ON-CHIP DEBUG ARCHITECTURES FOR IMPROVING OBSERVABILITY DURING POST-SILICON VALIDATION

ON-CHIP DEBUG ARCHITECTURES FOR IMPROVING OBSERVABILITY DURING POST-SILICON VALIDATION

BY
EHAB ALFONS ANIS DAOUD, B.Sc., M.Sc.
SEPTEMBER 2008

A THESIS
SUBMITTED TO THE SCHOOL OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE
DOCTOR OF PHILOSOPHY

McMaster University

DOCTOR OF PHILOSOPHY (2008) — McMaster University
(Electrical and Computer Engineering) — Hamilton, Ontario, Canada

TITLE:	On-Chip Debug Architectures for Improving Observability during Post-Silicon Validation
AUTHOR:	Ehab Alfons Anis Daoud, B.Sc., M.Sc. (Electronics and Communications Engineering, Cairo University, Egypt)
SUPERVISOR:	Dr. Nicola Nicolici
NUMBER OF PAGES:	xiv, 139

Abstract

Post-silicon validation has become an essential step in the design flow of system-on-chip devices for the purpose of identifying and fixing design errors that have escaped pre-silicon verification. To address the limited observability of the circuits during post-silicon validation, embedded logic analysis techniques are employed in order to probe the internal circuit nodes at-speed and in real-time. In this dissertation, we propose novel on-chip debug architectures and the associated debug methods, which improve observability during at-speed post-silicon validation.

First, we propose a novel embedded debug architecture that enables real-time lossless data compression in order to extend the observation window of a debug experiment. The proposed architecture is particularly suitable for in-field debugging on application boards that have sources of non-deterministic behavior, such as asynchronous interfaces. To quantify the performance gain from using lossless compression in embedded logic analysis, we present a new compression ratio metric that captures the trade-off between the area overhead and the increase in the observation window.

Second, we propose a novel architecture based on lossy compression. This architecture enables a new debug method where the designer can iteratively zoom only in the intervals that contain erroneous samples. Thus, it is tailored for the identification of the hard-to-detect functional bugs that occur intermittently over a long execution time. When compared to increasing the size of the trace buffer, the proposed architecture has a small impact on silicon area, while significantly reducing the number of debug sessions. The new debug method is applicable to both automatic test equipment-based debugging, as well as in-field debugging on application boards, so long as the debug experiment can be reproduced synchronously.

Third, we address the problem of the presence of blocking bugs in one erroneous module that inhibit the search for bugs in other parts of the chip that process data received from the erroneous module. We propose a novel embedded debug architecture for bypassing blocking bugs. This architecture enables a hierarchical event detection mechanism to provide correct stimuli from an embedded trace buffer, in order to replace the erroneous samples caused by the blocking bugs.

It is anticipated that the main contributions presented in this dissertation will help further the adoption of embedded logic analyzers, as the main alternative to scan chains for gathering data during post-silicon validation in real-time debug environments.

Acknowledgments

I would like to thank God for giving me the strength, peace, and guidance to complete this work. I believe that all things work together for good to those who love God and all things are possible through God who strengthens me.

I am greatly indebted to my supervisor, Nicola Nicolici, for his research guidance and generous support. I greatly appreciate his enthusiasm, encouragement for high standard applied research, and his depth of knowledge in many research areas that helped me to complete this work. I am very grateful to him for enriching my knowledge with stimulating and fruitful discussions. It was an honour for me to conduct this research under his supervision.

I am grateful to all my colleagues in the computer-aided design and test research group at McMaster University, Qiang Xu, Henry Ko, Adam Kinsman, David Leung, Kaveh Elizeh, Roomi Sahi, Mark Jobes, Zahra Lak and Jason Thong, for their support, stimulating discussions during our group meetings, and for providing a lively working environment. Special thanks to my friends, Amir Gourgy, Michael Botros, Sameh Saad, and all of my friends at Saint Mina Church for their love and support.

I would like to thank the faculty, administrative and technical members in Department of Electrical and Computer Engineering at McMaster University for their assistance during my study and research. Many thanks to Cheryl Gies, Helen Jachna, Terry Greenlay and Cosmin Coroiu for their endless help in many matters.

Finally, I would like to express my sincere gratitude to all my family members for their love and continuous support. Words cannot express my love and great appreciation to my father Alfons, my mother Hoda, my brother Alfred, my sister Eman, my sister-in-law Vivianne, my brother-in-law Emad, my nice Alexandra and my nephew Antoine.

List of Abbreviations

ABV	Assertion-Based Verification
ASIC	Application Specific Integrated Circuit
ATE	Automatic Test Equipment
ATPG	Automatic Test Pattern Generation
BIST	Built-In Self-Test
CAM	Content Addressable Memory
CMOS	Complementary Metal Oxide Silicon
CUD	Circuit under Debug
CUT	Circuit under Test
DFD	Design for Debug
DFT	Design for Testability
FF	Flip-Flop
FIFO	First-In First-Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Description Language
I/O	Input/Output
IC	Integrated Circuit
LFSR	Linear Feedback Shift Register
LFU	Least Frequently Used
LRU	Least Recently Used
MISR	Multiple Input Signature Register
MTF	Move-to-Front

NoC	Network-on-Chip
PI	Primary Input
PO	Primary Output
PSL	Property Specification Language
RAM	Random Access Memory
RTL	Register Transfer Level
SA	Signature Analyzer
SFF	Scanned Flip-Flop
SoC	System-on-Chip
TPG	Test Pattern Generator
VLSI	Very Large Scale Integration
WDLZW	Word-based Dynamic Lempel-Ziv

Contents

Abstract **iii**

Acknowledgments **v**

List of Abbreviations **vii**

1 Introduction **1**

1.1 Design Flow of VLSI Circuits 2

1.1.1 Pre-Silicon Verification 4

1.1.2 Manufacturing Test 5

1.1.3 Post-Silicon Validation 8

1.2 Dissertation Motivation 10

1.3 Dissertation Organization and Contributions 11

2 Background and Related Work **13**

2.1 Overview of Post-silicon Validation 13

2.1.1 Debug Environments 14

2.1.2 Post-silicon Validation Techniques 15

2.2 Scan-based Observability 15

2.2.1 Scan-based DFT Methodology 16

2.2.2 Scan-based Debug Methodology 17

2.3 Embedded Logic Analysis 23

2.3.1 Basic Principle of Embedded Logic Analysis 23

2.3.2 Embedded Debug Module Architecture 25

2.3.3 Embedded Logic Analysis Related Work 27
2.4 Recent Advancements in DFD Techniques 29
2.4.1 Assertion-based Debug 30
2.4.2 Transaction-level Debug 33
2.4.3 Compression in Embedded Logic Analysis 34
2.5 Concluding Remarks 38

3 Embedded Debug Architecture for Lossless Compression 40
3.1 Preliminaries and Summary of Contributions 41
3.2 Embedded Logic Analysis Framework 42
3.2.1 Lossless Compression Requirements 43
3.2.2 The Proposed Compression Ratio Metric 44
3.2.3 Performance Analysis of Compression Algorithms 44
3.3 The Proposed Debug Architecture 46
3.3.1 Overview of Embedded Debug Module 46
3.3.2 The Proposed Encoder Architecture 47
3.4 Dictionary-based Compression Algorithms 56
3.4.1 BSTW Dictionary-based Compression Algorithm 57
3.4.2 MBSTW Dictionary-based Compression Algorithm 59
3.4.3 WDLZW Dictionary-based Compression Algorithm 60
3.5 Experimental Results 62
3.5.1 Area of the Proposed Encoder Architecture 62
3.5.2 MP3 Decoder Experiments 65
3.6 Summary 73

4 On Extending the Observation Window through Lossy Compression 75
4.1 Preliminaries and Summary of Contributions 76
4.2 Proposed Iterative Silicon Debug Framework 77
4.3 Debug Architecture for Lossy Compression 80
4.3.1 The Embedded Debug Module 80
4.3.2 Features to Extend the Observation Window 82
4.4 The Proposed Scheduling Algorithms 87

4.4.1 Algorithm for Scheduling Debug Sessions 87
4.4.2 Variable Segments Sizes at the Last Debug Level 89
4.4.3 Spatial Compression 90
4.4.4 Combining Streaming and Compression 90
4.5 Sensitivity Analysis 92
4.6 Experimental Results 94
4.6.1 Area of the Proposed Debug Module 94
4.6.2 MP3 Decoder Experiments 96
4.6.3 Random Data Experiments 98
4.7 Summary 99

5 Embedded Debug Architecture for Bypassing Blocking Bugs 100
5.1 Preliminaries and Summary of Contributions 101
5.2 Methodology for Bypassing Blocking Bugs 103
5.3 Proposed Embedded Debug Module 106
5.3.1 Overview of the Embedded Debug Module 106
5.3.2 Stimuli Selection Module 107
5.3.3 Embedded Trace Buffer Control 110
5.4 Experimental Results 112
5.4.1 Area of the Proposed Debug Module 112
5.4.2 MP3 Decoder Experiments 113
5.5 Summary 119

6 Conclusion and Future Work 120
6.1 Summary of Dissertation Contributions 121
6.2 Future Research Directions 122

Bibliography 124

List of Tables

3.1 Terminology for the Proposed Encoder Architecture 49
3.2 Area of The Proposed Encoder Architecture in NAND2 Equivalents . 63
3.3 Average Compression Ratios of Encoding MP3 Data (12×2^{20} samples) at Stereo Decoder Output using BSTW Algorithm, M=16k Bytes . . 67
3.4 Average Compression Ratios of Encoding MP3 Data (12×2^{20} samples) at Stereo Decoder Output using MBSTW and WDLZW Algorithms, M=16k Bytes 68
3.5 Average Compression Ratios of Encoding MP3 Data (12×2^{20} samples) at MDCT Input using BSTW Algorithm, M=16k Bytes 69
3.6 Average Compression Ratios of Encoding MP3 Data (12×2^{20} samples) at MDCT Input using MBSTW and WDLZW Algorithms, M=16k Bytes 70

4.1 Terminology for the Proposed Embedded Debug Module 82
4.2 Terminology for the Proposed Debug Method 85
4.3 Area of The Embedded Debug Module Architecture in NAND2 Equivalents 95
4.4 Reduction in Debug Execution Time (T_{seq}/T_{prop}) for the MP3 Data with $N = 2^{21}$, $M = 512$, $S = 4$, $Ch = 2$ 97
4.5 Reduction in Debug Execution Time (T_{seq}/T_{prop}) versus Error Percentage of Random Data for Different Burst Lengths with $N = 2^{27}$, $M = 2048$, $S = 4$ 99

5.1 Area of The Proposed Debug Module in NAND2 Equivalents 112

List of Figures

1.1 Design Flow and Verification Techniques of VLSI Circuits 3
1.2 Basic Principle of VLSI Circuits Testing 5
1.3 Basic Principle of BIST 6
1.4 Deterministic BIST with Multiple Scan Chains Architecture 7

2.1 A Single Scan Chain Architecture 16
2.2 Example of Breakpoint Debug Module 19
2.3 Scan-based Debug Architecture [93] 21
2.4 Debug Flow during Post-silicon Validation 22
2.5 Debug Framework of Embedded Logic Analysis 24
2.6 Circuit under Debug with Centralized Debug Module 24
2.7 Embedded Debug Module Architecture 25
2.8 Triggering Conditions Flow with Segmented Trace Buffer 26
2.9 Multi-core Debug Architecture with Centralized Trace Buffer 27
2.10 Multi-core Debug Architecture with Multiple Trace Buffers 28
2.11 Basic Principle of Assertion-based Debug 31
2.12 Illustrative Example of Assertion 32
2.13 Transaction-Based Communication-Centric Debug [42] 33
2.14 Compression Phases of Real-time Address Trace Compressor [60] . . . 35
2.15 Illustrative Example of State Restoration [67] 38

3.1 Embedded Logic Analysis Framework Based on Lossless Compression 43
3.2 The Proposed Embedded Debug Module 47

3.3 The Proposed Encoder Architecture with support of FIFO and Random Replacement Policies 48
3.4 The Proposed Encoder Architecture with support of Modified LRU and Modified LFU Replacement Policies 51
3.5 Illustrative Examples for Dictionary-based Compression Algorithms . 58
3.6 MP3 Decoder Architecture 65

4.1 The Iterative Flow of the Silicon Debug Process Based on Compression and Cyclic Debugging 78
4.2 The Proposed Embedded Debug Module Architecture 81
4.3 An Iterative Debug Example for the Proposed Iterative Debug Flow using Lossy Compression 84
4.4 Reduction in Debug Sessions versus Failing Intervals 93

5.1 Debug Scenarios with and without Bypassing Blocking Bugs Feature 102
5.2 Bypassing Blocking Bugs Framework 104
5.3 The Proposed Embedded Debug Module 106
5.4 Example of On-chip Stimuli Pointers 108
5.5 Stimuli Selection Module With S Stimuli Control Registers 109
5.6 Stimuli Selection Module With Stimuli Control Stored in the Trace Buffer 110
5.7 Embedded Trace Buffer CTRL 111
5.8 The Effect of the On-chip Stimuli Pointers on the Total Number of Stimuli and on the Total Number of Stimuli Pointers that are used during debug the entire Observation Window (Song = 462.38 k Samples, Sample = 2-byte word) 114
5.9 The Length of the Observation Window (k Samples) versus the On-chip Stimuli Pointers for different Sizes of Trace Buffers (k Bytes); Half of each Trace Buffer Size is used for Capturing Debug Data (Song = 462.38 k Samples) 115

5.10 The Length of the Observation Window (k Samples) versus Stimuli Segment Size Ratio without using Stimuli Data Streaming, and the Number of On-chip Stimuli Pointers equals 2 and 4, respectively (Song = 462.38 k Samples) 117
5.11 The Length of the Observation Window (k Samples) versus Stimuli Segment Size Ratio without using Stimuli Data Streaming, and the Number of On-chip Stimuli Pointers equals 8 and 16, respectively (Song = 462.38 k Samples) 118

Chapter 1

Introduction

The continuous advancements in semiconductor manufacturing have enabled the implementation of digital integrated circuits (ICs) with millions of transistors on a single silicon die. The increased design complexity leads to a longer implementation cycle, which is mainly driven by increased design and verification times. To address this problem, the core-based system-on-chip (SoC) design paradigm has emerged at the end of the past decade [61]. In this paradigm, to address the length of the implementation cycle, the system integrators reuse pre-designed and pre-verified intellectual property blocks (or embedded cores) developed by core providers. Nonetheless, given the increase in the number of embedded cores and the growing size of the user defined logic, as well as the shrinking time-to-market cycles, the contribution of verification to the overall implementation time is considerable. As a consequence, pre-silicon verification techniques, such as formal verification using model checking or functional verification through biased-random simulation, have been established as a necessary step to identify design errors (or bugs) before the design is manufactured [70].

Nonetheless, the growing complexity of SoC designs and the limitations in modeling all the physical characteristics of the design, make pre-silicon verification techniques inadequate to guarantee the first-silicon to be error-free. Because finding bugs in a fabricated design will cause a silicon re-spin, which impacts both the product cost and the time-to-market, it is desirable for design bugs to be identified and fixed

as soon as the first silicon is available. Consequently, post-silicon validation (or *silicon debug*) [3] has emerged as a necessary step in the implementation cycle of SoC designs.

This introductory chapter is organized as follows. Section 1.1 provides an overview of the design flow and the verification techniques for very large scale integrated (VLSI) circuits. Section 1.2 describes the motivation of this research. Finally, Section 1.3 outlines the main contributions of this research and presents the organization of the dissertation.

1.1 Design Flow of VLSI Circuits

The design flow of VLSI circuits is comprised of three steps: specification, implementation and manufacturing as shown in Figure 1.1. The specification step typically describes the expected functionality of the VLSI circuits. Design specifications are described in a high-level language such as SystemC [43], or in hardware description languages (HDLs) such as VHDL [79], Verilog [80] or System Verilog [98]. The register transfer level (RTL) abstraction is used in HDLs to describe the circuit's behavior as a set of transfer functions which represent the flow of signals between the registers, and the logical operations performed on those signals [39]. The process of transforming the circuit model from a higher (or less detailed) abstraction level to lower (or more detailed) abstraction level is called synthesis. The logic synthesis performs the translation of the HDL design into the generic library of components followed by optimization and mapping into the gate-level components [32].

The subsequent step in the design flow is to synthesize a circuit description into a gate-level netlist that represents the logical functionality of the circuit (i.e., implementation). As shown in Figure 1.1, the design can be implemented based on different design methodologies either full custom or semi-custom. In the full custom design methodology, the designer creates the layout of the circuit and the interconnections between the circuit components in order to maximize the performance of the circuit, and minimize its area. On the other hand, in the semi-custom design methodology, the implementation is performed using standard cell libraries or gate arrays through

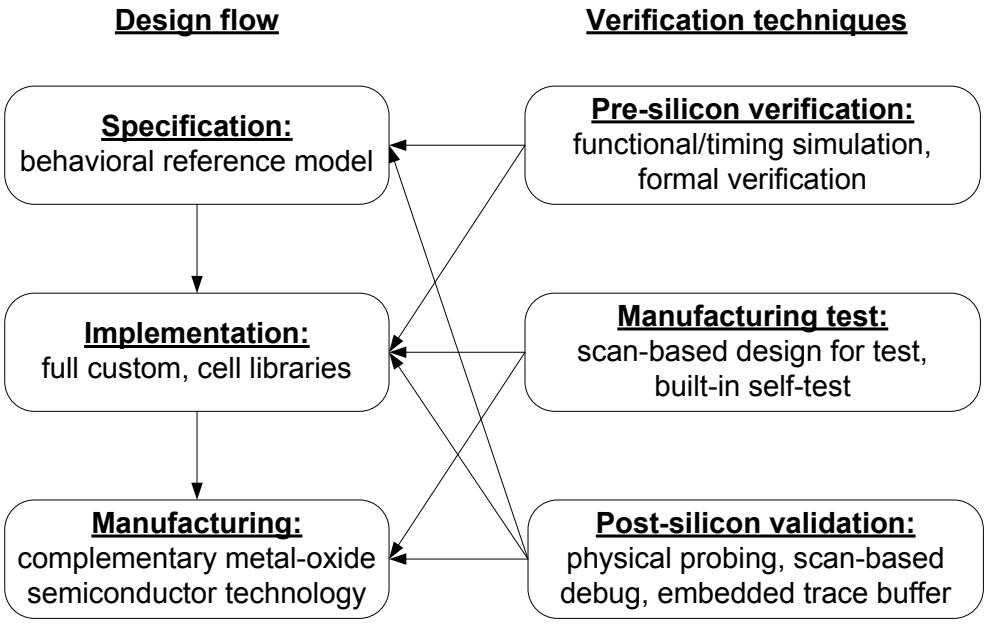


Figure 1.1: Design Flow and Verification Techniques of VLSI Circuits

computer-aided design (CAD) tools. These tools transform the design from the RTL abstraction level into the gate level structural netlist and perform the translation from the gate level netlist into a physical layout [18].

The final step in the design flow of VLSI is the process of creating the integrated circuit (i.e., manufacturing). The fabrication process is a multiple-step sequence of photographic and chemical processing steps during which the integrated circuits are gradually created on a wafer made of semiconducting material based on the specific transistor technology (e.g., 90 nm) [34]. Silicon is the most commonly used semiconductor material in VLSI circuits and complementary metal-oxide semiconductor (CMOS) is the main type of transistors. The increasing complexity of VLSI design makes pre-silicon verification, manufacturing test and the validation of the first silicon a bottleneck in the design flow.

1.1.1 Pre-Silicon Verification

As the complexity of SoC designs increases, the shrinking time-to-market makes design verification one of the major challenges in the implementation flow. As a consequence, pre-silicon verification techniques such as formal verification and simulation are used extensively to identify design errors before the design is manufactured [70]. The purpose of these pre-silicon verification techniques is to verify the implemented design against its specification.

Simulation techniques are used to verify the circuit behavior at various levels of abstraction. For example, the design can be simulated at a higher behavioral level that does not contain detailed timing information. The more detailed information the circuit model has (e.g., logic-level netlist), the more time will be needed to verify its functional correctness. Thus, exhaustive simulation for complex VLSI designs is practically infeasible. Formal verification, as a complementary technique to simulation, is the process of mathematically proving or disproving the correctness of the design with respect to a certain formal specification or property [62]. There are two main approaches to formal verification: equivalence checking and property (also known as model) checking. The purpose of equivalence checking is to ensure that two given designs are functionally identical by comparing one design called the reference model with the targeted design. These designs may be given on different levels of abstraction, i.e., register transfer level or gate level [35]. Property (or model) checking is used to check a given design for the satisfaction of its properties which are formulated in a dedicated verification language. Model checking is a commonly used approach in formal verification of finite state concurrent systems, where exhaustive exploration of all states and transitions in the circuit model can be performed through efficient techniques that reduce computing time [28].

Formal verification has proven its significance in identifying the logic bugs during pre-silicon verification of the Intel processor [16]. Nevertheless, due to the inherent trade-off between state coverage and verification time, as well as the limitations in modeling all the physical characteristics of the design, pre-silicon verification techniques have become insufficient to guarantee the first-silicon to be error-free.

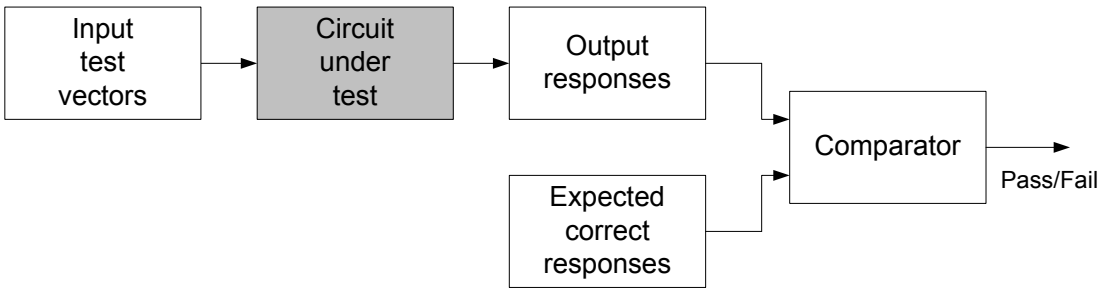


Figure 1.2: Basic Principle of VLSI Circuits Testing

1.1.2 Manufacturing Test

Manufacturing test has become an essential step in the implementation flow of the VLSI circuits to ensure that the functionality of the manufactured circuits matches its implementation [24]. It is used to identify the physical defects that lead to faulty behaviors in the fabricated circuits. Figure 1.2 shows the basic principle of manufacturing test. The entire test process is typically controlled by a powerful test instrument called automatic test equipment (ATE). Circuit under test (CUT) is the entire chip or part of the chip to which the input test vectors are applied. The input test vectors are binary patterns applied to the input of the circuit. These test vectors are obtained using automatic test pattern generation (ATPG) algorithms [24]. The circuit is identified as a fault-free circuit if the observed output responses match the expected correct responses. If the circuit failed the test, the fault diagnosis process will be started to diagnose and identify the root cause of the failure.

The circuit under test can be tested using functional or structural test. Functional test requires a complete set of the test patterns to verify every entry of the truth table (e.g., for a circuit with n inputs, the number of input test vectors will be 2^n). For example, for exhaustively testing a 64-bit ripple-carry adder, 2^{129} input test vectors are needed. If the ATE operates at 1 GHz, it would take 2.158×10^{22} years to apply the complete test set [24]. Therefore, complete functional testing leads to extremely long testing time, which makes it practically infeasible for testing complex integrated circuits. On the other hand, structural testing depends on the netlist structure of the

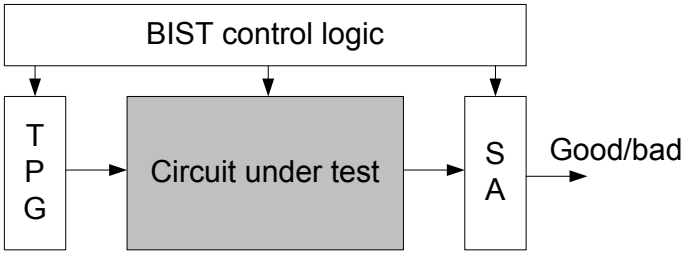


Figure 1.3: Basic Principle of BIST

design. In this technique, the defects are usually modeled at a certain level of design abstraction for the purpose of efficient testing (e.g., gate level or transistor level) [24]. Some typical fault models are stuck-at fault model, open fault model and path delay fault model. The most commonly used fault model is the single stuck-at fault model where a single node in the structural netlist of the logic network is assumed to have a fixed logic value and hence it is stuck-at either 0 or 1. When using the stuck-at model for the 64-bit ripple-carry adder, only 1728 stuck-at faults would need to be excited with 1728 input test patterns [24].

Achieving high fault coverage during manufacturing test is obtained through the inclusion of design for testability (DFT) circuitry that enhances the controllability and observability of the circuit's internal nodes. Scan based DFT is the most commonly used DFT methodology, in which all or part of the sequential elements (i.e., flip flops) of the circuit are replaced with scanned flip-flops (SFFs). In the test mode, these SSFs form one or more long shift registers called scan chains. The input test patterns are applied serially to the input of these scan chains which are connected to the primary inputs of CUT. The states of these SFFs are observed by shifting out the contents of the shift registers whose outputs are connected to the primary outputs of CUT [24].

Another DFT method is the built-in self-test (BIST). BIST can be used as a low cost testing approach when compared to ATE-based testing. In ATE-based testing, test stimuli are applied to the chip pins from the ATE and test responses are shifted out and compared with the correct responses stored in the ATE memory. Due to the large memory and bandwidth requirements for testing state-of-the-art SoCs, BIST

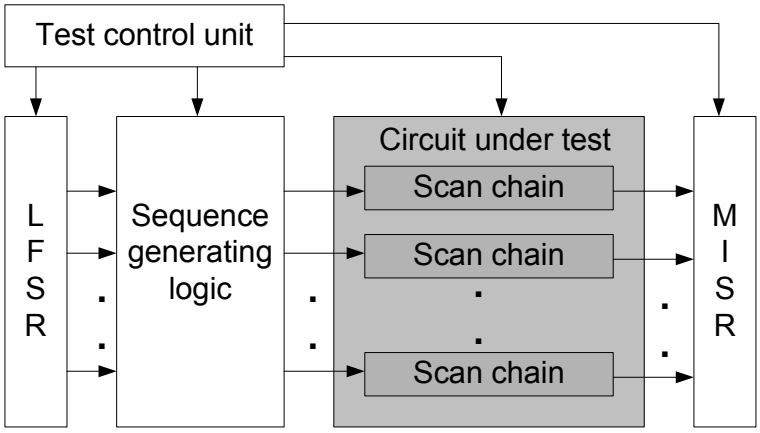


Figure 1.4: Deterministic BIST with Multiple Scan Chains Architecture

can be used as an alternative approach to external test for complex VLSI circuits. As shown in Figure 1.3, a test pattern generator (TPG) generates a set of test stimuli and a signature analyzer (SA) analyzes the test responses and makes the decision of good/bad chip. The BIST control block controls the test sequence. In BIST approach, a linear feedback shift register (LFSR) can be used as the TPG, and a single input signature analyzer (SISR) or a multiple input signature analyzer (MISR) can be used as the SA [13]. BIST methods can be classified into pseudo-random BIST and deterministic BIST. Applying only pseudo random test stimuli cannot guarantee sufficiently high fault coverage because the majority of the circuits are not completely random testable. Thus, the pseudo-random test-patterns can be extended with deterministic test patterns using more sophisticated pattern generators in order to achieve high fault coverage [26, 105, 120]. The pseudo-random BIST can be combined with deterministic BIST methodology to generate the scan inputs of the circuit under test for multiple scan chains architecture.

Figure 1.4 shows scan-based BIST architecture that supports deterministic BIST pattern generation. The test pattern generator consists of an LFSR and sequence generating logic (SGL). The purpose of SGL is to modify the pseudo-random patterns generated from the LFSR into deterministic ones by changing certain bit positions [63, 64]. The deterministic patterns are then applied to the CUT via scan chains.

The outputs of the scan chains are compacted into a signature stored in a multiple input shift register (MISR) to be compared with the correct expected signature.

By cost-effectively inspecting whether the implemented design is identical to the fabricated one, manufacturing test has become a key enabling technology that accelerates yield ramp-up and guarantees product quality. Nonetheless, manufacturing test is concerned only with fabrication defects and it relies on the circuit netlist as a golden reference. Therefore, to make sure the product does not contain any design bugs (which are present in the netlist and hence affect every fabricated device), manufacturing test must be complemented by post-silicon validation (or silicon debug) in order to identify the design errors that have not been caught by pre-silicon verification.

1.1.3 Post-Silicon Validation

The purpose of post-silicon validation is to identify and fix design bugs that are present in the fabricated circuit. These bugs can be classified into functional (logic) bugs where the circuit does not correctly match its specified function, or electrical bugs where the circuit does not meet the correct behavior over the entire operating region of its voltage, temperature or frequency [57]. The limited observability of the internal circuit signals is the major concern during post-silicon validation. To address this problem, physical probing techniques have emerged to improve the ability to probe internal circuit nodes and hence find the root cause of the electrical bugs [75, 97]. Probing techniques such as time resolved photoemission [33] and laser voltage probing [85] provide essential debug information that aids in the localization of the electrical bugs. Given the importance of physical probing techniques and the increasing complexity of SoC designs, a localization step, which we denote as logic probing, that compares the simulation data with the information observed in silicon, is required to identify a subset of circuit nodes that need to be physically probed [107]. Consequently, logic probing techniques have emerged as a complementary approach to physical techniques not only to aid in finding the electrical bugs but also to help in identifying the functional bugs that have escaped pre-silicon verification.

Reusing the scan chains, which are present in the circuit for manufacturing test, is one of the most commonly used logic probing techniques during post-silicon validation [30, 118]. In this scan-based debug approach, the circuit response is captured after the occurrence of a specific trigger event and, subsequently, the captured response is transferred to the debug software (this process is called scan dump). The debug software communicates with the circuit under debug (CUD) either through scan channels when debugging on the automatic test equipment (ATE), or through a low bandwidth interface (e.g., Boundary Scan [53]) when debugging on the target application board. The captured circuit response can be analyzed off-chip using post-processing algorithms, such as latch divergence analysis [29] or failure propagation tracing [25], to identify the first failing state elements.

The limitation of scan-based debug lies in the fact that halting the execution during scan dump may destroy the system's state. Thus, capturing debug data in consecutive clock cycles can not be achieved by using only the available scan chains. Although the use of shadow scan latches, (which may incur a substantial area penalty [58]), can avoid the destructive nature of the scan dump, a new capture cannot occur until the scan dump has been completed. Because capturing debug data in consecutive clock cycles is essential for identifying the timing related bugs, complementary techniques to scan-based method have been developed to provide real-time observability of a limited set of internal signals during post-silicon validation.

The debug data of a subset of internal signals can be acquired off-chip in real-time through dedicated chip pins [112]. The limitation of this debug technique for the state-of-the-art SoCs lies not only in the difficulty to drive high internal clock frequencies onto external chip pins but also in the limited pin counts allocated for debug. Consequently, *embedded logic analysis* methods that rely on on-chip trace buffers have emerged as a complementary approach to scan-based methods to enable real-time and at-speed observation of a limited set of internal signals [71]. The captured data is subsequently transferred, via a low bandwidth interface, from the internal debug module to the external debug software for post-processing. Trace buffer-based techniques have been recently used for debugging microprocessors [38, 46, 86, 95, 99], SoC designs [48, 71], and designs mapped to field programmable gate arrays [5, 100, 122].

1.2 Dissertation Motivation

The focus of this dissertation is to develop design-for-debug (DFD) architectures and methods that facilitate the identification of the functional bugs, when the validation is performed at-speed and in-system, i.e., on prototype application board. Although they have been validated for functional bugs, the proposed architectures and methods can be used also as a front end to physical probing techniques to assist in electrical bugs validation. As explained next, they are motivated by the limited real-time observability, as well as by the difficulty of dealing with blocking bugs during post-silicon validation.

The amount of debug data that can be captured into an on-chip trace buffer during embedded logic analysis is limited by the trace buffer *width*, which constrains the number of signals to be probed, and its *depth*, which limits the number of samples to be stored. To extract as much data as possible from a given debug experiment, i.e., to increase its *observation window*, without increasing the area of the on-chip trace buffer, *compression* can be employed. The existing compression methods can be categorized into special-purpose methods for microprocessors [46, 52, 60, 99] and generic methods for custom SoC designs [6, 7, 48, 67, 123]. The special-purpose methods rely on trace reduction or on lossless compression techniques. Trace reduction techniques are based on sampling (or filtering out) data at specific intervals, using programmable start/stop flags or by recording only program counter changes or branch addresses. In the lossless compression techniques, the compression can be accomplished by eliminating the redundant temporal information on data or address busses (e.g., through differential compression) [46].

For custom SoC designs, the compression techniques are currently emerging. For example, the compression can be achieved by analyzing the design and identifying a subset of essential signals which, after being captured on-chip, would then be expanded in the debug software using the knowledge of the design data [48, 67]. We refer to this method as a *width compression technique* where the length of the observation window remains the same and compression is achieved by re-constructing the values of more signals from the ones that are captured in the trace buffer. To

capture as many samples as possible while running a long debug experiment, one can employ also *depth compression*. It is important to note that the two main directions for compressing debug data for custom SoC designs (width and depth compression) are orthogonal to each other. In this dissertation we focus only on depth compression, which to the best of our knowledge has not been covered in the public domain (other than the prior works by the authors [6, 7]). Moreover, a new debug technique has been recently proposed in [123], as a follow up to our research work presented in [6]. The key differences between the debug technique introduced in [123] and our work will be discussed in Chapter 4.

Because of the presence of *blocking bugs* in one erroneous module inhibits the search for bugs in other parts of the chip that process data received from the erroneous module, it is important to bypass its erroneous behavior. In this dissertation, we address this problem by developing an embedded debug architecture that enables a hierarchical event detection mechanism in order to provide stimuli from an on-chip trace buffer, for the purpose of replacing the erroneous behavior caused by the blocking bugs [8].

1.3 Dissertation Organization and Contributions

The rest of this dissertation is organized as follows. Chapter 2 provides the background and a review of related work in post-silicon validation.

Chapter 3 introduces a novel debug architecture for embedded logic analysis that enables real-time lossless data compression. The proposed debug technique extends the depth of the observation window of a debug experiment. To quantify the performance gain from using lossless compression in embedded logic analysis, we present a new compression ratio metric that measures the trade-off between the area overhead and the increase in the observation window. We use this metric to quantify the performance gain of three dictionary-based lossless compression algorithms. The proposed architecture is based on one pass scheme algorithms which do not require re-running the debug experiment. Thus, the proposed architecture is particularly useful for in-field debugging on application boards which have non-deterministic input

sources, such as asynchronous interfaces.

Chapter 4 presents a novel architecture for at-speed silicon debug based on lossy compression. In order to accelerate the identification of the design errors, we have developed a new debug method where the designer can iteratively zoom only in the intervals that contain erroneous samples. By extending the silicon debug observation window using a short sequence of debug sessions, the proposed approach is useful in aiding the identification of hard-to-detect functional bugs that occur intermittently over a long period of time [57], which is computationally-infeasible to be simulated during pre-silicon verification. When compared to increasing the size of the trace buffer, the proposed architecture has a small impact on silicon area, while significantly reducing the number of debug sessions. The proposed method is applicable to both automatic test equipment-based debugging and in-field debugging on application boards, so long as the debug experiment can be reproduced synchronously.

Chapter 5 introduces a novel embedded debug module architecture for bypassing blocking bugs that occur intermittently over a long execution time. This architecture facilitates the validation of the other parts of the chip that process data received from the erroneous module. The proposed approach enables a hierarchical event detection mechanism to provide correct stimuli from an embedded trace buffer, in order to replace the erroneous samples caused by the blocking bugs.

