

NEW ALGORITHMS AND ARCHITECTURES FOR
POST-SILICON VALIDATION

NEW ALGORITHMS AND ARCHITECTURES FOR
POST-SILICON VALIDATION

BY

HO FAI KO, B.Eng. & Mgt., M.A.Sc.

APRIL 2009

A THESIS

SUBMITTED TO THE SCHOOL OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE

DOCTOR OF PHILOSOPHY

McMaster University

© Copyright 2009 by Ho Fai Ko

All Rights Reserved

DOCTOR OF PHILOSOPHY (2009)
(Electrical and Computer Engineering)

MCMMASTER UNIVERSITY
Hamilton, Ontario, Canada

TITLE: New Algorithms and Architectures for Post-Silicon Validation

AUTHOR: Ho Fai Ko, B.Eng. & Mgt., M.A.Sc. (Electrical and Computer Engineering, McMaster University, Canada)

SUPERVISOR: Dr. Nicola Nicolici

NUMBER OF PAGES: xvii, 170

Abstract

To identify design errors that escape pre-silicon verification, post-silicon validation is becoming an important step in the implementation flow of digital integrated circuits. While many debug problems are tackled on testers, there are hard-to-find design errors that are activated only in-system. A key challenge during in-system debugging is to acquire data from internal circuit's nodes in real-time. In this thesis, we propose several techniques to address this problem, ranging from resource-efficient and programmable trigger units to automated selection of trace signals to a distributed architecture for embedded logic analysis.

Deciding when to acquire data on-chip is done using trigger units. Because there is an inherent tradeoff between the size of the trigger units and the types of events that can be programmed into them, we first explore a resource-efficient and programmable trigger unit implementation. We show how the on-chip buffers used for data acquisition can be leveraged to store information regarding the logic functions that are programmed at runtime as the trigger events. This reduces the requirement in terms of logic resources for the trigger unit, while enlarging the set of programmable trigger events supported by these resources. We also propose a new algorithm to automatically map trigger events onto the proposed trigger unit.

Next we shift the focus from the trigger units to the sample units available on-chip. Once the real-time debug experiment has been completed, the amount of data available to the user is limited by the capacity of the on-chip trace buffers. For logic bugs, where the circuit implementation matches the physical prototype, we show how the structural information from the circuit netlist can be leveraged to expand the amount of data made available off-line to the user. To ensure that data expansion

can scale as the amount of debug data that is acquired increases, we propose a fast algorithm that leverages the bitwise parallelism of logic operations available in the instruction set of microprocessors. In a follow-up chapter, we also discuss how trace signals can be automatically selected in order to improve the amount of data that can be restored off-line. To achieve this objective, we propose two new metrics and two new algorithms for automatically identifying the circuit nodes which, if traced, will aid data expansion for the neighboring nodes in the circuit.

The last contribution of this thesis is concerned with managing multiple trace buffers in complex designs with many logic blocks. We propose a new distributed embedded logic analysis architecture that can dynamically allocate the trace buffers at runtime based on the needs for debug data acquisition coming from multiple logic blocks. We also leverage the real-time offload capability through high-speed trace ports in order to extend the duration of a debug experiment. It is shown how with little investment in on-chip debug logic resources, the length of debug experiments can be expanded for multi-core designs without increasing the number of on-chip trace buffers.

Acknowledgments

Over the course of my doctoral studies, I always thought I was the person who had the influence over the people around me. However, what I did not realize was that, in fact, the people around me had also influenced the way my personality was developed over the past five years.

I would like to first give my gratitude to my parents, my brother Chris, and my sister Eska. Their generous patience had provided me the best shelter whenever I lost confidence in myself. Although it has been a long time since the five of us sat together as a family, each of us knows that we are always connected in our hearts, and we support each other in every way.

Secondly, I would like to express my deepest thanks to my supervisor Dr. Nicola Nicolici. His constant battering gave me the opportunity to learn so much from him both technically, and personally. Although I had doubted myself, he had always stood by me and believed in me to give me the success I have today.

My special thanks also go to not only my colleague, but one of my best friends Adam Kinsman and his family. Their comforting home, delicious meals prepared by Pamela and cheerful laughs from Eric and Robyn provided me the energy whenever I felt like I was running out of battery. I would also like to acknowledge Adam for his effort on developing the MPEG decoder, which was used in the experiments for the work of this thesis.

Drs. Sirouspour and Haddara have been constructive during the supervisory committee meetings and I acknowledge them for their continuous support. My colleagues in the Computer-Aided Design and Test (CADT) Research Group: Johnny Xu, Baihong Fang, David Lemstra, David Leung, Ehab Anis, Kaveh Elizeh, Zaha Lak, Mark

Jobes, Jason Thong and Phil Kinsman have assisted me in so many ways. I would like to thank them for their great company when disasters strike. I would also like to express my gratitude to the graduate students, faculty, administrative and technical members in Department of Electrical and Computer Engineering at McMaster University for their assistance during my study and research.

I would also like to thank all my friends. Their continuous yelling, arguments, opinions and support had pushed me from behind to give me the extra lift I needed.

And last, but the most important person I would like to thank, is Nana. She had given me the happiest days of my life that gave me strength to continue on my studies without me realizing it. She had also taught me what it is like to feel truly appreciated for who I am, but not what I do. I hope that in the future, I can make her feel the same way whenever she is with me.

List of Abbreviations

ASIC	Application Specific Integrated Circuit,
ATE	Automatic Test Equipment,
ATPG	Automatic Test Pattern Generation,
CAD	Computer-Aided Design,
CMOS	Complementary Metal-Oxide Semiconductor,
CUD	Circuit under Debug,
CUT	Circuit under Test,
DA	Debug Agent,
DFD	Design for Debug,
DFT	Design for Testability,
DP	Debug Probe,
ELA	Embedded Logic Analyzer,
FF	Flip-Flop,
FIFO	First-In First-Out,
FPGA	Field Programmable Gate Array,
FSM	Finite State Machine,
HDL	Hardware Description Language,
IP	Intellectual Property,
K-map	Karnaugh Map,
MISR	Multiple Input Signature Register,
NOC	Network-on-Chip,
OCI	On-Chip Instrumentation,
OVL	Open Verification Library,

PSL	Property Specification Language,
PTE	Programmable Trigger Engine,
QBF	Quantified Boolean Formulation,
RTL	Register Transfer Level,
SFF	Scanned Flip-Flop,
SOC	System-on-a-Chip,
TDI	Test Data In,
TDO	Test Data Out,
VLSI	Very Large Scale Integration,

Contents

Abstract	iii
Acknowledgments	v
List of Abbreviations	viii
1 Introduction	1
1.1 VLSI design flow	1
1.2 Pre-silicon verification	4
1.3 Manufacturing test	5
1.4 Post-silicon validation	7
1.4.1 Circuit bugs	9
1.4.2 Logic bugs	9
1.4.3 System bugs	10
1.4.4 Terminology definitions	10
1.4.5 Non-deterministic and deterministic replay experiments	12
1.4.6 Scan chain-based technique	12
1.4.7 Trace-based technique	13
1.5 Contributions and organization of the thesis	14
2 Background and related work	16
2.1 Scan chain-based technique	16
2.1.1 Scan for access	16
2.1.2 Breakpoints	18

2.1.3	Clock control	19
2.2	Trace-based technique	21
2.2.1	Software debug using traces	21
2.2.2	External logic analysis	23
2.2.3	Internal/embedded logic analysis	24
2.2.4	Basic features in embedded logic analysis	26
2.3	Related work on embedded logic analysis	34
2.3.1	Programmable trigger engines	34
2.3.2	Assertion checkers in hardware	35
2.3.3	Trace compression	37
2.3.4	Data restoration in combinational circuits	39
2.3.5	Centralized/distributed trace buffers for core-level data acquisition	42
2.3.6	System-level debug techniques	45
2.3.7	Concluding remarks on related works	46
3	Resource-efficient and programmable trigger units	47
3.1	Preliminaries	47
3.1.1	Trigger event mapping with comparators	48
3.1.2	Trigger event mapping with equality units	48
3.1.3	The concept of under-triggering and over-triggering	52
3.2	New architecture of a resource-efficient programmable trigger unit	55
3.3	Algorithm for trigger event mapping	62
3.4	Experimental results	65
3.4.1	Area analysis on the proposed architecture	65
3.4.2	Analysis on the proposed algorithm	66
3.4.3	Miss trigger analysis for the proposed solution	68
3.5	Summary	74
4	Algorithms for state restoration	75
4.1	Algorithmic solution for state restoration	76
4.2	Exploiting bitwise parallelism for state restoration	79

4.3	Experimental results	84
4.3.1	Experiments with randomly generated stimuli on all inputs	86
4.3.2	Experiments with constrained generated stimuli for control inputs	89
4.4	Summary	95
5	Automated trace signal selection	97
5.1	Trace signal selection using circuit topology	98
5.2	Trace signal selection using topology and logic gate behavior	105
5.3	Experimental results	108
5.3.1	Experiments with randomly generated stimuli on all inputs	108
5.3.2	Experiments with constrained generated stimuli for control inputs	115
5.4	Summary	118
6	Distributed embedded logic analysis	119
6.1	Motivation	120
6.2	Proposed design-for-debug methodology	120
6.2.1	Challenges	121
6.2.2	Architecture	124
6.3	New on-chip hardware circuitry for enabling the proposed methodology	127
6.3.1	Handling of simultaneous, overlapping, and overflow sample requests (Features A-C)	128
6.3.2	Data sampling before trigger (Feature D)	130
6.3.3	Out-of-order data offloading (Feature E)	131
6.3.4	Data transfers between buffers (Feature F)	132
6.3.5	Programmable priority (Feature G)	134
6.4	Experimental results	137
6.4.1	Case studies	137
6.4.2	Analyzing area investment of the proposed architecture	139
6.4.3	Analyzing data acquisition of the proposed architecture	145
6.5	Summary	147

7 Conclusion and future work	148
7.1 Summary of dissertation contributions	148
7.2 Possible future research directions	150
Bibliography	153

List of Tables

3.1	Results for analyzing the trigger event mapping algorithm	68
4.1	Two bit codes for data representation	80
4.2	Number of bitwise operations for the two-input primitive gates	85
4.3	State restoration results for s38584 when trace signals are selected randomly and control signals are driven randomly	87
4.4	State restoration results for s38417 when trace signals are selected randomly and control signals are driven randomly	87
4.5	State restoration results for s35932 when trace signals are selected randomly and control signals are driven randomly	88
4.6	Signals with constrained values	93
4.7	State restoration results for s38584 when trace signals are selected randomly and control signals are driven deterministically	94
4.8	State restoration results for s35932 when trace signals are selected randomly and control signals are driven deterministically	94
5.1	State restoration results for s38584 when all primary inputs are driven randomly and trace signal are selected using the topology-only metric	109
5.2	State restoration results for s38417 when all primary inputs are driven randomly and trace signal are selected using the topology-only metric	109
5.3	State restoration results for s35932 when all primary inputs are driven randomly and trace signal are selected using the topology-only metric	110
5.4	State restoration results for s38584 when all primary inputs are driven randomly and trace signal are selected using the topology+logic metric	110

5.5	State restoration results for s38417 when all primary inputs are driven randomly and trace signal are selected using the topology+logic metric	111
5.6	State restoration results for s35932 when all primary inputs are driven randomly and trace signal are selected using the topology+logic metric	111
5.7	State restoration results for s38584 when the control inputs are driven deterministically and trace signal are selected using the topology-only metric	115
5.8	State restoration results for s35932 when the control inputs are driven deterministically and trace signal are selected using the topology-only metric	116
5.9	State restoration results for s38584 when the control inputs are driven deterministically and trace signal are selected using the topology+logic metric	116
5.10	State restoration results for s35932 when the control inputs are driven deterministically and trace signal are selected using the topology+logic metric	117

List of Figures

1.1	VLSI design flow and verification techniques	2
1.2	Scan infrastructure for manufacturing test	8
2.1	Scan infrastructure for manufacturing test and post-silicon validation based on [115]	17
2.2	Example of a breakpoint control unit	19
2.3	Example of a clock control unit and its timing diagram for different operating modes	20
2.4	Software debug flow	22
2.5	Example of an embedded logic analyzer	24
2.6	Debug flow when using the trace-based technique with embedded logic analyzer	25
2.7	Level-sensitive trigger event detection	27
2.8	Edge-sensitive trigger event detection	29
2.9	Event sequencing	30
2.10	Sample before triggering	31
2.11	Segmentation in trace buffer	32
2.12	Streaming of data from trace buffer	33
2.13	ELA using outputs of assertion checkers for triggering	37
2.14	ELA with real-time trace data compression	39
2.15	Principal operations for data restoration	41
2.16	Example for data restoration in combinational logic	42
2.17	DFD architecture for trace-based technique on core-based SOCs	44

3.1	Debug flow in FPGAs	48
3.2	Example of trigger event mapping with comparators	49
3.3	Example of trigger event mapping with equality units	51
3.4	Example on minimization of logic function using K-map	52
3.5	Timing diagram for under-triggering	53
3.6	Timing diagram for over-triggering	54
3.7	New architecture of a resource-efficient programmable trigger unit for level sensitive trigger events	56
3.8	Example of trigger event analysis	58
3.9	Example of trigger event being missed	59
3.10	New architecture of a resource-efficient programmable trigger unit with support for event sequencing	61
3.11	Example of mapping three ON primes to two equality units	63
3.12	Area investment analysis when varying the number of equality units	65
3.13	Area investment analysis when varying the level of event buffers	67
3.14	Miss trigger analysis when varying the number of equality units with frequent triggering	69
3.15	Miss trigger analysis when varying the number of equality units with non-frequent triggering	70
3.16	Miss trigger analysis when varying the level of event buffers with frequent triggering	71
3.17	Miss trigger analysis when varying the level of event buffers with non-frequent triggering	72
4.1	Sample circuit for state restoration	76
4.2	Derivation of forward and backward equations for the <i>AND</i> gate	81
4.3	Experimental flow for randomly generated input stimuli	86
4.4	Procedure for identifying signals with constrained value	91
4.5	Revised experimental flow	92
5.1	Equations for restorability calculation using only topology	99
5.2	Restorability calculation for three iterations using the topology metric	102

5.3	Equations for restorability calculation using topology + logic	106
5.4	Restorability calculation for three iterations using the topology + logic metric	107
6.1	The proposed design-for-debug architecture based on distributed em- bedded logic analysis	123
6.2	Sampled data organization in trace buffers using multiple segments .	125
6.3	The allocation unit	128
6.4	Example of overwriting low priority data during overflowing of trigger events	129
6.5	Example of using windows to support sampling before trigger	131
6.6	Out-of-order offloading	132
6.7	Example of how data transfer between buffers can maintain acquisition bandwidth	133
6.8	Example of how data transfer between buffers can reduced data seg- mentation	135
6.9	Example of data sampling with different priority settings	136
6.10	MPEG-2 decoder implementation for case studies	138
6.11	Area investment analysis when varying the number of cores	140
6.12	Area distribution among hardware components when varying the num- ber of cores	141
6.13	Area investment analysis when varying the size and number of trace buffers	142
6.14	Area investment analysis when varying the organization of trace buffers	143
6.15	Area investment analysis when varying the number of trace ports . .	144
6.16	Area distribution among hardware components when varying the num- ber of trace ports	144
6.17	Impact of various features on acquisition of debug data	145
6.18	Effect of programmable priority on data loss	146

Chapter 1

Introduction

Modern process technologies enable very large scale integrated (VLSI) circuits to be built with multi-million transistors. This increased design complexity is one of the main reasons why existing verification techniques are insufficient for eliminating design errors (or bugs) before the design is manufactured (i.e., *pre-silicon*)[69]. The aim of the work described in this thesis is to provide new structured methods for assisting designers to identify design errors after the design is manufactured (i.e., *post-silicon*). To better illustrate how the proposed methods can be beneficial in the development of VLSI circuits, it is essential to understand the design flow and the state-of-the-art verification techniques. The VLSI design flow will be outlined in Section 1.1. The verification techniques, *pre-silicon verification*, *manufacturing test* and *post-silicon validation*, used in the VLSI design flow will be discussed in Sections 1.2, 1.3 and 1.4 respectively. Finally, the contribution and organization of this thesis will be given in Section 1.5.

1.1 VLSI design flow

The design flow of VLSI circuits is broken down into three stages: *specification*, *implementation* and *manufacturing* as shown in Figure 1.1. The specification gives the expected functionalities of the design. It can be described using high-level description

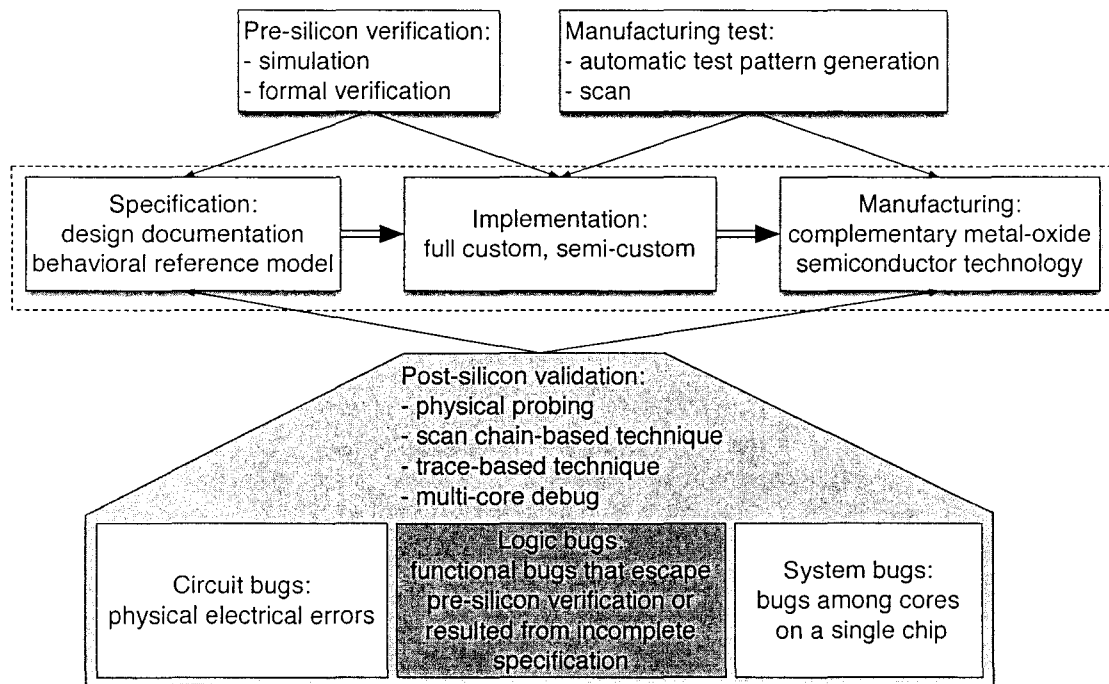


Figure 1.1: VLSI design flow and verification techniques

languages such as SystemC [36]. At this level of design abstraction, the circuit is described as a set of operations using data computations and transfers without detailing the actual implementation of the main components, such as processors, memories and buses in the design. Thus, the size and complexity of the specification is reduced. To give more detail about the implementation of a circuit, the specification can be given in hardware description languages (HDLs) like VHDL [84], Verilog [85] or System Verilog [104]. When using HDLs, the register-transfer level (RTL) abstraction is used to describe the expected functionalities as a set of transfer functions, which detail the flow of data between registers. Any logical operations performed on the data in-between registers are also specified. Using the RTL description of a design, logic synthesis can be performed to transform and optimize the transfer functions

into logic equations evaluated by gate-level components available from the targeted technology library [108]. These logic equations (i.e., implementation of the design) are then recorded in the gate-level netlist [79].

As shown in Figure 1.1, a digital circuit can be implemented using different design methodologies such as full custom and semi-custom using cell libraries and gate arrays. In the full custom design methodology, the designer creates a custom layout of the gate-level components, as well as the interconnect of these components in order to maximize circuit performance and minimize area. However, the amount of time required to create full custom designs becomes unreasonably long for complex designs. On the other hand, the circuit can be implemented using gate-level components found in standard cell libraries or gate arrays. This is done by using computer-aided design (CAD) tools to transform the RTL description into gate level netlist. These tools also translate the gate level netlist into the physical layout [23]. Although these tools help reduce design time, the synthesized circuits may not have optimal performance with minimum area.

Using the information in the physical layout, the integrated circuit can be manufactured [13]. The fabrication process gradually creates the integrated circuit through a sequence of photographic and chemical processing steps on a wafer made of semiconducting material and/or mixture of other metals. While silicon is the most commonly used semiconducting material in VLSI circuits and complementary metal-oxide semiconductor (CMOS) is the main type of transistors, different semiconductor technologies create transistors in specific geometries (e.g., 90 nm) [30].

To ensure the fabricated circuit operates correctly according to the specification, various checkpoints are placed in the design flow as shown in Figure 1.1. Within each checkpoint, different verification techniques are employed as will be detailed in the following three sections of the thesis.

1.2 Pre-silicon verification

To ensure the implemented circuit behaves correctly according to the specification, pre-silicon verification techniques are used extensively to eliminate design errors before the design is manufactured. The two main types of pre-silicon verification techniques are simulation and formal verification [69].

The most commonly used simulation technique is the event-based simulator. It operates by taking events one at a time and propagating them through the design. However, the problem with all the simulation techniques is that in order to conclude that the simulated design is 100% error-free, the simulation will have to take all possible behaviors of the system into consideration. As the number of possible behaviors increases exponentially with the number of inputs and the number of circuit states, exhaustive simulation becomes impractical for circuits of moderate size [91]. Thus, simulation techniques have evolved to use testbenches that drive the design-under-test with constrained-random or coverage-driven input stimuli. These testbenches target to verify a design only up to an acceptable simulation coverage. Recently, assertions are being increasingly used to verify the intended behaviors of a design.

Formal verification has emerged as a supplementary approach to simulation by making propositions with regard to the complete behavior of the design using mathematical proofs [54, 67]. There are two main methods of formal verification techniques: formal model checking and formal equivalence checking. While formal model checking uses mathematical techniques to verify behavioral properties of a design, formal equivalence checking uses mathematical techniques to verify equivalence of a reference design and a modified design. These designs may be obtained from the circuit descriptions from different levels of design abstraction (i.e., RTL or gate level) [107]. As formal verification techniques gain more attention in the verification community of VLSI designs, techniques for automating this process have emerged [31]. However, there is one key limitation with formal verification techniques: the design is only verified against the specification from which the circuit model is derived. Thus, the correctness of the verification will be compromised if the specification is faulty itself.

Using simulation and formal verification techniques together has proven significant in eliminating design errors during pre-silicon verification. However, error-free first silicon still cannot be guaranteed using only pre-silicon verification techniques due to the inherent tradeoffs between verification time and state coverage, and the limitations on modeling all physical characteristics of the design at higher levels of design abstraction [46].

1.3 Manufacturing test

Fabrication anomalies of integrated circuits in the manufacturing process may cause some circuits to behave erroneously. Manufacturing test helps to detect the physical defects that lead to faulty behaviors of the fabricated circuits. Thus, manufacturing test is the verification of circuit fabrication against its intended implementation in the VLSI design flow [79].

Manufacturing test checks for the proper operation of a fabricated circuit by testing the internal chip nodes using input vectors. The corresponding circuit responses are then compared to the expected responses for pass/fail analysis. If the circuit fails the test, the fault diagnosis process can be started to identify the root cause of the failure. The process of applying the input vectors and comparing the corresponding responses for the circuit-under-test (CUT) is usually controlled by an automatic test equipment (ATE). The input vectors and expected responses are generated according to the two types of test methods: functional test and structural test.

Functional tests verify the functionality of the CUT by exercising all the circuit functions. Similar to performing exhaustive simulation during pre-silicon verification, this requires a complete set of test patterns on all the circuit inputs. For a circuit with n inputs, the number of input vectors will be 2^n . For example, a 64-bit ripple-carry adder will have 2^{129} input vectors. To apply the complete test set to the CUT using an ATE, it would take 2.158×10^{22} years, assuming that the tester and circuit can operate at 1 GHz [20]. Due to the exhaustive nature of complete functional tests, testing time is prohibitively large for logic blocks, which makes them infeasible for testing complex digital integrated circuits. As a result, proposals on applying functional test

at operational speed for testing parts of a circuit were recently introduced [21, 113].

On the other hand, structural tests depend on the netlist structure of a design. Depending on the logic and timing behavior of electrical defects, different fault models are introduced to allow automatic test pattern generation (ATPG) algorithms to be developed for test generation, test application and test evaluation. Some typical fault models are single stuck-at fault model, bridging fault model and delay fault model. These fault models capture the behaviors of physical defects on silicon into the logic domain, such that they can be detected using structural tests. The most commonly used fault model is the single stuck-at fault model [89]. It assumes a single line of the logic network to be stuck at a logic 0 (s-a-0) or logic 1 (s-a-1). When using the single stuck-at fault model for the 64-bit ripple-carry adder, only 1728 stuck-at faults would need to be excited with 1728 test patterns in the worst case scenario [20].

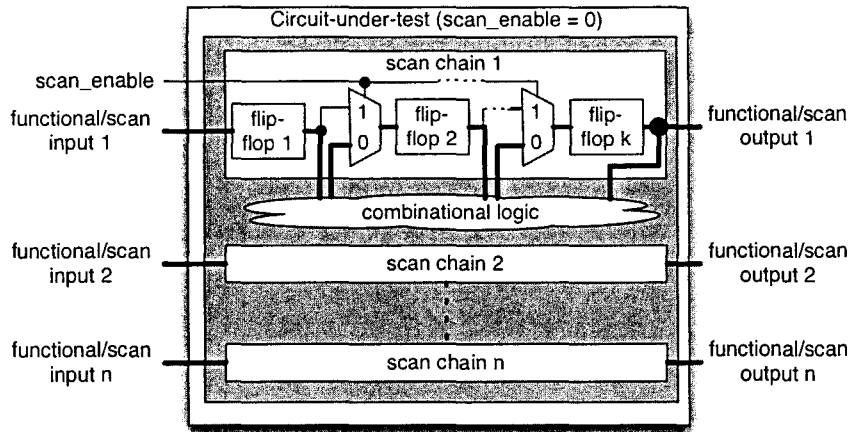
It is common for VLSI designs today to have internal state signals which cannot be easily controlled from primary inputs or observed at primary outputs. As a result, to achieve high fault coverage during manufacturing test, design-for-test (DFT) circuitries are inserted to improve controllability and observability of internal nodes in the CUT. The most common DFT methodology used is the scan method, whose infrastructure is shown in Figure 1.2. Using this method, all or parts of the internal state elements (i.e., flip-flops) are replaced with scan flip-flops (SFFs) by inserting a multiplexer at the input of each flip-flop. During normal functional operations, the scan_enable signal is inactive and the circuit data flows in-between the state elements and combinational logic normally as shown in Figure 1.2(a). When in the test mode, the scan_enable will be set to connect the SFFs together to form one or more shift registers called scan chains, as illustrated in Figure 1.2(b). During test application, input test vectors are shifted into the scan chains serially through the scan inputs to initialize the CUT into a known state, while circuit responses in the internal states are shifted out of the scan chains via the scan outputs for analysis [20]. Due to the ability to provide controllability and observability of internal state elements, test generation for a circuit with scan allows the use of the simpler combinational ATPG, which also generates a smaller number of test patterns, when compared to sequential ATPG. This indirectly reduces the cost of manufacturing test. As a result, a number

of improvements on how scan chains are built have been proposed in the literature [25, 82, 94]. Also, scan chains have been adopted for performing structural tests with various fault models [86, 97, 121]. Although it is not the focus of this thesis, the author had also published three papers on building scan chains at the RTL ([55, 59] during his master studies) and ([65] during his doctoral studies). He had also published four papers on creating power-constrained scan chains for the delay fault model [60–63] during his early doctoral studies before he switched his research focus.

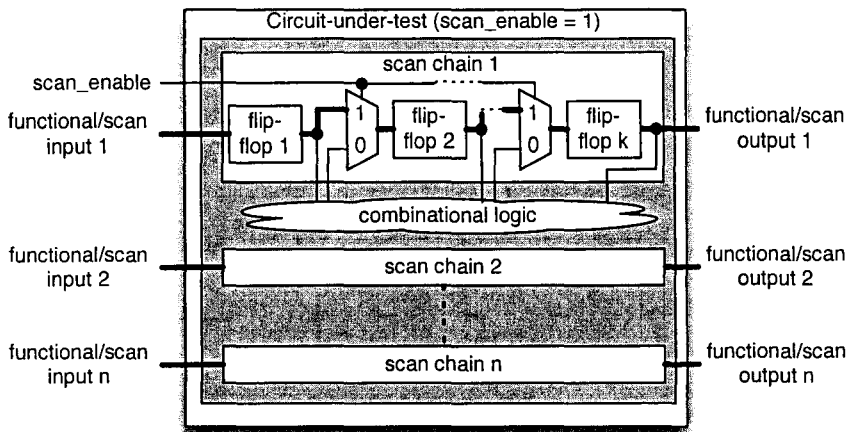
Manufacturing test has been established to be an essential stage in the VLSI design flow for verifying the fabricated circuit against the implemented design. However, manufacturing test relies on the circuit netlist as the reference for detecting physical defects that lead to faulty behaviors in the fabricated circuit. If design errors escape the pre-silicon verification stage, they will also exist in the circuit netlist, and thus, cannot be detected by using only manufacturing test. As a result, it must be augmented by post-silicon validation techniques to identify these escaped errors in the fabricated circuit.

1.4 Post-silicon validation

The limited accuracy in circuit modeling and the exponential size of state spaces are the main reasons why pre-silicon verification is insufficient in eliminating all the design errors before tape-out. This problem is further aggravated by the increasing number of on-chip logic blocks and the complex transactions between them, as well as the continuous growth of embedded software as the product differentiating component in system-on-a-chip (SOC) designs. In order to reduce the cost of re-spins (both mask costs and the implementation time), it is essential to identify the escaped bugs as soon as the first silicon is available [119]. We call this step in the implementation flow *post-silicon validation* and it is focused on identifying and localizing bugs in silicon. However, the inability to access internal signals in the silicon, which results in limited observability of the circuit, becomes the major obstacle in post-silicon validation. Hence, a number of proposed solutions in this area focus on addressing this obstacle. It is important to note that depending on the type of errors one is trying to locate,



(a) Scan operation during normal functioning operations



(b) Scan operation during manufacturing test

Figure 1.2: Scan infrastructure for manufacturing test

different information should be acquired from the design. Thus, we introduce three types of bugs that are concerned during post-silicon validation. They are *circuit bugs*, *logic bugs*, and *system bugs* as shown in Figure 1.1.

1.4.1 Circuit bugs

We define circuit bugs as errors that arise from the mismatches between circuit models in different design abstraction levels, as well as the effects from the use of deep sub-micron technologies on signal integrity and semiconductor manufacturing process variations. To tackle these circuit bugs, which are mainly electrical bugs that drive the yield to an unacceptably low level, physical probing techniques such as [12, 83, 88, 99, 100] can be used to probe the internal wires to extract their electrical behavior in the silicon. However, due to the rising complexity of current SOCs, a localization step, which compares the simulation data and the information acquired in the silicon using design-for-debug (DFD) hardware, is needed to identify the subset of circuit nodes that should be physically probed [114].

1.4.2 Logic bugs

In addition to localization, the data acquired by the DFD hardware can also be used to identify logic bugs, which are functional errors that have escaped the pre-silicon verification stage. One technique for acquiring functional data in the silicon is the scan chain-based technique, which will be introduced in SubSection 1.4.6. One drawback with the scan chain-based debug technique is that the circuit has to be stopped when data is shifted out of the scan chains. This prevents data to be acquired in real-time. As functional bugs can span thousands of clock cycles [51], it is beneficial to maintain circuit operation during scan dumps. Although double buffering the scan cells (which may incur unacceptable area [52]) can achieve this goal, data sampling at consecutive clock cycles, which is essential for timing-related bugs, is still not possible. In this case, the *trace-based* technique, which employs on-chip memories for at-speed data sampling, can be used. Details on the trace-based technique, and the various proposed methods for enhancing such a technique will be discussed later in this thesis.

1.4.3 System bugs

Finally, system bugs are errors that exist among multiple cores in a SOC. As multiple cores interact with each other when software is executed on the system, identifying such bugs requires acquisition of data among the interrelated cores. This poses a different set of challenges in the design of DFD hardware for the system when compared with locating circuit bugs or logic bugs in a single core. For instance, the DFD infrastructure has to be designed while taking into consideration triggering multiple cores at the right time, and acquiring data both in the cores and on the interconnects between cores.

It should be noted that techniques for detecting logic bugs can aid the localization of errors in the design such that physical probing techniques can be effectively applied for the detection of circuit bugs. Also, system bugs in multiple cores cannot be tackled unless the infrastructure for locating logic bugs in a single core is in-place. Thus, in this thesis, we focus our contributions to aid data acquisition for detecting logic bugs during post-silicon validation.

To better understand the contributions of the works presented in this thesis, it is essential to first define the terminologies *verification*, *test*, *validation*, *debug* and *diagnosis* used in the context of VLSI design. Also, the two main phases of experiments called *non-deterministic replay* and *deterministic replay* performed during post-silicon validation should be explained. After that, the two most commonly used DFD techniques: scan chain-based and trace-based technique during post-silicon validation will be briefly introduced.

1.4.4 Terminology definitions

The five terms that are widely used in different contexts in the VLSI design flow are: *verification*, *test*, *validation*, *debug* and *diagnosis*.

Verification The task of verifying a VLSI design is to check whether the behaviors of the design match with the functionalities defined in the specification. Thus, verification deals with functional errors that reside in the logical domain. For

instance, as discussed in Section 1.2, pre-silicon verification techniques are used to check the implemented design against the specification for logic errors.

Test When testing an integrated circuit, the implemented design is used as a golden reference model to check whether there are any physical defects caused by fabrication anomalies in the manufacturing process. This is done usually in the manufacturing test stage of the VLSI design flow using structural tests accompanied by the insertion of DFT hardware, as elaborated in Section 1.3.

Validation When the final design is put to the actual operating environment, it is validated to see if the design performs what it is intended to do. This is where post-silicon validation techniques and the inclusion of DFD hardware can help. When erroneous behaviors are detected, the debug process can be started to locate the bugs in the design.

Debug The notion of debug can be further divided into pre-silicon debug and post-silicon debug. In pre-silicon debug, the circuit description is checked against the specification to identify the location of the bugs in the design using pre-silicon verification techniques. In post-silicon debug, DFD hardware can be used to gather information about the operation of the design in order to localize the bugs within the design.

Diagnosis When a bug has been localized, diagnosis helps to identify the root-cause of the bug. In pre-silicon diagnosis, the information from pre-silicon techniques such as simulation is analyzed to determine whether the failure is caused by incorrect implementation of the specification, or if it results from incomplete specification. On the other hand, when a design fails in post-silicon validation, the problem can be caused by functional errors that have escaped pre-silicon verification. Another reason for having misbehaviors in the silicon can be the presence of electrical errors or fabrication defects that have escaped manufacturing test. They can also be system errors that arise when the chip is put to the actual operating environment under specific voltage, temperature and/or frequency.

1.4.5 Non-deterministic and deterministic replay experiments

The post-silicon validation process consists of two main phases.

Non-deterministic replay In the first phase, the error manifests itself on the application board and its occurrence cannot be reproduced immediately in a deterministic environment. This happens when the error is triggered by non-deterministic input sources of the design. Two examples of these input sources are asynchronous interfaces between buses and interrupts from peripherals [96]. In this phase, the objective is to understand and isolate the input behaviors that cause the bug.

Deterministic replay After the behaviors are isolated, the error can be triggered in a controlled environment and the post-silicon validation process can move on to the next phase. In this controlled environment, the input behaviors are reproduced on the ATE or the application board to trigger the error deterministically. This allows the debug experiment to be run repeatedly in order to locate the root cause of the bug.

In either phase of the post-silicon validation process, it is important for one to be able to gather as much data as possible from the design in order to gather information for understanding the nature of the error. This is especially important for non-deterministic replay experiments since one may not be able to reproduce the erroneous behavior again. As a result, in this thesis, we focus on the techniques for improving observability of a design during post-silicon validation for non-deterministic replay experiments. The increased observability resulted from the use of DFD hardware can help engineers to be more confident in concluding if an erroneous behavior has occurred, as well as to help locate the bug during the debug process. In the next two subsections, the two most commonly used DFD techniques will be briefly discussed.

1.4.6 Scan chain-based technique

Reusing the internal scan chains, which is the most widely used technique to increase observability of a circuit during manufacturing test, is the primary goal in

scan chain-based technique. It first captures all the internal state elements using the scan technique in a design when a specific breakpoint condition occurs. After that, the captured data can then be offloaded through the scan chains for failure analysis. One problem with scan chain-based technique is that it will not be able to acquire data in real-time during post-silicon validation. This is because the circuit has to stop and then resume its execution during scan dump. Since functional bugs can sometimes appear in circuit states that may be exercised thousands of clock cycles apart [51], it is therefore desirable to maintain circuit execution during scan dumps. Although this can be overcome by double buffering the scan elements, it will lead to a substantial area penalty [52]. Even if this penalty would be acceptable, data sampling in consecutive clock cycles using only the available scan chains will not be possible. However, this ability to acquire data continuously is an essential requirement for identifying timing-related problems in a design during post-silicon validation.

1.4.7 Trace-based technique

To be able to acquire data in real-time during post-silicon validation, one can connect the signals of interest directly to the device pins so that it can be monitored by external logic analysis equipments [39, 40]. However, the difficulty of driving device pins with high internal clock frequencies and the limited number of available pins used only for the purpose of post-silicon validation makes external logic analysis insufficient for complex SOC designs. As a result, internal/embedded logic analysis has emerged as a complement to the scan chain-based technique.

Integrating the functionalities of logic analyzers into the CUD using additional hardware is the idea behind embedded logic analysis. Inside embedded logic analyzers (ELAs), a trigger unit can be programmed to monitor desired trigger conditions to determine when to initiate data sampling on a small set of internal signals in real-time using on-chip trace buffers. The acquired data can then be transported through low bandwidth device pins, such that post-processing algorithms can analyze the acquired data and identify design errors off-chip [72]. When data on more signals from a different number of clock cycles can be obtained through the scan-based technique in

the same experiment, the two sets of data can better aid the identification of logic bugs during post-silicon validation. As a consequence, embedded logic analysis has emerged as a popular solution for debugging microprocessors [43, 105], designs on field programmable gate arrays (FPGAs) [6, 110, 122], application specific integrated circuits (ASICs) and complex SOCs [3, 44, 72].

1.5 Contributions and organization of the thesis

In this section we summarize the main contributions of this thesis, by outlining its organization. We have tackled four complementary problems: improving the effectiveness of triggering in embedded logic analysis (Chapter 3); speeding up the expansion of the acquired debug data (Chapter 4); providing automated techniques to guide the selection of signals that should be traced (Chapter 5); and understanding how to manage multiple on-chip trace buffers dynamically at runtime (Chapter 6).

Before introducing the contributions done by the author, Chapter 2 provides the background material on scan chain-based and trace-based techniques employed for post-silicon validation. A review of the related works focused on the use of ELA for improving real-time observability for in-system debugging will also be given in this chapter.

During post-silicon validation a mechanism must be provided to control the acquisition of debug data. As a result, additional hardware has to be inserted in the CUD for detecting any specific events to determine when data should be sampled. However, due to the limited real estate available on the silicon, the inserted hardware has to be simple and small, which limits the amount of functionalities one can make available in the trigger unit. In Chapter 3, we investigate how to reduce the amount of logic in the trigger unit for ELA, without compromising its ability to monitor complex trigger conditions. This is achieved by a resource-efficient architecture which, in order to reduce the number of events checked concurrently, monitors simplified logic conditions on trigger signals that include both true and false trigger events. When false trigger events are identified through on-chip trigger analysis, the false decisions can be reverted in real-time. The contribution from this chapter will appear in [58].

Despite the numerous DFD techniques that have been proposed in the literature on improving the ability to acquire data on-chip, only a small amount of data can be captured either by using scan chains, or stored in on-chip trace-buffers such that the data can be transferred off-chip later for data analysis. This limited observability of internal signals may lengthen the post-silicon validation process. In Chapter 4, a technique to better utilize the limited storage available during data acquisition is introduced. By consciously selecting the trace signals when designing the ELA, we will show how one can restore a significant amount of missing data from the state elements that are not traced.

The decision on which signals should be hardwired to the trace buffers has to be made when designing the ELA at design time. However, it is not always possible for the designer to predict which signals may help provide more information about the design during post-silicon validation. As a result, Chapter 5 introduces algorithmic solutions for trace signal selection. By analyzing structural information of a design, we will show how metrics can be developed to guide the selection of trace signals. When coupled with the technique from Chapter 4, the sampled data from these chosen signals can better help reconstruct missing information in the CUD. The contributions from Chapters 4 and 5 are jointly published in two papers [56, 64].

As complexity of SOC designs continues to increase, data acquisition using one centralized trace buffer inside the ELA becomes insufficient. For instance, when multiple cores in the system notify that their trigger conditions are met and data acquisition should be started for all these cores, a centralized trace buffer will not be able to satisfy the high data acquisition bandwidth requirement for core-based SOCs. In Chapter 6, a novel methodology for improving the real-time observability when distributed ELAs will be introduced for SOCs with an increasing number of internal cores. In addition, when high-speed trace ports are made available, the proposed methodology can utilize these trace ports to further improve real-time observability during post-silicon validation. The work from this chapter of the thesis is published in [66].

Finally, Chapter 7 summarizes the contributions of this research and provides directions for future work.

Chapter 2

Background and related work

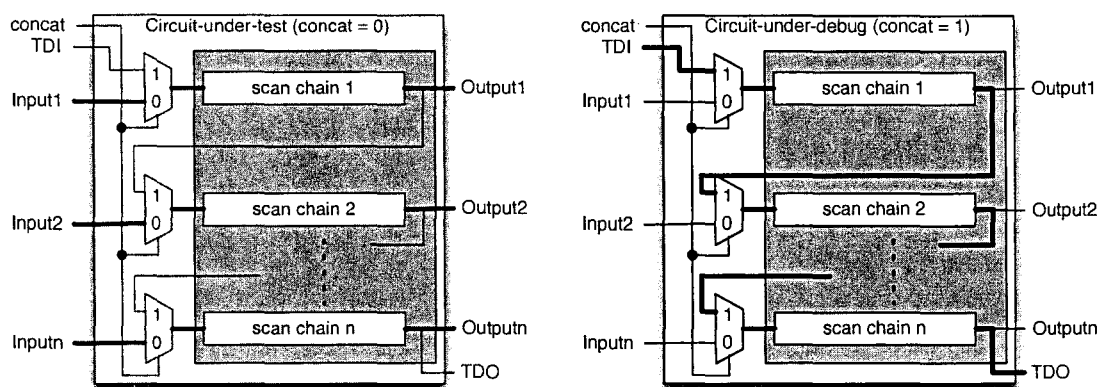
To address the problem of limited observability during data acquisition when performing post-silicon validation, a number of solutions had been developed. In this chapter of the thesis, the advantages and disadvantages of some of these solutions will be discussed. Section 2.1 details the scan chain-based technique. Section 2.2 elaborates on the trace-based technique. Finally, Section 2.3 gives the related works on embedded logic analysis, which is the DFD technique on which the work in this thesis was built.

2.1 Scan chain-based technique

Reusing the internal scan chains, which is the most widely used technique to increase observability of a circuit during manufacturing test, is the primary goal in scan chain-based technique [29, 37]. To reuse the scan chains for acquiring data during post-silicon validation, additional hardware is inserted to allow the scan infrastructure to support three basic features: scan for access, breakpoints and clock control.

2.1.1 Scan for access

During manufacturing test, the SFFs are connected together to one or multiple shift registers called scan chains. When test is applied in the test mode, the input test



(a) Scan access during manufacturing test (b) Scan access during post-silicon validation test

Figure 2.1: Scan infrastructure for manufacturing test and post-silicon validation based on [115]

vectors are shifted into the scan chains through scan input pins. At the same time, the circuit responses are shifted out of the scan chains via scan output pins for pass/fail analysis, as shown in Figure 2.1(a). In order to keep testing time short, multiple scan chains are shifted simultaneously. When more scan chains are available, a smaller number of SFFs will be in a chain, and thus, takes less time to be scanned. However, the decision of how many scan chains should be employed is limited by the available IO pins. As a result, in order to reduce pin usage, the scan pins are usually time-shared with the functional pins on the chip. This is because during scan in manufacturing, the CUT will be stopped and functional data will not be supplied to the circuit [92].

