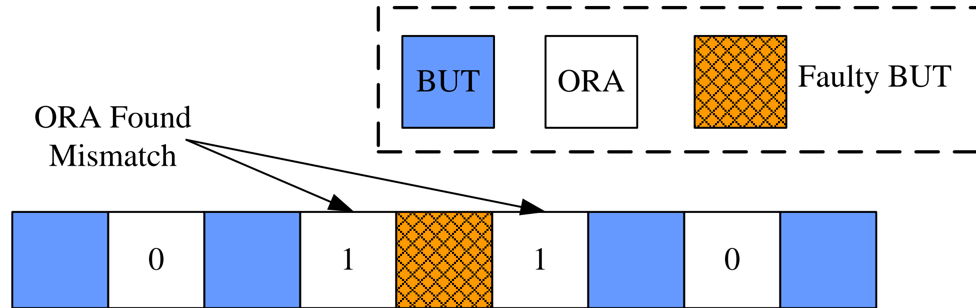
1) Configure the FPGA to a BIST structure

2) Execute the BIST sequence

3) Retrieve ORA results

4) Analyze ORA results to find faulty PLBs.

Step 1 requires the configuration of resources in the FPGA to perform the BIST. Typically this is done by external configuration download to the FPGA for every BIST configuration. Next, the test controller initiates the BIST (Step 2) by applying BIST clocks to the FPGA so that the test patterns are applied to the BUT inputs while the BUT outputs are monitored. At the end of each BIST sequence, ORA results are retrieved so that they can be analyzed (Step 3 and 4). When analyzing the results, as shown in Figure 2.14, the faulty PLB can be found based on the locations of the ORAs that observed mismatches [21]. BIST steps 1 through 4 are repeated until all the modes of operation for the BUT are tested.



2.14 Diagnosis PLBs from Analyzing Comparison Based ORA Results

In the case of the Atmel AT94K series SoCs, the BIST architecture shown in Figure 2.15 is used wherein each ORA monitors one diagonal X-output and one direct Y-output from the neighboring BUTs. The architecture shown in Figure 2.11 is not applicable, since a PLB cannot be configured to monitor more than one diagonal X-input and one direct Y-output selected at the same time [33]. Therefore, two different routing schemes are needed in order to observe both the diagonal X and direct Y connections for complete testing of the PLB logic resources. For all PLBs except the ones located in corners of the FPGA array, a total of four configurations of the BUTs are needed to obtain a fault coverage of 99.7% with only one fault left which is potentially detected [17][33]. The potentially detected fault is detected during routing BIST to result in 100% fault coverage with a complete set of BIST configurations [28].

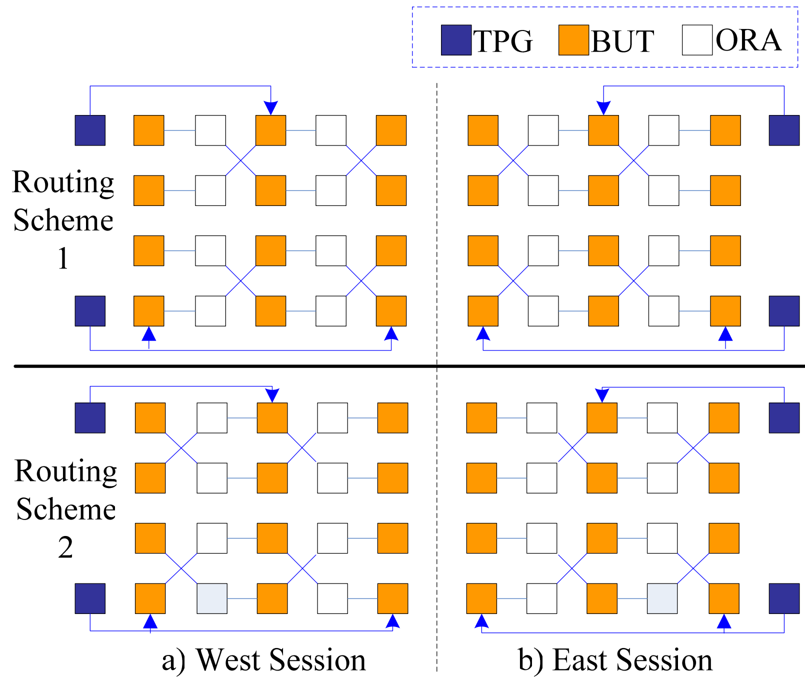


Figure 2.15 Logic BIST Architecture of AT94K Series SoCs

2.4 Thesis Restatement

BIST approaches have been developed for FPGAs by programming some of the PLBs as TPGs and ORAs to test the remaining programmable logic and interconnect resources [10]. However, these techniques typically require downloading a large number of BIST configurations into the FPGA one at a time, executing each BIST sequence, and retrieving the BIST results at the end of each BIST sequence. While this problem can be reduced by minimizing the total number of BIST configurations and/or by taking advantage of the partial reconfiguration capabilities provided in recent FPGAs, the total test time and memory storage requirements are still dominated by the download process. For SoC testing, the embedded microprocessor cores in SoCs can be programmed to test other accessible cores such as FPGA cores. Dynamic, partial, and full reconfiguration of

FPGA cores by embedded processor between each test phase can reduce the total test time. After completion of BIST, the embedded processor can retrieve the test results, perform diagnosis, and report the faults and their locations to a higher computing resource for fault recovery or fault-tolerant applications.

The dynamic partial reconfiguration capability of the embedded processor core was previously used to a limited extent in [28]. However, this approach needed to download each and every BIST configuration into the FPGA core. The primary focus of this thesis is to investigate potential improvements in the total test time and memory requirements by avoiding any and all downloads into the FPGA. By programming the embedded processor core to execute algorithmic reconfiguration routines, the amount of memory required for storing BIST configurations is reduced since no configuration data is downloaded into the FPGA. The fine-grain architecture, in conjunction with the PLB addressable configuration memory of the AT94K series SoCs, helps to configure the BIST structures without the need for excessive configuration clock cycles. If small enough, the BIST program can remain resident in the program memory for on-demand reconfiguration and execution of BIST, requiring no download at all. If fast enough, the BIST program can be more frequently used during idle intervals in system operation for high reliability, high availability applications.

CHAPTER THREE
AVR ASSISTED FPGA LOGIC BIST

The goal in this thesis is to reduce the total test time and configuration memory storage requirements associated with BIST of the PLBs in the FPGA core. In order to do so, previous work [33] which required FPGA configuration download for each test phase was used, which served as a fundamental model so that any improvements could be measured. Since the work in [33] has realized some of the problems associated with BIST for FPGAs in AT94K series SoCs, it has proposed ways to overcome these problems. In this chapter, improvements over the previous work [33] are discussed with experimental data taken from the execution of BIST on actual Atmel AT94K series SoCs.

The flow of the chapter is according to the improvements made throughout the thesis work, which divides into three phases. First, the shift register reconfiguration development for retrieving ORA results at the end of each BIST sequence [33] is discussed. This was an essential development for [33] as well as fundamental work for the next two phases. As the second phase of the development, the AVR processor is used to assist the BIST developed in [33] by not only reconfiguring ORAs into a shift register but also reconfiguring the BUTs for each test phase based on BIST structures and requirements in [33]. This approach replaces most time-consuming FPGA configuration downloads with simple AVR programs and is denoted as 'AVR-assisted BIST' in this thesis. The third phase of the development yields a modified version of the AVR-assisted

BIST which has a new way of testing X and Y direct PLB connections and reads test results at the end of multiple test phases.

3.1 Implementation of ORA and Shift Register

One of the issues in the previous work [33] was the fact that the ORA results could not be read back directly from the FPGA configuration memory to the test controller (a PC in our case), which made the implementation of a shift register, or scan chain, necessary. For implementing the scan chain, another problem arose due to the PLB size and a small number of input lines. In order to configure a comparison-based ORA with a shift register feature shown in Figure 3.1, a total of five inputs are needed for one PLB [33]. Since the PLBs present in Atmel AT94K series SoCs have only four inputs there is no way to implement the ORA with a shift register feature as shown in Figure 3.1.

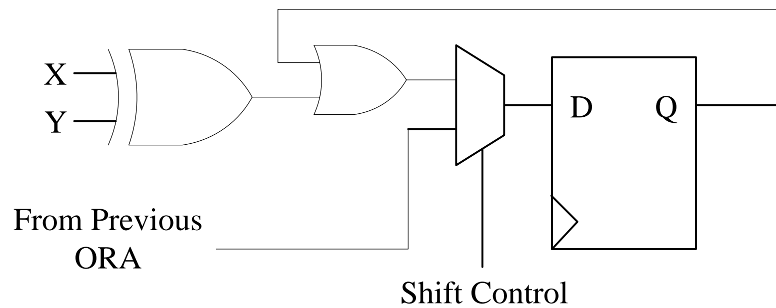


Figure 3.1 ORA Structure for Logic BIST

Initially when the ORA structure was investigated, there were two possible models that could be implemented [33]. As shown in Figure 3.2, the ORA and the scan chain can be implemented together by using two PLBs per ORA with four BUTs being compared at once. However, using this model results in loss of diagnosis resolution compared to using the model shown in Figure 3.3 [33]. In Figure 3.3a, all ORAs are reconfigured by the

AVR core to form the scan chain shown in Figure 3.3b. As a result, the initial BIST architecture has the simple ORA shown in Figure 3.4a. After the BIST sequence is executed and ORA results are ready to be read, the ORAs are reconfigured as a scan chain (shift register) and the results are scanned (shifted) out for analysis, as is illustrated in Figures 3.3b, 3.4b and 3.5. The main difference between the two-PLB ORA and one-PLB ORA is the need for partial dynamic reconfiguration of the ORA cells.