Chapter 6 summarizes the contributions of this research and provides directions for future work.

Chapter 2

Background and Related Work

This chapter provides the background and a review of related work on post-silicon validation. We first introduce an overview of post-silicon validation in Section 2.1. The scan-based debug technique is described in Section 2.2. The basic principle of embedded logic analysis and the trace buffer-based debug techniques are introduced in Section 2.3. The recent advancements in the design for debug techniques for improving the observability of the internal circuit's nodes are described in Section 2.4. Finally, Section 2.5 concludes this chapter.

2.1 Overview of Post-silicon Validation

As outlined in the previous chapter, due to the growing complexity of system-on-chip (SoC) designs and the ever increasing demand of time-to-market, pre-silicon verification techniques such as simulation and formal verification [70] have become insufficient to guarantee the first-silicon to be error-free. Furthermore, many electrical bugs (e.g., leakage, charge sharing, noise coupling between interconnect lines or drive fights) cannot be screened without the existence of the fabricated IC [57]. Given the escalating mask costs, it is imperative that the design bugs are identified and fixed as soon as the first silicon is available. As a consequence, structured *post-silicon validation* techniques have emerged to reduce the time required to find design bugs, the number of silicon spins, and hence the time-to-market [112].

The design bugs that can be identified during post-silicon validation can be classified as either functional bugs or electrical bugs, as described in Chapter 1. These design bugs types are analyzed and debugged differently according their validation plans [58]. For *functional validation*, validation patterns are applied to circuit under debug (CUD) in order to ensure its functional correctness. For *electrical validation*, validation patterns are used to ensure functional correctness of the CUD across the entire operation region of voltage, temperature and frequency. A characterization window is used to specify the range of voltage and frequency across which the circuit is guaranteed to work [58]. In order to identify the electrical bugs, a two dimensional plot (also referred to as shmoo plot) is used to show how the circuit performs against the variation of the voltage and the frequency [11]. It should be noted that the debug process which targets different types of bugs depends on the underlying debug environment and the design-for-debug (DFD) features implemented in the CUD to facilitate the identification of design bugs.

2.1.1 Debug Environments

There are two main types of environments for debugging prototype silicon: digital IC testers (also referred to as ATE) and prototype application boards. Digital IC testers are used in manufacturing test to screen the circuit for manufacturing defects. The importance of the tester environments lies in the capability to apply the input stimuli and compare the responses on the circuit I/O pins against the expected responses. Part of electrical validation is performed on the tester because of the ability to precisely control the voltage and frequency. Due to the limitation of tester storage and the necessity to cover the functional correctness of the CUD, ATEs may not be sufficient for debug, e.g., to replay several minutes of digital video [112]. Therefore, in-system debugging (on application boards) is usually used as a complementary approach to ATE-based debug. For in-system debug approach, no extensive stimuli and responses storage are needed as in the case of ATE-based debug but the controllability and observability are less straightforward. The debug engineer can chose between the debug environments based on the validation objectives.

2.1.2 Post-silicon Validation Techniques

The main challenge in post-silicon validation is the limited observability of the internal circuit's signals. To address this problem, physical probing techniques have emerged to improve the ability to probe internal circuit's nodes and hence find the root cause of the electrical bugs [33, 75, 85, 97]. However, despite the recent advancements in the physical probing techniques, the complexity of state-of-the-art devices requires a localization step to precede the destructive IC failure analysis. This step, which we denote as *logic probing*, correlates the simulation data to what is observed in the silicon (either on the (I/Os) or on the internal signals) in order to identify a subset of circuit nodes that need to be physically probed [107]. This is achieved by adding DFD hardware for the purpose of improving the internal observability and accelerating the debug process. The DFD techniques are not only useful in finding the electrical bugs but they are also essential for locating the functional bugs.

In order to identify the design bugs during post-silicon validation, DFD infrastructure is employed to enable capturing and accessing the internal state of the CUD. Two DFD complementary techniques have been used to provide the observability of the system's state: scan chain debug technique based on run-control debug approach, and embedded logic analysis based on real-time trace approach. These debug techniques are essential to narrow down the search for design bugs by pinpointing both temporally (when) and spatially (where) a bug occurs.

2.2 Scan-based Observability

The scan methodology is the most commonly used technique in manufacturing test for the purpose of allowing the observation of internal circuit's full state. Scan chains have been used extensively for debugging complex digital ICs [12, 14, 30, 44, 59, 69, 72, 89, 108, 111, 115, 117, 118]. To distinguish between the usage of scan chains in manufacturing test and post-silicon validation, we first illustrate their functionality during the manufacturing test process.

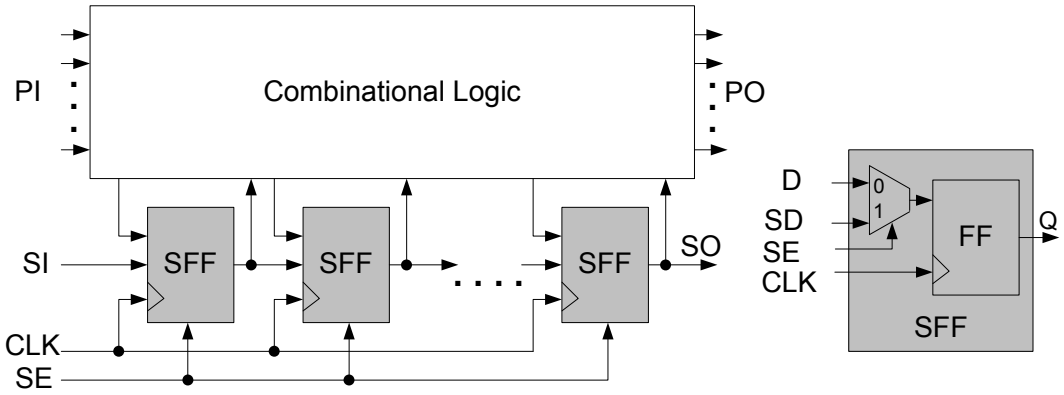


Figure 2.1: A Single Scan Chain Architecture

2.2.1 Scan-based DFT Methodology

When employing scan chains in scan-based DFT, the sequential elements (i.e., flip-flops, latches) are connected in such a manner that allows two modes of operation. In the test mode, the scanned flip-flops are reconfigured as one or more shift registers (known as scan chains). The input test vectors are serially loaded to the CUT. Once a vector is loaded, the CUT is allowed to work in its normal mode where the circuit response is captured in the flip-flops. Thereafter, the captured response is shifted out and compared versus the expected response [24].

Figure 2.1 illustrates the architecture of a single scan chain. Each scanned flip-flop (SFF) is composed of a D flip-flop and a 2-to-1 multiplexer. A scan enable signal SE selects between two data inputs: the original data input D, and the scan-data input SD. The primary inputs/outputs (PI/PO) are connected directly to the combinational logic. The scan chain is constructed by connecting the data output Q of one SFF to the SD signal of the following SFF, as illustrated in the Figure 2.1. The scan-based DFT procedure during the test process can be explained as follows: (i) assert scan enable signal (i.e., SE is set to 1), and load the test vector through the scan input (SI); (ii) de-assert scan enable signal (i.e., SE is set to 0), and apply one clock cycle in order to capture the circuit response from the combinational logic outputs in SFFs; and (iii) re-assert scan enable signal (i.e., SE is set to 1), and scan out the test

responses. While the test response is shifted out, a new test vector is simultaneously shifted in. The length of the scan chain and the number of test patterns are the two contributing factors that influence the test time. To reduce the test application time, multiple scan chains can be used. Each scan chain usually has its dedicated SI and SO pins and hence the test data/response can be shifted in/out in parallel. It is important to note that most blocks of a SoC are designed with full-scan capability, where all internal sequential elements are replaced by SFFs, in order to achieve high fault coverage [24].

2.2.2 Scan-based Debug Methodology

In the scan-based debug approach, the circuit response is captured after the occurrence of a specific trigger event and subsequently, the captured response is transferred to the debug software (this process is called scan dump). The debug software communicates with the circuit under debug either through scan channels when debugging on the automatic test equipment (ATE), or through a low bandwidth interface (e.g., boundary scan IEEE 1149.1 standard interface [53]) when debugging on an application board. To control the debug process, DFD hardware (i.e., debug module) is integrated with scan chains to enable start, stop, resume or single-step the program execution [116]. This approach obviously provides high observability and controllability of the circuit behavior and hence facilitates the identification of the error-capturing scan cells. For example, scan-based techniques rely on stopping the functional test program when a failure is observed on an output and subsequently, by re-running the debug experiment with different trigger points, it can scan dump the state before and after the observed failure point. Thereafter, post-processing algorithms, such as latch divergence analysis [29] or failure propagation tracing [25], can be used to analyze the scan dumps and identify the first failing state elements.

To support scan-based debug, two essential features have to be enabled by the DFD infrastructure: breakpoint mechanism (i.e., trigger conditions detector) and execution control. Breakpoints (or trigger conditions) determine the points in time at which the system execution is stopped. Once the circuit is stopped, the internal

state is transferred to debug software to be compared with the obtained correct state from simulation. The breakpoint circuitry can be reprogrammed to specify a different trigger condition to get an insight in the actual behavior of the CUD. The process of stopping the circuit at a certain time and observing its internal state can be repeated in a sequence of debug experiments (or sessions) until the bug is identified.

The operations, that control the execution of the CUD during a debug session, can be explained as follows [112]:

- Reset: to reset the chip and its environment to a known state;
- Run: to start the application;
- Stop: to stop the on-chip clocks once the trigger condition occurred;
- Scan out: to dump the content of the scan chains;
- Single step: to step through the execution of the chip, one clock cycle at a time; and
- Continue: to resume operation after execution has previously been stopped.

These functions require an on-chip clock reset circuitry and a clock controller that provides stop, scan, continue, and single-step operations.

The controlled execution of the circuits's internal functions is similar to the process of software debug. During software debug, the debug engineer iteratively sets custom breakpoints to stop the program and then examines the state of the application. Various breakpoints configurations and single step operations can be performed iteratively until the bugs are identified. During post-silicon validation, an on-chip breakpoint module is programmed to stop the CUD at a specific point in time. Thereafter, the contents of all flip-flops (i.e., scan chains) can be transferred off-chip via a low bandwidth interface to be analyzed.

Example of a Breakpoint Debug Module

Figure 2.2 shows an example of a breakpoint module, which is connected to several trigger signals from an embedded core. The selection of targeted trigger signals, that

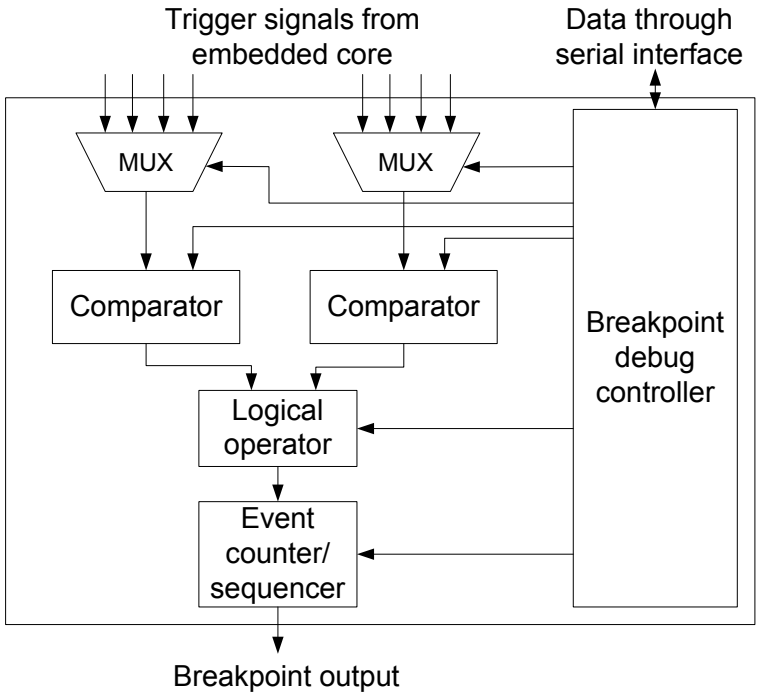


Figure 2.2: Example of Breakpoint Debug Module

debug engineer wants to trigger on, is uploaded to the breakpoint debug controller through the serial interface, e.g., boundary scan IEEE 1149.1 standard interface [53]. The capability of the breakpoint module for detecting a certain trigger condition is essential in the identification of design bugs. Thus, the breakpoint debug module contains comparators which are used to compare the selected trigger signals against preprogrammed breakpoint registers existing in the breakpoint debug controller. The targeted comparison operations (e.g., less than, greater than or equal) are selected based on the control information uploaded in the breakpoint controller.

As shown in Figure 2.2, two trigger signals can be selected using two multiplexes (MUXs) and these signals can be compared against the loaded values of the two comparators. A logical operation can be performed on the output of the two comparators such as OR, XOR, or AND operation. The output of the logical operator is connected to an event counter/sequencer. The event counter is used to determine

a certain event that occurred a pre-defined number of times. The event sequencer is used to determine multiple different events that occurred in a pre-defined order. The debug configuration that specifies the breakpoint condition can be uploaded to the breakpoint debug controller at run-time through the serial interface. When the targeted trigger condition occurs, the output of the breakpoint debug module triggers the clock controller that provides stop, scan, and single-step functions. It is important to note that the breakpoint module can contain more levels of logic than the ones shown in Figure 2.2. For example, bitwise operators can be inserted to precede the comparison operators to perform bitwise operation between a selected trigger signal and a pre-defined constant value uploaded in the breakpoint controller.

To facilitate the inclusion of breakpoint modules during the design stage of the chip, a novel approach has been proposed to automatically generate breakpoints modules using breakpoint description language (BDL) [116]. BDL allows the designer to write the description that captures the specification of the breakpoint module. A BDL compiler tool then reads this description and generates the hardware implementation of the corresponding breakpoint module. However, the area of the generated breakpoints using BDL can be larger than their custom implementations. This area increase is because the generated breakpoints do not share common resources (i.e., comparators) [116].

Scan-based Debug Architecture

Figure 2.3 illustrates the scan-based architecture for post-silicon validation [93]. Note that the surrounding combinatorial logic, associated with each scan chain, is not shown in this figure. There are two modes of operation: the test (or debug) mode on ATE and the debug mode on prototype target application board. In the test mode, the concat signal is set to 0 (i.e., $Concat = 0$) and the scan is configured for manufacturing test on ATE, where the scan chains are accessed in parallel through the functional pins. As mentioned earlier, part of electrical validation is performed on the ATE because of the ability to precisely control the voltage and frequency. In the debug mode where the CUD is debugged in-system, once the breakpoint debug module detects the trigger event, the clock controller stops all on-chip clocks. Thereafter, the

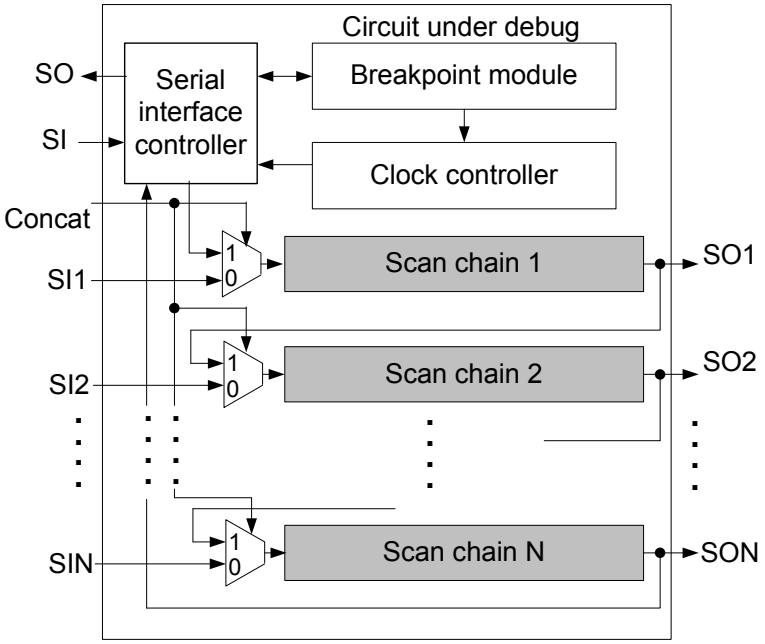


Figure 2.3: Scan-based Debug Architecture [93]

concat signal is set to 1 (i.e., $Concat = 1$) and hence the multiplexers concatenate all scan chains into one long shift register. The content of the scan chain is then shifted out via the SO pin through the serial interface (e.g., JTAG [53]). The received data can be applied to the chip via the SI pin during shifting out scan chains data. This procedure is repeated until all internal scan chains have been scanned out. After comparing the circuit's complete state against the expected one, the debug engineer can reprogram the breakpoint debug module to specify another trigger condition in order to capture and analyze the chip's state at another particular event. Upon observing a mismatch, the engineer tries iteratively to zoom in on the time and location of the error's occurrence by setting another trigger condition and analyzing the captured data. This process is continued until the root cause of bugs is identified. Then, the bug is fixed in the original design and new silicon is manufactured. Figure 2.4 shows the iterative debug flow during post-silicon validation as described above. It is important to note that stopping a system with multiple clock domains may lead

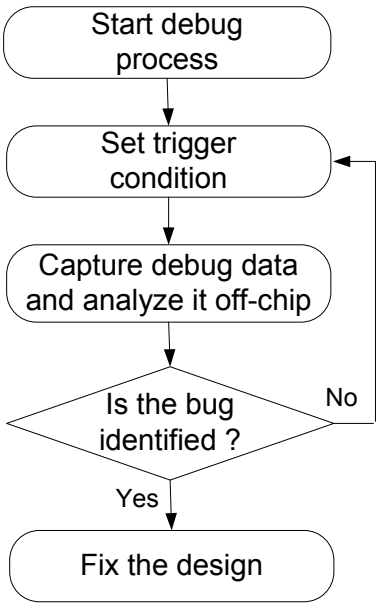


Figure 2.4: Debug Flow during Post-silicon Validation

to data invalidation where some clock domains might use data from clock domains that have already stopped because the on-chip clocks do not stop simultaneously [41]. A novel data invalidation detector has been proposed in [41] to identify the scan elements (i.e., flip-flops) that capture invalid data, and hence the content of these elements will not be used for the comparison with the simulation model.

The clock controller can enable other control functions such as single step and continue operations. The single step operation is implemented as an extension to the debug scan-out operation, where the circuit is allowed to work in the normal mode and the scan enable on the scannable flip-flops is not activated. The continue operation is achieved by deactivating the breakpoint mechanism that releases the gating on the clocks, so that chip operation can resume. It is important that the phases and frequencies of the on-chip clocks are the same as at the moment the chip was stopped; otherwise, the resumed operation will not be the same as in the case of a chip that has not been stopped [31].

Because halting the execution during scan dump may destroy the system's state, capturing debug data in consecutive clock cycles can not be achieved by using only

the available scan chains. As hard-to-detect functional bugs appear in circuit states which may be exercised billions of cycles apart [57], it is desirable to maintain the circuit operation during scan dumps. Although the use of shadow scan latches, which may incur a substantial area penalty [58], can avoid the destructive nature of the scan dump, a new capture cannot occur until the scan dump has been completed. Because capturing debug data in consecutive clock cycles is essential for finding timing-related bugs, new debug methods have been developed to provide real-time observability of a limited set of internal signals.

Real-time observation of a subset of internal signals off-chip, can be achieved by using dedicated chip pins to output the debug data to on-board acquisition buffers [112]. The limitation of this approach for state-of-the-art SoCs lies in the limited number of pins used for debug, as well as in the difficulty to drive internal data at high rates onto the board. As a consequence, *embedded logic analysis* methods that rely on on-chip trace buffers have emerged as a complementary approach to scan-based methods in order to enable real-time and at-speed sampling of a limited set of internal signals [71, 78].

2.3 Embedded Logic Analysis

Embedded logic analysis is a DFD technique used for improving the observability of internal signals of SoC design by acquiring debug data on-chip into embedded trace buffer. This is achieved through a DFD event detection mechanism that can be configured to a specific trigger event at which the acquisition process starts or stops.

2.3.1 Basic Principle of Embedded Logic Analysis

Figure 2.5 illustrates the basic principle of an embedded logic analysis debug framework. In this framework, the off-chip debug software communicates with the embedded debug module through a serial interface such as the Boundary Scan interface (i.e., JTAG [53]). The embedded debug module contains an embedded trace buffer to provide real-time observability for a subset of internal signals. As shown in Figure

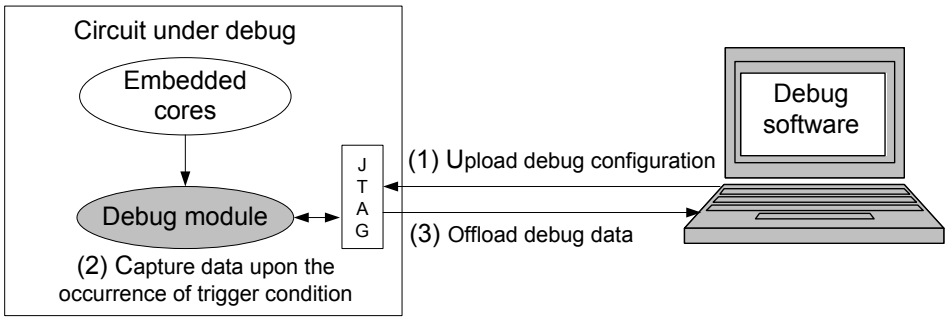


Figure 2.5: Debug Framework of Embedded Logic Analysis

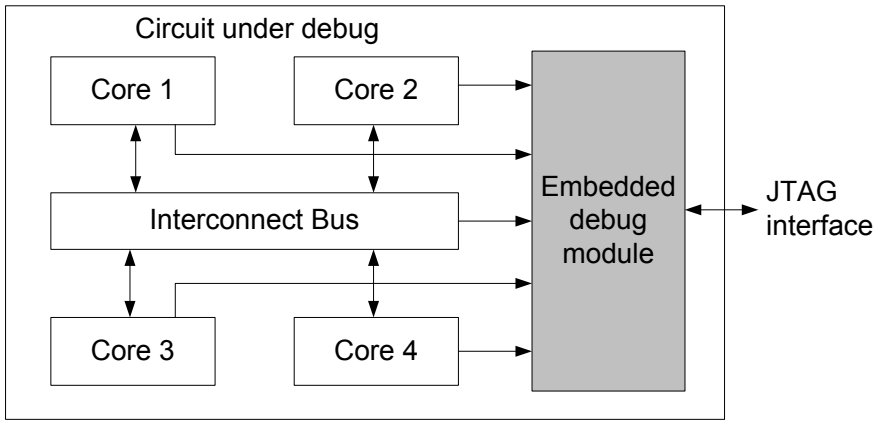


Figure 2.6: Circuit under Debug with Centralized Debug Module

2.6, a centralized embedded debug module monitors a limited set of internal signals coming from embedded cores and the interconnect bus of the CUD. The concept of a centralized debug module has been used in debugging multi-core processor design [91]. The debug flow during embedded logic analysis is explained as follows.

A debug experiment (or session) starts by uploading the embedded debug module with the debug configuration (Step (1)). This configuration contains debug control information such as the trigger condition that specifies the point in time at which the acquisition of debug data starts, and the control data that specifies the selection of nodes that need to be probed. Once the trigger condition occurs during the execution of the on-chip application, the debug module will start capturing the selected signals

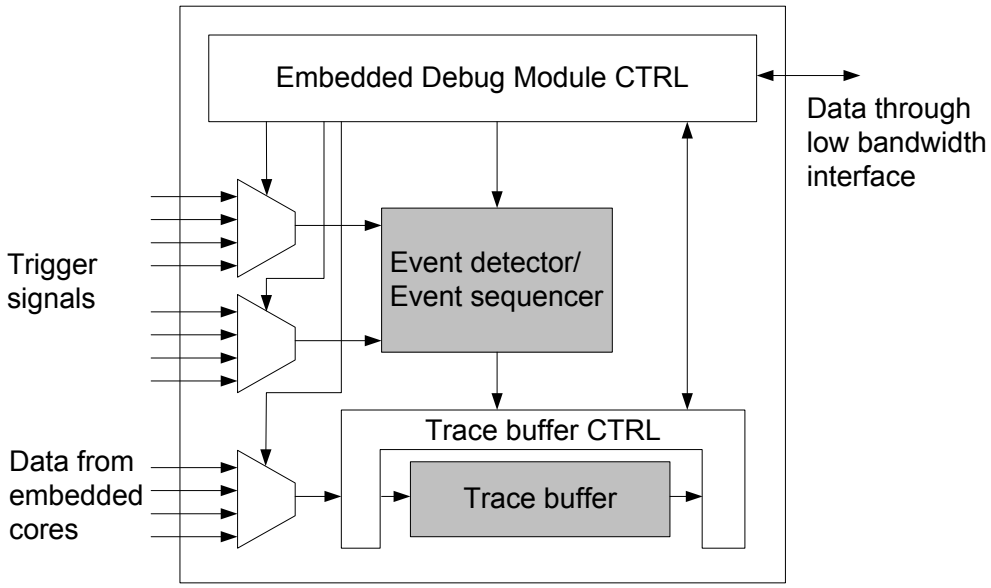


Figure 2.7: Embedded Debug Module Architecture

behavior into an on-chip trace buffer (Step (2)). The debug experiment is completed by transferring the trace buffer's content to debug software, where the captured data is analyzed (Step (3)). The debug experiment can be iteratively repeated with different debug configuration until the design bugs are identified.

2.3.2 Embedded Debug Module Architecture

The key feature of an embedded debug module is to capture the behavior of selected internal signals upon the occurrence of a certain triggering condition (i.e., trigger event). This is achieved using an event detector that monitors a group of trigger signals to determine when the debug data is captured in the trace buffer, as shown in Figure 2.7. The trigger condition can be performed based on bitwise, comparison or logical operations between a certain selected trigger signal and a specified constant value stored into a control register which exists in the embedded debug module control. To further enhance the detection ability of the debug module, the event detector is augmented by an event sequencer to monitor a specified sequence of events. The

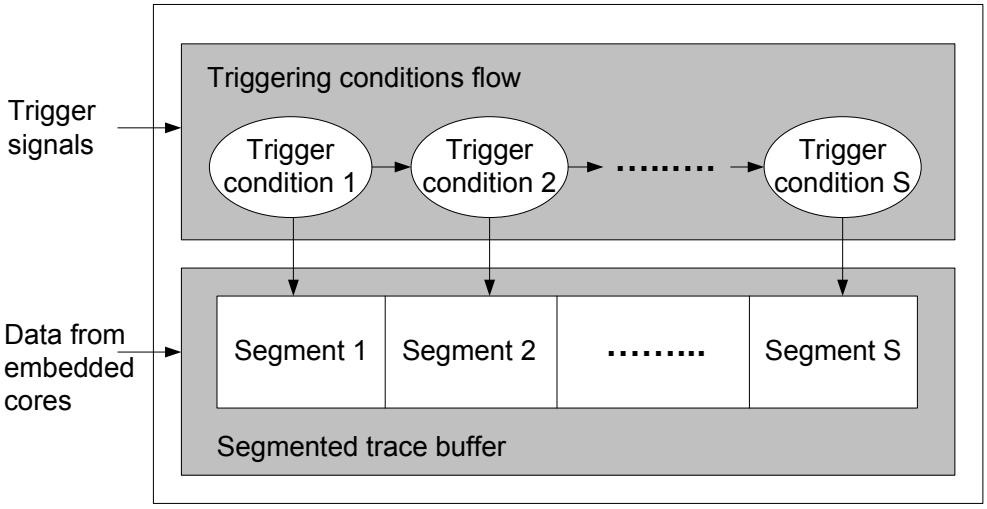


Figure 2.8: Triggering Conditions Flow with Segmented Trace Buffer

configuration of the trigger signal selection, the trigger conditions, and the choice of the signals that need to be probed are uploaded to the embedded debug module control through the low bandwidth interface (e.g., JTAG [53]). It is important to note that the event detector/seqencer functionality used in the embedded debug module is similar to the one used in the breakpoint module of scan-based debug architecture described in Section 2.2.2.

The embedded trace buffer can allow flexible acquisition by employing it as a segmented buffer. Figure 2.8 shows the segmented trace buffer with S segments. Each segment captures the data upon the occurrence of the corresponding triggering condition, e.g., Segment 1 starts the acquisition upon the occurrence of trigger condition 1. Once the first trigger condition occurred, the event detector circuit is configured to monitor the second trigger condition. The event detection circuit is sequentially updated with the control information that specifies the required triggering condition until the last triggering condition is reached. Each segment size can be configured to work as a circular buffer segment and thus in this case the triggering condition can be used to stop the data acquisition. The feature of a segmented trace buffer described above has been used in debugging FPGA designs [5].

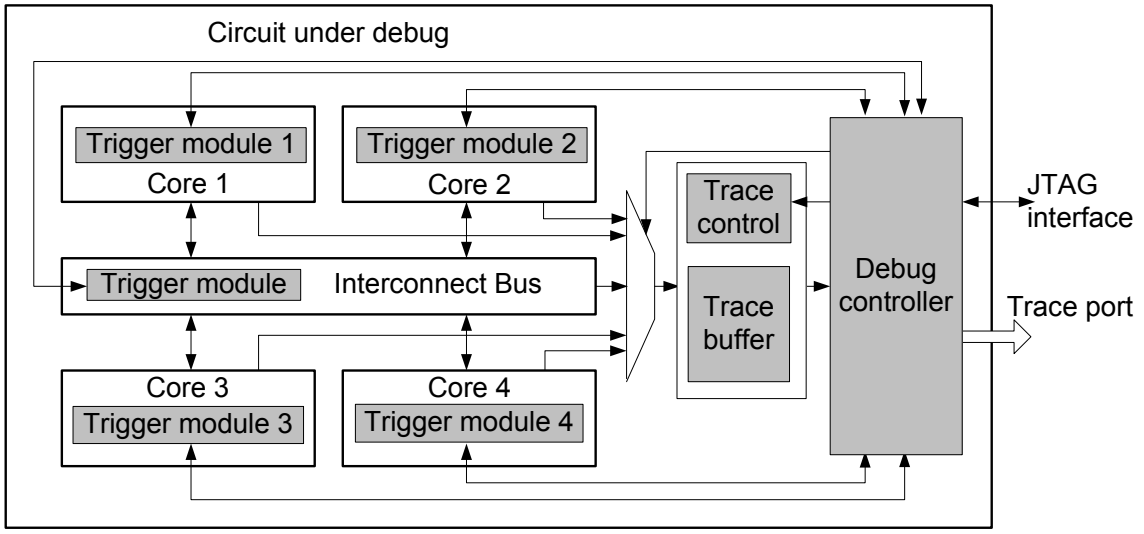


Figure 2.9: Multi-core Debug Architecture with Centralized Trace Buffer

2.3.3 Embedded Logic Analysis Related Work

While scan-based debug concepts have emerged from the manufacturing test research, trace buffer-based debug has been influenced by software debugging used in real-time embedded systems [76]. Real-time systems, centered around embedded processors or micro-controllers, have been traditionally debugged using in-circuit emulator (ICE) devices. ICEs are constructed using bond-out chips, which connect internal nodes to additional device I/Os in order to make them visible *off-chip* to external instruments. The limitation of using ICEs for state-of-the-art SoC devices lies not only in the increasing gap between the on-chip and off-chip frequencies, but also in a large footprint of the bound-out chips caused by the additional I/Os used only for debug. As a consequence, for SoC designs there has been a trend toward placing the instrumentation *on-chip*, thus enabling at-speed sampling through embedded logic analysis [5, 71, 100, 122].

The *trace buffer-based debug* methods can be broadly classified as: *special-purpose* (i.e.., specific to embedded processors) [47, 49, 77, 91, 95, 99] or *generic* (i.e., applicable to any type of custom SoCs) [1, 2, 48, 71, 90]. In order to concurrently monitor

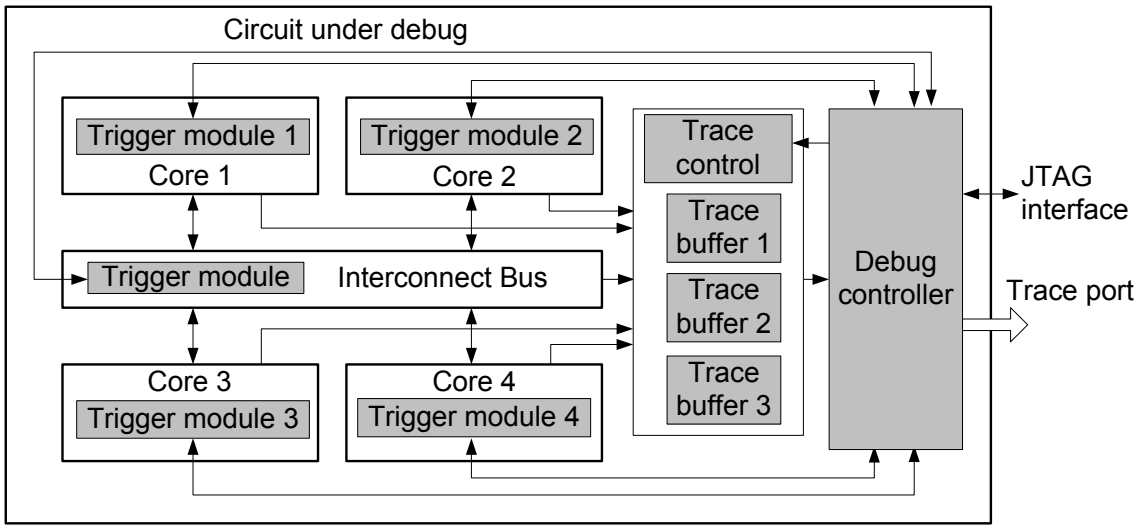


Figure 2.10: Multi-core Debug Architecture with Multiple Trace Buffers

signals behavior from different cores in complex SoCs, distributed trigger modules are allocated in multi-core architecture [46]. Moreover, two approaches can be used to support data tracing from embedded cores either: *centralized* or *distributed* trace. In the *centralized* trace approach, one trace buffer is employed for the entire SoC, where the trace data is selected through a multiplexer network [2]. Figure 2.9 shows the multi-core debug architecture with central trace buffer. The centralized trace buffer approach has been used in debugging the Cell processor which contains nine processing cores [91]. Due to the limited speed and bandwidth of the serial interface, high speed trace port can be used for streaming the data that is captured into the trace buffer [36, 46]. Thus, the trace port can facilitate the debug process by offloading the trace buffer contents while running the on-chip application and hence allowing more data to be captured on chip. Nonetheless, due to the growing complexity of SoCs and the increasing number of embedded cores [109], the capacity and bandwidth of a single trace buffer used in the *centralized* trace approach limit the observability when signals from different cores need to be concurrently acquired. To address this problem, *distributed* trace buffers can be used to improve the real-time observability of the multi-core architecture. In *distributed* tracing approach, multiple trace buffers

are allocated to multiple cores, where the data can be acquired simultaneously from different cores during the debug process [66, 71]. In order to efficiently utilize the available on-chip trace buffers, they can be dynamically allocated based on the trigger conditions that may occur consecutively over a short interval from different cores [66]. For example, in Figure 2.10, three trace buffers can be shared in such a manner, data can be captured simultaneously from three different sources. e.g., three cores or two cores and the interconnect bus.

In summary, the event detection capability described previously and the embedded trace buffer are the two main essential features in embedded logic analysis. The event detection mechanism helps the debug engineer to specify the trigger condition which starts and/or stops the capturing process of the debug data into trace buffer. The capacity of the embedded trace buffer limits the amount of data that can be captured during a debug session and hence can lengthen the debug process. In order to enhance the event detection capability and extract more data than the one that is captured on-chip, a number of debug techniques have been proposed to address these issues.