On the other hand, functional data needs to be supplied to the CUD during post-silicon validation. As a result, the access mechanism of the scan infrastructure has to be modified as shown in Figure 2.1(b). In this infrastructure, the scan chains are concatenated to form a single scan chain. This chain is accessed through low bandwidth dedicated pins such as the test data in (TDI) port in the JTAG interface [47]. Whenever data is captured into the scan chain, they can be offloaded via the

test data out (TDO) port for data analysis using techniques such as latch divergence analysis [28] or failure propagation tracing [24] to identify the failing state elements. This information can then be further analyzed for finding the root-cause of the error.

2.1.2 Breakpoints

It is obvious that using the scan chain-based technique, debug data should only be captured into the scan chains and subsequently offloaded when a particular event of interest has happened. This is because it would require multiple clock cycles to offload the data from the SFFs, and thus, consecutive data acquisition can only be initiated after each successful offload.

To reuse the scan chains for post-silicon validation, additional hardware needs to be inserted into the CUD in order to monitor the single event or the sequence of events so that data acquisition can be initiated at the right time. An example of such hardware called the breakpoint control unit is shown in Figure 2.2. It usually comprises one or more comparators that monitor a set of trigger signals. Examples of such trigger signals can be the program counters and the internal instruction/data buses [119]. When the values on the trigger signals match with the desired breakpoint conditions specified in the programmable registers, the breakpoint controller will be notified. To support more complex breakpoint conditions, different logical operators controlled by the breakpoint controller can be used with the comparators. When counters and/or sequencers are employed, the breakpoint controller will decide if the detected condition has happened a sufficient number of times according to the counters, or if the appropriate sequence of conditions has occurred as indicated by the sequencers. When the breakpoint controller concludes that the breakpoint condition is reached, the breakpoint flag will be raised.

The programmable registers that store the desired breakpoint conditions, and the configuration registers in the breakpoint controller for setting up the the logical operators, counters and sequencers are connected to a serial interface such as JTAG [47]. During post-silicon validation, different configurations can be programmed into the breakpoint control unit to acquire different sets of data in multiple experiments.

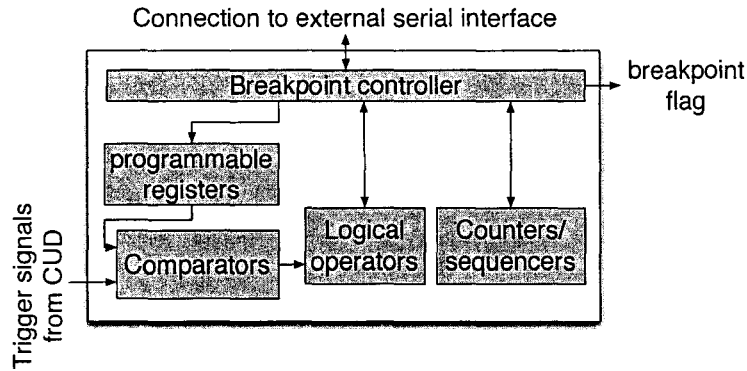
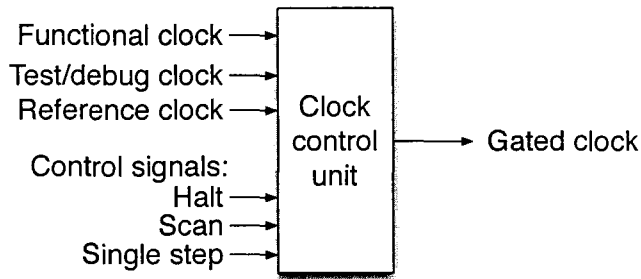


Figure 2.2: Example of a breakpoint control unit

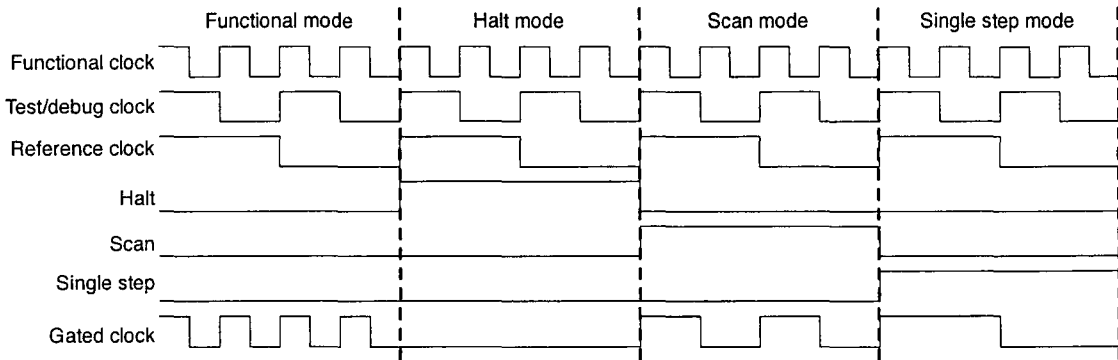
2.1.3 Clock control

After the breakpoint control unit detects the desired breakpoint condition and has raised the flag, the circuit has to be stopped in order to preserve the current state of operation and wait for further instruction. This can be done by providing a clock control unit which provides explicit controls to the clock signal that drives the state elements of the circuit. An example of a clock control unit is shown in Figure 2.3(a). The inputs to the clock control unit are control signals that indicate what functions are to be performed. The output of the unit is the gated clock signal that drives the circuit. By controlling the gated clock, the clock control unit should provide at least the following three functions during post-silicon validation [119] as illustrated in Figure 2.3(b):

- Halt: stop the on-chip clocks when the breakpoint is reached;
- Scan: the test/debug clock is passed to the gated clock for unloading the scan chains;
- Single step: generate one clock pulse at a time using the reference clock to step through the execution of the chip;



(a) Example of a clock control unit



(b) Timing diagram of the clock control unit in different operating modes

Figure 2.3: Example of a clock control unit and its timing diagram for different operating modes

When none of the control signals are activated, the output of the clock control unit should be the same as the functional clock to facilitate normal circuit operations.

Although there are proposals for improving controllability and observability of internal state elements using the scan chain-based techniques during post-silicon validation (e.g. [115]), one will not be able to acquire data in real-time using this technique during post-silicon validation. This is because the circuit has to stop and then resume its execution when offloading the contents from the scan chains. Since logic bugs can sometimes appear in circuit states that may be exercised thousands of clock cycles apart [51], it is therefore desirable to maintain circuit execution during

scan dumps. Although this can be overcome by double buffering the scan elements, it will lead to a substantial area penalty [52]. Even if this penalty would be acceptable, data sampling in consecutive clock cycles using only the available scan chains will not be possible. This is because multiple clock cycles are required for the data in the shadowed buffers to be scanned out before they can be used for capture again. However, this ability to acquire data continuously is an essential requirement for identifying timing-related problems during post-silicon validation.

2.2 Trace-based technique

The ability to acquire data on internal signals using trace buffers without interrupting circuit execution is key to enable real-time data acquisition using the trace-based technique. It has been used successfully for performing software debug with microprocessors in the past. This idea is extended for digital circuits by acquiring data through the circuit pins using external logic analyzers. However, as design complexity increases, circuit performance improves, and logic-to-pin ratio rises, the functionalities provided by external logic analyzers are being embedded into the design.

2.2.1 Software debug using traces

The goal in software debug is to identify errors in the software. There are two commonly used techniques as shown in Figure 2.4. The first technique (Figure 2.4(a)) is based on controlling the execution of the program through the use of breakpoints similar to the scan chain-based technique discussed previously for post-silicon validation. Using this technique, the debug engineer iteratively sets custom breakpoints to stop the program and examines the state of the application until the bugs are understood. One disadvantage with this technique is the difficulty for debugging real-time systems. This is because stopping the system may be insufficient for detecting timing-related bugs for which it is important to capture data in consecutive clock cycles.

An alternative technique (Figure 2.4(b)) is based on real-time software trace, in which the sequence of instructions and data accesses of the running application are

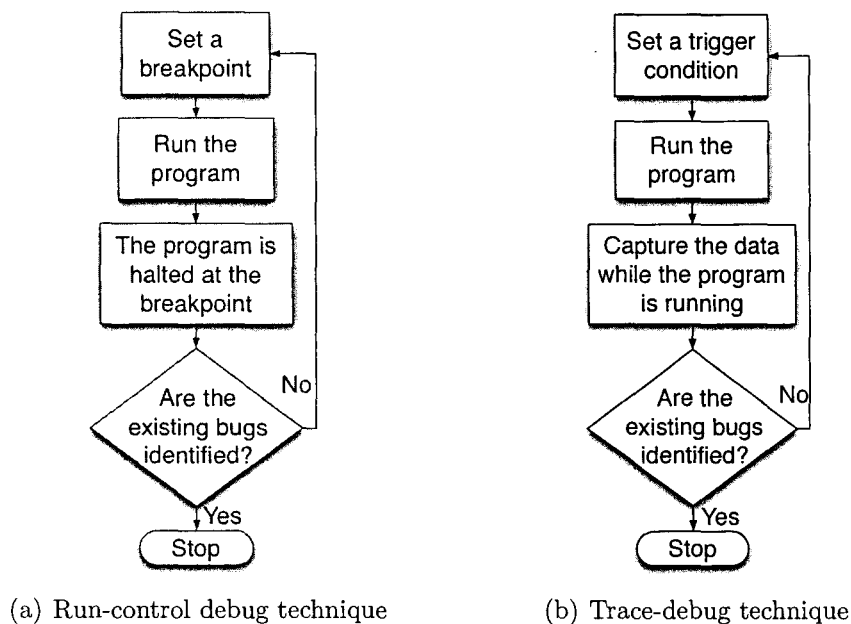


Figure 2.4: Software debug flow

captured after a certain triggering condition has occurred. The sampled data can then be compared against the expected outcome of the program. Since the execution of program during data capture is not interrupted, real-time debugging can be facilitated using this software trace technique. This technique is not only essential for debugging embedded systems [74], but also for performance analysis of microprocessors [103].

During the debug process, the debug engineer will have three parameters that can be changed to collect different information about the bug. The three parameters are *sampling frequency*, *triggering condition*, and *the amount of data to be acquired*. The sampling frequency determines how often data is acquired during debug (e.g., sampling at each clock cycle, or every other cycle). The triggering condition determines the point when acquisition of a trace should be started. The amount of data that should be acquired is limited by the size of the trace buffer that is used. It should be noted that the first two parameters are traditionally determined manually by the

debug engineer. On the other hand, the capacity of the trace buffer can be enlarged by utilizing automated software trace compression techniques [19, 50, 73].

It is interesting to note that the identification of logic bugs during post-silicon validation follows a similar approach as real-time debug of software using trace. For instance, to understand bugs in a SOC, the debug engineer will also like to acquire as much data as possible on the silicon after a specific event has occurred on the chip. To provide controllability and observability of internal signals in the silicon during the debug process, logic analysis has emerged as a popular solution for debugging microprocessors [43, 68, 105, 120], designs on FPGAs [6, 41, 110, 122], ASICs and complex SOCs [3, 44, 72].

2.2.2 External logic analysis

When extending the trace-based technique from software debug to digital circuits during post-silicon validation, instruments called logic analyzers can be used. It provides controllability and observability by accessing and monitoring the CUD through physical connections to the device pins. Inside the logic analyzer, storage space is provided for storing the input patterns to control the CUD, and for depositing the sampled data obtained from the chip. As digital circuits become more complex, capabilities of logic analyzers also improve. By using a modular approach in maintaining a common mainframe, engineers have the flexibility to specify different test systems and apply various algorithms for analyzing the acquired data [39, 40]. Also, by combining multiple modules in the logic analyzer, simultaneous data acquisition on multiple data sources with speed up to 100MHz, asynchronous timing analysis at 1GHz and parametric analysis using the oscilloscope module for standalone or built-in analogue circuitries can be performed [26].

To obtain meaningful data through the limited number of device pins using logic analyzers, additional effort is required from the designer to plan ahead on determining which control and data signals should be connected to the device pins. In the case when there are not enough pins available, data will have to be formatted in order to be passed into/out of the CUD for complex designs during post-silicon validation.

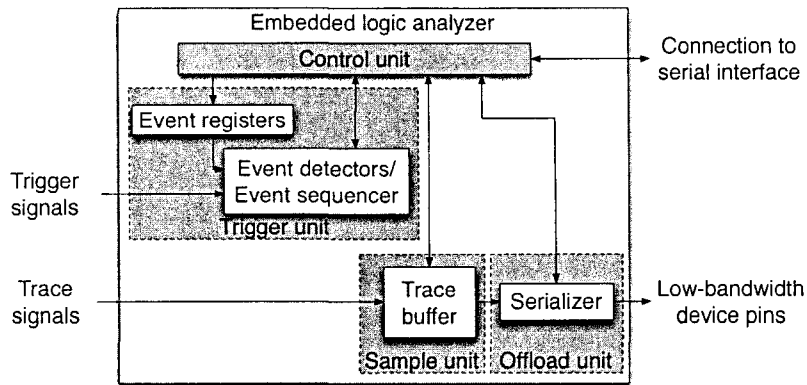


Figure 2.5: Example of an embedded logic analyzer

There is another problem when controlling and observing the CUD directly from device pins using logic analyzers. As the performance of digital circuits improves, it becomes increasingly difficult to drive the device pins at a high operating frequency. As a result, the idea of embedding the functionalities of logic analysis on-chip emerges [5, 27, 74, 75, 117].

2.2.3 Internal/embedded logic analysis

Integrating the functionalities available in external logic analyzers into the CUD is the idea behind embedded logic analysis. An example of an ELA is shown in Figure 2.5. The ELA can be divided into four components: control unit, trigger unit, sample unit, offload unit.

The control unit monitors the trigger unit, sample unit and offload unit during post-silicon validation. It contains one or more finite state machines (FSMs) with programmable registers. The programmable registers can be configured using a serial interface like JTAG [47] for receiving control instructions. This allows the FSMs to control the other units in the ELA to gather different sets of data in multiple experiments during post-silicon validation. The trigger unit has one or multiple event

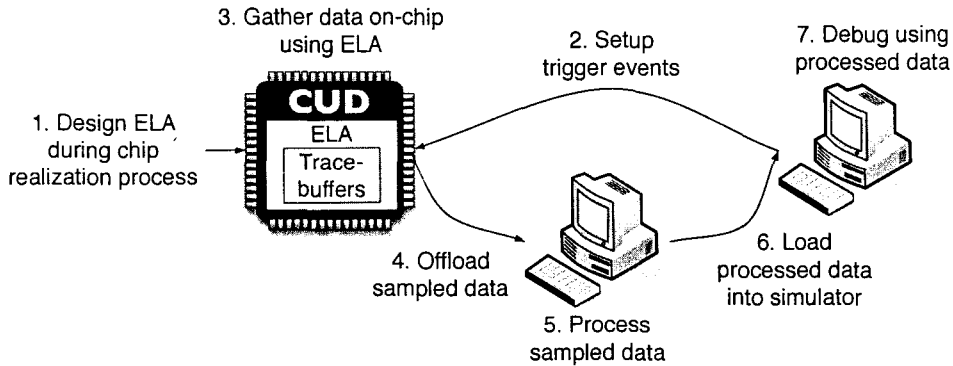


Figure 2.6: Debug flow when using the trace-based technique with embedded logic analyzer

detectors and/or event sequencers for detecting desired trigger conditions on the set of connected trigger signals. When the trigger conditions are matched, the control unit will be notified. To allow different trigger conditions to be detected in multiple experiments, the event registers can be configured by the control unit during post-silicon validation. The sample unit is responsible for acquiring data when it is notified by the control unit. To provide real-time observability, the sample unit contains an internal trace buffer such as embedded memories that can be clocked at the same operating frequency of the circuit. It acquires data on a subset of internal signals called trace signals. When the trace buffer is full, the control unit will be notified. At which point, the offload unit can be told to initiate data transfer for unloading the sampled data. This can be done by using a serializer to reformat the data such that low-bandwidth device pins can be used [81].

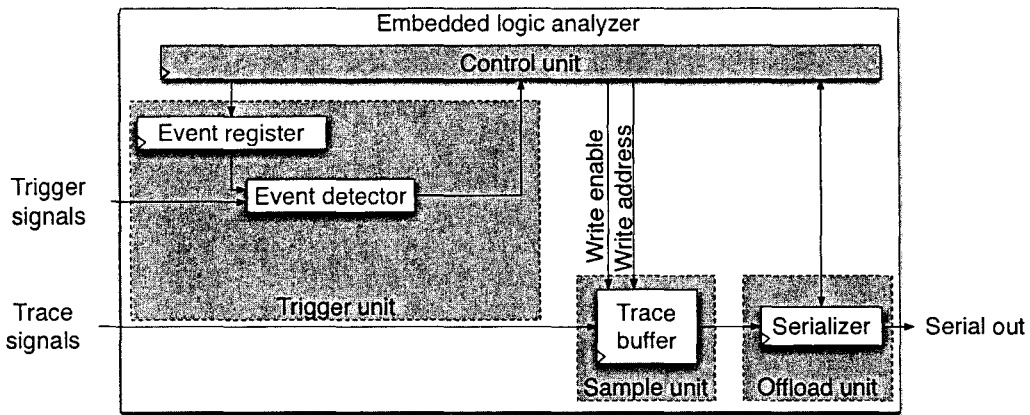
The debug flow for using the trace-based technique with ELA is shown in Figure 2.6. The first step of this approach is to design the ELA during the chip realization process. This includes determining the types of event detectors (bitwise, comparison, logical operations) to be used and designing the event sequencers in hardware. Also, the decision on which signals should be monitored by the trigger unit as the trigger signals, and which signals should be connected to the sample unit as the trace signals has to be made at this time [6, 122]. When the circuit manufactured with the ELA

is being validated under actual operating conditions, the debug engineer initiates the debug cycle by first setting up the trigger conditions. Then, the CUD can be put into the operational mode, and the ELA will monitor the trigger events, upon when data will be sampled in real-time into the on-chip trace-buffers. The sampled data is then subsequently transferred off the chip via a low bandwidth interface to a post-processing stage [81]. This stage includes organizing the sampled data such that it can be fed to a simulator, where the debug engineer can analyze the data to identify logic bugs. It has been shown in [95] that a complex design can contain tens to hundreds of bugs. As a result, it is very likely that the debug engineer will have to iterate steps 2 to 7 in Figure 2.6 during debug to gather the additional data for identifying all the concerned bugs. Also, since the set of trigger signals and trace signals in the ELA are determined at design time, it is not uncommon that the debug engineer will have to redesign the ELA and manufacture the modified CUD such that different data can be acquired for debug. However, this will require the debug engineer to perform more iterations of the entire debug flow, and thus, lengthening the total debug time.

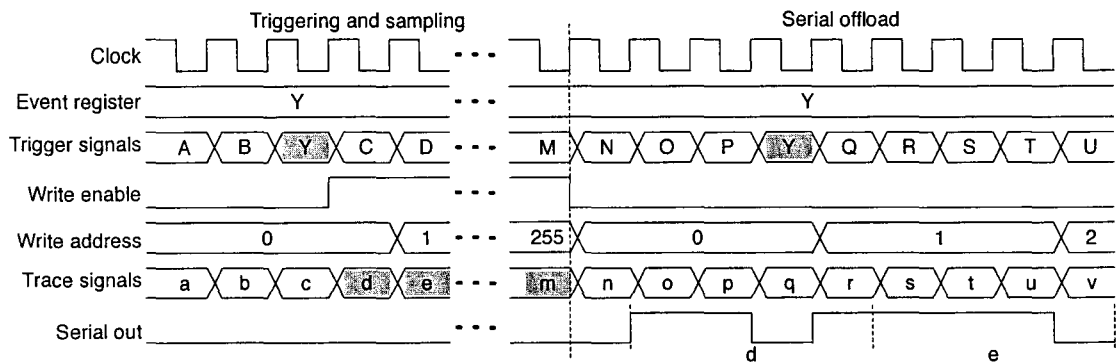
2.2.4 Basic features in embedded logic analysis

In the ELA shown in Figure 2.5, the three components: trigger unit, sample unit and offload unit, affect the total debug time. This is because these three components together control what data, and how much data can be obtained in one iteration of the debug flow. The trigger unit helps by identifying only the important data that is of interest. The sample unit can be modified to use larger buffers to store more data. And the offload unit can employ faster or more trace ports to unload more data from the trace buffer, so that storage space can be reclaimed for further data acquisition within a single experiment during post-silicon validation.

To help better utilize the limited storage space in the sample unit for improving real-time observability during post-silicon validation, the design of ELA should support the following basic features: level-sensitive trigger event detection, edge-sensitive trigger event detection, event sequencing, sample before triggering, segmentation in trace buffer, and streaming of data from trace buffer.



(a) Implementation of ELA



(b) Timing diagram

Figure 2.7: Level-sensitive trigger event detection

Level-sensitive trigger event detection

Level-sensitive trigger event detection requires the trigger unit to notify the control unit in the ELA when the trigger signals reach a specified value in any clock cycle. This can be explained using Figure 2.7, where Figure 2.7(a) shows the implementation of the ELA, and Figure 2.7(b) gives the timing diagram of the signals during triggering, sampling and offloading of data. In this ELA, the event detector can be implemented using a simple equality comparator between the event register and the

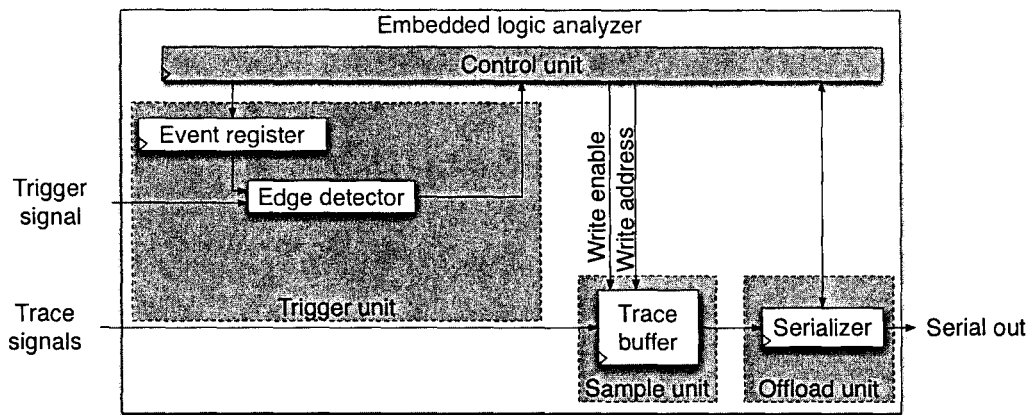
trigger signals. When the value on the trigger signals matches with the event register, write enable to the trace buffer is asserted and the write address will be updated by the control unit for data sampling as indicated by the grey area in Figure 2.7(b). When the trace buffer is full, write enable will be reset, and the sampled data can be offloaded serially. It should be noted that if a match is detected between the trigger signals and the event register during serial offload, data acquisition will not be initiated in order to prevent the sampled data from the previous match to be corrupted. This is illustrated in Figure 2.7(b) in the grey area when the value on the trigger signals matches with the event register for the second time.

Edge-sensitive trigger event detection

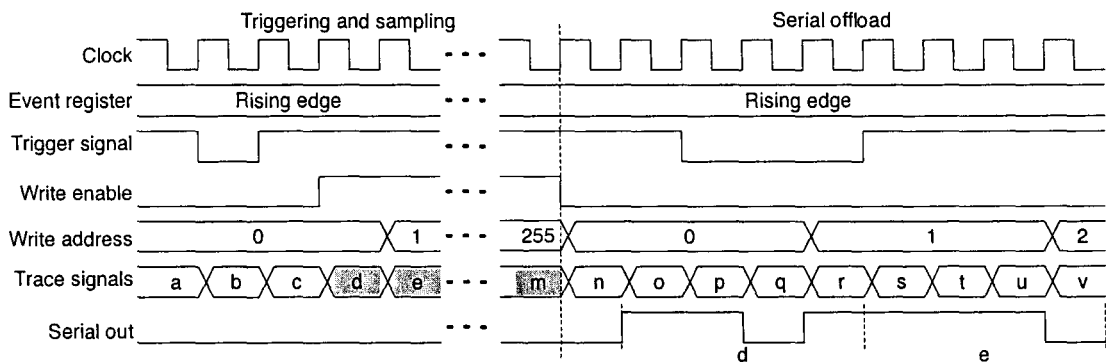
The implementation of the ELA for edge-sensitive trigger event detection is given in Figure 2.8(a). It is very similar to the implementation for level-sensitive trigger event detection. However, instead of using an event detector, an edge detector is used in this case. Each edge detector should be implemented to monitor one trigger signal for two clock cycles to detect the occurrence of a rising/falling edge. When the desired edge comes on the trigger signal, data acquisition will be started until the trace buffer is full as indicated in the grey area in Figure 2.8(b). After that, sampled data can be offloaded serially.

Event sequencing

When data acquisition should only start after a sequence of events occurs, the implementation of ELA in Figure 2.9(a) can be used. In this implementation, the ELA can accommodate two events using two sets of event registers and event flags. When the first event is detected, the first event flag will be set. When the second event comes, the second event flag will be set, and data acquisition will be initiated as illustrated in Figure 2.9(b). It should be noted that the second event flag will only be set after the previous event in the sequence has been detected as indicated with the first grey area on the trigger signals in Figure 2.9(b). Also, when a longer event sequence is desired, the ELA has to be re-designed with more hardware to accommodate the extra events.



(a) Implementation of ELA

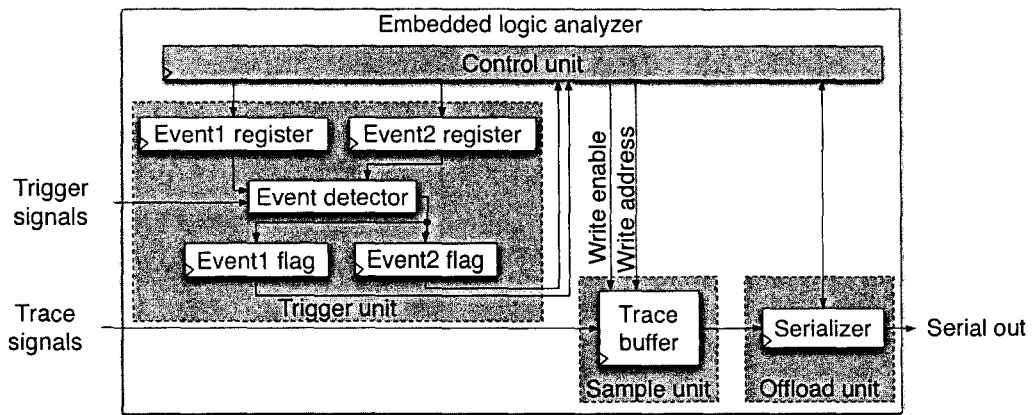


(b) Timing diagram

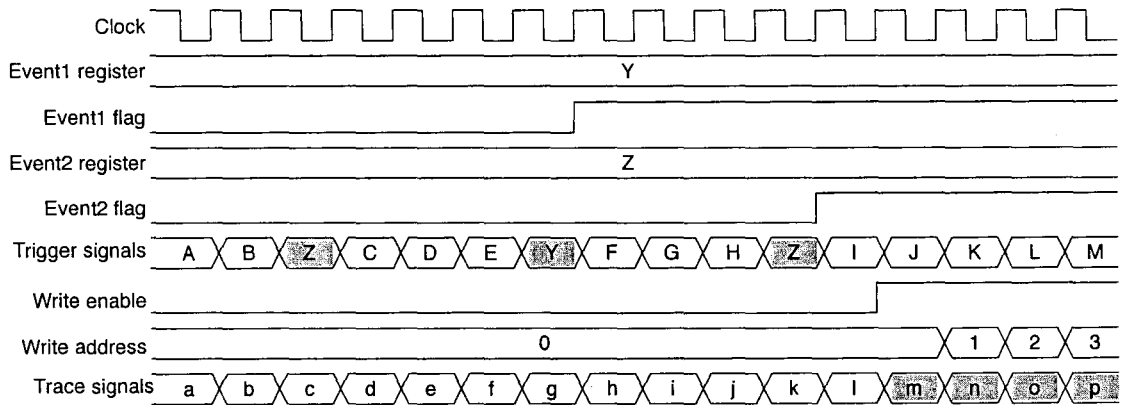
Figure 2.8: Edge-sensitive trigger event detection

Sample before triggering

The three features explained above are concerned about acquiring data after the desired trigger condition has been detected. As a result, the sampled data does not show the activities in the CUD that lead to the occurrence of the events. This is why the feature sample before triggering can be useful. The implementation and the timing diagram of the signals in the ELA that facilitate this feature are shown in Figure 2.10. As shown in Figure 2.10(b), the writing address is continuously incremented while



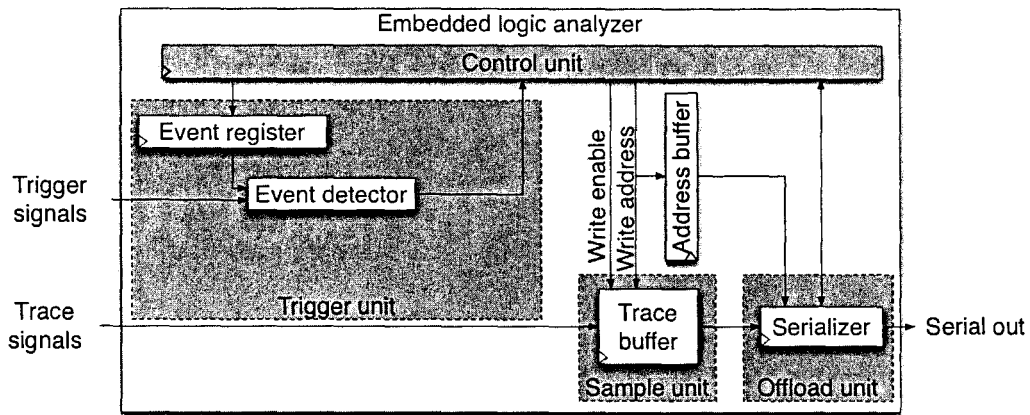
(a) Implementation of ELA



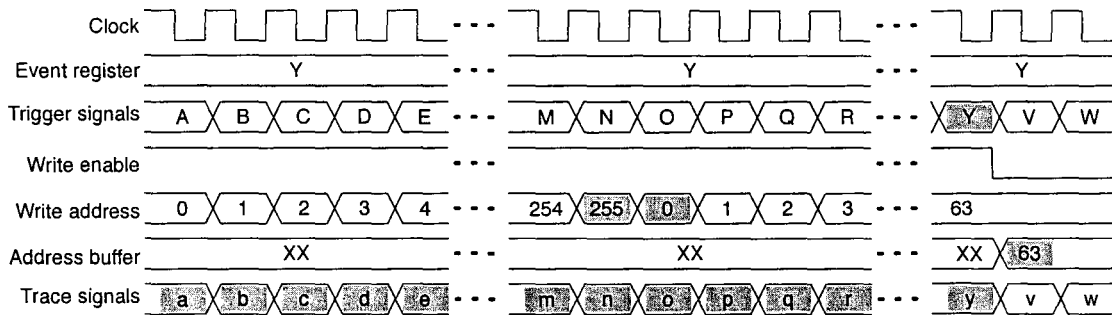
(b) Timing diagram

Figure 2.9: Event sequencing

the write enable is asserted in order to acquire data into the trace buffer before the desired trigger condition occurs. Note that the sample unit treats the trace buffer as a circular buffer by resetting the write address to the starting address during data acquisition. This will overwrite the old data that is captured in the trace buffer with the new data that was acquired more recently. When the desired trigger condition occurs, write enable is cleared and the address of the last written data will be stored into the address buffer. This address can be offloaded together with the sampled data



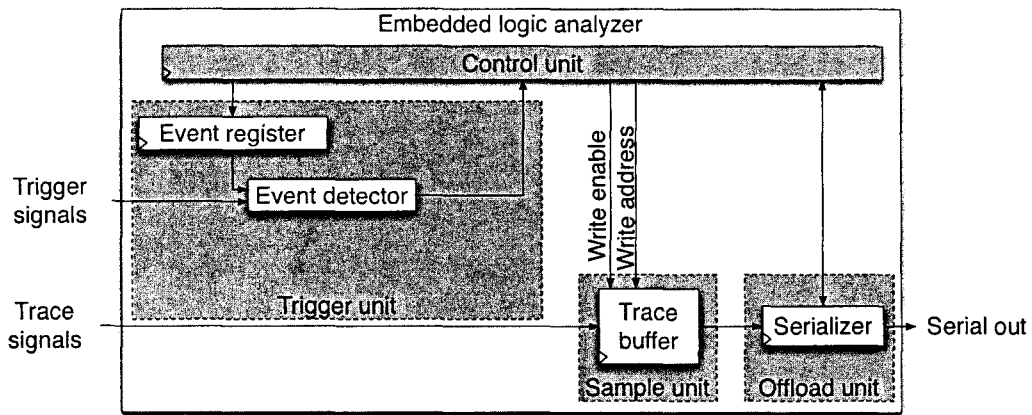
(a) Implementation of ELA



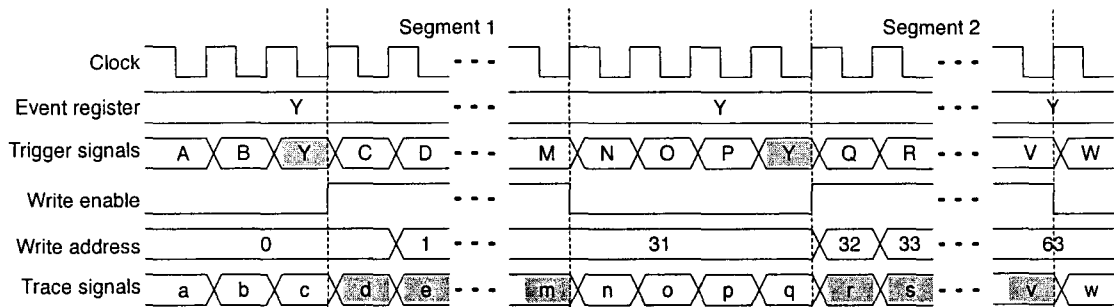
(b) Timing diagram

Figure 2.10: Sample before triggering

such that when analyzing the offloaded data, the behaviors of the CUD just before the occurrence of the trigger event can be observed. It should be noted that this feature can be further improved to acquire data sampling before and after the occurrence of an event. This can be done by allowing data acquisition to be continued for a short period of time after an event occurs.



(a) Implementation of ELA

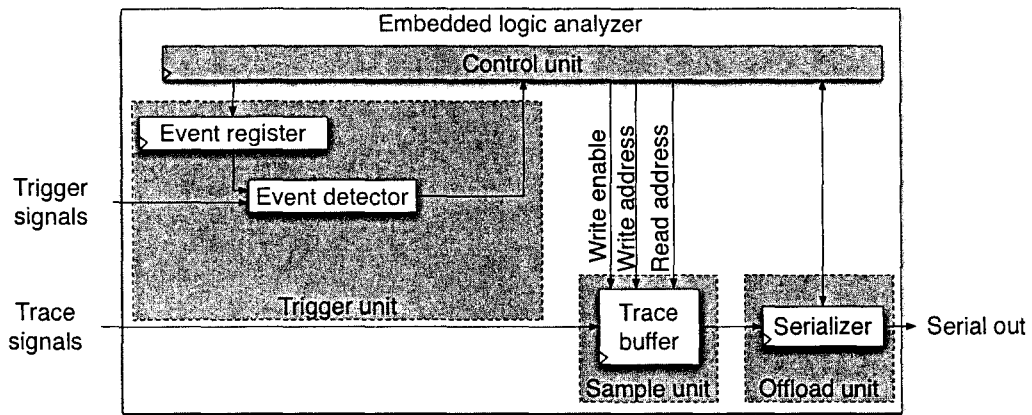


(b) Timing diagram

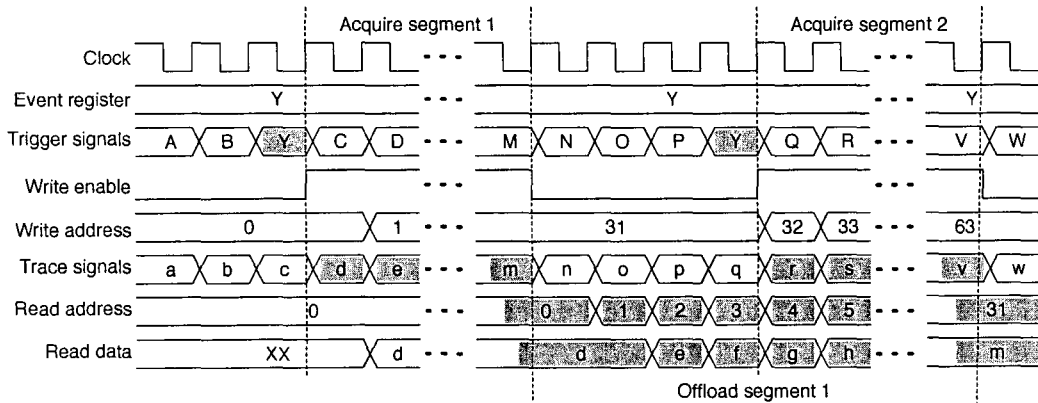
Figure 2.11: Segmentation in trace buffer

Segmentation in trace buffer

It is sometimes not necessary to acquire data using the whole trace buffer for the occurrence of only one trigger event. As a result, the trace buffer can be divided into multiple segments such that less data will be acquired for each event occurrence, while multiple sets of data for different appearances of the same event can be acquired as illustrated in Figure 2.11. Each time a match is detected between the trigger signals and the event register, data will only be acquired for a number of clock cycles defined by the user in the control unit. When enough data is stored, write enable will be



(a) Implementation of ELA



(b) Timing diagram

Figure 2.12: Streaming of data from trace buffer

reset and the write address will be kept constant. This allows data to be acquired in the following segment of the buffer when the desired trigger event occurs next time.

Streaming of data from trace buffer

When the trace buffer has an extra read data port that is controlled by a separate read address, writing and reading from different addresses can be performed simultaneously in the trace buffer. This allows sampled data in the trace buffer to be offloaded while

data acquisition is done in different segments of the trace buffer as illustrated using the ELA in Figure 2.12(a) and the timing diagram given in Figure 2.12(b). This capability to stream data from trace buffer can be beneficial because it helps reclaim valuable storage space from the trace buffer. This recovered space can then be used to acquire more data in the same experiment.

It is obvious that employing one or more of the above basic features in the ELA can help improve observability of internal signals in the CUD during post-silicon validation. However, in addition to providing more features in the ELA, other solutions that target a similar goal have been proposed in the literature. In the next section, the solutions most relevant to the contributions of this thesis will be discussed.

2.3 Related work on embedded logic analysis

In the ELA shown in Figure 2.5, the three components (i.e., trigger unit, sample unit and offload unit) affect the total debug time by controlling what data, and how much data can be obtained during post-silicon validation. As a result, different solutions that target various components in the ELA for improving observability of internal signals in the CUD are proposed.

2.3.1 Programmable trigger engines

For FPGAs, the user can redesign the FSM and its surrounding logic in the trigger unit based on the desired trigger conditions [122]. Therefore, any trigger events can be programmed. This gives the user the ability to only acquire the necessary data in each experiment during post-silicon validation. So long as the recompilation time is acceptable, full flexibility on designing the ELA can be achieved using FPGAs.

On the other hand, one will not have the freedom to modify the design with a new debug module for each experiment in ASICs. Instead, the debug engineer can only reprogram the configuration registers in the control unit and the event registers in the ELA shown in Figure 2.5. In this case, the set of trigger events that can be programmed is limited by the available hardware in the trigger unit.

In order to address this problem for ASICs, [3] introduced the idea of programmable trigger engines (PTEs). Instead of providing the full flexibility to redesign the FSMs as in an FPGA, small portion of logic with limited programmability called PTE can be placed onto the ASIC. The input to the PTEs can be a multiplexer network that is connected to different groups of trigger signals. The limited programmability of the PTE comes from the fact that it is a specialized instrument that can implement only one FSM of certain types. However, it also contains built-in counters, timers and comparators, so that it can be configured to detect more complex trigger conditions on a different set of signals for each experiment. Due to the smaller size and higher performance of the PTE, one can afford to employ one or more PTEs in a design for controlling data acquisition in various parts of the CUD simultaneously. The results of various PTEs may also be combined in order to describe trigger conditions that are based on different parts of the CUD.

To reduce manual effort being spent during post-silicon validation, automated solutions for generating the hardware of the trigger unit based on given trigger conditions were introduced [1, 118]. In [118], the trigger conditions are specified by the user using concise descriptions. These descriptions explain the trigger conditions in terms of a set of exact behaviors on the selected trigger signals. On the other hand, these behaviors are specified by the user through a graphical user interface based on a selection of available operations in [1]. The automated solutions then analyze the behaviors and translate them into boolean relations which can be implemented using the available hardware such as counters and comparators in the PTEs.

2.3.2 Assertion checkers in hardware

Another technique to monitor a design to detect incorrect behavior at runtime is using assertion. An assertion is a statement about a design's intended behavior (i.e., property), which must be verified. The hardware associated with an assertion should not be considered part of the design, as its sole purpose is to ensure consistency between the intended behaviors and what is actually created in the design [33].

Assertion-based verification has been extensively used during pre-silicon verification [33, 111]. The intended behavior specified by an assertion can be described using languages like Property Specification Language (PSL) [49] or SystemC [48] using a library called Open Verification Library (OVL) [4]. It can also be written in hardware description languages like VHDL [84], Verilog [85] and System Verilog [104], which can be embedded into the design during simulation in pre-silicon verification.

The benefits of using assertions for verification is multifold. Since assertions are embedded in the code, they help improve observability inside the design. In this case, the detection of misbehavior can be caught closer to the source of the bug. Thus, using assertions can help pinpoint the time and the location of the bug faster. And this eventually leads to reduced debug time and faster time-to-market. Another benefit of embedding assertions in a design is that they work simultaneously all the time, which is unlike conventional verification methods that require designers to modify their test cases to catch various incorrect behaviors. This is especially useful when developing and integrating intellectual property (IP) components. This is because the IP components can be self-checked by the assertions, and the user of the IP can quickly identify whether a problem belongs to the IP itself, or from incorrect usage of the IP. In addition, as assertions can be described using generally accepted design languages such as System Verilog, they are supported by various commercial simulators [22, 78, 106].

These benefits from assertions are being recognized and carried towards the process of post-silicon validation recently [15, 70, 111]. Since assertions can be described using various design languages for detailing the intended behaviors of a design, automated techniques were introduced to translate and implement assertions in hardware [14, 16, 17]. Using assertion checkers in hardware, the CUD can be monitored in real-time using real-time input stimuli. When combining assertion checkers and ELA as shown in Figure 2.13, the output of the assertions can be used to trigger data acquisition in the sample unit. When an assertion is violated, data can be captured in real-time traces for failure analysis.

Although hardware implementation of assertions helps improve observability inside the CUD during post-silicon validation, the cost of employing them can be quite

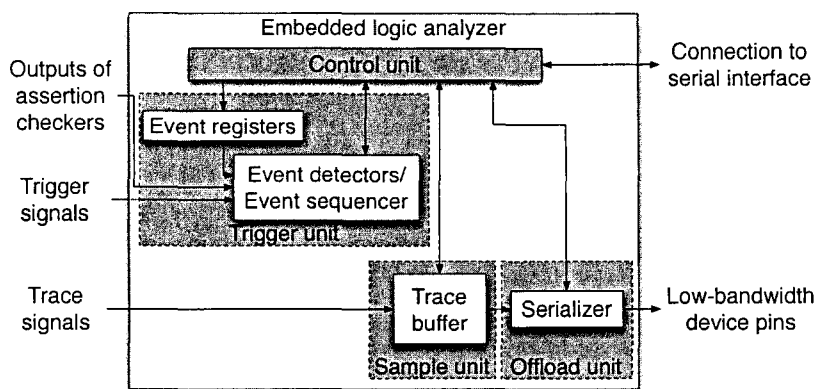


Figure 2.13: ELA using outputs of assertion checkers for triggering

high [34]. This is because it is usually desirable to employ multiple assertions for monitoring different behaviors in various parts of a design. Also, when multiple assertions are violated at the same time, there must be a mechanism present to decide which data sources associated with the individual assertions should be selected. Thus, additional silicon area, as well as design effort must be put in when employing assertions in hardware for post-silicon validation [34].

