Doctoral Dissertation
Doctoral Program in Control and Computer Engineering ($37^{th}$cycle)

# System-Level Test techniques for Automotive SoCs

By

**Francesco Angione**

******

**Supervisor(s):**
Prof. Paolo Bernardi, Supervisor
Prof. Riccardo Cantoro, Co-Supervisor

**Doctoral Examination Committee:**
Referee, Prof. Leticia M. Bolzani Poehls, Leibniz Institute for High Performance Microelectronics, Germany,
Referee, Prof. Artur Jutman, Tallinn University of Technology, Estonia
Prof. Maria Michael, University of Cyprus, Cyprus
Prof. Adit Singh, Auburn University, Alabama
Prof. Maurizio Rebaudengo, Politecnico di Torino, Italy

Politecnico di Torino
2025

# Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Francesco Angione
2025

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

*To whom love me,*

Even when I do not deserve your love,

Thank you.

I love you back.

# Acknowledgements

*Je so' pazzo, je so' pazzo*
*e vogli'essere chi vogli'io*
*ascite fora d'a casa mia.*

Pino Daniele

In my humble opinion, I think this is the hardest part of the thesis to write. Conventionally, I would start by acknowledging the people who allowed me to write this PhD thesis. However, let me start in a different way: as I always do, I will navigate upstream.

These are the acknowledgments of a survivor—a survivor of a flawed system (hopefully in good faith); a survivor who was told (by some teachers!) since middle school that he would never be good at speaking English, that he would never become an engineer because he did not know multiplication by heart (well, I still do not know them today, but I cannot occupy my mind with information I can find in a book), and that he was not good enough to start university or, even less, to pursue a PhD.

Well, I am still here.

And after all that, I survived. Shine on, you crazy diamond.

I would like to heartfully thank:

The people I went through in Torino, Fogna, and Gothenburg, thanks to all the experiences together, I always learned a bit from each one of you.

To my old Italian and English teachers, for believing in me since the beginning and not letting me down when I was not even able to write in Italian!

LAB3, for being a welcome, and sometimes drunk and noisy, place even in the darkest hours.

Paolo, for the fun, the wise smoky discussions, the trust, and the friendship.

"Paprikati rosa", for being a relief valve during the PhD.

"Via Massena", for being a shameless family for a brief chapter of our lives.

"I fugnisi r m#*!#", for all the sleepless nights, parties, and fun.

"Casa Toppino," for founding a brotherhood.

Those who are not here and who left by their decision, for leaving me a piece of experience and wisdom.

Eleonora, for accepting, understanding, supporting, and not giving up on me when I am lost in my mind's meanders.

My family, uncles, and cousins, my holding foundation, my pillar, the building foundation of the man I am today, a messy out-of-perspective nest I will always be grateful for not letting me down even when I deserved it. I am You.

My feelings, for remembering me I am human.

I am the person that I am today thanks to whoever crossed my path, in good and bad, and the experiences shared; I would not have been here without you coming through, on and between my path.

I am not a genius, as everybody probably knows, I have never been and I never will, I do not have that sparkling, intuition, the talent like people like Leonardo Da Vinci. I am curious like a kid, curious about the things that surround me, curious about having a different perspective of what is here, and what will be. Unfortunately, or luckily, talent can be compensated with obsession, the same obsession that keeps you awake at night, the one that hits your head when wakes you up in the morning and goes to bed at night. The same obsession that drives you to strive and struggle to understand, to dive deep, to listen, and to let a problem become part of yourself, when it is something you love. It lets you keep trying and trying until you are exhausted. And in the darkest moment, when you are lost and everything seems hopeless, a small, warm light of hope appears, and you embrace it with all your strength.

And you fail again.

But everything can be fixed, only death is unfixable, as Totò would say.

I would not have been here without my failures, on which I built myself.

A‘ fin, u‘ viecch a‘nuvant’ann ancór s’avija mbàrà.

Probably, the last few words to share are about the land that shaped me. The same land that causes you ’appocundria’ when you are arriving, leaving, and missing it. The land that welcomes you every time you lose your way, "u’ ciliendo," the lonely, isolated, unexplored, hidden place in the south of Italy (not the Shire, but very close). The same desolate land to which "i’ cilendani" are so attached that they would trade all the gold in the world to come back, even for one day. "U’ ciliendo" is the land of forgotten memories, the land of slow life, the land of heritage, breathtaking landscapes and people.