2.4 Recent Advancements in DFD Techniques

As stressed earlier the *event detection mechanism* used in the DFD infrastructure of both scan-based debug methodology and embedded logic analysis, and the *capacity of on-chip trace buffer* employed in the embedded logic analysis are two essential features in identifying design bugs during post-silicon validation. In this section, we describe the debug techniques that have been recently developed to address these features and thus improve the observability during post-silicon validation. First, we describe the assertion-based debug technique which has been recently used as a detection mechanism for identifying the violation of design properties, and thus it facilitates the localization of the root cause of a faulty execution behavior in the circuit under debug. Second, we present a new detection mechanism based on transaction-level debug techniques for debugging designs that contain complex on-chip communication infrastructures. Third, we introduce the emerging techniques to address the limited capacity of the on-chip trace buffers used in embedded logic analysis.

2.4.1 Assertion-based Debug

Assertion-based verification (ABV) is widely used to shorten the pre-silicon verification process in the design flow [37]. An assertion is a statement that describes a design's intended behavior (also referred to as a property), which must be satisfied in an error-free design. For example, an assertion may state that a FIFO buffer overflow should never occur. i.e., If the data is written into a full FIFO buffer, the assertion will be fired to indicate the occurrence of a faulty behavior which caused the FIFO overflow. Assertions describe the intended circuit functionality using hardware verification language such as System Verilog [54] or Property Specification Language (PSL) [4, 55]. The following example illustrates how an assertion is written in PSL to express a certain property of a bus arbiter:

*assert always ({!req;req} |=> {req[*0:1];grant})*

In this example, the bus arbiter assertion states that the bus should give a bus grant within two clock cycles when the request signal goes from low to high and the request signal remains high until the grant is received. The |=> operator is a temporal implication (the preconditions and the post-conditions represent its left and right arguments respectively), the [*low:high] operator specifies a range of repetition, and the semicolon ";" represents temporal concatenation. Upon the occurrence of the preconditions ({!req;req}), if any of post-conditions ({req[*0:1];grant}) is not satisfied, the assertion will be asserted to indicate an error. The distinguished benefit of ABV methodology lies in improving the observability within the design by revealing any unexpected behavior caused by a bug that leads to an assertion violation (or failure). Thus, an assertion failure is used as a beginning step for the debugging process of identifying design errors and hence accelerates the error localization.

Assertions have been used extensively in formal verification and simulation techniques during pre-silicon verification [51, 70]. In formal verification, assertions are used to indicate if the design properties are proven correct. Nonetheless, the growing complexity of SoC designs makes proving the correctness of properties computationally impractical. On the other hand, assertions are used in simulation in such a manner that the actual circuit responses obtained during simulation can be checked

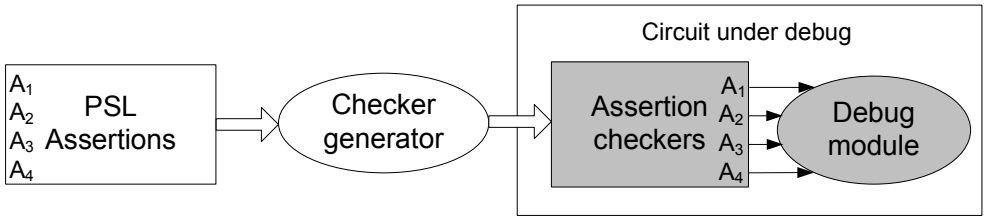
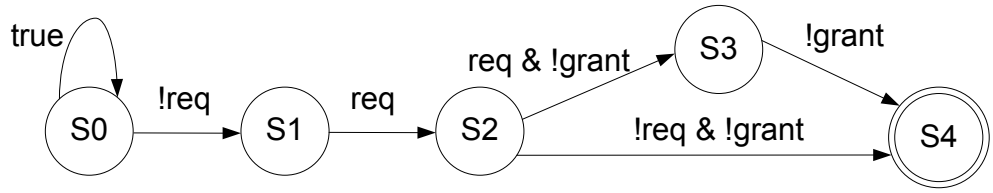


Figure 2.11: Basic Principle of Assertion-based Debug

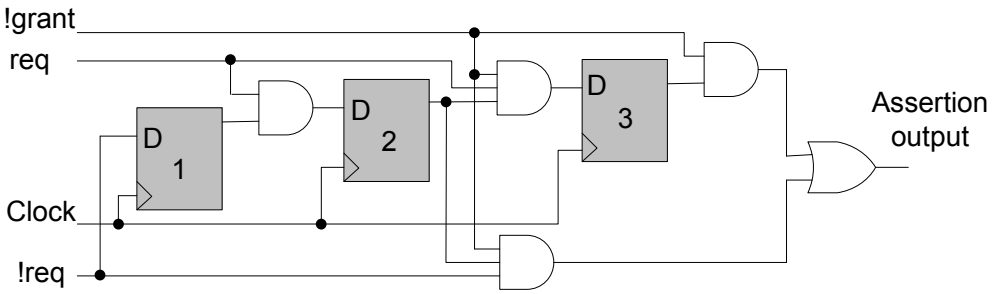
dynamically by the simulator against the specified properties. When any of the specified assertion rules is violated, a report containing information such as when and where the violation occurs is generated so that the debug engineer can study and fix the corresponding error. It is essential to note that the existence of no assertion failure is not an indication that the design is error-free. This indicates only that the design properties specified by the assertions are satisfied.

Synthesizing assertions (or design properties) into hardware checkers has recently become an effective technique to perform in-system post-silicon design validation [2, 19]. To enable assertion-based debug during post-silicon validation, a checker generator is used to produce hardware assertion checkers from assertion statements written using a hardware description language [20, 22]. As shown in Figure 2.11, the checker generator is used to generate the assertion checkers from the assertions which are specified in PSL. The outputs of the assertion checkers are monitored by the embedded debug module. The output of a certain checker will be asserted to indicate an assertion failure of the corresponding design property specified by the checker. The checker output can be used as a trigger signal to stop capturing the data into trace buffer, as explained later in this section. The advantage of using assertion checkers lies in facilitating the process of locating the root cause of observed errors by starting the analysis process of detected errors from the places that fire the assertions.

Figure 2.12 shows an illustrative example of the bus arbiter assertion described earlier: *assert always ({!req;req} |=> {req[*0:1];grant})*. The output of this checker circuit will be asserted when the request (req) signal goes high and the bus arbiter does not grant a bus grant within at most two clock cycles during which the request signal remains high. The assertion checker generator builds the circuit based on various



(a) Automaton for *assert always ({!req;req} |=> {req[*0:1];grant})* [20]



(b) Example of An Assertion Checker Circuit

Figure 2.12: Illustrative Example of Assertion

automata for PSL properties [22, 23]. As shown in Figure 2.12(a), an automaton is illustrated as a directed graph, where vertices represent the states, and the conditions for the transitions among the states are written on the edges. The checker generator transforms the automaton of the assertion into RTL code. Based on the assertion checkers described in [21, 23], the assertion checker can be implemented in hardware using combinational logic and flip-flops as shown in Figure 2.12(b). In this example, the state $S0$ is the initial state, which is activated upon a reset and the state $S4$ is the final state, which is reached upon an assertion violation. The output of an assertion checker can be connected to a flip-flop to enhance the frequency performance [20].

Assertion checkers can be dynamically configured into a dedicated reconfigurable logic [2, 19]. Thus, assertions can be integrated in embedded logic analysis to facilitate the debug process. For example, a firing assertion can be used as a trigger to stop capturing data into the trace buffer (i.e., the trace buffer is employed to work as a circular buffer during the debug experiment) and hence the captured data represents the behavior of the internal signals, which occurs over an interval that precedes the bug detected by the assertion. Thereafter, the captured data can be analyzed in order to identify the root cause of the fired assertion.

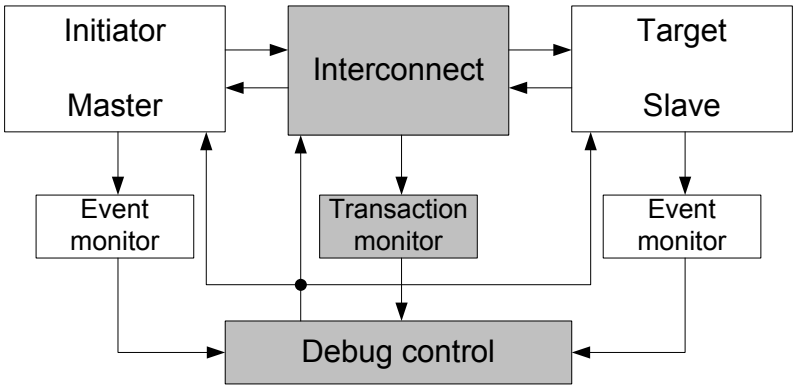


Figure 2.13: Transaction-Based Communication-Centric Debug [42]

2.4.2 Transaction-level Debug

The increasing complexity of the on-chip communication infrastructures of SoC designs has recently motivated the researchers to introduce communication-centric debug techniques. As a consequence, transaction-level debug techniques have been proposed to aid in speeding up the debug process of SoC designs that have a complex on-chip communication infrastructure [42] (e.g., a network-on-chip (NoC) [15]). A transaction can be defined as an exchange of a data or a sequence of related events between two components. e.g., request and acknowledge signals. These transaction events will cause data transfers on the on-chip communication infrastructure. The concept of transactions-level modeling (TLM) has been used, as a high level of abstraction, in pre-silicon verification to boost the performance of simulation [101]. In the TLM approach, the details of communication among design modules are separated from the details of the implementation of these modules. The communication mechanism at the transaction level can be described using a high-level modelling language (e.g., SystemC [43]). Transaction-level debug approach has been recently adopted in post-silicon validation, as described next.

Figure 2.13 shows transaction-based communication centric debug architecture, where the transaction monitor is used to observe the communication transactions between the embedded cores [27]. The debug control manages the interaction between

the cores by controlling the generation, and delivery of transactions during the debug process [42, 113]. A transaction is initiated by a master port on the initiator core, by sending a request which is executed by the receiving slave port on the target core that responds by sending an acknowledgment to the master. The SoC communication infrastructure uses specific communication protocols such as Open Core Protocol (OCP) [83], Device Transaction Level (DTL) [87], and AXI [9]. During the communication-centric debug session, upon the detection of a transaction with specific characteristics (e.g., with a specific destination target, data value, or frequency of occurrence), the transaction monitor can signal to the debug control unit that the interconnect has to stop the transportation of transactions [113]. The control debug unit can also stop the suspected core in order to inspect the system's state through a scan dump using scan-based debug methodology. The proposed debug approaches implemented in [42, 113] have been extended to address the distributed-shared-memory communication infrastructure for multi-processor SoCs [114].

Other transaction-level debug architectures are currently emerging. For example, novel transaction-based debug architecture has been recently proposed in [104] to enable an efficient and effective cross-trigger mechanism among multiple embedded cores. Debug features, such as transaction trace module and debug access module, have been introduced for the architecture presented in [102] in order to facilitate concurrent debug of the embedded cores and the inter-core communication infrastructure [103]. It should be noted that the debug architectures and techniques proposed in this dissertation can be used as complementary approaches to the architectures and methods discussed in this section to further increase their effectiveness in debug.

2.4.3 Compression in Embedded Logic Analysis

The capacity of on-chip trace buffers employed for embedded logic analysis limits the observation window of a debug experiment. To address this problem, compression techniques have been introduced to extract as much data as possible from a given debug experiment, i.e., to increase the observation window. The existing compression methods can be classified into special-purpose methods for embedded processors and

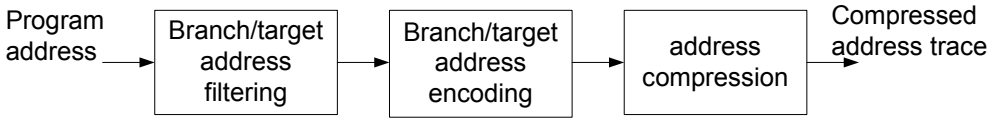


Figure 2.14: Compression Phases of Real-time Address Trace Compressor [60]

generic methods for any custom SoC design.

The trace compression techniques for embedded processors are based on exploiting the unique characteristics of the running program. For example, the running software contains code segments (or basic blocks) that are executed repeatedly and hence the address value of these blocks is increased by a certain offset. The address trace reduction can be achieved by eliminating the redundant temporal information on the address buses. It is essential to note that the hardware-based compression techniques for compressing the data traces are different from the software-based trace compression techniques that are used for simulation environments [106]. For the software-based compression techniques, the address trace can be compressed using a two-pass algorithm [88]. In the first pass, an intermediate partially compressed trace is created, and the dynamic control flow of the program is recorded. During this pass, the unique instruction addresses are expressed as offsets from their previous references. During the second pass, the generated trace from the first pass and the associated dynamic control flow information are used to encode the dynamic basic block successors and the data offsets using run length encoding. Despite the higher compression ratios that can be achieved using these software-based techniques, they are not suitable for real-time trace compression because of the area overhead required to store the trace data and the associated control information before compression. As a consequence, hardware-based trace compression techniques have been introduced to enable real-time trace compression, as described next.

Figure 2.14 shows the three phases of a hardware approach to real-time address trace compression for embedded processors [60]. In this approach, the program addresses generated from the address bus of the microprocessor are first passed through the branch/target filtering phase to filter out the sequential addresses and retain the

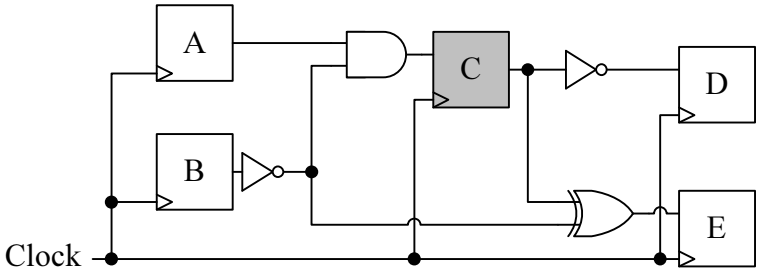
non-sequential addresses. The concept of trace filtering exploits the characteristics of the basic block which consists of a sequence of executed instructions. The basic block starts with a target instruction and it ends with a branch instruction which redirects the execution of the running software to another basic block. Because the addresses of the instructions between the target and the branch instructions within a basic block are incremented linearly with a certain offset, the branch/target filtering phase records only the addresses of the branch and the target instructions in order to reduce the trace size. The trace reduction techniques that are based on filtering out data at specific intervals by recording the target and branch addresses have been used extensively in debugging embedded processors [10, 52, 77].

In the second phase in Figure 2.14, the filtered addresses are encoded to eliminate the redundant address information by reducing the average bit width of the address data trace. Because the basic block contains few instructions, the difference between the address of the target instruction and the address of the branch instruction for the same block is small and hence this difference can be encoded using fewer bits than the bits required for encoding the branch address itself. During this second phase, the target address is encoded using the following two successive steps; the binary address patterns are first partitioned into equal slices, e.g., 32-bit word can be partitioned into 8 slices and each has 4 bits; then, the encoding is performed on these slices. The purpose of this address data partitions step is to facilitate the encoding process and to reduce the bit width of the address in order to achieve good compression during the final phase. To further reduce the size of the addresses, the third phase compresses the encoded address trace using a lossless data compressor of the Lempel-Ziv (LZ) compression algorithm [124]. This dictionary-based compression approach is employed in the third phase because its hardware implementation can perform compression in real-time, and it can achieve high compression ratio due to the repeated addresses patterns that result from the execution of the loop statements. The differences between the above described address trace compression technique [60] and our proposed generic compression techniques, which are described in this dissertation, are discussed in Chapter 3. Because the data in custom SoC designs may exhibit less repeated patterns or span larger ranges than the address data of the

embedded processors, different trace compression techniques have emerged to address this issue, as described next.

For custom SoC designs, several compression techniques have been recently proposed to extend the width and the depth of the observation window. In the width compression technique, the length of the observation window remains the same and compression is achieved by re-constructing the values of more signals from the ones that are captured in the trace buffer. For example, the width compression can be achieved by analyzing the design and identifying a subset of essential signals which, after being captured on-chip, would then be expanded in the debug software using the knowledge of the design data [48, 67]. By applying the sampled data onto a behavioral model of the design, the debug engineer can reconstruct data for signals that are not monitored by the embedded logic analyzer. The concept of signal restoration have been proposed to restore data in the combinational nodes of the circuit under debug [48]. By exploiting the boolean relationships among logic gates, data sampled in the state elements can be forward propagated and backward justified through the circuit netlist to reconstruct the behavioral information of the combinational nodes. The method proposed in [67] allows data to be restored in sequential elements across multiple clock cycles. In this approach, by only monitoring a subset of state elements, data can be reconstructed for other sequential elements, as well as combinational nodes in the design.

The basic principle of state restoration from [67] is demonstrated in the example shown in Figure 2.15. Figure 2.15(a) shows the CUD with five flip-flops (FFs), and Figure 2.15(b) gives the data in the state elements after the restoration algorithm from [67] is applied. In this example, only $FF\ C$ is sampled during clock cycles $0-3$. It should be noted that the x in the table refers to values that cannot be restored using only the subset of sampled data. Using backward justification, when a logic 1 is captured in $FF\ C$, the values of $FF\ A$ and $FF\ B$ can be evaluated as logic 1 and 0 respectively in the previous clock cycle. On the other hand, the values of $FF\ C$ can be forward propagated to indicate the values of $FF\ D$ in the following clock cycle. However, for $FF\ E$, its value can be reconstructed when the values of $FF\ B$ and $FF\ C$ are known in the previous clock cycles. As shown in this example, using only



(a) Circuit under Debug

Clock cycles

	0	1	2	3	4
A	1	1	x	x	x
B	0	0	x	x	x
C	0	1	1	0	x
D	x	1	0	0	1
E	x	1	0	x	x

(b) Restored Data in Sequential Elements

Figure 2.15: Illustrative Example of State Restoration [67]

the four values obtained in $FF\ C$ for clock cycles $0-3$, data can be restored for other state elements across multiple clock cycles; this results in a restoration ratio of $14/4 = 3.5X$. However, the number of clock cycles for which the data is restored depends on the length of the observation window determined by the depth of the on-chip trace buffer. Hence, any complementary *depth* compression techniques that can extend the observation window (as the ones presented later in this thesis) will also improve the methods from [48, 67].

2.5 Concluding Remarks

In this chapter, we have discussed the background and the related work of the existing techniques for post-silicon validation. We presented two complementary approaches. Although the first approach that relies on scan-based debug methodology provides full observability of the internal system's state, it is not capable of handling real-time

acquisition in consecutive clock cycles. It is also impractical to apply this technique every clock cycle over a long execution time. Thus, embedded logic analysis techniques, which are based on real-time trace, have been introduced in order to aid scan-based methodology and improve real-time observability. Finally, we discussed the existing techniques used to improve the observability of the internal circuit's nodes and accelerate the post-silicon validation process.

Chapter 3

Embedded Debug Architecture for Lossless Compression

In this chapter, we propose a novel architecture for embedded logic analysis that enables real-time lossless data compression in order to extend the observation window of a debug experiment and hence improve the observability. The proposed architecture is particularly suitable for in-field debugging of SoCs that have sources of non-deterministic behavior such as asynchronous interfaces. In order to measure the tradeoff between the area overhead and the increase in the observation window, we also introduce a new compression ratio metric. We use this metric to quantify the performance gain of three dictionary-based compression algorithms tailored for embedded logic analysis.

The rest of this chapter is organized as follows. Section 3.1 gives preliminaries for the research work presented in this chapter, outlines the motivation behind it and summarizes its contributions. Section 3.2 analyzes the performance of different compression algorithms based on the proposed new compression ratio metric. Section 3.3 presents the proposed debug architecture. Section 3.4 describes our proposed implementations of three dictionary-based lossless compression algorithms. Experimental results from Section 3.5 show how the proposed compression ratio metric is used to quantify the performance gain of these three compression algorithms. Finally, Section 3.6 concludes this chapter.

3.1 Preliminaries and Summary of Contributions

Embedded logic analysis has emerged as a powerful technique for the purpose of enabling at-speed acquisition of data from a limited set of internal signals in real time. Embedded logic analysis methods that rely on on-chip trace buffers are used as a complementary approach to scan-based methods to facilitate the identification of functional bugs during post-silicon validation [110]. Trace buffer-based techniques have been recently employed for debugging microprocessors [10, 38, 46], SoC designs [2, 48, 71, 110], and field programmable gate arrays (FPGAs) [5, 100, 122]. Nonetheless, the amount of debug data that can be captured into an on-chip trace buffer during embedded logic analysis is limited by the trace buffer width, which constrains the number of signals to be probed, and its depth, which limits the number of samples to be stored. To address this problem, compression techniques have been introduced for the width and/or depth of the trace buffer, as described in Section 2.4.3 from Chapter 2. In this chapter, we introduce a novel debug architecture that enables a depth compression technique for embedded logic analyzers.

The choice of the depth compression method relies on the type of the silicon debug experiment. The validation process generally consists of two main phases. In the first phase the bug appears on an application board and its occurrence cannot be reproduced immediately in a deterministic fashion. This happens if the bug is triggered by *non-deterministic input sources* coming from the design environment, such as asynchronous interfaces or interrupts from peripherals [94]. In this phase the main objective is to understand and isolate the input behavior that causes the bug to manifest itself. This helps defining the subsequent debug experiments when the bug can be triggered deterministically by controlling the inputs. In the second phase the silicon debug experiment becomes deterministic in the sense that the same behavior can be reproduced consistently either on an ATE or on an application board. The main objective in this phase is to trace the root cause of the bug and for this to be achieved it is necessary to collect as much data as possible (this data is subsequently passed to post-processing software). This type of debugging with *deterministic input sources* is referred to as deterministic replay or cyclic debugging.

For compressing debug data for the former problem (non-deterministic input sources), in this chapter we propose a novel debug architecture that supports real-time lossless data compression. Because the compression algorithms may vary in terms of their resource requirements for real-time compression, a trade-off between the amount of additional on-chip area required by the compression architecture and the compression ratio needs to be taken into account. Therefore, in this chapter we introduce a new compression ratio metric that measures the trade-off between the area overhead of the compression architecture and the increase in the observation window. The main contributions of this chapter are summarized as follows:

- we analyze the requirements for lossless data compression and introduce a new compression ratio metric that captures the real compression benefits of using compression in embedded logic analysis. Based on these requirements and the proposed compression ratio metric, we present the performance analysis of different lossless compression algorithms;
- we propose novel implementations of dictionary-based compression algorithms that satisfy high-throughput/real-time encoding to aid in embedded logic analysis. The proposed dictionary-based compression architectures support the most commonly used replacement policies.

3.2 Embedded Logic Analysis Framework

This section introduces the embedded logic analysis framework based on lossless compression. We first analyze the requirements for lossless data compression in embedded logic analysis. Second, we introduce a new compression ratio metric that measures the trade-off between the area overhead of the compression architecture and the increase in the observation window. Finally, we provide performance analysis of different compression algorithms based on the proposed compression ratio metric.

Figure 3.1 shows the basic principle of embedded logic analysis framework based on lossless compression. The off-chip debug software communicates with the on-chip debug module through a serial interface (e.g., JTAG [53]) as shown in Figure 3.1. A

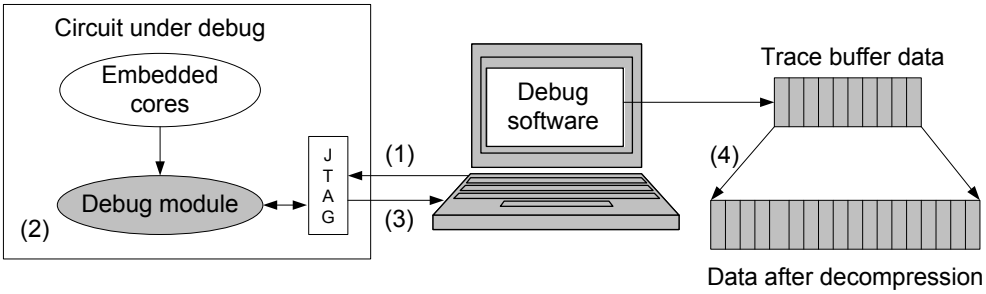


Figure 3.1: Embedded Logic Analysis Framework Based on Lossless Compression

debug experiment starts by uploading the debug module with its configuration (Step (1)), which specifies the trigger event at which the acquisition process starts. After the trigger event occurs, the compressed data is stored in the trace buffer (Step (2)). After the trace buffer is filled, the debug experiment is completed by offloading the compressed data to the debug software (Step (3)), where the debug information is decompressed (Step (4)).

3.2.1 Lossless Compression Requirements

Three main requirements have to be satisfied in order to enable lossless data compression in embedded logic analysis. The first requirement is obviously a good *compression ratio*. The second requirement is the *real-time compression*, which is motivated by the demand of at-speed sampling when hard-to-find bugs occur only due to external events that are dependent on the application environment. Because the application board may have sources of non-determinism that prohibit deterministic replay of the debug experiment, a lossless compression method should be used to exactly retrieve the sequence of events that have lead to or have happened after the trigger event has occurred. The third requirement is that the *area overhead* of the proposed architecture should have an acceptable impact on the silicon area. Having a compressor with a very large area will defeat the very purpose of using compression in embedded logic analysis, which is to extend the observation window of a debug experiment at low cost. For example, if the compression ratio is only 2x and the area overhead of

the encoder is larger than the trace buffer itself, then it is more convenient to double the size of the trace buffer, as we will obtain the same observation window with less silicon area. These three requirements uniquely characterize the compression method in embedded logic analysis. Therefore, we propose a novel *compression ratio metric* to quantify the performance gain from using a specific compression algorithm in an embedded debug module.

3.2.2 The Proposed Compression Ratio Metric

The proposed metric is called *adjusted compression ratio* and it is defined as $CR_{adjust} = \frac{CR_{data}}{(1+\frac{AreaOverhead}{M})}$; where CR_{data} is the data compression ratio achieved by the compression algorithm; M is the area of the trace buffer; and $AreaOverhead$ is the area required by the hardware implementation of the compression algorithm. If the compression is not employed and the trace buffer is increased to include the area overhead of the compression architecture, the observation window size would increase by $(\frac{M+AreaOverhead}{M})$. Intuitively, the proposed compression ratio measures the trade-off between the area overhead of the compression architecture and the increase in the observation window. For example, if the trace buffer size is 8 kbytes and the equivalent area overhead of a specific compression architecture in terms of trace buffer area is also 8 kbytes and the achieved data compression ratio is 2, then using compression will bring no benefit in extending the observation window because $CR_{adjust} = 1$. In summary, this new metric captures the authentic merit of the compression method, by taking into account both the compression ratio and the area overhead. Before justifying our selection of compression algorithms to be used in embedded logic analysis, we first review and analyze several lossless compression algorithms based on the previous requirements and the proposed compression ratio metric.

3.2.3 Performance Analysis of Compression Algorithms

Lossless data compression algorithms can be classified into two main categories either *statistical coding algorithms* or *dictionary coding algorithms*. It is known that statistical-based compression algorithms, such as Huffman coding [50] or arithmetic

coding [92], can lead to an optimal average code length and hence a good *compression ratio* can be achieved [50]. For instance, in Huffman coding, variable length codewords are used to replace the most frequently occurring symbols with shorter codewords and the less frequently ones with longer codewords. The static implementation of these methods requires two passes over the same data, in the first pass the tree-based codewords structure is built and in the second pass the encoding is performed.

For the *real-time* requirement and the fact that debug data is not known a-priori as mentioned earlier, these static methods are not suitable for embedded logic analysis. Nonetheless, statistical-based methods can be implemented in an adaptive way, as in the case of dynamic Huffman coding algorithm [65] where the tree-based code is created and maintained according to the changes in the incoming symbols' probabilities. The hardware implementation of adaptive Huffman coding, based on a tree-based codewords structure, is presented in [74] with a throughput of approximately 1 bit/cycle. Consequently, for a *high throughput* application, the tree-based implementation will not be suitable. However, the adaptation can be done in real time with extra *area overhead* based on ordered codeword tables [68, 73]. Nevertheless, according to the proposed compression ratio metric CR_{adjust}, these adaptive statistical-based methods implementations will bring no benefit in embedded logic analysis. This is because their silicon area is prohibitively large to be implemented into an embedded debug module.

The second category of data compression algorithms is the *dictionary-based compression algorithms*. Representative examples of this type of algorithms are locally adaptive data compression algorithm (BSTW) [17] or Lempel-Ziv (LZ77) compression algorithm [124] and its variants LZ78 [125], LZW [119] and WDLZW [56]. The compression in these algorithms is achieved by encoding a symbol or a sequence of consecutive symbols into shorter codewords which are represented by indices to the dictionary entries. The adaptive dictionary-based algorithms can achieve competitive compression ratios compared to the adaptive statistical algorithms [17]. In order to perform fast search in hardware between the incoming symbol and the dictionary entries, a dedicated fast parallel search engine called a content-addressable memory

(CAM) is used [84]. The hardware implementation of the adaptive dictionary-based algorithms can achieve both *high throughput* and good *compression ratio* as presented in [73, 81, 82]. However, the area reported in the previous implementations is too large to be acceptable in embedded logic analysis. Therefore, in this chapter, we propose a debug architecture, that supports adaptations of dictionary-based compression algorithms, to achieve *real-time compression* and an acceptable balance between the *compression ratio* and the *area overhead* as quantified by the proposed compression metric CR_{adjust}.

3.3 The Proposed Debug Architecture

This section describes the proposed debug architecture which enables real-time lossless data compression. The main contribution in this architecture is the encoder module which is shaded in Figure 3.2. First, we provide an overview of the main features in an embedded debug module. Second, we introduce the proposed encoder architecture, which supports dictionary-based lossless compression algorithms, followed by its variants of implementations that support different replacement policies.

3.3.1 Overview of Embedded Debug Module

The embedded debug module enables capturing a set of internal samples after the occurrence of a certain triggering condition. By monitoring a group of trigger signals, an event detector determines when the debug data is captured in the trace buffer as shown in Figure 3.2. In our implementation, the triggering condition can be performed based on bitwise, comparison or logical operations between certain selected trigger signal and a specified constant value. To further enhance the detection ability of the debug module, the event detector is followed by an event sequencer to monitor a specified sequence of events. The configuration of the trigger signal selection, the triggering conditions, and the choice of the signals that need to be probed are uploaded to the embedded debug module control through the low bandwidth interface (e.g., JTAG [53]).

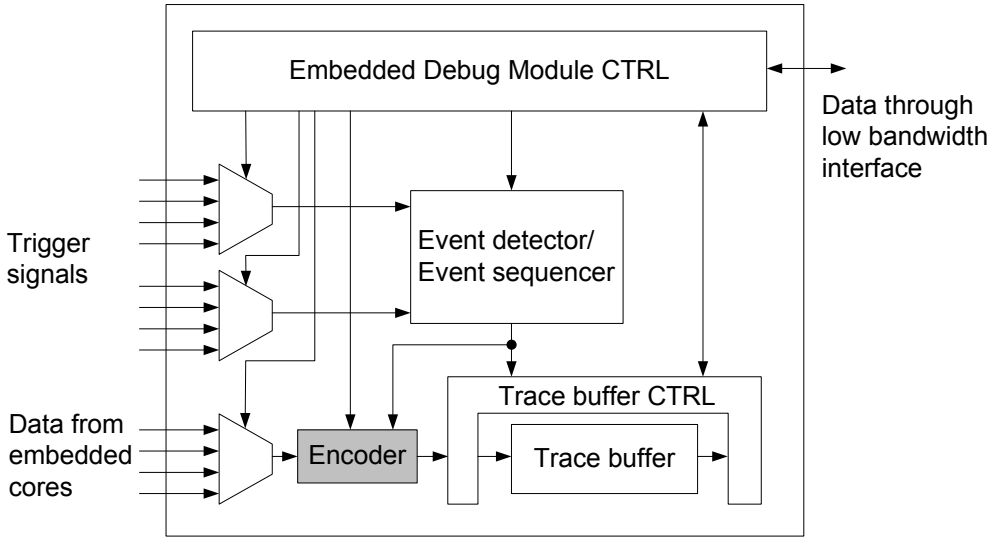


Figure 3.2: The Proposed Embedded Debug Module

3.3.2 The Proposed Encoder Architecture

In this chapter, we propose three implementations of dynamic dictionary-based compression algorithms for embedded logic analysis. First, a locally adaptive data compression algorithm or BSTW [17] is used based on a fixed width dictionary structure. Second, a modified version of this algorithm is presented. Third, a word-based dynamic Lempel-Ziv (WDLZW) data compression algorithm [56] is employed based on a hierarchy variable word dictionary width structure. In these algorithms, the dictionary is implemented in hardware using CAM whose depth represents the total number of entries in the dictionary.

Figure 3.3 shows the proposed encoder architecture that supports these dictionary-based compression algorithms and Table 3.1 gives the CAM terminology used in this architecture. To illustrate the basic principles of using the CAM for real-time data compression, we first introduce the hardware implementation of the single-symbol width dictionary BSTW algorithm as follows. At the beginning of a debug experiment, the encoder is enabled by setting $Encoder - en$ flag and the user can configure

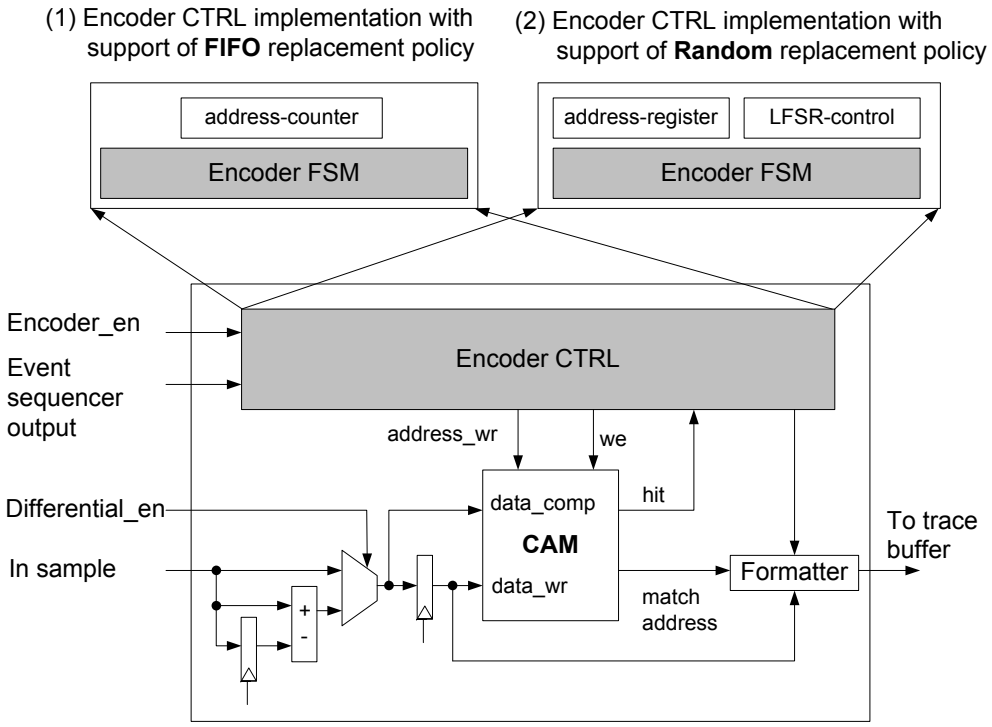


Figure 3.3: The Proposed Encoder Architecture with support of FIFO and Random Replacement Policies

$Differential-en$ flag to select one of the two lossless compression techniques: direct encoding or differential encoding (where the difference between two successive symbols is encoded). The direct encoding method is explained as follows: after the trigger condition occurs, the encoder starts its operation; if the symbol exists in the CAM then the match address, which represents the codeword, is written in the trace buffer; if the incoming symbol does not exist in the CAM, a codeword 0 (this codeword is reserved for the symbols that do not exist in the CAM), indicating that this symbol is an un-encoded symbol followed by the symbol, to be written in the trace buffer; then, this symbol is written in the CAM at the address pointed by a control address counter in the *Encoder CTRL*. This address counter is initialized to 1 at the beginning of the debug experiment and it is incremented each time a mismatch occurs. Once the counter reaches the threshold of the CAM's depth, which indicates

Table 3.1: Terminology for the Proposed Encoder Architecture

Name	Representation
we	CAM write enable
$data - wr$	CAM write data
$data - comp$	CAM compare data
$address - wr$	CAM write address
$match - address$	The match address (codeword)

that the CAM is filled with symbols, a replacement policy is employed. Because the replacement policies vary in terms of their resource requirements, we discuss the implementations of the most commonly used replacement policies in the following subsections. It should be noted that the proposed architecture provides simultaneous data searching and writing in the same clock cycle, and it has been verified using Xilinx CAM [121].