2.3.3 Trace compression

In addition to the techniques for improving the trigger units to better filter out unwanted data, a number of solutions have been proposed to improve the sample unit such that more data can be acquired. Inside the sample unit, the major component that consumes most of the silicon real estate is the trace buffer. When more real estate is allocated for the inclusion of trace buffers, more data can be acquired and thus, real-time observability of the CUD is improved. However, as the trace buffers are only employed for the purpose of post-silicon validation, there is a constant reluctance to invest additional silicon area for large trace buffers. As a result, techniques for compressing data in real-time before they are stored into trace buffers are proposed.

Trace compression techniques can be divided into two categories: microprocessor and random logic. When compressing trace data for microprocessors, [19] discussed a technique to implement value prediction-based compression (VPC) algorithms in software. [9, 87, 101] provide additional hardware to compress trace data before being stored in the trace buffer in real-time. Trace compression techniques for microprocessors exploit the unique characteristics of running programs. For example, the VPC algorithms predict values in program traces and values stored in registers by identifying patterns in value sequence from the past executed instructions on the processor. If the current value matches with any of the predicted values, a smaller encoded symbol for the current value will be stored instead [19]. On the other hand, [9] introduces additional hardware into the microprocessor to provide features such as trace filtering to eliminate unwanted data from the address and data buses. For example, if the data appears on the bus is a sequence of continuous addresses, one can filter out the sequential addresses and only store the non-sequential addresses.

[7, 8] provide two methods for compressing data traces in real-time for random logic circuits when using ELA during post-silicon validation. In this case, a real-time compressor is added into the sample unit of the ELA as shown in Figure 2.14. In [8], dictionary-based lossless compression algorithms are implemented using efficient hardware. When the current value can be matched with any of the values from the dictionary, the corresponding code will be stored into the buffer. Otherwise, the dictionary can be updated with the new value using various replacement schemes such as first-in-first-out (FIFO) or least recently used (LRU). In order to further extend the amount of data that can be stored on the trace buffer for random logic circuits that may have only a small amount of correlations between the data on trace signals, [7] introduces a signature-based lossy compression algorithm. Instead of storing every data into the trace buffer, their technique uses a multiple input signature register (MISR) to combine multiple sample data into a single signature. The generated signature will then be compared against a reference signature for failure analysis. If the signatures are not matched, a second experiment can be run with a different configuration to obtain only the data associated with the failed signature, and thus, effectively zooming into the failing data region. However, this technique requires the

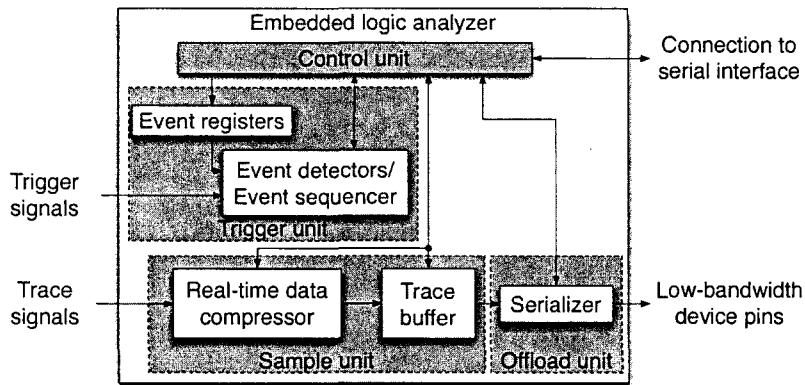


Figure 2.14: ELA with real-time trace data compression

debug experiments to be repeatable that when the same input stimuli are applied, the same behaviors will be observed in multiple experiments.

Although using any trace compression technique can increase the number of samples stored for each trace signal, the amount of signals that can be monitored by the trace buffers is still limited. As a result, important information on the signals that are not traced will be lost. To address this problem, it is desirable to find a way to reconstruct the missing data from those signals using the acquired data and the structural information of the CUD.

2.3.4 Data restoration in combinational circuits

When using the scan chain-based technique for capturing information in every state element and offloading them during scan dump for data analysis, observability of the CUD can be improved during post-silicon validation as discussed in Section 2.1. However, even if full scan (i.e., all state elements can be scanned) is employed, information on signals in the combinational logic between flip-flops is not acquired. This information may be able to help pinpoint problems in the CUD. To recover the information in the combinational logic gates, [44] proposed a technique to automatically

reconstruct the value on the signals using the sampled data from the state elements. This can be done by exploiting the logic behavior of the boolean gates.

It is noted that the combinational logic part of any digital circuit can be decomposed into a network of two input primitive gates (e.g. *AND*, *OR*, *XOR*, *NOT*). By exploiting the logic behaviors of these logic gates, two principal operations can be applied for data restoration. We call them *forward* and *backward* operations, and they are illustrated in Figure 2.15. These operations are very simple and have been discussed before in different perspectives (e.g. for ATPG) [20]. However, the descriptions of the operations are provided here as they are crucial for understanding the contribution of the works in this thesis.

A forward operation is applied to a gate when the input values to the gate are known, and it will try to determine the output value of the gate using boolean algebra. This is similar to what is normally done in functional simulators. An example is shown in Figure 2.15(a) using an *AND* gate and an *OR* gate. In the case of an *AND* gate, when one of the inputs is logic 0, the output can be concluded as logic 0 without analyzing the other input of the gate. On the other hand, when the output value of a gate is known, the backward operation can be applied to determine the values on the inputs of a gate. This is similar to backward justifying data in ATPG as elaborated with two examples in Figure 2.15(b). When the output of an *AND* gate is logic 1, both of its inputs can be immediately justified to be logic 1. Likewise, the inputs of an *OR* gate evaluate to logic 0 if the output is logic 0. In the case when the forward and backward operations are not sufficient, a combined method, during which both the known input and output values are evaluated, can be employed to reconstruct the missing value. This operation, which is explored for ATPG, is shown in Figure 2.15(c). In the *AND* gate, when the output and one of the inputs is known to be logic 0 and 1 respectively, the remaining input can be set to logic 0. The similar behavior can also be observed for the *OR* gate. Note that the same principles from boolean algebra on the forward and backward operations can also be applied to other primitive gates (i.e. *NAND*, *NOR*, *XOR*, *XNOR* and *NOT*). It is obvious that the principal operations from Figures 2.15(a), 2.15(b) and 2.15(c) will not always be able to reconstruct the missing values of a gate. For example, as shown in Figure 2.15(d),

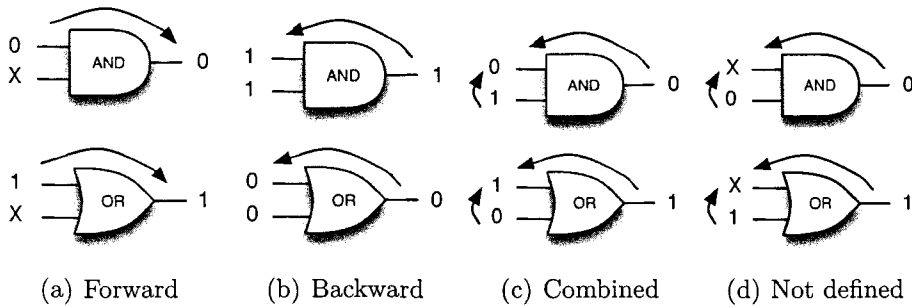


Figure 2.15: Principal operations for data restoration

when the output and the known input of an *AND* gate are both logic 0, there is insufficient information to conclude the missing value on the unknown input.

By applying the principal operations recursively, the data in the state elements can be forward propagated or backward justified among the logic gates in a circuit. This allows information to be reconstructed on signals in the combinational circuit. This recursive approach for data restoration using the principal operations is illustrated in Figure 2.16. In this figure, a 2-to-1 multiplexer is used as an example. The inputs s , $w0$ and $w1$, as well as the output f of the multiplexer are from flip-flops, which are scanned. However, the signals a , b and c in the combinational logic are not sampled. Using the forward operation and the data from the input signal s , the value of signal a can be found to be 0. Using this new value on a , the forward operation can be applied again to conclude the value of b is 0 through the *AND* gate. The value of c can also be obtained from input $w1$ to be 0 using the same operation. On the other hand, the values of b and c can also be backward justified using the data from output f . The value of a can also be reconstructed using the combined operation with the value of $w0$ and the new value of b .

It can be seen that when full scan is employed for the CUD, the values of all the signals can be reconstructed through multiple paths in the combinational logic since the inputs and outputs of the circuit are known. As a result, [44] introduced the idea of identifying the essential signals in the circuit. When only the essential signals are scanned, data from these signals can be used to restore values on all the other

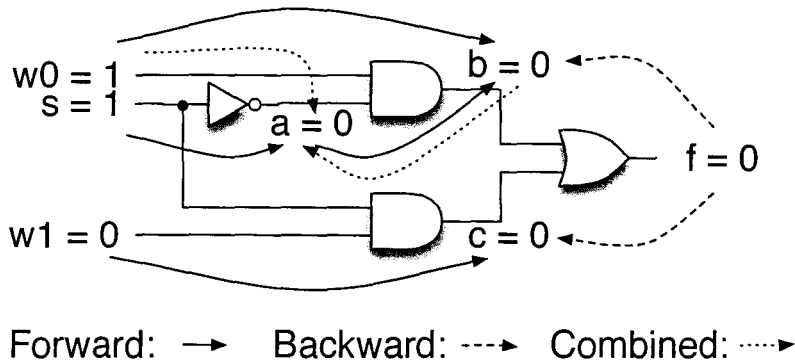


Figure 2.16: Example for data restoration in combinational logic

signals during data restoration. This maintains full observability of the design, while reducing the amount of data to be offloaded since less scan flip-flops will be employed.

One problem associated with the scan chain-based technique for post-silicon validation is the inability to capture data over consecutive clock cycles during post-silicon validation. This limits the acquisition of data in real-time for the identification of timing-related problems. As a result, the trace-based technique may be used for sampling data in real-time using trace buffers in the ELA as discussed in Section 2.1. In this case, the technique proposed in [44] will not be able to reconstruct real-time data for internal signals in multiple clock cycles. This is because their technique is only able to restore data for signals within the combinational logic of the CUD.

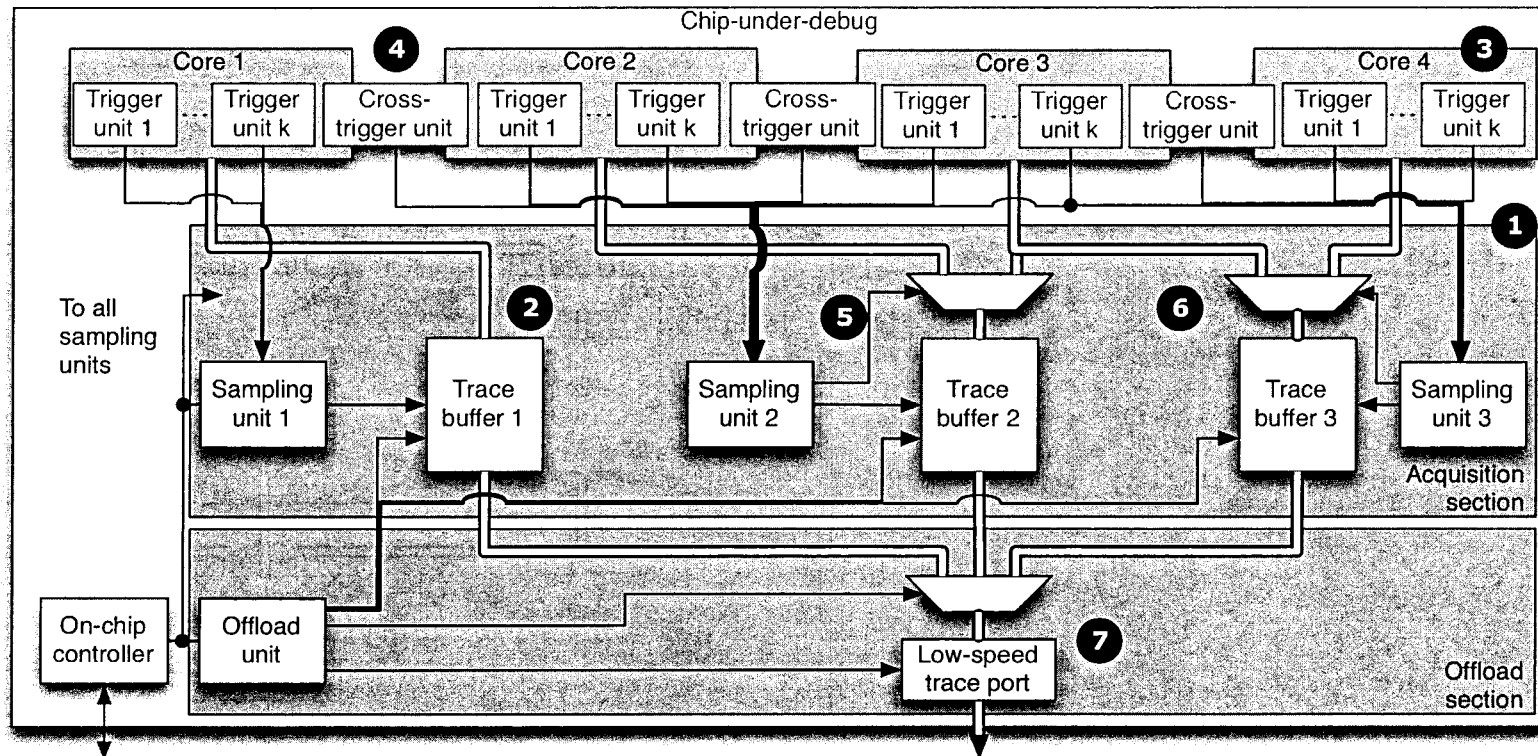
2.3.5 Centralized/distributed trace buffers for core-level data acquisition

The development of SOCs has evolved based on the design reuse philosophy, where two parties called core providers and system integrators are involved [53]. Core providers create pre-designed and pre-verified building blocks known as embedded cores, and system integrators can reuse these cores together with custom user-defined logic to

create SOCs [38]. As design complexity continues to increase, in order to include additional functionalities to satisfy consumers' needs, it is expected the number of cores in an SOC will keep on rising in the foreseeable future [11, 123].

Core-based design SOCs introduce new challenges in using ELA during post-silicon validation [42]. This can be illustrated using Figure 2.17. In this figure, a generic DFD architecture for using the trace-based technique on an SOC with four cores is given. As pointed out by *Label 1* in the figure, trace buffers are used to capture debug data at the operating frequency. Also, trigger units are employed to detect the occurrence of specific events and notify the sample units when data should be sampled [6, 57, 122]. The sampled data can then be offloaded through a low-speed trace port for data analysis. When only one core is present in the SOC, the single data source can occupy one trace buffer in the sample unit continuously (*Label 2*).

To deal with concurrent activity in multiple cores in SOCs, the idea of distributed triggering was discussed in [3, 90]. By placing trigger units, which are based on assertion checking and/or advance triggering conditions [17] (*Label 3*), in different parts of a design, different cores can be monitored simultaneously. This concept was put into practice in the Cell Broadband Engine, which has nine processor cores on a single chip [93]. Note, more advanced features, such as cross-triggering (*Label 4*) can be used [72]. Whenever the specified trigger event occurs, [93] employs a centralized trace buffer to store data (*Label 5*). In this case, when multiple trigger events occur simultaneously in multiple cores, it is unclear how data from different sources of the circuit can be acquired at the same time. As a result, the idea of static allocation of trace buffers among data sources are discussed in [76, 77, 98] (*Label 6*). However, since the trace buffers can only sample data from their dedicated sources, one may not be able to efficiently utilize the available storage space in *all* the trace buffers. For instance, consider that *Core 1* in Figure 2.17 generates sample requests frequently, while the other cores are rarely triggered. Since data from *Core 1* can only be sampled using *Trace buffer 1*, even if the other buffers have storage space available due to the inactivity of other cores, data from *Core 1* will still be lost when *Trace buffer 1* is full. At the end of each experiment, the trace buffers can be offloaded through a trace port such as the JTAG interface (*Label 7*) for post-processing.



- JTAG interface
- (1) Embedded logic analyzers with on-chip trigger units and trace buffers
 - (2) Trace compression techniques for compressing debug data on-chip to enhance observability
 - (3) Assertion and advance triggering
 - (4) Trigger on Core 1, sample from Cores 1 and 2
 - (5) Multiple data sources sharing a single trace buffer
 - (6) Multiple data sources mapped to multiple trace buffers, each data source sampled by only one available trace buffer
 - (7) Multiple trace buffers sharing one trace port to offload data

Figure 2.17: DFD architecture for trace-based technique on core-based SOCs

2.3.6 System-level debug techniques

In addition to gathering data at the core level, challenges also arise when debugging multi-core SOCs at the system level. For instance, since embedded cores communicate with each other during normal operation, gathering data for one core at a time will not be able to help tackle problems related to the entire system. In this case, concurrent data sampling on multiple cores, as well as monitoring the interactions among cores is crucial for system-level debug. However, it is very difficult to acquire and analyze such high amount of data. As a result, techniques such as trace qualification, trace filtering and trace compression can be incorporated into the debug architecture. These techniques are utilized in the debug platform introduced in [102] for structured ASICs, while [112] proposed another architecture for network-on-chip (NOC) designs.

In [102], dedicated on-chip instrumentation (OCI) blocks are inserted into the individual component cores. These OCI contain different trigger units tailored to the specific core. They can also include small OCI embedded trace for data sampling from the attached core. These embedded traces can then be channeled to a bigger trace memory located internally (on-chip memories) or externally (off-chip memories).

The platform in [112] focuses on debugging the interactions between cores in NOCs. In addition to individual core trace modules used for buffering data within the core, they employ core-level debug probes (DPs) in between every core and the network interface. The DPs will then monitor and record the transactions between cores at real-time. These components are controlled by a system-level debug agent (DA) using off-chip debug controller for data acquisition and data offloading through the available trace ports.

It should be noted that these system-level debug techniques suffer the same problem as what was mentioned in the previous subsection for core-level data acquisition using distributed trace buffers. This is because these techniques employ individual trace modules dedicated for each core or for interconnect between cores. In this case, if the amount of data generated for each trace module is not balanced, it is unclear how extra storage space can be allocated from other trace buffers to store the excess amount of data generated by the separate core. As a result, storage space on all the trace buffers are not utilized efficiently.

2.3.7 Concluding remarks on related works

To improve real-time observability of internal signals, ELA has become one of the most widely studied DFD techniques. Although recent advancements in the design of trigger units help better utilize the limited storage space in the trace buffer by identifying only the wanted data, the implementation of these complex trigger units results in a high investment in silicon area. As a result, it is desirable to develop efficient hardware that is still able to support complex trigger conditions.

To better utilize the trace buffer in the ELA, trace compression can be used. After the sampled data is offloaded, it can be used to restore information on signals that are never traced. However, the existing techniques can only reconstruct data within the combinational logic of the CUD.

The set of data one can acquire using ELA during post-silicon validation depends on which signals are connected to the trace buffer. This decision on how the trace signals should be wired in the ELA has to be made at design time. However, there are no existing techniques available to recommend which signals should be traced.

When multiple data sources are presented on a SOC, using one centralized trace buffer results in loss of information when more than one trigger unit request data acquisition simultaneously. Although using distributed trace buffers can be used to address this problem, the connections between data sources and trace buffers are predefined during design time. As a result, one may not be able efficiently utilize all the available space in the trace buffers during post-silicon validation.

In the remaining chapters of this thesis, the four contributions focus on the problems related to different challenges when using ELA during post-silicon validation will be discussed.

Chapter 3

Resource-efficient and programmable trigger units

The decisions on when to acquire debug data during post-silicon validation are determined by trigger events that are programmed into on-chip trigger units. In this chapter of the thesis, we first show how trigger events are mapped onto available hardware using examples in Section 3.1. We then investigate how to design trigger units that are both resource-efficient and runtime programmable. To achieve these two goals, we introduce new architectural features in Section 3.2, as well as algorithms for automatically mapping trigger events onto trigger units in Section 3.3. Experimental results for area analysis on the proposed architecture and analysis on the algorithm for trigger event mapping are given in Section 3.4, followed by concluding remarks in Section 3.5.

3.1 Preliminaries

For FPGAs, the debug flow using ELAs shown in Figure 3.1 can be employed [6, 122]. The user can redesign the trigger unit based on the desired trigger events. Therefore, any trigger event can be programmed, so long as the recompilation time is acceptable. On the other hand, one will not have the freedom to modify the design with a new debug module for each debug experiment in ASICs. Instead, the debug engineer can

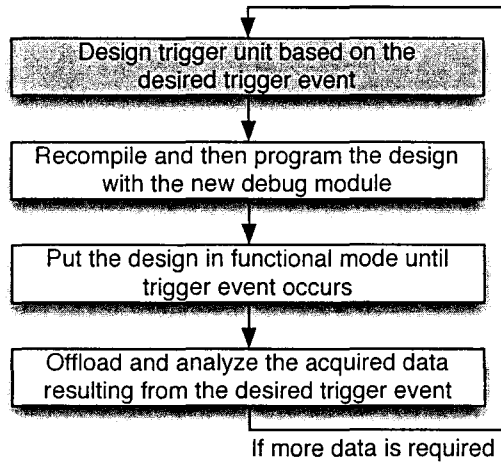


Figure 3.1: Debug flow in FPGAs

only reprogram the event registers in the trigger unit of the ELA shown in Figure 2.5. In this case, the set of trigger events that can be programmed during post-silicon validation is limited by the available hardware in the trigger unit.

3.1.1 Trigger event mapping with comparators

Figure 3.2 shows how a trigger event is mapped to a trigger unit with two comparators. The trigger event in Figure 3.2(a) is described by the user as a logic condition on the 4-bit trigger signals TS . The trigger unit given in Figure 3.2(b) monitors TS using two comparators. Using this trigger unit, the desired trigger event can be programmed into the event registers using the values in Figure 3.2(c). These values will configure the trigger unit to behave according to the timing diagram shown in Figure 3.2(d). When the value on TS is between 2 and 5 inclusive, the output of the trigger unit z will be active. This signals the control unit in the ELA to initiate data acquisition.

3.1.2 Trigger event mapping with equality units

When the provided hardware in the trigger unit is changed, the same trigger event will be mapped differently onto the event registers. This is shown in Figure 3.3 using

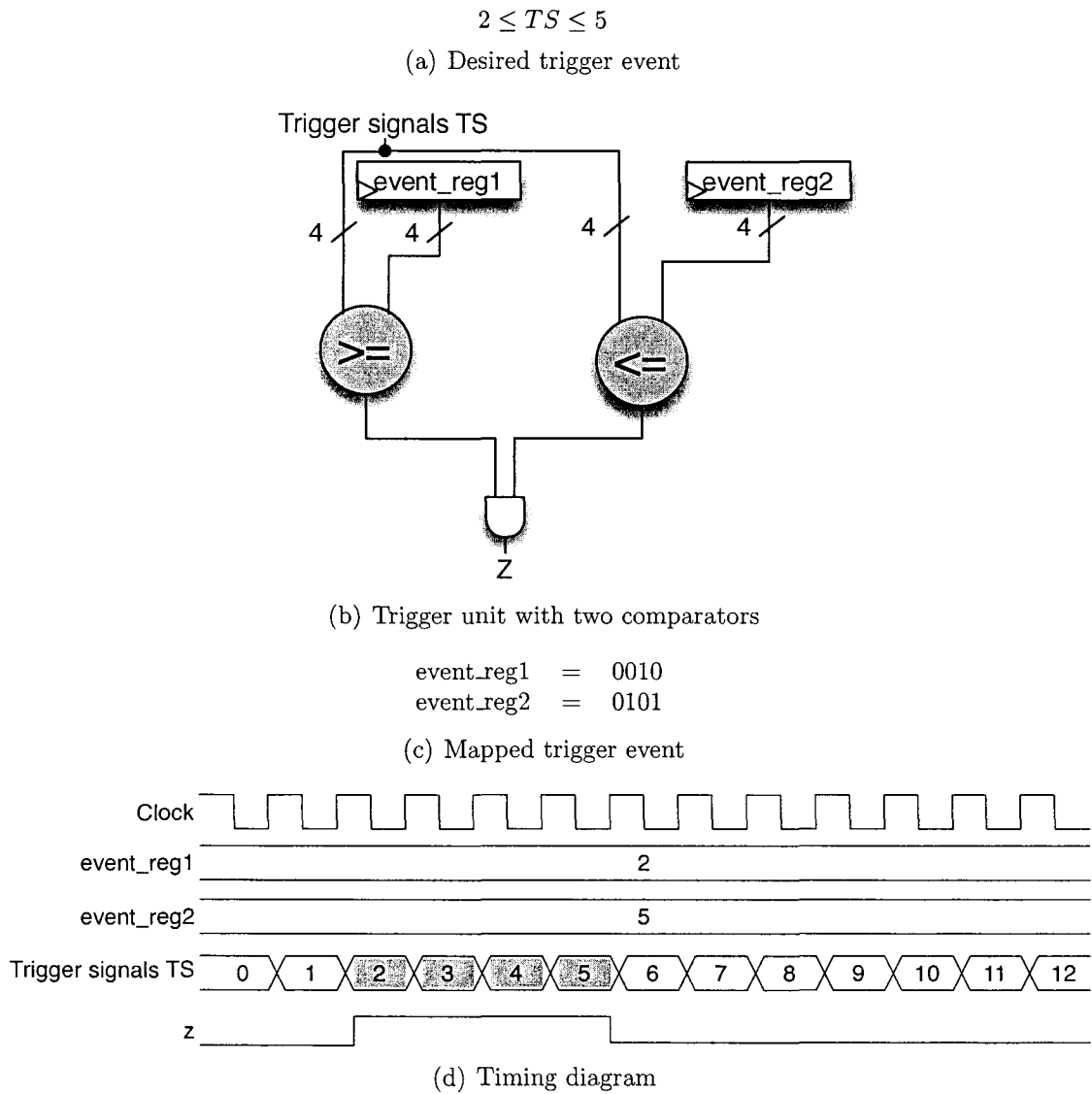


Figure 3.2: Example of trigger event mapping with comparators

a trigger unit with equality units. The trigger unit given in Figure 3.3(b) monitors TS using two equality units. For each equality unit, a mask register and an event register can be programmed at runtime to monitor one prime implicant on TS . For instance, if the value in the mask register is 0011 and the value in the event register is 0010, the *prime implicant* $XX10$, where X represents a don't care value, will be monitored by the equality unit. In this case, the equality unit can be used to monitor four *minterms* $\{0010, 0110, 1010, 1110\}$. Thus, the implementation of this equality unit may become more resource efficient than using the comparators when the set of values that need to be monitored are disjointed.

In this example for mapping trigger events onto equality units, we use the same trigger event as the one used in the previous example with comparators and is shown in Figure 3.3(a). When expressing this trigger event using logic conditions, the set of minterms that is covered by the event is: $\{0010, 0011, 0100, 0101\}$. We call this the *ON-set minterms*. This is because when any one of the minterms in the set is detected by the equality unit, the output z should be activated to initiate data acquisition. These minterms can be formulated into the following logic function with four input variables named $\{A, B, C, D\}$:

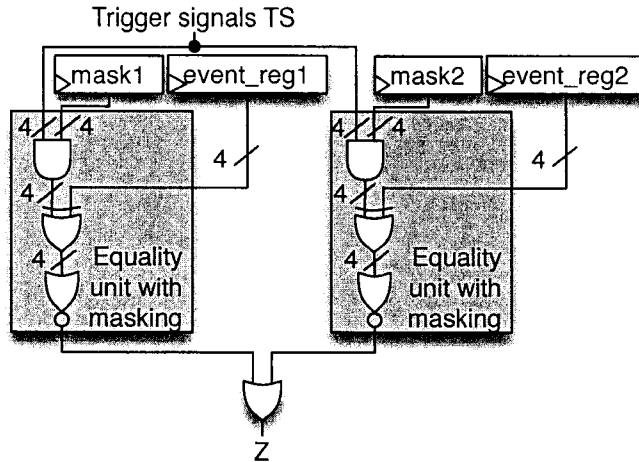
$$z = \bar{A}\bar{B}C\bar{D} + \bar{A}\bar{B}CD + \bar{A}B\bar{C}\bar{D} + \bar{A}B\bar{C}D$$

This logic function can then be minimized using known two-level logic synthesis techniques [79]. For illustrative purpose, Figure 3.4 uses the Karnaugh map (K-map) for performing the minimization. By minimizing the number of prime implicants in the logic function, the number of equality units required to describe the logic function can also be reduced. In this case, the resulting *ON-set prime implicants* are $\{001X, 010X\}$, which gives the minimized logic function

$$z = \bar{A}\bar{B}C + \bar{A}B\bar{C}$$

$$2 \leq TS \leq 5$$

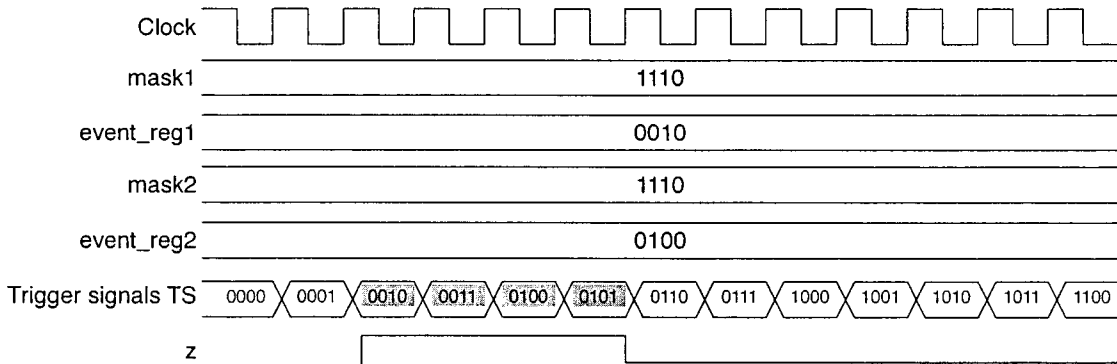
(a) Desired trigger event



(b) Trigger unit with two equality units with masking

mask1 = 1110 event_reg1 = 0010
 mask2 = 1110 event_reg2 = 0100

(c) Mapped trigger event



(d) Timing diagram

Figure 3.3: Example of trigger event mapping with equality units

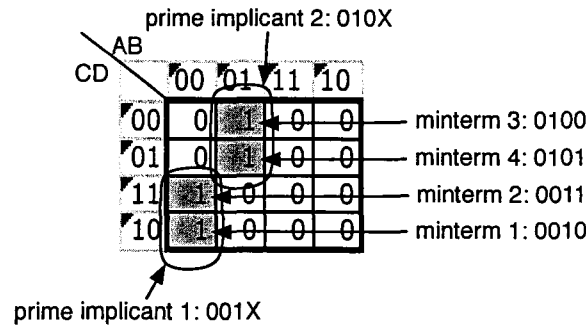


Figure 3.4: Example on minimization of logic function using K-map

These two prime implicants can then be monitored by the two equality units using the mask registers and event registers with values shown in Figure 3.3(c). And the trigger unit should behave according to the timing diagram given in Figure 3.3(d).

For more complex trigger events, more equality units may be required in the trigger unit. Consider the following trigger event: $2 \leq TS \leq 5$ or $TS = 10$, where the values on the 4-bit TS signal are given in the unsigned decimal format. For this event, three equality units will be required to detect the three prime implicants $001X$, $010X$, and 1010 in order to cover all the minterms of TS in the ON-set.

When performing post-silicon validation in ASICs, one will not be able to redesign the trigger unit to support any custom trigger events. If only the trigger unit shown in Figure 3.3(b) is given, the question is how to setup the mask and event registers in the two equality units to detect the three prime implicants $001X$, $010X$, and 1010 on TS ? There are two ways to tackle this problem and we define them as *under-triggering* and *over-triggering*.

3.1.3 The concept of under-triggering and over-triggering

Under-triggering

For *under-triggering*, multiple debug experiments are required. Using the trigger unit from the previous example, the timing diagram of the two debug experiments for the

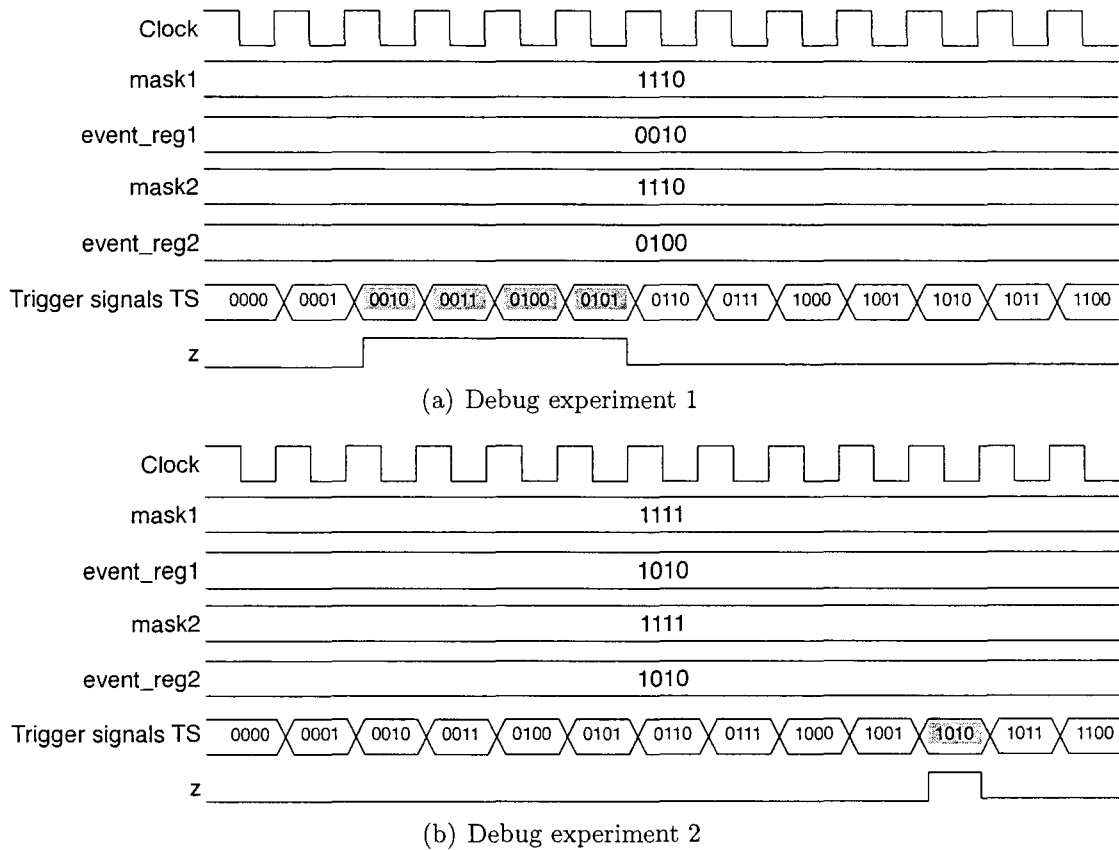


Figure 3.5: Timing diagram for under-triggering

trigger event $2 \leq TS \leq 5$ or $TS = 10$ is shown in Figure 3.5. First, the mask and event registers are programmed in order to cover the first part of the desired trigger event ($2 \leq TS \leq 5$) as given in Figure 3.5(a). In the next debug experiment, the registers are re-programmed to cover the remaining conditions as illustrated in Figure 3.5(b). Using under-triggering, all the debug data for the complex trigger event can be obtained in multiple iterations of the same debug experiment by reconfiguring the trigger unit. A key requirement for *under-triggering* to work is to have repeatable debug experiments, which is not always the case. For example, asynchronous peripherals may cause different execution traces in different debug experiments [8].

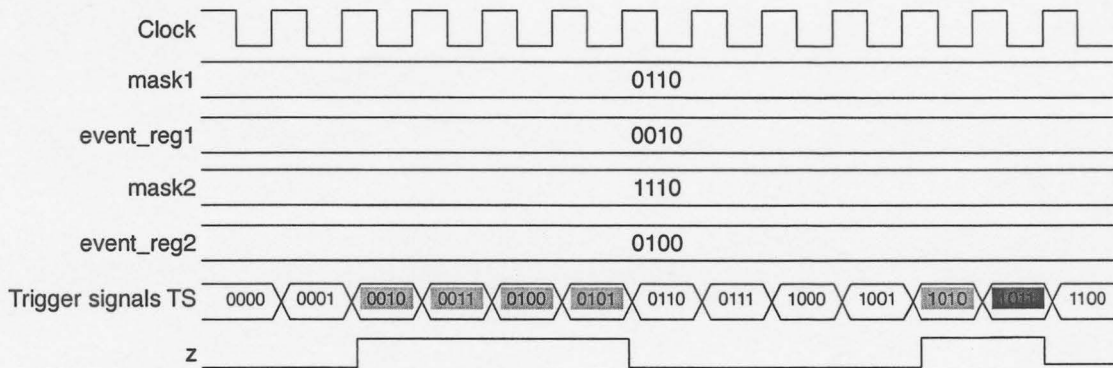


Figure 3.6: Timing diagram for over-triggering

Over-triggering

Alternatively, one can setup the mask and event registers with the following values to *over-trigger* the design:

$$\begin{aligned} \text{mask1} &= 0110 & \text{event_reg1} &= 0010 \\ \text{mask2} &= 1110 & \text{event_reg2} &= 0100 \end{aligned}$$

Using this configuration, the prime implicants $X01X$ and $010X$ are monitored by the trigger unit. In this case, the trigger will happen not only for $TS = \{0010, 0011, 0100, 0101, 1010\}$ (the ON-set minterms), but also for the false triggers (e.g., $TS = 1011$, which belongs to the OFF-set) as illustrated in the timing diagram from Figure 3.6. Hence, when a false trigger occurs, the data which has been acquired in the trace buffer will be of no interest. Therefore, *we propose a solution that enables over-triggering while reducing the impact of false triggers*. The key to our approach is to identify the false triggers in *real-time* without stopping the debug experiment. When false triggers are detected (using resource-efficient equality units), trace buffer space wasted by false triggers can be reclaimed for usage in the same debug experiment.

3.2 New architecture of a resource-efficient programmable trigger unit

Before we introduce the proposed architecture, we re-iterate the three types of trigger events that are commonly used during post-silicon validation as discussed earlier in Section 2.2.4 of the thesis.

Level sensitive trigger events - The example shown in Figure 3.3 is a typical scenario where level sensitive trigger events are involved. When the trigger signals reach a specified value in any clock cycle, the trigger unit will notify the debug control unit to initiate data acquisition. In this case, event detection happens only in one time frame.

Edge sensitive trigger events - It is also common to initiate data acquisition when a transition on the trigger signals is detected. For example, it may be desirable to acquire data when one of the trigger signals have a rising or falling edge. In this case, the event detector has to be capable of monitoring the trigger signals in two consecutive time frames in order to identify the signal transitions.

Trigger event sequencing - When sequencing is used, data acquisition starts when a sequence of trigger events occurs in a predefined order. For example, when debugging a design that contains a control bus, only when the sequence of events that define the bus protocol has occurred debug data needs to be acquired. To support sequencing, the trigger unit has to monitor the trigger signals in multiple time frames as defined by the sequence length.

Figure 3.7 shows the trigger unit that supports the detection of level sensitive trigger events with real-time trigger analysis. By programming the mask and event registers in the k equality units, k prime implicants can be monitored on the n -bit trigger signals *TS simultaneously*. Whenever the output of any equality units becomes 1, the FSM will be notified and debug data will be acquired in a segment of the trace buffer indicated by the write address on one of the two ports of the trace buffer. To detect complex trigger events that may need more than k equality units, the mask and event registers can be programmed to also cover some minterms from the OFF-set (in addition to all the ON-set values) to facilitate over-triggering.

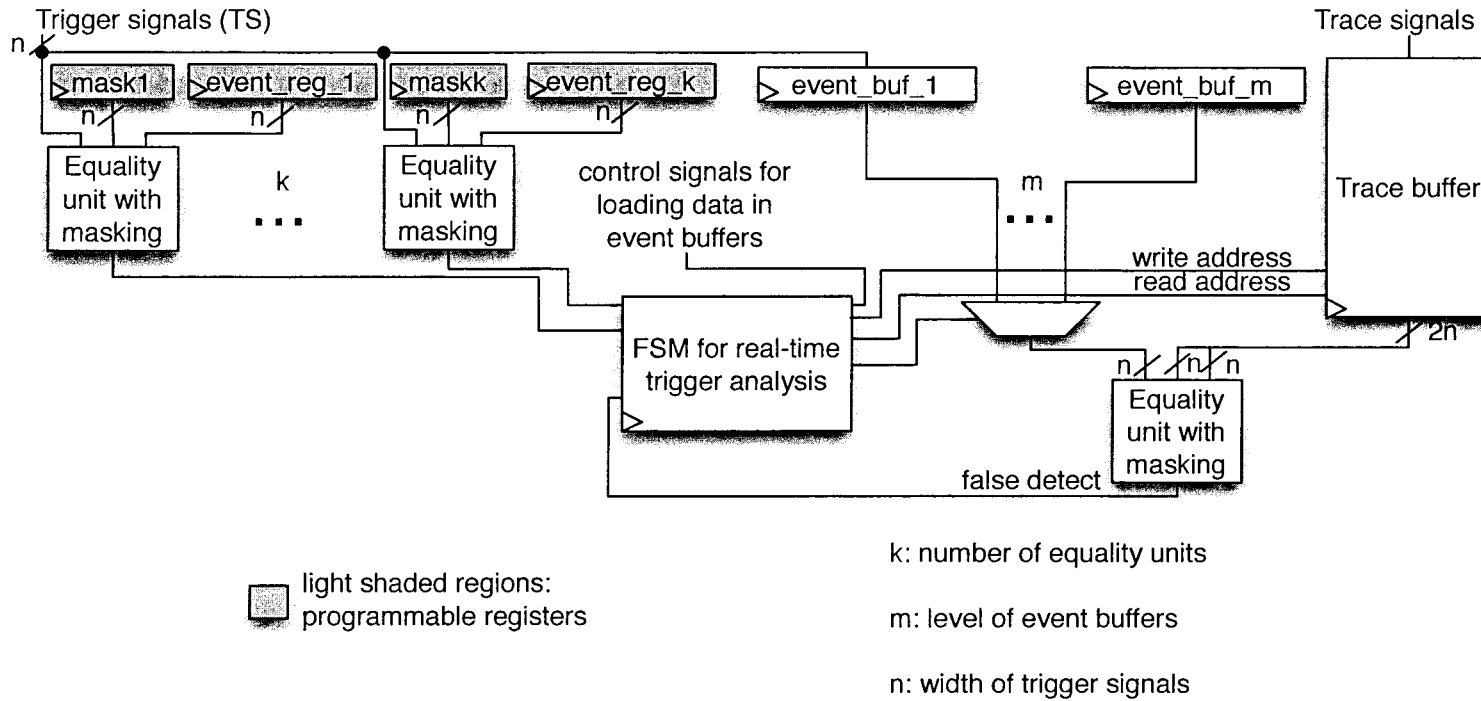
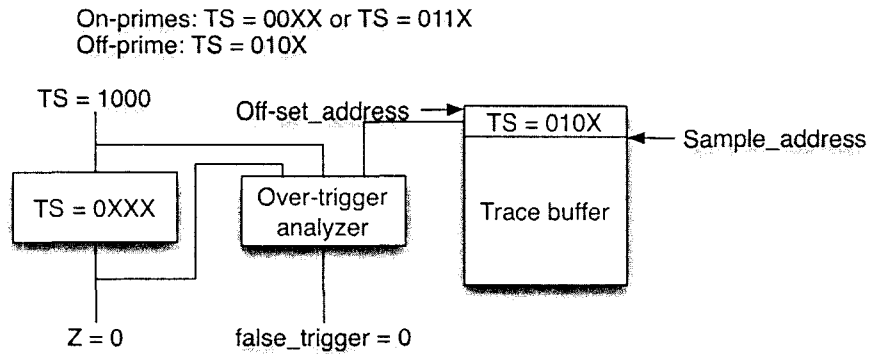


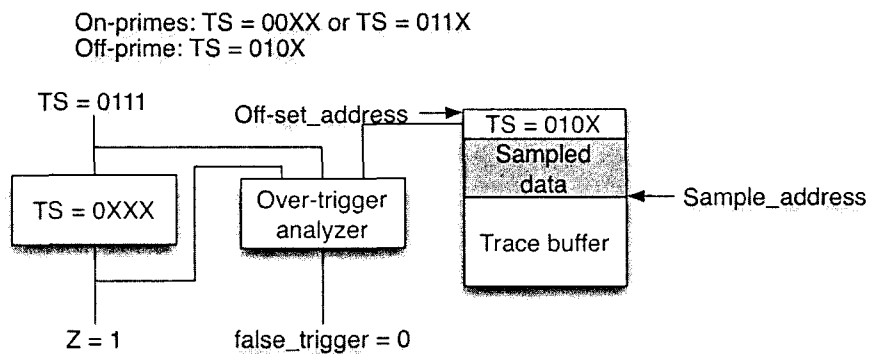
Figure 3.7: New architecture of a resource-efficient programmable trigger unit for level sensitive trigger events

Since over-triggering the debug module during post-silicon validation may risk filling the trace buffer only with data associated to the OFF-set minterms (as discussed in the previous section), *real-time trigger analysis* is performed in the proposed architecture. Whenever false triggering is identified, data acquisition is stopped and the segment of trace buffer used to store the corresponding debug data will be marked as invalid and thus, be reused for future trigger events. This is illustrated by the examples given in Figure 3.8. In these examples, the equality unit is setup to monitor the prime implicant $\{0XXX\}$, which covers the ON-primes $\{00XX, 011X\}$ as well as the OFF-prime $\{010X\}$. When the value on TS is 1000 as shown in Figure 3.8(a), it does not match with the programmed prime implicant, and thus, will not activate the output of the equality unit. In Figure 3.8(b), the value on TS becomes 0111, which belongs to one of the ON-primes. As a result, the output z is activated. When the value on TS comes up to be 0101, the output z is also activated, since it matches with the programmed prime implicant on the equality unit. However, the output of the trigger analyzer will also be activated since the minterm 0101 belongs to the OFF-prime $\{010X\}$, which is retrieved from a designated portion of the trace buffer. This concludes it is a false trigger, and thus the sample address of the trace buffer is reverted so that the storage space can be reused later. The real-time trigger analysis is done by employing *only one additional equality unit*, as shown in the lower right corner of Figure 3.7.