Conversely, "brànt s’nàsc e brànt s’mór".

# Abstract

The increasing complexity and heterogeneity of System-on-Chip (SoC) designs, particularly in the automotive domain, pose significant challenges to achieving comprehensive fault coverage during manufacturing testing. While effective at a component level, traditional structural testing methods, such as Scan-Based Testing and Built-In Self-Test (BIST), often fail to fully address faults arising from system-level interactions, particularly in safety-critical applications compliant with ISO 26262 quality standards. In addition to the structural tests, the manufacturing test flow has been enhanced by incorporating a functional-based test known as *System-Level Test*.

The research explores different key contributions at different levels: (1) A stress-optimization methodology for SLT that complements structural stress methods in critical non-uniform stressed areas; (2) guidelines for developing an SLT suite for testing communication peripherals; (3) the development of automated SLT workload generation techniques leveraging graph-based SoC abstractions and Device Tree Source (DTS) files, reducing dependency on manual efforts or through genetic frameworks; (4) Grading methodologies for evaluating SLT effectiveness, using high-level metrics derived from instruction traces, enabling early-stage feedback without requiring exhaustive fault simulation. The proposed methodologies are validated on a 40*nm* automotive SoC case study with three 32-bit cores and approximately 20 million logic gates by using for some experimental setup a low-cost, FPGA-based modular tester capable of executing structural tests, functional test programs, and advanced protocol-based communication tests. Experimental results underscore the potential of SLT suites to significantly improve the quality and reliability of automotive SoCs.

These contributions collectively advance SLT methodologies, offering scalable, automated, practical, and effective solutions to meet the stringent quality and reliabil-

ity requirements of modern automotive SoCs, paving the way for broader applications beyond automotive domains, such as high-end processors in data center fleets.

# Contents

**List of Figures** **xv**

**List of Tables** **xx**

**1 Introduction** **1**

1.1 The raising need for System-Level Test . . . . . . . . . . . . . . . . 2

1.2 Structural test weaknesses . . . . . . . . . . . . . . . . . . . . . . . 4

1.2.1 Enabling stress for SLT phase . . . . . . . . . . . . . . . . . 6

1.2.2 Communication peripherals . . . . . . . . . . . . . . . . . . 6

1.2.3 Uncore logic . . . . . . . . . . . . . . . . . . . . . . . . . . 8

1.3 Automatic generation of System-Level Test workload . . . . . . . . 9

1.4 The Achilles' Heel of System-Level Test: Grading and Quality Metrics 10

1.5 The case study: An Automotive System-on-Chip . . . . . . . . . . 11

**2 Background** **13**

2.1 Manufacturing Test flow . . . . . . . . . . . . . . . . . . . . . . . . 13

2.1.1 System-Level Test . . . . . . . . . . . . . . . . . . . . . . . 14

2.2 System-Level Test State-of-the-art . . . . . . . . . . . . . . . . . . 17

2.2.1 Enabling stress for SLT phase . . . . . . . . . . . . . . . . . 18

2.2.2 State-of-the-art for communication peripherals testing . . . 19

2.2.3 State-of-the-art for uncore logic/crossbars testing . . . . . . 20

2.3 State-of-the-art for System-Level Test workload generation . . . . . 21
2.4 State-of-the-art for System-Level Test grading . . . . . . . . . . . . 22