The First-in First-out Replacement Policy

The first-in first-out (FIFO) replacement policy is employed to replace the symbols that have been written in the CAM in sequence starting from the oldest symbol. Regardless of the recency or the frequency of the symbol occurrence, the replacement is decided based on its first occurrence when it was written in the CAM. To enable this replacement policy in the proposed architecture shown in Figure 3.3, the address counter which exists in the $Encoder - CTRL$ restarts from address location 1 once it reaches the threshold of the CAM's depth.

Random Replacement Policy

Random replacement can be enabled using a linear feedback shift register (LFSR). A LFSR is a shift register whose input bit is a linear function which is provided by exclusive-or (xor) of some bits of the entire shift register value. In this replacement policy, the LFSR is employed as a Pseudo-random number generator [13], where the generated number represents the address of the symbol that needs to be replaced. To generate a long output sequence that covers the entire address space of the CAM

entries, the LFSR has to represent a maximal polynomial (i.e., LFSR with width n will produce $2^n - 1$ states within the shift register except the state where all of its bits equal zero). The initial value of the LFSR is called the seed and it has to be not equal 0 since this value will lead to no change in the LFSR state. Because the location 0 is not used in the encoding process as described above, the address register, shown in Figure 3.3, can be employed to work as a LFSR with an initial seed equals 1 once the replacement starts. Note that the LFSR will eventually enter a repeating cycle after it generates all of its possible states that represent the maximal sequence.

The hardware implementation of either the FIFO or the random replacement policy requires a smaller area overhead than the one required by the least recently used (LRU) replacement policy or the move-to-front (MTF) replacement policy [17]. For the MTF replacement policy, most recently used symbol will be stored in the first CAM cell whereas the contents of CAM cells will be shifted down and hence the LRU symbol will be replaced when a new symbol needs to be added in the CAM. Thus, the MTF replacement policy requires a specific construction of the CAM cells [81, 82], and hence a substantial area is required. This makes this type of replacement policy unacceptable for embedded logic analysis, because the compression gain is offset by the silicon area, as discussed earlier in Section 3.2.2. Nonetheless, we propose a modified LRU replacement policy of relatively low cost in order to achieve a reasonable compression ratio and attain an acceptable area overhead.

Modified Least Recently Used Replacement Policy

The proposed modified least recently used (LRU) replacement policy is based on combining FIFO replacement policy with LRU replacement policy to obtain an approximate LRU replacement policy that has a small impact on the silicon area. In the proposed architecture shown in Figure 3.4, the CAM is divided into n segments and the index of the most recently used (MRU) segment is stored in register 0 ($Reg(0)$) while the index of the LRU segment is stored in register $n-1$ ($Reg(n-1)$). This register array is used to keep track of the order of recently accessed CAM segments during the encoding process. The value stored in address segment register ($address - seg$) represents the address of the symbol that needs to be added or replaced within the

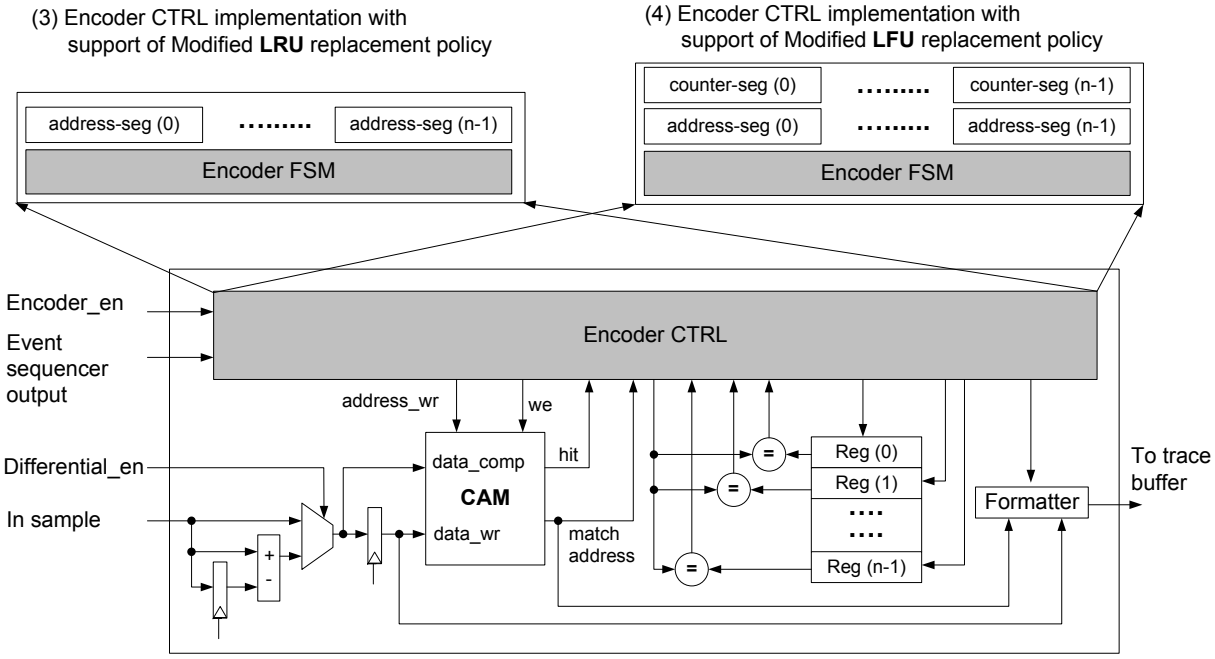


Figure 3.4: The Proposed Encoder Architecture with support of Modified LRU and Modified LFU Replacement Policies

segment during the encoding process. Because we use FIFO replacement within each segment, this register is employed to operate as a counter. The width (and depth) of the register array and the width of the address segment register ($address - seg$) are dependent on both the CAM depth and the number of segments (e.g., if CAM depth equals 64 and the number of segments equals 4, then the width of register array equals 2 bits and its depth is 4 registers while the width of $address - seg$ register is 4).

Algorithm 1 describes the updating process of the register array during each clock cycle, and it also shows how the writing address $address-wr$ of the CAM is computed to enable the proposed modified LRU replacement policy. Before the updating process of the register array starts, each register of this array is initialized to its segment index (lines 1 and 2). It is essential to note that before the CAM is filled with symbols, the segment addresses operate in sequence starting from segment 0 to $n-1$ and each

Algorithm 1: Modified LRU Replacement Policy

```
1  for (i = 0 to n − 1) do
2    Reg (i) ← i;   (initialize the register array
                       with segment indices)
   end
3  while (the debug experiment is running on CUD) do
4    if (miss and CAM is filled) then
5      address-wr = {Reg(n − 1),address-seg(Reg(n − 1))};
6      Reg (0) ← Reg (n − 1);
7      for (i = 0 to n − 2) do
8        Reg (i + 1) ← Reg (i);
       end
     else  (match occurs (or miss occurs and CAM is not
             filled where the writing address occurs) on segment
             whose index is stored in Reg(k))
9      if (k! = 0) then
10       Reg (0) ← Reg (k);
11       for (i = 0 to k − 1) do
12         Reg (i + 1) ← Reg (i);
         end
       end
     end
   end
```

address segment ($address - seg$) counts from 0 to its maximum value (e.g., a value of 31 for a 5-bit register). As shown in Algorithm 1, three cases may occur during the updating process of the register array.

- If the incoming symbol does not exist in the CAM and CAM is not filled, the register array is updated as shown from lines 9 to 12. In this case, if the writing address of this symbol is located on the segment whose index is stored in $Reg(k)$, then the index of this segment will be moved to $Reg(0)$ (i.e., it becomes MRU segment) and its upper register values will be shifted down in the following clock cycle. It is obvious that the state of the array register will remain the same when $k = 0$ because $Reg(0)$ contains the index of the MRU segment.

- If the incoming symbol does not exist in the CAM and the CAM is filled, the register array is updated as shown from lines 4 to 8 in Algorithm 1. As it can

be noted from line 5, the CAM writing address of this symbol is located in the LRU segment whose index is stored in $Reg(n-1)$. In the following clock cycle, the stored value in $Reg(n-1)$ which represents the index of the LRU segment will be moved to $Reg(0)$ since this segment is updated by the MRU symbol. In addition, the other registers indices will be shifted down once in the register array.

- If the incoming symbol exists in the CAM, the register array is updated as shown from lines 9 to 12 in Algorithm 1. This updating criterion is similar to the one described in the first case, with the exception that the match address occurs on the segment whose index is stored in $Reg(k)$.

The drawback of the LRU replacement policy lies in the inability to distinguish between the frequently and the infrequently accessed segments. In order to take the frequency of the accessed segment into consideration, next we propose a modified least frequently used (LFU) replacement policy.

Modified Least Frequently Used Replacement Policy

The proposed LFU replacement policy replaces the symbol that is located in the LFU CAM segment. We use a similar approach to the one proposed for the modified LRU replacement policy where the CAM is divided into n segments. In the proposed architecture shown in Figure 3.4, the index of the most frequently used (MFU) segment is stored in $Reg(0)$ while the index of the LFU segment is stored in $Reg(n-1)$. This register array is used to keep track of the order of frequently accessed CAM segments during the encoding process. In order to record the frequency of the accessed segment, a counter ($counter-seg$) is employed to be associated with each segment. The width of this counter can be selected to allow as many accesses as a multiple of the depth of its segment (e.g., a 6-bit counter is associated with a segment whose depth is 32, which provides 64 accesses).

Algorithm 2 shows the updating process of the register array during each clock cycle in order to enable the proposed modified LFU replacement policy. Before the updating process of the register array starts, each register of this array is initialized

to its segment index and each of the segment counters ($counter - seg$) is initialized with 0 as shown from lines 1 to 3. The updating criterion of the register array is decided based on the frequency of the accessed segment. Whenever the segment is accessed (i.e., in the case of a miss, a symbol is added or replaced, whereas in the case of a hit, the match address is located on this segment), its associated $counter - seg$ is incremented by one. The main procedures of this updating criterion are described as follows.

- For the case of the miss occurrence and the CAM is not filled, the register array is updated as shown from lines 15 to 23 in Algorithm 2. In this case, if the writing address of the incoming symbol is located on a segment whose index is stored in $Reg(k)$, then the register array will be updated based on the comparison result between the value of $counter - seg(Reg(k)) + 1$ and the value of $counter - seg$ associated with each of the segment indices that are stored in the upper registers. This proposed updating criterion is taking into account the recency of the segment in the case of the equality between these values (i.e., the index of the most recently used segment will be moved above the least recently used one in the following clock cycle).

- For the case of the miss occurrence and the CAM is filled, the register array is updated as shown from lines 5 to 14. As a consequence, $Reg(n-1)$ which contains the index of the LFU segment will be moved to a specific upper register based on the comparison result between the value of $counter - seg(Reg(n-1)) + 1$ and the value of each $counter - seg$ associated with the segment indices that are stored in the upper registers. As it can be noted, if the value of $counter - seg(Reg(n-1)) + 1$ is less than any of the upper segment counters, the state of the upper registers will remain the same in the following clock cycle as shown from lines 13 and 14 in Algorithm 2.

- For the case of the match occurrence, the register array is updated as shown from lines 15 to 23 in Algorithm 2. This updating criterion is similar to the first one described above for the case of a miss occurrence and the CAM is not filled.

Algorithm 2: Modified LFU Replacement Policy

```
1  for (i = 0 to n − 1) do
2    counter-seg(i) ← 0 ;
3    Reg (i) ← i;   (initialize the register array with segment indices)
   end
4  while (the debug experiment is running on CUD) do
5    if (miss and CAM is filled) then
6      address-wr = {Reg(n − 1),address-seg(Reg(n − 1))};
7      counter-seg(Reg(n − 1))←counter-seg(Reg(n − 1))+1;
8      for (i = n − 2 downto 0) do
9        if (counter-seg(Reg(n − 1))+1) ≥ counter-seg(Reg(i)) then
10         Reg (i + 1) ← Reg (i);
11         if (i = 0) then
12           Reg (0) ← Reg (n − 1);
           end
         else
13           Reg (i + 1) ← Reg (n − 1);
14           break;   (break the for loop)
         end
       end
     else  (match occurs (or miss occurs and CAM is not filled where the
             writing address occurs) on segment whose index is stored in Reg(k))
15     counter-seg(Reg (k))←counter-seg(Reg(k))+1;
16     if (k! = 0) then
17       for (i = k − 1 downto 0) do
18         if (counter-seg(Reg(k))+1) ≥ counter-seg(Reg(i)) then
19           Reg (i + 1) ← Reg (i);
20           if (i = 0) then
21             Reg (0) ← Reg (k);
             end
           else
22             Reg (i + 1) ← Reg (k);
23             break;   (break the for loop)
           end
         end
       end
     end
24   if (counter-seg(Reg (k))=Maximum-count or
         counter-seg(Reg (n − 1))=Maximum-count) then
25     for (i = 0 to n − 1) do
26       Reg (i) ← Reg (i)/2 ;  (a right shift by one)
       end
     end
   end
```

- If the $counter - seg$ reaches its maximum value (e.g., a value of 63 for a 6-bit counter), each counter of the segment counters will be divided by 2 (i.e., shifted right by one) to maintain the order of the frequency of the accessed segments, as illustrated from lines 24 to 26 in Algorithm 2.

Because the replacement decision in this policy is based on the frequency of the accessed segments, the replacement can be performed on the recently accessed segment that has the lowest access frequency. However, depending on how the input symbols occur frequently over short intervals, this modified LFU replacement policy can achieve better compression ratios than the proposed modified LRU replacement policy as demonstrated later in the experimental results section.

In the previous implemented replacement policies, the debug experiment ends on the CUD after the trace buffer is filled with the compressed debug data. Thereafter, its content is decompressed at the debug software, as illustrated in Figure 3.1. Because the un-encoded symbol always comes after a codeword 0, the decoder at the software side builds a lookup table by adding the un-encoded symbol to the table or retrieving a symbol from the lookup table addressed by its codeword. The updating process of the lookup table uses the same replacement policy that is used during the encoding process and hence the decompression process requires a relatively simple software implementation.

3.4 Dictionary-based Compression Algorithms

This section describes three proposed dictionary-based compression algorithms for embedded logic analysis and analyzes the trade-off between the area and the achieved compression ratio. In the previous section, we have introduced, for illustrative purposes, the single-symbol width dictionary BSTW algorithm with different implementations of the most commonly used replacement policies. To improve the compression ratio we need to support multiple-symbol encoding. First, we implement the BSTW dictionary-based algorithm based on a fixed width dictionary structure for multiple-symbol encoding. Second, to address the limitations of the first implementation, we

propose a new compression algorithm, called modified BSTW (MBSTW), to support multiple-symbol encoding with a reduction in the encoder area. Third, we explore the suitability of WDLZW dictionary-based compression algorithm for embedded logic analysis, by implementing it using a hierarchy of dictionaries with successive increase in the word width.

3.4.1 BSTW Dictionary-based Compression Algorithm

A locally adaptive data compression algorithm (BSTW) [17] is implemented based on a fixed width dictionary structure. To achieve a good compression ratio, multiple symbols should be mapped into a single codeword; i.e., the dictionary's width has to accommodate more than a single symbol. However, depending on how the input symbols occur frequently over short intervals, there is a trade-off between the dictionary's width and the compression ratio: the larger the dictionary's width, the higher the compression for highly correlated data. We refer to successive symbols which are occurring frequently over short intervals as a *correlated data*, i.e., if the interval between the occurrence of successive symbols and the subsequent occurrence is shorter than the dictionary's depth, then the match occurs and hence greater compression is achieved. Nonetheless, large dictionary's width may not give better compression than a smaller one (e.g., three successive symbols may not occur as frequently as two successive symbols). In addition, the larger the dictionary's depth, the higher the probability of finding the symbols and hence the higher the compression that can be achieved. Nonetheless, note that the compression ratio can be affected by large dictionary's depth if input data requires only few entries in the dictionary.

Figure 3.5(a) shows an example of the updating process for the 2-symbol width BSTW dictionary, where the dictionary is updated with a new entry every two clock cycles. The generation of an output (either a codeword or un-encoded two symbols) occurs every two clock cycles. It should be noted that at the beginning, the dictionary is empty and whenever two successive symbols are not in the dictionary, they are added in the appropriate location (as shown in Figure 3.5(a), *ab* added to location 1, *cb* added to location 2, *cc* added to location 3, etc.). As observed from this example,

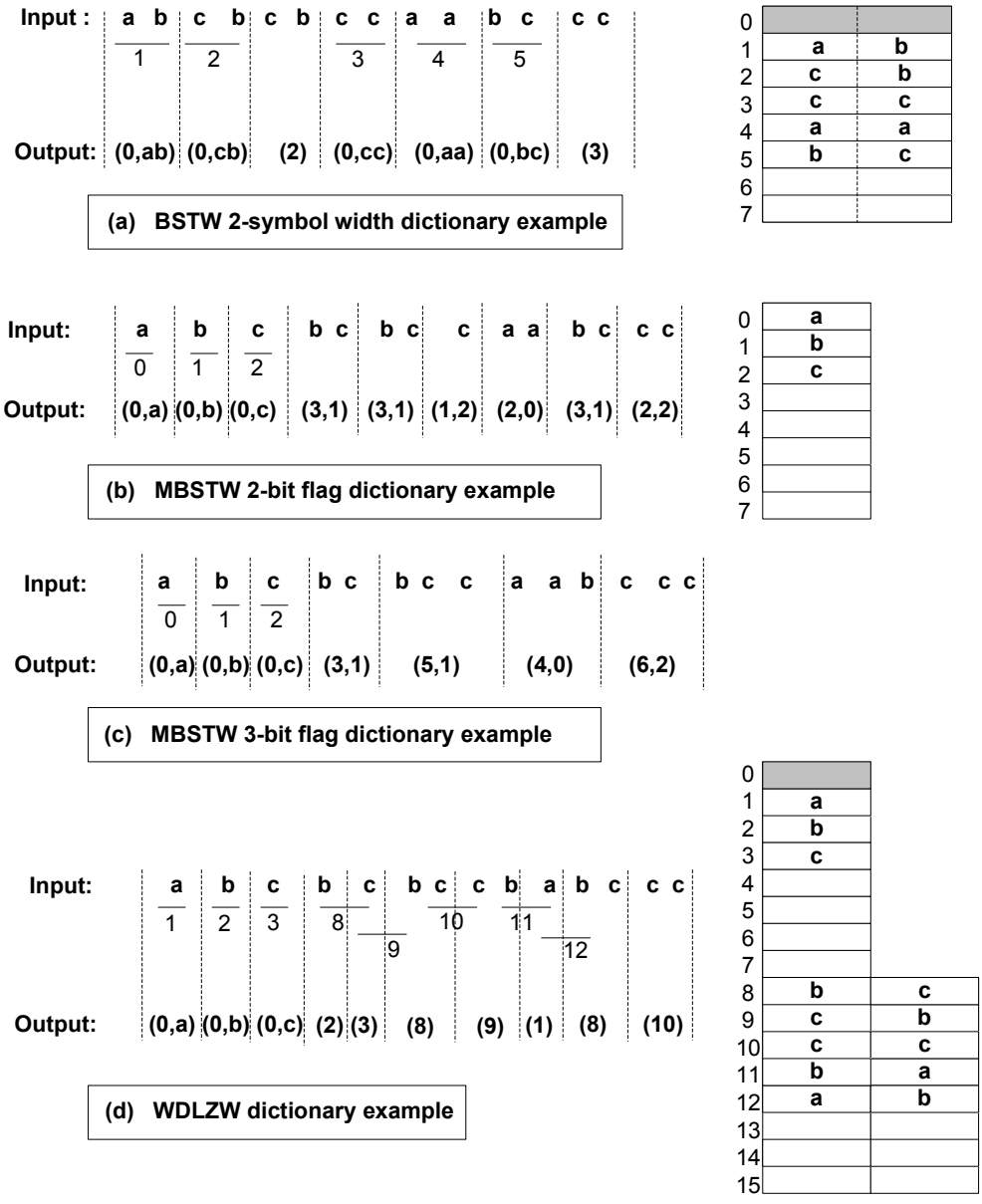


Figure 3.5: Illustrative Examples for Dictionary-based Compression Algorithms

every mismatch generates a codeword 0 (this indicates that the two successive symbols do not exist in the CAM) followed by the two symbols (as shown in Figure 3.5(a), where (0,ab) means un-encoded two successive symbols). If the match occurs, the match address represents the codeword (as shown in Figure 3.5(a), where (2) means match address of two successive symbols existed in the dictionary at location 2). Once the dictionary is filled, the replacement policy is employed. The proposed replacement policies described in Section 3.3.2 can be applied to a multiple-symbol dictionary taking into account how often the updating process is performed (i.e., for 2-symbol dictionary structure, the update is executed every two clock cycles).

The implementation of a 2-symbol (or 3-symbol) dictionary algorithm requires twice (or three times) as much area as the one required by a single-symbol dictionary algorithm. In addition, the mismatches when using multiple-symbol dictionary greatly influence the compression ratio. Therefore, in the next subsection, we propose a modification of the single-symbol dictionary that takes into account few cases of the correlation between two successive symbols or among three successive symbols.

3.4.2 MBSTW Dictionary-based Compression Algorithm

The objective of the proposed MBSTW algorithm is to achieve a good compression ratio at low cost and to support multiple-symbol encoding. Therefore in our implementation, a single-symbol dictionary is combined with tag information to account for the correlation among successive symbols.

To consider the correlation between two successive symbols, we use a 2-bit flag as a tag information as follows: 00 means that no encoding is done (un-encoded symbol); 01 indicates single encoded symbol; 10 means that the two successive symbols are equal; and 11 indicates that the two successive symbols are located in two consecutive locations in the CAM. The compression ratio increases when the last two cases occur frequently. In this algorithm, we do not need to send a codeword 0 with the un-encoded symbols (i.e., the dictionary is started from location 0) since the 00 flag implies that the symbol is un-encoded. As shown in Figure 3.5(b), the outputs are

written in parenthesis (e.g., (0,a)), where the first element represents the tag information (2-bit flag represented in decimal) and the second element represents either the un-encoded symbol or the match address. The compressed stream is generated as follows: flag 0 followed by un-encoded symbol; flag 1 followed by the match address as a codeword; flag 2 followed by the match address and flag 3 followed by the match address of the first symbol of the two successive symbols. The tag information along with the un-encoded symbol or the codeword are written in a chronological order in the trace buffer and this tag information facilitates the decoding process. At the debug software side, this flag is read first and based on the flag value, the symbol is retrieved or added (when the flag is 0) to the lookup table.

To increase the compression ratio for correlated data, correlation among three successive symbols can be taken into account. To address different cases for three successive symbols, a 3-bit flag is employed. The first four cases are the same as the cases of the 2-bit flag and another four cases are handled based on three successive matches as follows: 100 means the first two symbols are identical successive symbols and the third is located in the CAM location that is next to the second; 101 indicates that the first two symbols are located in two consecutive locations in the CAM and the third is identical to the second; 110 means three identical successive symbols; and 111 indicates that three different successive symbols are located in three consecutive locations in the CAM. Figure 3.5(c) shows few cases of the previous description.

The proposed implementation of MBSTW dictionary-based algorithm has a small impact on the area overhead of the single-symbol width dictionary. Therefore, it provides a performance gain in CR_{adjust} when compared to the case of using multiple-symbol dictionary BSTW. This improvement is demonstrated later in the experimental results section.

3.4.3 WDLZW Dictionary-based Compression Algorithm

The word-based dynamic dictionary algorithm (WDLZW) is based on using a hierarchy of dictionaries with successive increase in the word's width [56]. This algorithm uses MTF replacement policy. As discussed in Section 3.3.2, the hardware implementation of this replacement policy incurs a significant area overhead and hence it

adversely impacts CR_{adjust}. Therefore, in our implementation we use the proposed replacement policies described in Section 3.3.2 for all the dictionaries.

As shown in Figure 3.5(d), there are two dictionaries: one has a single-symbol width and the other has a 2-symbol width. The construction of the dictionary may account for more than 2-symbol, however this is achieved at the expense of an extra area overhead. Figure 3.5(d) illustrates the updating process of the dictionaries. It shows that the 2-symbol width dictionary accommodates the recent two symbols that exist in the input stream and also are located in the single-symbol width dictionary. The main limitation of this algorithm is related to using different CAM sizes and hence more hardware resources are needed for this architecture. However, depending on how the input symbols occur frequently over short intervals, this algorithm can achieve better compression ratio compared to the multiple-symbol BSTW dictionary-based algorithm. This is because the WDLZW algorithm uses different dictionaries with different word widths and hence any single-symbol match is encoded either combined with another consecutive matching symbol, or separately when it is followed by a mismatch.

In Chapter 2, we described a recent hardware approach to real-time address trace compression for embedded processors [60]. To emphasize the contributions of our methods, we explain the differences between the address compression approach introduced in [60] and the proposed methods from our work which target the compression of any type of data buses for custom SoCs:

- the implementation of the LZ dictionary-based method introduced in [60] at the third phase for compressing the address trace is different from the implementations of the dictionary-based methods described in this section. Our proposed implementations are based on the CAM architecture that supports different replacement policies. On the other hand, the implementation of the LZ dictionary-based method [60] is based on the sliding dictionary structure which is represented by a shift register (i.e., it supports only FIFO replacement policy). As demonstrated later in the experimental results, the area of the hardware implementation of LZ-algorithm reported in [60] is at least five times greater than the area of the hardware implementation of the proposed MBSTW

(or WDLZW) dictionary-based method using a CAM depth of 256.

- the dictionary entry in our implementations can represent one or multiple symbols based on the dictionary structure, while the dictionary entry in [60] is represented by an encoded address slice generated from the second phase. The high compression ratios achieved using LZ-algorithm reported in [60] are primarily due to the repeated patterns of the encoded address slices. These address patterns are commonly generated during the program execution in the embedded processors. If the implementation of LZ-algorithm [60] is extended to support the compression for any type of data buses, both the area and the compression ratio will be adversely impacted.

In summary, in this section we have adapted two existing dictionary-based compression algorithms (BSTW and WDLZW) for real-time hardware encoding, as required by embedded logic analysis. We have also proposed a new MBSTW algorithm that improves on BSTW by addressing the CR_{adjust} metric. An experimental comparison for these three algorithms is reported in the following section.

3.5 Experimental Results

This section discusses the experiments concerning the area investment and the compression benefits of the proposed dictionary-based compression algorithms. The area of the proposed encoder is estimated using a 180nm ASIC standard cell library and the debug data has been collected from an FPGA prototype of an MP3 audio decoder [45].

3.5.1 Area of the Proposed Encoder Architecture

The area overhead of the proposed encoder architecture is estimated for different lossless dictionary-based compression algorithms in terms of 2 input NAND (NAND2) gates. Table 3.2 provides the estimated area overhead of the encoder for different replacement policies. The area represents the size of the control logic (including the

Table 3.2: Area of The Proposed Encoder Architecture in NAND2 Equivalents

Rep. Policy	CAM Depth	BSTW			MBSTW		WDLZW	
		1-Sym	2-Sym	3-Sym	2-Sym	3-Sym	2-Sym	3-Sym
FIFO	16	1194	1606	2035	1515	1665	1544	2226
	64	2194	3455	4748	2532	2716	3014	3956
	256	5563	10264	15014	5945	6162	8155	9998
	1024	19601	38112	56687	20000	20255	29142	34484
RAN	16	1216	1621	2050	1553	1704	1584	2242
	64	2222	3477	4770	2589	2771	3071	4027
	256	5600	10293	15043	6017	6233	8226	10101
	1024	19641	38139	56714	20076	20331	29221	34613
LRU	16	1287	1682	2111	1614	1766	1685	2402
	64	2482	3724	5017	2834	3021	3221	4222
	256	6356	11038	15785	6761	6980	8811	10557
	1024	20574	39057	57631	21001	21258	29958	35181
LFU	16	1494	1888	2318	1821	1972	1999	2777
	64	3090	4327	5619	3437	3625	3716	4872
	256	8105	12782	17529	8508	8727	10213	11845
	1024	22783	41261	59835	23209	23469	31781	36865

data formatter) and the size of the CAM and it does not account for the trace buffer area. The area of the CAM was approximated to be twice as much as the area of a RAM of the same capacity, as estimated in [84].

The results shown in Table 3.2 are reported for different CAM sizes, replacement policies and encoding approaches. The depth of the single-symbol width dictionary in the WDLZW dictionary-based algorithm equals half of the CAM depth, whereas the depth of the two-symbol width dictionary is the remaining depth in the case of two-symbol (2-Sym) approach. The depth of the 2-symbol width dictionary is half of the remaining depth for the three-symbol (3-Sym) approach, whereas the depth of the 3-symbol width dictionary is the other half. The variation in the encoder area for a specific encoding approach is due to the data formatting of the codewords, the replacement policy and the CAM size. The more symbols are accounted for during compression, the larger the area overhead, which will likely offset the compression

benefits when using a specific dictionary structure as explained in the next subsection. What needs to be noted is that the proposed MBSTW algorithm leads to the lowest area overhead (e.g., for CAM depth 256 (or 1024), the area of MBSTW algorithm is at least 30 % less than the area of BSTW or WDLZW algorithm), which is a key requirement for embedded logic analysis.

The area results of both FIFO and random replacement implementations are for the unsegmented CAM. The area of the random replacement implementation is slightly greater than the one for FIFO. This difference is due to the employed linear feedback shift register (LFSR) and the associated logic, which are used to enable the Pseudo-random replacement as described in Section 3.3.2. Because the generated sequence from the LFSR does not include 0, we have added a control logic in the implementation of MBSTW random replacement in order to replace the symbol that is located at CAM address location 0. Similarly, an added control logic is employed in the implementation of the random replacement policy of the multiple-symbol CAMs that are used in WDLZW algorithm.

Table 3.2 shows the results for the segmented CAM for both modified LRU and modified LFU replacement implementations. The number of allocated segments for CAM depths 16, 64, 256, 1024 of BSTW and MBSTW encoding algorithms are 2, 4, 8, 8, respectively. The number of allocated segments for CAM depths 16, 64, 256, 1024 of 2-symbol WDLZW encoding algorithm are (2,2), (2,2), (4,4), (4,4) respectively; where the first element written in the parenthesis (,) represents the number of segments for the single-symbol width dictionary and the second element represents the one for the 2-symbol width dictionary. The number of allocated segments for CAM depths 16, 64, 256, 1024 for 3-symbol WDLZW encoding algorithm are (2,2,2), (2,2,2), (4,2,2), (4,2,2) respectively.

In the case of modified LFU implementation, there is a counter associated with each segment in order to keep track of the frequency of occurrence for any accessed segment, as explained in Section 3.3.2. The width of each segment counter is selected to allow twice as many accesses as the segment depth (e.g., for a CAM depth 256 which is divided into 8 segments, each segment contains 32 locations and hence the

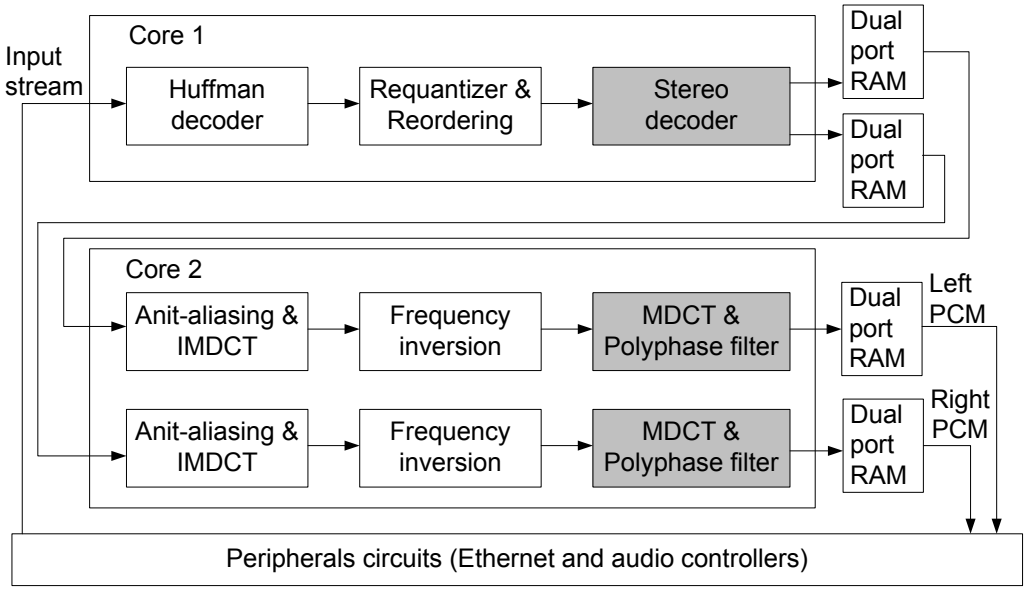


Figure 3.6: MP3 Decoder Architecture

counter width equals 6 bits). Thus, the implementation of the modified LFU replacement policy has a higher impact on the encoder area than the one for the modified LRU replacement policy. The influence of these replacement implementations on the compression ratio is explained in the following sub-section.

3.5.2 MP3 Decoder Experiments

The debug experiments have been performed on an FPGA prototype of an MP3 audio decoder [45]. Figure 3.6 shows the architecture of the MP3 audio decoder. Tables 3.3, 3.4, 3.5 and 3.6 show the average data compression ratio CR_{data} and the average adjusted compression ratio CR_{adjust} for 10 songs with different replacement policies (i.e., FIFO, Random, modified LRU, modified LFU); CR_d stands for CR_{data} and CR_a stands for CR_{adjust}. These results are for direct/differential encoding of the debug data probed at the stereo decoder output of the MP3 decoder as demonstrated in Tables 3.3 and 3.4, and at the input of the polyphase filter bank before the modified discrete cosine transform (MDCT) as demonstrated in Tables 3.5 and 3.6.

The data compression ratio is calculated as $CR_{data} = (\Sigma_{i=1}^{K} \frac{W_i}{M})/K$. Where K is the total number of debug experiments (one debug experiment is considered to end

when the trace buffer is filled), M is the size of the trace buffer that equals to its width times its depth and W_i is the size of observation window i which equals to its total number of samples times the sample width. The volume of data that is reconstructed after the decompression process at the debug software side is W_i; i.e., the length of the observation window in debug experiment i weighted by the width of each sample. The relative increase in the size of the observation window for each debug experiment $\frac{W_i}{M}$ is summed for all the debug experiments and then averaged to obtain CR_{data}.

Because the compression hardware can add a considerable area cost to the size of the debug module, it could be argued that by increasing the size of the trace buffer (with the same area as the compression hardware) will lead to similar benefits in terms of increasing the observation window size. Therefore, we use the proposed compression ratio metric defined in section 3.2.2 to measure the performance gain of the proposed architecture. It is essential to note that this new metric (CR_{adjust}) captures the real compression benefit not only of the proposed solution but also of any other compression algorithm that requires on-chip hardware to aid embedded logic analysis. In other words, this metric shows that in some cases it may be cheaper to extend the size of the trace buffer than to invest in the area of the encoder.

Based on the results from Tables 3.3, 3.4, 3.5 and 3.6, we emphasize the following points:

- The average compression ratios for the direct encoding at stereo decoder output and at MDCT input are better than those achieved through differential encoding. This implies that the debug data at these probe points show more correlation among successive symbols than the correlation among the differences of successive symbols (i.e., the symbols are occurring more frequently than the differences between successive symbols).