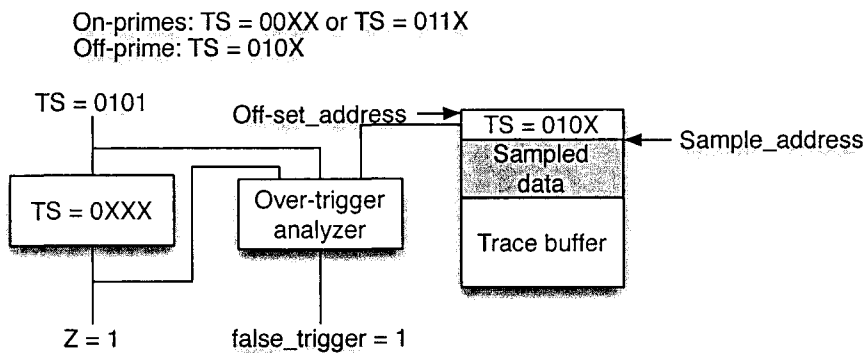
Whenever data acquisition starts, the value on TS corresponding to the particular trigger event will be buffered (using the *event_buf_XX* from Figure 3.7). Concurrently with data acquisition, the FSM will retrieve the prime implicants of the OFF-set (called OFF primes) one at a time from the trace buffer. The buffered event is checked against each OFF prime *sequentially* by the additional equality unit. False triggering is concluded if the buffered TS matches any of the OFF primes. At this point, data acquisition may be interrupted if there are no other triggers in the event buffers. On the other hand, if the buffered TS does not match any of the OFF primes, a valid trigger will be concluded and data acquisition will not be interrupted. However, any further triggering will be ignored until a segment of trace buffer has been filled in order to preserve the sampled data of the related trigger.



(a) No triggering



(b) Valid triggering



(c) False triggering

Figure 3.8: Example of trigger event analysis

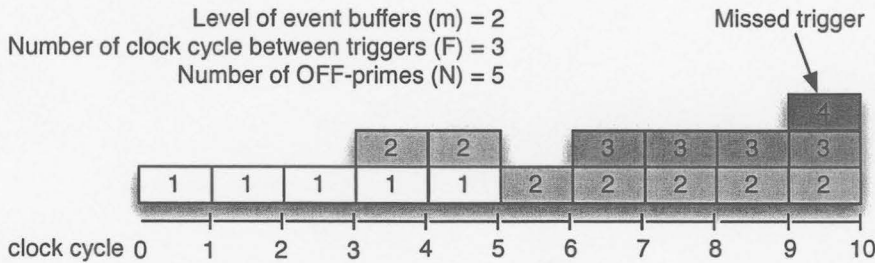


Figure 3.9: Example of trigger event being missed

As will be discussed in the next section, the OFF primes can be computed off-line when the trigger event is specified. Then they are uploaded in an unused region in the trace buffer at the same time when the trigger unit is programmed at runtime. The trigger analysis for one trigger event is done in multiple clock cycles, because the OFF primes are checked sequentially (they are extracted from the trace buffer one at a time). Since valid trigger events can occur in consecutive clock cycles as they are checked simultaneously using k equality units, m levels of event buffers may be needed for storing individual trigger events for trigger analysis. However, no matter how many levels of event buffers are employed, so long as the number of OFF primes that need to be checked ($N_{OFF-primes}$) is larger than the number of clock cycles between consecutive trigger events ($F_{trigger}$), eventually the event buffers will be filled and further trigger events will be missed. This is illustrated in Figure 3.9. In this example, two levels of event buffers are employed ($m = 2$). The number of OFF primes that need to be checked is five ($N_{OFF-primes} = 5$). But the number of clock cycles between consecutive trigger events is three ($F_{trigger} = 3$). This means that a new trigger will come at clock cycles 0, 3, 6 and 9. Each of these new triggers has to be buffered for five clock cycles to perform false trigger analysis on five OFF primes. This means only one buffer will be emptied every five clock cycles (i.e., clock cycles 5 and 10 in Figure 3.9). As a result, when the fourth new trigger arrives at clock cycle 9, the two event buffers will be full and this trigger will be missed.

There are three parameters one can manipulate to prevent trigger events to be missed using the proposed architecture. Since one cannot control how often do trigger

events occur ($F_trigger$) during post-silicon validation, many levels of event buffers (m) can be used. However, this will increase the area of the trigger unit. As a result, in the next section, we introduce an efficient algorithm for mapping trigger events with a large number of ON primes to the limited number of equality units with only a small number of OFF primes ($N_{OFF-primes}$). This ensures trigger analysis can be done in a low number of clock cycles.

In order to support edge sensitive trigger events, two equality units can be used to detect one type of transition on the trigger signals. For example, to detect a rising edge on the first bit of TS , the following values can be programmed into the trigger unit:

$$\begin{aligned} \text{mask1} &= 1000 & \text{event_reg1} &= 0000 \\ \text{mask2} &= 1000 & \text{event_reg2} &= 1000 \end{aligned}$$

For this purpose, the FSM should be modified to monitor the two equality units in two consecutive clock cycles for detecting edge sensitive trigger events. However, the hardware for trigger analysis will not be modified. This is because the sequence of two events for edge detection are correlated to each other (i.e., the occurrence of the first event is guaranteed when an edge is detected upon the arrival of the second event). As a result, false triggering can be concluded only by checking the value of TS of the second event with the OFF primes.

For trigger event sequencing, the equality units can be divided into g groups to detect g distinct events in an event sequence as shown in Figure 3.10. In this case, each event group will require m levels of event buffers for storing trigger events associated with the individual group. Also, a counter with g programmable registers is added to indicate when each event is expired after it is detected. This is useful for event sequences, such as a bus protocol, where each event in the sequence should come within a specific interval. Finally, the FSM will be modified to monitor different groups of equality units in different timeframes, as well as for retrieving the appropriate OFF primes to identify the false event sequences.

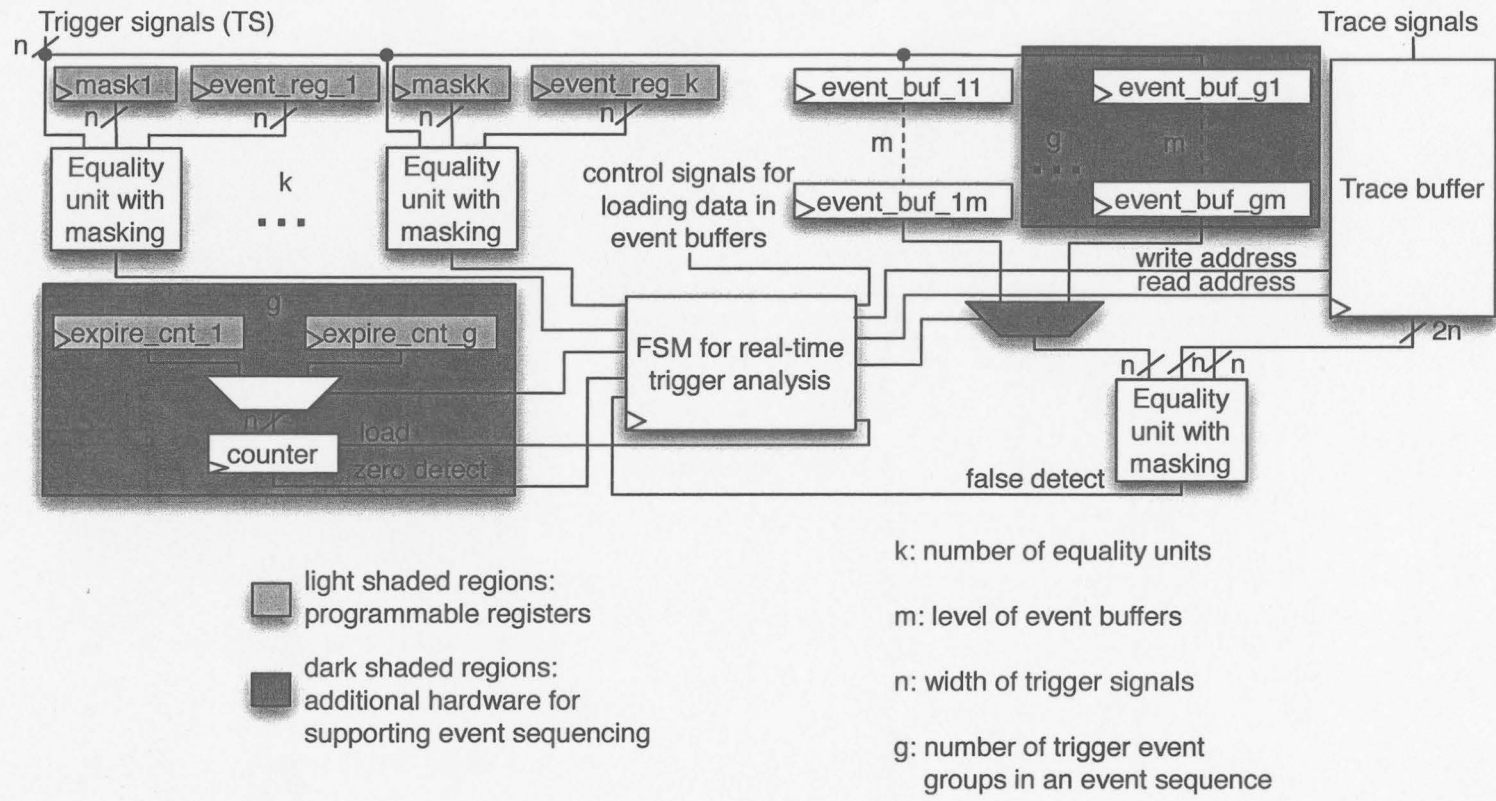


Figure 3.10: New architecture of a resource-efficient programmable trigger unit with support for event sequencing

3.3 Algorithm for trigger event mapping

As mentioned previously, it is important to include only a small number of OFF minterms when mapping the ON primes onto the equality units during over-triggering. This prevents all the event buffers to be filled and valid trigger events to be missed. As a result, an efficient algorithm for this purpose is proposed in Algorithm 3.1.

Algorithm 3.1 Trigger event mapping algorithm

```

1: Perform logic minimization on ON implicants to get ON primes
2: LargePrimes = GetLargerPrimes(ON primes)
3: OldPrimes = ON primes
4: while (NumOldPrimes > NumEqUnit) do
5:   CurPrime = GetBestGain(LargePrimes)
6:   while (CurPrime = NULL) do
7:     LargePrimes = GetLargerPrimes(LargePrimes)
8:     CurPrime = GetBestGain(LargePrimes)
9:   end while
10:  Put CurPrime into OldPrimes
11:  Perform logic minimization on OldPrimes to get NewPrimes
12:  if (NumNewPrimes >= NumOldPrimes) then
13:    Backtrack over line 5 to new choice of CurPrime
14:  else
15:    OldPrimes = NewPrimes
16:    UpdateOffMintermList(CurPrime)
17:  end if
18: end while
19: Perform logic minimization on OFF minterms to get OFF primes
20: return NewPrimes and OFF primes

```

The inputs to the algorithm include the width of trigger signals (*TSWidth*), the number of equality units in the trigger unit (*NumEqUnit*), and the minterms in the ON set (*ON minterms*) from a trigger event. When the algorithm finishes, it returns the set of primes (*NewPrimes*) which gives information about how the mask and event registers should be programmed for the particular trigger event, and the set of OFF primes (*OFF primes*), which should be stored in the trace buffer for trigger analysis. The algorithm starts by first performing logic minimization on the ON minterms that described the trigger event, in order to obtain the ON primes (line

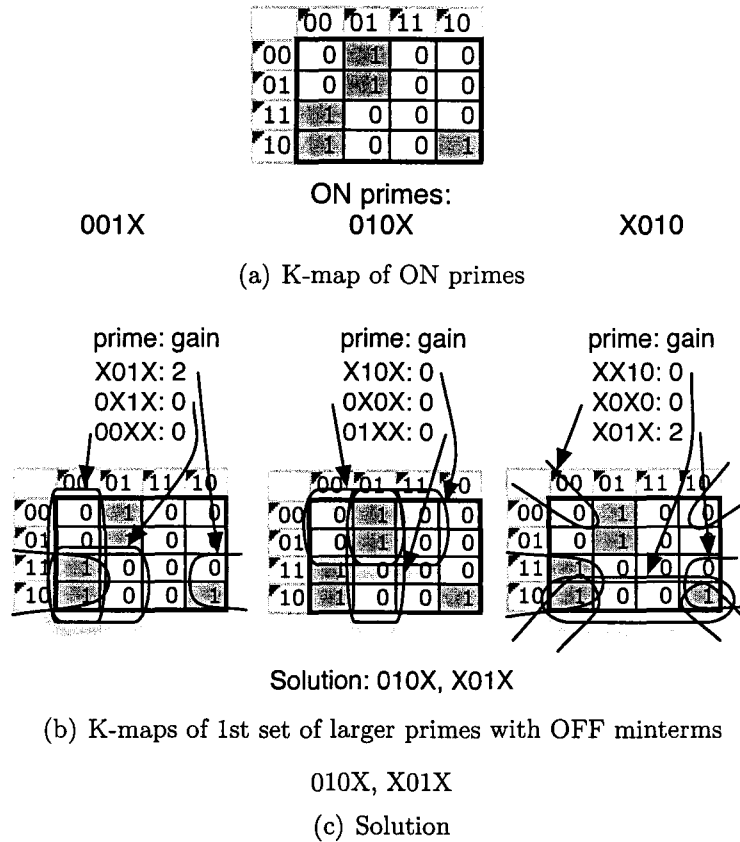


Figure 3.11: Example of mapping three ON primes to two equality units

- 1). Using these ON primes, larger primes can be obtained by gradually increasing the number of don't cares (X) in each ON prime (function *GetLargerPrimes* in line 2).
- 2). This process of enlarging primes can be explained in Figure 3.11, using K-maps for illustrative purposes.

In Figure 3.11(a), the K-map of the three ON primes are shown for the desired trigger event $2 \leq TS \leq 5$ or $TS = 10$. For each of these ON primes, three new primes that cover some ON minterms and their adjacent OFF minterms can be obtained by injecting one don't care (X) on each bit of one ON prime, as shown in Figure 3.11(b). A gain is then calculated by subtracting the number of OFF minterms from the number of ON minterms covered by each of the enlarged primes. Note,

when creating the enlarged primes, the same prime may result from different ON primes (e.g., the prime $X01X$ in Figure 3.11(b)). These duplicated primes should be eliminated to reduce runtime. When mapping the three ON primes in Figure 3.11(a) to two equality units, the final primes shown in Figure 3.11(c) should be used to cover the ON minterms $\{0010, 0011, 0100, 0101, 1010\}$ and one OFF minterm 1011.

When the number of ON primes ($NumOldPrimes$) exceeds $NumEqUnit$, Algorithm 3.1 iteratively (line 4) tries to replace one original prime with one of its associated enlarged primes (line 10). To ensure that only a small number of OFF minterms is included, Algorithm 3.1 will greedily select an enlarged prime that has the highest gain as a candidate (function *GetBestGain* in line 5). It then performs logic minimization with the chosen prime and the original primes to see if they can be reduced (line 11). If the number of new primes ($NumNewPrimes$) is not reduced, the choice will be reverted (line 13) and Algorithm 3.1 will repeat and try another enlarged prime. Otherwise, the chosen enlarged prime will be included and the covered OFF minterms will be identified (function *UpdateOffMintermList* in line 16). When Algorithm 3.1 has tried all the enlarged primes in the candidate list, it will further enlarge the existing primes by injecting more don't cares to obtain new primes that cover more OFF minterms (lines 7-8). Finally, when all the new primes can be fit into the available equality units, logic minimization will be performed on the newly included OFF minterms to obtain the OFF primes (line 19).

One interesting point to note is that in [57], an algorithmic solution was introduced to identify the ON minterms on the available trigger signals from a trigger event that is described by signals that are not connected to the trigger unit. Afterwards, they showed how the problem of mapping this set of ON minterms to the available equality units by adding OFF minterms can be solved by quantified boolean formulation (QBF). However, their QBF solution does not scale well when the number of available trigger signals increases. By replacing their QBF solution with our trigger event mapping algorithm, which has a faster runtime, one can set up the equality units to trigger on events from signals that are not connected to the trigger unit.

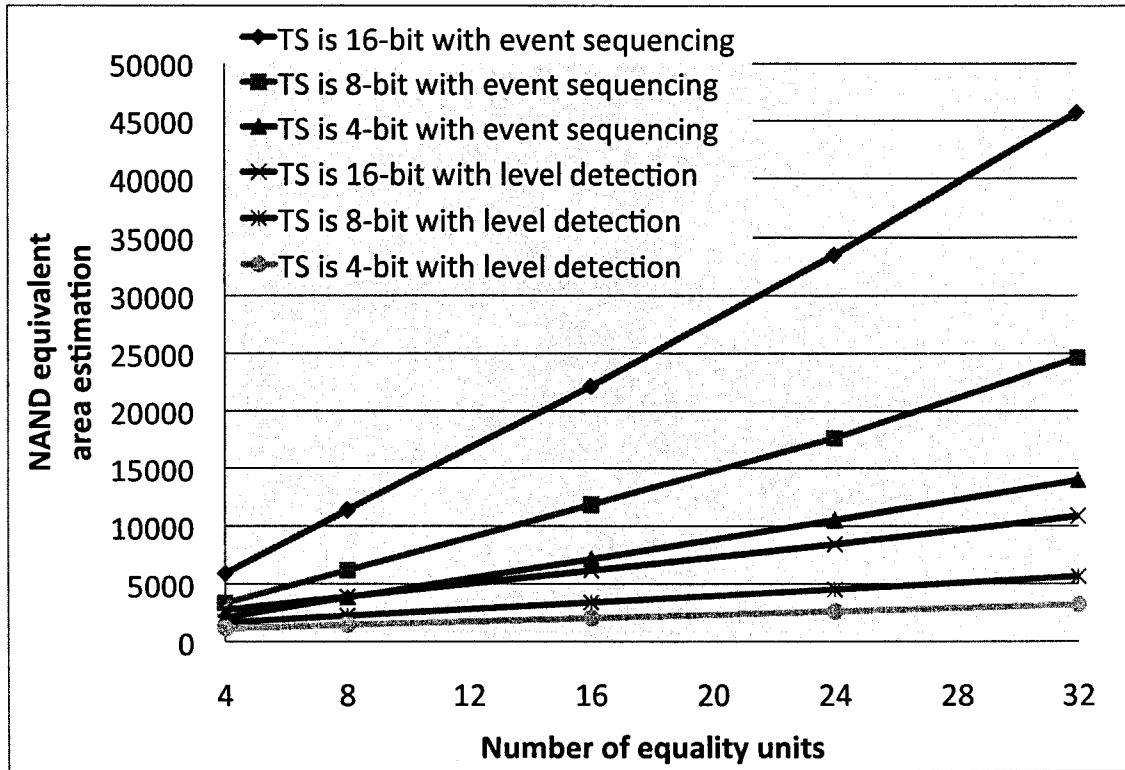


Figure 3.12: Area investment analysis when varying the number of equality units

3.4 Experimental results

In this section, we discuss the area for the control features in the new architecture of the resource-efficient trigger unit in SubSection 3.4.1. The results on analyzing the proposed trigger mapping algorithm is given in SubSection 3.4.2. Finally, SubSection 3.4.3 shows the miss trigger analysis when employing the proposed solution.

3.4.1 Area analysis on the proposed architecture

Figure 3.12 shows the area of the trigger units with 4, 8, 16, 24 and 32 equality units with 4, 8 and 16 trigger signals, when using 7 levels of event buffers. For trigger units with event sequencing, one equality unit is assigned to one group of trigger events in

the sequence (i.e. $g = k$ from Figure 3.10). In our experiments, a third party tool [108] is used to synthesize our designs for area estimation.

It can be seen that when the number of equality units increases, the area of the trigger unit rises accordingly. However, the growth of area for the trigger units that support event sequencing is worse than trigger units that support only level sensitive trigger events. This is because when supporting event sequencing in our experiments, increasing the number of equality units means the number of trigger event groups in an event sequence also grows (i.e., since $g = k$ as mentioned). Thus, when more equality units are used, more event buffers will also be employed. Also, the area of the trigger units that support only level sensitive trigger events are smaller than the trigger units with event sequencing for the same number of equality units. This is due to the addition of the expiration counter and event buffers for each trigger event group in the sequence, and the increased complexity of the FSM. When the number of trigger signals rises, the size of the trigger units grows accordingly. This is because increasing the number of trigger signals requires the width of all registers and equality units to be enlarged.

When the level of event buffers increases for trigger units with 32 equality units, as shown in Figure 3.13, the area of the trigger units that support level sensitive trigger events is only affected slightly. This is because varying the level of event buffers for this type of trigger unit only requires a few buffers to be added together with a small modification to the FSM for controlling these buffers. However, for trigger units with event sequencing, the effect of the expansion in level of event buffers is multiplied with the number of trigger event groups in the sequence.

3.4.2 Analysis on the proposed algorithm

Table 3.1 gives the results for the proposed trigger event mapping algorithm when mapping trigger events with different numbers of ON primes with varying number of equality units when the number of trigger signals is eight. In our implementation of Algorithm 3.1, the logic minimization tool *espresso* [79] is employed, and the program is written in ANSI C and executed on a computer with dual-Xeon processors at 2.4

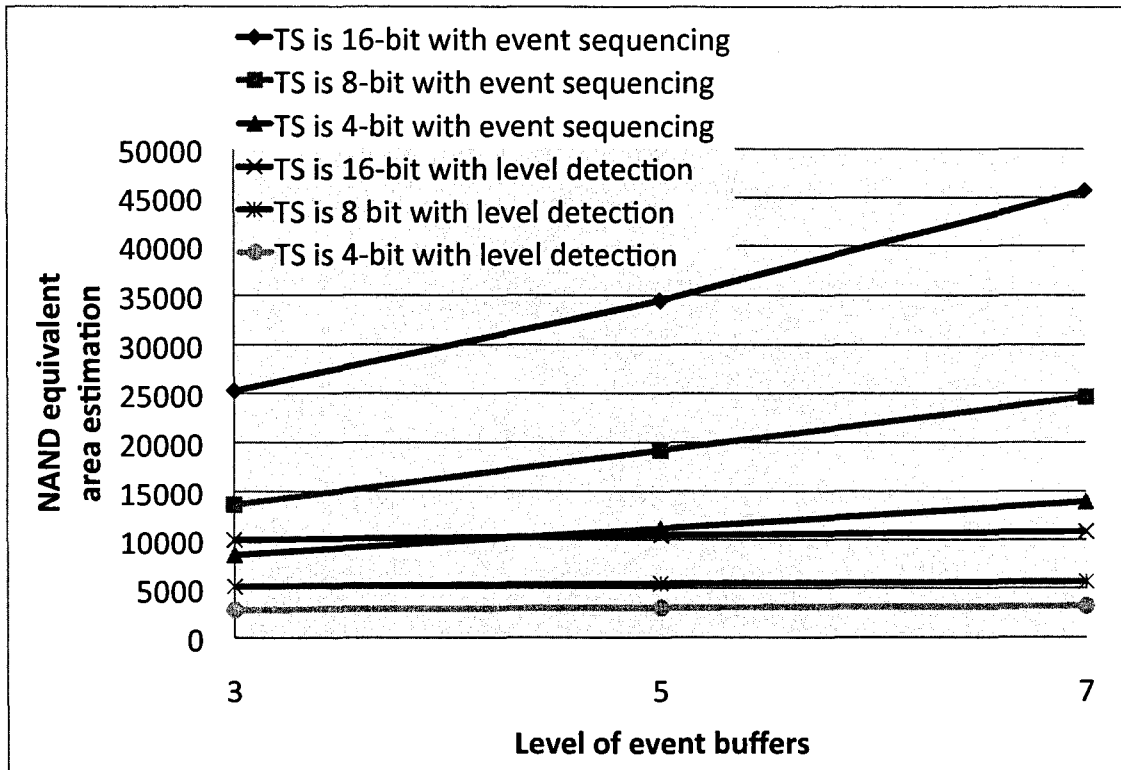


Figure 3.13: Area investment analysis when varying the level of event buffers

GHz with 1GB of RAM. In the column labeled *Area (%)*, the area of the trigger unit that supports only level sensitive trigger events with four equality units and seven levels of event buffers is used as the basis for showing the tradeoff between increasing the number of equality units and the growth in number of OFF primes for trigger analysis. Note, when the number of ON primes is smaller than or equal to the number of available equality units, no OFF primes will be required.

From Table 3.1, it can be seen that the number of OFF minterms grows as the ratio between the ON primes and equality units increases. However, when the number of OFF minterms decreases, the number of OFF primes may not be reduced accordingly (e.g., mapping trigger events with 28 and 33 ON primes onto 4 equality units). This is because even if less OFF minterms are used, if they cannot be grouped together,

Table 3.1: Results for analyzing the trigger event mapping algorithm

Num of EQ unit	Area %	Num of ON primes	Num of OFF minterms	Num of OFF primes	Runtime (sec)
4	100	8	13	5	0.02
		13	58	10	0.05
		18	54	13	0.21
		23	85	20	0.22
		28	96	24	0.26
		33	79	26	0.26
8	134	13	6	4	0.03
		18	14	7	0.03
		23	52	17	0.20
		28	50	19	0.22
		33	55	23	0.22
16	202	18	2	2	0.02
		23	5	4	0.02
		28	13	8	0.05
		33	25	15	0.06
24	269	28	2	2	0.02
		33	7	7	0.06
32	338	33	1	1	0.01

the number of OFF primes can increase. The main point here is that with a relatively low set of false triggers (related to the number of OFF primes), which are checked in real-time on-chip, we can afford implementing trigger units with less comparators (and hence less area). The key to success, however, is a fast heuristic algorithm that can determine the OFF primes as soon as the trigger conditions are specified.

3.4.3 Miss trigger analysis for the proposed solution

When employing the proposed resource-efficient programmable trigger units, false triggers are analyzed in real-time in order to avoid storage space in the trace buffer being wasted. However, since false trigger analysis is done sequentially, when there are more than one OFF prime needing to be checked, multiple clock cycles will be

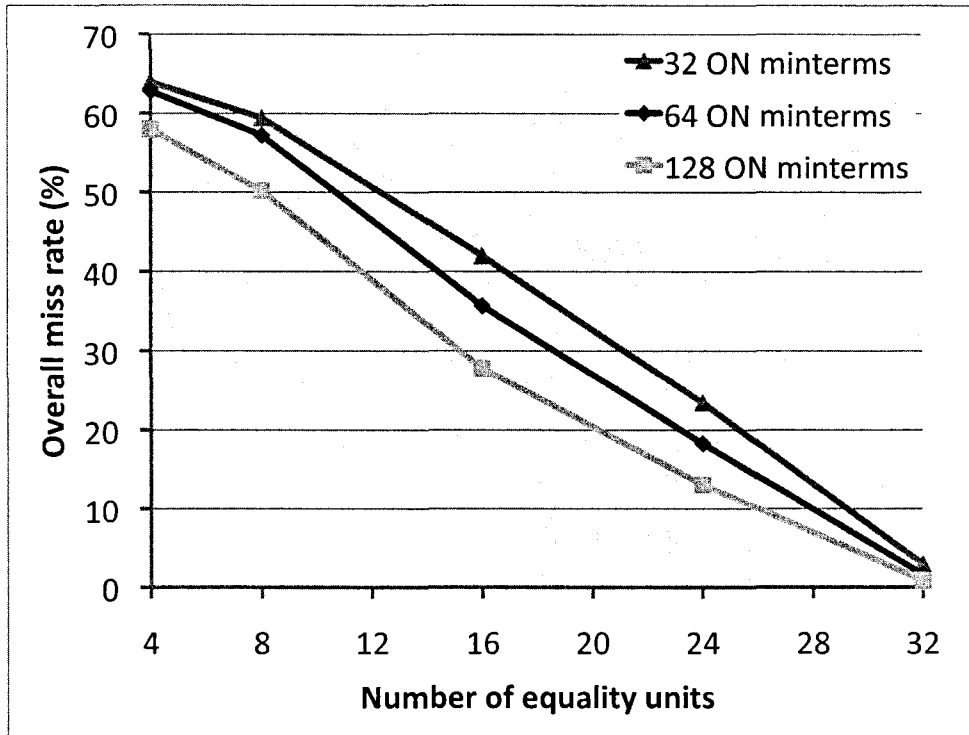


Figure 3.14: Miss trigger analysis when varying the number of equality units with frequent triggering

required to determine the validity of each trigger event. As a result, if a new trigger event comes when the trigger analysis engine is busy, this new event will have to be stored in the event buffer. However, when the event buffers are full, this new event will have to be discarded. In order to determine how often trigger events will be missed, random simulation experiments are performed and their results are analyzed.

Figures 3.14 and 3.15 show the results on miss trigger analysis when the number of equality units increases. In these experiments, three levels of event buffers are used. It should be noted that the number of trigger events that will be missed during post-silicon validation depends not only on the configuration of the proposed architecture, but also on the input stimuli and the desired trigger conditions, which invoke various trigger events at random point of time during a debug experiment. To account for

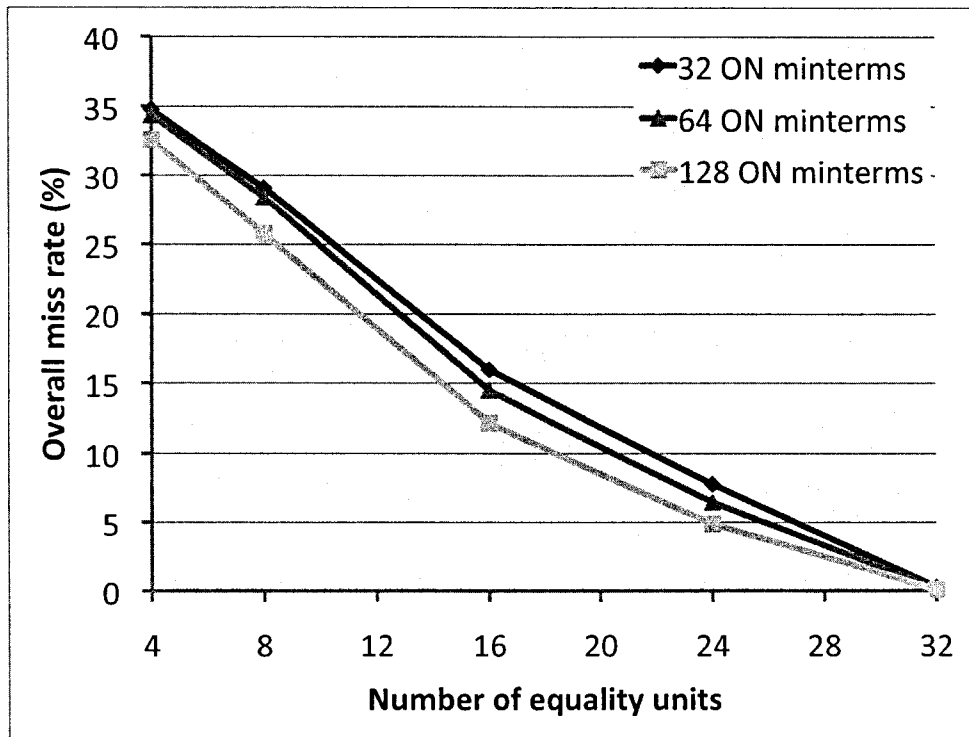


Figure 3.15: Miss trigger analysis when varying the number of equality units with non-frequent triggering

this randomness, in these experiments, three different sets of trigger conditions with 32, 64 and 128 ON minterms are used. There are five different trigger conditions with varying number of OFF minterms in each of these sets. Also, for each condition, 10 sequences of trigger events are randomly generated over the period of 10 million clock cycles. The number of trigger events that are missed due to overflow of event buffers is counted for each of these sequences by simulating the application of the trigger sequence onto each configuration of the proposed architecture. This number is then divided by the total number of trigger events in the sequence to obtain the miss rate. The calculated miss rates for all the trigger sequences are then averaged to obtain the overall miss rates in the figures. For Figure 3.14, an average of four clock cycles between the occurrence of two consecutive trigger events is used to simulate frequent

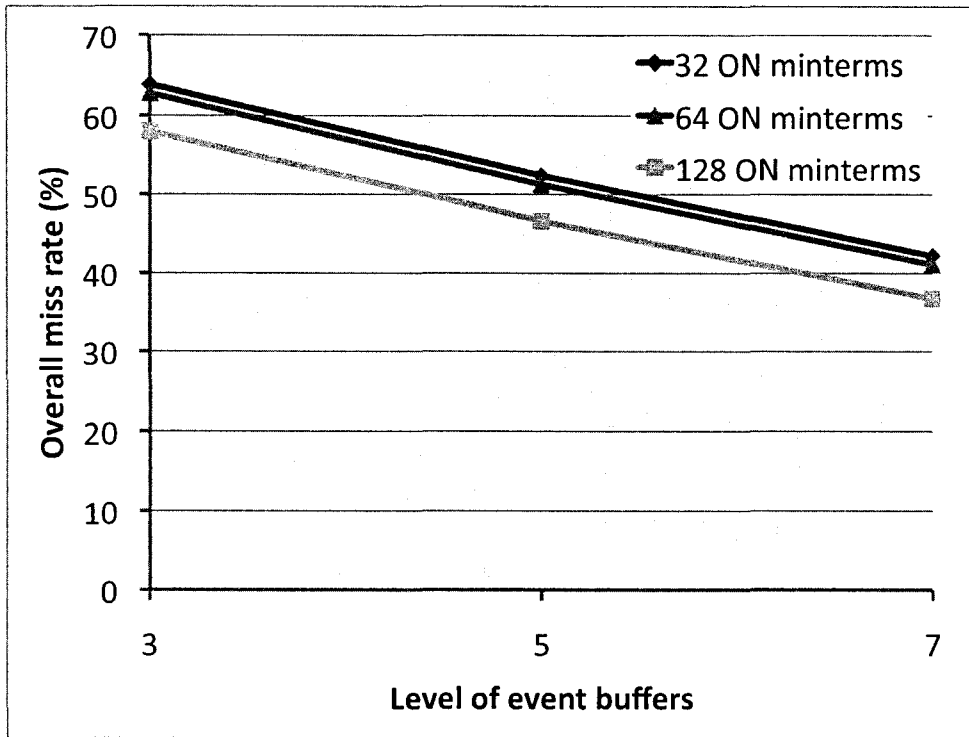


Figure 3.16: Miss trigger analysis when varying the level of event buffers with frequent triggering

appearances of trigger events. This generates trigger events more frequently than the experiments in Figure 3.15, which use an average of 12 clock cycles when the trigger sequences are randomly generated.

It can be seen from both figures that when the number of equality units increases, the overall miss rate decreases. This is because when more equality units are available, less number of OFF minterms will be used for describing the trigger events. This results in less number of false triggers, and thus reducing the overall number of triggers that will occur during a debug experiment. It is also noted that the overall miss rates in both figures become close to 0 when employing 32 equality units. However, they become 0 in our experiments because the chosen trigger conditions tend to have a very small number of OFF minterms. Thus, this does not mean that employing 32

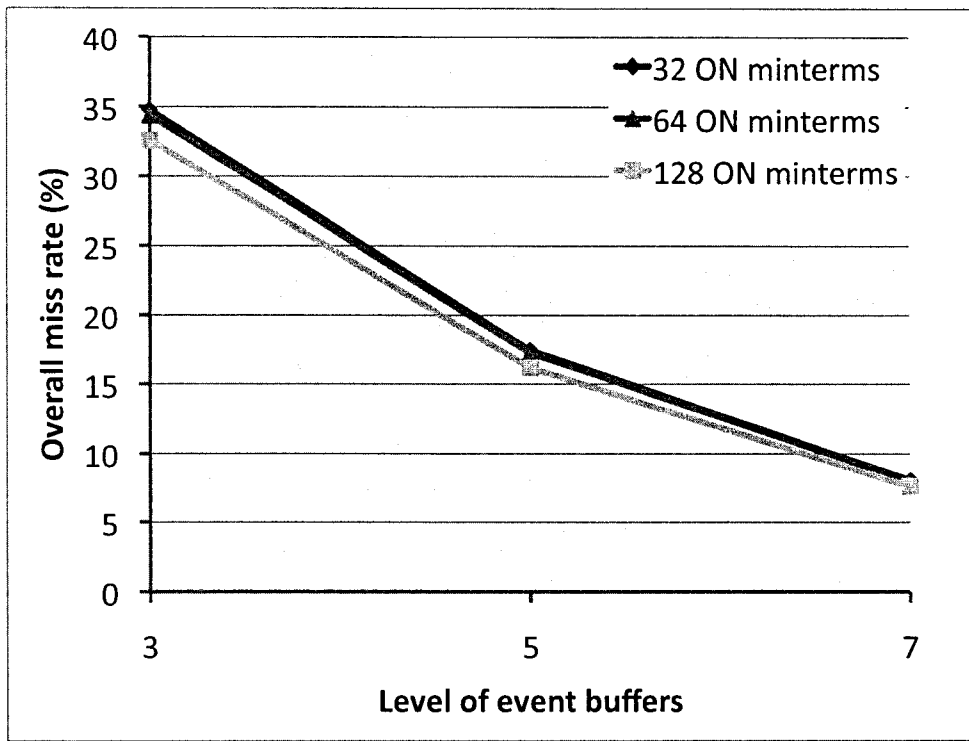


Figure 3.17: Miss trigger analysis when varying the level of event buffers with non-frequent triggering

equality units will always be able to avoid trigger events being missed for any trigger condition, since there could be trigger conditions that require more equality units to keep the number of OFF minterms small.

Figures 3.16 and 3.17 give the results on miss trigger analysis when varying the level of event buffers. While Figure 3.16 contains the results for experiments with frequent trigger occurrences, the experiments in Figure 3.17 are performed with non-frequent trigger occurrences. The trigger sequences used in the experiments for these figures are the same as the ones described previously for Figures 3.14 and 3.15. However, these sequences are applied with different configurations of the architecture in terms of level of event buffers, while the number of equality units is set to four.

It can be seen that by increasing the level of event buffers, the overall miss rate decreases in both figures. This is because having more buffers will allow more trigger events to be buffered during trigger analysis. Thus, reducing the likelihood that a trigger event to be missed due to buffer overflow.

It is also noted from Figures 3.15 and 3.17 that the overall miss rate decreases more rapidly when compared to that of Figures 3.14 and 3.16. This is because when trigger events are not occurring frequently, depending on when each trigger event arrives, more clock cycles may be passed between consecutive trigger events. This may give the trigger analysis engine more time to empty the event buffers between consecutive trigger events. As a result, increasing the number of equality units or level of event buffers will produce a more effective result in reducing the number of missed trigger events when they do not occur frequently. Moreover, non-frequent triggering also produces less number of trigger events during the considered time period of 10 million clock cycles. This reduced number of total trigger events also decreases the load on the trigger analysis engine. Thus, as shown in the figures, the overall miss rates are always lower when the experiments are run with non-frequent trigger occurrences.

The results from Figures 3.14, 3.15, 3.16 and 3.17 have shown that the miss rate can be reduced by either increasing the number of equality units or the level of event buffers. However, as pointed out earlier in SubSection 3.4.1, employing more equality units incurs more area in terms of logic resources due to the added hardware and complexity of the FSM, when compared to using more event buffers, for which case the added area is lower. As a result, it may be desirable to utilize more event buffers even though they may not help reduce the miss rate as rigorously when compared to employing more equality units.

3.5 Summary

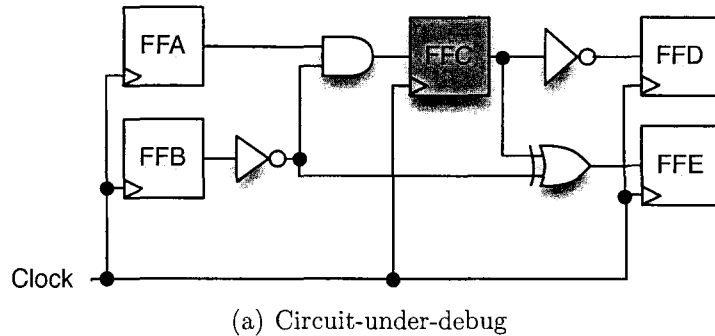
In this chapter of the thesis, we have investigated how to design programmable trigger units that enable real-time debug data acquisition during in-field/at-speed post-silicon validation. In our solution, we map the user-programmable trigger conditions onto a lower number of comparators than the number of prime implicants in the condition function. We mitigate false triggering by using an efficient approach for checking on-chip if the event that triggered data acquisition is valid.

Chapter 4

Algorithms for state restoration

The previous chapter has discussed how to improve the programmability of trigger units without an excessive cost in hardware. This will provide the user with the ability to better decide when to start data acquisition in the trace buffer. However, the amount of data that can be acquired is still limited by the trace buffer depth, which limits the number of samples to be stored, and its width, which limits the number of trace signals sampled in each clock cycle. In this chapter we investigate a post-processing step that can be applied to the acquired data in order to improve the observability before the user explores it in a simulation window.

To the best of the author's knowledge, the only solution available in the public domain with a similar goal is [44]. However, their algorithm restores data only in the combinational logic nodes of the circuit as discussed in Section 2.3.4. In this chapter of the thesis, we show an automated algorithm in Section 4.1 for restoring data in state elements (called *state restoration*) for enlarging the set of debug data during post-silicon validation. Moreover, in order for the state restoration algorithm to be applicable to large circuits, it is essential for the algorithm to be compute-efficient. As a result, a technique for enhancing the performance of the algorithm is presented in Section 4.2. The proposed algorithmic solution can help restore missing data on other signals as supported by the results shown in Section 4.3. Finally, a summary of the contributions is provided in Section 4.4.



Clock cycles

	0	1	2	3	4
FFA	1	1	X	X	X
FFB	0	0	X	X	X
FFC	0	1	1	0	X
FFD	X	1	0	0	1
FFE	X	1	0	X	X

(b) Restored data in sequential elements

Figure 4.1: Sample circuit for state restoration

4.1 Algorithmic solution for state restoration

It has been shown in SubSection 2.3.4 how the principal operations in Figure 2.15 can be used to reconstruct data on the signals in the combinational logic of a circuit. In Figure 4.1, an example is given to demonstrate how these principal operations can be applied to a circuit that has both combinational and sequential elements. As Figure 4.1(a) shows the simple circuit with five flip-flops (FFs), Figure 4.1(b) gives the data in the state elements after the restoration algorithm is applied. In this example, only *FFC* is sampled during clock cycles 0 – 3. It should be noted that the *Xs* in the table in Figure 4.1(b) refer to values that cannot be restored using only the available sampled data.