**3 Main Contributions 24**
3.1 Author's Publication List . . . . . . . . . . . . . . . . . . . . . . . . 29

**4 System-Level Test Techniques 31**
4.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.2 Covering structural weaknesses . . . . . . . . . . . . . . . . . . . . 31
4.2.1 Enabling stress for SLT phase . . . . . . . . . . . . . . . . . 31
4.2.1.1 Basic Working Principles . . . . . . . . . . . . . 33
4.2.1.2 Architecture-independent methodology . . . . . . 36
4.2.1.3 Analytical Formulas . . . . . . . . . . . . . . . . 40
4.2.2 Communication peripherals . . . . . . . . . . . . . . . . . . 41
4.2.2.1 Structural Test Weaknesses Analysis . . . . . . . 43
4.2.2.2 SLT functional program suite generation . . . . . 44
4.2.2.2.1 Embedded Memory access ports (A) . . 45
4.2.2.2.2 Interfaces to other SoC components (B) 46
4.2.2.2.3 Transmission/Reception Interfaces (C) . 47
4.2.2.2.4 Detection and correction unit (D) . . . . 48
4.2.2.2.5 Complex hardware-software functions (E) 49
4.2.2.3 Companion module . . . . . . . . . . . . . . . . 50
4.2.3 Uncore logic . . . . . . . . . . . . . . . . . . . . . . . . . . 52
4.2.3.1 Proposed System-on-Chip model . . . . . . . . . 54
4.2.3.2 Grind Nexum algorithm . . . . . . . . . . . . . . 57
4.2.3.2.1 Creating the model . . . . . . . . . . . 57
4.2.3.2.2 Visiting the model . . . . . . . . . . . . 60

4.2.3.3 The System-Software Test Library . . . . . . . . 64
4.2.3.4 Assessment on structural weaknesses . . . . . . . 66
4.3 Automatic generation of System-Level Test workloads . . . . . . . 68
4.3.1 Leveraging High-Level Models . . . . . . . . . . . . . . . . 69
4.3.2 Leveraging non-functional properties . . . . . . . . . . . . 69
4.3.2.1 Microarchitectural Simulation . . . . . . . . . . . 71
4.3.2.2 Tuning the Initial Register Content . . . . . . . . 72
4.4 Assessment . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
4.4.1 Simulation-based . . . . . . . . . . . . . . . . . . . . . . 74
4.4.1.1 Logic Simulation: Toggle coverage analysis . . . 74
4.4.1.2 Fault Simulation . . . . . . . . . . . . . . . . . . 76
4.4.2 Indirect methodologies: Execution trace analysis . . . . . . 77
4.4.2.1 The Connectivity Algorithms . . . . . . . . . . . 81
4.4.2.2 Graph-based Metrics: The Connectivity Metrics . 88
4.4.2.3 Weaknesses identification in the source code . . . 91
4.4.2.3.1 Traceback From Assembly to Source Code 92
4.4.2.3.2 Shadowed Instruction Analysis . . . . . 92
4.4.2.4 Multi processor connectivity computation . . . . 93

**5 Experimental results 95**

5.1 Experimental setup . . . . . . . . . . . . . . . . . . . . . . . 95
5.1.1 FPGA-based Tester . . . . . . . . . . . . . . . . . . . . . 97
5.2 Simulation-based experimental results . . . . . . . . . . . . . . . 100
5.2.1 Toggle coverage analysis . . . . . . . . . . . . . . . . . . . 100
5.2.1.1 Enabling stress for SLT phase . . . . . . . . . . . 100
5.2.1.1.1 Toggle Coverage results . . . . . . . . . 100
5.2.1.1.2 Toggle activity on Memory Busses . . . 101

5.2.1.1.3 Analytical Evaluation . . . . . . . . . . . 102
5.2.1.2 Final remarks . . . . . . . . . . . . . . . . . . . . 103
5.2.2 Fault Simulation . . . . . . . . . . . . . . . . . . . . . . . . . 105
5.2.2.1 Communication peripherals . . . . . . . . . . . . 105
5.2.2.2 Uncore logic . . . . . . . . . . . . . . . . . . . . . 110
5.2.2.2.1 The functional SLT suite . . . . . . . . . 111
5.2.2.2.2 Effects on the crossbar logic . . . . . . 113
5.2.2.2.3 Effects on the un-core logic . . . . . . . 115
5.2.2.3 Final remarks . . . . . . . . . . . . . . . . . . . . 117
5.3 Indirect methodologies experimental results . . . . . . . . . . . . . 122
5.3.1 Execution trace analysis . . . . . . . . . . . . . . . . . . . . 122
5.3.1.1 Evaluation of Functional Test Programs . . . . . 122
5.3.1.2 Developing New Functional Test Programs . . . . 126
5.3.1.3 Multicore Connectivity analysis . . . . . . . . . . 129
5.3.1.4 Final remarks . . . . . . . . . . . . . . . . . . . . 131
5.3.2 Generating power hungry System-Level Test workloads . . 132
5.3.2.1 Experimental setup . . . . . . . . . . . . . . . . . 133
5.3.2.2 Two Step Generation . . . . . . . . . . . . . . . . 134
5.3.2.2.1 Random Evaluator . . . . . . . . . . . 134
5.3.2.2.2 Microarchitectural Simulation . . . . . 135
5.3.2.3 Final remarks . . . . . . . . . . . . . . . . . . . . 136