- For the direct encoding of the debug data at stereo decoder output, the smallest CAM depth gives better compression ratios than the case of the direct encoding of MDCT input data. This implies that the data at stereo decoder output shows more correlation than the data observed at MDCT input. Therefore, the achieved compression ratios are dependent on the symbols correlation *over*

Table 3.3: Average Compression Ratios of Encoding MP3 Data (12×2^{20} samples) at Stereo Decoder Output using BSTW Algorithm, M=16k Bytes

	Rep. Policy	CAM Depth	BSTW					
			1-Symbol		2-Symbol		3-Symbol	
			CR_d	CR_a	CR_d	CR_a	CR_d	CR_a
Direct Encoding	FIFO	16	2.73	2.69	2.39	2.34	2.00	1.95
		64	2.21	2.15	2.26	2.16	1.99	1.87
		256	1.85	1.72	2.20	1.93	2.03	1.69
		1024	1.57	1.24	2.13	1.40	2.07	1.17
	RAN	16	2.73	2.69	2.39	2.34	2.00	1.95
		64	2.21	2.15	2.26	2.16	1.99	1.87
		256	1.85	1.72	2.20	1.93	2.03	1.69
		1024	1.57	1.24	2.13	1.40	2.07	1.17
	LRU	16	2.74	2.69	2.40	2.35	2.01	1.95
		64	2.21	2.14	2.26	2.15	2.00	1.87
		256	1.85	1.70	2.20	1.91	2.03	1.67
		1024	1.57	1.23	2.13	1.39	2.07	1.16
	LFU	16	2.74	2.69	2.42	2.36	2.02	1.96
		64	2.21	2.12	2.27	2.14	2.01	1.87
		256	1.85	1.67	2.21	1.88	2.04	1.65
		1024	1.57	1.20	2.14	1.37	2.08	1.15
Differential Encoding	FIFO	16	2.14	2.11	1.97	1.93	1.78	1.73
		64	1.88	1.83	1.89	1.81	1.77	1.66
		256	1.71	1.59	1.86	1.63	1.80	1.50
		1024	1.53	1.21	1.84	1.21	1.82	1.03
	RAN	16	2.14	2.11	1.97	1.93	1.78	1.73
		64	1.88	1.83	1.89	1.80	1.77	1.66
		256	1.71	1.59	1.86	1.63	1.80	1.50
		1024	1.53	1.21	1.84	1.21	1.82	1.03
	LRU	16	2.14	2.10	1.96	1.92	1.78	1.73
		64	1.88	1.82	1.89	1.80	1.77	1.66
		256	1.71	1.57	1.86	1.62	1.79	1.47
		1024	1.53	1.20	1.83	1.20	1.82	1.02
	LFU	16	2.15	2.11	1.96	1.91	1.78	1.73
		64	1.90	1.82	1.90	1.79	1.78	1.65
		256	1.71	1.54	1.88	1.60	1.82	1.47
		1024	1.53	1.17	1.85	1.19	1.83	1.01

Table 3.4: Average Compression Ratios of Encoding MP3 Data (12×2^{20} samples) at Stereo Decoder Output using MBSTW and WDLZW Algorithms, M=16k Bytes

	Rep. Policy	CAM Depth	MBSTW				WDLZW			
			2-Symbol		3-Symbol		2-Symbol		3-Symbol	
			CR_d	CR_a	CR_d	CR_a	CR_d	CR_a	CR_d	CR_a
Direct Encoding	FIFO	16	3.01	2.95	3.07	3.00	3.20	3.13	3.40	3.30
		64	2.64	2.55	2.78	2.68	3.00	2.88	3.37	3.20
		256	2.29	2.12	2.46	2.27	2.76	2.49	3.20	2.82
		1024	1.98	1.56	2.16	1.69	2.54	1.82	3.06	2.08
	RAN	16	3.00	2.94	3.04	2.97	3.20	3.13	3.41	3.31
		64	2.62	2.53	2.74	2.64	3.00	2.88	3.37	3.20
		256	2.28	2.11	2.44	2.25	2.76	2.48	3.20	2.81
		1024	1.98	1.56	2.15	1.69	2.54	1.82	3.06	2.08
	LRU	16	3.00	2.94	3.05	2.98	3.22	3.15	3.43	3.32
		64	2.62	2.52	2.74	2.63	3.00	2.87	3.36	3.18
		256	2.28	2.09	2.44	2.23	2.76	2.47	3.20	2.80
		1024	1.98	1.54	2.15	1.67	2.54	1.81	3.06	2.07
	LFU	16	3.01	2.94	3.07	2.99	3.22	3.14	3.45	3.32
		64	2.63	2.51	2.75	2.62	3.02	2.88	3.39	3.18
		256	2.30	2.06	2.47	2.21	2.77	2.43	3.20	2.76
		1024	1.99	1.51	2.18	1.65	2.56	1.79	3.08	2.05
Differential Encoding	FIFO	16	2.36	2.31	2.35	2.30	2.27	2.22	2.36	2.29
		64	2.22	2.15	2.27	2.19	2.29	2.20	2.47	2.34
		256	2.04	1.89	2.12	1.96	2.25	2.03	2.51	2.21
		1024	1.82	1.43	1.91	1.50	2.24	1.61	2.60	1.77
	RAN	16	2.33	2.28	2.31	2.26	2.27	2.22	2.36	2.29
		64	2.18	2.11	2.21	2.13	2.29	2.20	2.47	2.34
		256	2.01	1.86	2.08	1.92	2.25	2.02	2.51	2.21
		1024	1.80	1.41	1.89	1.48	2.24	1.60	2.60	1.77
	LRU	16	2.33	2.28	2.30	2.25	2.27	2.22	2.37	2.30
		64	2.18	2.10	2.21	2.12	2.29	2.19	2.47	2.34
		256	2.01	1.84	2.08	1.90	2.25	2.01	2.51	2.20
		1024	1.80	1.40	1.89	1.47	2.24	1.59	2.60	1.76
	LFU	16	2.35	2.29	2.33	2.27	2.28	2.22	2.37	2.28
		64	2.20	2.10	2.23	2.13	2.31	2.20	2.49	2.34
		256	2.04	1.83	2.12	1.90	2.26	1.99	2.52	2.17
		1024	1.83	1.39	1.94	1.47	2.27	1.59	2.62	1.75

Table 3.5: Average Compression Ratios of Encoding MP3 Data (12×2^{20} samples) at MDCT Input using BSTW Algorithm, M=16k Bytes

	Rep. Policy	CAM Depth	BSTW					
			1-Symbol		2-Symbol		3-Symbol	
			CR_d	CR_a	CR_d	CR_a	CR_d	CR_a
Direct Encoding	FIFO	16	1.27	1.25	1.28	1.25	1.22	1.19
		64	1.31	1.27	1.26	1.20	1.22	1.15
		256	1.37	1.27	1.25	1.10	1.20	1.00
		1024	1.39	1.10	1.25	0.82	1.18	0.67
	RAN	16	1.27	1.25	1.28	1.25	1.22	1.19
		64	1.31	1.27	1.26	1.20	1.22	1.15
		256	1.37	1.27	1.25	1.10	1.20	1.00
		1024	1.39	1.10	1.25	0.82	1.18	0.67
	LRU	16	1.27	1.25	1.28	1.25	1.22	1.19
		64	1.31	1.27	1.26	1.20	1.22	1.14
		256	1.38	1.27	1.26	1.10	1.20	0.99
		1024	1.40	1.09	1.25	0.82	1.19	0.67
	LFU	16	1.31	1.28	1.30	1.27	1.25	1.21
		64	1.35	1.30	1.28	1.21	1.24	1.15
		256	1.40	1.26	1.27	1.08	1.21	0.98
		1024	1.41	1.08	1.27	0.81	1.19	0.66
Differential Encoding	FIFO	16	1.17	1.15	1.21	1.18	1.17	1.14
		64	1.19	1.16	1.19	1.14	1.17	1.10
		256	1.24	1.15	1.17	1.03	1.15	0.96
		1024	1.30	1.03	1.17	0.77	1.13	0.64
	RAN	16	1.17	1.15	1.21	1.18	1.17	1.14
		64	1.19	1.16	1.19	1.14	1.17	1.10
		256	1.24	1.15	1.17	1.03	1.15	0.96
		1024	1.30	1.03	1.17	0.77	1.13	0.64
	LRU	16	1.18	1.16	1.21	1.18	1.18	1.15
		64	1.19	1.15	1.19	1.13	1.17	1.10
		256	1.25	1.15	1.18	1.03	1.16	0.96
		1024	1.31	1.02	1.17	0.76	1.14	0.64
	LFU	16	1.20	1.18	1.22	1.19	1.19	1.15
		64	1.23	1.18	1.21	1.14	1.19	1.11
		256	1.28	1.15	1.19	1.01	1.16	0.94
		1024	1.31	1.00	1.19	0.76	1.15	0.63

Table 3.6: Average Compression Ratios of Encoding MP3 Data (12×2^{20} samples) at MDCT Input using MBSTW and WDLZW Algorithms, M=16k Bytes

	Rep. Policy	CAM Depth	MBSTW				WDLZW			
			2-Symbol		3-Symbol		2-Symbol		3-Symbol	
			CR_d	CR_a	CR_d	CR_a	CR_d	CR_a	CR_d	CR_a
Direct Encoding	FIFO	16	1.42	1.39	1.37	1.34	1.26	1.23	1.29	1.25
		64	1.54	1.49	1.50	1.45	1.30	1.25	1.34	1.27
		256	1.60	1.48	1.57	1.45	1.42	1.28	1.47	1.29
		1024	1.55	1.22	1.54	1.21	1.59	1.14	1.66	1.13
	RAN	16	1.42	1.39	1.36	1.33	1.26	1.23	1.29	1.25
		64	1.54	1.49	1.50	1.45	1.30	1.25	1.34	1.27
		256	1.60	1.48	1.57	1.45	1.42	1.28	1.47	1.29
		1024	1.55	1.22	1.54	1.21	1.58	1.13	1.66	1.13
	LRU	16	1.42	1.39	1.36	1.33	1.26	1.23	1.29	1.25
		64	1.54	1.48	1.50	1.44	1.31	1.26	1.34	1.27
		256	1.61	1.47	1.59	1.45	1.43	1.28	1.47	1.29
		1024	1.56	1.21	1.55	1.20	1.59	1.13	1.67	1.13
	LFU	16	1.46	1.42	1.41	1.37	1.28	1.25	1.30	1.25
		64	1.59	1.52	1.56	1.49	1.38	1.31	1.41	1.32
		256	1.64	1.47	1.61	1.44	1.50	1.32	1.54	1.33
		1024	1.58	1.20	1.57	1.19	1.63	1.14	1.72	1.15
Differential Encoding	FIFO	16	1.31	1.28	1.27	1.24	1.18	1.16	1.20	1.16
		64	1.41	1.36	1.38	1.33	1.18	1.13	1.21	1.15
		256	1.48	1.37	1.45	1.34	1.27	1.14	1.30	1.14
		1024	1.47	1.16	1.46	1.15	1.41	1.01	1.46	0.99
	RAN	16	1.31	1.28	1.26	1.23	1.18	1.16	1.20	1.16
		64	1.41	1.36	1.37	1.32	1.18	1.13	1.21	1.15
		256	1.48	1.37	1.45	1.34	1.27	1.14	1.30	1.14
		1024	1.47	1.16	1.46	1.14	1.41	1.01	1.46	0.99
	LRU	16	1.31	1.28	1.26	1.23	1.18	1.15	1.20	1.16
		64	1.41	1.36	1.37	1.32	1.18	1.13	1.21	1.14
		256	1.49	1.36	1.46	1.33	1.27	1.13	1.31	1.15
		1024	1.48	1.15	1.47	1.14	1.42	1.01	1.47	1.00
	LFU	16	1.33	1.30	1.29	1.26	1.18	1.15	1.20	1.16
		64	1.45	1.39	1.41	1.34	1.22	1.16	1.25	1.17
		256	1.52	1.36	1.49	1.33	1.33	1.17	1.37	1.18
		1024	1.49	1.13	1.49	1.13	1.47	1.03	1.52	1.01

short intervals of the observation window, as explained in Section 3.4.1. In the case of encoding the debug data at stereo decoder output, larger CAM depth does not bring any compression benefits because the larger the CAM depth, the larger the width of the codeword while the input data shows high correlation over only few entries in the CAM.

- The achieved results show that by increasing CR_{data} through using larger CAM depth in few cases, does not necessarily lead to improving CR_{adjust} (e.g., In the case of the differential encoding of MDCT input, using a CAM depth 1024 does not improve CR_{adjust}). This demonstrates that increasing the CAM depth beyond a certain limit (which is dependent on both the size of trace buffer and how the input symbols occur frequently over short intervals), will not bring any compression benefits.

- The results of the multiple-symbol encoding for the MBSTW algorithm are better than the ones of using multiple-symbol BSTW algorithm. This difference is because the mismatches in the case of multiple-symbol dictionary BSTW influence the compression ratio more than the case of using a single-symbol dictionary MBSTW, as discussed in Section 3.4.1. In addition, the impact of the area of the multiple-symbol BSTW dictionary architectures on CR_{adjust} is higher than the area required by the proposed single-symbol MBSTW dictionary. This area reduction (and hence CR_{adjust} improvement) was the justification for the development of the proposed MBSTW algorithm.

- The average compression ratios achieved using WDLZW algorithm are better than the ones achieved using MBSTW algorithm for highly correlated data as observed from the direct encoding results of stereo decoder output in Table 3.4. This is because the 2-symbol (or 3-symbol) dictionary in the case of WDLZW algorithm contains any 2 (or 3) successively matched symbols. However, MBSTW algorithm provides better results than WDLZW algorithm for less correlated data as manifested from the differential encoding of MDCT input results (Table 3.6) due to the following two reasons. First, in the case of MBSTW, the mismatch is represented by a 2 or 3 bit flag but in the case of WDLZW, it is

represented by a codeword 0 that has a width larger than the flag information. Second, the single-symbol dictionary depth in the case of MBSTW algorithm is larger than the single-symbol dictionary depth used in WDLZW algorithm.

- There is no difference between FIFO and random replacement policies on the compression ratios when using the BSTW or WDLZW encoding algorithm. This implies that regardless where the data will be replaced in the CAM, the compression ratios will not be influenced because both FIFO and random replacement policies will enter into repeating cycles after all the symbols are replaced from the CAM (i.e., each of these replacement policies restarts from its initial address location).

- The FIFO replacement policy of the MBSTW encoding algorithm shows better compression ratios than the random replacement policy for highly correlated data and smaller CAM sizes, as observed from the direct encoding of the stereo decoder output in Tables 3.3 and 3.4. This is because the MBSTW algorithm is based on the correlations between two successive symbols or among three successive symbols but the random replacement policy influences how the successive symbols are placed in the CAM.

- Based on the previous two emphasized points, the modified LRU or the modified LFU replacement policy will have similar influence on the data compression ratios CR_{data} for highly correlated data, as observed from the direct encoding of the stereo decoder output in Tables 3.3 and 3.4. Therefore, the achieved CR_{adjust} of the FIFO replacement policy is better than the one achieved using either the modified LRU or modified LFU replacement policy.

- The achieved results using the modified LFU replacement policy are better than those achieved using the modified LRU replacement policy for less correlated data, as observed from Tables 3.5 and 3.6. This implies that in the case of the modified LFU replacement policy, the most frequently accessed segments contain symbols that are occurring more frequently than the symbols that are existed in the most recently used segments in the case of the modified LRU

replacement policy.

- The data compression ratios CR_{data} of the modified LFU replacement policy are better than those of FIFO and random replacement policies for encoding the debug data at MDCT input, as observed from Tables 3.5 and 3.6. However, the achieved adjusted compression ratios CR_{adjust} of the modified LFU replacement policy can be similar or less than those achieved using FIFO or random replacement implementation. Therefore, the improvement in CR_{data}, due to using a certain replacement policy, does not necessary lead to a similar improvement in CR_{adjust} because the area has been taken into account. Nonetheless, if a larger trace buffer is used, then the improvement in CR_{data} will be noticeable in CR_{adjust} as well.

Finally, it should be noted that in the special case of uncorrelated debug data, which is unlikely, even the dictionary-based compression algorithms do not provide an increase in the observation window. As a consequence, the encoder can be disabled and the trace buffer can be used to capture the debug data without any compression. This case was not observed in any of our experiments because CR_{data} is always above 1 in Tables 3.3, 3.4, 3.5 and 3.6. Nonetheless, as discussed previously in this section, CR_{adjust} can go below 1, especially for large CAM sizes. This confirms that the smaller the area of the encoder, the better the compression benefits will be, and this is the key observation specific to embedded logic analysis that leads to the development of the proposed MBSTW algorithm.

3.6 Summary

In this chapter, we have explored the suitability of different dictionary-based compression algorithms for lossless data compression in embedded logic analysis. We have analyzed the specific requirements for lossless data compression in embedded logic analysis, and subsequently, we have proposed a new compression ratio metric that captures the real benefits of a compression algorithm that needs to satisfy high-throughput/real-time encoding with an acceptable area. This metric is used to

quantify the performance gain of the proposed dictionary-based compression architectures that support the most commonly used replacement policies. The architectures proposed in this chapter are particularly suitable for in-field debugging on application boards, which have non-deterministic events that inhibit the deterministic replay of debug experiments.

Chapter 4

On Extending the Observation Window through Lossy Compression

The capacity of on-chip trace buffers employed for at-speed silicon debug limits the observation window in any debug session. To increase the debug observation window, we propose a novel architecture for at-speed silicon debug based on lossy compression. In order to accelerate the identification of the design errors, we have developed a new debug method where the designer can iteratively zoom only in the intervals that contain erroneous samples. When compared to increasing the size of the trace buffer, the proposed architecture has a small impact on silicon area, while significantly reducing the number of debug sessions. The proposed method is applicable to both automatic test equipment-based debug and in-field debug on application boards, so long as the debug experiment can be reproduced synchronously.

This chapter is organized as follows. Section 4.1 gives preliminaries for the research work presented in this chapter, outlines the motivation behind it and summarizes its contributions. Sections 4.2 and 4.3 describe the proposed debug framework and the proposed debug architecture. Section 4.4 introduces the algorithms for scheduling debug sessions, while Section 4.5 discusses the sensitivity of the proposed method to the failing samples distribution. Section 4.6 shows the experimental results for an MP3 decoder hardware and Section 4.7 concludes the chapter.

4.1 Preliminaries and Summary of Contributions

As discussed in Chapter 3, the choice of the depth compression technique relies on the type of the silicon debug experiment. For the first phase of the validation process at which the bug appears on an application board and its occurrence cannot be reproduced in a deterministic manner due to the existence of *non-deterministic input sources*, we proposed a novel debug architecture that supports real-time lossless data compression in Chapter 3. Once the input behavior that causes the bug to manifest itself is identified, a subsequent validation phase can be applied, where the bug can be triggered deterministically by controlling the inputs. In the second phase the debug experiment becomes deterministic in the sense that the same behavior can be reproduced consistently either on an ATE or on an application board. To extend the observation window for the latter validation phase, we propose a novel debug architecture based on lossy compression in this chapter.

By extending the silicon debug observation window using a short sequence of debug sessions, the proposed approach is useful in aiding the identification of hard-to-detect functional bugs that occur intermittently over a long period of time [57], which is computationally-infeasible to be simulated during pre-silicon verification. The main contributions of this chapter are as follows:

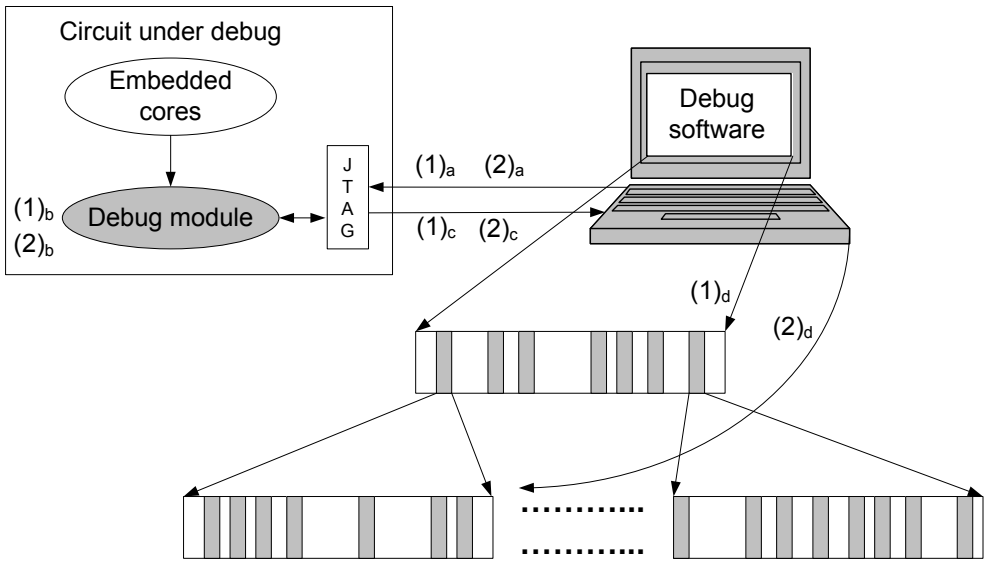
- by employing a signature register placed in front of the trace buffer (with a programmable number of samples that map onto a single signature) combined with a segmentation of the trace buffer we introduce a new architecture for the embedded debug module;
- the proposed architecture enables a new debug methodology where the debug engineer can iteratively zoom only in the intervals containing erroneous samples; this leads to a significant reduction in the number of debug sessions for a large interval that needs to be observed;
- we develop new algorithms for scheduling debug sessions and we show that by leveraging the streaming feature of the low-bandwidth interface to the debug software, we can further improve the effectiveness of our solution.

4.2 Proposed Iterative Silicon Debug Framework

The method proposed in this chapter is applicable to the scenario where the debug data is known a-priori and a deterministic execution of input data will always produce the same output data. This is the case when debugging on an ATE and it is also common when having a target application board where stimuli are applied synchronously (e.g., when debugging an audio/video decoder) and thus the expected responses can be computed using a reference behavioral model of the circuit under debug (CUD). This debugging method is also referred to as *cyclic debugging* or *deterministic replay* in the software engineering literature and having an experimental testbed that supports it is a pre-requisite for the proposed iterative flow of the silicon debug process based on lossy compression.

As shown in Figure 4.1, the debug framework consists of an off-chip debugger software which interacts with the on-chip debug module through a serial interface (such as JTAG [53]). The basic intuition of our approach can be explained as follows (the specific technical details of the debug module and debugger software are given in the following sections). In cyclic debugging, by re-running the same experiment deterministically, the use of different trigger points in each debug session will facilitate the reconstruction of the circuit behavior over a long observation window. However, in order to avoid sampling data that does not carry any failing (erroneous) information, after the first experiment, it would be useful if the user can learn which *intervals* are error-free. Therefore, at the center of the proposed debug methodology is *lossy compression* through which we map a sequence of samples (determined by the interval length) into one *signature*. If a signature is error-free, then, in the remaining debug sessions, the interval whose sequence of samples maps onto the respective signature, will be skipped and no samples will be extracted from it. The length of the targeted observation window and the size of the on-chip trace buffer will determine the starting *compression level* for the initial debug session, which equals the length of the intervals that will be processed one at a time in each of the following debug sessions.

The cyclic debug process starts by uploading the debug module with the debug configuration (e.g., trigger event condition, the trigger signal selection, debug data



Step (1) Initial Debug Session
a. Upload debug module with debug configuration which includes the compression level
b. Run debug experiment
c. Transfer trace buffer signatures to debug software
d. Determine trigger pointers of the failing signatures

Step (2) Iterative Debug Sessions
a. Upload debug module with debug configuration which includes the trigger pointers
b. Run debug experiment
c. Transfer trace buffer samples to debug software
d. Determine the failing samples

Figure 4.1: The Iterative Flow of the Silicon Debug Process Based on Compression and Cyclic Debugging

selection). The user can specify the targeted observation window whose length divided by the trace buffer depth will give the initial compression level. This value will be uploaded together with the debug configuration in the initial debug session as illustrated with step (1) in Figure 4.1. When running the debug experiment for the first time and after the trigger event occurs, the compressed signatures of the sequences of samples will be stored in the trace buffer. Once the trace buffer is filled with the signatures (for all the intervals in the observation window), the initial debug session is completed and the trace buffer content is offloaded to the debugger software, where the failing signatures (and hence intervals) are identified. In the following debug sessions (Step (2) from Figure 4.1), the user can set up the triggering events such that each debug session will *zoom* in only into specific failing interval (more failing intervals can be sampled in one debug session if the trace buffer has multiple segments, as explained in the following section). This process is repeated *iteratively* in the succeeding debug sessions until all the failing intervals are extracted. As shown in the figure, both steps (1) and (2) consist of 4 consecutive sub-steps: uploading the debug configuration, running the debug experiment, transferring the debug data to debug software and analyzing failing samples. The benefits of our proposed method stem from the fact that we can observe the failing behaviors within a long observation window without wasting too many debug sessions for sampling the error-free intervals.

As stressed earlier the proposed silicon debug framework requires an experimental testbed that supports cyclic debugging, which is facilitated by both ATEs and by most target application boards (for example, in our experiments when bringing up multimedia designs where stimuli are applied synchronously). In addition, for both special-purpose and generic trace buffer-based debug methods, the proposed approach will further extend the observation window while supporting the existing depth or width compression techniques, which are *complementary* to our method. Furthermore, it can also assist the scan-based debug methods by pin-pointing a few sparse failures over a large observation window, where scan dumps and latch divergence analysis need to be done. Also, by analyzing how the failing intervals are related to each other, the user can narrow down the logic block that causes the erroneous behavior. For example, if all the failing intervals are happening when the circuit is in a particular configuration

mode, then this is an indication that the logic for that particular configuration mode is a suspect.

It is important to note that deterministic replay (or cyclic debugging) is not an assumption that is specific only to the work presented in this chapter. Because eliminating the non-determinism can improve the ability to debug any complex device (even without compression), several techniques have been developed to ensure determinism when searching for the root cause of failure. For example, to eliminate the non-determinism caused by I/O devices, a buffering module can be used to record the input data and its time stamps as described in [94]. In this technique, when replaying the execution, the I/O devices can be temporarily suspended and the buffer will *deterministically* reproduce the input stimuli that have been recorded. Another technique with higher hardware complexity has been employed to record the external activity from I/O pins of the Intel Pentium M processor [96]. This technique has been successfully deployed to reproduce the system failures deterministically.

4.3 Debug Architecture for Lossy Compression

This section describes the proposed debug architecture which facilitates the iterative debug flow described in the previous section. The main contributions in this debug architecture are shaded in Figure 4.2 and the terminology is given in Table 4.1. First, we describe the main features in the embedded debug module which is used for at-speed silicon debug. Second, we introduce the distinguishing features of our work and explain their usage to extend the observation window through an illustrative example.

4.3.1 The Embedded Debug Module

We first overview the basic blocks in an embedded debug module. The essential feature of an embedded debug module is the ability to detect a certain event. Therefore, an event detector is used to monitor a group of trigger signals to determine when the debug data signal is captured in the trace buffer, as shown in Figure 4.2. In our implementation, triggering can be performed based on bitwise, comparison or logical

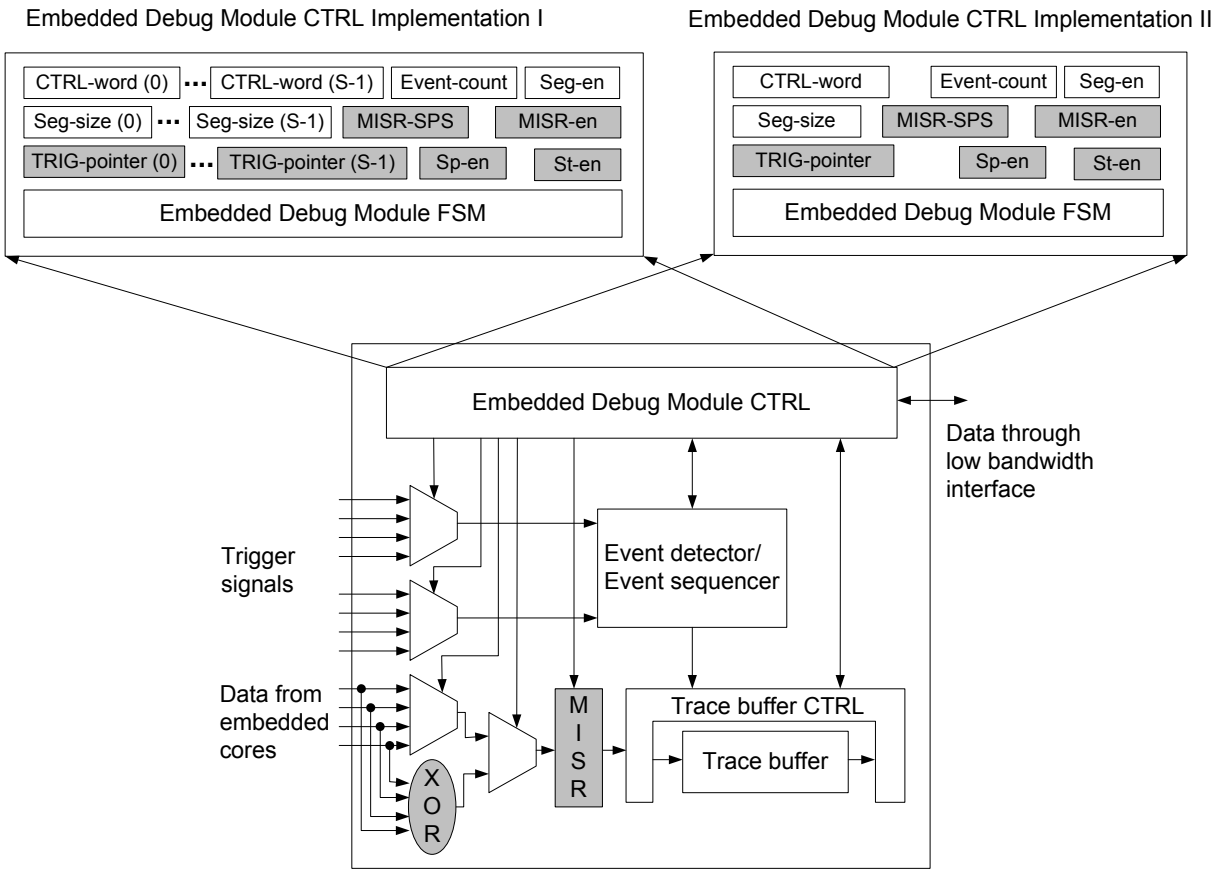


Figure 4.2: The Proposed Embedded Debug Module Architecture

operations between any selected trigger signal and a specified constant value. The control words ($CTRL - word$) specify the trigger signal selection, the trigger events, and the choice of signals that need to be probed. The event detector, which contains many levels of logic, is pipelined in order to support high-speed sampling. To further enhance the detection ability, the event detector is followed by an event sequencer to monitor a specified sequence of events determined by the control words. The configuration of the control registers are uploaded to the embedded debug module control through the low bandwidth interface (e.g., JTAG).

Table 4.1: Terminology for the Proposed Embedded Debug Module

Name	Representation
Sp-en	Spatial enable
St-en	Stream enable
MISR-en	MISR enable
CTRL-word	Control word
TRIG-pointer	Trigger pointer
Seg-size	Segment size
Seg-en	Segmented mode enable
Event-count	Number of trigger events
MISR-SPS	Number of samples per signature

4.3.2 Features to Extend the Observation Window

Figure 4.2 shows the proposed debug architecture which has a segmented trace buffer. The basic principle of a segmented trace buffer was introduced in Chapter 2. In the proposed architecture, the number of control words equals the number of segments and they can be used in different ways based on the $Event-count$ and $Seg-en$ fields. If both of them are zero then the trace buffer is unsegmented and $CTRL-word(0)$ provides the condition for triggering. If $Event-count$ is set to zero and $Seg-en$ is active then segment i starts sampling at the event defined by $CTRL-word(i)$. If $Event-count$ is not zero and $Seg-en$ is deactivated then the trace buffer is unsegmented and the event sequencer monitors the conditions described in the first $Event-count$ control words in order to start sampling. If the $Event-count$ is not zero and $Seg-en$ is active then the triggering is sequential as described above, nonetheless the trace buffer is segmented and each segment i starts sampling $Trigger-pointer(i)$ clock cycles after the event detection and the number of samples in each segment is specified by $Seg-size(i)$.

As shown in the embedded debug module CTRL implementation I from Figure 4.2, S control registers (i.e., S control words, S trigger pointers and S segments sizes) are allocated in the embedded debug module control. As the number of segments increases, the area of the debug module can become large. Since these control registers are accessed in sequence, an alternative is to store the values of these S control words,

S trigger pointers and S segment sizes into the last few locations of the trace buffer. These locations can be accessed through the embedded debug module control that now contains only one control register as shown in the embedded debug module CTRL implementation II from Figure 4.2. Although these will consume several locations from the trace buffer, because the debug data will be captured *after* the occurrence of the trigger conditions that are specified by these control data, the captured data can overwrite the control information. So long as the size of the last segment is larger than the amount of control information (i.e., control word, trigger pointer and segment size) stored in the trace buffer for this last segment (which is the case for most practical implementations), the length of the observation window will not be affected.

The distinguishing features of our architecture, stem from the multiple input signature register (MISR), commonly used in built-in self-test (BIST), which is placed at the input of the trace buffer. This MISR performs lossy compression on the selected debug data signals with a compression level determined by the *samples per signature* ($MISR-SPS$) parameter, whenever $MISR-en$ is enabled. At the beginning of the debug session, the MISR counter (which is placed in the controller of the debug module) is initialized to zero. After the trigger condition occurs, the MISR counter starts counting and it resets back to zero each time it reaches $MISR-SPS$, at which time the signature is written in the trace buffer. The number of signatures stored in each segment is determined by the segment size $Seg-size$. When all the failing intervals (whose length is determined by the size of a segment) have been identified, the compression does not need to be performed any more and $MISR-en$ is deactivated.

Example 1 Figure 4.3 shows an illustrative example of how the above-described features can be used by an hierarchical debug flow. In the highest level of debug hierarchy (level 0) samples from the entire targeted observation window are mapped onto signatures and stored in the trace buffer. In the subsequent levels of debug hierarchy only samples from the intervals represented by the failing signatures (in the level of debug hierarchy above the one that is currently considered) are acquired (either compressed or uncompressed depending whether it is the last level of debug hierarchy or not). Table 4.2 gives the terminology which is used both by this example

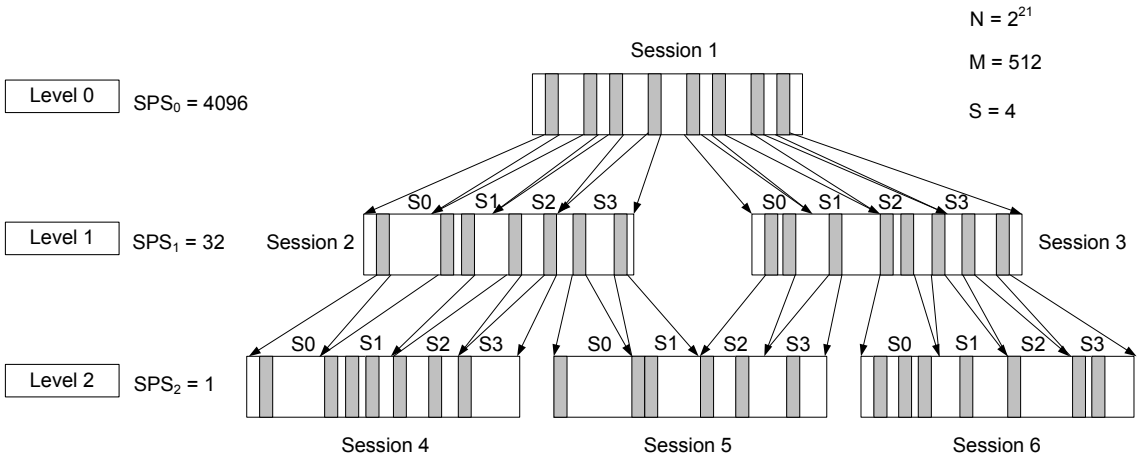


Figure 4.3: An Iterative Debug Example for the Proposed Iterative Debug Flow using Lossy Compression

and in the following sections.