Before applying the state restoration algorithm shown in Algorithm 4.1, the circuit netlist is translated into a graph where the nodes represent logic gates, state elements,

primary inputs and outputs; the directed edges represent signal dependencies. For example, given a two-input logic gate, it will be translated into a single node, with two directed edges connecting from its parent nodes, which represent the two circuit elements that drive the logic gate. Likewise, a directed edge will be added from the newly created node to each of its child nodes, which represent the circuit elements that it drives.

With the translated circuit graph, the principal operations will be applied to each node repeatedly until no more data can be reconstructed for all signals from the given subset of data. This can be shown by applying Algorithm 4.1 to the circuit in Figure 4.1(a). Using backward justification (line 5), whenever a logic 1 is captured in FFC , the values of FFA and FFB can be evaluated as logic 1 and 0 respectively in the previous clock cycle. In addition, the inverted values of FFC can be forward propagated to be the values of FFD in the following clock cycles (line 11). For FFE , its values in the current clock cycles can only be reconstructed when the values of FFB and FFC are known in the previous clock cycles. This will be done by the next iteration of the loop (line 2) with the updated information on FFB . Note that whenever the *BackwardOperation* or the *ForwardOperation* is applied to a node, the operation will try to perform state restoration from that node for all the considered clock cycles. The reason for this is to lower the number of times a node has to be re-visited, and thus, reducing the CPU runtime of the state restoration algorithm. As will be discussed in the following subsection, bitwise parallelism can also be exploited when using this implementation.

By forward propagating and backward justifying known data between gates in the circuit, data can be restored for other state elements one clock cycle at a time. It is essential to note that our proposed algorithm aims at restoring data for sequential elements across multiple time frames, which is different to [44] where only values in the combinational logic are reconstructed with the known data in the sequential elements. When comparing the amount of data that is available before and after state restoration, fourteen data will be available in the entire circuit after applying the restoration process on only four initial data from FFC . This gives a *restoration ratio* of $14/4 = 3.5X$ for the elaborated example. It should be noted that the amount

of data that can be reconstructed using state restoration greatly depends on the initial set of sampled data. For instance, if only FFE is sampled in Figure 4.1(a), no new data can be reconstructed for any state elements in the circuit. The computation time for the state restoration algorithm is proportional to the amount of nodes presented in the circuit graph. In addition, unlike ATPG, when restoring state data for a circuit, a large number of clock cycles will have to be considered. This affects the CPU runtime for reconstructing the missing data. Although bitwise parallelization has been explored to speed up logic simulation across multiple clock cycles, it is only performed for the forward operation [45]. As a result, in the following section, we introduce how bitwise parallelization is exploited by eliminating any branching decisions in the state restoration process for all the principal operations introduced in Figure 2.15.

Algorithm 4.1 Algorithm for state restoration

```

1: search_list = Trace_Signal_List
2: while search_list is not empty do
3:   cur_node = first node in search_list
4:   for (each parent_node of cur_node) do
5:     BackwardOperation(cur_node, parent_node)
6:     if (new data is restored for parent_node) then
7:       Put parent_node at end of search_list
8:     end if
9:   end for
10:  for (each child_node of cur_node) do
11:    ForwardOperation(cur_node, child_node)
12:    if (new data is restored for child_node) then
13:      Put child_node at end of search_list
14:    end if
15:  end for
16: end while

```

4.2 Exploiting bitwise parallelism for state restoration

In order for the state restoration algorithm to be applicable to large circuits, it is essential for the state restoration algorithm to be compute-efficient. This is because the designer may need to test the circuit with different stimuli when iterating through steps 2 to 7 in Figure 2.6 during the debug process. Thus, it is desirable to restore data as fast as possible for each stimulus to reduce debug time. In order to reduce computation time for the state restoration algorithm, we explore two facts when applying the principal operations on a node. Firstly, to restore data for one clock cycle in a given node, a branch decision will have to be made to see if the data can be reconstructed. Thus, the computation time for restoring data across all the circuit nodes for a large number of clock cycles depends on how well these decisions are made during program execution. Also, one can parallelize the algorithm by allowing multiple branch decisions to be evaluated at the same time. This is feasible for state restoration because when performing principal operations on a node for multiple clock cycles, the results are independent of each other for each data point in different clock cycles. This idea can be better explained using the example shown in Figure 4.1. In order to restore data for the circuit for five clock cycles, one can iteratively apply the principal operations to each node in each clock cycle. However, the same outcome can be achieved by allowing five different branch decisions to be evaluated at the same time with the corresponding data for the specific clock cycles. Nevertheless, since a debugged circuit can contain tens of thousands of logic gates, and data is usually restored over thousands of clock cycles, speeding up the algorithm by parallelizing the branch decisions may still incur large computation time due to the high execution penalty from mispredictions. As a result, we derive new logic operations such that the principal operations can be applied concurrently at a node across multiple clock cycles, without the need to evaluate any branch decisions during state restoration.

We exploit the integer data type in ANSI C on a 32-bit platform to enhance the performance of our algorithm by storing data for 32 consecutive clock cycles in two integers (8 bytes) for each node. For example, to represent the data $[0, 1, 1, 0, X]$ for

Table 4.1: Two bit codes for data representation

Logic value	Two bit code
0	00
1	11
undefined	01, 10

clock cycles 0-4 for FFC in Figure 4.1(a), using the two-bit codes in Table 4.1, we can store the data for FFC using two integer variables as follows:

$$int0 = 0, 1, 1, 0, 1, \dots, 1$$

$$int1 = 0, 1, 1, 0, 0, \dots, 0$$

In these equations, the first 5 bits of the two variables store the data for clock cycles 0-4 for FFC , and the remaining 27 bits store the code for undefined data. By working with two integer variables, the algorithm can restore data for 32 consecutive clock cycles at a time using a sequence of logic equations based on the bitwise operations provided by ANSI C for each of the primitive gates. For each principal operation, two different equations (one for each integer) will be developed in such way that the number of 2-operand bitwise operations are minimized. Although the formalism of multi-valued logic and input/output encoding from logic synthesis can be used to derive these systems of equations [79], the following discussion relies on the illustrative advantage of the K-map representation.

Figures 4.2(a) and 4.2(b) show the K-maps for deriving the logic equations for the forward operation at the output z , while Figures 4.2(c) and 4.2(d) give the K-maps for the backward equations at the input a of a two-input AND gate. Note that the inputs of the AND gate are labeled a and b , and since two bits are needed for data representation, the variables for the logic equations are labeled a_0, a_1, b_0, b_1 for the inputs, and z_0, z_1 for the output. From boolean algebra we know that when any input of an AND gate is 0, the output should also be a 0. This is why the entries are set to 0 on the first rows and the left-most columns of the K-maps for z_0 and z_1 in Figures 4.2(a) and 4.2(b). Also, when both inputs are logic 1, z_0 and z_1 are both

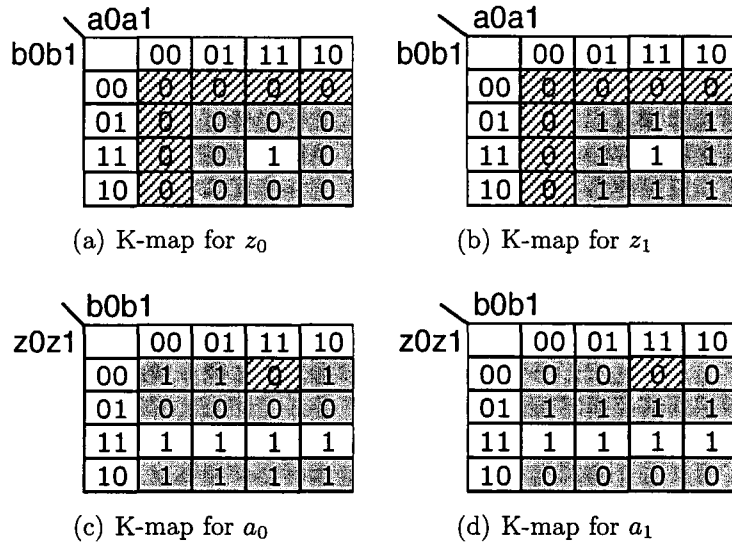


Figure 4.2: Derivation of forward and backward equations for the *AND* gate

set to 1 to represent a logic 1 on the output of the *AND* gate. The shaded regions of the K-maps in the Figures 4.2(a) and 4.2(b) represent that the output port of the *AND* gate is inconclusive due to the insufficient data from the input ports. In these regions, the values of z_0 and z_1 can be filled in such way that the resulting code is 01 or 10 (in order to represent the undefined values according to the two-bit codes in Table 4.1) and, at the same time, the number of bitwise operations is minimized. The K-maps for deriving the logic equations for the backward operation can also be constructed using the same principle. For instance, as can be seen in Figures 4.2(c) and 4.2(d), when the output of the *AND* gate is logic 1, the inputs of the gate can be justified to logic 1. Also, input a can be concluded as logic 0 when the output z is 0, and the other input b is 1. In any other cases, the value for a is inconclusive, as indicated by the shaded regions in Figures 4.2(c) and 4.2(d). Note that for backward operation, four logic equations (two equations for each input port) will be derived. However, the variables in the logic equations among input ports will be exchanged.

In this example for an *AND* gate, the equations for forward and backward operations are:

$$\begin{aligned}
 z_0 &= a_0 a_1 b_0 b_1 \\
 z_1 &= (a_0 + a_1)(b_0 + b_1) \\
 a_0 &= z_0 + \overline{(z_1 + b_0 b_1)} \\
 a_1 &= z_1 \\
 b_0 &= z_0 + \overline{(z_1 + a_0 a_1)} \\
 b_1 &= z_1
 \end{aligned}$$

Using these logic equations, if the input ports of an *AND* gate have the following data for 32 clock cycles:

$$\begin{aligned}
 a &= 0, X, 1, \dots, 1, X, \dots, X \\
 b &= X, 1, 1, \dots, 1, 0, \dots, 0
 \end{aligned}$$

Using our two-bit code in Table 4.1, the input to the logic equations will be translated into four integer variables as:

$$\begin{aligned}
 a_0 &= 0, 1, 1, \dots, 1, 1, \dots, 1 \\
 a_1 &= 0, 0, 1, \dots, 1, 0, \dots, 0 \\
 b_0 &= 1, 1, 1, \dots, 1, 0, \dots, 0 \\
 b_1 &= 0, 1, 1, \dots, 1, 0, \dots, 0
 \end{aligned}$$

The output z of the *AND* gate can then be obtained by applying the a_0 , a_1 , b_0 and b_1 data into the forward operation:

$$\begin{aligned}
 z_0 &= 0, 0, 1, \dots, 1, 0, \dots, 0 \\
 z_1 &= 0, 1, 1, \dots, 1, 0, \dots, 0
 \end{aligned}$$

Referring back to the two-bit code, the output z of the *AND* gate for the 32 clock cycles can be found as:

$$z = 0, X, 1, \dots, 1, 0, \dots, 0$$

In the above equations, if the values of the output variables are known, these values will be overwritten by the results of the equations. For example, in the forward equations introduced for z_0 and z_1 , if z_0 and z_1 are known from sampling the circuit, while a_0 , a_1 , b_0 and b_1 are not known, applying the forward equations will overwrite the sampled values in z_0 and z_1 with unknowns. This is because the existing values in the output variables are not considered in the equations. As a result, additional operations will have to be added to preserve the existing values in the output variables if they are known. The modified forward and backward equations for the *AND* gate will then become:

$$\begin{aligned}
z_0 &= \overline{(z_0 \oplus z_1)}z_0 + (z_0 \oplus z_1)(a_0a_1b_0b_1) \\
z_1 &= \overline{(z_0 \oplus z_1)}z_1 + (z_0 \oplus z_1)(a_0 + a_1)(b_0 + b_1) \\
a_0 &= \overline{(a_0 \oplus a_1)}a_0 + (a_0 \oplus a_1)(z_0 + \overline{(z_1 + b_0b_1)}) \\
a_1 &= \overline{(a_0 \oplus a_1)}a_1 + (a_0 \oplus a_1)z_1 \\
b_0 &= \overline{(b_0 \oplus b_1)}b_0 + (b_0 \oplus b_1)(z_0 + \overline{(z_1 + a_0a_1)}) \\
b_1 &= \overline{(b_0 \oplus b_1)}b_1 + (b_0 \oplus b_1)z_1
\end{aligned}$$

Note that the \oplus symbol represents the *XOR* operation. With these equations, the values of the output variables will only be overwritten when the existing values are unknown (i.e., the variable pair has a value of 01 or 10 as defined in Table 4.1).

By replacing the implementation of the *BackwardOperation* and *ForwardOperation* in Algorithm 4.1 with these logic equations to restore data in 32 clock cycles, the total number of bitwise operations for the *AND* gate can be calculated as follows. For z_0 , one operation will be needed for the *XOR* between its previous values. Note that the result of this *XOR* can be reused to reduce the number of bitwise operations in the equations. In addition, one inversion is needed for the *XNOR*, five bitwise *AND* operations and one bitwise *OR* operation are needed. On the other hand, three bitwise *AND* operations and three bitwise *OR* operations are required for z_1 . Thus, 14 bitwise operations will be performed for forward propagating data in the *AND* gate. For backward justifying data in the *AND* gate, there are one *XOR*, two *NOT*, three *AND* and three *OR* bitwise operations for a_0 , while a_1 requires only two *AND* and one *OR* operations. Together, 12 bitwise operations will be needed

to backward justify data for one input. Thus, a total of 24 bitwise operations are necessary for backward justifying data on a_0 , a_1 , b_0 and b_1 for the *AND* gate. Although the derivation of forward and backward equations for other primitive gates are not shown, they can be obtained using similar concepts from the above elaborated example. They range from 10 bitwise operation for forward operations of *NOT*, to 33 bitwise operations for backward operations of *XOR* as shown in Table 4.2. It should be noted that the proposed method requires multiple CPU instructions to restore data for 32 clock cycles. In fact, the proposed method will need 33 CPU instructions if the 32 data are reconstructed using the backward operation on the *XOR* gate. On the other hand, one can implement only 32 if-then-else instructions to reconstruct the same amount of data. However, these 32 if-then-else instructions do not necessarily translate to only 32 CPU instructions since mispredictions during branching incur execution penalty. As will be discussed in the experimental results, using the proposed method, which eliminates branching, can reduce execution time of the state restoration algorithm. Moreover, the proposed method can further scale down the execution time when the executing platform has higher bitwidth (e.g., 64-bit CPUs). This is because the number of CPU instructions required to perform the principal operations with the proposed method will not change, while the number of clock cycles in which one can reconstruct data increases with the bitwidth of the platform. It should also be emphasized that digital circuits often involve more complex logic gates, or logic gates with higher fan-in. These complex gates can be either decomposed into a hierarchy of two-input primitive gates (such that the derived operations can be applied on the circuit graph that has more nodes, and thus prolongs the computation time), or additional equations specific to their behavior can be generated.

4.3 Experimental results

Experimental studies from [2] indicate that trace buffers of size 1k x 8 (i.e., depth of 1024 and width of 8 bits) to 8k x 32 are acceptable in the practice today. If one needs to debug larger logic blocks, it is common that the trace buffer is used as a time-shared resource [93]. Time sharing is also justified by the fact that if at most

Table 4.2: Number of bitwise operations for the two-input primitive gates

Logic operation	# of bitwise operations	
	Forward	Backward
<i>NOT</i>	10	10
<i>AND</i>	14	24
<i>NAND</i>	16	25
<i>OR</i>	14	24
<i>NOR</i>	16	25
<i>XOR</i>	17	33
<i>XNOR</i>	16	32

32 bits per sample are acquired, it is difficult to restore values to more than 2,000 FFs (which normally belong to a logic block in the range of 50 thousand gates). Given this expected logic block size, we perform our experiments on the three largest ISCAS89 benchmark circuits [18] (i.e. s38584, s38417 and s35932), which fit the gate count range for acceptable trace buffer sizes as of today. Moreover, since the ISCAS89 benchmark circuits are publicly available, we hope that future proposals on this emerging area can benchmark their algorithms against ours. For our experiments, the state restoration algorithm is implemented using ANSI C and executed on a PC with dual-Xeon processors at 2.4 GHz with 1 GB of RAM. Also, all the high fan-in gates are decomposed into two-input logic gates when translating the ISCAS circuits into circuit graphs for the state restoration algorithm.

There are two sets of experimental results for the state restoration algorithm discussed in this chapter of the thesis. The first set of results shown in SubSection 4.3.1 is based on randomly generated stimuli on all primary inputs of the circuits. And the second set of results discussed in SubSection 4.3.2 is based on randomly generated stimuli on the data inputs, and the values supplied to the control inputs are constrained. Note that for all of these experiments, the trace signals are selected randomly.

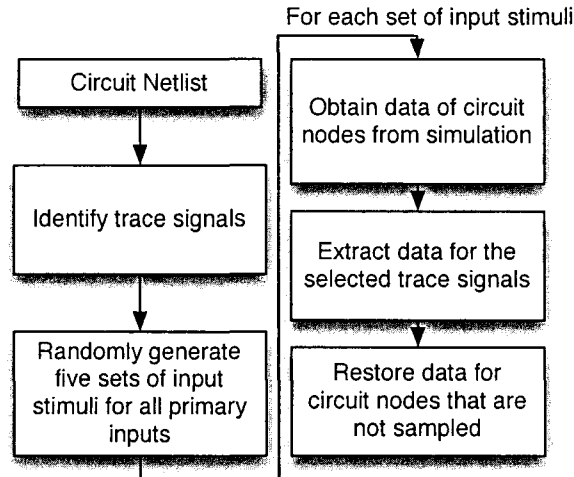


Figure 4.3: Experimental flow for randomly generated input stimuli

4.3.1 Experiments with randomly generated stimuli on all inputs

Experimental setup for randomly generated input stimuli

The experimental flow for generating random stimuli on all primary inputs for state restoration and trace signal identification algorithms, which will be discussed in the following chapter of the thesis, is shown in Figure 4.3. In order to obtain the set of debug data on which state restoration will be applied during the experiment, input stimuli for *all* the primary inputs of the circuit are randomly generated and then fed to a simulator for a number of clock cycles (the clock cycle count depends on the trace buffer depth). The results from these experiments with random input stimuli are then used for evaluating the state restoration algorithm.

Results with randomly generated stimuli on all primary inputs

Tables 4.3, 4.4 and 4.5 show the state restoration ratio and the data reconstruction time between the implementation exploiting the bitwise parallelism as discussed in Section 4.2 (columns labeled *Non-branching*), and the if-then-else implementation

Table 4.3: State restoration results for s38584 when trace signals are selected randomly and control signals are driven randomly

Buffer depth	Buffer width	Seed	Ratio	Time (sec)	
				Non-branching	Branching
8192	8	1	1.00	0.02	0.13
		1000	1.00	0.03	0.11
		10000	1.00	0.02	0.07
		50000	2.75	0.04	0.21
		100000	2.75	0.03	0.15
	16	1	1.00	0.04	0.18
		1000	1.00	0.04	0.17
		10000	1.00	0.03	0.14
		50000	3.31	0.18	0.91
		100000	3.25	0.17	0.84
	32	1	4.16	0.09	0.42
		1000	3.01	0.08	0.33
		10000	3.50	0.09	0.37
		50000	8.27	0.23	1.11
		100000	7.21	0.20	0.96

Table 4.4: State restoration results for s38417 when trace signals are selected randomly and control signals are driven randomly

Buffer depth	Buffer width	Seed	Ratio	Time (sec)	
				Non-branching	Branching
8192	8	1	12.00	0.84	3.71
		1000	6.83	53.49	259.85
		10000	14.25	1.09	5.06
		50000	8.72	0.82	3.82
		100000	1.38	0.03	0.12
	16	1	6.75	0.73	3.60
		1000	7.62	49.66	294.26
		10000	10.00	0.95	4.40
		50000	7.05	1.01	4.34
		100000	4.88	0.18	0.70
	32	1	6.97	1.38	6.02
		1000	5.43	53.20	261.28
		10000	6.81	1.21	4.53
		50000	6.22	1.02	4.70
		100000	4.21	0.28	1.24

Table 4.5: State restoration results for s35932 when trace signals are selected randomly and control signals are driven randomly

Buffer depth	Buffer width	Seed	Ratio	Time (sec)	
				Non-branching	Branching
8192	8	1	1.70	0.01	0.03
		1000	1.88	0.01	0.03
		10000	3.27	0.23	2.40
		50000	1.78	0.01	0.03
		100000	2.57	0.85	3.95
	16	1	27.30	2.40	6.77
		1000	9.14	1.60	6.59
		10000	15.00	1.63	5.82
		50000	108.03	2.48	6.55
		100000	43.88	2.67	7.18
	32	1	24.62	2.66	7.23
		1000	20.27	4.07	10.81
		10000	22.41	2.75	7.61
		50000	54.51	2.45	6.57
		100000	28.99	3.17	8.13

(columns labeled *Branching*) for s38584, s38417 and s35932 respectively. In these tables, the trace signal selection was done randomly by using five different seeds for the pseudo-random generator in ANSI C. Also, fifty different sets of random data are generated for all the primary inputs (both control and data inputs) of the circuits. The random data on the primary inputs will then be fed to a simulator to obtain the debug data on the trace signals for state restoration as illustrated in Figure 4.3.

As shown in these tables, the computation time for the data restoration algorithm using the parallel equations is on average about four to five times less than the non-accelerated method. The speedup falls short of the theoretical upper bound of 32X (since the accelerated algorithm restores data for 32 clock cycles at a time on a 32-bit machine). This is because restoring data in 32 clock cycles using the accelerated method involves more than one CPU instruction as shown in Table 4.2. On the other hand, the non-accelerated method will need one if-then-else statement to perform the

principal operations for each data point. This results in 32 if-then-else statements to restore data for 32 clock cycles. However, the number of CPU instructions executed to restore each data with the non-accelerated method depends on how a branch is taken during program execution. As verified by the results, the total number of CPU instructions required to perform data restoration across multiple clock cycles using the non-accelerated method is higher than that of the accelerated method. Moreover, as machines with higher bitwidth become available, the accelerated method can better scale to utilize the larger bitwidth and further speed up the state restoration process when compared with the if-then-else implementation, as discussed in Section 4.2.

Another interesting point to note is that in Table 4.4, with the same trace buffer depth and width, by randomly selecting different trace signals, the restoration time increases significantly even though the restoration ratio decreases (which indicates less data to be restored). This is because if the sampled signal resides in a sequential loop, and the missing data in the loop cannot be reconstructed by the side signals that are connected to the gates from this loop, the restoration algorithm may have to iterate in the loop to restore data one clock cycle at a time. In this case, more decisions will need to be made by the algorithm to check if the newly restored data can help reconstruct any other data in the neighboring nodes in every clock cycle, and thus, increasing the runtime. It is also interesting to note that with the same trace buffer depth and width, sampling different signals varies the amount of data that can be reconstructed even if the input data is identical. This is why in the next chapter of the thesis, an algorithmic solution for selecting trace signals will be detailed.

4.3.2 Experiments with constrained generated stimuli for control inputs

In practice, the circuit is exercised with real-time stimuli coming from the environment. However, for benchmarking purposes, the experimental results from the previous subsection relies on *random* input stimuli. While it is important to provide random stimuli for benchmarking purposes on the *data inputs*, there may be exceptions on *control inputs* that do not need to always be random. For example, the

values on bus control signals or signals that setup the circuit's mode of operation or synchronous resets/enables, will not be updated randomly during the normal circuit operation. Rather, they will follow pre-defined protocols based on how the chip is supposed to operate in the native mode. As a result, using random stimuli on these types of control signals for benchmarking purposes may not reflect how the circuit behaves in a field debug experiment.

The focus of this subsection of the thesis is to first provide a method for *automatically* identifying signals that need to have constrained input values during the normal circuit operation. Then, we elaborate on the experimental flow for providing a *constrained-random* benchmarking experiment to evaluate the state restoration algorithm and the trace signal selection algorithms presented in the following chapter of the thesis. We do recognize that this type of constrained-random experiments have little significance in a practical debug setup (because the data will come from the real-life sources). Nonetheless, developing experimental flows for generating constrained-random experiments is essential for researchers to benchmark, as well as for designers to evaluate in more depth, the proposed algorithm.

Identification of constrained signals

Figure 4.4 shows the procedure for identifying the control signals and its constrained value by using an ATPG tool. Note, the circuit is considered to have full scan and the ATPG tool generates patterns only for combinational logic. Hence, fault effects are observed at both primary outputs and pseudo-outputs (i.e., scan flip-flops). The generated test cubes are not compacted nor compressed in order to preserve don't care (X) bits in the patterns. Then test cubes are analyzed to identify the signals that have high care bit density (i.e., for most of the patterns the value is not an X). When a signal has high care bit density, this signal *must* be controlled in order for faults to be detected and propagated during test. For example, if a synchronous reset signal is used, then the ATPG patterns will drive it to its non-controlling value because this is the only way how the fault-effects from the circuit will be propagated to the observable outputs (i.e., scan flip-flops).

When a signal that has high care bit density is identified, it should be checked

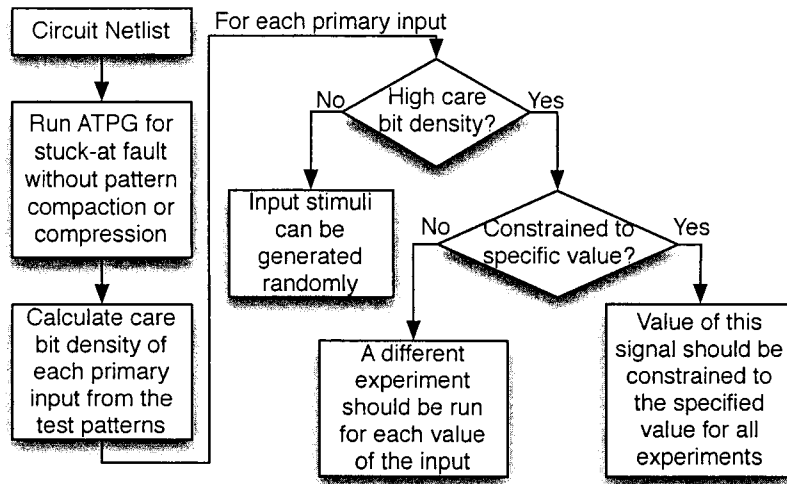


Figure 4.4: Procedure for identifying signals with constrained value

whether the signal is constrained to the same value all the time. If that is the case, the generated input stimuli for such signals should be a constant value over the number of clock cycles considered during the debug experiment. For example, the input stimuli for a global active-high reset signal should be 0 in order to avoid over-resetting the circuit during the experiment. On the other hand, a primary input used for mode selection will also have high care bit density from the test patterns, but it will not be forced to be a constant value for every single clock cycle. In this case, the experiment should be run with different sets of input stimuli such that the effectiveness of the state restoration algorithms can be benchmarked for *every mode* of the circuit operation.

The benefit of employing ATPG at the core of our flow is threefold. First, ATPG tools are available both commercially and in the public domain; hence no specialized algorithms need to be developed and shared by the researchers who work on identification of trace signals for debug. Second, ATPG results provide explicitly not only which signals need to be constrained, but they also indicate the values to which these signals need to be constrained. Third, the existing ATPG tools are optimized for runtime. Even if generating test cubes that are not compacted nor compressed may seem intractable, for the purpose of identifying the constrained signals it is sufficient

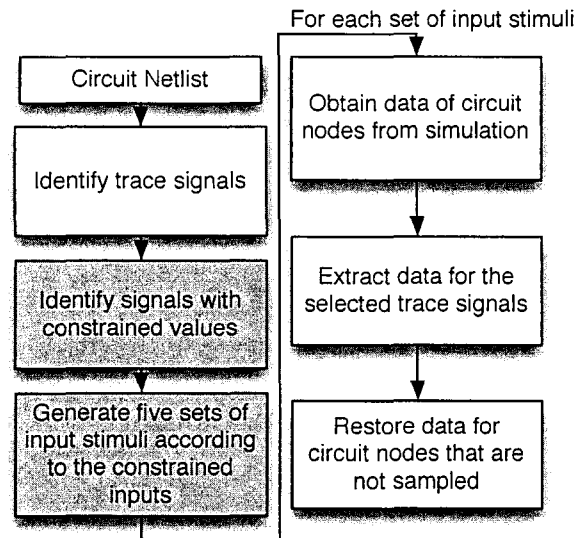


Figure 4.5: Revised experimental flow

to generate only a few hundreds to thousands of patterns. The entire test set is not required and, more importantly, the abort time for identifying untestable faults can be set to a low value.

New setup for constrained-random benchmarking experiments

The revised experimental flow for generating constrained-random benchmarking experiments for state restoration and trace signal identification algorithms that will be discussed in the following chapter of the thesis is shown in Figure 4.5. Instead of generating input stimuli randomly for all the primary inputs, the stimuli for the constrained-signals should be generated according to the information provided by the ATPG engine, as explained in the previous section.

It should be noted that randomness is still employed for all the data-intensive inputs that have very low care-bit density in the ATPG test set. Based on this new experimental flow, we expect the trace signal identification and state restoration algorithms presented in this thesis to be evaluated in a virtual environment that better matches a practical setup for post-silicon validation.

Table 4.6: Signals with constrained values

Circuit name	Signals with constant value		Signals with high care bit density
	name	value	
s38584	g35	1	
s35932	RESET	1	TM0 TM1

Results with deterministically driven control inputs

The new experimental flow has been applied to the ISCAS89 benchmark circuits [18] to re-evaluate the algorithms presented in this chapter of the thesis with constrained-random input stimuli. We have applied the ATPG flow using a third party ATPG tool [109] for identifying constrained signals on s38584 and s35932. Table 4.6 shows the identified signals that have constant constrained value and signals with high care bit density. It should be noted that the results for s38417 with deterministically driven control inputs are not reported here. This is because after applying the procedure for identifying the signals that have constrained value on s38417, we have found out that there are no signals which needed to be constrained.

It can be seen from Table 4.6 that for s38584, the primary input *g35* should be constrained to have a value of 1 during native mode since it is an implied reset that is active-low. With this constrained value of *g35*, five different sets of stimuli are generated randomly for all the other primary inputs (that have low care bit density). Similarly, the signal *RESET* in s35932 is an active-low reset and should be constrained to be 1. In addition, the signals *TM0* and *TM1* together act as mode selection signals for s35932. Thus, in the new experimental flow described in the previous subsection, four sets of experiments are run using values $\{TM0, TM1\} = \{00, 01, 10, 11\}$, while for each combination of the *TM0* and *TM1* values, five different sets of random stimuli are generated for all the other primary inputs. The results from the $4 \times 5 = 20$ sets of input stimuli are then averaged for providing the results on the presented algorithms.

Table 4.7: State restoration results for s38584 when trace signals are selected randomly and control signals are driven deterministically

Buffer depth	Buffer width	Seed	Ratio	Time (sec)	
				Non-branching	Branching
8192	8	1	1.00	0.04	0.12
		1000	1.00	0.04	0.11
		10000	1.00	0.03	0.07
		50000	2.75	0.06	0.20
		100000	2.75	0.05	0.14
	16	1	1.00	0.05	0.17
		1000	1.00	0.05	0.16
		10000	1.00	0.04	0.14
		50000	3.31	0.26	0.91
		100000	3.25	0.24	0.82
	32	1	1.14	0.11	0.39
		1000	1.83	0.46	1.68
		10000	2.32	0.44	1.63
		50000	3.53	0.68	2.47
		100000	3.82	0.64	2.31

Table 4.8: State restoration results for s35932 when trace signals are selected randomly and control signals are driven deterministically

Buffer depth	Buffer width	Seed	Ratio	Time (sec)	
				Non-branching	Branching
8192	8	1	1.13	0.09	0.46
		1000	1.09	0.10	0.74
		10000	2.13	0.07	0.40
		50000	1.13	0.09	0.46
		100000	1.89	0.11	0.73
	16	1	2.06	0.06	0.39
		1000	2.46	0.12	0.76
		10000	3.56	0.12	0.52
		50000	2.07	0.07	0.40
		100000	3.05	0.14	0.78
	32	1	2.91	0.15	0.62
		1000	3.14	0.18	0.66
		10000	3.53	0.19	0.71
		50000	3.03	0.15	0.63
		100000	3.26	0.18	0.70

Tables 4.7 and 4.8 show the state restoration ratio and the data reconstruction time using enhanced state restoration algorithm discussed in Section 4.2 for s38584 and s35932 respectively. In these tables, the trace signal selection was done randomly by using five different seeds for the pseudo-random generator in ANSI C. Also, five different sets of random data are generated for all the data inputs, while the values on the control inputs are constrained according to the results reported in Table 4.6. The generated data on the primary inputs will then be fed to a simulator to obtain the debug data on the trace signals for state restoration as illustrated in Figure 4.5.

It can be seen that the restoration ratios obtained using the new experimental flow are lower than what was provided in SubSection 4.3.1. This is because the results for those experiments are performed with randomly generated stimuli for all primary inputs, and thus, did not provide any constraints on what the value should be on the reset signals for both s38584 and s35932. In this case, every time the circuits are reset, the state restoration algorithm will be able to reconstruct the values of all the circuits' nodes easily.

It should be noted that although the restoration ratios are lower when signals with constrained values are considered, it does not mean that the benefit from employing the state restoration algorithm is diminished. In fact, the new results actually strengthen the ability of the proposed algorithm in improving observability of internal signals during post-silicon validation. This is because it is currently proven that debug data in circuit nodes that are not monitored by trace buffers can actually be restored under realistic input stimuli.

4.4 Summary

The limited storage space available from trace buffers inside ELAs constrains how much data can be acquired from the CUD during a post-silicon validation and debug experiment. In this chapter of the thesis, we have shown how the acquired data can be used to reconstruct missing information on signals that are not traced. For logic bugs for which the circuit implementation matches its physical prototype, our algorithmic solution effectively helps improve observability of the CUD.

We have first discussed a sequential algorithm for restoring data in state elements over multiple time-frames. We have defined a state restoration metric (called restoration ratio) to assess the effectiveness of the algorithm. Then we have shown how the state restoration algorithm can be sped up by exploiting the bitwise parallelism inherent in data representation in computers. Because conditional instructions are avoided (and hence execution time penalty due to branch mis-prediction is eliminated), the objective is to minimize the number of bitwise logic operations that manipulate the restored data over multiple time-frames. It was shown how concepts from two-level logic synthesis can be leveraged for this purpose. This parallel algorithm is approximately four to five times faster than its sequential counterpart.

Chapter 5

Automated trace signal selection

When designing the debug infrastructure, the designer will decide which key signals will be traced in silicon. However, it is often the case that the width of the trace buffer is larger than the number of the key signals known by the designer. Hence, a question faced by designers is which other signals are most suitable to be connected to the trace buffer? For example, if the trace buffer width is 32 bits and only 10 signals are key signals selected manually by the designer, then which other 22 signals should be fed into the trace buffer. Because the problems that will be examined in silicon will obviously not be known at design-time, we advocate that the state restoration ratio introduced in the previous chapter can be used as a driving metric that can assist designers with the decision on which signals to trace. This is because the signals which will lead to an improved restoration ratio will also improve the real-time observability by providing more data to the user at the same hardware cost.

In this chapter, two new metrics that will influence the selection of trace signals will be presented. The first metric discussed in Section 5.1 accounts for the topology of the CUD, while the second metric detailed in Section 5.2 considers the logic behavior in addition to the circuit topology. These metrics help identify trace signals that should be sampled such that high state restoration ratio can be achieved as supported by the experimental results given in Section 5.3. Finally, Section 5.4 summarizes the contribution of this work.

5.1 Trace signal selection using circuit topology

The first metric for aiding the selection of trace signals utilizes the information obtained by analyzing the topology of a circuit. If a trace signal has large input and output logic cones (i.e., if the trace signal is driven by a large number of signals or if it drives a large number of other signals), it is obvious that the likelihood of restoring data for other signals through forward and backward operations on this signal will be higher. This is because when the circuit netlist is translated into a graph as described in Section 4.1, the number of parent and child nodes that can be reached from the trace signal is higher. As a result, by analyzing the topology of a circuit, one can select a set of trace signals that can help restore a significant amount of missing data for other signals. This observation is captured by the equations shown in Figure 5.1 for calculating the *restorability* of all the nodes in a circuit when a particular signal is monitored by the DFD hardware. We define the *forward restorability* of a node to be the likelihood of restoring data of that node through forward propagation (Figure 2.15(a)), while *backward restorability* represents the chance of restoring data of the node from backward justification (Figure 2.15(b)) or the combined operation (Figure 2.15(c)). When a node can be fully restored through forward (backward) operations, the forward (backward) restorability will be 1. If data cannot be reconstructed, the restorability will then be 0.

In Figure 5.1, the equations for calculating the forward restorability of the output, as well as the backward restorability of the inputs of a gate are shown. Although it is shown with an *AND* gate in the figure, it should be noted that the same set of equations are applied to all the gates in a design. The forward restorability of a gate is calculated by summing the forward restorability of its inputs, then dividing the sum by the number of inputs of the gate. This is because the more parent signals of a node are monitored, the higher chance the data for that node can be restored by the proposed state restoration algorithm through forward operations. On the other hand, the backward restorability of an input of a gate is calculated by first finding the maximum backward restorability from its children, since it can be restored through any one of its fan-out branches (as indicated in Figure 5.1 with the term $\max\{B(z)\}$)

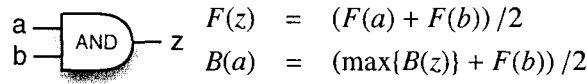


Figure 5.1: Equations for restorability calculation using only topology

in the equations). The result is then added with the forward restorability of other inputs of the gate. This is because it is sometimes insufficient to restore the input values from only the output values of a gate. However, if the values of the output and the other inputs are known, the chances of restoring values of the targeted input will be higher. The sum from the equation is then divided by the number of inputs of a gate to normalize the restorability to 1.

After defining the equations that look at the topology of a circuit for calculating restorability of a node, Algorithm 5.1 can be used to show how these equations are applied to aid the selection of trace signals. The algorithm uses a breadth-first-search approach to calculate restorability values for all the nodes. The calculation starts by first computing the forward restorability of all the child nodes of the first node in the search list (line 8). It then works out the backward restorability of all the parent nodes of the same node (line 14). When sequential loops are found in a circuit, the algorithm will iterate the forward and backward calculations for the nodes in the loop. This is because the state restoration algorithm may be able to restore data for multiple clock cycles by iterating in the loop.

Figure 5.2 can be used to explain the greedy nature of the algorithm when trying to select the first trace signal using Algorithm 5.1 for the circuit shown in Figure 4.1(a). In the figure, the F and B values represent the forward and backward restorability of each flip-flop respectively for three iterations. Note that for the sake of clarity, only the restorability values of the flip-flops are shown, but in fact, the restorability for the logic gates between flip-flops are also calculated.

In Figure 5.2(a), the restorability values of all the signals in the circuit when FFE is selected as the trace signal using Algorithm 5.1 are shown. In the first iteration, all the F and B values of the signals are set to 0, except for FFE , for which the

Algorithm 5.1 Algorithm for identifying trace signals incrementally

```

1: while  $cur\_width < TB\_width$  do
2:   while not all nodes in Circuit are calculated do
3:      $search\_list = \text{Get chosen nodes}$ 
4:     Set initial values for chosen nodes
5:     while  $search\_list$  is not empty do
6:        $cur\_node = \text{first node in } search\_list$ 
7:       for (each  $child\_node$  of  $cur\_node$ ) do
8:          $\text{CalculateForward}(child\_node)$ 
9:         if  $(new\_value - old\_value \geq Threshold)$  then
10:          Put  $child\_node$  at end of  $search\_list$ 
11:        end if
12:      end for
13:      for (each  $parent\_node$  of  $cur\_node$ ) do
14:         $\text{CalculateBackward}(parent\_node)$ 
15:        if  $(new\_value - old\_value \geq Threshold)$  then
16:          Put  $parent\_node$  at end of  $search\_list$ 
17:        end if
18:      end for
19:    end while
20:    Sum the restorability of all nodes in the circuit
21:  end while
22:  Select the node with highest restorability
23:   $cur\_width++$ 
24: end while
25: return  $signal\_selection\_list$ 

```

values are set to 1 since it is traced. In the second iteration, the backward restorability values of FFC and FFB are calculated to be 0.5 according to the backward equation in Figure 5.1. It should be noted that although the forward restorability values of FFC and FFB should be 0 according to the forward equation, the values will be updated to be 0.5 at the end of the iteration. This is to reflect that no matter if data is restored through forward or backward operation, the restored data can still be used for state restoration in the next iteration. The updated values will then be used in the third iteration of Algorithm 5.1 for further calculation.

It can be seen in Figure 5.2 that as the algorithm iterates further to propagate the metric, the restorability value of each node either stays the same or it gradually increases (it never decreases). As a result, a user-defined parameter *Threshold* is provided to control the amount of metric propagation among circuit nodes, and to limit computation time for the trace signal selection algorithm (lines 9 and 15). This threshold is used to check the newly computed values against the ones from the previous iteration. It is obvious that the lower the threshold, the more effort the algorithm will spend on calculating the restorability of each signal in the circuit.

After the restorability of all nodes are calculated and the *Threshold* parameter is satisfied, the calculated values are summed together to give the restorability of the circuit given that a certain FF was selected as the trace signal (line 20). To decide which node to select as the trace signal, the algorithm will calculate the circuit restorability for when each node is selected, it will then choose the node that produces the highest circuit restorability as the trace signal (line 22). To select the targeted number of trace signals, Algorithm 5.1 incrementally calculates circuit restorability to select one signal at a time in a greedy manner. Using the circuit in Figure 4.1(a) as an example when choosing two signals, and assuming signal FFE is chosen in this iteration, Algorithm 5.2 will then select the second trace signal by trying to select signals in this sequence: FFE & FFA , FFE & FFB , \dots , until all other signals are selected together with FFE as trace signals. It will then choose the signal pair that will produce the highest restorability values across the circuit as trace signals. This gradual approach for trace signals selection follows the same philosophy on how new states are chosen to be probed during microprocessor debug, when signals are also

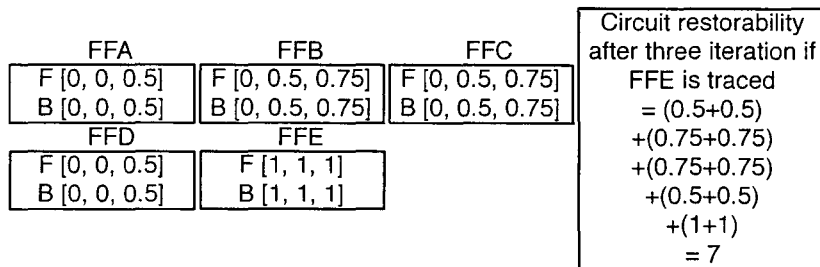
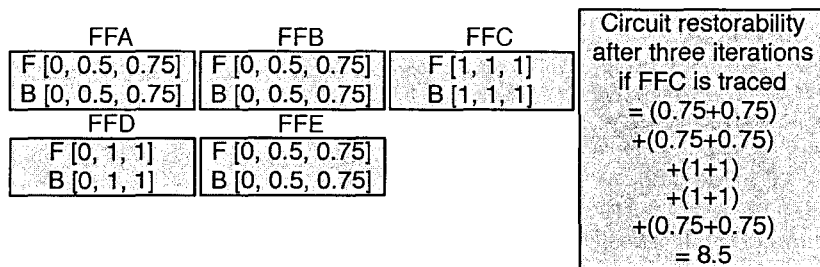
(a) Circuit restorability when *FFE* is selected(b) Circuit restorability when *FFC* is selected

Figure 5.2: Restorability calculation for three iterations using the topology metric

selected incrementally to determine what additional data can be gathered from the microprocessor.

Although Algorithm 5.1 can identify the trace signals that would give high restoration ratio by propagating the restorability metric over and over with each incremental signal selection, this could lead to prohibitively high computation time for large circuits. Thus, we introduce Algorithm 5.2, which greedily selects trace signals by estimating how much data from other signals can be restored when each signal is chosen as a trace signal.