**6 Conclusions 137**

6.1 Beyond Automotive . . . . . . . . . . . . . . . . . . . . . . . . . . 139

**References 141**

**Appendix A The toolchain for evaluating Burn-In related stress metrics 153**

A.1 State-of-the-art tools for BI metrics . . . . . . . . . . . . . . . . . . 153
A.2 The proposed Toolchain . . . . . . . . . . . . . . . . . . . . . . . . . 154
A.2.1 Tool A: The Logic Simulator . . . . . . . . . . . . . . . . . . 156
A.2.2 Tool B: The VCD File Analyzer . . . . . . . . . . . . . . . . 158
A.2.2.1 The VCD File Format . . . . . . . . . . . . . . . 158
A.2.2.2 The VCD File Analyzer . . . . . . . . . . . . . . 159
A.2.3 Tool C: Layout-Aware Elaboration Scripts . . . . . . . . . . 162
A.2.3.1 Virtual Node Elimination . . . . . . . . . . . . . 163
A.2.3.2 Topology Analysis . . . . . . . . . . . . . . . . . 164
A.2.3.3 Metrics Weighting . . . . . . . . . . . . . . . . . 166
A.2.4 Tool D: The Set Covering Tool . . . . . . . . . . . . . . . . . 166
A.2.5 Tool E: The Hierarchical Analysis Tool . . . . . . . . . . . 168
A.2.6 Tool F: The Chip-Surface Stress Plotter . . . . . . . . . . . 170

**Appendix B Trace Analysis** **172**
B.1 State-of-the-art for functional test programs evaluation . . . . . . . 172
B.2 The Trace Analysis Methodology . . . . . . . . . . . . . . . . . . . . 174
B.2.1 The Basic Algorithm . . . . . . . . . . . . . . . . . . . . . . 179
B.2.2 Optimized Algorithm . . . . . . . . . . . . . . . . . . . . . . 182
B.2.3 Load/Store Instructions . . . . . . . . . . . . . . . . . . . . . 183
B.2.4 Branch Instructions . . . . . . . . . . . . . . . . . . . . . . . 185
B.2.5 Multiple Destination Instructions . . . . . . . . . . . . . . 186

**Appendix C FPGA-based tester** **191**
C.1 State-of-the-art for FPGA-based tester . . . . . . . . . . . . . . . . 191
C.2 Proposed Hardware architecture . . . . . . . . . . . . . . . . . . . . 193
C.2.1 Functional Module . . . . . . . . . . . . . . . . . . . . . . . 194

C.2.2 Structural Module . . . . . . . . . . . . . . . . . . . . . . . . 195
C.3 Proposed Software Framework . . . . . . . . . . . . . . . . . . . . . . 196
C.4 Enhancing Modularity . . . . . . . . . . . . . . . . . . . . . . . . . 197
C.5 Final Considerations . . . . . . . . . . . . . . . . . . . . . . . . . 198

# List of Figures


1.1 Fault universe with detected faults from different test strategies. . . 3
1.2 Cross domain faults example. . . . . . . . . . . . . . . . . . . . . . . 5
1.3 ATPG fault excitement and propagation. . . . . . . . . . . . . . . . . 5
1.4 Detail of a Generic Communication Peripheral [1]. . . . . . . . . . . 7
1.5 SoC Example Architecture [2]. . . . . . . . . . . . . . . . . . . . . . 9
1.6 Evaluation board for the case study and its physical layout. . . . . . 11