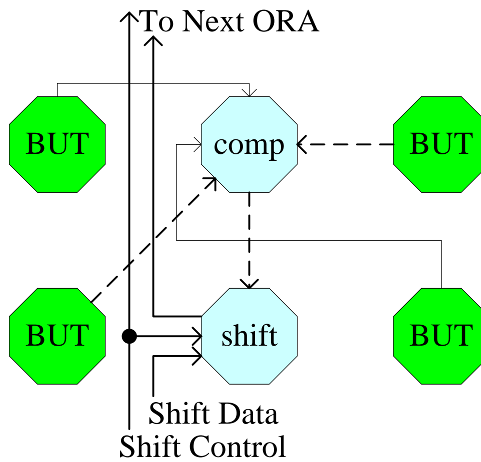


Figure 3.2 Two-PLB ORA

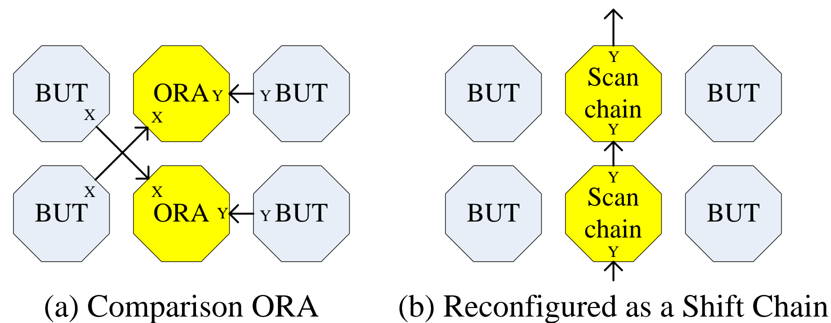


Figure 3.3 High Level Structure of ORA and After the Reconfiguration

In order to reconfigure the comparison based ORAs (Figure 3.4a) as a shift register (Figure 3.4b), the AVR first writes to the PLB configuration memory for each ORA to change the functionality of these PLBs. As the reconfiguration is being performed by the

AVR core, since the ORA scan chain is a directional shift register, diagonal X connection and orthogonal Y connection which was being compared in Figure 3.4a would be reconfigured so that each shift register cell is routed to the neighboring shift register by using the direct Y connection as shown in Figure 3.3b.

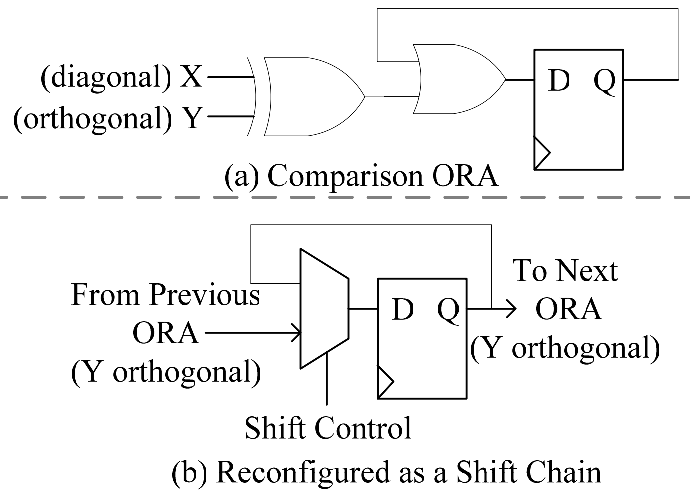


Figure 3.4 Comparison ORA and the ORA After Reconfiguration

Figure 3.5 shows a simplified illustration of the shift register layout for one of the logic BIST configurations. Note that the ORA results are scanned out via an external pin. The shift register reconfiguration program, which will be introduced in next section, controls the AVR writes to the configuration memory of the FPGA, reconfigures all the ORAs as a scan chain to have shift-up and shift-down columns as well as to have center route-through PLBs where the BUTs are reconfigured as part of the scan chain. The route-through PLBs don't require any flip-flop to be involved but a simple routing connection between the shift-up and shift-down columns as illustrated in Figure 3.5

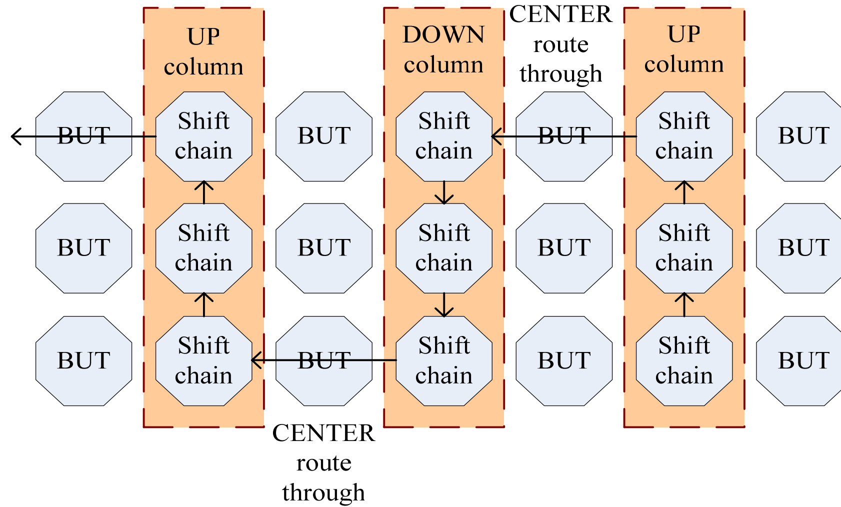


Figure 3.5 Shift Register Layout

3.2 Implementing Shift Register Reconfiguration for Logic BIST

In order to implement the dynamic reconfiguration routine of converting ORAs into a scan chain, assembly or C programming language can be used to develop the AVR program. There are advantages and disadvantages of using assembly over C. If the program is written in assembly, which is "machine-level”, it can provide an educative approach to what goes on inside the processor. Also the assembly program is the best way to optimize the code because it enables user to control behavior of the processor in detail. However, C programming language was chosen because of its convenient features and the support of well-performed compilers that optimize the compiled program fairly effectively. Although different compilers have slightly different notations and rules to do the same thing, most compilers do a better job of code size and execution speed optimization compared to most of the user’s assembly code if the code is long. For our development needs, due to the usage of many parameterized files and modification throughout the development sequence, programming in C language was the best choice

to keep the development process efficient.

A compiler called “Codevision AVR” was used [7]. It enables use of C language to generate programs for execution on the AVR microcontroller. The compiler converts the user’s C program to assembly language and generates an Intel HEX file. The Intel HEX file is then combined with the FPGA bitstream generated from the Figaro [33]. The combined file (a combined bitstream) is downloaded to the SoC. The resultant bitstream programs program/data memory of the AVR, the FPGA configuration memory, and peripherals around the AVR processor. The compiler can optimize compiled AVR program size by grouping common tasks into subroutines. When optimizing for speed, the compiler tries to generate smallest number of subroutines possible so that fewer branch instructions occur during execution.

```
interrupt [EXT_INT0] void isr_reconf_ORA(void) {
          //UP                                                          ··· (a)
          for (FPGAX = 2; FPGAX < size; FPGAX+=4) {
                    for (FPGAY = 0; FPGAY < size; FPGAY++) {
                    //connecting shift register for UP direction        ··· (b)
                              FPGAZ = [PLB Tag]+[Byte#];
                              FPGAD = [Byte]
                    }
          }
          //DOWN                                                        ··· (c)
          for (FPGAX = 4; FPGAX < size; FPGAX+=4) {
                    for (FPGAY = 0; FPGAY < size; FPGAY++) {
                    //connecting shift register for DOWN direction      ··· (d)
                    }
          }
          //LEFT-TOP                                                    ··· (e)
          FPGAY = size-1;  //Top = Row 47
          for (FPGAX = 4; FPGAX < size; FPGAX+=4) {
          //connecting shift register for every corners                 ··· (f)
          }
          //LEFT-BOTTOM                                                 ··· (g)
          FPGAY = 0;  //Bottom = Row 0
          for (FPGAX = 2; FPGAX < size; FPGAX+=4) {
          //connecting shift register for every corners(2)              ··· (h)
          }
          //CENTER-TOP                                                  ··· (i)
          FPGAY = size-1;   //Top = Row 47
          for (FPGAX = 5; FPGAX < size; FPGAX+=4) {
          //Route throughs between ORAs                                 ··· (j)
          }
          //CENTER-BOTTOM                                               ··· (k)
          FPGAY = 0;   //Top = Row 0
          for (FPGAX = 3; FPGAX < size; FPGAX+=4) {
          //Route throughs between ORAs                                 ··· (l)
          }
          //START                                                       ··· (m)
          FPGAX = size-2;
          FPGAY = 0;
          //Putting logic "1" at the end of the scan chain              ··· (n)
}
```

Figure 3.6 AVR Code of ORA Reconfiguration to Shift Register

A C code example for the ORA-to-Shift Register reconfiguration is shown in Figure 3.6. The example shows the AVR routine for reconfiguring ORAs into shift registers at the end of a logic BIST sequence. It consists of various for-loops which reconfigure the ORAs into shift registers. In order to form the shift register illustrated in

Figure 3.5, first, half of the ORA columns are configured to shift data up and the other half to shift down as shown in Figure 3.6 lines (a) through (d). To connect every column of shift register pieces into one scan chain, the routines shown in Figure 3.6 lines (e) through (l) reconfigure either top or bottom PLBs of the ORA columns and some of BUT PLBs that are adjacent to one end of each shift register column as route-though PLBs according to Figure 3.5. Finally at the end of the scan chain, the look-up-table is reconfigured to generate a constant logic value "1" as shown in Figure 3.6 lines (m) and (n) to verify the integrity of the scan chain during BIST results retrieval [10]. This also helps to check the consistency of the cache logic mode itself to ensure that the dynamic partial reconfiguration was correct. After specifying 'FPGAX' and 'FPGAY' location, 'FPGAZ' is written followed by 'FPGAD' from the AVR so that FPGA configuration memory can be written.