To estimate how much data can be restored when a signal is traced, Algorithm 5.2 first sets the restorability value of the selected signal to 1 (line 3). It then employs a breath-first-search approach to propagate the restorability value using the equations in Figure 5.1 to its child nodes and parent nodes through forward propagation

Algorithm 5.2 Algorithm for selecting trace signals using coverage estimation

```

1: while not all signals in Circuit are evaluated do
2:   search_list = one of the non-evaluated signal
3:   Set initial values for chosen signal
4:   while search_list is not empty do
5:     cur_node = first node in search_list
6:     for (each child_node of cur_node) do
7:       if (number of parent nodes with non-zero restorability values has increased
           since the last visit) then
8:         CalculateForward(child_node)
9:         Put child_node at end of search_list
10:      end if
11:    end for
12:    for (each parent_node of cur_node) do
13:      if (number of child nodes with non-zero restorability values has increased
           since the last visit) then
14:        CalculateBackward(parent_node)
15:        Put parent_node at end of search_list
16:      end if
17:    end for
18:  end while
19:  Record restorability values of all nodes in the circuit for current signal selection
20: end while
21: cur_high_value = 0.9
22: while (cur_width < TB_width) do
23:   chosen_signal = FindMaxCover(cur_high_value)
24:   if (chosen_signal == NULL) then
25:     cur_high_value = cur_high_value - 0.1
26:   else
27:     Put chosen_signal in signal_selection_list
28:     cur_width++
29:   end if
30: end while
31: return signal_selection_list

```

(line 8) and backward justification (line 14) respectively in the same way as Algorithm 5.1 propagates the restorability metric among circuit nodes for calculating the restorability values of the circuit when selecting the first signal. However, instead of using a custom parameter, another mean is provided to control the amount of metric propagation among circuit nodes to limit computation time. Whenever a node is being visited, the algorithm will only proceed to update the restorability value of the current node through forward propagation when the number of parent nodes that have non-zero restorability values has been increased since the last time the current node has been visited (line 7). Likewise, the restorability values of its child nodes will be checked to determine if backward justification will be performed (line 13). These checks ensure that a node will only be revisited if the changes of the restorability value come from a different circuit path when compared to its previous visit.

Algorithm 5.2 selects trace signals based on how many other signals will likely be *covered* after state restoration. Using the restorability values in Figure 5.2(b) as an example, when selecting *FFC* as the trace signal, *FFD* will also be covered since data for *FFD* can be fully reconstructed during state restoration as indicated with a restorability value of 1 in *FFD*. As a result, although both *FFC* and *FFD* will likely help reconstruct data for other signals if any of them is selected as a trace signal, the two signals together will not help reconstruct more data if they are both selected as trace signals for the circuit shown in Figure 4.1(a). This is why Algorithm 5.2 analyzes how many signals will be covered at line 23 and will greedily select trace signals so that the maximum amount of signals can be covered in order to achieve high restoration ratio.

It should be noted that it is not necessary for the restorability value of a signal to be 1 before it can be classified as covered. For example, *FFD* is the only signal that will achieve a restorability value of 1 when *FFC* is selected as the trace signal as shown in Figure 5.2(b). In this case, Algorithm 5.2 will gradually decrease the requirement by lowering the targeted restorability value used for classifying signal coverage when no signal can be found to cover additional signals (25).

This technique on selecting trace signals based on signal coverage in Algorithm 5.2 differs from the incremental trace signal selection technique by re-calculating restorability values of the circuit with each additional signal selection in Algorithm 5.1. As will be shown by the experimental results, although the signal selection method in Algorithm 5.1 may be able to select trace signals that yield higher restoration ratio, the computation time for calculating restorability values repeatedly for each incremental signal selection can become prohibitively high with large circuits. On the other hand, by selecting trace signals that give high coverage for restoring data in other signals without re-evaluating the restorability values of the circuit, a good restoration ratio can still be achieved, while the computation time for Algorithm 5.2 falls into an acceptable range.

5.2 Trace signal selection using topology and logic gate behavior

To further refine the equations for calculating the restorability of a node, the logic behavior of each gate can be taken into consideration. The new metric that considers both the topology and the logic behavior of a gate is given in Figure 5.3. In the new equations, the restorability is further divided into $F0$ and $F1$, which represent the likelihood of restoring a logic 0 and 1 respectively on the output of a gate using forward operations. Likewise, $B0$ and $B1$ depict the chance of restoring a logic 0 and 1 respectively on the input of a gate using backward operations. Note that the equations for the *AND* gate, the *OR* gate and the *XOR* gate are different as shown in Figure 5.3 due to their different logic behaviors. For the *AND* gate, when any of the inputs is logic 0, the output can be concluded as logic 0, whereas to restore a logic 1 on the output through forward operations, both inputs have to be logic 1. On the other hand, an input of an *AND* gate can be justified as logic 1 whenever the output is also logic 1. In order to evaluate the input of an *AND* gate to logic 0, the output of the gate has to be logic 0, and other inputs have to be logic 1. The equations for other primitive gates are derived using similar reasoning. It should be

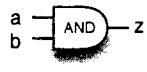
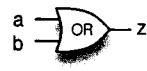

	$F0(z) = \max\{F0(a), F0(b)\}$ $F1(z) = (F1(a) + F1(b))/2$ $B0(a) = (\max\{B0(z)\} + F1(b))/2$ $B1(a) = \max\{B1(z)\}$
	$F0(z) = (F0(a) + F0(b))/2$ $F1(z) = \max\{F1(a), F1(b)\}$ $B0(a) = \max\{B0(z)\}$ $B1(a) = (\max\{B1(z)\} + F0(b))/2$
	$F0(z) = \frac{(F0(a) + F0(b)) + (F1(a) + F1(b))}{4}$ $F1(z) = \frac{(F0(a) + F1(b)) + (F1(a) + F0(b))}{4}$ $B0(a) = \frac{(\max\{B0(z)\} + F0(b)) + (\max\{B1(z)\} + F1(b))}{4}$ $B1(a) = \frac{(\max\{B0(z)\} + F1(b)) + (\max\{B1(z)\} + F0(b))}{4}$

Figure 5.3: Equations for restorability calculation using topology + logic

noted that although both Algorithms 5.1 and 5.2 can be used to select trace signals with both of the proposed metrics, the CPU runtime increases when choosing signals using the second metric. This is due to the fact that the restorability equations are more complex for each node, due to the increased accuracy of the metric.

Figure 5.4 shows the restorability values calculated using the equations from Figure 5.3 for all the FFs in the circuit from Figure 4.1(a) for three iterations. It can be seen from Figures 5.2 and 5.4 that *FFC* should be chosen as the trace signal, since in both cases it covers more signals with higher restorability values in the circuit. Note the differences between restorability values in other signals in Figures 5.2 and 5.4 when using the two proposed metrics for signal selection. This is due to the further refinement in the equations that consider topology and logic behavior of the circuit for restorability calculation. When working with large circuits, this refinement can help better evaluate the signal coverage when choosing the trace signals.

FFA		FFB		FFC	
F0 [0, 0, 0.5]	F0 [0, 0.5, 0.75]	F0 [0, 0.5, 0.75]	F0 [0, 0.5, 0.75]	Circuit restorability after three iterations if FFE is traced $= (0.5+0.5+0.5+0.5)$ $+(0.75+0.75+0.75+0.75)$ $+(0.75+0.75+0.75+0.75)$ $+(0.5+0.5+0.5+0.5)$ $+(1+1+1+1)$ $= 14$	
F1 [0, 0, 0.5]	F1 [0, 0.5, 0.75]	F1 [0, 0.5, 0.75]	F1 [0, 0.5, 0.75]		
B0 [0, 0, 0.5]	B0 [0, 0.5, 0.75]	B0 [0, 0.5, 0.75]	B0 [0, 0.5, 0.75]		
B1 [0, 0, 0.5]	B1 [0, 0.5, 0.75]	B1 [0, 0.5, 0.75]	B1 [0, 0.5, 0.75]		
FFD		FFE			
F0 [0, 0, 0.5]	F0 [1, 1, 1]				
F1 [0, 0, 0.5]	F1 [1, 1, 1]				
B0 [0, 0, 0.5]	B0 [1, 1, 1]				
B1 [0, 0, 0.5]	B1 [1, 1, 1]				

(a) Circuit restorability when FFE is selected

FFA		FFB		FFC		
F0 [0, 0.5, 1]	F0 [0, 1, 1]	F0 [1, 1, 1]	Circuit restorability after three iterations if FFC is traced $= (1+1+1+1)$ $+(1+1+1+1)$ $+(1+1+1+1)$ $+(1+1+1+1)$ $+(0.875+0.875+0.875+0.875)$ 5×0.875 $= 19.5$			
F1 [0, 1, 1]	F1 [0, 0.5, 1]	F1 [1, 1, 1]				
B0 [0, 0.5, 1]	B0 [0, 1, 1]	B0 [1, 1, 1]				
B1 [0, 1, 1]	B1 [0, 0.5, 1]	B1 [1, 1, 1]				
FFD		FFE				
F0 [0, 1, 1]	F0 [0, 0.5, 0.875]					
F1 [0, 1, 1]	F1 [0, 0.5, 0.875]					
B0 [0, 1, 1]	B0 [0, 0.5, 0.875]					
B1 [0, 1, 1]	B1 [0, 0.5, 0.875]					

(b) Circuit restorability when FFC is selected

Figure 5.4: Restorability calculation for three iterations using the topology + logic metric

One may argue that the proposed restorability equations that consider both topology and behavior of logic gates (shown in Figure 5.3) resemble the SCOAP controllability/observability concept that guides the ATPG process for manufacturing test [35]. There is, however, a fundamental difference between the proposed restorability metrics and the SCOAP metrics. SCOAP captures the controllability/observability of the internal circuit nodes by measuring how the assignments on neighboring signals will affect the targeted signal. Thus, the controllability/observability of a node does not give any information on how the specific value obtained for the targeted node can help restore data for other circuit nodes. For instance, for the circuit in Figure 4.1(a), the SCOAP measure will identify FFE as a hard to control signal due to the

presence of the *XOR* gate. However, tracing only *FFE* will not be able to restore data for any other signals in the circuit. Unlike SCOAP, the proposed restorability metrics capture how the data acquired on a node during a debug session will give more information about the internal state of the circuit.

5.3 Experimental results

Based on the same reasonings on the expected logic block size for the experimental setup discussed in Section 4.3, the experiments for evaluating the trace signal selection algorithms are also performed on the three largest ISCAS89 benchmark circuits [18] (i.e. s38584, s38417 and s35932). For these experiments, the state restoration algorithm detailed in the previous chapter is used to obtain the restoration ratios for evaluating the two trace signal selection algorithms. These algorithms are implemented using ANSI C and executed on a PC with dual-Xeon processors at 2.4 GHz with 1 GB of RAM. Also, in our experiments, all the high fan-in gates are decomposed into two-input logic gates when translating the ISCAS circuits into circuit graphs for both the state restoration algorithm, and the trace signal selection algorithm.

There are two sets of experimental results discussed in this chapter of the thesis. The first set of results shown in SubSection 5.3.1 is based on randomly generated stimuli on all primary inputs of the circuits using the experimental flow shown in Figure 4.3. While the second set of results discussed in SubSection 5.3.2 is based on randomly generated stimuli on the data inputs, and the values on the control inputs are constrained as explained in the experimental flow given in Figure 4.5.

5.3.1 Experiments with randomly generated stimuli on all inputs

Tables 5.1, 5.2 and 5.3 give the state restoration ratio and the restoration time for s38584, s38417 and s35932 respectively for comparing different signal selection methods using the topology only metric. Similarly, Tables 5.4, 5.5 and 5.6 give the same type of results for s38584, s38417 and s35932 respectively for comparing the signal

Table 5.1: State restoration results for s38584 when all primary inputs are driven randomly and trace signal are selected using the topology-only metric

Buffer		Algorithm 5.1			Algorithm 5.2		
depth	width	Select time (min)	Ratio	Restore time (sec)	Select time (min)	Ratio	Restore time (sec)
1024	8	221	3.51	0.00	11	83.48	9.00
	16	425	2.26	0.00	11	43.17	9.00
	32	828	1.82	0.00	11	21.89	9.00
4096	8	221	3.52	0.00	11	82.38	52.00
	16	425	2.26	0.00	11	43.72	52.40
	32	828	1.82	0.00	11	22.16	52.00
8192	8	221	3.52	0.00	11	84.79	134.30
	16	425	2.26	0.00	11	44.05	134.40
	32	828	1.82	0.00	11	22.33	133.20

Table 5.2: State restoration results for s38417 when all primary inputs are driven randomly and trace signal are selected using the topology-only metric

Buffer		Algorithm 5.1			Algorithm 5.2		
depth	width	Select time (min)	Ratio	Restore time (sec)	Select time (min)	Ratio	Restore time (sec)
1024	8	294	16.75	0.00	10	16.34	0.00
	16	604	8.48	0.00	10	8.17	0.00
	32	1227	6.73	0.00	10	4.10	0.00
4096	8	294	16.75	0.00	10	16.33	0.00
	16	604	8.54	3.20	10	8.17	0.00
	32	1227	6.72	7.40	10	4.09	0.00
8192	8	294	16.75	0.00	10	16.33	0.00
	16	604	8.59	8.80	10	8.17	0.00
	32	1227	6.81	4.20	10	4.08	0.00

Table 5.3: State restoration results for s35932 when all primary inputs are driven randomly and trace signal are selected using the topology-only metric

Buffer		Algorithm 5.1			Algorithm 5.2		
depth	width	Select time (min)	Ratio	Restore time (sec)	Select time (min)	Ratio	Restore time (sec)
1024	8	266	29.31	0.00	29	216.41	0.00
	16	598	86.58	0.80	29	108.29	0.00
	32	1255	44.05	0.00	29	54.23	0.00
4096	8	266	31.87	2.00	29	215.86	1.00
	16	598	89.06	2.00	29	108.01	1.00
	32	1255	44.98	1.40	29	54.09	1.00
8192	8	266	32.31	4.00	29	216.19	2.00
	16	598	90.82	4.60	29	108.17	2.00
	32	1255	45.79	3.60	29	54.17	2.00

Table 5.4: State restoration results for s38584 when all primary inputs are driven randomly and trace signal are selected using the topology+logic metric

Buffer		Algorithm 5.1			Algorithm 5.2		
depth	width	Select time (min)	Ratio	Restore time (sec)	Select time (min)	Ratio	Restore time (sec)
1024	8	574	124.34	18.00	60	85.43	6.60
	16	1225	64.08	21.00	60	42.95	6.60
	32	2493	36.59	28.80	60	23.57	7.40
4096	8	574	126.92	125.80	60	86.52	39.20
	16	1225	65.43	164.60	60	43.50	39.20
	32	2493	37.27	255.20	60	23.90	43.80
8192	8	574	127.20	345.80	60	87.18	101.20
	16	1225	65.57	470.00	60	43.83	101.20
	32	2493	37.36	834.40	60	24.08	114.80

Table 5.5: State restoration results for s38417 when all primary inputs are driven randomly and trace signal are selected using the topology+logic metric

Buffer		Algorithm 5.1			Algorithm 5.2		
depth	width	Select time (min)	Ratio	Restore time (sec)	Select time (min)	Ratio	Restore time (sec)
1024	8	470	19.67	0.00	19	17.15	0.00
	16	1151	12.16	0.80	19	9.01	0.00
	32	2499	7.00	0.80	19	6.60	0.00
4096	8	470	19.62	1.00	19	17.15	0.00
	16	1151	11.22	10.60	19	9.01	0.00
	32	2499	6.73	10.60	19	6.61	0.00
8192	8	470	19.64	2.00	19	17.14	0.20
	16	1151	10.01	2.00	19	9.01	0.20
	32	2499	6.40	2.00	19	6.59	1.00

Table 5.6: State restoration results for s35932 when all primary inputs are driven randomly and trace signal are selected using the topology+logic metric

Buffer		Algorithm 5.1			Algorithm 5.2		
depth	width	Select time (min)	Ratio	Restore time (sec)	Select time (min)	Ratio	Restore time (sec)
1024	8	524	255.11	9.00	53	255.13	6.80
	16	1145	127.89	8.60	53	127.56	7.00
	32	2380	64.64	8.60	53	63.77	7.00
4096	8	524	254.90	52.80	53	254.91	45.60
	16	1145	127.80	52.40	53	127.45	46.40
	32	2380	64.59	50.80	53	63.72	46.00
8192	8	524	254.85	132.20	53	254.85	115.60
	16	1145	127.77	130.40	53	127.43	116.40
	32	2380	64.58	126.40	53	63.71	116.80

selection methods using the topology+logic metric. For the results generated using Algorithm 5.1, a threshold parameter of 0.1 is used. In these tables, the restoration ratios are obtained by comparing the total number of restored values versus the number of trace signals allowed on-chip. It is obvious that with lower number of trace signals, more data will need to be restored. The experiments are carried out for trace buffer depths of 1k, 4k and 8k, and trace buffer widths of 8, 16 and 32.

The reported results in Tables 5.1, 5.2, 5.3, 5.4, 5.5 and 5.6 stand for five different sets of debug data obtained by simulating the randomly generated data on all primary inputs. Although using random data on the primary inputs for obtaining the debug data on the trace signals in the experiments is sufficient for evaluating the proposed metrics and algorithms for trace signal selection, one might argue that during an actual debug experiment, the global control signals of a circuit will not be driven randomly. For example, a global reset signal that is active low should be driven by a zero for a few clock cycles for bringing the CUD into a known state. After that, a constant 1 should be asserted to the reset signal for the remaining part of the debug experiment. This is why in the next subsection, the experimental results with constrained values on the control inputs will be discussed.

There are several important points to be noted. First, using any of the proposed metrics for trace signal selection gives higher restoration ratios when compared with the results based on random signal selection. Secondly, using the metric that considers both circuit topology and logic behavior (Figure 5.3) helps Algorithms 5.1 and 5.2 to select trace signals that give higher restoration ratio when compared with the less sophisticated metric that considers only circuit topology (Figure 5.1), except when selecting 16 trace signals for s38584 using Algorithm 5.2. This is because when selecting 16 signals for s38584 using either of the proposed metrics, the differences in signal coverage between each candidate signals are very small. In this case, the algorithm may conclude that the coverage from each candidate signal is virtually the same, and thus, the greedy nature of the algorithm will not help select the proper trace signals. On the other hand, when the topology and logic behavior-based metric yields higher variation for the calculated restorability values (as in the case of s35932), Algorithm 5.2 will be able to identify the high coverage signals, and thus select a set

of trace signals that will yield high restoration ratios. Another interesting point that should be noted is that with s38584, even though when restoration ratios are lower, which indicate less data to be restored, the restoration times are actually longer. This is due to the same reason discussed above on the variations in restoration time in Table 4.4 from the previous chapter of the thesis that if the chosen trace signals reside in sequential loop, it may require data to be reconstructed one step at a time.

It is also interesting to note that increasing the trace buffer depth for s35932 does not help improve the restoration ratios due to the low sequential depth of the circuit. This is unlike s38584 where the sequential depth is larger and it is visible from the results that as the trace buffer depth increases, better ratios are achieved for the same trace buffer widths. This obviously comes at the expense of higher restoration time which is still within an acceptable range of a few minutes. Another factor that significantly contributes to high restoration ratio is the presence of large fan-ins, as in the case of s35932. When using the metric that only considers the topology of a circuit for signal selection, one can already improve the restoration ratios when compared to the random signal selection. Another interesting point is that when the trace buffer width is increased, more data will be restored. However, at a lower rate than the increase in the number of trace signals. One notable exception is s35932 when increasing the number of trace signals from 8 to 16 and driving the global control inputs deterministically.

When comparing the restoration ratios between the trace signal selection algorithms, it can be seen that the results from Algorithm 5.1 are better when the metric that considers both the circuit topology and logic behavior is used. This is because when setting the threshold parameter to 0.1, Algorithm 5.1 spends more effort on recalculating the restorability values whenever a new trace signal is selected. This gives a more accurate evaluation on how much data may be restored with the chosen trace signals. However, this increased accuracy comes with two limitations. Firstly, when the less accurate metric that considers only the circuit topology is used for signal selection, Algorithm 5.1 does not perform well against Algorithm 5.2 for s38584 and s35932. This is because when combining the monotonically increasing nature of the

metric and the high computation effort, Algorithm 5.1 may over-evaluate the restorability values for the signals placed in sequential loops. On the other hand, Algorithm 5.2 only calculates the restorability values once; thus it is not prone to inflating the restorability values in sequential loops. The second limitation of Algorithm 5.1 is the prohibitively high computation time, which is in the range of tens of hours for s35932 when selecting 32 trace signals. This is significantly larger than Algorithm 5.2 that only calculates restorability values once, and then selects trace signals by evaluating how many signals will be covered during state restoration. When Algorithm 5.2 is used to select 32 trace signals, the runtime is reduced to within one hour. Another interesting note when comparing runtime for the two trace signal selection algorithms is that the runtime scales proportionally with the number of trace signals when using Algorithm 5.1, while the runtime stays the same for Algorithm 5.2. This is because Algorithm 5.1 selects signals incrementally by re-evaluating the restorability metrics during each selection, while Algorithm 5.2 only calculates the restorability values once no matter how many trace signals are being selected.

One last important point to discuss is the runtime for the state restoration algorithm. Unlike the trace signal selection algorithm, which is run only once during implementation, the state restoration algorithm is run repeatedly after each debug session, as illustrated in Figure 2.6. Thus, it is important for the state restoration algorithm to be compute-efficient. As shown in Tables 5.1, 5.2, 5.3, 5.4, 5.5 and 5.6, the runtime for this algorithm is in the range of a few seconds to minutes. These results have been obtained using the bitwise parallelism exploited by the technique from Section 4.2. Note, the state restoration runtimes are greater than the ones observed for the same trace buffer capacity in Tables 4.3, 4.4 and 4.5. However, this increase in runtime is due to larger restoration ratios caused by the improved choice of trace signals, where the random trace signals selection has been replaced by the deterministic Algorithms 5.1 and 5.2. Having more useful debug data extracted from the circuit obviously causes the state restoration algorithm to process more circuit nodes over the same number of clock cycles. Nonetheless, despite this increase, the runtime is practical and the state restoration algorithm fits seamlessly between the data acquisition step and the data analysis step.

Table 5.7: State restoration results for s38584 when the control inputs are driven deterministically and trace signal are selected using the topology-only metric

Buffer		Algorithm 5.1			Algorithm 5.2		
depth	width	Select time (min)	Ratio	Restore time (sec)	Select time (min)	Ratio	Restore time (sec)
1024	8	221	2.98	0.00	11	19.00	0.00
	16	425	1.87	0.00	11	10.57	0.00
	32	828	1.32	0.00	11	6.29	0.00
4096	8	221	2.99	0.00	11	19.00	0.00
	16	425	1.87	0.00	11	10.57	0.00
	32	828	1.32	0.00	11	6.29	0.00
8192	8	221	2.98	0.00	11	19.00	0.00
	16	425	1.87	0.00	11	10.57	0.00
	32	828	1.32	0.00	11	6.29	0.00

5.3.2 Experiments with constrained generated stimuli for control inputs

Tables 5.7, 5.8, 5.9 and 5.10 compare the results that were generated using Algorithms 5.1 and 5.2 for s38584 and s35932 with the experimental flow described in Figure 4.5. Note that no results for s38417 have been reported because as mentioned in SubSection 4.3.2, there are no signals need to be constrained in this circuit. While Tables 5.7 and 5.8 show the results when trace signal selection is performed using the metric that concerns only about circuit topology, Tables 5.9 and 5.10 give the results for when the metric that considers both circuit topology and logic behaviors of logic gates is used. In these tables, the results obtained using Algorithm 5.1 are performed with a threshold parameter of 0.1. The results on trace signal selection time in these experiments are the same as the ones shown in the previous experiments. This is because no changes have been made to the trace signal selection algorithm with these new experiments.

The restoration ratios obtained using the new experimental flow are lower than what was provided in SubSection 5.3.1. This is due to the same reasons discussed in

Table 5.8: State restoration results for s35932 when the control inputs are driven deterministically and trace signal are selected using the topology-only metric

Buffer		Algorithm 5.1			Algorithm 5.2		
depth	width	Select time (min)	Ratio	Restore time (sec)	Select time (min)	Ratio	Restore time (sec)
1024	8	266	6.68	0.00	29	2.08	0.00
	16	598	8.39	0.00	29	2.05	0.00
	32	1255	9.24	0.00	29	3.15	0.00
4096	8	266	6.76	0.00	29	2.08	0.00
	16	598	8.48	0.25	29	2.05	0.00
	32	1255	9.35	0.50	29	3.17	0.00
8192	8	266	6.77	0.25	29	2.08	0.00
	16	598	8.50	0.50	29	2.05	0.00
	32	1255	9.37	1.00	29	3.17	0.25

Table 5.9: State restoration results for s38584 when the control inputs are driven deterministically and trace signal are selected using the topology+logic metric

Buffer		Algorithm 5.1			Algorithm 5.2		
depth	width	Select time (min)	Ratio	Restore time (sec)	Select time (min)	Ratio	Restore time (sec)
1024	8	574	10.13	0.00	60	3.13	0.00
	16	1225	5.75	0.00	60	2.06	0.00
	32	2493	6.32	0.00	60	1.53	0.00
4096	8	574	10.13	0.00	60	3.13	0.00
	16	1225	5.75	0.00	60	2.06	0.00
	32	2493	6.32	0.00	60	1.53	0.00
8192	8	574	10.13	0.00	60	3.13	0.00
	16	1225	5.75	0.00	60	2.06	0.00
	32	2493	6.32	0.00	60	1.53	0.00

Table 5.10: State restoration results for s35932 when the control inputs are driven deterministically and trace signal are selected using the topology+logic metric

Buffer		Algorithm 5.1			Algorithm 5.2		
depth	width	Select time (min)	Ratio	Restore time (sec)	Select time (min)	Ratio	Restore time (sec)
1024	8	524	41.66	0.25	53	40.39	0.25
	16	1145	39.48	0.25	53	36.69	0.50
	32	2380	24.79	0.25	53	18.35	0.50
4096	8	524	41.45	0.25	53	40.38	0.25
	16	1145	39.31	0.75	53	36.69	0.50
	32	2380	24.76	1.00	53	18.34	0.50
8192	8	524	41.41	0.75	53	40.38	0.70
	16	1145	39.28	1.50	53	36.69	1.00
	32	2380	24.75	2.00	53	18.34	1.00

SubSection 4.3.2 that when the experiments are run using randomly generated stimuli for all primary inputs, no constraints have been provided on the reset signals for both s38584 and s35932. Thus, the state restoration algorithm will be able to reconstruct the values of all the circuits' nodes easily every time the circuits are reset.

It is also noted that the restoration ratios for both circuits is lower when Algorithm 5.2 is used with deterministically driven control inputs. This is because when compared with Algorithm 5.1, which selects trace signals by propagating the metrics thorough the logic gates, Algorithm 5.2 only chooses trace signals based on signal coverage. As a result, it is more likely for Algorithm 5.2 to select control signals as the trace signals since they usually drive more signals in a circuit. Since some of these control signals are now driven deterministically to their non-controlling values, the sampled data will not help reconstruct data for other signals. However, when the trace signals are selected using the topology-only metric for s38584 as shown in Table 5.7, Algorithm 5.2 gives better restoration ratios than Algorithm 5.1. This is because when the less accurate metric is used, Algorithm 5.1 tends to over-evaluate the restorability values when sequential loops are presented in s38584.

Another observation on the results from the new experimental flow is that the results state restoration time are very small when compared to that in the results from the previous experimental flow. This is due to the lower amount of data that can be restored when the control inputs are constrained.

We do recognize that there is no practical value of this new experimental flow when debugging circuits on an application board. However, we want to emphasize that during the design phase, it is essential to decide which signals are to be traced during post-silicon validation. Therefore, in addition to aiding researchers with benchmarking their algorithms under practical conditions, setting up an experimental flow can also better assist the decision-making process on which trace signals should be hardwired to the trace buffers. As a result, the flow is designed in such way that it relies on third party tools and simple algorithms with small runtime; thus fitting seamlessly in the current implementation flows for digital integrated circuits.

5.4 Summary

Chip designers traditionally rely primarily on design knowledge and intuition to decide which signals to probe and how many state elements to observe. Because the design complexity will continue to increase, structured debug methods with automated support will become crucial for decreasing the length of the debug cycle during post-silicon validation. In this chapter of the thesis, we have presented two new metrics and two algorithms for automatically selecting the trace signals. These algorithms show how by consciously choosing only a small number of signals to be probed in real-time, the observability of the CUD can be improved through the algorithmic solutions on state restoration presented in the previous chapter of the thesis.

Chapter 6

Distributed embedded logic analysis

So far we have presented techniques that can be used to improve post-silicon validation for a single block (or logic core) in a complex design. In multi-core designs, distributed ELAs with multiple trigger units and trace buffers can be placed on-chip. This brings the new challenge on how to connect the various units together in such way that the limited storage space in the trace buffers can be used efficiently. Although there are existing techniques that detail how the trace buffers are connected for various industrial designs (as discussed in SubSection 2.3.5), they are mainly ad-hoc and use static connections to the trace buffers. As a result, when the number of available trace buffers is smaller than the number of simultaneous sample requests coming from different trigger units, information from some of the data sources will have to be dropped. This results in loss of debug information and hence reduced observability.

In this chapter of the thesis, we explore the consequences of two assumptions that we anticipate will become common in future complex SOCs. These assumptions are summarized in Section 6.1. Motivated by this, we propose a new methodology based on distributed ELAs in Section 6.2. The accompanying hardware is detailed in Section 6.3. Experimental results given in Section 6.4 shows that real-time observability can be improved using only small amounts of logic hardware while avoiding excessive storage on-chip. Finally, Section 6.5 summarizes the contributions from this chapter.

6.1 Motivation

Our work is motivated by the following two assumptions that we anticipate will become common in future SOCs.

Assumption 6.1: Growing number of cores in future SOCs

Recent research has shown that the number of cores in a design will continue to increase while incorporating even more distributed embedded memories, whose integration costs are decreasing [11, 116]. Thus, finding a way to automatically control the acquisition of bursts of high-bandwidth debug data on-chip in a user-defined and prioritized manner will be key to improving the real-time observability during post-silicon validation. This motivates us to investigate a scalable DFD architecture that better utilizes the available on-chip storage by automatically allocating the trace buffers to handle concurrent sample requests.

Assumption 6.2: Adoption of high-speed trace ports

It has been reported that high-speed trace ports are used for streaming data to external memories during software debug [10, 32, 71, 77, 80]. We expect this technology to gain a wider adoption with the proliferation of high-speed I/Os in future SOCs. As we will show later, the ability to offload debug data while maintaining the real-time execution can further enhance real-time observability.

Based on these two assumptions, we explore how a new debug methodology can be developed. This methodology should improve real-time observability during post-silicon validation for multi-core SOCs that utilize distributed ELAs. Also, the amount of accompanying hardware for the architecture in this methodology should be small.

6.2 Proposed design-for-debug methodology

Unlike pre-silicon verification, where any number of data sources in a design can be sampled at the same time in the testbench, the number of available trace buffers limits how many data sources can be captured simultaneously during post-silicon validation.

To exploit the technology trends captured in the assumptions stated in the previous section, in this section we discuss the challenges and proposed debug architecture that can help bridge the gap between pre-silicon verification and post-silicon validation.

6.2.1 Challenges

As described in Assumption 6.1, designs with an increasing number of cores are expected to come in the foreseeable future. For example, [116] demonstrated a microprocessor with 80 cores and is capable of achieving a peak performance of over one teraflop at an operating frequency of 4GHz. Meanwhile, [11] introduces the idea of manycore designs, which can contain in excess of 1000 heterogeneous cores in a design. To debug these complex designs, we expect that the role of distributed embedded logic analysis will be of increasing importance. By placing standalone trigger units in various parts of the design, concurrent executions in the cores can be monitored. Moreover, as data sources can be scattered in various locations, debugging such a design also requires a way to acquire a large amount of debug data from these data sources *at any given time*. To sustain this high bandwidth requirement, the trace buffers should be distributed across different corners of the design.

As stated in Assumption 6.2, the use of trace ports for streaming data off the chip has been explored for debugging microprocessors-based systems [10, 32, 71, 80]. As the adoption of high-speed I/Os gains wider acceptance, we investigate the use of these high-speed I/Os as trace ports to stream data off the chip during debug. Although it has been mentioned in [10] that the bandwidth of one such trace port can go up to 10 Gbit/s, this is still a fraction of the rate at which data is acquired (e.g., 128-bit samples @ 1.6 GHz [93]). However, it should be noted that with the advancement in the design of trigger units, debug engineers will be able to better control what data to acquire on-chip. Unless one would require data from all the data sources in every clock cycle, trace buffers will only be used for data acquisition in a fraction of the debug time. Thus, its content can be offloaded through the available trace ports, such that storage space can be reclaimed for future data acquisition.

Under the expectation of having a growing number of cores to be monitored and the ability to stream debug data off-chip using high-speed trace ports, we argue that the tasks of allocating the available trace buffers for data sampling from different data sources, as well as assigning trace ports to idle trace buffers, should be done automatically by intelligent control on-chip. The controller should be built in such a way that the following scenarios are considered.

Scenario 6.1: When there are multiple trigger events occurring simultaneously, how to choose trace buffers to sample data from different data sources?

Scenario 6.2: When some of the trace buffers are already occupied, is it necessary to re-allocate the trace buffers when a new trigger event from a different data source occurs?

Scenario 6.3: How to allocate trace buffers when the number of sample requests is more than the number of available trace buffers?

Scenario 6.4: How to allocate trace buffers for data sampling before knowing when trigger events from multiple data sources will happen?

Scenario 6.5: How to decide which trace buffers to offload first when multiple trace buffers are idle?

Scenario 6.6: How to balance the sampled data among trace buffers such that more trace buffers will have available space for fulfilling upcoming data acquisition requests?

Scenario 6.7: In the case when debug experiments are repeatable, can the controller be re-programmed to acquire different sets of debug data during each re-run of the experiment?

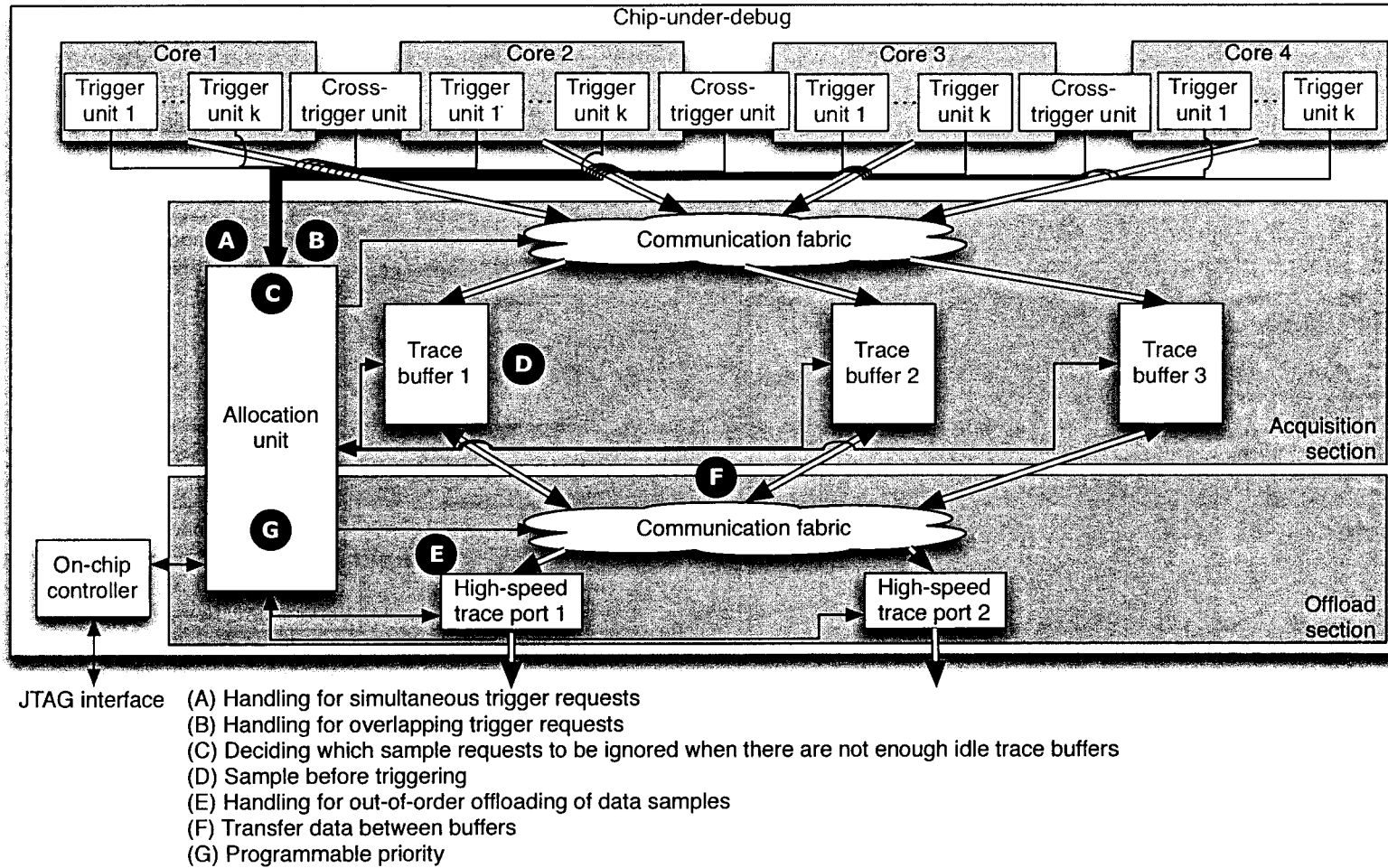


Figure 6.1: The proposed design-for-debug architecture based on distributed embedded logic analysis

6.2.2 Architecture

To address the various scenarios presented in the previous subsection, Figure 6.1 illustrates the proposed debug architecture with seven new features, discussed next. It is important to emphasize that these features are not meant to be seen as the only solutions for addressing the mentioned scenarios. Rather, they are given as examples on how the proposed architecture provides a hardware framework for tackling the previously discussed scenarios when employing distributed embedded logic analysis and high-speed trace ports in an SOC during post-silicon validation. Thus, it is up to the designers to further improve the hardware of each feature for their problems at hand.

Feature A: Handling simultaneous trigger requests

Feature B: Handling overlapping trigger requests

Feature C: Deciding which sample requests to be ignored when there are not enough idle trace buffers

Feature D: Sample before triggering for multiple trace buffers connected to multiple data sources

In this architecture, trigger units, which can be programmable trigger engines [3] or hardware assertions [17], are distributed within each core, while cross-trigger units can be setup among cores to monitor multiple trigger events across the design at the same time. To address Scenarios 6.1-6.4, we introduce *Features A – D* in the *Allocation unit* (new hardware circuits needed to make this *Allocation unit* efficient are given in the following subsection). When one or more specified trigger events occur, the allocation unit will be notified and will automatically decide where the debug data should be stored among the available trace buffers. Also, the decisions should be made according to a user-defined priority scheme that needs to reflect the importance of the debug data from various data sources.

Because the on-chip area used for debug is dominated by trace buffers, as will be discussed in the experimental results, it is essential to leverage their capacity across

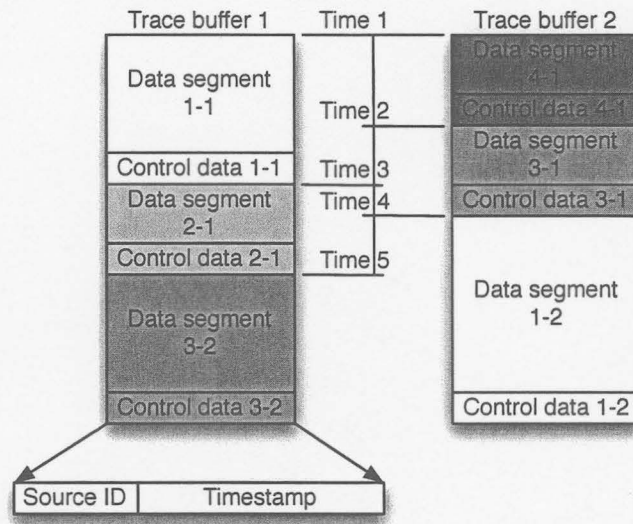


Figure 6.2: Sampled data organization in trace buffers using multiple segments

multiple data sources and reconfigure their usage *on-the-fly* based on the bandwidth requirements in different areas of the chip. Figure 6.2 shows how the data can be organized in the trace buffers. It should be noted that different amounts of debug data may be generated from different data sources each time they are triggered. To deal with this problem, one can insert control information such as *Source ID* and *Timestamp* for every data sample that is acquired. This is obviously inefficient, as it requires a huge amount of storage. Thus, we introduce the use of a *Control data* field that will be appended at the end of a sampling cycle such that the debug data can be grouped together into segments in the trace buffers. As the *communication fabric* can connect any data sources to the available trace buffers, data acquired from the same source at different times can be stored in separate trace buffers. This allows the allocation unit to better utilize the trace buffers, as discussed next.

Figure 6.2 uses two segments of data from *Source 1* (*Data segment 1-1* triggered at *Time 1*, and *Data segment 1-2* triggered at *Time 4*). In this example, when the request for capturing *Data segment 1-2* comes at *Time 4*, *Trace buffer 1* is already occupied by *Source 2*. In this case, the *Allocation unit* can allocate *Trace buffer 2* to

capture *Data segment 1-2*, while *Trace buffer 1* can be used to store *Data segment 3-2* starting at *Time 5*. As a result, using the same amount of storage space, the proposed architecture will be able to acquire more debug data *since no data will need to be dropped* and the load on multiple trace buffers is balanced.

Feature E: Handling for out-of-order offloading of data samples

When high speed trace ports are available, the allocation unit can utilize any idle trace ports to offload the sampled data from the trace buffers for off-chip analysis. As data is being offloaded, spaces in the trace buffers can be reused. To deal with Scenario 6.5, the same priority scheme defined in the *Allocation unit* can be used when allocating trace ports for data offload. Hence, *Feature E* is included in the proposed DFD architecture.

Feature F: Transfer data between buffers

The proposed DFD architecture also includes the hardware for supporting *Feature F* in Figure 6.1 in order to address Scenario 6.6. By allowing data to be transferred between idle trace buffers, the available storage space can be better balanced among trace buffers. This helps maintain the high bandwidth requirement when multiple data acquisition requests arise during post-silicon validation.

Feature G: Programmable priority

To deal with Scenario 6.7, the user should be able to modify the priority scheme for each debug experiment. This is because the priority scheme directly affects how the *Allocation unit* arranges the trace buffers when sample requests arise and data is offloaded through the trace ports. Thus, changing the way different data sources are prioritized enables the allocation unit to acquire distinct sets of data when the experiment is repeated. As a result, we incorporate *Feature G* in the *Allocation unit*.

The *Allocation unit* also provides the appropriate control signals to the *Communication fabric* that carries the data transport between data sources, trace buffers and trace ports. In our current implementation, the communication fabric uses a

pipelined multiplexer network. It should be noted however that the functionality of the communication fabric can also be provided by other means, such as a data bus, or a network-on-chip. Although this would incur more complex control from the allocation unit when configuring the communication fabric, the hardware for deciding how trace buffers are allocated for data acquisition in real-time would still be the same. As the number of data sources and available trace buffers increases, one may argue that providing a communication fabric (i.e., a pipelined multiplexer) that can allow any data sources from the different cores to be connected to any trace buffer can potentially lead to large area. As it will be shown in our experimental results, when compared with the area of the distributed trace buffers, the communication fabric only contributes to a small portion of the total area for the proposed architecture.

It is obvious that the implementation of the allocation unit closely impacts how the trace buffers would be selected to acquire data from multiple cores, while at the same time utilizing the available trace ports to stream debug data off the chip in an automated manner. As a result, in the following section, we detail new hardware circuits that should be incorporated into the *Allocation unit* for supporting *Features A–G*.