2.1 Manufacturing test flow. . . . . . . . . . . . . . . . . . . . . . . . . 14

4.1 High-level view of the proposed generation approach. . . . . . . . . . 32
4.2 Memory overview. . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
4.3 Address toggle example. . . . . . . . . . . . . . . . . . . . . . . . . 34
4.4 Example of generated code . . . . . . . . . . . . . . . . . . . . . . . 34
4.5 Normal Data Structure vs optimized data structure used for memory operations. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.6 Bypassed register-based DfT for memory collar. . . . . . . . . . . . 36
4.7 Multiplexer isolation-based DfT for memory collar. . . . . . . . . . 37
4.8 Chip select isolation-based DfT for the memory collar. . . . . . . . 37
4.9 Pseudo code of functional routine for architecture where misaligned memory access is illegal. . . . . . . . . . . . . . . . . . . . . . . . 38
4.10 Pseudo code of functional routine for misaligned memory access . . 39

4.11 Pseudo code of optimized functional routine for misaligned memory access . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
4.12 Flow diagram of the proposed methodology. . . . . . . . . . . . . . 42
4.13 Companion module view with a companion memory storing error injection information. . . . . . . . . . . . . . . . . . . . . . . . . 43
4.14 Detail of a Generic communication peripheral. . . . . . . . . . . . . 44
4.15 Proposed methodology workflow. . . . . . . . . . . . . . . . . . . . 53
4.16 Example architecture with N masters (red), M slaves (yellow), and K crossbars interconnected by a cross-lake. . . . . . . . . . . . . . 56
4.17 Generated graph model for the example SoC architecture with two linked crossbars, crossbar master ports (green) and slave ports (blue). 56
4.18 Software modules decomposition of a $S^2$TL. . . . . . . . . . . . . 64
4.19 Proposed approach workflow. . . . . . . . . . . . . . . . . . . . . . 70
4.20 Pipeline example without and with stalls. . . . . . . . . . . . . . . 71
4.21 Stress program structure. . . . . . . . . . . . . . . . . . . . . . . . 72
4.22 Original development flow for SLT workloads. . . . . . . . . . . . 74
4.23 Optimized development flow for SLT workloads. . . . . . . . . . . 74
4.24 The high-level view of the proposed toolchain for BI stress evaluation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
4.25 Data partitioning for SLT workloads before the fault simulation campaigns. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
4.26 Workflow for the Trace analysis methodology. . . . . . . . . . . . 79
4.27 The CDFG of the code snippet of Table B.3 is represented in Figure (a). Figure (b) illustrates the vertex-coloring process obtained following WAW edges. Figure (c) shows the colors obtained at the end of the RAW visit. . . . . . . . . . . . . . . . . . . . . . . . . 82
4.28 An example in which our algorithm is applied to a code section includes arithmetic operations and load/store instructions. . . . . . 85
4.29 The graph example includes a branch instruction. . . . . . . . . . . 86

4.30 Another branch instructions example. . . . . . . . . . . . . . . . . . 87

4.31 Fault injection in branch-control registers for undecided branches to improve the representativity of the execution trace, . . . . . . . . . 88

4.32 Weaknesses identification flow. . . . . . . . . . . . . . . . . . . . . 91

4.33 Single and multicore connectivity. . . . . . . . . . . . . . . . . . . . 94

5.1 Experimental setup. . . . . . . . . . . . . . . . . . . . . . . . . . . 96

5.2 Experimental setup with Xilinx ZCU 104 evaluation board. . . . . . 97

5.3 Experimental setup with Xilinx ZCU 104 evaluation board and peripheral bus (CAN). . . . . . . . . . . . . . . . . . . . . . . . . . 99

5.4 Chip select isolation-based DfT for the memory collar. . . . . . . . 100

5.5 Bit-by-bit comparison between the toggle activity achieved by the functional and BIST approaches. . . . . . . . . . . . . . . . . . . . 102

5.6 Visualization of the overall stress provided by the superimposition of all stress patterns. . . . . . . . . . . . . . . . . . . . . . . . . . 105

5.7 Stress-colored heatmaps in terms of toggle coverage with detailed zoom on some regions. Figure 5.7a: 32 scan based ATPG patterns, 5.7b: 12 Selective ATPG patterns; 5.7c: 1024 Scan based pseudo-random patterns; 5.7d: LBIST patterns; 5.7e: MBIST patterns; 5.7f: RTOS boot. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106

5.8 Layout view of the four CAN controllers in the CAN peripheral of the DUT. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107

5.9 Venn diagrams between Structural, LBIST, and functional (SLT) test approaches. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108

5.10 A qualitative contrast between the SoC Architecture from the User Manual with the SoC Graph model. . . . . . . . . . . . . . . . . . 110