The routine is executed through external interrupts. At the end of each BIST sequence, the higher test controller unit (such as PC) activates the interrupt to reconfigure the shift register, after which the ORA results can be retrieved by the test controller. All BIST configurations reported in [33] used similar shift register reconfiguration programs developed as part of this thesis and were a necessary part of the development in [33]. Dynamic partial reconfiguration of ORAs into a shift register is also used in routing BIST due to the fact that the ORA contents cannot be read directly from FPGA configuration memory. Most of the ORA layouts are regular, thus algorithmic AVR reconfiguration routines which reconfigure the ORAs to a scan chain were implemented in a similar way to the logic BIST shown in Figure 3.6 [33].

3.3 Dynamic AVR Reconfiguration of BUTs and ORAs for BIST

Improvements of the test time and configuration memory requirement can be made by developing a new BIST sequence using the AVR microcontroller to dynamically reconfigure FPGA configuration memory. For example, the BIST sequence in the previous work on the same device is as follows [33]:

1) Reconfigure FPGA for the BIST (download configuration file).
2) Run BIST (BIST clock is applied).
3) Reconfigure ORAs to form a scan chain (dynamic partial reconfiguration via the AVR).
4) Retrieve ORA results (to the external controller such as PC).
5) Reconfigure FPGA for the next BUT mode of operation (configuration file download required).
6) Repeat step 2) – 5) until all modes of BUT operation are tested.
7) Diagnose the retrieved ORA results to locate the faulty blocks (if there are any ORA failures).

Improvement can be made to Step 5 above when the FPGA has to be reconfigured externally via downloading an external configuration file for the next BIST configuration. Instead of the external configuration download, this BUT reconfiguration can be done by the AVR processor through partial reconfiguration of BUTs to the next mode of operation. Since the BUTs in logic BIST architecture have either column or row oriented structure, they can be easily reconfigured by an algorithmic AVR routine to save test time.

Additional improvements can be made to step 4. After the BIST clock cycles are applied for each BIST sequence, the ORA results are retrieved in this step. This is

because the shift registers are overwritten to ORAs when the next BIST configuration download occurs, resulting in a chip reset and the contents of ORAs are lost. However, from another aspect, the new BIST configuration downloads are required in order to reset the contents of ORAs as well as to reinitialize comparison-based ORA functionality for the next BIST configuration. Improvements can be made by utilizing the AVR's dynamic partial reconfiguration capability since the AVR can reconfigure the shift chain back to ORAs and clear the flip-flop contents of ORAs so that the next test phase can be run. Therefore, only one initial BIST configuration download to the FPGA is needed for each test session and subsequent BIST configuration downloads can be replaced with a small AVR program.

As a result, only one BIST configuration needs to be downloaded along with a program to be executed by the processor core for the reconfiguration of subsequent BIST configurations. In the previous work [33], the FPGA is tested for four directions (Figure 3.7) and one direction consists of four modes of BUT configuration. Therefore a total of 16 BIST configurations must be downloaded. These sixteen configuration downloads can be replaced by four downloads with the AVR-assisted BIST. This provides an improvement to total test time when compared to downloading individual BIST configurations because the download time dominates the total test time since the configuration clock usually runs at a lower frequency than the processor clock.

3.7 Four Layouts for Logic BIST [33]

To reconfigure for the next test phase from the AVR, an external interrupt routine is used which has a global variable 'phase'. The variable 'phase' is initialized to 1 during the initial download to the FPGA which means the first BUT mode is configured in the FPGA. When the interrupt occurs, the 'phase' variable is incremented to configure BUTs differently and appropriately as the test sequence proceeds, as shown in Figure 3.8 line (c). In order to achieve maximum speedup, the use of arrays is avoided and a 'switch-case-break' scheme is used (Figure 3.8a) which saves execution time but requires more program memory size. After the reconfiguration of the BUTs, the existing shift register is reconfigured back to ORAs by a routine similar to the BUT reconfiguration routine shown in Figure 3.8.

```
interrupt [EXT_INT1] void isr_reconf_BUT(void)
{
        //BUT reconfig                                          ··· (a)
        switch (phase) {
        case 1:
        for (FPGAX = 1; FPGAX < size; FPGAX+=2) {
                for (FPGAY = 0; FPGAY < size; FPGAY++) {
                //Reconfigure BUTs for phase 2
                }
        }
        break;
        case 2:
        for (FPGAX = 1; FPGAX < size; FPGAX+=2) {
                for (FPGAY = 0; FPGAY < size; FPGAY++) {
                //Reconfigure BUTs for phase 3
                //Set up for FF set/reset test
                }
        }
        break;
        case 3:
        for (FPGAX = 1; FPGAX < size; FPGAX+=2) {
                for (FPGAY = 0; FPGAY < size; FPGAY++) {
                //Reconfigure BUTs for phase 4
                }
        }
        break;
        }

        //Routines for reconfiguring shift register back to ORA    ··· (b)

        phase++;                                                ··· (c)
}
```

Figure 3.8 AVR Code of BUT Reconfiguration and ORA Initialization

Detailed analysis of the AVR reconfiguration is shown in Table 3.1 in terms of the number of processor clock cycles required to perform the various functions associated with reconfiguration and execution of logic BIST. The number of non-commented lines of C source code and the number of bytes of program memory storage required for the compile program are also given. In logic BIST for AT94K40 (a 48x48 array) there are 1,152 BUTs and 1,104 ORAs in each BIST configuration [28]. Therefore, BUT reconfiguration requires about 61 cycles per BUT while reconfiguration of the ORAs

into a shift register requires about 23 cycles per ORA and reconfiguration back to ORAs after retrieval of the BIST results requires about 34 cycles per ORA.

Table 3.1 Logic BIST Reconfiguration

Reconfiguration function	Average processor execution cycles	Number of lines of code	Program memory bytes
ORA to shift register	25,570	127	764
Shift register to ORA	37,220	102	328
Reconfigure BUT	70,023	154	756

As illustrated in Table 3.2, in the initial work [33], each of the four BIST configurations associated with each of the four test sessions contains approximately 65 Kbytes of configuration data including the program for reconfiguration of the ORAs into a shift register at the end of the BIST sequence for retrieving ORA results.

Table 3.2 Total Memory Reduction

	Download [33]	AVR-assisted	Memory Reduction
Total Configurations	65 Kbytes × 16 files	67.5 Kbytes x 4 files	3.9

Therefore, a total of approximately 1.04 Mbytes of memory is needed to store all sixteen logic BIST configurations. A single configuration for a given logic BIST session with a program to reconfigure the subsequent three BIST configurations requires only 67.5 Kbytes of configuration and program data (total of approximately 270 Kbytes), giving a factor of 3.9 reduction in memory storage for four test sessions. In these cases, the AVR program performs the following steps during the BIST sequence to obtain improvements over the previous work [33]:

1) Execute the BIST sequence for the current BIST configuration.

2) Reconfigure the ORAs into a shift register at the end of the BIST sequence.

3) Retrieve the BIST results.

4) Reconfigure the shift register back to ORAs for the next BIST configuration.

5) Reconfigure BUTs for the next BIST configuration.

6) Repeat steps 1 through 5 until all of the BIST configurations have been executed.

The test time is determined by the total time required to download the BIST configuration and the time for the processor to execute the steps listed above. At the maximum download (1MHz) and processor clock (25MHz) frequencies, it takes total of 523 milliseconds for a single logic BIST configuration, 2.1 seconds for the test session of four logic BIST configurations, and a total of 8.4 seconds for the complete set of 16 logic BIST configurations as shown in Table 3.3. Using the processor core for reconfiguration (AVR-assisted) of the four BUT configurations within a given test session, it takes a total of 559 milliseconds per test session, giving a speed-up of 3.75.

Table 3.3 Total Test Time and Speed Up

	Download [33]	**AVR-assisted**	**Speed up Factor**
Download (1MHz)	523msec x 16	540msec x 4	3.87
Run time (25MHz)	1msec x 16	19msec x 4	0.21
Total BIST Time	8.4 sec	2.24 sec	3.75

3.4 A Better Logic BIST Sequence

Taking the AVR-assisted logic BIST idea one step further, additional improvements can be made to the BIST sequence. In the previous work [33], one BIST session consists of four BIST configurations to be downloaded and two different routing schemes, shown in Figure 3.9, which alternate as the phase increases, requiring retrieval of ORA results after every BIST configuration [33].