The trace buffer has $M = 512$ locations, $S = 4$ segments, and the targeted observation window is $N = 2^{21}$ samples. Debug level 0 has only one debug session, in which the trace buffer is unsegmented, with the compression level given by $SPS_0 = N/M = 4096$. After the initial debug session, each of the eight failing signatures covers 4096 samples, which, for this particular case, is greater than the size of the trace buffer (512 locations). Therefore, we will have an intermediate debug level 1, which will further filter-out the error-free intervals. For all the subsequent debug sessions, the trace buffer segments are used in such a way that instead of expanding one failing signature per debug session, S failing signatures are expanded. SPS_1 can be calculated as $SPS_0/(M/S) = 32$, and therefore another level of debug is needed to zoom in and detect when exactly the failing samples occur. In the last debug level 2, no compression will be applied ($SPS_2 = 1$). To better utilize the available trace buffer size, before running any session in debug level 2, the debugger software will check if any neighboring signatures from level 1 can be merged such that more than one signature can be expanded into one segment in debug level 2. In Figure 4.3, two of the failing signatures from session 2 at debug level 1 can be expanded in one segment in session 4 at debug level 2. Finally, as observed in the figure, when using

Table 4.2: Terminology for the Proposed Debug Method

Name	Representation
M	Trace buffer depth
N	The length of the observation window
S	The number of segments in the trace buffer
k	The index of the last level of debug
SPS_i	Samples per signature at debug level i
DS_i	Number of debug sessions at debug level i
FS_i	Number of failing signatures at debug level i
DS_{CUD}	Total number of debug sessions for the CUD
T_{CUD}	Time for running debug sessions on the CUD

lossy compression over a long observation window, the number of debug sessions for the above example is as low as 6. For this example, because the number of erroneous intervals is only 12 (it equals the number of segments with failing samples in the last debug level), there is a significant reduction in debug sessions when compared to the sequential debug case (where the number of debug sessions would be equal to $N/M = 4096$).

As observed from this example, the proposed method can be considered as a generalization of the binary search approach that is used in scan-based BIST diagnosis [40]. This fault diagnosis method relies on the binary search approach for determining the error-capturing scan cells by iteratively applying the generated test patterns to the scan chains and capturing the signature of selected scan cells. The selection of the scan cells in each BIST session is decided based on whether or not the captured signature from the previous BIST session is faulty. The binary search is continued until all the scan cells are diagnosed or a desired accuracy is achieved. To emphasize the contributions of our method, we explain the differences between the known scan-based BIST diagnosis method [40] and the proposed method from our work which focuses on reducing the number of debug experiments during in-system silicon debug.

A key observation is that the length of the observation window, which is usually targeted in silicon debug, can be significantly larger than the length of the internal scan chains. Therefore collecting only one signature per debug session will be too

lengthy. Because the trace buffer is already available on-chip to collect the fault-free samples in the last debug session, we reuse it in order to capture a stream of signatures. Consequently, as many signatures as the trace buffer depth can be extracted and analyzed to determine the intervals which contain erroneous samples. This leads to a *multiple interval search* algorithm for scheduling debug sessions that replaces the known binary search used in scan BIST diagnosis (this scheduling is also dependent whether the trace buffer can be offloaded at the same time as the debug session is running, as discussed in the following section). Thereafter, the multiple interval search is performed in the succeeding debug sessions by expanding each of the failing signatures into multiple signatures (with lower number of samples per signature) within a single segment at the subsequent level of debug. It is the segmented trace buffer feature that enables multiple interval search, which is continued until all the failing intervals are identified at the last level of debug.

A novel debug technique has been recently proposed in [123], as a follow up to our research work presented in this chapter. The differences between the introduced method in [123] and the proposed method from our work are as follows:

- the method introduced in [123] is based on a three pass methodology. In the first pass, the trace buffer is employed to store the parity information of the debug data, then this data is compared with the fault-free parity information obtained from simulation in order to measure an approximate error rate. During the second pass, a set of suspect clock cycles where errors may be present is determined through a two dimensional (2-D) compaction technique using a combination of MISR signatures and cycling register signatures. In the third debug pass, the trace buffer captures only during the suspect clock cycles. The advantage of this technique lies in expanding the observation window by one to two orders of magnitude for low error rate using only 3 debug sessions.

- in our proposed method, we set the length of the observation window at the initial debug session, and then we start to zoom into the erroneous intervals using a sequence of debug sessions. In order to target the same length of the long observation window, a multiple of 3 debug sessions will be needed using the

method introduced in [123]. As explained above, out of these 3 debug sessions only the last one will capture uncompressed samples. Hence for [123], if the error rate is high and the errors occur in bursts, there will be a high overhead of debug sessions (2 out of 3) that do not capture error-free samples. This overhead will be lower in our case, because the number of debug sessions that capture compressed samples will have a large value for samples per signature (it changes from one level to another by a factor of M/S). On the other hand, if the error rate is lower and the errors do not occur in bursts, the method from [123] will achieve a lower number of debug sessions for the same target size of the observation window. It is worth mentioning, however, that due to the tag information uploaded in the buffer in the last session of [123], the number of samples that will be captured in this last session is predetermined. Hence, it is cumbersome to include a streaming feature that can further reduce the number of debug sessions, as it is possible in our work (this will be discussed in the next section).

4.4 The Proposed Scheduling Algorithms

This section describes the proposed debug algorithms for scheduling the debug sessions. To support further reductions in the debug sessions, we introduce three important features in the architecture shown in Figure 4.2. The first feature shows the benefit of using variable segments sizes at the last level of debug. In the second feature, we introduce a spacial compactor before the MISR in order to reduce the number of debug sessions when probing multiple signals. The third feature discusses how streaming the captured signatures as the debug session is running will further improve the effectiveness of our proposed methodology.

4.4.1 Algorithm for Scheduling Debug Sessions

Algorithm 3 discusses how the compression level is updated in each debug level, as well as it shows how the total number of debug sessions (DS_{CUD}) and the time for

Algorithm 3: Scheduling Debug Sessions

Input : M, S, SPS_0 and FS_0
Output : DS_{CUD} and T_{CUD}

```
1  DS_CUD = 1; T_CUD = N; i = 0 (set the initial debug level);
2  while (last debug level not reached) do
3    if (SPS_i > M) then
4      SPS_{i+1} = SPS_i/(M/S) (set SPS for the the next debug level);
5    elseif (SPS_i > M/S) then
6      SPS_{i+1} = 1 and do segment merging;
     else
7      SPS_{i+1} = 1 and check for failing signatures merging;
     end
8    Increment the debug level (i++)
9    while (more failing signatures exist in the current debug level) do
10     Generate S trigger pointers;
11     Run debug experiment on the CUD;
12     Detect failing signatures or samples in the debugger software;
13     Update DS_CUD, T_CUD;
     end
   end
14 return DS_CUD, T_CUD;
```

running all the debug sessions (T_{CUD}) are computed. It is assumed that all the S segments are equal in size (i.e., M/S). We start with an initial debug session that compresses the entire observation window using $SPS_0 = N/M$. At this time DS_{CUD} is set to 1 and T_{CUD} is equal to N, i.e., the length of the first debug session is the entire observation window (line 1). The segmented mode feature is used thereafter, one signature from level i is mapped to one segment at level $i+1$. Each iteration from the outer loop (lines 2 to 13) stands for one debug level. The update of SPS can be explained as follows (lines 3 to 7). If the compression level at the current level i is greater than the trace buffer size M, then $SPS_{i+1} = SPS_i/(M/S)$. If SPS_i is in between M/S and M then $SPS_{i+1} = 1$ (i.e., we move to the last debug level that does not have any compression) and *segment merging* can be done, where instead of S segments we will have $SPS_i/(M/S)$ segments. The intuition behind this segment merging step is to avoid an intermediate debug level that will not increase the resolution of the failing intervals. Finally, if SPS_i is smaller than M/S we will

move to the last debug level (i.e., $SPS_{i+1} = 1$) and check if any two neighboring signatures can be mapped onto the same segment.

After incrementing the debug level (line 8), the inner loop (lines 9 to 13) iterates through each debug session at the current debug level. The trigger pointers in each debug session are calculated based on the updated SPS and failing signatures from the previous debug level. After a new debug experiment is run on the CUD, the debugger software extracts the failing signatures (if the last debug level is not reached) or samples (if the last debug level is reached) and it updates the DS_{CUD} and T_{CUD} accordingly, which will be returned at the end of the algorithm (line 14).

4.4.2 Variable Segments Sizes at the Last Debug Level

In Section 4.4.1, we have assumed that each of the segments of the trace buffer has a *fixed* size, which is equal to M/S. If $SPS_{k-1} < M/S$, multiple failing signatures at level $k-1$ (where k is the index of the last debug level) can be mapped onto the same segment at level k (line 7 of Algorithm 3). However, if the debug architecture supports *variable* segment sizes then the number of debug sessions in the last level can further be reduced. For this particular feature, the minimum segment size is SPS_{k-1}, while the maximum segment size is set up to be a multiple of SPS_{k-1}. In Figure 4.2, $Seg - size(i)$ is the variable segment size for segment i. This variable segments sizes feature is only exploited at the last debug level k. Given that there are S segments, the size of the maximum segment is selected in order to employ the trace buffer of at most S segments. Some of the S segments can have a maximum segment size and each of the remaining ones will have a size smaller than the maximum segment size. In Figure 4.3, the maximum segment size is selected to be five times SPS_{k-1} and hence three segments each with this size along with one more segment of size equals SPS_{k-1} can be used in any debug session at the last level of debug. Thus, because small segments (that store isolated failures) can be combined with large segments (that store bursts of failures) in the same debug session, further savings in terms of debug sessions can be achieved, as demonstrated later in the experimental results.

4.4.3 Spatial Compression

Figure 4.2 shows an architectural feature that is used to enable spatial compression (i.e., $Sp-en$ flag) in order to reduce the number of debug sessions of multiple probed signals. To achieve this improvement, width compression is employed before the MISR by using an XOR network in which multiple channels of debug data are compressed into a single one. This feature can be used in all the debug levels except the last one, where no compression is performed and the debug module selects only one channel at a time. The methodology for calculating the DS_{CUD} and T_{CUD} before the last debug level is the same as in Algorithm 3, however, their number at the last debug level will be scaled by the number of channels. Note, while both the spatial compressor and the MISR may run into the aliasing problem, it is unlikely that all the erroneous samples caused by a particular bug will lead to fault-free signatures in all the possible intervals of occurrence. As demonstrated later in the experimental results, the effects of aliasing (if it does occur) are negligible.

4.4.4 Combining Streaming and Compression

This subsection discusses the architectural and algorithmic support for combining streaming with the proposed debug methodology. An interesting observation is that if the length of the debug experiment is large and if the failing signatures are sparse, there is a lot of idle time in between any two trigger points in the trace buffer. Therefore, an additional architectural feature, which is enabled using stream enable flag $St-en$ shown in Figure 4.2, would enable the streaming of the samples stored in the trace buffer while the debug experiment is still running on the CUD. This feature will provide further savings in the number of debug sessions, as demonstrated later in the experimental results.

If the frequency of the off-chip serial interface is f_{JTAG} and L is the width of a word in the trace buffer, then the time needed to offload a segment to the debugger software is $Seg_{size} \times L \times (1/f_{JTAG})$, where Seg_{size} is M/S for the fixed segment case and $Seg-size(i)$ for the variable segment case. If the on-chip sampling frequency is f_{CUD}, then the time d (the number of on-chip clock cycles) needed to stream

Algorithm 4: Scheduling Debug Sessions with Streaming

Input : M, S, SPS_0 and FS_0
Output : DS_{CUD} and T_{CUD}

```
1  DS_CUD = 1; T_CUD = N; i = 0 (set the initial debug level);
2  while (last debug level not reached) do
3    if (SPS_i > M) then
4      SPS_{i+1} = SPS_i/(M/S) (set SPS for the the next debug level);
5    elseif (SPS_i > M/S) then
6      SPS_{i+1} = 1 and do segment merging;
     else
7      SPS_{i+1} = 1 and check for failing signatures merging;
     end
8    Increment the debug level (i++)
9    while (more failing signatures exist in the current debug level) do
10     Generate the last S − 1 trigger pointers;
11     Generate the first trigger pointer;
12     Run debug session on CUD;
13     while (more failing signatures satisfying streaming condition) do
14       Stream out the first segment contents;
15       Update the first trigger pointer;
16       Detect the failing signatures or samples of the streamed segment;
       end
17     Detect the failing signatures or samples of the trace buffer;
18     Update DS_CUD and T_CUD;
     end
   end
19 return DS_CUD, T_CUD;
```

out the contents of a segment must satisfy the following condition: $(1/f_{CUD}) \times d > Seg_{size} \times L \times (1/f_{JTAG})$. We define the *streaming distance* $Stream_{dist}$ as the minimum distance between two trigger points that can be mapped onto the first segment of the trace buffer, without overflowing it when the stream mode is enabled. Therefore, $Stream_{dist} = Seg_{size} \times L \times f_{ratio}$, where $f_{ratio} = f_{CUD}/f_{JTAG}$.

The first part of Algorithm 4 (lines 1 to 8), which updates SPS, is identical to Algorithm 3. The second part (the inner loop described between lines 9 to 18) exploits the streaming feature. The intuition behind the proposed algorithm is to start with the debug session that takes the longest time to run and hence enables the streaming of the largest number of the failing signatures. Then these numbers start

to decrease in the following debug sessions at the same debug level. When switching to a new debug level, we select the first failing signature and the last $S-1$ failing signatures from the previous debug level to be expanded into the first debug session at the current debug level. When running the first debug session, it is expected there will be sufficient time to stream out the contents of the first segment before the last $S-1$ trigger pointers occur. While streaming out the contents of a segment, another trigger pointer that satisfies the $Stream_{dist}$ condition can be uploaded to the debug module. The process of streaming out the first segment contents and uploading a new trigger pointer for the same segment is repeated until the last $S-1$ signatures are reached. Subsequently, after offloading the content of the trace buffer to the debugger software, the above sequence of events is repeated iteratively until all failing intervals are identified.

4.5 Sensitivity Analysis

This section analyzes the sensitivity of the proposed debug methodology to the failing samples distribution. If the entire observation window is divided into a number of intervals (bursts) and the length of each burst equals to the segment size M/S, then the total number of bursts that contain failing samples is labeled as B. Because B is directly related to the failing samples distribution in a long observation window, we analyze the sensitivity of the proposed debug methodology to B, as well as to the observation window length (N), trace buffer depth (M) and the number of trace buffer segments (S).

At debug level 0, the initial samples per signature $SPS_0 = N/M$ represents the maximum number of debug sessions (DS_{seq}) when applying the sequential debug methodology (i.e., in each debug session, we iteratively zoom into the observation window until we reach its end). In the proposed methodology, the updating criteria for the compression level at each debug level is $SPS_{i+i} = SPS_i/(M/S)$. Since $SPS_k = 1$, then $k = \lceil \log_{M/S} SPS_0 \rceil$ (for the sake of simplicity in this analysis we assume that N, M and S are powers of 2). The number of debug sessions on the last level is $DS_k = \lceil B/S \rceil$ (i.e., S bursts can be mapped onto S segments in each debug

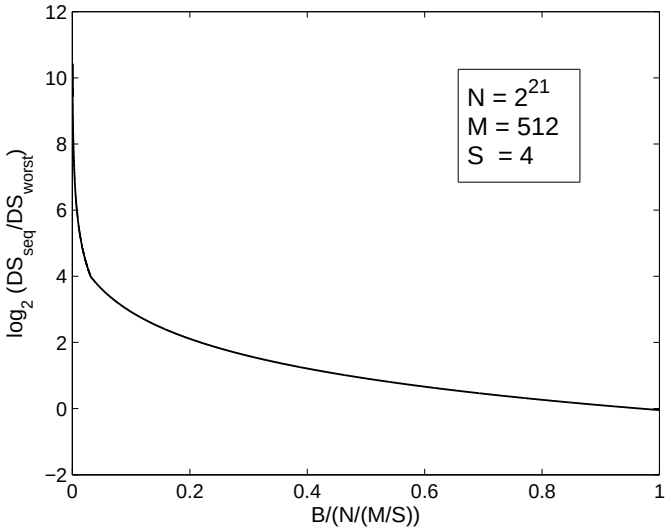


Figure 4.4: Reduction in Debug Sessions versus Failing Intervals

session) and the total number of debug sessions can be calculated as $\sum_{i=0}^{k} DS_i$. For the worst case scenario of the failing sample distribution, all the bursts are equally distributed over the entire observation window. If the number of debug sessions is equal to $\lceil B/S \rceil$ at a specific debug level j, then the number of debug sessions at each of the following debug levels will also be equal to $\lceil B/S \rceil$ (by construction the Algorithm 3 cannot decrease the number of debug sessions when moving to an upper level). Therefore, the total number of debug sessions is $\frac{(\frac{M}{S})^j - 1}{\frac{M}{S} - 1} + (k + 1 - j)\lceil \frac{B}{S} \rceil$ where $(\frac{M}{S})^{j-1} \leq \frac{B}{S} \leq (\frac{M}{S})^j$. Since from the above inequality j can be rewritten as $\lfloor \log_{\frac{M}{S}}(\frac{B}{S}) \rfloor + 1$, the maximum number of debug sessions for a given B, N, M and S can be expressed as $DS_{worst} = \frac{(\frac{M}{S})^{\lfloor \log_{\frac{M}{S}}(\frac{B}{S}) \rfloor + 1} - 1}{(\frac{M}{S}) - 1} + (\lceil \log_{\frac{M}{S}}(\frac{N}{M}) \rceil - \lfloor \log_{\frac{M}{S}}(\frac{B}{S}) \rfloor)\lceil \frac{B}{S} \rceil$. Using the above formula for DS_{worst}, which is the number of debug sessions for the proposed method for the *worst case* distribution of the failing samples for a given B, we can learn when Algorithm 3 will *guarantee* better results than in the sequential debug case DS_{seq}. Figure 4.4 considers the same values for N, M and S as in Figure 4.3 and it shows how the reduction in debug sessions (defined logarithmically as $log_2(DS_{seq}/DS_{worst})$) as a function of B. There are two main points that need

to be emphasized. Firstly, as the number of failing intervals B approaches to total number of intervals $N/(M/S)$, the advantages of our method are diminished, which is obvious because we will have to zoom in every single interval of the observation window. Secondly, for low values of B (less than 10% of the total number of intervals), which is a realistic case when searching *hard-to-detect* bugs that occur intermittently in large observation windows [57], the reduction in the number of debug sessions is *guaranteed to be significant even for the worst case distribution of B.*

4.6 Experimental Results

This section discusses the experiments concerning the area investment and the compression benefits of the proposed iterative debug method. The area of the proposed debug architecture has been estimated using a 180nm ASIC standard cell library. The debug data has been collected from an FPGA prototype of an MP3 audio decoder [45]. To show the sensitivity of the proposed method to the distribution of failing samples, we have also used random data with different failing samples distributions.

4.6.1 Area of the Proposed Debug Module

Table 4.3 shows the area of the debug module (excluding the trace buffer area) in terms of 2 input NAND (NAND2) gates. The results shown in this table are for different number of variable-sized segments ($S = 2$, $S = 4$ and $S = 8$) with the embedded debug module CTRL implementation I from Figure 4.2 and with the the embedded debug module CTRL implementation II from Figure 4.2 (that uses a user-programmable number of segments), for the following cases: no compression and no high speed sampling features; no spatial compression ($Ch = 1$) and spatial compression with different number of channels ($Ch = 2$, $Ch = 4$ and $Ch = 8$) where streaming compression is enabled and high speed sampling is facilitated by pipelining the event detector. The results from this table refer *only* to the logic area and do not account for the trace buffer. To ensure that the debug module can be shared between multiple logic cores on the SoC, we connect 8 different groups of signals to the same debug

Table 4.3: Area of The Embedded Debug Module Architecture in NAND2 Equivalents

	Segments Number	No Compression	With Compression Features			
			$Ch = 1$	$Ch = 2$	$Ch = 4$	$Ch = 8$
Implementation I in Figure 4.2	$S = 2$	2886	4970	5048	5125	5287
	$S = 4$	4559	7036	7106	7187	7344
	$S = 8$	7799	11014	11083	11165	11322
Implementation II in Figure 4.2	program-mable	2037	3970	4040	4121	4278

module. In addition, because it is important to have as many features as possible in the control word (such as compare against a constant or combine several types of logic or relational operators), the control word (which is stored for each segment) will grow in size. Should all these features be removed, the area of the debug module can be significantly reduced, nevertheless the debug capabilities will be severely limited, which cannot consider to be a good motivation.

It is essential to note that the proposed debug module architecture with the embedded debug module CTRL implementation II from Figure 4.2 uses the control register information that is stored in the last few locations of the trace buffer. This architecture contains one control word register to specify the required triggering condition, one trigger pointer register along with one segment size register. In order to support sequential event detection and multiple trigger pointers in each debug session, these registers are updated from the values that are stored into the trace buffer as explained in Section 4.3.2. As it can be noted, the area results from Table 4.3 for the debug architecture with the embedded debug module CTRL implementation II from Figure 4.2 are significantly smaller than CTRL implementation I from Figure 4.2. The different variants of the proposed architecture are obviously larger than the debug architecture that does not have any compression or high-speed features. It is essential to note that the increase in the logic area of the proposed debug module with the embedded debug module CTRL implementation II from Figure 4.2 has significantly less penalty than scaling the trace buffer. For example, for 8 channels that are spatially compressed, the added logic area for compression is still less than one

tenth of the size of an embedded memory of 4Kbytes implemented in the same technology. What is also interesting to note, as the number of channels used for spatial compression increases, the added area becomes insignificant. Nonetheless, as shown later in this section the spatial compression feature can further reduce the number of debug sessions.

4.6.2 MP3 Decoder Experiments

An MP3 decoder has been implemented and prototyped on an FPGA board. In the MP3 debug experiments performed in this work, the deterministic-replay is achieved by uploading the input stimuli to an on-board buffer. When replaying a debug experiment, the input stream is read from this buffer and applied synchronously to the circuit under debug. We have investigated how functional errors in the RTL code, can be detected using the proposed methodology. Table 4.4 shows the reduction ratios in terms of the *debug execution time* for the entire observation window, where T_{seq} and T_{prop} are the debug execution times for the sequential (i.e., where the entire observation window is sampled sequentially and no compression is performed) and the proposed debug method respectively. To compute the debug execution time we need to consider not only the number of the debug sessions, but also the *on-chip sampling time* and *the communication time* (needed for off-loading the trace buffer content to the debugger software through the JTAG interface) for *each of the debug sessions*. On the one hand, the communication time is determined by the JTAG frequency and the capacity of the trace buffer and therefore it is constant for all the debug sessions. On the other hand, for the proposed method the *on-chip sampling time* is dependent on how many on-chip clock cycles elapse from the trigger event until the trace buffer is filled only with failing intervals, which are of interest in the current debug session (e.g., in Figure 4.3 the on-chip sampling time for debug session 3 will be larger than for debug session 2). Therefore, this time varies from one debug session to another and it is dependent on the distribution of the failing samples over the observation window. It should be noted that the debugger software does not incur any additional latency because the processing of debug data can be done at the same time while

Table 4.4: Reduction in Debug Execution Time (T_{seq}/T_{prop}) for the MP3 Data with $N = 2^{21}$, $M = 512$, $S = 4$, $Ch = 2$

Error %	No Streaming				Streaming			
	No Spatial		Spatial		No Spatial		Spatial	
	Fix	Var	Fix	Var	Fix	Var	Fix	Var
0.16	49.4	58.5	58.3	71.4	216.9	239.6	256.3	288.5
1.59	7.9	9.9	8.7	11.2	99.0	108.8	106.7	118.2
6.14	3.0	3.8	3.1	4.0	78.9	80.7	83.7	85.8

the debug experiments are running on-chip and/or the data is transferred to/from the debugger software. Using the terminology introduced in Sections 4.3 to 4.5, the reduction in terms of debug execution time can be calculated as follows.

For the sequential debug case, the total number of debug sessions is $DS_{seq} = N/M$ and the communication time of the entire observation window is $DS_{seq} \times M \times L \times 1/f_{JTAG}$, where L is width of the trace buffer. The on-chip running time, in terms of on-chip clock cycles, can be approximated to $(M * \sum_{i=1}^{N/M} i) = (1 + N/M) \times N/2$ (note, the amount of time spent for detecting the first triggering event is not considered in this equation and it cannot be calculated deterministically because it depends on the specific triggering condition). As a consequence, T_{seq}=$(1 + N/M) \times N/2 \times 1/f_{CUD} + N \times L \times 1/f_{JTAG}$. The communication time for the proposed methodology is $DS_{CUD} \times M \times L \times 1/f_{JTAG}$ and the on-chip running time is $T_{CUD} \times 1/f_{CUD}$, where DS_{CUD} and T_{CUD} are computed by Algorithms 3 and 4 (note, T_{CUD} is computed in terms of on-chip clock cycles). Therefore, T_{prop}=$DS_{CUD} \times M \times L \times 1/f_{JTAG} + T_{CUD} \times 1/f_{CUD}$.

The data reported in Table 4.4 is for probing the data buses at the output of the stereo decoder module in the MP3 decoder's pipe [45] for 3 different functional bugs affecting only several stereo modes. The error choices are motivated by the fact that only a few music frames throughout an entire song will use a specific stereo mode, thus justifying the condition that the bugs are very difficult to find and they manifest themselves only a few times over very long observation windows (the error rates are .16%, 1.59% and 6.14% respectively). In this particular case, the observation window represents 1820 MP3 frames (there are 1152 samples per MP3 frame), which gives $N = 2^{21}$. With $M = 512$ and $S = 4$, there will be three debug levels with

$SPS_0 = 4096$, $SPS_1 = 32$ and $SPS_2 = 1$. To compute T_{seq} and T_{prop}, we have considered $L = 16$ (the sample width is 16 bits) and $f_{ratio} = f_{CUD}/f_{JTAG} = 2$ (The MP3 decoder has been implemented to work at low frequencies).

In Table 4.4, Fix stands for fixed segment size and Var for variable segment size. Because $SPS_1 = 32 < \frac{M}{S} = 128$, the use of the variable segment size at the last debug level leads to a further reduction in the debug execution time as described in Section 4.4.2. The spatial compression is applied on both music channels of the MP3 decoder and, as clearly shown in Table 4.4, streaming can bring substantial improvements. The significant reduction in the debug execution time when using streaming is due to the fact that by using the proposed iterative debug flow, by sampling only the failing intervals from the previous debug level, we use the time in between two intervals to stream out samples (or signatures as in the case of intermediate debug levels) that have been sampled in the same debug session. As it can be noted, the debug sessions are scheduled with the streaming feature after the initial debug session as described in Algorithm 4 from Section 4.4.4. It is important to note that these experimental results have shown only one case where the aliasing occurred. In Table 4.4, when the error rate is 6.14%, aliasing causes a loss of 0.005% of the erroneous data samples. As noted from these results, the effects of aliasing are insignificant.

4.6.3 Random Data Experiments

To illustrate the sensitivity of the proposed debug method to the distribution of the failing samples as described in Section 4.5, Table 4.5 shows the reduction in debug execution time for different sets of random data experiments that have distinct error distributions. For the random data experiments, $N = 2^{27}$, $M = 2048$, $S = 4$, and no spatial compression and no streaming compression are enabled. The burst length represents the number of erroneous samples that occur consecutively and all the bursts are randomly distributed over the entire observation window. As it can be observed from Table 4.5, when the burst length increases, the reduction ratios grow. This is because when the burst length is approaching the segment size, for a given error percentage, the number of uncompressed debug sessions (the last level

Table 4.5: Reduction in Debug Execution Time (T_{seq}/T_{prop}) versus Error Percentage of Random Data for Different Burst Lengths with $N = 2^{27}$, $M = 2048$, $S = 4$

Error %	Burst length			
	64	128	256	512
0.78	13.2	21.9	32.8	48.9
1.55	7.2	12.1	18.6	26.4
2.32	5.0	8.4	13.2	18.6
3.08	4.0	6.6	10.3	14.6
3.83	3.3	5.4	8.4	12.0
4.58	2.8	4.6	7.2	10.3

of debug) are reduced (also the number of debug sessions in the intermediate levels are indirectly reduced as less failing signatures need to be processed). Finally, as the error percentage increases, the reduction ratios decrease for the same burst length because more debug sessions are necessary to detect the erroneous intervals (as also illustrated in Figure 4.4 from Section 4.5).

4.7 Summary

In this chapter, we have shown how lossy compression can be used for developing a new debug architecture and an iterative debug flow that provide an increase in the observation window for at-speed silicon debug of generic digital ICs. The proposed debug method accelerates the identification of the hard-to-find functional bugs that occur intermittently in long observation windows. This method enables an iterative debug technique for zooming only in the intervals containing erroneous samples and hence a significant reduction in the number of debug sessions can be achieved. The proposed solution is tailored for deterministic replay circuits and applicable for both ATE-based debug and in-field debug on application boards.

Chapter 5

Embedded Debug Architecture for Bypassing Blocking Bugs

Once a design bug is found during post-silicon validation, before committing to a re-spin of a design it is expected any other bugs, which have escaped pre-silicon verification, to be also identified. This will minimize the number of re-spins, which in turn will reduce both the implementation costs and the time-to-market. However, this is hindered by the presence of blocking bugs in one erroneous module that inhibit the search for bugs in other parts of the chip that process data received from this erroneous module. To address this problem, in this chapter we propose a novel embedded debug architecture for bypassing the blocking bugs.

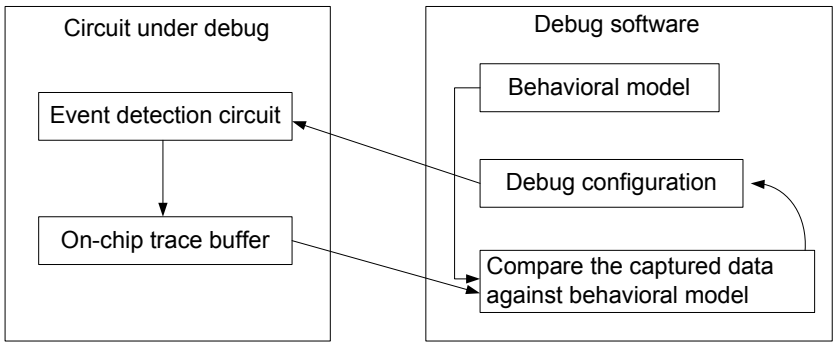
This chapter is organized as follows. Section 5.1 gives preliminaries for the research work presented in this chapter, outlines the motivation behind it and summarizes its contributions. Section 5.2 describes the proposed debug methodology for bypassing blocking bugs. Section 5.3 presents the proposed debug architecture. Experimental results from Section 5.4 show how the proposed debug architecture is used to aid in debugging an MP3 audio decoder. Finally, Section 5.5 concludes this chapter.

5.1 Preliminaries and Summary of Contributions

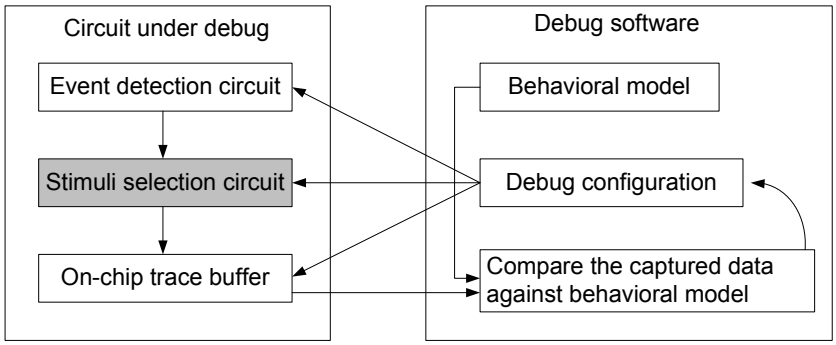
Due to the escalating mask costs, it is imperative to identify the design bugs that have escaped pre-silicon validation as soon as the first silicon is available. Because locating a design bug at a certain point makes it an obstacle for debugging the remaining parts of the design that are connected to this point, it is important to bypass its erroneous behavior. This type of bug is called a *blocking bug*. In the presence of blocking bugs, the erroneous samples have to be replaced in real time with the correct stimuli. To achieve this goal, the following two assumptions need to be satisfied. Firstly, the debug data has to be deterministically computed and reproduced using a reference behavioral model of the CUD. Secondly, the target application board (on which the CUD is located) has a deterministic execution behavior where re-applying the same input data will always produce the same output data. This deterministic behavior is common for target application boards where stimuli are applied synchronously (e.g., when debugging an audio/video decoder).

To illustrate the problem solved in this chapter, we show two different debug scenarios in Figure 5.1. In Figure 5.1(a), the on-chip trace buffer is used just for capturing the debug data based on the debug configuration that specifies the trigger condition at which the acquisition process starts. After the trace buffer is filled, the captured data is offloaded to the debug software, where the debug information is compared against the behavioral model. In order to replace the erroneous behavior caused by the blocking bugs, another level of triggering is needed to enable the trace buffer to provide the correct stimuli *only at the specific times determined by the occurrence of the blocking bugs*. This motivates our research to develop the stimuli selection module, which is shaded in Figure 5.1(b). In this scenario, the debug configuration includes the following:

- Initial trigger event for providing the stimuli data;
- The stimuli control information to be uploaded into stimuli selection circuitry; this information specifies the times at which the stimuli data will be provided and the duration of the stimuli intervals as well;



(a) Debug Scenario without Bypassing Blocking Bugs Feature



(b) Debug Scenario with Bypassing Blocking Bugs Feature

Figure 5.1: Debug Scenarios with and without Bypassing Blocking Bugs Feature

- The stimuli data to be uploaded into on-chip trace buffer.

Our objective in this chapter is to develop a debug methodology, and an associated logic circuitry to be integrated in the embedded logic analyzers, to enable the real-time replacement of the erroneous behavior (caused by blocking bugs) with the correct stimuli. Our contribution is motivated by the observation that design errors which escape to silicon will manifest in a burst-mode and only several times over a large execution time (bugs of this type are indeed the most hard-to-detect, as discussed in [57]). This key observation enables also to stream the stimuli data and control in real-time through a low-bandwidth interface connected to the debug software. The main contributions of this chapter are summarized as follows:

- we propose a novel architecture that enables hierarchical event detection mechanism to provide correct stimuli from an embedded trace buffer, in order to replace the erroneous samples caused by the blocking bugs;
- we develop an architectural feature in the embedded trace buffer controller in order to enable sharing the stimuli data, stimuli control and capture data in a segmented trace buffer that can be configured with different segments sizes for the purpose of debugging the targeted observation window;
- we show that by leveraging the streaming feature of the low-bandwidth interface to the debug software, we can further improve the observability by streaming the stimuli data.

5.2 Methodology for Bypassing Blocking Bugs

As outlined in the previous section, our debug methodology for dealing with blocking bugs relies on: (i) validation data can be deterministically computed from a reference behavioral model of the design, which avoids time-consuming circuit simulation; (ii) during post-silicon validation the circuit exhibits deterministic behavior, i.e., the non-determinism caused by asynchronous events is masked out. The proposed debug methodology relies on two phases.

Consider the two embedded cores shown in Figure 5.2. In the first phase, the lossy compression technique presented in Chapter 4 is used to identify the hard-to-find functional bugs in Core 1. This technique enables an iterative debug flow for zooming only in the intervals containing erroneous samples that occur intermittently in long observation windows. After identifying the exact times when the bugs from Core 1 are activated (and knowing from the behavioral model the correct values that need to appear on the core's output), the erroneous behavior should be *bypassed*, in order to validate in silicon the other parts of the design that are connected to the erroneous core (i.e., Core 2 in Figure 5.2).