5.11 Classification of Undetected faults from structural tests for different fault models. All the values illustrated are shown in percentages. . . 111

5.12 Classification of Undetected faults from structural tests and SLT covered faults. All the values illustrated are shown in percentages. . 115

5.13 $S^2TL$ Test Scheduler. . . . . . . . . . . . . . . . . . . . . . . . . 119

5.14 Evolutionary generation of SBST for the multiplier unit. . . . . . . 127

5.15 Evolutionary generation of SBST for the integer divider unit. . . . 128

5.16 Experimental setup for power consumption measurements. . . . . . 133

5.17 Average, Best, and Worst fitness values per generation of Baseline (dashed line) and register data tuning for the two-step generation based on microarchitectural simulation. . . . . . . . . . . . . . . . 136

A.1 Possible evaluation flows for stress metrics. . . . . . . . . . . . . . 153

A.2 The picture represents a high-level view of the proposed toolchain. The stress pattern, either structural or functional, is validated on the silicon implementation of the SoC. Afterwards, a logic simulation based on the given stress pattern is performed to provide a VCD file. This file is then analyzed to provide a single or multi-point stress coverage for the stress pattern. The Layout-Aware elaboration links the stress pattern to the SoC layout by weighting the stress metrics with the gate density. There are several options for the final step, from plotting the stress over a layout heatmap to superimposing different stress patterns or presenting stress coverage metrics for each module. . . . . . . . . . . . . . . . . . . . . . . . . . 155

A.3 The VCD file format. . . . . . . . . . . . . . . . . . . . . . . . . 159

A.4 The different stages of the pipeline: [D]iscovery, [R]eading, [P]arsing, [E]laboration, [W]rite-Back. In a realistic scenario, some [S]talls might be present. . . . . . . . . . . . . . . . . . . . . . . . . . 160

A.5 Layout-Aware elaboration substeps. . . . . . . . . . . . . . . . . . 162

A.6 An example of the filtering method result. . . . . . . . . . . . . . . 163

A.7 An example of the clustering method (DBSCAN) on layout front-end. 165

A.8 Rising and Falling transition set cover: From file to set interacts. . . 168

A.9 A high-level view of the tree data structure containing the coverage for each node. . . . . . . . . . . . . . . . . . . . . . . . . . . . 169

A.10 An example of stress heatmap over a generic SoC layout. . . . . . . 171

B.1 The logic flow of our testing framework. . . . . . . . . . . . . . . . 175
B.2 A graphical representation of Example 6. . . . . . . . . . . . . . . . 178
B.3 The CDFG of the code snippet of Table B.3 in represented in Figure (a). Figure (b) illustrates the vertex-coloring process obtained following WAW edges, and Figure (c) the colors obtained at the end of the RAW visit. . . . . . . . . . . . . . . . . . . . . . . . . . . 180
B.4 An example including arithmetic operations and load/store instructions. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 184
B.5 A graph example including a branch instruction. . . . . . . . . . . 186
B.6 Another branch instructions example. . . . . . . . . . . . . . . . . 187
B.7 A code snippet with instructions with multiple destinations: Our instruction-oriented analysis is modified to be destination-based. . . 188

C.1 FPGA-based Tester Architecture. . . . . . . . . . . . . . . . . . . . 193
C.2 Functional Module Block Design . . . . . . . . . . . . . . . . . . . 194
C.3 Structural Module Block Design . . . . . . . . . . . . . . . . . . . 195
C.4 Software stack . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 196
C.5 Experimental setup for SLT of communication peripherals (CAN). . 198
C.6 Experimental setup with power controller board. . . . . . . . . . . . 199
C.7 Experimental setup for power measurements. . . . . . . . . . . . . 200

# List of Tables


1.1 Structural tests coverages, number of patterns, and test escapes. . . . 11

2.1 Comparison between different test approaches for communication peripherals testing. . . . . . . . . . . . . . . . . . . . . . . . . . 19