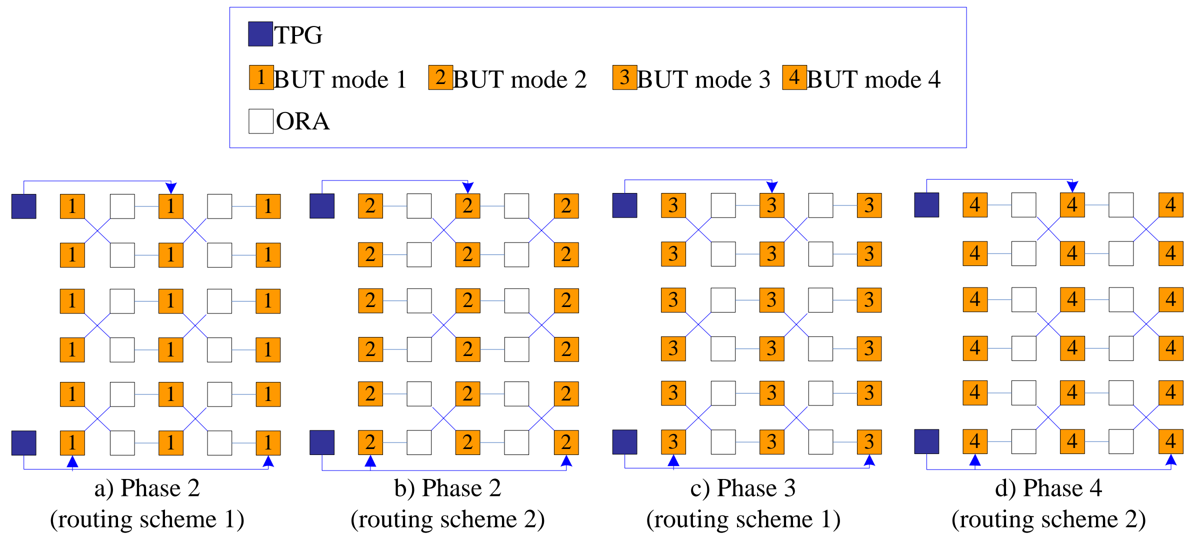


Figure 3.9 Four BIST Phases in One Session for AT94K SoCs

The *MGL-based AVR-assisted* logic BIST described in the previous section had to scan out the ORA results at the end of each BIST sequence. It is due to the fact that the routing scheme alternated after every test phase, resulting in the inability to locate faulty PLBs based on failing BIST results. By reordering the BIST configurations and by grouping the same routing schemes together, the ORA result can be retrieved after multiple test phases. This results in saving total test time by saving ORA reconfiguration and retrieval time. As a result, the contents of the ORAs are not cleared in between test phases because the ORA contents have to be maintained throughout the different BIST phases in order to scan out results at the end of the test session. In this case, there is some loss in diagnostic resolution but it does not degrade any fault detection capabilities. Thus, BIST still detects any faulty PLBs while attaining faster test time while diagnosis can still identify faulty PLBs. The loss in diagnostic resolution is only to the extent that the failing BUT mode of operation cannot be identified.

In the case of the X and Y direct PLB connection tests, which are located on four

corners of the FPGA, avoiding alternating routing schemes results in complete testing of the PLBs, including X and Y direct connections on the four corners. The work in [33] reported lower fault coverage in eight PLBs located in the four corners.

This modified approach to dynamic partial reconfiguration of the FPGA core by the embedded processor core is analyzed and illustrated in Table 3.4 in terms of the number of processor execution clock cycles and program memory size required for reconfiguration of BIST, execution of the BIST sequence, and retrieval of the BIST results for diagnosis. This new logic BIST approach consists of the following steps:

1) Reconfigure the FPGA for BIST (download configuration file).
2) Run BIST (BIST clock is applied).
3) Partially reconfigure for the next BUT configuration via the microcontroller.
4) Repeat steps 2) and 3) until all modes of operation with the same routing scheme are run.
5) Reconfigure the ORAs into a shift register.
6) Retrieve the BIST results for diagnosis to locate faulty PLBs.
7) Reconfigure the shift register back to ORAs for a different routing scheme.
8) Reconfigure BUTs for another ORA-to-BUT routing scheme.
9) Repeat steps 2 through 6 for the next test session.

As shown in Table 3.4, the new BIST sequence produces a 41% reduction in the average number of execution clock cycles per test phase and a 49% reduction in program memory storage requirements. This is due in part to the fact that ORA results can be retrieved after each group of four BIST configurations without loss of fault detection information, instead of after every BIST configuration as is the case in the externally

controlled logic BIST approach in [33]. Another factor is that the externally controlled logic BIST approach required running four test sessions (west, east, south, and north) for complete testing of PLB logic while the modified AVR-assisted logic BIST approach only requires running two test sessions (west and east), twice each (one for each routing scheme). Thus, the modified AVR-assisted logic BIST requires fewer reconfiguration clock cycles to completely test the PLBs in the FPGA core. One penalty of having only west and east test sessions is that they are not sufficient to test additional routing faults associated with horizontal transmission gates, which make PLB-to-global bus connections. Whereas the externally controlled logic BIST approach [33] is able to detect them in addition to the PLB logic faults. However, it can be solved by running similar sessions for north and south, or as an alternative, routing BIST requires north and south test sessions for repeaters and the horizontal transmission gates should be tested in those BIST configurations.

Table 3.4 Logic BIST Reconfiguration Improvement

Compared Features	AVR-assisted	Modified AVR-assisted
Total Number of Test Phases	16	16
Number of Downloads Required	4	2
Average Number of Lines of Code	350 x 4	450 x 2
Total Program Memory Size (Bytes)	1,694 x 4	1,736 x 2
Total Execution Clock Cycles	1,844,916	1,086,462
Average Cycles per Test Phase	115,307	67,904

By comparing the improved and original BIST configurations from [33], a total memory reduction factor of 7.7 and a test time speedup by a factor of 7.4 are achieved, as shown in Tables 3.5 and 3.6 respectively. Note that all the data shown are based on the AT94K40 device which has a PLB array size of 48x48. But it should be noted that all

tests were also developed for and executed on AT94K10 devices, which have a PLB array size of 24x24. The speedup in testing time in the AT94K10 is less due to the smaller array size.

Table 3.5 Total Memory Reduction

	Download [33]	**Modified AVR-assist**	**Memory Reduction**
Total Configurations	65 Kbytes × 16 files	67.6 Kbytes x 2 file	7.7

Table 3.6 Total Test Time and Speed Up

	Download [33]	**Modified AVR-assist**	**Speed up Factor**
Download (1MHz)	523msec x 16	541msec x 2	7.75
Run time (25MHz)	1msec x 16	22msec x 2	0.36
Total BIST Time	8.4 sec	1.13 sec	7.4

CHAPTER FOUR
AVR GENERATED FPGA LOGIC BIST

The previous chapter described how the embedded AVR microcontroller can assist in the BIST of the embedded FPGA core using an initial external configuration download to the FPGA. Significant improvements in the BIST performance were obtained by using the microcontroller to reconfigure the FPGA for the subsequent BIST configurations instead of downloading those BIST configurations. This chapter extends the idea to eliminate all external downloads to the FPGA by replacing those download bitstream files with a single AVR program. The program contains algorithmic routines to reconfigure the FPGA core for every BIST configuration. Furthermore, the AVR becomes the test controller by executing the BIST sequence and retrieving the BIST results. The detailed development and debugging process of AVR generated BIST configurations will be discussed. Finally, the improvement over the conventional FPGA BIST will be presented by showing the BIST time speedup and configuration memory storage reduction factor.

4.1 Development of C Program for Logic BIST Generation

The goal is to develop a program for algorithmic reconfiguration of the FPGA core for every BIST configuration. This requires only a single download to the program memory without configuration of the FPGA core. If the program is sufficiently small, it can reside in the program memory without the need for any download. The key point in

this approach is to have an algorithmic routine to reconfigure the FPGA for different BIST configurations [28]. If fast enough, the BIST program can be more frequently used during idle intervals in system operation for high reliability, high availability applications.

To accomplish the first goal, minimizing the size of the program, the BIST architecture must be regular to facilitate an efficient reconfiguration algorithm. In addition, the order of the configuration process must be efficient. The configuration order also impacts the second goal, minimizing test execution time. The test execution time can also be reduced by not retrieving test results from the ORAs after each BIST configuration but instead, using dynamic partial reconfiguration to execute many BIST configurations before retrieving test results. There is some loss of diagnostic resolution in that the faulty functionality within a PLB can no longer be identified. However, there is no loss in diagnostic resolution in that faulty PLB(s) can still be identified [34].

4.1.1 Implementation Issues and Considerations

Since BIST configurations generated from MGL in [33] test all logic resources in the BUTs with total of four BIST configurations, the first goal was to program the embedded processor to perform the same tests by replicating the BIST structures (TPGs, BUTs, ORAs, etc) which were generated by MGL, replacing all the BIST configuration downloads with a single AVR program.

One of the limitations of this approach, of having a single processor program to test all the resources in an FPGA, is that the time required for developing and debugging the program can be significant. Most of the FPGA design tools provide a graphical representation of the design to be implemented in the FPGA to help in debugging the design. Atmel provides a tool called Figaro which graphically represents how the design

is mapped onto the FPGA, provided the original design is described using MGL, VHDL or Verilog. On the other hand, if the entire BIST configuration is generated through partial reconfiguration by the AVR, debugging the design without any tool support can become quite tedious and error-prone. If the AT94K series SoCs were capable of dynamic configuration readback via the AVR processor core (which is not the case), BIST development time would be greatly reduced by facilitating read-modify-write operations to the configuration memory. Instead, the BIST configurations previously developed and verified using MGL as described in [33] must serve as a baseline for developing and debugging the desired program for the AVR processor core.

In order to develop the AVR program, we must determine the BIST configuration that has to be generated initially and also the proper order of subsequent configurations so as to minimize the configuration time from the AVR. We use the BIST configurations originally developed using MGL [33] to help determine these two issues. While the graphical representation of the design helps in planning the reconfiguration routines as to how the different resources (logic, routing, repeaters, and clocks) have to be configured, the MGL generated bitstream helps in determining the order in which to write various configuration bytes for different resources so as to make the algorithmic reconfiguration routines efficient in terms of speed and size as well as power dissipation during reconfiguration.

After developing and verifying the routines for the initial configuration, routines are then developed for reconfiguring the BUTs to test the subsequent modes of operation. The BIST reconfiguration order has to be carefully considered and arranged since, if different resources are configured independently, there is possibility of destroying the

previously configured bytes, since some of the configuration bytes are shared by different programmable resources. For example, as shown in Figure 4.1, if reconfiguring repeater connection ‘A-C’ requires writing a logic 1 on the least significant bit location of the repeater configuration byte, then writing a byte ‘00000001’ may turn off the existing activated CIP that is needed for BIST.