In phase 2, discussed in this chapter, we replace the effect of the blocking bugs from Core 1 with the correct stimuli required for the validation of Core 2. We rely

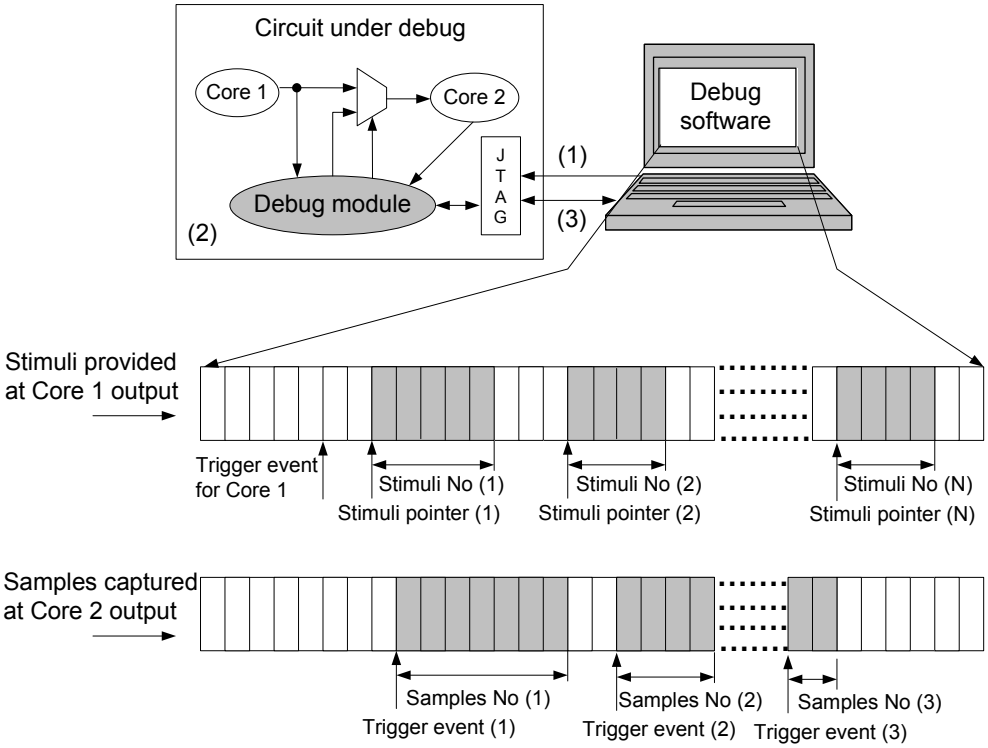


Figure 5.2: Bypassing Blocking Bugs Framework

on the fact that a blocking bug generates erroneous patterns in bursts that are not consecutive and the duration of each burst is different from one another as shown in Figure 5.2 (the shaded segments at the output of Core 1 represent the erroneous patterns). Note, in addition to uploading the replacements for the erroneous samples, a stimuli pointer (*Stimuli pointer*) that represents the beginning of the erroneous samples and an associated stimuli number (*Stimuli No)* are also uploaded in the embedded debug module. The debug steps for validating Core 2 are explained next.

The debug module is first uploaded with the debug configuration (step (1) in Figure 5.2). The debug configuration includes the trigger event that represents the beginning of the data block that has erroneous samples, as well as the stimuli data and stimuli control (i.e., stimuli pointer and stimuli number) for the first few erroneous samples to be replaced (the layout and the update of this information in the embedded

trace buffer are detailed in the following section). The stimuli pointer represents the time at which the stimuli will be provided at the probe point. The stimuli number specifies the number of the stimuli that are needed to replace the erroneous samples upon the occurrence of the stimuli pointer.

After the CUD starts execution (step (2) in Figure 5.2) and once the trigger event occurs, an internal counter from the debug module starts its operation and it increments whenever specific stimuli selection condition occurs. This stimuli selection condition represents the event at which the stimuli data should be made available to Core 2. Once this internal counter reaches the first stimuli pointer, the stimuli are read from the trace buffer until the number of erroneous samples to be replaced equals the stimuli number (step (3) in Figure 5.2). By exploiting the fact that erroneous values caused by Core 1 are sparse, the stimuli data control can be streamed in real-time through a low-bandwidth interface connected to the debug software, as discussed in the following section.

In order to capture data from Core 2 which is connected to Core 1 that has blocking bugs, trigger events for Core 2 need to be uploaded into the embedded debug module. These trigger events determine the times at which the acquisition processes start, while the associated samples numbers specify the required amount of sampling data as shown in Figure 5.2. Thus, the trace buffer is employed as a segmented buffer to allow providing stimuli at the output of Core 1 and capturing data at the output of Core 2. The acquisition process can be configured to capture a selected debug data every a specified number of clock cycles. This process is stopped once the required amount of the sampling data is captured or the capture data segment is filled. Thereafter, the captured data is offloaded to debug software. To increase the effectiveness of the embedded trace buffer during the debug process, we develop an architectural feature that enables sharing the stimuli data, stimuli control and capture data in a segmented trace buffer that can be configured with different segments sizes in order to debug the targeted observation window, as described in the following section.

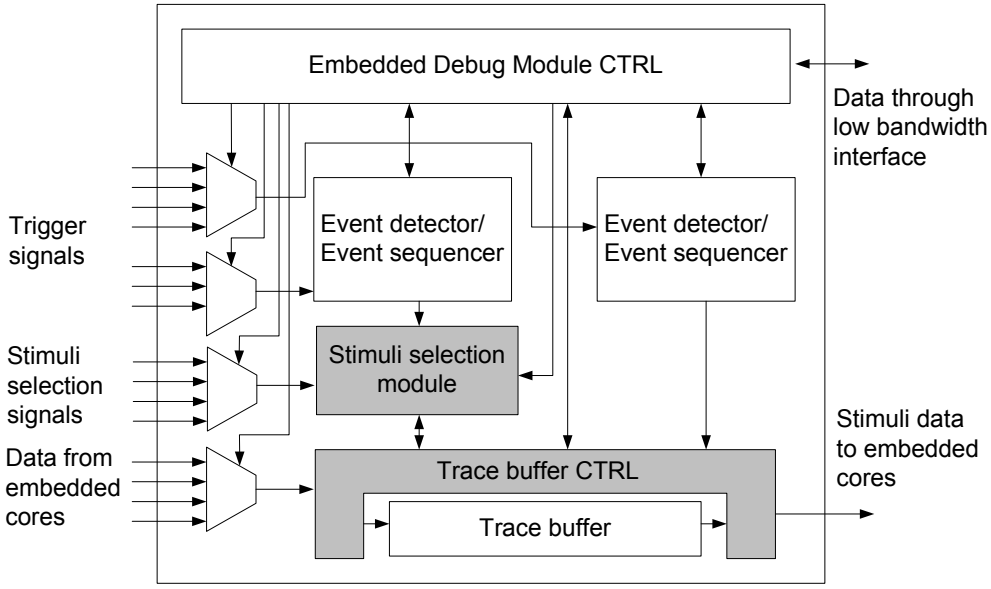


Figure 5.3: The Proposed Embedded Debug Module

5.3 Proposed Embedded Debug Module

This section introduces a modified embedded debug module, which enables bypassing the blocking bugs, based on the methodology described in the previous section. Our contributions to this embedded debug module are the stimuli selection module and the embedded trace buffer control, which are shaded in Figure 5.3.

5.3.1 Overview of the Embedded Debug Module

A standard embedded debug module can capture a set of internal samples after the occurrence of a certain triggering condition (i.e., trigger event). This is achieved by a detection mechanism using the event detector as shown in Figure 5.3. In our implementation, the triggering condition can be performed based on bitwise, comparison or logical operations between any selected trigger signal and a specified constant value. The purpose of using two event detector circuits is to concurrently monitor trigger signals from two different cores.

5.3.2 Stimuli Selection Module

The event detection capability of the embedded debug module can be extended to enable a mechanism for bypassing blocking bugs. This mechanism provides a second level of triggering at which the erroneous behavior of blocking bugs is replaced with the correct stimuli from the trace buffer. The first level of triggering represents the beginning of the first set of data that has erroneous samples and the second level indicates when the erroneous samples start to occur after the first level trigger condition is satisfied. The second trigger level is specified by the stimuli pointer.

The process of updating stimuli control registers is constrained by the time needed to upload new values in these registers through the low-bandwidth interface (also referred to as the serial interface). This time depends on the serial interface frequency and the width of the stimuli control (i.e., the width of the stimuli pointer register combined with stimuli number register). Therefore, the number of on-chip clock cycles required for updating the stimuli control equals $W \times f_{ratio}$, where $f_{ratio} = f_{CUD}/f_{JTAG}$ and f_{CUD} is the on-chip sampling frequency; f_{JTAG} is the frequency of the serial interface; W is the width of the stimuli control. If the interval between two erroneous samples is less than $W \times f_{ratio}$, then even the correct samples that occur *between* these two erroneous samples must also be loaded in the on-chip stimuli memory. Hence one stimuli pointer is used to indicate the beginning of this interval whose length is identified by the stimuli number. If the time between two stimuli pointers is sufficient to update the on-chip stimuli control registers during the debug experiment, the amount of stimuli will be reduced. We illustrate the importance of having multiple *on-chip stimuli pointers* with the following example.

Figure 5.4 shows part of the observation window that has multiple erroneous bursts over two interval groups. We assume that on-chip stimuli registers are uploaded with the information of stimuli control 1, 2, 3 and 4 in group A. In Figure 5.4, the time between the two groups is sufficient to upload these registers with new values (i.e., $d > 4 \times W \times f_{ratio}$). As a result, for the case of having four on-chip stimuli pointers, their values will be updated and hence any correct samples between any two consecutive erroneous samples within each group do not need to be stored on-chip as stimuli. On the other hand, for the case of having just one on-chip stimuli

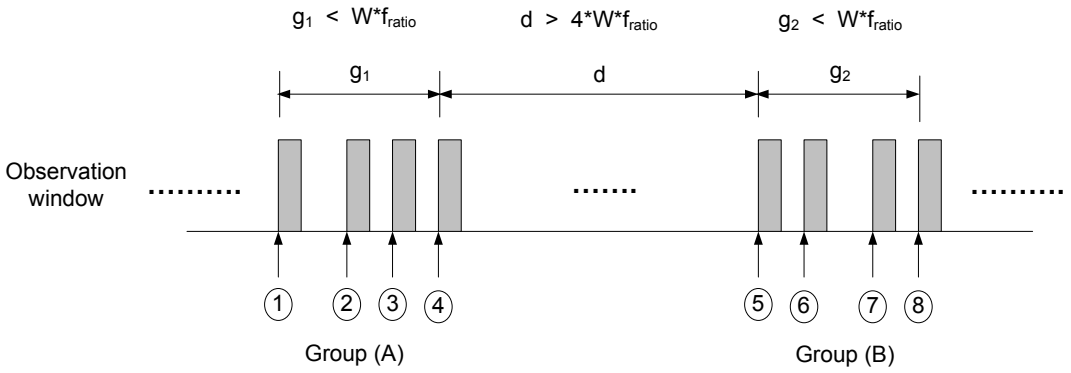


Figure 5.4: Example of On-chip Stimuli Pointers

control register (i.e., one register for the stimuli pointer and one register for the stimuli number), the information of the stimuli control 1 from Figure 5.4 will indicate all the erroneous bursts in group A. Since the time between any two bursts within group A is less than the time needed to upload one stimuli control information (i.e., $g_1 < W \times f_{ratio}$), the correct samples which occur between any two bursts within this group need to be stored in the on-chip trace buffer. Because similar patterns occur in group B (i.e., $g_2 < W \times f_{ratio}$), the information of stimuli pointer number 5 will be uploaded to the on-chip stimuli pointer register to indicate the erroneous bursts in group B. Given the importance of having on-chip information for stimuli control, we discuss different approaches to implement them.

In Figure 5.5, the individual registers that store the stimuli pointers and the associated stimuli numbers are accessed by the index counter. When the stimuli pointer counter reaches a specific stimuli pointer, which is determined by the index counter, the stimuli are enabled from the trace buffer for a certain duration specified by the stimuli number. The stimuli flag is enabled after the occurrence of the stimuli pointer. When the stimuli number counter reaches the value of the associated stimuli number register, this flag is disabled. The more stimuli pointers are stored on-chip, the less stimuli data needs to be stored. However, the drawback of this architecture is the area overhead caused by the physical registers used for storing the on-chip stimuli pointers.

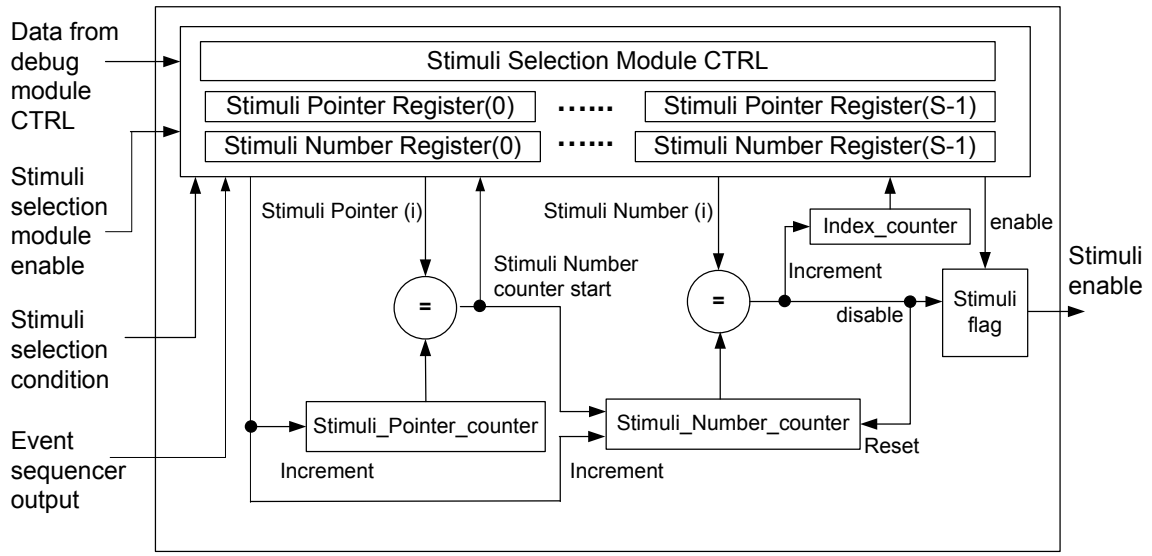


Figure 5.5: Stimuli Selection Module With S Stimuli Control Registers

An alternative is to store all of the stimuli control information for the entire observation window into the trace buffer (along with the stimuli) and allocate one stimuli control register in the stimuli selection module. These stimuli control values are accessed from the trace buffer by the stimuli selection module control. This obviously has a smaller area overhead than the approach described in the previous paragraph. However, it will consume as many locations of the trace buffer as the amount of the total stimuli control information. This solution is constrained by the limited capacity of the trace buffer and hence it impacts the length of the observation window.

The solution adopted in our work chooses only a user-programmable number of the stimuli pointers (and the associated stimuli numbers) to be stored in a segment of the trace buffer. Subsequently, one stimuli pointer register and one stimuli number register can be allocated in the stimuli selection module, as shown in Figure 5.6. The values for stimuli pointers and stimuli numbers can be updated from the off-chip software via the serial interface by exploiting the slack between consecutive bursts of erroneous samples that need to be substituted on-chip due to the blocking bugs.

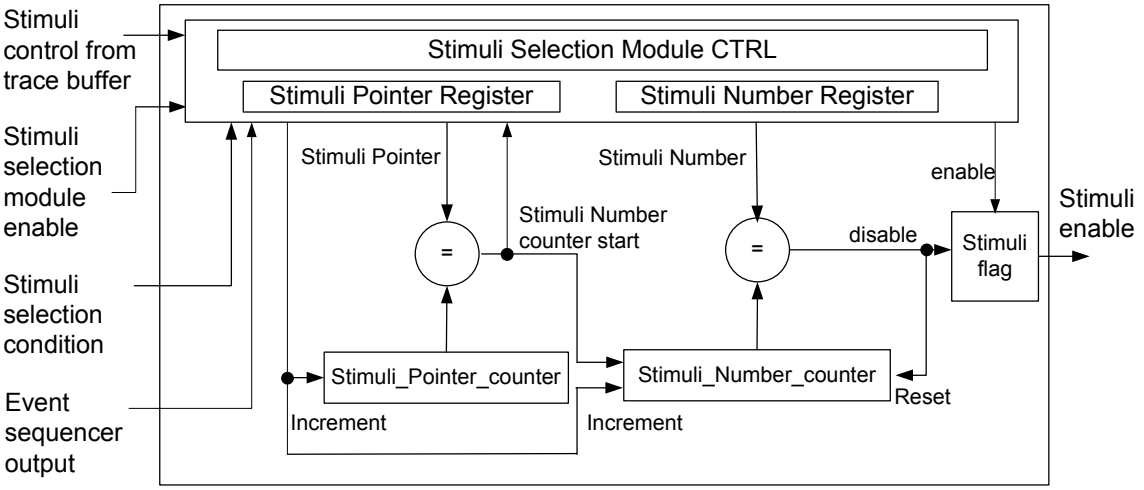


Figure 5.6: Stimuli Selection Module With Stimuli Control Stored in the Trace Buffer

This solution combines the benefits of the approaches described in the previous two paragraphs: it has an area overhead smaller than using dedicated physical registers for storing control information and it uses only a few locations in the trace buffer. Furthermore, by employing the low-bandwidth interface to the debug software, one can time-share this physical link to update also the stimuli data in the embedded trace buffer.

5.3.3 Embedded Trace Buffer Control

As shown in Figure 5.7, the dual-port embedded trace buffer has three segments: the first segment stores the stimuli control, the second one is used to store the stimuli data and the third one is used to capture the data responses from the core that is currently validated and it is connected to the core that has the blocking bugs. The segments for stimuli data and stimuli control work as circular buffers (i.e., if the reading address of any segment reaches its depth, it starts again from the beginning address for this segment). This is necessary in order to stream data in the trace buffer while running the CUD for long observation windows. This feature is essential for the stimuli control segment because having a few on-chip stimuli control values

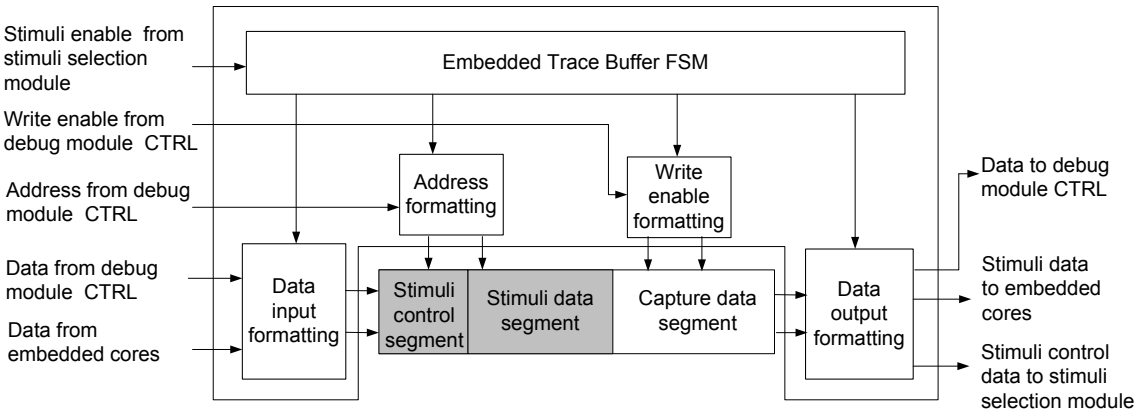


Figure 5.7: Embedded Trace Buffer CTRL

would help reducing the amount of the stored stimuli data (as explained in Section 5.3.2 and substantiated later in the experimental results). Because the hard-to-detect bugs occur intermittently over long observation windows, we have observed that there will be sufficient time to stream in new stimuli data during a long error-free interval that occurs between two erroneous intervals. It is also important to note that the embedded debug module CTRL is capable to time-share the low-bandwidth interface between stimuli control and data. It distinguishes between the stimuli data and control based on an one-bit tag information embedded in the stream that is supplied from the off-chip debug software. In summary, the embedded trace buffer CTRL controls the following processes:

- Writing the captured data into the trace buffer and reading it to be streamed out while running the debug experiment.
- Reading the stimuli control and stimuli data from the trace buffer and writing new stimuli control and new stimuli data, which are streamed in through the low-bandwidth interface. The accessed stimuli control and stimuli data are overwritten by the streamed new stimuli control and stimuli data, respectively.

Because the trace buffer is used for stimuli and data capture, it is essential to configure the size of stimuli segment and the size of capture segment such that as

much length of the observation window as possible can be debugged. These sizes can be changed from debug experiment to another in order to target a length of observation window longer than the previous debugged one. For example, in Figure 5.2, when running the debug experiment that targets capturing the debug data from Core 2 upon trigger event 2, the stimuli segment size can be selected to be smaller than the capture segment size. This is because the amount of stimuli, that is needed to bypass the blocking bugs at Core 1, does not require a large segment size to reach this trigger event. On the other hand, when running the debug experiment that target capturing the debug data from Core 2 upon trigger event 3, the stimuli segment has to accommodate all the stimuli required to bypass the blocking bugs until the occurrence of this trigger event and hence larger stimuli segment size will be needed.

5.4 Experimental Results

This section discusses the area and the advantages of the proposed embedded debug module for bypassing blocking bugs. The area results have been estimated using a 180nm ASIC standard cell library. The debug data has been collected from an FPGA prototype of an MP3 audio decoder [45] and the stimuli are used from a reference behavioral model of the MP3 decoder design.

5.4.1 Area of the Proposed Debug Module

Table 5.1 shows the area of the proposed debug module (without the trace buffer) in terms of 2 input NAND (NAND2) gates. The area results are for the debug module

Table 5.1: Area of The Proposed Debug Module in NAND2 Equivalents

No Stimuli Selection Module	With Stimuli Selection Module				
	Number of Stimuli Control Registers				
	1	2	4	8	16
5298	6758	6929	7539	8694	11050

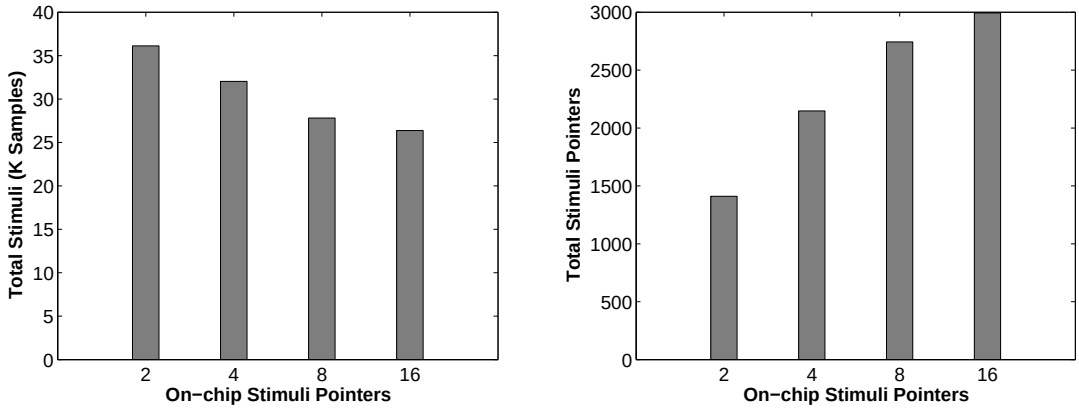
excluding the implemented stimuli selection module and for the debug module including the stimuli selection module with different number of stimuli control registers (1, 2, 4, 8 and 16). The stimuli selection module with one stimuli control register is shown in Figure 5.6. The results for 2, 4, 8 and 16 stimuli control registers are for the stimuli selection module shown in Figure 5.5.

It is essential to note that as the number of on-chip stimuli control registers increases, the area overhead of the debug module can be significantly impacted when compared to the debug module that has one stimuli control register. Therefore, it is desirable to store the values of the stimuli pointers and stimuli numbers into a few locations in the trace buffer and access these values through the proposed low cost architecture shown in Figure 5.6. As it can be noted, there is an approximately 30% impact on the silicon area when using the debug module with the architecture shown in Figure 5.6 when compared to the one that has no stimuli selection module. Note, however, when the area of the trace buffer is accounted for, this overhead is substantially diminished.

5.4.2 MP3 Decoder Experiments

The debug experiments have been performed on an FPGA prototype of an MP3 audio decoder and the stimuli are used from a reference behavioral model of the MP3 decoder design. It should be noted that for the experiments we have performed, the output of the stereo decoder module at one channel has erroneous sample patterns similar to the ones illustrated in Figure 5.2. These erroneous samples are due to a functional bug (in the RTL code) that was identified in the stereo decoder module. After analyzing the erroneous behavior of this blocking functional bug, we have noted that the errors occur only within a few music frames (each frame has two granules and each granule has 576 samples for each of the two channels) throughout the songs that have specific stereo decoding configuration; thus justifying the condition that the blocking bugs that are very difficult-to-find manifest themselves only intermittently over long observation windows (as noted also in [57]).

Figure 5.8 shows the effect of the on-chip stimuli pointers on the total number



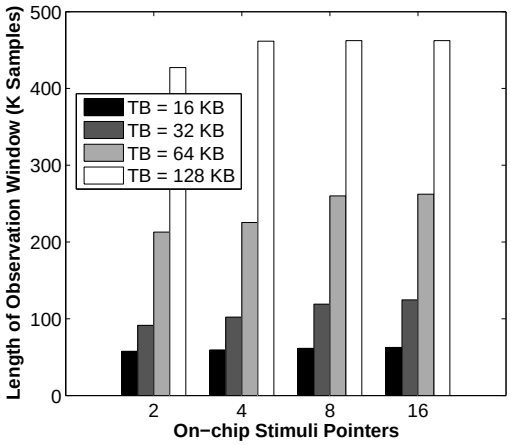
(a) The Total Number of Stimuli versus the Number of On-chip Stimuli Pointers

(b) The Total Number of Stimuli Pointers versus the Number of On-chip Stimuli Pointers

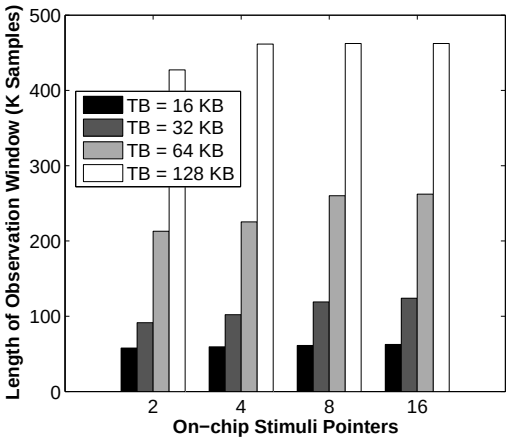
Figure 5.8: The Effect of the On-chip Stimuli Pointers on the Total Number of Stimuli and on the Total Number of Stimuli Pointers that are used during debug the entire Observation Window (Song = 462.38 k Samples, Sample = 2-byte word)

of stimuli and on the total number of stimuli pointers that are used during debug the entire observation window of an MP3 song. As discussed in Section 5.3.2, the number of on-chip stimuli pointers will influence the total number of stimuli pointers that need to be uploaded through the low-bandwidth interface. As the number of on-chip stimuli pointers increases, the total amount of stimuli decreases, as shown in Figure 5.8(a). This is due to the fact that the erroneous behavior of the detected blocking bugs occurs over intermittent intervals throughout the entire song. The time between two intervals is used to upload the on-chip stimuli pointers with new values and hence enables a large number of trigger pointers to be used, as illustrated in Figure 5.8(b). In Figure 5.8, we have considered $W = 32$ (the stimuli pointer combined with the stimuli number width is 32 bits) and $f_{ratio} = 2$ (the MP3 decoder has been implemented for energy-efficiency at low frequencies comparable with the speed of the low-bandwidth interface).

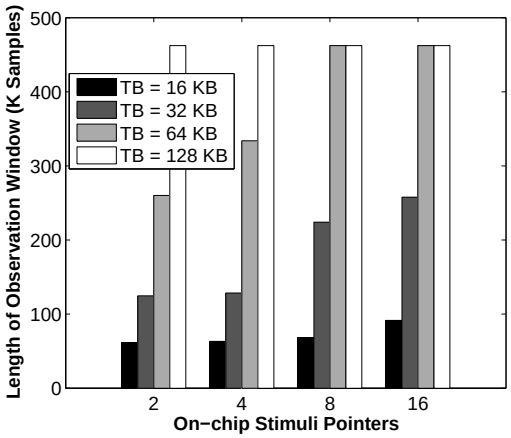
Figure 5.9 shows the length of the observation windows (i.e., the number of samples that can be observed from the entire song when using the stored stimuli in the trace



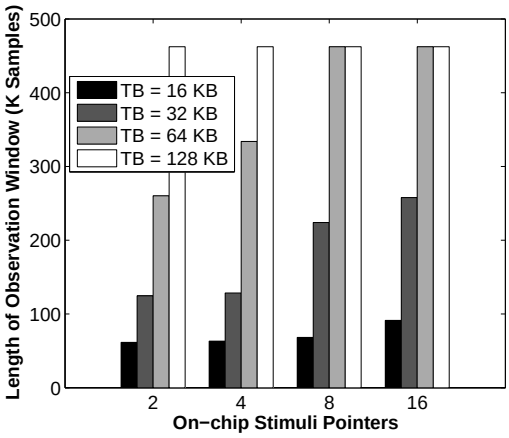
(a) Results with Arch. in Figure 5.5 (no stream)



(b) Results with Arch. in Figure 5.6 (no stream)



(c) Results with Arch. in Figure 5.5 (stream)



(d) Results with Arch. in Figure 5.6 (stream)

Figure 5.9: The Length of the Observation Window (k Samples) versus the On-chip Stimuli Pointers for different Sizes of Trace Buffers (k Bytes); Half of each Trace Buffer Size is used for Capturing Debug Data (Song = 462.38 k Samples)

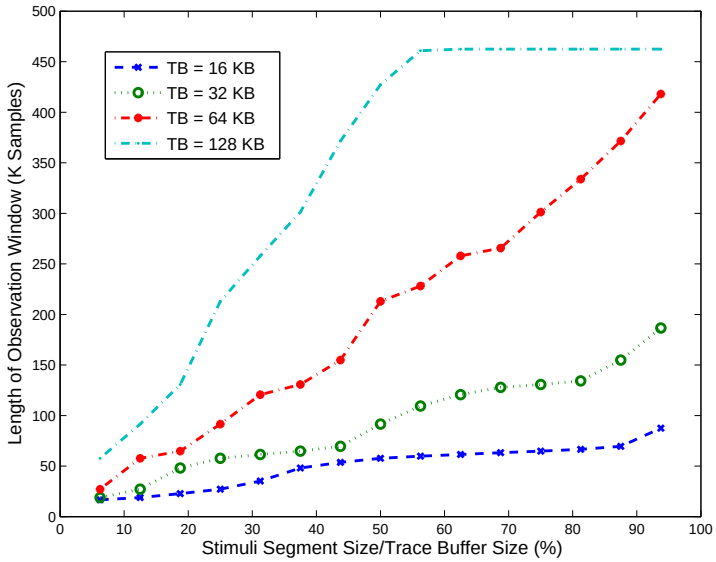
buffer) for different on-chip stimuli pointers and different sizes of the trace buffers (labeled as TBs in Figure 5.9). These results are for the case of no streaming is used for the stimuli data (as shown in Figures 5.9(a) and 5.9(b)); and for the case of streaming stimuli data (as shown in Figures 5.9(c) and 5.9(d)) to overwrite the stimuli that have been accessed by the embedded debug module. The total number of samples for the MP3 data are 462.38k (sample width = 2-byte word). There are three points that need to be emphasized.

Firstly, the stored stimuli represent intermittent intervals through the entire observation window. Thus, the larger the trace buffer we use, the larger the amount of stimuli that can be stored and hence the longer the observation window that can be achieved. If stimuli segment in the trace buffer is greater than the amount of the used stimuli, the blocking bugs are bypassed over the entire observation window, as observed when the trace buffer size is 128 Kbytes and the number of on-chip stimuli pointers equals either 8 or 16.

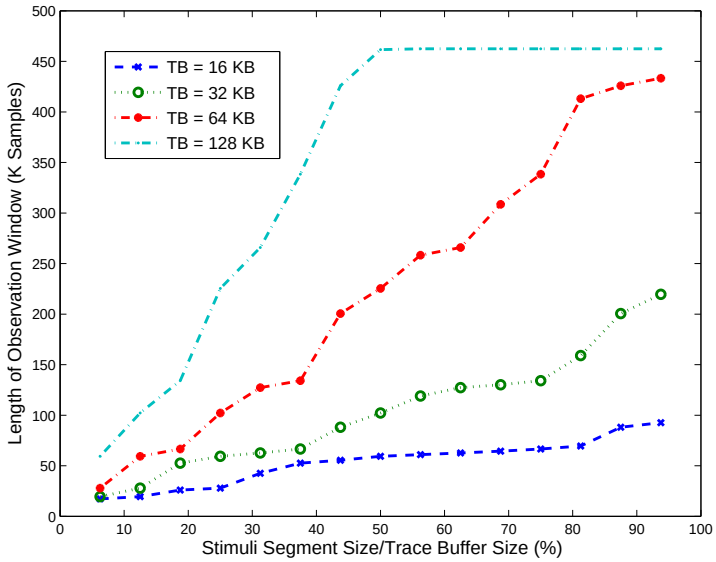
Secondly, as shown in Figure 5.9, the stimuli data streaming feature provides an increase in the length of observation window. This improvement is primarily due to exploiting the time between two erroneous intervals to stream in new stimuli data.

Thirdly, there is no significant difference between the length in observation windows when using the architecture from Figure 5.5 and the architecture from Figure 5.6. As pointed out in Section 5.3, the architecture from Figure 5.5 has more stimuli stored on-chip than the architecture from Figure 5.6 (this is due to the space allocated for the stimuli control segment in Figure 5.6). Note, these few extra stimuli in Figure 5.5 are used after the end of the observation window for Figure 5.6. Hence, the observation window length will be affected only if these extra stimuli are applied at an interval long after the end of the observation window achieved by the architecture from Figure 5.6.