6.3 New on-chip hardware circuitry for enabling the proposed methodology

The *Allocation unit* presented in the proposed architecture in Figure 6.1 is the key to efficiently utilize the available trace buffers and trace ports for acquiring debug data in real-time during post-silicon validation for future core-based designs. In Figure 6.3, the implementation of the allocation unit is shown. It contains an *Allocation FSM* that gathers information from all the trace buffers and trace ports. Using this information, it makes real-time decisions on where debug data should be stored whenever sample requests come from the trigger units, as well as how the trace ports should be used to offload the captured debug data. After a decision has been made, it updates the status registers that control the read/write operations of the trace buffers, as well as providing the appropriate controls to the *Communication fabric* to facilitate

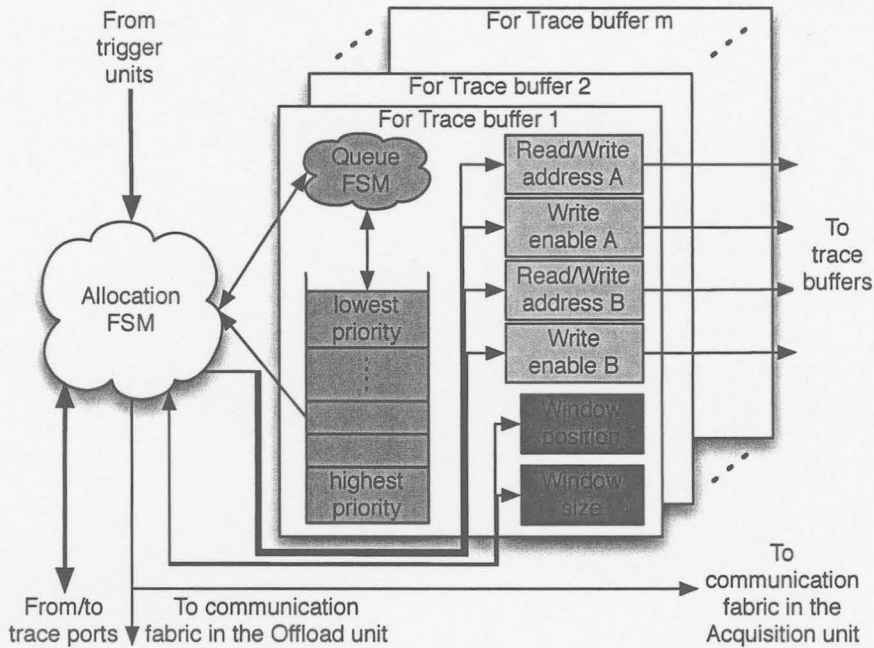


Figure 6.3: The allocation unit

data acquisition and offloading. In addition to the allocation FSM, extra hardware is introduced in the allocation unit to support the features that are discussed.

6.3.1 Handling of simultaneous, overlapping, and overflow sample requests (Features A-C)

In pre-silicon verification, any number of data sources can be monitored and displayed simultaneously during simulation. However, during post-silicon validation, when the number of sample requests exceeds the number of available trace buffers, it is imperative that some of these requests will have to be ignored and debug data will be lost. In this case, the designer will have to provide *priority information* for the data sources to the *Allocation FSM* so that only sample requests from low priority data sources should be ignored.

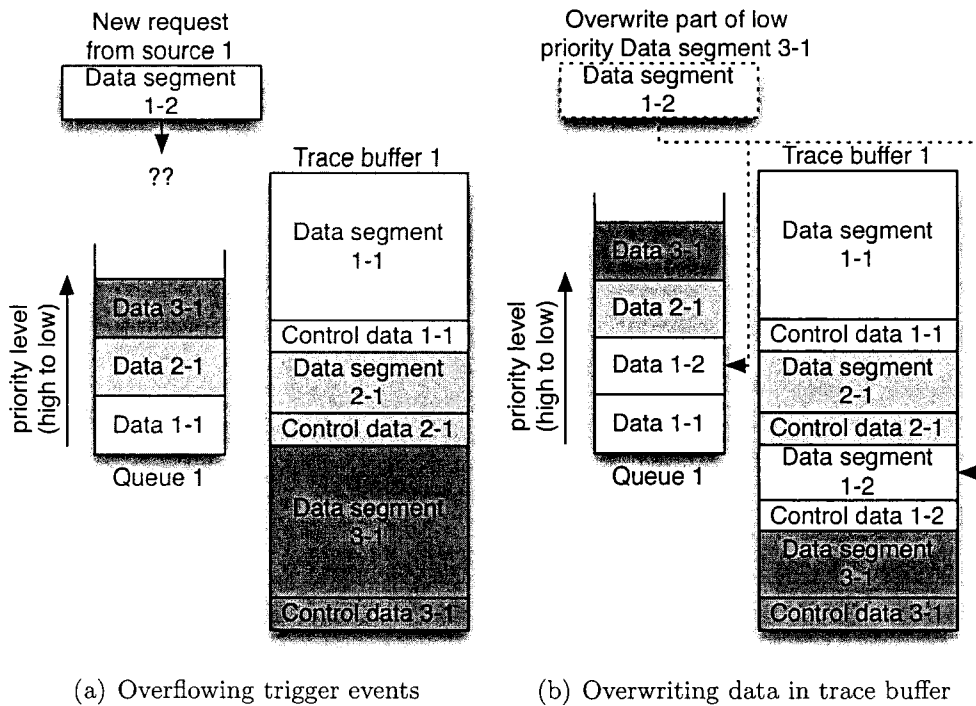


Figure 6.4: Example of overwriting low priority data during overflowing of trigger events

The limited storage spaces provided by the trace buffers also constrains debug data to be dropped when sample requests come after the trace buffers have been filled. However, instead of dropping all the sample requests when the buffers are full, it may be desirable to accept data from high priority data sources by overwriting low priority data in the trace buffers. To keep track of where the different segments of prioritized debug data are stored in the trace buffers, a queue, which is maintained by a *Queue FSM*, is introduced for each trace buffer in the allocation unit.

Figure 6.4 shows how the information from the queue can help the allocation unit decide how data should be overwritten in the trace buffers. It may seem that having a separate queue for each trace buffer requires significant area investment. However,

one should note that the length of the queues is small since only one entry will be created for each segment of data in the trace buffer. Also, for each entry in the queue, only simple information such as priority and the starting address of a data segment is stored. Thus, the queue can either be embedded in its corresponding trace buffers, or a dedicated embedded memory can be used to store the information for all the queues in the design.

It should be noted that in order to provide real-time observability during post-silicon validation, the allocation unit has to reach its decision on trace buffer assignment in the same clock cycle as the arrival of any sample requests. By using these queue FSMs to organize prioritized data in the trace buffers, the complexity of the allocation FSM can also be reduced since the queue FSMs can operate in the background to prepare all the necessary information (e.g., amount of available space in trace buffers, distribution of prioritized data among buffers) for the allocation unit.

6.3.2 Data sampling before trigger (Feature D)

Another powerful feature that will aid identifying root-cause of a bug is to acquire data that precedes the triggering of specific events. Although this capability to sample data before trigger events occur has been explored, this ability is only implemented for designs with only one trace buffer [6, 122]. When multiple data sources are connected with distributed trace buffers, it is not obvious how one should allocate the available trace buffers to allow a continuous sampling of data before the trigger event arrives.

To efficiently utilize the limited storage in the trace buffers, while supporting data sampling before trigger, two status registers called *Window size* and *Window position* are introduced in the allocation unit in Figure 6.3. In this case, continuous data sampling is only allowed within the allocated window in a trace buffer as shown in Figure 6.5(a). When the trigger event occurs, these sampled data are marked by appending a control data at the end of the segment, and the *Window position* register will be updated such that a different part of the buffer will be used (Figure 6.5(b)). This ability to reposition the sampling window among trace buffers provides further flexibility to the allocation unit to free up trace buffers from continuous data

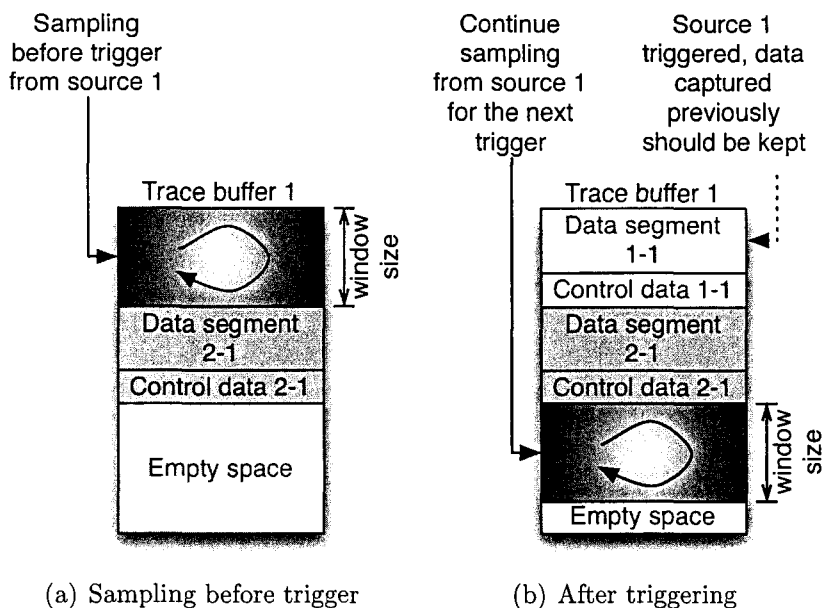


Figure 6.5: Example of using windows to support sampling before trigger

sampling and assign them to service upcoming data acquisition requests. It is obvious that the amount of sample-before-trigger data that can be acquired is limited by the size of the window. However, this can be changed during post-silicon validation by programming at runtime the *Window size* register.

6.3.3 Out-of-order data offloading (Feature E)

As previously mentioned, when high-speed trace ports are available, debug data can be offloaded from the trace buffers to regain valuable storage space for any upcoming sample requests. Figure 6.6 can be used to show how the information from the queue can be used to decide which segment of data should be offloaded. When data segments with different priorities are stored in the trace buffer, the queue FSM uses the priority information in the queue to inform the trace ports to offload all the high priority data segments before servicing the data segments with lower priorities. In the example

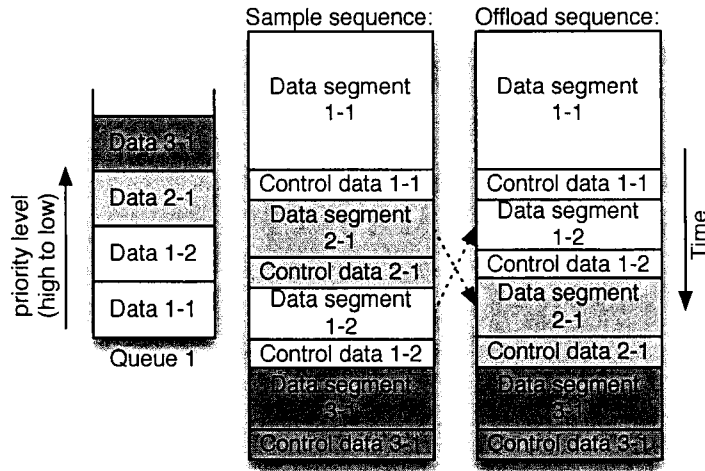


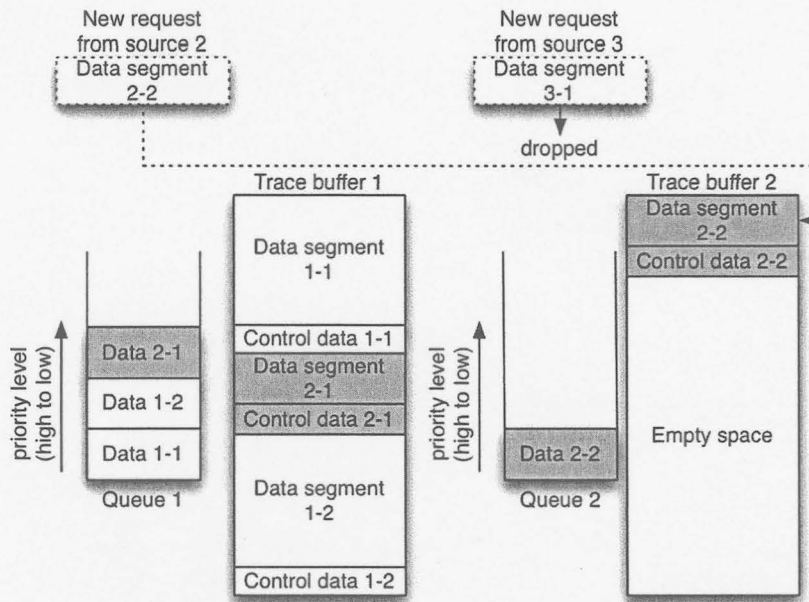
Figure 6.6: Out-of-order offloading

shown in Figure 6.6, *Data segment 1-2* is offloaded before *Data segment 2-1*, even though *Data segment 2-1* is acquired into the trace buffer before *Data segment 1-2*. This ensures that high priority data will be offloaded before being overwritten, and guarantees high observability to the data sources that are of significant importance.

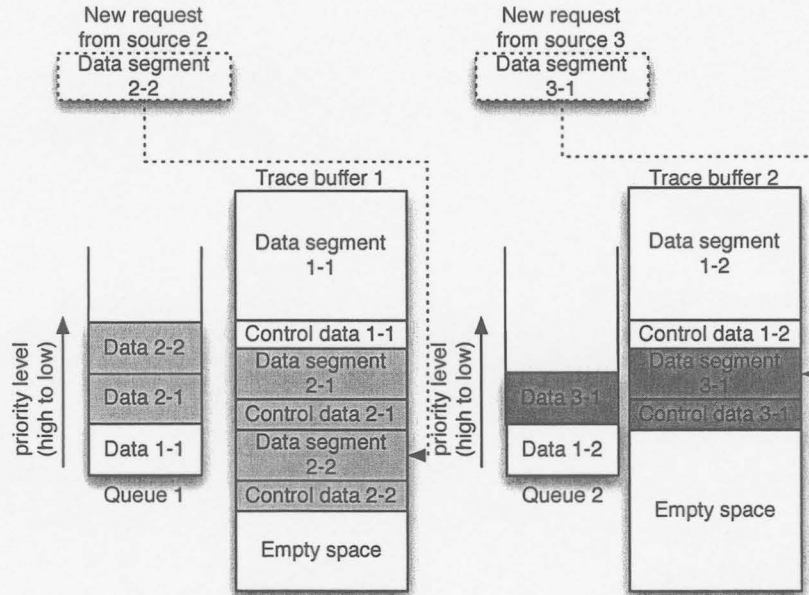
6.3.4 Data transfers between buffers (Feature F)

In the case when the trace buffers have more than one access port, one can chain the trace buffers together such that while one port is used to acquire debug data, the other port can be used to transfer the sampled data between trace buffers. This feature of being able to transfer data between trace buffers provides three benefits.

Firstly, by balancing the amount of stored data among trace buffers, empty spaces can be fairly distributed among trace buffers. This allows the allocation unit to allocate these partially empty trace buffers to service more simultaneous sample requests. This is shown in Figure 6.7. If the empty space is only concentrated in *Trace buffer 2*, when new sample requests from *Data source 2* and *Data source 3* come simultaneously, due to the low priority of *Data source 3*, its request will be dropped and only data from *Data source 2* will be stored as shown in Figure 6.7(a). However, if the empty space is fairly distributed among the buffers, as in Figure 6.7(b), the two



(a) Not enough bandwidth, resulting sample request being dropped



(b) Enough bandwidth to satisfy all sample requests

Figure 6.7: Example of how data transfer between buffers can maintain acquisition bandwidth

sample requests can be satisfied. Secondly, if the amount of debug data that will be generated from a data source is known, the allocation unit can move data between trace buffers so that sufficient space can be reserved for this data source in order to reduce data segmentation.

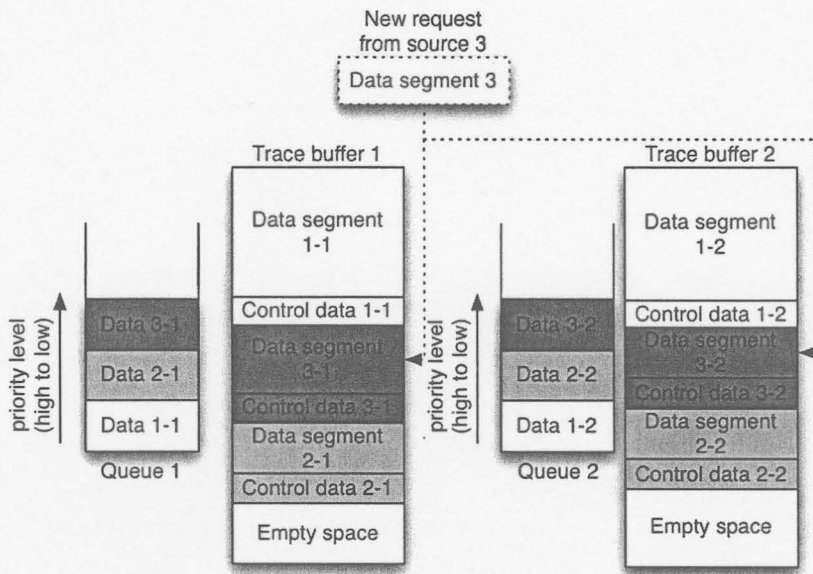
Moreover, when data can be offloaded in an out-of-order fashion using *Feature E*, transferring data between trace buffers helps de-fragment the empty space within each trace buffer. This can be shown using the example in Figure 6.8. When *Data segment 3* arrives, fragmented empty spaces among the trace buffers will force the new data to be segmented with their corresponding control data (*Data segment 3-1* and *Data segment 3-2* in Figure 6.8(a)). If the data is re-organized using this feature, the new data can be stored in just one data segment (*Data segment 3-1* in Figure 6.8(b)). Recall that control data will have to be appended to each segment of debug data, thus, de-fragmenting empty space can help reduce data segmentation, which in turns decreases the amount of control information being stored in the trace buffers.

To support this feature, the trace buffers can be daisy-chained together using the *Communication fabric* in the *Offload unit* in Figure 6.1. Moreover, additional control such as the preparation of read/write addresses and write enables to the appropriate trace buffers must be included in the trace buffer control unit as shown in Figure 6.3.

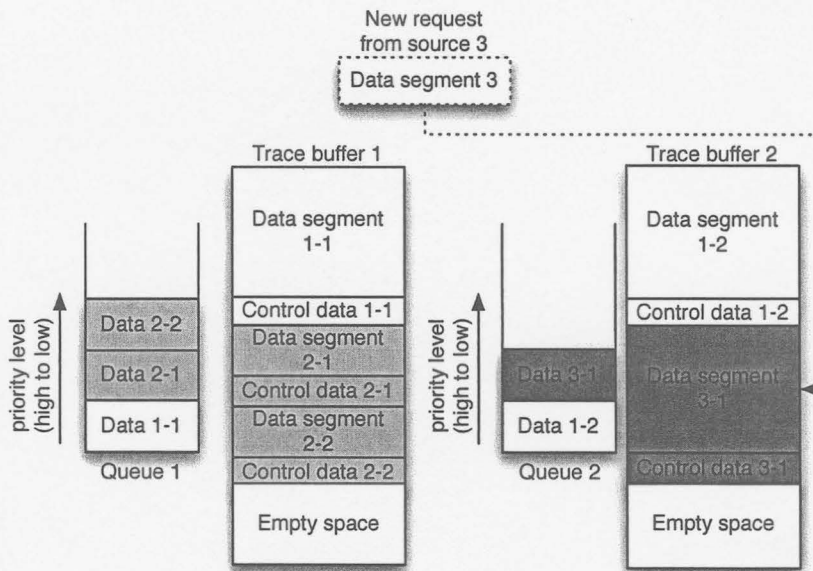
6.3.5 Programmable priority (Feature G)

In the case when the debug experiment is deterministic (i.e., the debug experiment can be regenerated with the same set of input stimuli), it may be beneficial to gather different sets of debug data each time the experiment is run. Thus, we introduce a programmable register for each data source and additional control logic in the allocation unit such that the user can re-arrange the priority scheme. When the new priority information is uploaded to the priority registers, the decision made by the allocation unit for *Features A – E* will be changed, resulting in a different set of debug data to be captured into the trace buffers.

Figure 6.9 shows how different priority settings will affect data acquisition. In Figure 6.9(a), *Data source 1* has the highest priority. When a new sample request from



(a) Segmented data



(b) Non-segmented data

Figure 6.8: Example of how data transfer between buffers can reduced data segmentation

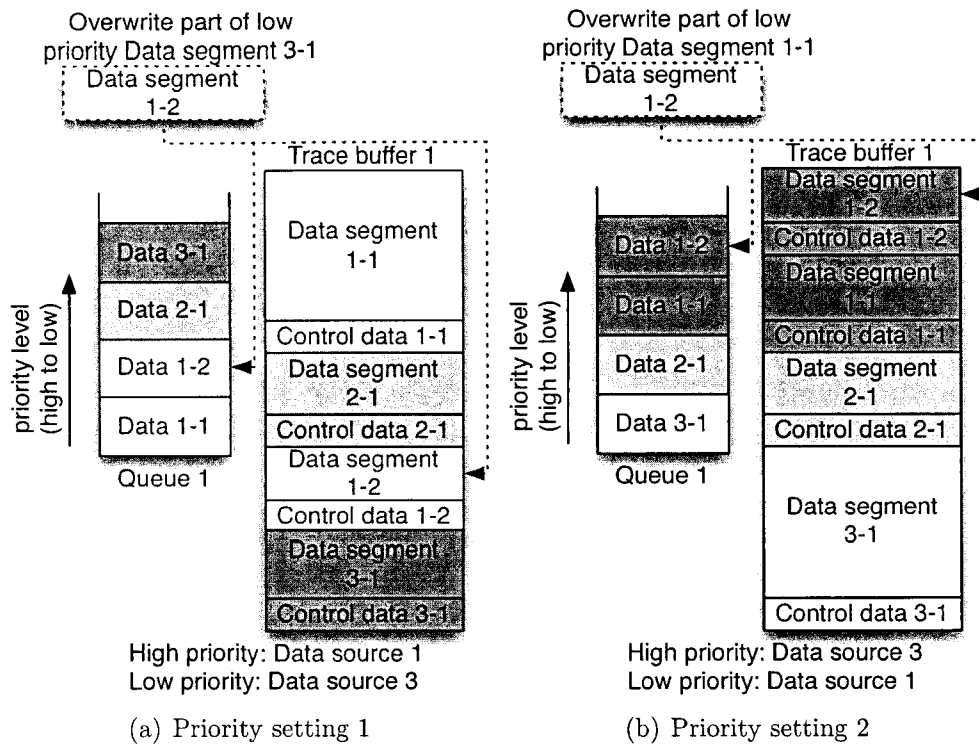


Figure 6.9: Example of data sampling with different priority settings

that data source arrives, the queue FSM will notify the allocation unit to overwrite part of the low priority *Data segment 3-1* with the new *Data segment 1-2*. On the other hand, when *Data source 3* is given the highest priority in Figure 6.9(b), part of *Data segment 1-1* will be overwritten instead.

One may argue that by only changing the priority setting, the amount of distinct debug data that will be gathered between different experiments may be very small. For example, when comparing the amount of distinct debug data obtained in Figure 6.9 between the two priority settings, only the small amount of *Data segment 3-1* that was overwritten with the first priority setting is recovered from the second run of the debug experiment. However, one should note that without programmable priority, it

will be impossible to retrieve this small set of *Data segment 3-1*, and thus, may lose valuable debug information during post-silicon validation.

6.4 Experimental results

Three types of experiments are covered in this section. First we present two case studies on a digital video decoder (shown in Figure 6.10) implemented at McMaster University, which illustrates how the discussed features were used in a practical environment in SubSection 6.4.1. Then, we investigate how different configurations of the proposed DFD architecture tradeoff between area investment and debug data acquisition in SubSections 6.4.2 and 6.4.3 respectively.

6.4.1 Case studies

Figure 6.10 shows the organization of the MPEG-2 audio/video decoder used in the case studies presented below.

Case Study 1: During digital video decoding, starvation of the audio or video decoder may occur during extended idle periods which are interrupted by bursts of activity across the 3 interacting clock domains of 25 MHz, 54 MHz and 27 MHz. While concurrent monitors in pre-silicon verification (for instance on the *Bitstream Parser*, *Video Shift Controller* and *Audio Shift Controller* from Figure 6.10) can indeed identify these corner cases, it can take extensive time for them to arise. Using the proposed architecture, we are first capable (by cross triggering) to identify which domain starves and with how much time before the next data would have arrived. Out-of-order offloading (Feature E) lets the user balance offload bandwidth against acquisition bandwidth (video will produce more samples than audio) and transferring data between trace buffers (Feature F) gives more capacity for capturing video samples, thus improving the real-time observability of the debug experiment. Finally, sample before trigger (Feature D) allows us to identify the sequence of events that lead up to the starvation, by which we can assess buffering requirements.

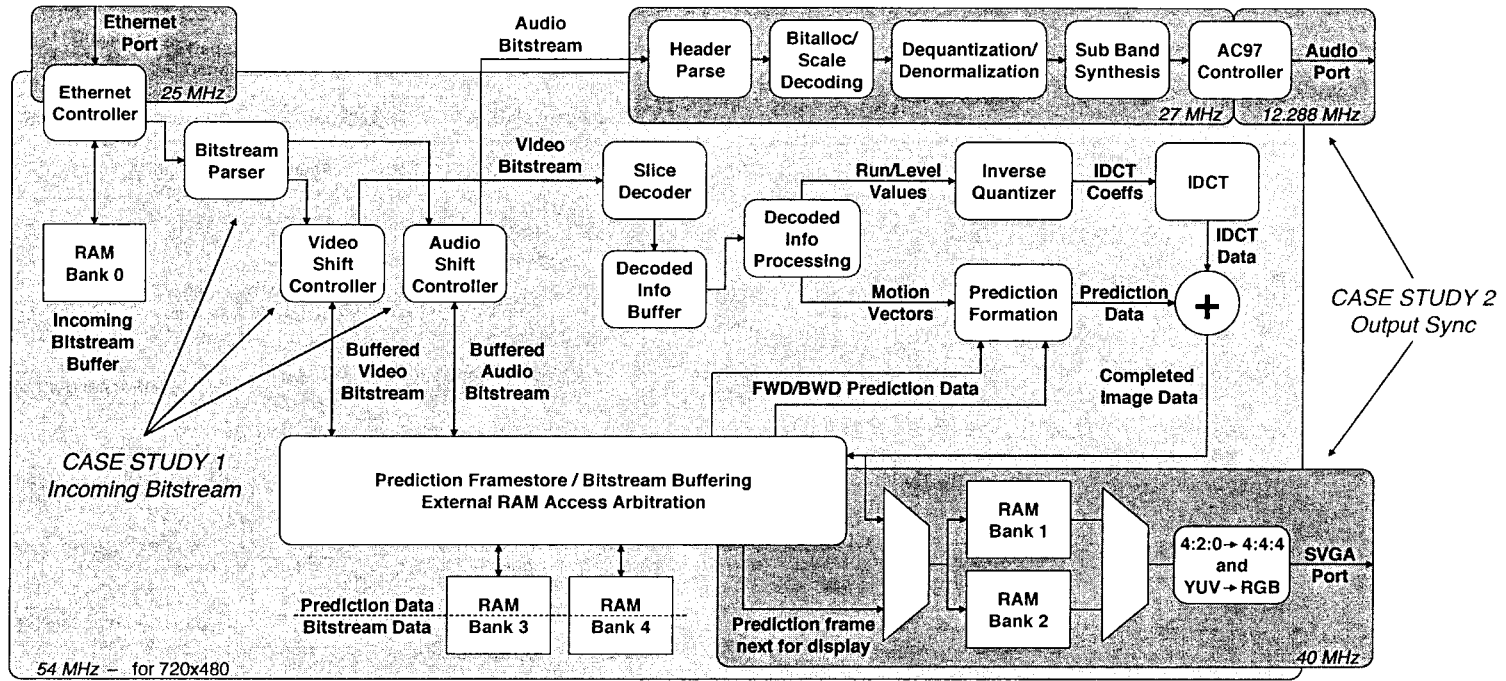


Figure 6.10: MPEG-2 decoder implementation for case studies

Case Study 2: This case study deals with the difficulty of synchronizing the output audio and video streams (as shown to the right of Figure 6.10). Having distributed embedded logic analyzers with cross triggering helps us zoom in on regions of interest, such as transitions from one batch of output samples to another (audio/video frame crossings), as these transitions provide points of reference for synchronization. Giving highest priority to temporally proximal transitions among the streams (Features A-C and G) allows us to sample continuously without fear of exhausting the trace buffers before reaching a more informative scenario (such as when transitions on both streams occur close to one another). This gives a better picture of the synchronization status of the audio and video streams. As before, sampling before trigger also enables obtaining synchronization status information both before and after transition points. In both case studies, the concurrency and high data acquisition bandwidth inherent to pre-silicon verification are made possible during post-silicon validation due to the proposed DFD features.

6.4.2 Analyzing area investment of the proposed architecture

In this subsection, the experimental results on area investment for the DFD hardware required to support all the features discussed in Section 6.2.2 under different architectural setups are shown. These area results are obtained using a 90nm standard cell library and a third party synthesis tool [108]. Figures 6.11, 6.13 and 6.15 show how the proposed architecture scales in terms of area when the number of cores, trace buffers, and trace ports change respectively. For the bar charts shown in Figures 6.12, 6.14 and 6.16, the columns are broken down into various sections. This helps better understand how much each part of the inserted DFD hardware contributes to the overall area. The *Top control unit* oversees the entire architecture and communicates with the JTAG port for controlling the DFD hardware during post-silicon validation. The *Communication fabric for sampling* connects the data sources to the trace buffers as shown in Figure 6.1. The *Data source select unit* provide control to the communication fabric for connecting the data sources to trace buffers. The *FSM for real-time decisions* determines how the trace buffers and trace ports are utilized.

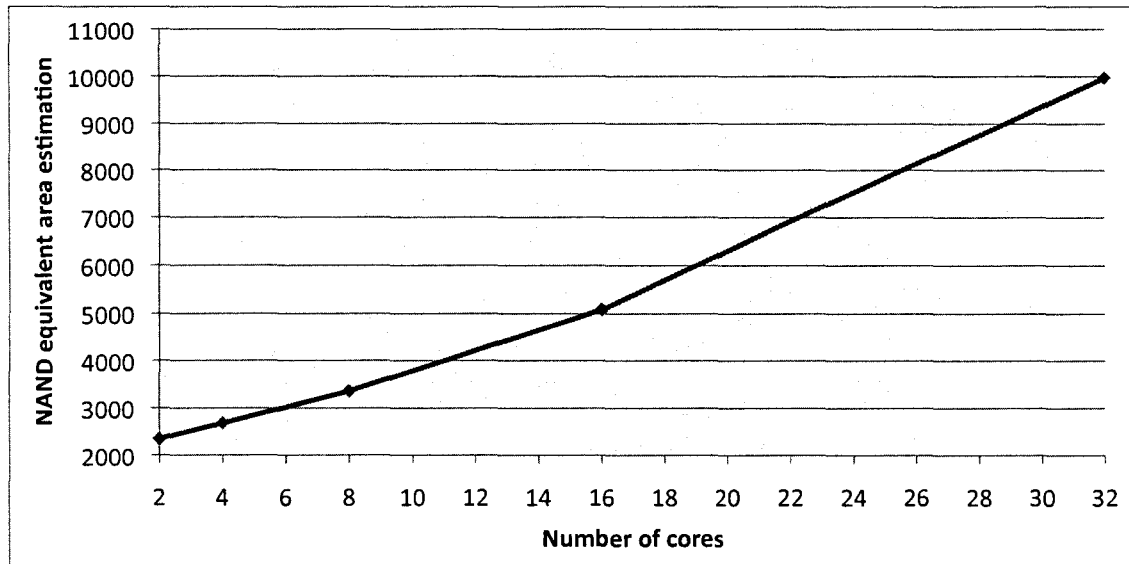


Figure 6.11: Area investment analysis when varying the number of cores

It also communicates with the top control unit. The *Queue control unit* controls the queue, whose area estimation is not included in the results since the queue can be embedded into the trace buffers. The *Trace buffer control unit* provides the write address and activates the write enable for individual trace buffer during data acquisition, and supplies the read address during data offload. The *Offload management unit* provides control to the communication fabric to connect the trace buffers to the trace ports, as well as activating the trace ports when needed. These units just mentioned reside in the Allocation unit shown in Figure 6.3. Finally, the *Communication fabric for offloading* connects the trace buffers to the trace ports. Note that in our implementation, the communication fabrics are implemented using multiplexer networks.

Figures 6.11 and 6.12 show the area investment for the DFD hardware required to support all the features discussed in Section 6.2.2 when varying the number of cores in an SOC. The DFD hardware for these experiments contains one 16 KB trace buffer and one trace port. It can be seen from Figure 6.11 that when the number of cores in

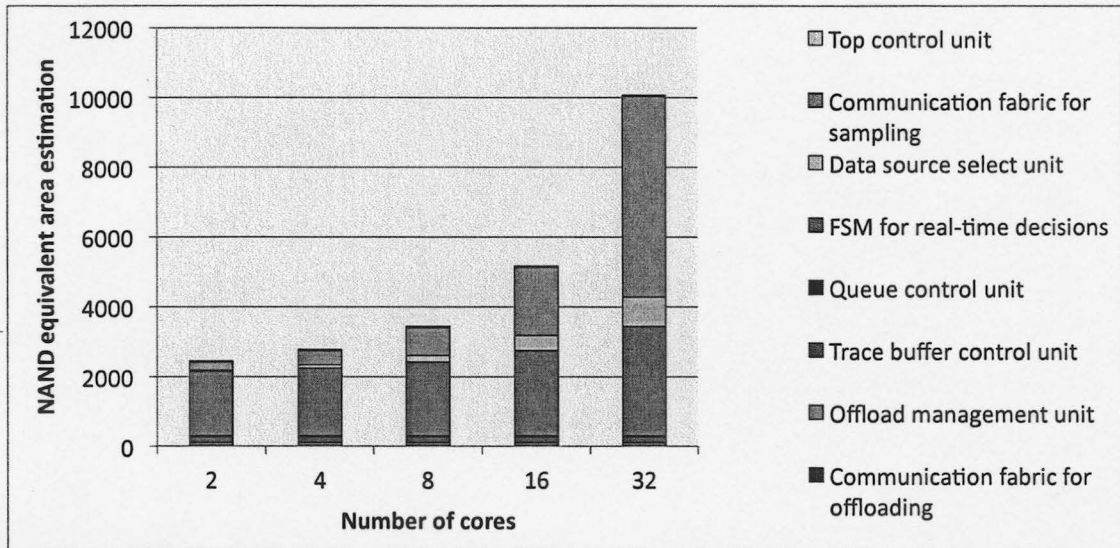


Figure 6.12: Area distribution among hardware components when varying the number of cores

an SOC increases, the area of the DFD hardware scales accordingly. The reason for this can be found in Figure 6.12. When the number of cores increases, the size of the communication fabric between data sources and the single trace buffer, and the data source select unit, also grow in order to provide the proper connections and control between the data sources and the trace buffer. When the number of cores rises, the FSM for making real-time decision on trace buffer allocation becomes more complex, and thus, enlarging the *FSM for real-time decisions*.

Figure 6.13 analyzes the area investment in the DFD architecture when the size, as well as the number of on-chip trace buffers varies. In this figure, all features discussed in Section 6.2.2 are supported for an SOC with 32 data sources and one trace port. As shown in Figures 6.1 and 6.3, a major part of added hardware comes with the *Allocation unit* and *Communication fabric* for controlling and transporting debug data among trace buffers and trace ports. As a result, the communication fabric will have to be expanded and the control logic in the allocation unit will be modified to incorporate the added flexibility to manage sampling and offloading of debug data

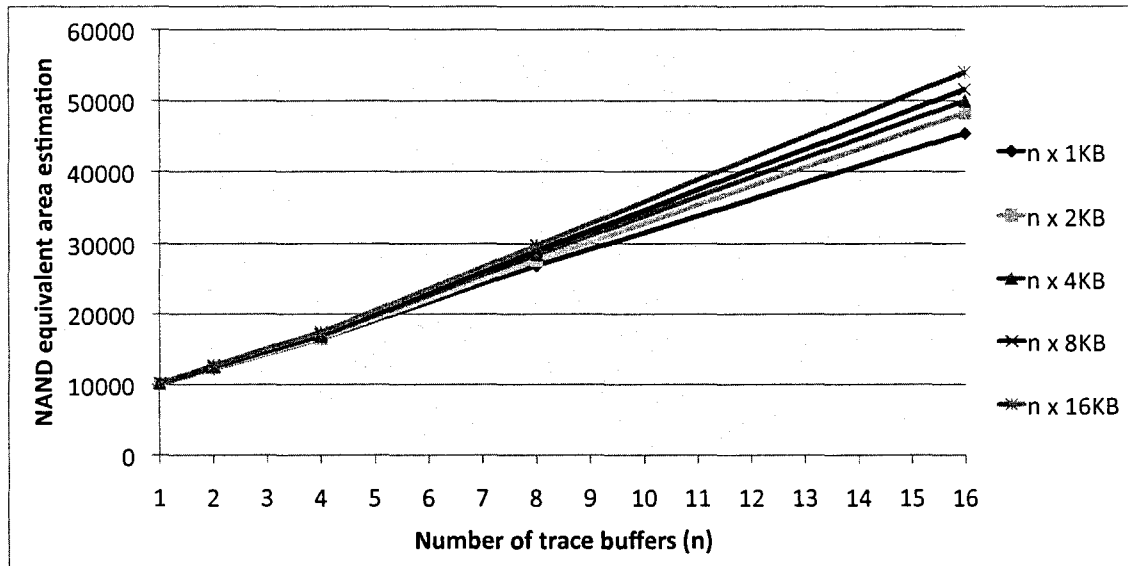


Figure 6.13: Area investment analysis when varying the size and number of trace buffers

when more trace buffers are available. However the depth of the trace buffers does not have a visible impact on logic area because it is only the bitwidth of address registers that scales logarithmically with the trace buffer depth. It is important to note that even with 16 trace buffers and all the features considered, the logic area of the proposed DFD architecture is still below 30% of the total area of the on-chip debug resources (i.e., the embedded memories dominate the total area). It will be shown in the later discussion on experimental results for data acquisition how this 30% area from the control logic enables acquisition of more useful data over the case where it is used simply for more storage capacity.

In order to show how the area investment scales when the number of trace buffers increases, while the size of the trace buffers in terms of storage capacity remains the same in Figure 6.14, the experiment was conducted with 32 data sources connected to five different arrangements of trace buffers: $1 \times 16KB$, $2 \times 8KB$, $4 \times 4KB$, $8 \times 2KB$ and $16 \times 1KB$ using one trace port. It can be seen that with the same storage capacity, the additional area required varies with the arrangement of trace buffers. This is

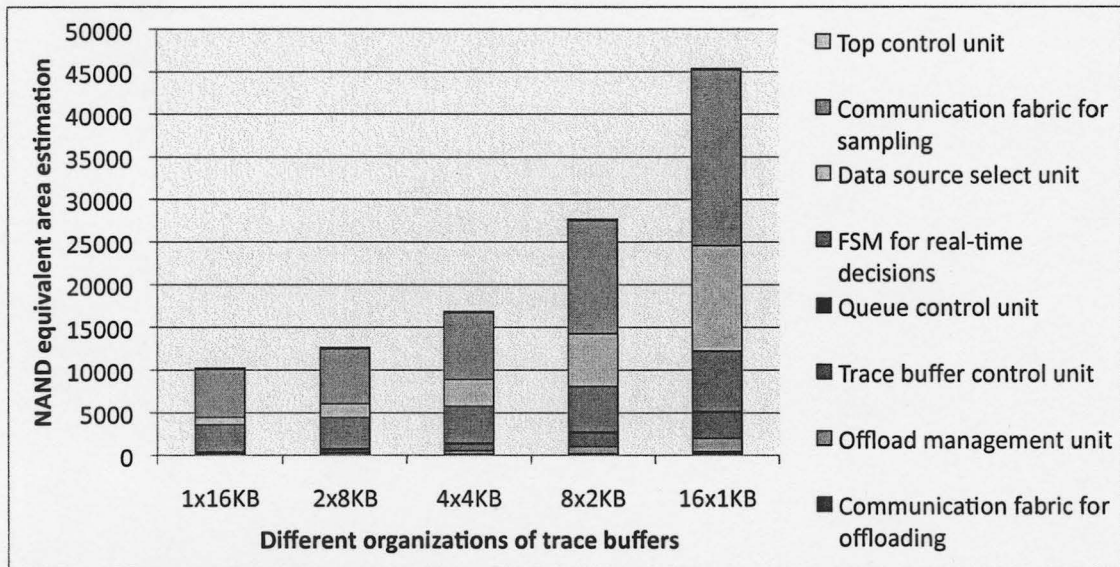


Figure 6.14: Area investment analysis when varying the organization of trace buffers

because as the number of trace buffers increases, the two communication fabrics have to be extended to provide the necessary connections between the data sources, trace buffers and the trace ports. Also, extra hardware will be required for the added complexity in the FSM for making real-time decisions, as well as in the control units (i.e., Data source select unit, Trace buffer control unit and Offload management unit) for providing control to the trace buffers and the expanded communication fabrics.

When the number of trace ports increases, the area investment scales up linearly as shown in Figure 6.15. In this figure, as well as in Figure 6.16, which gives the distribution of additional hardware among different components, there are 32 data sources and 16 1KB trace buffers. As can be seen in Figure 6.16, the increase in area when the number of trace ports rise is contributed from the larger communication fabric between the trace buffers and the trace ports, as well as from the offload management unit for providing controls to the fabric.