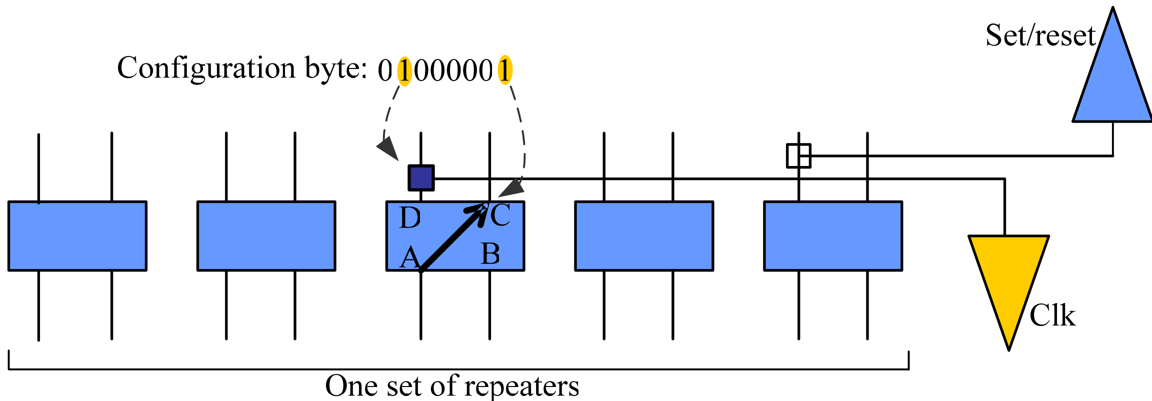


Figure 4.1 Illustration of Configuration Byte Shared by More then One Resources

4.1.2 Efficient Sequence of On-chip Dynamic Configuration of FPGA BIST from AVR

To find an efficient configuration sequence when reconfiguring the FPGA core from scratch, a primary goal is to avoid the risk of overwriting a configuration bit that has been previously written and, as a result, inadvertently injecting errors into a BIST configuration. The following considerations help to minimize this risk. First, do not configure more than what is needed when configuring the FPGA for the test. For example, the BIST clock routing need not be configured until the other BIST components are configured and ready for the BIST clock. When the BIST clock is ready to be applied for the BIST sequence, the scan chain output path from the ORAs is not needed and should

only be configured right before the BIST results are to be retrieved. Second, keep track of configuration bytes that control more than one kind of programmable component (such as repeaters with global clocks and resets, for example). Third, configure resources that are regular and repeat over the entire array first (such as the BUTs and ORAs, for example) and then configure the resources that are local to a specific area in the FPGA array (such as the clock, scan chain output signal, and TPGs).

The algorithmic reconfiguration program for the embedded AVR core was developed in C. The program's subroutines and reconfiguration sequence is arranged in the following order:

1. *Clear the FPGA* - Instead of the chip reset, this subroutine clears the entire FPGA configuration memory contents to ensure that the BIST components will be configured into an empty FPGA. It clears all configuration memory bytes associated with PLBs, repeaters, clocks, set/resets, flip-flops, free RAMs, and I/O buffers [7]. This routine is also executed when there are transitions between test sessions as shown in Figure 2.15.

2. *Initialize the ORAs* - This subroutine configures the local routing resources associated with each ORA and its LUTs to function as a comparison-based ORA. It configures the ORAs to either routing scheme 1 or 2, as shown in Figure 2.15, and resets the ORA flip-flop contents to logic 0.

3. *Initialize/reconfigure the BUTs* - This subroutine first configures the cross points where the TPG signals and buses to the BUT inputs are connected along the very top and the bottom of the FPGA array. When the routine is used to reconfigure the BUTs for the next BIST configuration, depending on the current test session and the BIST configuration, it changes the local routing connecting the BUTs as well as the

programmable logic resources inside the BUTs. The BUTs are also reset through this subroutine, meaning the flip-flops in all of the BUTs are initialized to either logic 0 or logic 1 (depending on the BUT configuration) to ensure correct BIST operation. In fact, this feature provides additional testing of the flip-flops that cannot be tested by downloading individual BIST configurations into the FPGA core and illustrates the improved controllability obtained with partial reconfiguration from the embedded processor core.

4. *Initialize the TPGs* - This subroutine programs two 5-bit counters in the TPG column of the PLB array. It also performs all local, global, and repeater routing between the TPG PLBs, as well as the TPG to BUT signal connections as shown in Figure 2.15. When configuring repeaters in this step, writing to some of the repeater bytes needs extra attention because some of the bytes in repeaters also include global clock and set/reset control bits. This subroutine also initializes the TPG flip-flops to logic 0 to ensure that the TPGs are synchronized prior to execution of the BIST sequence.

5. *Route BIST clock controlled by the AVR interface* - This subroutine connects the FPGA Write Enable line (FPGAIOWE as shown in Figure 2.8 and 4.2) from the AVR interface to one of the global clock input lines of the FPGA core so that the BIST clock signal can be distributed to all of the PLBs. FPGAIOWE is used to generate and control the BIST clock from the AVR. Since the AVR-FPGA interface cannot be described and programmed from the MGL, the circled points shown in Figure 4.2 illustrate dynamic reconfiguration from the AVR. Finally, this subroutine configures the clock control settings such as clock invert bits for the TPGs, BUTs, and ORAs which are the last steps before running the BIST.

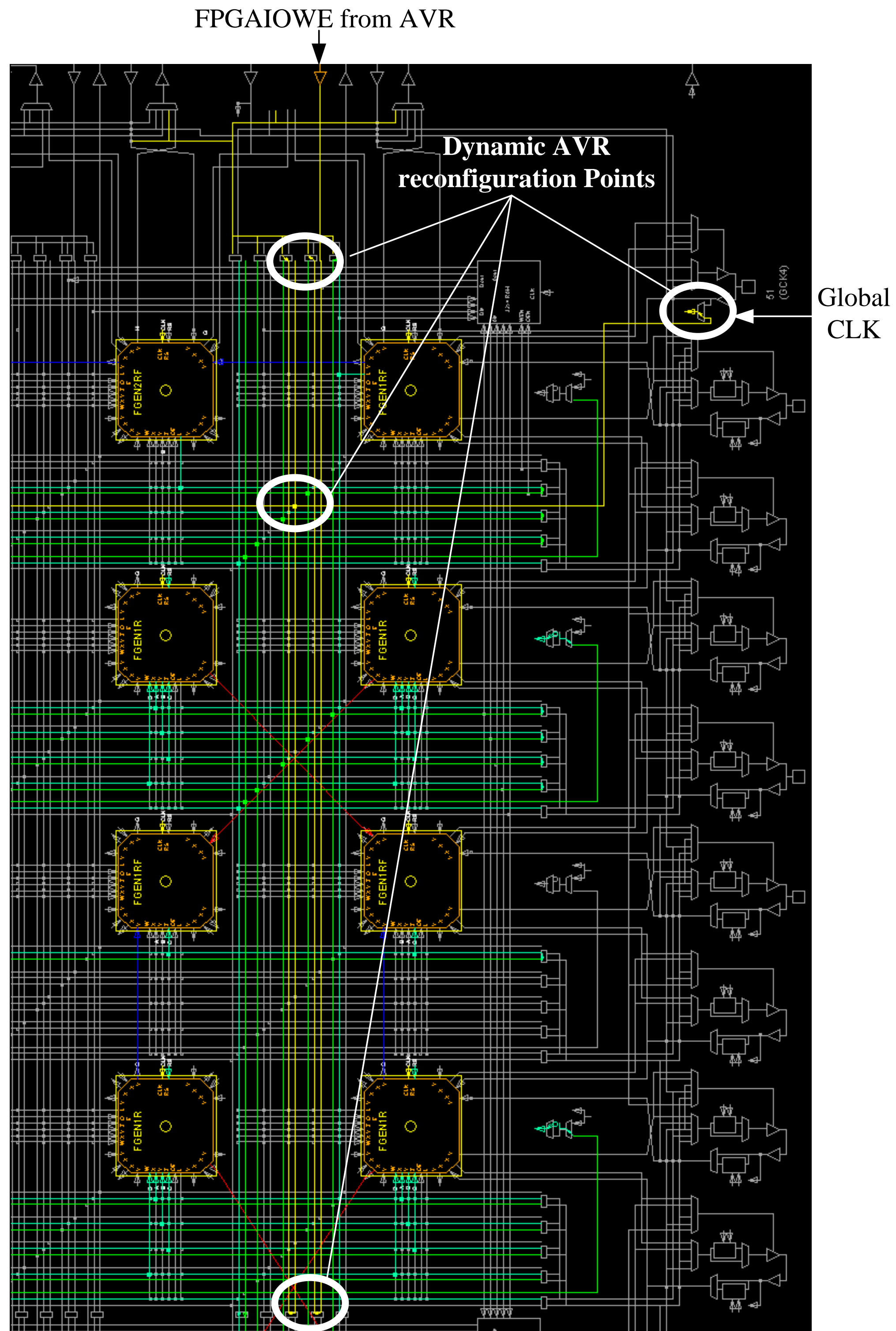


Figure 4.2 FIGARO Illustration of How AVR Connects to a Global Clock Buffer

6. *Run the BIST* - In this subroutine, the embedded AVR processor generates the BIST clock cycles to the FPGA core to run the complete BIST sequence. A control register 'FISCR' is assigned to decode and connect one of four I/O registry addresses ('FISUB' is decoded in Figure 4.3) to the AVR-FPGA data bus. A clock cycle is generated by writing a dummy value to the 8-bit 'FISUB' as shown in Figure 4.3 which causes a clock cycle to be generated at the 'FPGAIOWE' pad. The TPGs generate the test patterns and any ORAs that observe mismatches in the outputs of their two neighboring BUTs will latch a logic 1.

```
void run_lclk(unsigned char n) {
        unsigned char i;
        FISCR=0x02; //Setting control register to decode FISUB
        for (i = 0; i < n; i++) {
         FISUB=0x00; //Writing a dummy value to AVR-FPGA data bus
        }
}
```

Figure 4.3 AVR Code of Generating N Clock Cycles to 'FPGAIOWE'

7. *Reconfigure the ORAs as a scan chain* - At the completion of the BIST sequence, the ORAs will hold the test results to be read by the AVR. During this subroutine, all of the ORAs are dynamically reconfigured as a scan chain without affecting the contents of the ORA flip-flops as discussed in previous chapter.