Figures 5.10 and 5.11 show the length of the observation windows (i.e., the number of samples that can be debugged from the entire song when using the stored stimuli in the trace buffer) versus the stimuli segment size ratio for different on-chip stimuli pointers and different sizes of the trace buffers (labeled as TBs in Figures 5.10 and 5.11). These results show how the stimuli segment size constrains the length of the

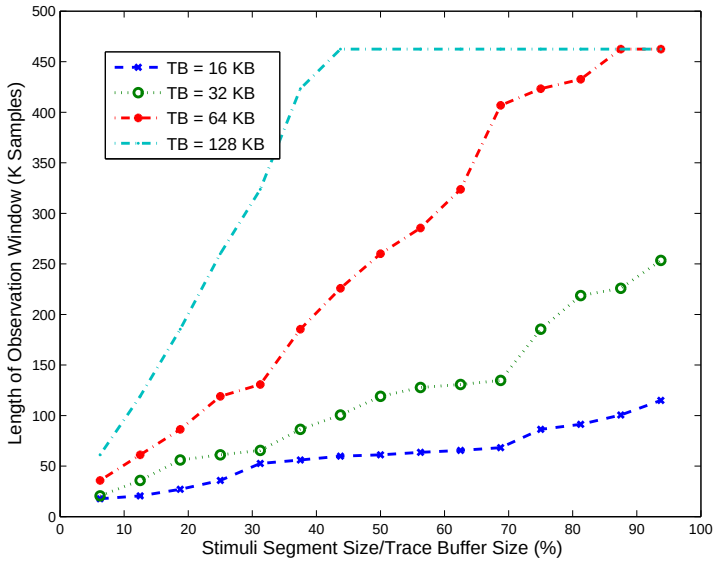


(a) Number of On-chip Stimuli Pointers equals 2

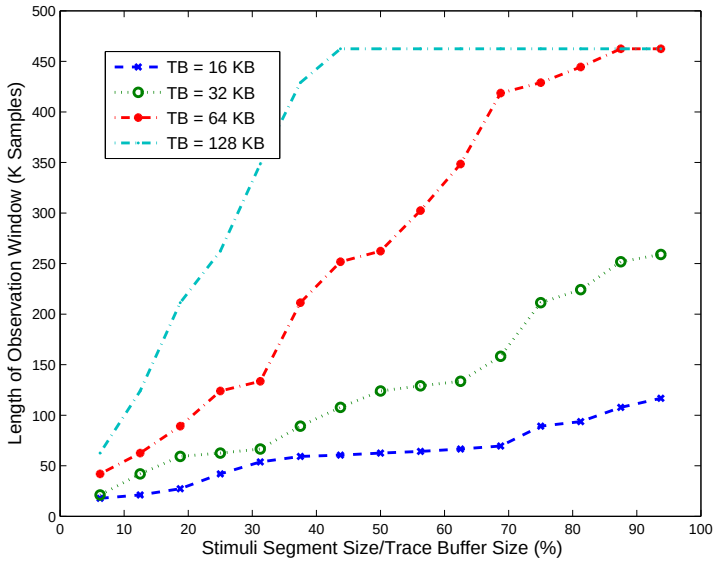


(b) Number of On-chip Stimuli Pointers equals 4

Figure 5.10: The Length of the Observation Window (k Samples) versus Stimuli Segment Size Ratio without using Stimuli Data Streaming, and the Number of On-chip Stimuli Pointers equals 2 and 4, respectively (Song = 462.38 k Samples)



(a) Number of On-chip Stimuli Pointers equals 8



(b) Number of On-chip Stimuli Pointers equals 16

Figure 5.11: The Length of the Observation Window (k Samples) versus Stimuli Segment Size Ratio without using Stimuli Data Streaming, and the Number of On-chip Stimuli Pointers equals 8 and 16, respectively (Song = 462.38 k Samples)

observation window that can be debugged. As discussed in Section 5.3.3, the stimuli segment size can be configured to target a specific length of the observation window and hence the capture data segment is used for capturing the data at specific intervals during the debug process. When the streaming feature for the capture data is enabled, the amount of data that can be captured, while running a long debug experiment, will be more than the amount of data that can be accommodated in the capture data segment.

It is interesting to note that in order to extend the length of the observation window without increasing the size of the trace buffer, compression techniques can be employed to store the stimuli in a compressed form. Based on the same principles of dictionary-based compression, as discussed in Chapter 3 for compressing capture data, we can build an on-chip de-compressor to extract the stimuli in real-time. Although this lossless decompression architecture is orthogonal to the main contribution presented in this chapter, it is worth mentioning nonetheless that the two contributions can be combined to further extend the observation window.

5.5 Summary

In this chapter, we proposed a novel embedded debug architecture for bypassing blocking bugs. This architecture facilitates the validation of the other parts of the chip that process data received from the erroneous module. The proposed approach enables a hierarchical event detection mechanism to provide correct stimuli from an embedded trace buffer, in order to replace the erroneous samples caused by the blocking bugs. We developed an architectural feature in the embedded trace buffer controller to enable sharing the stimuli data, stimuli control and capture data in a segmented trace buffer that can be configured with different segments sizes for the purpose of debugging the targeted observation window. Moreover, we have shown that by leveraging the streaming feature of the low-bandwidth interface to the debug software, we can further improve the observability by streaming the stimuli data.

Chapter 6

Conclusion and Future Work

Due to the growing complexity of SoCs and the ever increasing demand for shorter time-to-market, post-silicon validation has become an essential step in the implementation flow. As a result, design for debug techniques have emerged to accelerate the identification of design bugs during post-silicon validation. The purpose of these techniques is to enable the capturing and accessing the internal circuit's state of the circuit under debug. Two complementary DFD techniques have been introduced in Chapter 2: scan-based debug and embedded logic analysis. Although the first technique, which relies on scan-based debug methodology, can provide full observability of the internal system's state, it is not capable of handling real-time acquisition in consecutive clock cycles. Thus, embedded logic analysis techniques, which are based on real-time trace, have been introduced in order to aid scan-based methodology and accelerate the debug process. In this dissertation, we have proposed novel architectures and debug methods to be employed with embedded logic analysis techniques in order to facilitate the identification of the functional bugs. The rest of this chapter is organized as follows. Section 6.1 summarizes the main contributions of this dissertation, and Section 6.2 outlines directions for future work.

6.1 Summary of Dissertation Contributions

In Chapter 3, we proposed a novel debug architecture that enables real-time lossless data compression in embedded logic analysis. We analyzed the specific requirements for lossless data compression in embedded logic analysis. When evaluating the effectiveness of various compression techniques in improving data acquisition for post-silicon verification, one should also take the size of the compressor into account. As a consequence, we proposed a novel compression ratio metric that captures the real benefits of a compression algorithm that needs to satisfy high-throughput/real-time encoding with an acceptable area overhead. This metric is used to quantify the performance gain of the proposed dictionary-based compression architectures that support the most commonly used replacement policies. The proposed architectures are based on one pass scheme algorithms which do not require re-running the debug experiment. Thus, the proposed architectures are particularly suitable for in-field debug on application boards, which have non-deterministic events that inhibit the deterministic replay of debug experiments.

In Chapter 4, we introduced a novel debug architecture based on lossy compression. The proposed architecture enables an iterative debug approach that accelerates the identification of the hard-to-find functional bugs which occur intermittently in long observation windows. By extending the silicon debug observation window using a short sequence of debug sessions, the debug engineer can iteratively zoom only in the intervals that contain erroneous samples and hence a significant reduction in the number of debug sessions can be achieved. In this debug approach, we have shown that by leveraging the streaming feature of the low-bandwidth interface to the debug software we can further improve the effectiveness of our solution. The proposed debug architecture has a small impact on the silicon area when compared to the increase of the trace buffer size. This proposed debug approach is tailored for both automatic test equipment-based debug and in-field debug on application boards, so long as the debug experiment can be reproduced synchronously.

In Chapter 5, we addressed the problem of the blocking bugs which exist in one erroneous module and inhibit the search for bugs in other parts of the chip that

process data received from the erroneous module. We proposed a novel embedded debug architecture that can bypass the erroneous behavior of blocking bugs and aid the designer in validating the first silicon. The proposed architecture enables a hierarchical event detection mechanism to provide correct stimuli from an embedded trace buffer, in order to replace the erroneous samples caused by the blocking bugs. To increase the effectiveness of the embedded trace buffer during the debug process, we developed an architectural feature that enables sharing the stimuli data, stimuli control and capture data in a segmented trace buffer that can be configured with different segments sizes in order to debug the targeted observation window.

6.2 Future Research Directions

Based on the work proposed in this dissertation, future research directions are briefly outlined next. As the demand for multi-core designs increases, the post-silicon validation process becomes more challenging. As a result, the on-chip debug architectures allocated for post-silicon validation will be increased to facilitate the debug process. For example, future SoCs will contain more embedded logic analysis architectures. As discussed in Chapter 2, debug techniques have been recently introduced to address the problem of scheduling the capturing and offloading of debug data when it is required to simultaneously monitor the behavior of internal signals from multiple cores [66]. In order to improve the effectiveness of distributed embedded logic analyzers, future work will address the integration between these emerging debug techniques and the proposed architectures and methods introduced in this dissertation. For example, new debug architectural features and efficient scheduling algorithms can be developed for the purpose of capturing the debug data from multiple cores through distributed compression architectures.

The future generation of SoCs will show a grow in on-chip communication infrastructure. As a consequence, multi-core debug architectures are currently emerging to address the communications debug issues among multiple cores [103]. Future work will investigate new debug techniques to improve the observability in the emerging architectures in order to accelerate the debug process. These techniques will explore

the embedded logic analysis of the communication transactions among the embedded cores. Because a transaction represents a sequence of related events (e.g., a request and an acknowledge), it is effective to compress a sequence of transactions in order to increase the observation window of the transactions trace. Thus, the challenge is to develop compression architectures that exploit the unique characteristics of the transactions (e.g., how often the transactions are repeated), and satisfy high-throughput/real-time encoding with an acceptable area overhead. In addition, new architectural features can be introduced to enable reading the transactions from the embedded trace buffer and comparing them with the observed transactions. In this approach, part of the trace buffer can be used as a circular segment to capture the behavior of specific signals and their associated time stamps during the communication transactions. This capture process will be stopped upon the observation of a mismatch. Then, the observed behavior, which is captured into the embedded trace buffer, is analyzed in order to aid in the identification of the bug that causes the mismatch. Moreover, the transactions can be compressed and stored in a trace buffer, and an on-chip de-compressor can be employed to extract the transactions in real-time.

Bibliography

[1] M. Abramovici. In-System Silicon Validation and Debug. *IEEE Design & Test of Computers*, 25(3):216–223, May-June 2008.

[2] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller. A Reconfigurable Design-for-Debug Infrastructure for SoCs. In *Proceedings ACM/IEEE Design Automation Conference (DAC)*, pages 7–12, July 2006.

[3] M. Abramovici, E. J. Marinissen, M. Ricchetti, and B. West. Suggested Terminology Standard for Silicon Debug and Diagnosis. In *IEEE International Silicon Debug and Diagnosis Workshop (SDD)*, November 2005.

[4] Accellera Organization Inc. Property Specification Language Reference Manual. http://www.eda.org/vfv/docs/PSL-v1.1.pdf, 2004.

[5] Altera Verification Tool. SignalTap II Embedded Logic Analyzer. http://www.altera.com/products/software/products/quartus2/verification /signaltap2/sig-index.html, 2008.

[6] E. Anis and N. Nicolici. Low Cost Debug Architecture using Lossy Compression for Silicon Debug. In *Proceedings Design, Automation, and Test in Europe (DATE)*, pages 225–230, April 2007.

[7] E. Anis and N. Nicolici. On Using Lossless Compression of Debug Data in Embedded Logic Analysis. In *Proceedings IEEE International Test Conference (ITC)*, pages 1–10, paper 18.3, October 2007.

[8] E. Anis and N. Nicolici. On Bypassing Blocking Bugs during Post-Silicon Validation. In *Proceedings IEEE European Test Symposium (ETS)*, pages 69 –74, May 2008.

[9] ARM Limited. AMBA AXI Protocol Specification. http://www.arm.com, 2008.

[10] ARM Limited. Embedded Trace Macrocells. http://www.arm.com /products/solutions/ETM.html, 2008.

[11] K. Baker and J. V. Beers. Shmoo Plotting: The Black Art of IC Testing. *IEEE Design & Test of Computers*, 14(3):90–97, July/September 1997.

[12] H. Balachandran, K. Butler, and N. Simpson. Facilitating Rapid First Silicon Debug . In *Proceedings IEEE International Test Conference (ITC)*, pages 628 – 637, October 2002.

[13] P. Bardell, W. McAnney, and J. Savir. *Built-In Self Test - Pseudorandom Techniques*. John Wiley & Sons, 1986.

[14] J. V. Beers and H. V. Herten. Test Features of a Core-based Co-processor Array for Video Applications. In *Proceedings IEEE International Test Conference (ITC)*, pages 638–647, September 1999.

[15] L. Benini and G. De Micheli. Networks on Chips: a New SoC Paradigm. *IEEE Computer*, 35(1):70–78, January 2002.

[16] B. Bentley. Validating the Pentium 4 Microprocessor. In *The International Conference on Dependable Systems and Networks*, pages 193–198, 2001.

[17] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A Locally Adaptive Data Compression Scheme. *Communications of the ACM*, 29(4):320–330, April 1986.

[18] H. Bhatnagar. *Advanced ASIC Chip Synthesis*. Kluwer Academic Publishers, 2002.

[19] M. Boule, J.-S. Chenard, and Z. Zilic. Assertion Checkers in Verification, Silicon Debug and In-Field Diagnosis. In *Proceedings International Symposium on Quality of Electronic Design (ISQED)*, pages 613–620, March 2007.

[20] M. Boule, J.-S. Chenard, and Z. Zilic. Debug Enhancements in Assertion-checker Generation. *IEE Proceedings, Computers and Digital Techniques*, 1(6):669–677, November 2007.

[21] M. Boule and Z. Zilic. Incorporating Efficient Assertion Checkers into Hardware Emulation. In *Proceedings International Conference on Computer Design (ICCD)*, pages 221–228, October 2005.

[22] M. Boule and Z. Zilic. Efficient Automata-Based Assertion-Checker Synthesis of PSL Properties. In *Proceedings of the IEEE International High Level Design Validation and Test Workshop*, pages 69–76, 2006.

[23] M. Boule and Z. Zilic. Efficient Automata-Based Assertion-Checker Synthesis of SEREs for Hardware Emulation. In *Proceedings IEEE Asia South Pacific Design Automation Conference (ASP-DAC)*, pages 324–329, 2007.

[24] M. Bushnell and V. Agrawal. *Essentials of Electronic Testing*. Kluwer Academic Publishers, 2000.

[25] O. Caty, P. Dahlgren, and I. Bayraktaroglu. Microprocessor Silicon Debug Based on Failure Propagation Tracing. In *Proceedings IEEE International Test Conference (ITC)*, pages 284–293, October 2005.

[26] M. Chatterjee and D. K. Pradhan. A Novel Pattern Generator for Near-perfect Fault Coverage. In *Proceedings IEEE VLSI Test Symposium (VTS)*, pages 417–425, May 1995.

[27] C. Ciordas, K. Goossens, T. Basten, A. Radulescu, and A. Boon. Transaction Monitoring in Networks on Chip: The On-Chip Run-Time Perspective. In *Proceedings International Symposium on Industrial Embedded Systems (ISIES)*, pages 1–10, October 2006.

[28] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[29] P. Dahlgren, P. Dickinson, and I. Parulkar. Latch Divergency in Microprocessor Failure Analysis. In *Proceedings IEEE International Test Conference (ITC)*, pages 755–763, October 2003.

[30] R. Datta, A. Sebastine, and J. A. Abraham. Delay Fault Testing and Silicon Debug using Scan Chains. In *Proceedings IEEE European Test Symposium (ETS)*, pages 46–51, May 2004.

[31] W. de Boer and B. Vermeulen. Silicon Debug: Avoid Needles Respins. In *Proceedings International Electronics Manufacturing Technology Symposium (IEEMTS)*, pages 277–281, July 2004.

[32] G. de Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill International Editions, 1994.

[33] R. Desplats, F. Beaudoin, P. Perdu, N. Natara, T. Lundquist, and K. Shah;. Fault Localization Using Time Resolved Photon Emission and STIL Waveforms. In *Proceedings IEEE International Test Conference (ITC)*, pages 254–263, October 2003.

[34] R. Doering and Y. Nishi. *Handbook of Semiconductor Manufacturing Technology*. Prentice Hall, 2007.

[35] B. J. Falkowski. Equivalence Checking for Digital Circuits. *IEEE Potentials*, 23(2):21–23, April/May 2004.

[36] First Silicon Solutions Inc. System Navigator Pro. http://www.fs2.com/snav-pro.html, 2008.

[37] H. D. Foster, A. C. Krolnik, and D. J. Lacey. *Assertion-based Design*. Kluwer Academic Publishers, May 2004.

[38] T. J. Foster, D. L. Lastor, and P. Singh. First Silicon Functional Validation and Debug of Multicore Microprocessors. *IEEE Transactions on VLSI Systems*, 15(5):495–504, May 2007.

[39] S. Gerez. *Algorithms for VLSI Design Automation*. John Wiley & Sons, 1999.

[40] J. Ghosh-Dastidar and N. A. Touba. A Rapid and Scalable Diagnosis Scheme for BIST Environments with a Large Number of Scan Chains. In *Proceedings IEEE VLSI Test Symposium (VTS)*, pages 79–85, April 2000.

[41] S. K. Goel and B. Vermeulen. Data Invalidation Analysis for Scan-Based Debug on Multiple-Clock System Chips. *Journal of Electronic Testing: Theory and Applications*, 19(1):407–416, 2003.

[42] K. Goossens, B. Vermeulen, R. van Steeden, and M. Bennebroek. Transaction-Based Communication-Centric Debug. In *ACM/IEEE International Symposium on Networks-on-Chip (NOCS)*, pages 95–106, May 2007.

[43] T. Grotker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Springer, 2002.

[44] X. Gu, W. Wang, K. Li, H. Kim, and S. Chung. Re-using DFT Logic for Functional and Silicon Debugging Test. In *Proceedings IEEE International Test Conference (ITC)*, pages 648–656, October 2002.

[45] S. Hacker. *MP3: The Definitive Guide*. O'Reilly & Associates, Inc., May 2000.

[46] A. Hopkins and K. McDonald-Maier. Debug Support for Complex Systems on-chip: A Review. *IEE Proceedings, Computers and Digital Techniques*, 153(4):197–207, July 2006.

[47] A. Hopkins and K. McDonald-Maier. Debug Support Strategy for Systems-on-Chips with Multiple Processor Cores. *IEEE Transactions on Computers*, 55(2):174–184, February 2006.

[48] Y.-C. Hsu, F. Tsai, W. Jong, and Y.-T. Chang. Visibility Enhancement for Silicon Debug. In *Proceedings ACM/IEEE Design Automation Conference (DAC)*, pages 13–18, July 2006.

[49] Y. Huang and W.-T. Cheng. Using Embedded Infrastructure IP for SOC Post-Silicon Verification. In *Proceedings ACM/IEEE Design Automation Conference (DAC)*, pages 674–677, June 2003.

[50] D. A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

[51] IBM. Formal CheckerS (FoCs). http://www.haifa.ibm.com/projects/verification /focs/, 2006.

[52] IEEE Industry Standards and Technology Organization. *The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface*. http://www.nexus5001.org, 2003.

[53] IEEE JTAG 1149.1-2001 Std. *IEEE Standard Test Access Port and Boundary-Scan Architecture*. IEEE Computer Society, 2001.

[54] IEEE Std. 1800-2005. *IEEE Standard for SystemVerilog-Unified Hardware Design, Specification, and Verification Language*. IEEE Computer Society, 2005.

[55] IEEE Std. 1850-2005. *IEEE Standard for Property Specification Language (PSL)*. IEEE Computer Society, 2005.

[56] J. Jiang and S. Jones. Word-based Dynamic Algorithms for Data Compression. *IEE Proceedings, Communications, Speech and Vision*, 139(6):582–586, December 1992.

[57] D. Josephson. The Manic Depression of Microprocessor Debug . In *Proceedings IEEE International Test Conference (ITC)*, pages 657–663, October 2002.

[58] D. Josephson and B. Gottlieb. The Crazy Mixed up World of Silicon Debug. In *Proceedings IEEE Custom Integrated Circuits Conference (CICC)*, pages 665 –670, October 2004.

[59] D. Josephson, S. Poehhnan, and V. Govan. Debug Methodology for the McKinley Processor. In *Proceedings IEEE International Test Conference (ITC)*, pages 451–460, October 2001.

[60] C.-F. Kao, S.-M. Huang, and I.-J. Huang. A Hardware Approach to Real-Time Program Trace Compression for Embedded Processors. *IEEE Transactions on Circuits and Systems I*, 54(3):530–543, March 2007.

[61] M. Keating and P. Bricaud. *Reuse Methodology Manual: For System-on-a-Chip Designs*. Kluwer Academic Publishers, 1998.

[62] C. Kern and M. R. Greenstreet. Formal Verification in Hardware Design: A Survey. *ACM Transactions on Design Automation of Electronic Systems*, 4(2):123–193, April 1999.

[63] G. Kiefer, H. Vranken, E. Marinissen, and H. Wunderlich. Application of Deterministic Logic BIST on Industrial Circuits. In *Proceedings IEEE International Test Conference (ITC)*, pages 105–114, 2000.

[64] G. Kiefer and H.-J. Wunderlich. Deterministic BIST with Multiple Scan Chains. In *Proceedings IEEE International Test Conference (ITC)*, pages 1057–1064, 1998.

[65] D. E. Knuth. Dynamic Huffman Coding. *Journal of Algorithms*, 6(2):163 – 180, 1985.

[66] H. F. Ko, A. B. Kinsman, and N. Nicolici. Distributed Embedded Logic Analysis for Post-Silicon Validation of SoCs. In *Proceedings IEEE International Test Conference (ITC)*, pages 1–10, paper 16.3, October 2008.

[67] H. F. Ko and N. Nicolici. Automated Trace Signals Identification and State Restoration for Improving Observability in Post-Silicon Validation. In *Proceedings Design, Automation, and Test in Europe (DATE)*, pages 1298–1303, March 2008.

[68] T. Kumaki, Y. Kuroda, T. Koide, H. Mattausch, H. Noda, K. Dosaka, K. Arimoto, and K. Saito. CAM-based VLSI Architecture for Huffman Coding with Real-time Optimization of the Code Word Table. In *Proceedings International Symposium on Circuits and Systems (ISCAS)*, pages 5202–5205, May 2005.

[69] Y.-J. Kwon, B. Mathew, and H. Hao. FakeFault: A Silicon Debug Software Tool for Microprocessor Embedded Memory Arrays. In *Proceedings IEEE International Test Conference (ITC)*, pages 727–732, October 1998.

[70] W. K. Lam. *Hardware Design Verification: Simulation and Formal Method-Based Approaches*. Prentice Hall, 2005.

[71] R. Leatherman and N. Stollon. An Embedded Debugging Architecture for SoCs. *IEEE Potentials*, 24(1):12–16, February 2005.

[72] M. E. Levitt, S. Nori, S. Narayanan, G. P. Grewal, L. Youngs, A. Jones, G. Billus, and S. Paramanandam. Testability, Debuggability, and Manufacturability Features of the UltraSPARC-I Microprocessor. In *Proceedings IEEE International Test Conference (ITC)*, pages 157–166, October 1995.

[73] M.-B. Lin, J.-F. Lee, and G. Jan. A Lossless Data Compression and Decompression Algorithm and Its Hardware Architecture. *IEEE Transactions on VLSI Systems*, 14(9):925–936, September 2006.

[74] L. Y. Liu, J. F. Wang, R. J. Wang, and J. Y. Lee. Design and Hardware Architectures for Dynamic Huffman Coding. *IEE Proceedings, Computers and Digital Techniques*, 142(6):411–418, November 1995.

[75] R. Livengood and D. Medeiros. Design For (Physical) Debug For Silicon Microsurgery and Probing of Flip-Chip Packaged Integrated Circuits. In *Proceedings IEEE International Test Conference (ITC)*, pages 877–882, September 1999.

[76] C. MacNamee and D. Heffernan. Emerging On-chip Debugging Techniques for Real-Time Embedded Systems. *IEE Computing & Control Engineering Journal*, 11(6):295–303, December 2000.

[77] A. Mayer, H. Siebert, and K. McDonald-Maier. Debug Support, Calibration and Emulation for Multiple Processor and Powertrain Control SoCs. In *Proceedings Design, Automation, and Test in Europe (DATE)*, pages 148–152, March 2005.

[78] K. Morris. On-Chip Debugging - Built-in Logic Analyzers on your FPGA. *Journal of FPGA and Structured ASIC*, 2(3), January 2004.

[79] Z. Navabi. *VHDL: Analysis and Modeling of Digital Systems*. McGraw-Hill, 1997.

[80] Z. Navabi. *Verilog Digital System Design (Professional Engineering)*. McGraw-Hill, 1999.

[81] J. L. Nunez and S. Jones. Gbit/s Lossless Data Compression Hardware. *IEEE Transactions on VLSI Systems*, 11(3):499 – 510, June 2003.

[82] J. L. Nunez-Yanez and V. A. Chouliaras. Gigabyte per Second Streaming Lossless Data Compression Hardware Based on a Configurable Variable-Geometry CAM Dictionary. *IEE Proceedings, Computers and Digital Techniques*, 153(1):47–58, January 2006.

[83] OCP International Partnership. Open Core Protocol Specification. http://www.ocpip.org, 2008.

[84] K. Pagiamtzis and A. Sheikholeslami. Content-Addressable Memory (CAM) Circuits and Architectures: a Tutorial and Survey. *IEEE Journal of Solid-State Circuits*, 41(3):712–727, March 2006.

[85] M. Paniccia, T. Eiles, V. R. M. Rao, and W. M. Yee. Novel Optical Probing Technique for Flip Chip Packaged Microprocessors. In *Proceedings IEEE International Test Conference (ITC)*, pages 740 – 747, October 1998.

[86] S.-B. Park and S. Mitra. IFRA: Instruction Footprint Recording and Analysis for Post-silicon Bug Localization in Processors. In *Proceedings ACM/IEEE Design Automation Conference (DAC)*, pages 373–378, June 2008.

[87] Philips Semiconductors. Device Transaction Level (DTL) Protocol Specification. http://www.philips.com, 2008.

[88] A. R. Pleszkun. Techniques for Compressing Program Address Traces. In *IEEE International Symposium on Microarchitecture (ISMICRO)*, pages 32–39, November-December 1994.

[89] C. Pyron, R. Bangalore, D. Belete, J. Goertz, A. Razdan, and D. Younger. Silicon Symptoms to Solutions: Applying Design for Debug Techniques. In *Proceedings IEEE International Test Conference (ITC)*, pages 664 –672, October 2002.

[90] B. Quinton and S. Wilton. Post-silicon Debug Using Programmable Logic Cores. In *Proceedings IEEE International Conference on Field-Programmable Technology (ICFPT)*, pages 241 – 248, December 2005.

[91] M. Riley, N. Chelstrom, M. Genden, and S. Sawamura. Debug of the CELL Processor: Moving the Lab into Silicon. In *Proceedings IEEE International Test Conference (ITC)*, pages 1–9, paper 26.1, 2006.

[92] J. J. Rissanen. Generalized Kraft Inequality and Arithmetic Coding. *IBM Journal of Research and Development*, 20(3):198–203, 1976.

[93] G. J. V. Rootselaar and B. Vermeulen. Silicon Debug: Scan Chains Alone Are Not Enough. In *Proceedings IEEE International Test Conference (ITC)*, pages 892–902, September 1999.

[94] S. Sarangi, B. Greskamp, and J. Torrellas. CADRE: Cycle-Accurate Deterministic Replay for Hardware Debugging. In *IEEE International Conference on Dependable Systems and Networks (IDSN)*, pages 301 – 312, June 2006.

[95] S. R. Sarangi, A. Tiwari, and J. Torrellas. Phoenix: Detecting and Recovering from Permanent Processor Design Bugs with Programmable Hardware. In *IEEE International Symposium on Microarchitecture (ISMICRO)*, pages 26–37, 2006.

[96] I. Silas, I. Frumkin, E. Hazan, E. Mor, and G. Zobin. System-Level Validation of the Intel Pentium M Processor. *Intel Technology Journal*, 7(2):37–43, May 2003.

[97] J. M. Soden and R. E. Anderson. IC Failure Analysis: Techniques and Tools for Quality and Reliability Improvement. *Proceedings of the IEEE*, 81(5):703–715, May 1993.

[98] S. Sutherland, S. Davidmann, and P. Flake. *A Guide to Using System Verilog for Hardware Design and Modeling.* Springer, 2006.

[99] A. Swaine and J. Horley. *Summary of new features in ETMv3.* http://www.arm.com/products/solutions/ETM.html, 2005.

[100] Synplicity Verification Tool. Identify. http://www.synplicity.com/products/identify/index.html, 2008.

[101] B. Tabbara and K. Hashmi. Transaction-level Modeling and Debug of SoCs. In *Proceedings of IP Based SoC Conference*, pages 1–6, December 2004.

[102] S. Tang and Q. Xu. A Multi-Core Debug Platform for NoC-Based Systems. In *Proceedings Design, Automation, and Test in Europe (DATE)*, April 2007.

[103] S. Tang and Q. Xu. A Debug Probe for Concurrently Debugging Multiple Embedded Cores and Inter-core Transactions in NoC-based Systems. In *Proceedings IEEE Asia South Pacific Design Automation Conference (ASP-DAC)*, pages 416–421, January 2008.

[104] S. Tang and Q. Xu. In-band Cross-Trigger Event Transmission for Transaction-Based Debug. In *Proceedings Design, Automation, and Test in Europe (DATE)*, pages 414–419, March 2008.

[105] N. Touba and E. McCluskey. Altering a Pseudorandom Bit Sequence for Scan-Based BIST. In *Proceedings IEEE International Test Conference (ITC)*, pages 167–175, 1996.

[106] R. A. Uhlig and T. N. Mudge. Trace-driven Memory Simulation: A survey. *ACM Computing Surveys (CSUR)*, 29(2):128–170, June 1997.

[107] D. P. Vallett. IC Failure Analysis: The Importance of Test and Diagnostics. *IEEE Design and Test of Computers*, 14(3):76–82, July 1997.

[108] K. van Kaam, B. Vermeulen, and H. Bergveld. Test and Debug Features of the RTO7 Chip. In *Proceedings IEEE International Test Conference (ITC)*, pages 274–283, October 2005.

[109] S. Vangal, J. Howard, G. Ruhl, D. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *IEEE Journal of Solid-State Circuits*, 43(1):29–41, January 2008.

[110] B. Vermeulen. Functional Debug Techniques for Embedded Systems. *IEEE Design & Test of Computers*, 25(3):208–215, May-June 2008.

[111] B. Vermeulen and S. Bakker. Debug Architecture for the En-II System Chip. *IEE Proceedings, Computers and Digital Techniques*, 1(6):678 – 684, November 2007.

[112] B. Vermeulen and S. K. Goel. Design for Debug: Catching Design Errors in Digital Chips. *IEEE Design and Test of Computers*, 19(3):35 – 43, May 2002.

[113] B. Vermeulen, K. Goossens, R. V. Steeden, and M. Bennebroek. Communication-Centric SoC Debug Using Transactions. In *Proceedings IEEE European Test Symposium (ETS)*, pages 69–76, May 2007.

[114] B. Vermeulen, K. Goossens, and S. Umrani. Debugging Distributed-Shared-Memory Communication at Multiple Granularities in Networks on Chip. In

ACM/IEEE International Symposium on Networks-on-Chip (NOCS), pages 3–12, April 2008.

[115] B. Vermeulen, S. Oostdijk, and F. Bouwman. Test and Debug Strategy of the PNX8525 NexperiaTM Digital Video Platform System Chip. In *Proceedings IEEE International Test Conference (ITC)*, pages 121–130, October 2001.

[116] B. Vermeulen, M. Z. Urfianto, and S. K. Goel. Automatic Generation of Breakpoint Hardware for Silicon Debug. In *Proceedings ACM/IEEE Design Automation Conference (DAC)*, pages 514–517, June 2004.

[117] B. Vermeulen and G. J. van Rootselaar. Silicon Debug of a Co-processor Array for Video Applications. In *IEEE International High-Level Design Validation and Test Workshop*, pages 47–52, November 2000.

[118] B. Vermeulen, T. Waayers, and S. K. Goel. Core-Based Scan Architecture for Silicon Debug. In *Proceedings IEEE International Test Conference (ITC)*, pages 638 – 647, October 2002.

[119] T. A. Welch. A Technique for High-Performance Data Compression. *IEEE Transactions on Computers*, 17(6):8–19, June 1984.

[120] H. Wunderlich and G. Kiefer. Bit-Flipping BIST. In *Proceedings International Conference on Computer-Aided Design (ICCAD)*, pages 337 –343, 1996.

[121] Xilinx Inc. Content-Addressable Memory v5.1. In *Xilinx Application Note DS253*, November 2004.

[122] Xilinx Verification Tool. ChipScope Pro. http://www.xilinx.com/ise /optional_prod/cspro.html, 2008.

[123] J.-S. Yang and N. A. Touba. Expanding Trace Buffer Observation Window for In-System Silicon Debug through Selective Capture. In *Proceedings IEEE VLSI Test Symposium (VTS)*, pages 345–351, April 2008.

[124] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

[125] J. Ziv and A. Lempel. Compression of Individual Sequences via Variable-rate Coding. *IEEE Transactions on Information Theory*, 24(5):530–536, September 1978.

Index

ABV, 30
ASIC, 62, 94, 112
ATE, 5, 6, 9, 14, 17, 20, 41, 76, 77
ATPG, 5

BIST, 6, 7, 83, 85, 86
blocking bugs, 10–12, 100–103, 105–107, 109, 110, 112–114, 116
Boundary Scan interface, 23
breakpoint, 17–22, 26
BSTW, 45, 47, 56, 57, 60–64, 67, 69, 71, 72

CAM, 46–55, 59–61, 63, 64, 66–73
CMOS, 3
CUD, 9, 14, 15, 18, 20, 24, 37, 52, 55, 56, 77, 85, 87–91, 97, 98, 101, 105, 107, 110
CUT, 5–7, 16

debug session, 18, 29, 34, 75, 77, 79, 83–86, 88, 89, 91–93, 95, 96, 98
depth compression, 11, 41, 76
deterministic replay, 41, 43, 77, 80
DFD, 10, 14, 15, 17, 23, 29
DFT, 6, 16
dictionary-based compression, 36, 40, 42, 45–47, 56, 57, 62, 73, 119

electrical bugs, 8, 10, 13–15
electrical validation, 14, 20
embedded cores, 1, 24, 28, 33, 34, 103
embedded logic analysis, 9–11, 13, 15, 23, 24, 27, 29, 32, 34, 40–47, 50, 56, 57, 62, 64, 66, 73

FIFO, 30, 48–51, 61, 63–65, 67–70, 72, 73
formal verification, 1, 4, 13, 30
FPGA, 26, 62, 65, 94, 96, 112, 113
functional bugs, 8, 10, 12, 14, 15, 23, 41, 76, 97, 103
functional validation, 14

HDL, 2

IC, 1

LFSR, 7, 49, 50, 64
LFU, 51, 53–56, 63–65, 67–70, 72, 73
lossless compression, 10, 11, 40, 42–44, 46, 48
lossy compression, 12, 75–77, 83, 85, 103
LRU, 50–53, 56, 63–65, 67–70, 72

Manufacturing test, 5

MISR, 7, 8, 82, 83, 86, 87, 90

netlist, 2–6, 8, 37
NoC, 33

observation window, 10–12, 34, 37, 38, 40, 42–44, 66, 71, 73, 75–77, 79, 80, 83–88, 92–94, 96–98, 103, 105, 107, 109, 112, 116, 119

post-silicon validation, 2, 8–11, 13–15, 18, 20, 21, 29, 31, 33, 41, 100, 103
pre-silicon verification, 1, 3, 4, 8, 12, 13, 30, 33, 76, 100
PSL, 30–32

RAM, 63
real-time compression, 42, 43, 46
RTL, 2, 3, 32, 96, 113

scan chain, 15–17, 20, 21
scan-based debug, 9, 13, 17, 26, 27, 29, 34, 79
scan-based DFT, 16
SFF, 16
signature analyzer, 7
silicon debug, 2, 8, 12, 41, 75–77, 79, 80, 85
SoC, 1, 2, 4, 8–11, 13, 17, 23, 27–30, 33–37, 41, 94
spatial compression, 90, 94, 96, 98
synthesis, 2

time-to-market, 1, 4, 13, 100
TPG, 7
trace buffer, 10–13, 23, 25–29, 31, 32, 37, 38, 41, 43, 44, 46, 48, 56, 60, 63, 66, 71, 73, 75–77, 79, 80, 82–86, 88–92, 94–97, 101–103, 105–113, 116, 119
transaction-level debug, 29, 33, 34
trigger condition, 18–21, 24–26, 29, 48, 83, 101, 107

Verilog, 2, 30
VHDL, 2
VLSI, 2–5, 7

WDLZW, 45, 47, 57, 60–64, 68, 70–72
width compression, 10, 37, 79, 90