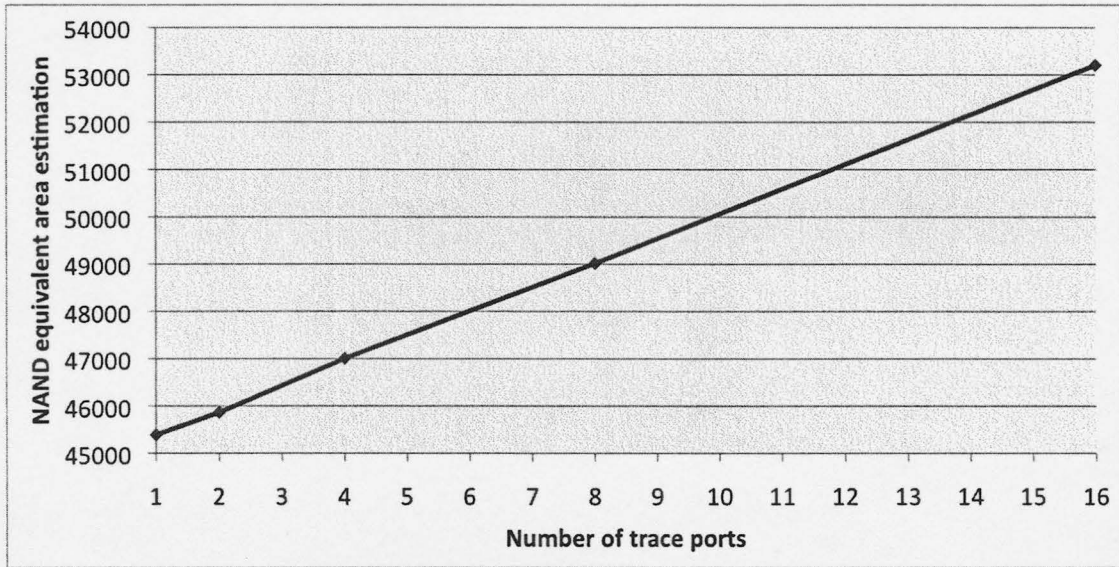


Figure 6.15: Area investment analysis when varying the number of trace ports

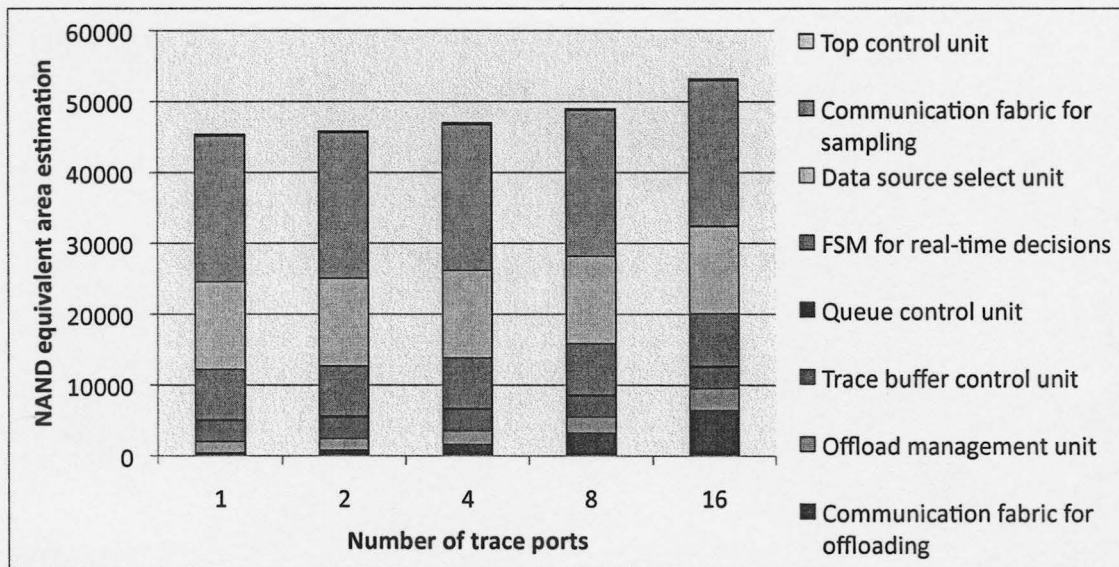


Figure 6.16: Area distribution among hardware components when varying the number of trace ports

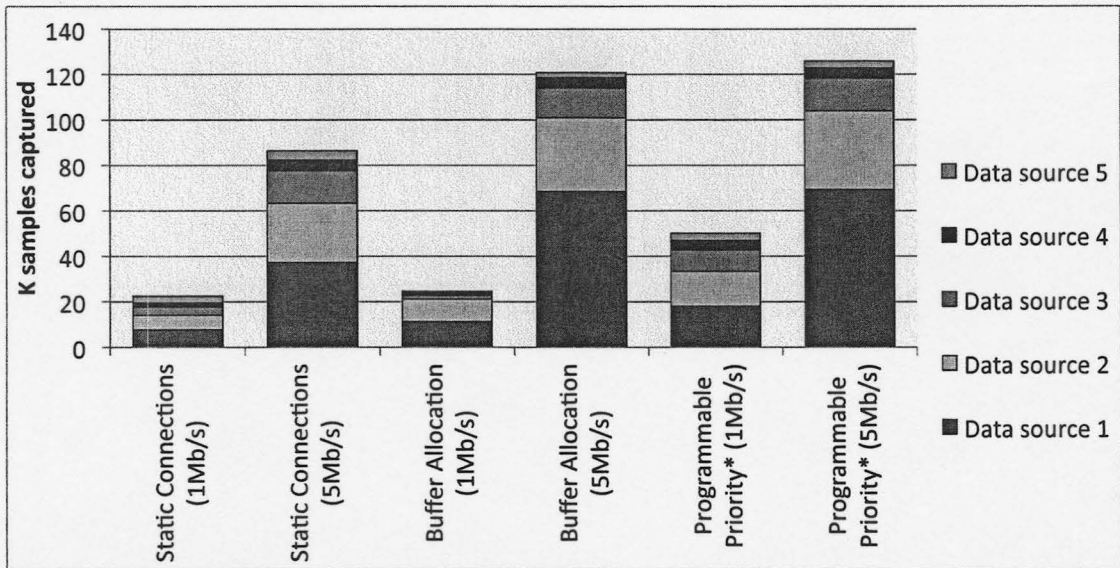


Figure 6.17: Impact of various features on acquisition of debug data

6.4.3 Analyzing data acquisition of the proposed architecture

Figure 6.17 shows data captured for an experiment consisting of the MPEG hardware discussed earlier, with 5 data sources (cores), three 16-bit trace buffers of depths 1K, 2K and 4K, and a trace port with bandwidth 1 Mb/s and 5 Mb/s. The first two columns show debug data obtained with static connections between the data sources and trace buffers. The next two columns show the improvement obtained by adding the communication fabric, and the last two columns show the benefit of having programmable priority. It can be seen that when programmable priority is supported, the proposed DFD architecture can gather the most amount of debug data. The improvement is not as significant when the bandwidth of the trace port is 5 Mb/s because the ratio between offload bandwidth and acquisition bandwidth is close to 1. However, when the acquisition bandwidth surpasses the offload bandwidth (as in the case when a trace port with only 1 Mb/s is available), a significant amount of previously lost data can be recovered by adjusting the priority settings during the debug

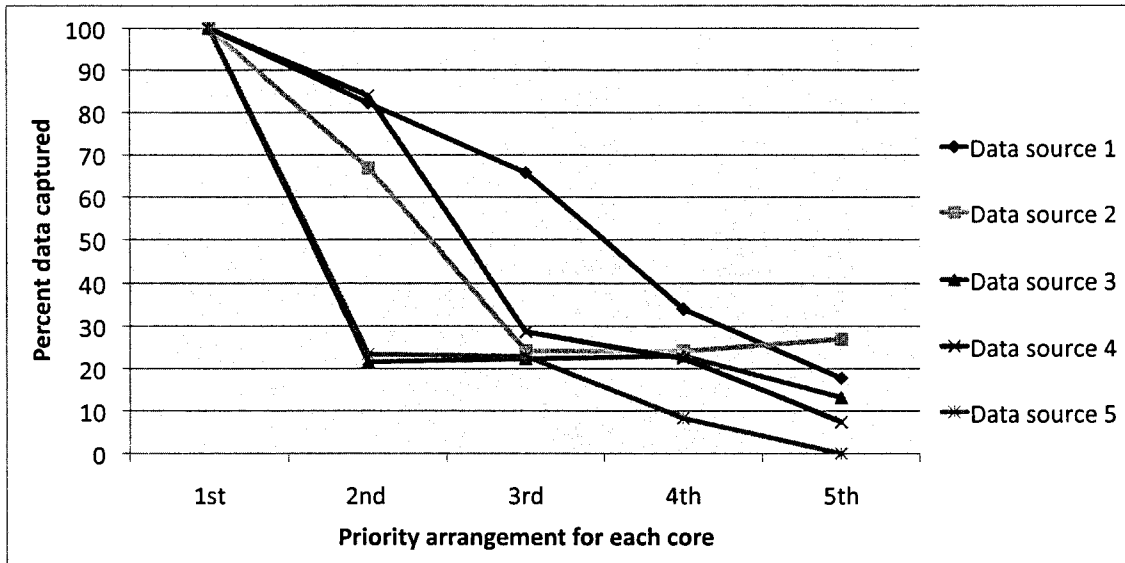


Figure 6.18: Effect of programmable priority on data loss

experiments. Although there is a noticeable initial area investment when supporting the programmable priority feature, if this area is used merely for deeper buffers, data loss can still occur. In this way, a high ratio between offload and acquisition bandwidth justifies the need for programmable priority. Despite the availability of higher bandwidth trace ports [10], off-chip bandwidth can never keep pace with on-chip signal density, especially when acquiring signals for use in software debug. Thus there will always be need for selective acquisition via programmable priority.

Finally, Figure 6.18 further illustrates how programmable priority functions when data acquisition rate exceeds offload bandwidth under the same architectural setup used for the experiment shown in Figure 6.17. Based on a rotating priority scheme, a data source will always acquire the most data when having the highest priority. Beyond highest priority, the amount of data captured is heavily dependent on the sampling behavior of other sources with higher priority, and when the offload bandwidth is much less than the acquisition rate, the amount of captured data can fall steeply when the data source is not at the top of the priority list.

6.5 Summary

In this chapter of the thesis, we have introduced a new design-for-debug architecture for distributed embedded logic analysis that enables real-time observability. Using two case studies on a digital video decoder we have presented its advantages in improving real-time observability of the design. This helps bridge the gap between pre-silicon verification and post-silicon validation. We have also analyzed the costs of managing on-chip distributed trace buffers. Because the area for on-chip real-time debug is dominated by embedded memories, we have learned that the area investment for the proposed architectural features and intelligent control is below 10% of the total area required for debug.

Chapter 7

Conclusion and future work

As the complexity of integrated digital circuits and systems continues to grow, it is increasingly difficult to guarantee error-free first silicon. In order to shorten time-to-market and avoid escalating fabrication costs due to design re-spins, DFD techniques have been experiencing a growing acceptance for reducing the burden of post-silicon validation.

Among the numerous DFD techniques that have been proposed in recent years, embedded logic analysis has received increased attention due to its ability to acquire debug data in real-time in-system. However, the amount of data one can acquire is limited by the capacity of the on-chip trace buffers in the ELAs. In this dissertation, we have proposed novel architectures and algorithmic solutions to address the challenges on various parts of the ELA, as well as when multiple ELAs are employed for utilizing the limited storage space more efficiently.

The rest of this chapter is organized as follows. Section 7.1 summarizes the contributions in this thesis. Section 7.2 discusses the direction for possible future work.

7.1 Summary of dissertation contributions

The decisions on when to acquire debug data during post-silicon validation are determined by trigger events that are programmed into on-chip trigger units. In Chapter 3 of this thesis, we investigate how to design trigger units that are both resource-efficient

and runtime programmable. To achieve these two goals, we introduce new architectural features, as well as an algorithm for automatically mapping trigger events onto trigger units. In this new architecture, we allow the user to program trigger conditions onto a lower number of comparators than the number of prime implicants in the condition function. We then perform real-time false trigger analysis to conclude if the event that triggers data acquisition is valid. Also, to maintain the efficiency of the false trigger analysis, an algorithm is introduced to map the trigger events onto trigger units with only a small number of false events. Together, the proposed architecture and algorithm improve controllability of the CUD by allowing more complex trigger events to be programmed onto an existing trigger unit, while preserving precious storage space through false trigger analysis during post-silicon validation.

Despite the recent advancement in the design of ELAs, the reluctance to invest additional area for large trace buffers only for the purpose of post-silicon validation limits the amount of available data that can be acquired on-chip. This indirectly translates into a more time-consuming process for identifying the design errors. Thus, it is desirable to find a way to better utilize the acquired data on the trace signals, such that as much missing data (for other internal signals) as possible can be reconstructed. This goal is achieved in Chapter 4 by introducing automated solutions for state restoration in the CUD. Using the proposed algorithm, data in state elements across multiple time frames can be restored using debug data from the trace signals. Also, in order for the state restoration algorithm to be applicable to large designs and data to be restored over thousands of clock cycles, a compute-efficient version of the same algorithm that exploits bitwise parallelism is introduced.

The decisions on which signals should be hardwired to the trace buffers of the ELA are made during the design cycle early in the design flow. However, it is not known a-priori what types of bugs should be expected, and thus, it is impossible to predict which signals will provide more information during post-silicon validation. In Chapter 5, two metrics are introduced for analyzing the topology as well as the logic behaviors of circuit nodes in a design. The result of the analysis is then used by the proposed algorithms for selecting the trace signals automatically. One of the two proposed algorithms provides a more detailed analysis during signal selection, while the other

algorithm performs a good estimate on which signals should be traced with reasonable runtime. When performing state restoration on the debug data acquired from the suggested trace signals, a significant amount of data can be reconstructed, and thus, effectively improving the observability of the CUD during post-silicon validation.

To deal with concurrent activity in multi-core SOCs, the idea of distributed triggering is introduced. However, this brings in new challenges on how to effectively utilize the limited storage space among all the trace buffers when multiple data sources are presented. Motivated by the assumptions that the number of cores in future SOCs will continue to increase, and high-speed trace ports will gain a wider adoption with the proliferation of high-speed I/Os, Chapter 6 introduces a new DFD architecture for core-based SOCs. In this resource-efficient and scalable architecture, intelligent control is placed on-chip to automatically allocate distributed trace buffers to handle debug data acquisition from multiple data sources located in different cores in real-time. At the same time, any idle high-speed trace ports will be allocated to stream the sampled data off-chip to reclaim the valuable storage space in the on-chip trace buffers. The proposed architecture also complements existing system-level debug techniques as it provides a means to transport debug data in an efficient manner, without concerning whether the debug data has been pre-processed using techniques such as trace filtering and trace compression.

7.2 Possible future research directions

A few possible future research directions have been identified and they are outlined in this subsection.

The proposed triggering methodology (architecture and algorithm) from Chapter 3 could be extended to allow trigger events based on different groups of signals, which may or may not be connected directly to the trigger unit, to be programmed at runtime. For example, due to space limitations only 32 signals can be connected to the trigger unit. However, these signals are logically related (in both space and time) to a larger pool of signals from the design. It would be useful to facilitate the description of logic conditions for triggering on the signals that are related, but not

necessarily connected directly, to the trigger unit. In this case, the proposed trigger event mapping algorithm will have to be modified accordingly.

The continuous improvement on the design of ELAs together with the use of the proposed state restoration algorithms discussed in Chapter 4 will enlarge the amount of available debug data during post-silicon validation. Although having more data helps provide more information to understand the errors, analyzing and isolating only the useful data is very time consuming and in the current state-of-the-art it is done using manual techniques. As a result, it will be beneficial to introduce algorithmic solutions to analyze and identify these useful data so that they can be presented to the debug engineer in a more meaningful way. This can help speed up the debug process and thus, reduce the overall time spent on post-silicon validation.

The metrics proposed in Chapter 5 for trace signal selection only provide a rough estimate on the restorability of each signal by analyzing the structure of a design. When more information such as the expected behavior of the design is available, these metrics can be refined and thus help produce more accurate measure on the influence of tracing each signal. Another interesting research direction is on mixing complementary debug techniques to gain the benefits of both worlds. If the trace buffer-based technique is combined with scan chain-based technique (for example a scan dump is done at the end of a debug session when the trace buffer is filled), new metrics and algorithms for selecting the trace signals will need to be investigated.

The future generation of SOCs is likely to have more cores that operates at different frequencies and have more complex interactions among them. Moreover, depending on the functionalities of these cores, they may be grouped together into smaller clusters to resemble a hierarchical structure in order to reduce the complexity of the communication channels on the chip. In this case, instead of utilizing one unit to allocate trace buffers in the proposed architecture in Chapter 6, it may be worth studying the advantages and disadvantages on employing a hierarchical allocation unit. This is done by using multiple simpler allocation units for each cluster of cores. These simpler allocation units will then be monitored by a global controller for transporting data between clusters to balance the acquisition bandwidth available from the trace buffers in each cluster.

In summary, post-silicon validation is a key step in the implementation flow of integrated circuits and systems. Although many ad-hoc techniques exist in practice, the escalating complexity of the state-of-the-art designs, combined with the emerging business models in the semiconductor industry based on core providers and system integrators, motivate the need for structured and algorithmic solutions in the future. This thesis has investigated several key research problems for the design of the debug infrastructure for post-silicon validation. New algorithmic solutions have been proposed and analyzed. Although not a single solution in the field of post-silicon validation is one-fit-all, the work from this thesis is an important step toward the adoption of more structured and algorithmic methods in the field.

Bibliography

- [1] Miron Abramovici. In-System Silicon Validation and Debug. *IEEE Design and Test of Computers*, 25(3):216–223, 2008.
- [2] Miron Abramovici. Experience and Opinion (Design for Debug). In *Proceedings of the IEEE International Workshop on Silicon Debug and Diagnosis*, 2006.
- [3] Miron Abramovici, Paul Bradley, Kumar Dwarakanath, Peter Levin, Gerard Memmi, and Dave Miller. A Reconfigurable Design-for-Debug Infrastructure for SoCs. In *Proceedings of the IEEE/ACM Design Automation Conference*, pages 7–12, 2006.
- [4] Accellera Standard. Open Verification Library, Jan 2009. URL <http://www.eda.org/ovl>.
- [5] W. James Allen, Bernard M. McFarland, Terry L. Fruehling, Kevin M. Gertiser, Roy M. Fildes, Bruce C. Young, and Mark T. Lowden. On-Chip Instrumentation. USPTO Patent Full-Text and Image Database, Apr 2005. US 6877114.
- [6] Altera Verification Tool. SignalTap II Embedded Logic Analyzer, 2006. URL <http://www.altera.com>.
- [7] Ehab Anis and Nicola Nicolici. Low Cost Debug Architecture using Lossy Compression for Silicon Debug. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe*, pages 225–230, 2007.
- [8] Ehab Anis and Nicola Nicolici. On Using Lossless Compression of Debug Data

- in Embedded Logic Analysis. In *Proceedings of the IEEE International Test Conference*, 2007. Paper 18.3.
- [9] ARM Limited. Embedded Trace Macrocells, Apr 2007. URL <http://www.arm.com>.
- [10] ARM Limited. CoreSight On-chip Debug and Trace Technology, Feb 2008. URL <http://www.arm.com>.
- [11] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006. URL <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html>.
- [12] Félix Beaudoin, Jean-Philippe Roux, Kevin Sanchez, and Philippe Perdu. Dynamic Optical Circuit Analysis for IC Design Debug. In *Proceedings of the IEEE International Workshop on Silicon Debug and Diagnosis*, pages 368–373, 2007.
- [13] H. Bhatnagar. *Advanced ASIC Chip Synthesis*. Kluwer Academic Publishers, 2nd edition, 2002.
- [14] Marc Boule and Zeljko Zilic. Efficient Automata-Based Assertion-Checker Synthesis of PSL Properties. In *Proceedings of the 11th IEEE International High-Level Design Validation and Test Workshop*, pages 69–76, 2006.
- [15] Marc Boule, Jean-Samuel Chenard, and Zeljko Zilic. Adding Debug Enhancements to Assertion Checkers for Hardware Emulation and Silicon Debug. In *Proceedings of the IEEE International Conference on Computer Design*, 2006.
- [16] Marc Boule, Jean-Samuel Chenard, and Zeljko Zilic. Debug Enhancements in Assertion-Checker Generation. *IET Computers and Digital Techniques*, 1(6): 669–677, Nov 2007.

- [17] Marc Boule, Jean-Samuel Chenard, and Zeljko Zilic. Assertion Checkers in Verification, Silicon Debug and In-Field Diagnosis. In *Proceedings of the IEEE International Symposium on Quality Electronic Design*, pages 613–620, 2007.
- [18] F. Brglez, D. Bryan, and K. Kozminski. Combinational Profiles of Sequential Benchmark Circuits. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 1929–1934, 1989.
- [19] M. Burtscher, I. Ganusov, S.J. Jackson, J. Ke, P. Ratanaworabhan, and N.B. Sam. The VPC Trace-Compression Algorithms. *IEEE Transactions on Computers*, 54(11):1329–1344, Nov 2005.
- [20] M. L. Bushnell and Vishwani D. Agrawal. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Kluwer Academic Publishers, Boston, 2000.
- [21] K.M. Butler and J. Saxena. An Empirical Study on the Effects of Test Type Ordering on Overall Test Efficiency. In *Proceedings of the IEEE International Test Conference*, pages 408–416, 2000.
- [22] Cadence Design System Inc. Cadence Assertion-Based Verification, Jan 2009. URL <http://www.cadence.com>.
- [23] Cadence Design System Inc. Digital Implementation, Jan 2009. URL <http://www.cadence.com>.
- [24] O. Caty, P. Dahlgren, and I. Bayraktaroglu. Microprocessor Silicon Debug Based on Failure Propagation Tracing. In *Proceedings of the IEEE International Test Conference*, 2005. Paper 12.2.
- [25] S. Chakravarty and V.P. Dabholkar. Two Techniques for Minimizing Power Dissipation in Scan Circuits during Test Application. In *Proceedings of the 3rd IEEE Asian Test Symposium*, pages 324–329, 1994.

- [26] M. D. Chaplain. Logic Analysis Systems - Today's Mixed Domain Measurement Solution. In *Proceedings of IEE Colloquium on Design and Test of Mixed Analogue and Digital Circuits*, pages 2/1–2/3, 1990.
- [27] William D. Corti, Jr. Robert Kenny, Joseph O. Marsh, Steven C. Parker, Frank X. Scanzano, and Michael Won. On-Chip Logic Analyzer. USPTO Patent Full-Text and Image Database, Dec 2004. US 6834360.
- [28] P. Dahlgren, P. Dickinson, and I. Parulkar. Latch Divergency in Microprocessor Failure Analysis. In *Proceedings of the IEEE International Test Conference*, pages 755–769, 2003.
- [29] R. Datta, A. Sebastine, and J.A. Abraham. Delay Fault Testing and Silicon Debug using Scan Chains. In *Proceedings of the IEEE European Test Symposium*, pages 46–51, 2004.
- [30] R. Doering and Y. Nishi. *Handbook of Semiconductor Manufacturing Technology*. Prentice-Hall, Inc, 2007.
- [31] V. D'Silva, D. Kroening, and G. Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, Jul 2008.
- [32] First Silicon Solutions. System navigator pro, Feb 2008. URL <http://www.fs2.com/snnav-pro.html>.
- [33] Harry D. Foster, Adam C. Krolnik, and David J. Lacey. *Assertion-Based Design*. Kluwer Academic Publishers, May 2004.
- [34] J. Geuzebroek and Bart Vermeulen. Integration of Hardware Assertion in Systems-on-Chip. In *Proceedings of the IEEE International Test Conference*, 2008. Digital Object Identifier 10.1109/TEST.2008.4700593.
- [35] Lawrence H. Goldstein. Controllability/Observability Analysis of Digital Circuits. *IEEE Transactions on Circuits and Systems*, 26(9):685–693, Sep 1979.

- [36] T. Grotker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Springer, 2002.
- [37] Xinli Gu, Weili Wang, K. Li, Heon Kim, and S.S. Chung. Re-using DFT Logic for Functional and Silicon Debugging Test. In *Proceedings of the IEEE International Test Conference*, pages 648–656, Oct 2002.
- [38] R. Gupta and Y. Zorian. Introducing Core-Based System Design. *IEEE Design and Test of Computers*, 14(4):15, 25 1997.
- [39] B. J. Hammond. Development in Logic Analysers. In *Proceedings of IEE Colloquium on Instrumentation in Electronic Product Manufacture*, pages 2/1 – 2/3, 1989.
- [40] B. J. Hammond. Testing Time-Advances in Logic Analysis. *IEE Review*, 36(3): 103–106, Mar 1990.
- [41] Alan L. Herrmann and Greg P. Nugent. Embedded Logic Analyzer for a Programmable Logic Device. USPTO Patent Full-Text and Image Database, May 2002. US 6389558.
- [42] A.B.T. Hopkins and K.D. McDonald-Maier. Debug Support for Complex Systems On-Chip: A Review. *IEE Proceedings of Computers and Digital Techniques*, 153(4):197–207, Jul 2006.
- [43] A.B.T. Hopkins and K.D. McDonald-Maier. Debug Support Strategy for System-on-Chips with Multiple Processor Cores. *IEEE Transactions on Computers*, 55(2):174–184, Feb 2006.
- [44] Yu-Chin Hsu, Furshing Tsai, Wells Jong, and Ying-Tsai Chang. Visibility Enhancement for Silicon Debug. In *Proceedings of the IEEE/ACM Design Automation Conference*, pages 13–18, 2006.
- [45] JL Huang, CML James, and DMH Walker. *VLSI Test Principles and Architectures: Design for Testability*. Academic Press, 2006.

- [46] Yu Huang and Wu-Tung Cheng. Using Embedded Infrastructure IP for SOC Post-Silicon Verification. In *Proceedings of the IEEE/ACM Design Automation Conference*, pages 674–677, Jun 2003.
- [47] IEEE JTAG 1149.1-2001 Std. *IEEE Standard Test Access Port and Boundary-Scan Architecture*. IEEE Computer Society, 2001.
- [48] IEEE Standard 1666-2005. *IEEE Standard System C Language*. IEEE Computer Society, 2005.
- [49] IEEE Standard 1850-2005. *IEEE Standard for Property Specification Language (PSL)*. IEEE Computer Society, 2005.
- [50] E.E. Johnson, Jiheng Ha, and M. Baqar Zaidi. Lossless Trace Compression. *IEEE Transactions on Computers*, 50(2):158–173, Feb 2001.
- [51] D. Josephson. The Manic Depression of Microprocessor Debug. In *Proceedings of the IEEE International Test Conference*, pages 657–663, Oct 2002.
- [52] D. Josephson and B. Gottlieb. The Crazy Mixed up World of Silicon Debug. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 665–670, 2004.
- [53] M. Keating and P. Bricaud. *Reuse Methodology Manual: For System-on-a-Chip Designs*. Kluwer Academic Publishers, 1998.
- [54] 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, Apr 1999.
- [55] Ho Fai Ko and Nicola Nicolici. Functional Scan Chain Design at RTL for Skewed-load Delay Fault Testing. In *Proceedings of the 13th IEEE Asian Test Symposium*, pages 454–459, 2004.
- [56] Ho Fai Ko and Nicola Nicolici. Automated Trace Signals Identification and State Restoration for Improving Observability in Post-Silicon Validation. In

- Proceedings of the IEEE/ACM Design, Automation and Test in Europe*, pages 1298–1303, 2008.
- [57] Ho Fai Ko and Nicola Nicolici. On Automated Trigger Event Generation in Post-Silicon Validation. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe*, pages 256–259, 2008.
- [58] Ho Fai Ko and Nicola Nicolici. Resource-Efficient Programmable Trigger Units for Post-Silicon Validation. In *Proceedings of the IEEE European Test Symposium*, 2009. To be appeared.
- [59] Ho Fai Ko and Nicola Nicolici. Functional Illinois Scan Design at RTL. In *Proceedings of the IEEE International Conference on Computer Design*, pages 78–81, 2004.
- [60] Ho Fai Ko and Nicola Nicolici. RTL Scan Design for Skewed-Load At-Speed Test under Power Constraints. In *Proceedings of the IEEE International Conference on Computer Design*, pages 237–242, 2006.
- [61] Ho Fai Ko and Nicola Nicolici. A Novel Automated Scan Chain Division Method for Shift and Capture Power Reduction in Broadside At-Speed Test. In *Proceedings of the IEEE International Symposium on Quality Electronic Design*, pages 649–654, 2008.
- [62] Ho Fai Ko and Nicola Nicolici. Scan Division Algorithm for Shift and Capture Power Reduction for At-Speed Test using Skewed-Load Test Application Strategy. *Journal of Electronic Testing: Theory and Application*, 24(4):393–403, Aug 2008.
- [63] Ho Fai Ko and Nicola Nicolici. Automated Scan Chain Division for Reducing Shift and Capture Power During Broadside At-Speed Test. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(11):2092–2097, Nov 2008.

- [64] Ho Fai Ko and Nicola Nicolici. Algorithms for State Restoration and Trace Signals Selection for Data Acquisition in Silicon Debug. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(2):285–297, Feb 2009.
- [65] Ho Fai Ko, Qiang Xu, and Nicola Nicolici. Register-Transfer Level Functional Scan for Hierarchical Designs. In *Proceedings of the IEEE/ACM Asian-South Pacific Design Automation Conference*, pages 1172–1175, 2005.
- [66] Ho Fai Ko, A.B. Kinsman, and Nicola Nicolici. Distributed Embedded Logic Analysis for Post-Silicon Validation of SOCs. In *Proceedings of the IEEE International Test Conference*, 2008. Paper 16.3.
- [67] Thomas Kropf. *Introduction to Formal Hardware Verification*. Springer-Verlag, 2000.
- [68] Michael Stephen Floyd Lakshminarayana Baba Arimilli, Larry Scott Leitner, Kevin F. Reick, and Jennifer Lane Vargus. Multi-State Logic Analyzer Integral to a Microprocessor. USPTO Patent Full-Text and Image Database, Oct 2003. US 6633838.
- [69] W. K. Lam. *Hardware Design Verification: Simulation and Formal Method-Based Approaches*. Prentice-Hall, Inc, 2005.
- [70] Mario Larouche and Doug Amos. We’re on the board... what now? Solving the visibility issue in FPGA prototyping. In *Proceedings of the IEEE International Workshop on Silicon Debug and Diagnosis*, pages 396–397, 2007.
- [71] Lauterbach. Lauterbach, Dec 2007. URL <http://www.lauterbach.com/frames.html>.
- [72] R. Leatherman and N. Stollon. An Embedding Debugging Architecture for SOCs. *IEEE Potentials*, 24(1):12–16, Feb-Mar 2005.
- [73] Y. Luo and L.K. John. Locality-based Online Trace Compression. *IEEE Transactions on Computers*, 53(6):723–731, Jun 2004.

- [74] C. MacNamee and D. Heffernan. Emerging On-chip Debugging Techniques for Real-Time Embedded Systems. *IEE Computing & Control Engineering Journal*, 11(6):295–303, Dec 2000.
- [75] John W. Mates. Method and Apparatus for Testing an Integrated Circuit Using an On-Chip Logic Analyzer Unit. USPTO Patent Full-Text and Image Database, May 2003. US 6564347.
- [76] A. Mayer, H. Siebert, and K.D. McDonald-Maier. Debug Support, Calibration and Emulation for Multiple Processor and Powertrain Control SoCs. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe*, volume 3, pages 148–152, 2005.
- [77] A. Mayer, H. Siebert, and K.D. McDonald-Maier. Boosting Debugging Support for Complex Systems on Chip. *IEEE Computers*, 40(4):76–81, Apr 2007.
- [78] Mentor Graphics Corp. Mentor Graphics Assertion-Based Verification, Jan 2009. URL <http://www.mentor.com>.
- [79] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Inc., 1994.
- [80] Microchip Technology Inc. MPLAB REAL ICE In-Circuit Emulator, Feb 2008. URL <http://www.microchip.com>.
- [81] Kevin Morris. On-Chip Debugging - Built-in Logic Analyzers on your FPGA. *Journal of FPGA and Structured ASIC*, 2(3), Jan 2004.
- [82] A.S. Mudlapur, V.D. Agrawal, and A.D. Singh. A Random Access Scan Architecture to Reduce Hardware Overhead. In *Proceedings of the IEEE International Test Conference*, 2005. Paper 15.1.
- [83] N. Nataraj, T. Lundquist, and Ketan Shah. Fault Localization using Time Resolved Photon Emission and STIL Waveforms. In *Proceedings of the IEEE International Test Conference*, pages 254–263, Sep 2003.

- [84] Z. Navabi. *VHDL: Analysis and Modeling of Digital Systems*. McGraw-Hill Inc., 1997.
- [85] Z. Navabi. *Verilog Digital System Design (Professional Engineering)*. McGraw-Hill Inc., 1999.
- [86] Robert B. Norwood and Edward J. McCluskey. Delay Testing for Sequential Circuits with Scan. Technical report, Center for Reliable Computing, Stanford University, 1997.
- [87] 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 of Computers and Digital Techniques*, 153(1):47–58, Jan 2006.
- [88] M. Paniccia, T. Eiles, V. R. M. Rao, and W. M. Yee. Novel Optical Probing Technique for Flip Chip Packaged Microprocessors. In *Proceedings of the IEEE International Test Conference*, pages 740–747, Oct 1998.
- [89] Janak H. Patel. Stuck-at Fault: A Fault Model for the Next Millennium. In *Proceedings of the IEEE International Test Conference*, 1998. Paper P10.2.
- [90] Bradley Quinton and Steven Wilton. Programmable Logic Core Based Post-Silicon Debug For SoCs. In *Proceedings of the IEEE International Workshop on Silicon Debug and Diagnosis*, pages 430–435, 2007.
- [91] P. Rashinkar, P. Paterson, and L. Singh. *System-on-a-chip Verification: Methodology and Techniques*. Kluwer Academic Publishers, 2001.
- [92] Jouni Riihimäki, Pasi Kolinummi, Mika Koikkalainen, and Jouko Skog. Scan Chain Based Functional Debugging. In *Proceedings of the IEEE International Workshop on Silicon Debug and Diagnosis*, pages 134–137, 2007.
- [93] M.W. Riley and M. Genden. Cell Broadband Engine Debugging for Unknown Events. *IEEE Design and Test of Computers*, 24(5):486–493, 2007.

- [94] Samitha Samaranayake, Emil Gizdarski, Nodari Sitchinava, Frederic Neuveux, Rohit Kapur, and T. W. Williams. A Reconfigurable Shared Scan-in Architecture. In *Proceedings of the 21st IEEE VLSI Test Symposium*, pages 9–14, 2003.
- [95] Smruti Sarangi, Satish Narayanasamy, Bruce Carneal, Abhishek Tiwari, Brad Calder, and Josep Torrellas. Patching Processor Design Errors with Programmable Hardware. *IEEE Micro Special Issue: Micro's Top Picks from Computer Architecture Conferences*, 27(1):12–25, 2007.
- [96] S.R. Sarangi, B. Greskamp, and J. Torrellas. CADRE: Cycle-Accurate Deterministic Replay for Hardware Debugging. In *Proceedings of International Conference on Dependable Systems and Networks*, pages 301–312, Jun 2006.
- [97] Jacob Savir and Srinivas Patil. Scan-Based Transition Test. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(8):1232–1241, 1993.
- [98] Josef Schicker, Christian Lipsky, Jens Braunes, and Stefan Weisse. A new Debug Methodology for Efficient Use of Resources offered by Complex On-Chip Debug Solutions. In *Proceedings of the IEEE International Workshop on Silicon Debug and Diagnosis*, pages 153–164, 2007.
- [99] Rudolf Schlangen, Uwe Kerst, and Christian Boit. New Circuit Edit and Probing Options directly to FET Device on Ultra Thin Silicon Backside processed by Focused Ion Beam. In *Proceedings of the IEEE International Workshop on Silicon Debug and Diagnosis*, pages 328–333, 2007.
- [100] J.M. Soden and R.E. Anderson. IC Failure Analysis: Techniques and Tools for Quality Reliability Improvement. In *Proceedings of IEEE*, volume 81, pages 703–715, May 1993.
- [101] IEEE Industry Standards and Technology Organization. The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface, 2003. URL <http://www.nexus5001.org>.

- [102] N. Stollon, R. Leatherman, B Ableidinger, and E Edgar. Multi-Core Embedded Debug for Structured ASIC Systems. In *Proceedings of DesignCon*, 2004.
- [103] C.B. Stunkel, B. Janssens, and W.K. Fuchs. Address Tracing for Parallel Machines. *IEEE Computers*, 24(1):31–38, Jan 1991.
- [104] S. Sutherland, S. Davidmann, and P. Flake. *A Guide to Using System Verilog for Hardware Design and Modeling*. Springer, 2006.
- [105] A. Swaine and J. Horley. Summary of New Features in ETMv3, 2005. URL <http://www.arm.com>.
- [106] Synopsys Inc. Synopsys Verification, Jan 2009. URL <http://www.synopsys.com>.
- [107] Synopsys Inc. Formality Equivalence Checker, Jan 2009. URL <http://www.synopsys.com>.
- [108] Synopsys Synthesis Tools. Design Compiler, 2003. URL <http://www.synopsys.com>.
- [109] Synopsys Test Tools. TetraMAX ATPG, 2003. URL <http://www.synopsys.com>.
- [110] Synplicity Inc. Identify, May 2007. URL <http://www.synplicity.com>.
- [111] B. Tabbara, Y.-C. Hsu, G. Bakewell, and S. Sandler. Assertion-Based Hardware Debugging. In *Design & Verification Conference & Exhibition*, 2003. Paper 1.2.
- [112] Shan Tang and Qiang Xu. A Multi-Core Debug Platform for NOC-Based Systems. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe*, 2007.
- [113] C. Thibeault, D. Tremblay, and Y. Hariri. Redefining the Role of Functional Testing. In *Proceedings of IEEE North-East Workshop on Circuits and Systems*, pages 133–136, 2006.

- [114] D.P. Vallett. IC Failure Analysis: The Importance of Test and Diagnostics. *IEEE Design and Test of Computers*, 14(3):76–82, Jul 1997.
- [115] G.J. Van Rootselaar and Bart Vermeulen. Silicon Debug: Scan Chains Alone are Not Enough. In *Proceedings of the IEEE International Test Conference*, pages 892–902, 1999.
- [116] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS. In *Proceedings of the IEEE International Solid-State Circuit Conference*, pages 98–99, 589, 2007.
- [117] Kerry Veenstra, Krishna Rangasayee, and Alan L. Herrmann. Enhanced Embedded Logic Analyzer. USPTO Patent Full-Text and Image Database, Sep 2001. US 6286114.
- [118] B. Vermeulen, M. Z. Urfianto, and Sandeep Kumar Goel. Automatic Generation of Breakpoint Hardware for Silicon Debug. In *Proceedings of the IEEE/ACM Design Automation Conference*, pages 514–517, 2004.
- [119] Bart Vermeulen and Sandeep Kumar Goel. Design for Debug: Catching Design Errors in Digital Chips. *IEEE Design and Test of Computers*, 19(3):35–43, May 2002.
- [120] Alexander Weiss and Christian Hochberger. A New Approach for Capturing Trace Data from SoCs. In *Proceedings of the IEEE International Workshop on Silicon Debug and Diagnosis*, pages 45–52, 2008.
- [121] Burnell G. West. At-Speed Structural Test. In *Proceedings of the IEEE International Test Conference*, pages 795–800, 1999.
- [122] Xilinx Verification Tool. ChipScope Pro, 2006. URL <http://www.xilinx.com>.

- [123] David Yeh, Li-Shiuan Peh, Shekhar Borkar, John Darringer, Anant Agarwal, and Wen-mei Hwu. Thousand-Core Chips. *IEEE Design and Test of Computers*, 25(3):272–278, 2008.

Index

- application specific integrated circuit (ASIC), 143, 145
 - 14, 23, 34, 35, 45, 47, 52
- assertions, 4, 35–37, 43, 124
- automatic test equipment (ATE), 5, 12, 62, 78, 100, 103
- automatic test pattern generation (ATPG), 6, 40, 78, 90–93, 107
- backward equations, 80, 83, 84, 101
- backward justification, 40, 77, 83, 84, 98, 104
- bitwise parallelism, iv, 77–79, 86, 96, 114, 149
- breakpoint, 16, 21
- breakpoint conditions, 13
- centralized trace buffer, 15, 43, 46
- circuit bugs, 9, 10
- circuit topology only metric, 97, 112, 113, 115
- circuit-under-debug (CUD), 13–15, 23, 24, 26, 29, 31, 34–37, 39, 41, 42, 46, 95, 97, 112, 118, 149, 150
- circuit-under-test (CUT), 5, 6, 17
- clock control, 16
- communication fabric, 126, 127, 139–141, 143, 145
- complementary metal-oxide semiconductor (CMOS), 3
- computer-aided design (CAD), v, 3
- core-based SOCs, 15, 150
- data acquisition, iii, iv, 10, 15, 16, 21, 23, 26, 28, 30, 31, 34–36, 45, 46, 48, 55, 57, 74, 75, 114, 121, 122, 126–128, 131, 134, 137, 139, 140, 142, 145, 146, 149, 150
- data samples, 125, 126
- data sampling, 9, 13, 21, 28, 31, 45, 122, 130, 131
- debug, iii, iv, vii, 9, 11, 12, 14, 21–23, 26, 34, 36, 39, 43, 45, 47, 52, 53, 55, 57, 69, 71, 74, 75, 79, 84, 86, 88, 90, 91, 95, 97, 101, 108, 112, 114, 118–122, 124–129, 131, 132, 134, 136, 137, 141, 142, 145–152
- debug agent (DA), 45
- debug flow, 25, 26, 47
- design abstraction, 2, 4, 5, 9
- design flow, 1, 3, 149
- design-for-debug (DFD), 9–12, 15, 16, 43, 46, 98, 120, 126, 137, 139–142,

- 145, 147, 148, 150
- design-for-test (DFT), 6, 11
- deterministic replay, 12
- diagnosis, 5, 11
- distributed embedded logic analyzers, 15, 119, 120
- distributed trace buffer, 42, 45, 46, 127, 130, 147, 150
- embedded logic analyzer (ELA), 13–15, 24–29, 34, 36, 38, 42, 43, 46–48, 95, 148, 149, 151
- equality units, 52, 55, 57, 62, 64–67, 69, 71–73
- event buffers, 57, 62, 65–67, 69, 70, 72, 73
- event flags, 28
- event registers, 25, 27, 28, 32, 34, 48, 52, 53, 55, 62
- event sequencers, 25
- event sequences, 28, 66
- false trigger analysis, 68, 149
- field programmable gate array (FPGA), 14, 23, 34, 35, 47
- finite state machines (FSMs), 24, 34, 35, 55, 57, 66, 73, 128, 130, 131, 136, 139, 141, 143
- first-in first-out (FIFO), 38
- flip-flops (FFs), 6, 39, 42, 62–64, 67, 70–72, 76, 85, 90, 99, 101, 106
- formal verification, 4, 5
- forward equations, 83, 101
- forward propagation, 77, 83, 98, 104
- forward restorability, 98, 99, 101
- functional test, 5, 6
- intellectual property (IP), 36
- JTAG, 24, 43, 139
- Karnaugh Map (K-map), 63, 80, 81
- level sensitive trigger events, 26–28, 55, 66, 67
- logic bugs, iii, 9, 10, 14, 20, 23, 26, 95
- manufacturing test, 5–7, 11, 12, 16, 107
- microprocessors, iv, 14, 21–23, 38, 101, 102, 121
- multiple inputs signature register (MISR), 38
- Network-on-a-Chip (NOC), 45
- non-deterministic replay, 12
- OFF-primes, 57, 62, 64, 67, 68
- offload unit, 24–26, 34
- on-chip instrumentation (OCI), 45
- ON-primes, 57, 62–64, 66, 67
- open verification library (OVL), 36
- out-of-order offloading, 126, 137
- over-triggering, 55, 57, 62
- overlapping trigger requests, 124
- post-silicon validation, iii, 7, 9–16, 20, 21, 23–26, 34–39, 42, 43, 46–48, 52,

- 55, 57, 69, 74, 75, 92, 95, 118–121, 124, 126–128, 130, 131, 137, 139, 147–152
- pre-silicon verification, iii, 4, 5, 7, 9, 11, 36, 120, 121, 128, 137, 139, 147
- programmable priority, 126, 134, 136, 145, 146
- programmable trigger engine (PTE), 35, 124
- programmable trigger unit, iii, 68, 74
- property specification language (PSL), 36
- quantified boolean formulation (QBF), 64
- queues, 129–131, 136, 140
- register-transfer level (RTL), 2–4, 7
- restorability values, 99, 101–106, 112–114, 117
- sample before triggering, 26, 29, 124
- sample requests, 43, 119, 120, 122, 124, 126–131, 134
- sample unit, 24–26, 30, 34, 36–38, 43
- scan chain-based technique, 9, 13, 16, 21, 39, 42, 151
- scan chains, 6, 7, 9, 12, 13, 15–17, 20, 21
- scan-flip-flops (SFFs), 6, 16, 17
- segmentation of data, 26, 32, 134
- serializers, 25
- simultaneous trigger requests, 124
- state coverage, 5
- state restoration, 75–79, 84–86, 88–92, 95–99, 101, 104, 108, 114, 117, 118, 149–151
- streaming of data, 26, 33, 120, 121
- structural test, 5
- system bugs, 10
- System Verilog, 2, 36
- System-on-a-Chip (SOC), 7, 9, 10, 13–15, 23, 42, 43, 45, 46, 119, 120, 124, 140, 141, 150, 151
- SystemC, 2, 36
- test, v, vii, viii, 4–6, 16, 17, 23, 36, 79, 90–92
- topology + logic metric, 106
- trace buffer, iii, iv, 13–15, 21–23, 25, 26, 28, 30, 32–34, 37–39, 42, 43, 45, 46, 55, 57, 62, 68, 75, 84–86, 89, 95, 97, 112–114, 118–122, 124–132, 134, 137, 139–143, 145, 148–151
- trace buffer-based technique, 151
- trace compression, 23, 38, 39, 45, 46, 150
- trace filtering, 38, 45, 150
- trace ports, iv, 15, 26, 43, 45, 120–122, 124, 126, 127, 131, 139–143, 145, 146, 150
- trace signal selection, 15, 88, 90, 95, 98, 101, 105, 108, 112–115, 151
- trace signals, iii, iv, 15, 25, 26, 38, 39, 46, 75, 85, 86, 88, 89, 91, 92, 95, 97–99, 101, 102, 104–106, 112–114, 117, 118, 149–151
- trigger analysis, 14, 55, 57, 62, 65, 67–69, 72, 73

trigger conditions, 13–15, 25, 26, 29, 30,
34, 35, 46, 68–72, 74, 149

trigger event mapping, 47, 48, 64, 66, 151

trigger events, iii, 14, 26, 28, 31–34, 43,
47, 48, 52, 53, 55, 57, 62–67, 69–
73, 122, 124, 130, 148–150

trigger signals, 14, 25–28, 32, 35, 48, 55,
62, 64–66

trigger unit, iii, 13, 14, 24–27, 34, 35, 46–
48, 52, 53, 55, 62, 64–67, 149–151

under-triggering, 52, 53

verification, vii, 1, 3–5, 10, 36

Verilog HDL, 2, 36

very large scale integrated (VLSI) circuits,
1, 3–7, 10, 11

VHDL, 2, 36