8. *Route the scan out data to the AVR interface* - When ORA results are scanned out to the AVR core, the bidirectional data bus between the AVR and FPGA core must be used to shift the ORA results to the AVR for storage in the data SRAM. This subroutine routes a signal path from the output of the last ORA in the shift register to one of the 8-bit data bus lines (AVR DATA IN 0 in Figure 4.4) to the AVR core.

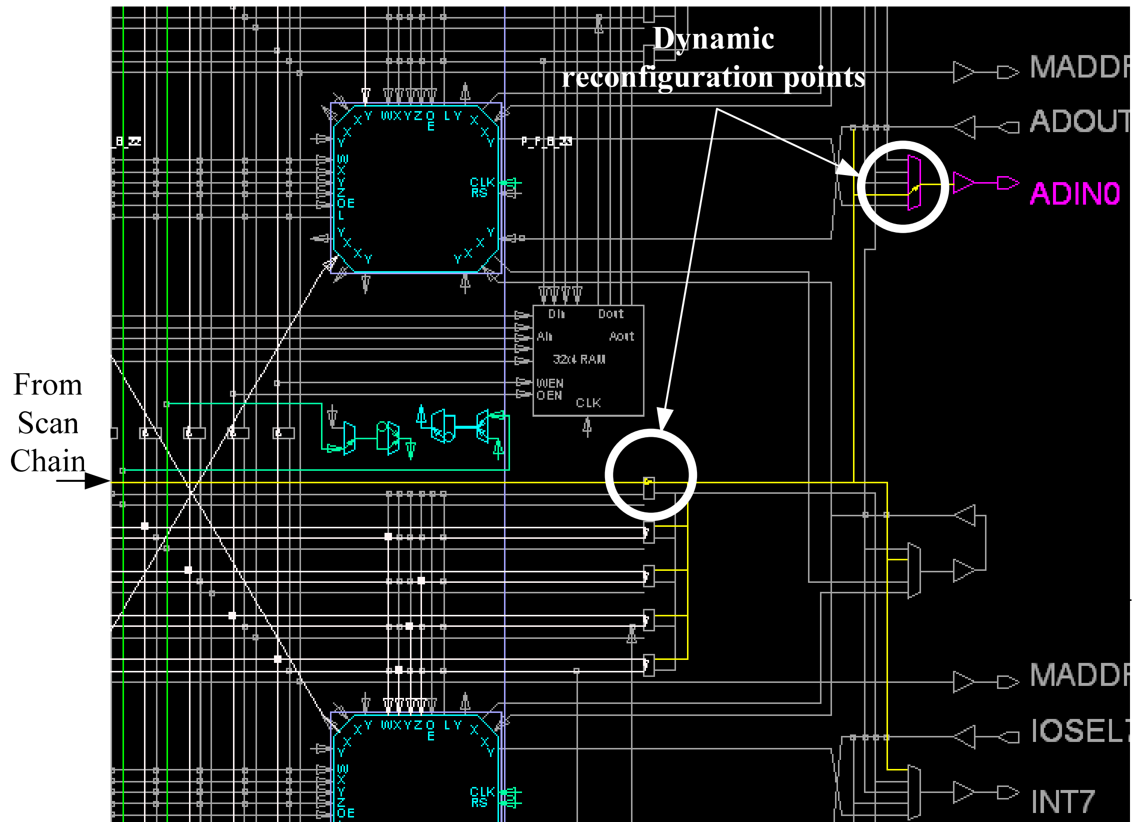


Figure 4.4 ADIN0 Pad connected from Scan Chain

9. *Retrieve ORA results and store in the data SRAM for fault detection analysis and/or diagnosis* - According to the instruction given to the embedded processor by a higher computing source (a PC in our case), the AVR can retrieve the ORA results after every BIST configuration or after multiple BIST configurations. In the latter case, there is some loss in diagnostic resolution but it does not degrade any fault detection capabilities. Thus, it still detects and identifies any faulty PLBs while attaining faster test time. The AVR can either return the actual test results (the contents of the ORAs) or it can perform an on-chip diagnostic procedure [34] as instructed by the higher computing source. In the event that the AVR is instructed to perform diagnosis, it returns a list of all faulty PLBs and their locations in the array for the BIST configuration(s) just executed.

4.2 Debugging Technique for Developing Logic BIST from Scratch

Atmel's MGL and Figaro IDS tools can be used to a certain extent to help in speeding up the development and debugging process for the AVR program, which consists of the various configuration subroutines. In order to use an MGL program in debugging, a completely developed and verified MGL-based BIST configuration from [33] was modified to omit certain configurations of the BIST components in the FPGA core, as illustrated in Figure 4.5. An AVR program was then developed to write the configuration of the original components missing in the modified MGL configuration. The MGL-generated bit stream and the compiled AVR code are then combined into a single bit stream using Atmel System Designer and downloaded into the SoC. The MGL-based BIST configuration, with missing BIST components, will report failures upon running BIST. However, if the BIST runs correctly after the execution of the AVR configuration routine, then we will have verified, at least to a certain extent, that the configuration subroutine correctly replaces the missing BIST component. Each BIST component is removed, one at a time, from the MGL code and combined with an appropriate AVR configuration subroutine to verify all of the AVR configuration subroutines for all BIST components. In this manner, we are essentially using the BIST architecture to test itself for design verification. A fault injection emulation technique is then used by reconfiguring certain PLBs to have faults and to verify that the BIST accurately detects and diagnoses these faults [17]. When there is no MGL-generated configuration data to be downloaded into the FPGA core, we are left with one AVR program which consists of all the logic BIST reconfiguration subroutines to be downloaded to the program memory of the SoC.

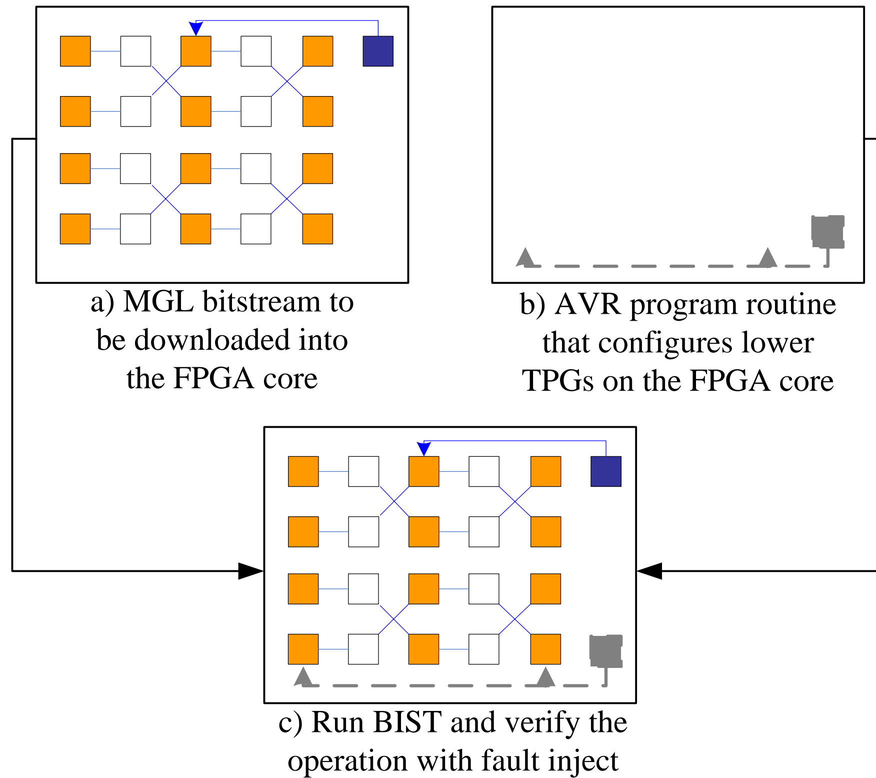


Figure 4.5 Use of MGL to Verify AVR Routines

4.3 Experimental Results

The AVR program, consisting of the various subroutines described Section 4.1, is summarized in Table 4.1 in terms of individual program memory storage requirements, number of non-commented lines of source code, and the number of processor execution cycles for each configuration subroutine. Note that Table 4.1 contains the detailed functional level analysis of the final program which compiles to an Intel HEX file format to be downloaded to the program memory of the chip to run all of the west and east test sessions, which are equivalent to the complete set of the logic BIST configurations developed in [33]. Almost all of the subroutines developed for the west test session were parameterized so that they can be reused in the east session to reduce the program memory size. The main difference between the two test sessions is the direction of the

TPG signal flow across the top and the bottom of the array, which corresponds to horizontal repeaters on the top and bottom rows. The rest of the configuration subroutines for the BUTs and ORAs are reused simply by applying offsets to the column locations. TPG configuration routines are also reused by changing the TPG column location from FPGAX = 0 for the first (west) test session to FPGAX = ArraySize-1 for the second (east) test session. Thus, most of the configuration subroutines take two parameters: directions of the TPG signal flow to the BUTs (west or east) and the BIST configuration for the particular BUT mode of operation to be tested.

Table 4.1 Total Configuration Routine Analysis

BIST Reconfiguration Subroutines	Program Memory Size (KBytes)	Number of Lines of Code (Approx.)	Processor Execution Cycles	
			K10	K40
Clear FPGA	0.492	150	59664	215128
Place/config BUT	0.834	300	25829	100360
Place/route ORA	0.22	70	14844	60686
Place/route TPG	1.486	600	4652	14866
Route BIST clock	0.234	40	1923	4911
ORA/shift reg	0.282	80	6371	24791
Route scan out	0.402	45	24879	97370
Misc.	0.726	2700	*	*
Total	4.676	4000	138162	518112

** Ignored in the total value.*

Due to the irregular structure of the TPG and associated routing, the subroutine for configuring the TPG PLBs and the TPG to BUT routing occupies a large portion of the program memory. The second biggest subroutine is the placement and reconfiguration of the BUTs, since this contains 16 different combinations of BUT test configurations as well as the flip-flop and set/reset tests in half of the BIST configurations. The complete

AVR program occupies 4.7 KBytes of program memory, which corresponds to only about 14% of the total 32 KByte program memory space available in the AT94K series SoC.

In contrast to the program memory size or the number of non-commented lines of C source code, the number of processor execution cycles listed in Table 4.1 shows a different aspect of the BIST reconfiguration program. For example, more execution cycles are required in the routines for clearing the FPGA, for placement and reconfiguring of the BUTs, and for placement and routing the ORAs. Fewer execution cycles are required for placing and routing the TPGs. This is because the first three subroutines contain extensive loops which travel along every X (FPGAX) and Y (FPGAY) location of the chip. This illustrates how the regular and algorithmic structure of the BIST architecture helps to reduce the program memory storage requirements. The K10 notation in Table 4.1 denotes AT94K10 devices, which have an array size of 24×24 PLBs, while the K40 denotes AT94K40 devices, which have a 48×48 PLB array. The column showing processor execution cycles for K40 is greater by a factor of approximately four, indicating that the increase in reconfiguration time and retrieval of results is linear with the device size.

Subroutines for applying the diagnostic procedure to the BIST results and for communicating with the higher controlling source also increase the program memory storage requirements. Also, due to the additional bits added from the tool that generates the final bit-stream, the actual file size to be downloaded to the program memory of the AVR increases from 4.7 Kbytes to 12.6 Kbytes as summarized in Table 4.2.

Table 4.2 Actual Download File Size (KBytes)

All Configurations	On-Chip Diagnosis + others	Added by System Designer Bit Generation	Total
4.676	2.5	5.419	12.6

With the internal BIST reconfiguration process executed by the AVR core, we achieve much better external memory storage requirements and faster testing time when compared to downloading individual BIST configurations into the FPGA. This is summarized in Table 4.3 for external memory storage and in Table 4.4 for total test time. The data shown in these tables are for a AT94K40 device with a 48×48 PLB array.

Table 4.3 Total Memory Reduction

	Download[33]	AVR-generated	Memory Reduction
Total Configurations	65 Kbytes × 16 files	12.6 Kytes x 1 file	83

Table 4.4 Total Test Time and Speed up

	Download[33]	AVR-generated	Speed up Factor
Download (1MHz)	8.371 sec.	0.101 sec.	83.077
Run-time (25MHz)	0.016 sec.	0.085 sec.	0.193
Total BIST Time	8.387 sec.	0.186 sec.	45.125

The total test time is calculated by adding the download time and BIST execution time (or run-time as listed in Table 4.4). The external download is done using a maximum clock speed of 1MHz since all external downloads, which involve a check for download errors (the check-sum function) at the FPGA, can run at a maximum configuration clock frequency of 1 MHz [7]. Since the AVR can run at 25 MHz clock speed, BIST execution time is calculated assuming that the BIST clock runs at 25 MHz. This data was obtained from simulation on the 'Codevision AVR' C compiler and 'AVR Studio' for both conventional and processor-only cases and was also verified against actual download and execution times in several AT94K40 devices.

As a result of the single AVR program for BIST reconfiguration, we obtain a factor of 45 speed-up in total test time and a factor of 83 reduction in external memory requirements for storing BIST configurations. It is interesting to note that the run-time in Table 4.4 increases for AVR BIST reconfiguration. This is due to the fact that the embedded processor core is doing all the reconfiguration, execution, and retrieval of BIST results while in the download of BIST configurations, the processor core is only used to reconfigure the ORAs into shift registers at the end of the BIST sequence for retrieval of the test results. With this consideration, the increase in run-time seems surprisingly small.

CHAPTER FIVE
SUMMARY AND CONCLUSIONS

This chapter summarizes the thesis and emphasizes the main contributions, followed by possible future research subjects. The summary section addresses problems in developing BIST configurations on commercially available SoC devices and discusses how the problems were solved through this thesis work. Experimental results summarize and discuss improvements in BIST which utilizes embedded AVR microcontroller as a BIST component, followed by a discussion of possible future research topics.

5.1. Summary

The PLBs in the embedded FPGA core in AT94K FPSLIC devices from Atmel were tested with 99.7% fault coverage in the thesis work described in [33]. One of the difficulties in developing BIST configurations on the device was due to that fact that each PLB has limited amount of resources which made it impossible to have an ORA with shift register capabilities in a single PLB [33]. With further investigation of the SoC device, we determined that the embedded FPGA can be dynamically reconfigured from the embedded AVR core, so that the BIST architecture could start with comparison-based ORAs, where each ORA monitors two BUTs, and scan out the ORA results by dynamically reconfiguring the ORAs to shift registers at the end of the BIST sequence. As a result, BIST configuration bitstreams that are downloaded to the SoC device consist of FPGA configurations as well as the AVR program for ORA to shift register

reconfiguration. Therefore, the work in [33] was completed and concluded with the support of the AVR partial reconfiguration which influenced most of the ideas in this thesis.

In order to improve test time, which is dominated by BIST configuration download time in [33] and other previous FPGA BIST works in [10]-[12], [18]-[21], the role of the AVR has been extended to do partial reconfiguration of BUTs for each test phase. This eliminates the need for new downloads to the FPGA and any chip reset between the BIST configurations. Without a chip reset, the AVR must reconfigure more resources, such as resetting flip-flops in ORAs and TPGs, which costs additional clock cycles. However, faster test time than the approach in [33] was achieved because the AVR reconfiguration runs at a clock frequency of 25MHz while the external download can only run at 1MHz in order to provide error checks on the configuration downloads files. The improved BIST approach, however, needed the initial download to the FPGA for each test session.

With the BIST approach that requires initial download to the embedded FPGA followed by dynamic reconfigurations of the FPGA from the AVR between test phases, we focused on imitating the exact BIST architecture and sequence as done in [33]. However, with the dynamic partial reconfiguration capability of the AVR, any modification can be made to the BIST architecture in such a way that improves the total test time. One of the modifications made was in the local X and Y routing scheme that alternates as shown in Figure 3.9. Instead of the alternate routing schemes, one routing scheme (scheme 1) is maintained to run four BUT configurations, with ORA results scanned out after the all modes of BUT operation have been tested. Next, the FPGA is reconfigured to have another routing scheme (scheme 2) to run four BUT configurations

again. This gave better speedup in test time, further reductions in memory storage requirements, and improved fault coverage at the corners on the FPGA core.

As the AVR's dynamic partial reconfiguration capability was proven to give flexibility of developing BIST configurations, the idea arose that the AVR could program the entire FPGA core from the very beginning without the need of any external download. All BIST components such as TPGs, BUTs and ORAs were carefully analyzed and a C program was developed so that the AVR can write certain configurations to certain parts of the FPGA to perform particular BIST functions such as TPG, ORA, or BUT. The drawback of this approach was excessive development time due to that fact that there are no tools that can visualize the dynamic cache logic of AVR writing to the FPGA configuration memory. In the end, this approach resulted in a single program that replaces 16 external downloads, achieving better test time speedup and memory storage requirements reduction than any other approach.

5.2. Improvements in Total Test Time and Configuration Memory Requirements

As a result, we have achieved improvements in the total test time and memory storage requirement for BIST configurations throughout the development. The final result is a single program executed by the embedded processor core for the complete reconfiguration, execution, and retrieval of test results during BIST of the programmable logic resources in the FPGA core of the Atmel AT94K series configurable SoC, as summarized in Tables 5.1, 5.2, and 5.3. As can be seen, replacing configuration downloads to the chip requires more AVR program size and processor execution cycles. However, this is a good trade-off since it eliminates FPGA configuration downloads which dominate total BIST configuration memory storage requirement and test time.

Table 5.1 Logic BIST Reconfiguration Comparison

Compared Features	Download [33]	AVR-assisted	Modified AVR-assisted	AVR-generated
# of Downloads Required	16	4	2	1
Total Number of ORA Retrieval	1 x 16	4 x 4	2 x 2	4 x 1
Number of lines of code	127 x 16	350 x 4	450 x 2	1,300 x 1
Total Program memory bytes	764	1,694 x 4	1,736 x 2	4,676 x 1
Total Processor execution cycles	25,570	1,844,916	1,086,462	2,127,686

Table 5.2 Total Configuration Memory Reduction

	Download [33]	AVR-assisted	Modified AVR-assisted	AVR-generated
Total Configurations	65 Kbytes × 16 files	67.5 Kbytes x 4 files	67.6 Kbytes x 2 files	12.6 Kbytes X 1 file
Memory Reduction	1	3.9	7.7	83

Table 5.3 Total Test Time and Speed Up

	Download [33]	AVR-assisted	Modified AVR-assisted	AVR-generated
Download (1MHz)	523msec x 16	540msec x 4	541msec x 2	101msec x 1
Run time (25MHz)	1msec x 16	19msec x 4	22msec x 2	85msec x 1
Total BIST Time	8.4 sec	2.24 sec	1.13 sec	0.186 sec
Speed Up Factor	1	3.75	7.4	45

5.3. Main Contribution

The ability to perform dynamic partial reconfiguration of the FPGA core from the embedded processor core provides a major testing capability. However, the non-existent configuration memory readback capability, as well as lack of graphical tool support that can show dynamic partial reconfigurations in the FPGA, make the SoC testing (and test development) much more difficult. Therefore, a unique way of debugging and verifying AVR's dynamic reconfiguration was used by combining the AVR program with previously verified MGL-generated FPGA BIST configurations so that the resultant download bitstream can be run in the chip for the AVR program verification. Finally, by having a single program downloaded into the program memory of the embedded processor to reconfigure the FPGA core algorithmically, downloads to the FPGA core are eliminated, resulting in significant reduction in the total testing time (a factor of 45) as well as the configuration memory required (a factor of 83) compared to the previous work done in [33]. The single AVR-generated BIST and diagnostic program is sufficiently small to reside on-chip for on-demand BIST and diagnosis of the programmable logic resources in the FPGA core of the SoC.

The same techniques discussed in this thesis can also be applied to BIST of the programmable routing resources. Previous work [33] showed that the number of routing BIST configurations required (48) was three times more than the logic BIST configurations (16) for the same device tested in this thesis. In other words, three times more configuration downloads could be replaced with a single AVR program resulting in a better reduction of configuration memory requirements and total test time.

5.4. Future Research

There are two areas that can be considered for future research related to this thesis. They are the embedded AVR microcontroller itself and a dynamic reconfiguration visualization tool. Throughout this thesis work, the microcontroller was assumed to be fault-free. Without this assumption, it is not certain that the FPGA BIST configurations generated from the AVR are correct. If the AVR can be tested also, then it would support the argument that the partial reconfigurations that are made by the AVR to the FPGA core can be trusted. An AVR test could be broken into several parts (such as ALU, stack, dynamic reconfiguration logic, peripherals and etc), and critical parts that are mostly used for the BIST reconfiguration shown in this thesis can be selected and tested individually.

The AVR leads to the other subject, the dynamic reconfiguration visualization tool. If developers can see how the FPGA configurations are being changed by the AVR, there would be no need for spending excessive time in developing AVR-generated BIST configurations that currently requires developer's ability of imagining any design changes (by looking at the AVR program) inside the FPGA core resulting from the AVR dynamic reconfiguration. Currently, a simulation program called "AVR Studio" has capabilities to simulate and record various parts in the AVR (such as registers, ports, processor clock cycles and etc), and the visualization tool could be built on top of the AVR simulator as a form of software plug-in module. The visualization tool will not help general users to debug designs since most designers would not consider the physical design structure or layout which is typically done by the CAD tool. However, the visualization tool will be very useful to test engineers, especially those who are related to

the topics and techniques discussed in this thesis.

REFERENCES

[1] D. W. Kim, W. I. Cho, "Development of Universal BIST Tool to provide BIST Environment," *Journal of New Technology*., Kwang-woon Univ, Vol. 26, pp. 65-75, 1997.

[2] R. Drechsler, "Synthesizing checkers for on-line verification of System-on-Chip designs," *Proc. IEEE International Symp. on Circuits and Systems*, Vol. 4, pp. 25-28, 2003.

[3] B. Bentley, "Validating the Intel Pentium 4 microprocessor," *Proc. ACM/IEEE Design Automation Conf.*, pp. 244-248, 2001.

[4] J. J. Engel, T. S. Guzowski, A. Hunt, D. E. Lackey, L. D. Pickup, R. A. Proctor, K. Reynolds, A. M Rincon, D. R. Stauffer, "Design methodology for IBM ASIC products," *IBM Journal of Research and Development,* Vol 40, No. 4, pp.387-407, 1996.

[5] S. Pontarelli, G. C. Cardarilli, A. Malvoni, M. Ottavi, M. Re, A. Salsano, "System-on-Chip Oriented Fault-Tolerant Sequential Systems Implementation Methodology," *Proc. IEEE International Symp. on Defect and Fault Tolerance in VLSI Systems*, pp. 24-26, 2001.

[6] __, Agilent Technologies Home Page: http://www.agilent.com

[7] __, Atmel Home Page: http://www.atmel.com

[8] Hyper Dictionary Home Page: http://www.hyperdictionary.com

[9] M. Schrader, R. McConnell, "SoC Design and Test Considerations," *Proc. Design, Automation and Test in Europ*e, pp. 202-207, 2003.

[10] M. Abramovici and C. Stroud, "BIST-Based Test and Diagnosis of FPGA Logic Blocks," *IEEE Trans. on VLSI Systems*, Vol. 9, No. 1, pp. 159-172, 2001.

[11] C. E. Stroud, K. N. Leach, T. A. Slaughter, “BIST for Xilinx 4000 and Spartan Series FPGAs: A Case Study,” *Proc. IEEE International Test Conf.*, pp. 1258-1267, 2003.

[12] S. Wijesuriya, “Built-In Self-Test of Field Programmable Gate Array Interconnect,” M.S.E.E. Thesis, University of Kentucky, 1997.

[13] __, Field Programmable Gate Arrays Data Book, Data Book, Lucent Technologies, January 1998.

[14] J. Rose, A. El Gamal, A. Sangiovanni-Vincentelli, "Architecture of Field-Programmable Gate Arrays," *Proc. IEEE*, Vol. 81, No. 7, pp. 1013-1029, 1993.

[15] S. Brown, and J. Rose, "FPGA and CPLD Architectures: A Tutorial," *IEEE Design & Test of Computers*, Vol. 13, No. 2, pp. 42-57, 1996.

[16] K. Iijima, A. Akar, C. McDonald, and D. Burek, “Embedded Test Solution as a Breakthrough in Reducing Cost of Test for System on Chips,” *Proc. IEEE Asian Test Symp.,* pp 311 – 316, 2002.

[17] C. E. Stroud, A Designer’s Guide to Built-In Self-Test, Springer-Verlag, New York, 2002.

[18] C. Stroud, S. Wijesuriya, C. Hamilton and M. Abramovici, “Built-In Self-Test of FPGA Interconnect,” *Proc. IEEE International Test Conf.*, pp. 404-411, 1998.

[19] C. Stroud, S. Konala, P. Chen, and M. Abramovici, “Built-In Self-Test for Programmable Logic Blocks in FPGAs (Finally, A Free Lunch: BIST Without Overhead!),” *Proc. IEEE VLSI Test Symp.*, pp. 387-392, 1996.

[20] C. Stroud, E. Lee, S. Konala, and M. Abramovici, "Using ILA Testing for BIST in FPGAs," *Proc. IEEE International Test Conf.*, pp. 68-75, 1996.

[21] C. Stroud, E. Lee, M. Abramovici, "BIST-Based Diagnostics for FPGA Logic Blocks," *Proc. IEEE International Test Conf.*, pp. 539-547, 1997.

[22] S. Brown, and J. Rose, "FPGA and CPLD Architectures: A Tutorial," *IEEE Design & Test of Computers*, Vol. 13, No. 2, pp. 42-57, 1996.

[23] K. Y. Ko, Mike W. T. Wong, and Y. S. Lee, "Testing System-On-Chip by Summations of Cores' Test Output Voltages," *Proc. IEEE Asian Test Symp.,* pp 350 – 355, 2002.

[24] J. Sunwoo, S. Garimella, C. Stroud, "On Embedded Processor Reconfiguration of Logic BIST for FPGA Cores in SoCs," *Proc. IEEE North Atlantic Test Workship*, pp. 15-22, 2005.

[25] R. Rajsuman, "Testing a System-On-a-Chip with Embedded Microprocessor," *Proc. IEEE International Test Conf.*, pp. 499-508, 1999.

[26] M. Abramovici, C. Stroud, and J. Emmert, "Using Embedded FPGAs for SoC Yield Improvement," *Proc. ACM/IEEE Design Automation Conf.*, pp. 713-724, 2002.

[27] G. Zeng, H. Ito, "Hybrid BIST for System-On-a-Chip Using an Embedded FPGA Core," *Proc. IEEE VLSI Test Symp.*, pp. 353-358, 2004.

[28] C. Stroud, J. Sunwoo, S. Garimella and J. Harris, "Built-In Self-Test for System-on-Chip: A Case Study", *Proc. IEEE International Test Conf.*, pp. 837-846, 2004.

[29] __, AVR Assembly Home Page: www.attiny.com

[30] S. Donthi and R. Haggard, "A Survey of dynamically reconfigurable FPGA devices," *Proc. IEEE Southeastern Symp. on System Theory,* pp. 422-426, 2003.

[31] __, "Integrated Development System – Figaro User Guide", Atmel Corp., 2002.

[32] C. Sroud, J. Harris, S. Garimella and J. Sunwoo, "Built-In Self-Test Configurations for Atmel FPGAs Using Macro Generation Language," *Proc. IEEE North Atlantic Test Workshop*, pp. 83-90, 2004.

[33] J. Harris, "Built-In Self-Test Configurations for Field Programmable Gate Array Cores in Systems-On-Chip," M.S.E.E. Thesis, Auburn University, 2004.

[34] C. Stroud, S. Garimella, J. Sunwoo, "On-Chip BIST-Based Diagnosis of Embedded Programmable Logic Cores in System-on-Chip Devices," *Proc. ISCA International Conf. on Computers and Their Applications*, pp. 308-313, 2005.