2.2 Comparison between different test approaches for crossbar testing. . 20

4.1 Transition on busses for single body loop execution. . . . . . . . . . 39

4.2 Comparison of different possible evaluation methods for SLT workload. 73

4.3 An example of *RiscV* code segment with the operation performed. . 78

4.4 Instruction sequence, source, and destination operands. . . . . . . . 81

4.5 Instruction sequence, source, and destination operands. . . . . . . . 84

4.6 Instruction sequence, source, and destination operands. . . . . . . . 85

4.7 Instruction sequence, source, and destination operands. . . . . . . . 87

5.1 Structural tests coverages, number of patterns, and test escapes. . . . 96

5.2 Incremental toggle coverage per memory collar. . . . . . . . . . . . 101

5.3 Analytical evaluation by approach. . . . . . . . . . . . . . . . . . 102

5.4 Characteristics of SLT suite for CAN peripheral. . . . . . . . . . . . 107

5.5 Fault coverage for Stuck-at fault model (435,967 faults) and Transition delay fault model (435,966 faults). . . . . . . . . . . . . . . . . 109

5.6 Generated SLT workload suite and RTOS characteristics targeting the crossbar module. . . . . . . . . . . . . . . . . . . . . . . . . . 112

5.7 Fault coverage for Stuck-at fault model (ca. 270k faults). . . . . . . 114

5.8 Fault coverage for Transition Delay fault model (ca. 270k faults). . . 114

5.9 Effects of SLT suite on different on-chip modules for SAF model. . 116

5.10 Effects of SLT suite on different on-chip modules for TDF model. . 116

5.11 Original SLT workload suite. . . . . . . . . . . . . . . . . . . . . . . . 120

5.12 Partitioned SLT workload suite for Crossbar Module. . . . . . . . . 121

5.13 Functional test programs evaluation for benchmark applications: STL for online testing and RTOSs for SLT. . . . . . . . . . . . . . 123

5.14 Connectivity metrics post-fault injection for undecided branches (the average injection time is about 2.0 s). . . . . . . . . . . . . . . . . . 126

5.15 Comparison of different generation strategies for developing the SBST for the divider unit. . . . . . . . . . . . . . . . . . . . . . . . . 129

5.16 Experimental results for different SLT workloads. . . . . . . . . . . 131

5.17 *MicroGP* genetic parameters. . . . . . . . . . . . . . . . . . . . . . . 134

5.18 Comparison of different generation strategies . . . . . . . . . . . . 134

A.1 The table compares our toolchain with state-of-the-art EDA tools. The Comparison is made only from the analysis perspective of stress metrics for BI. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 154

A.2 A simplified view of the coverage file. . . . . . . . . . . . . . . . . . 169

B.1 State-of-the-art methodologies for functional test program assessment for hardware testing of CPU-based SoCs. . . . . . . . . . . . 173

B.2 An example of instruction sequence with the operation performed. . 177

B.3 Instruction sequence, source, and destination operands. . . . . . . . 179

B.4 Instruction sequence, source and destination operands. . . . . . . . 184

B.5 Instruction sequence, source and destination operands. . . . . . . . 185

B.6 Instruction sequence, source and destination operands. . . . . . . . 187

B.7 Instruction sequence with multiple destinations. . . . . . . . . . . . 188

C.1 Comparison between different FPGA-based tester. . . . . . . . . . . . 192

# Chapter 1

# Introduction

*Sapere aude.*

Horace

More and more digital systems are present in everyday life, from the smallest Internet of Things (IoT) application, to Electronic Control Unit (ECU) in cars, to the fleet of processing systems in enterprises or data centers by Cloud Service Providers (CSPs), such as Google or Meta and many others [3–5]. CSPs companies, or Original Equipment Manufacturers (OEM) for automotive, rely heavily on silicon device product quality to guarantee reliability and continuous system availability, especially for the safety-critical applications (i.e., automotive); CSPs and OEMs trust the testing strategies of their semiconductor suppliers (Tier 1), such as Intel, AMD, NVIDIA, STMicroelectronics and many others [6–11]. Nevertheless, the growing demand for computational abilities led to abnormous high density heterogeneously-composed silicon devices [12] and pushed testing capabilities to extremely hard-to-reach coverage objectives [13], an outcome bounded by factors not only related to the transistor density itself but to the computing time for generating test patterns, the Automatic Test Equipment (ATE) memory size limitations which impacts the test cost; and structural related issues introduced by Design-for-Testability boundaries (DfT) [14–16, 1], mainly for reducing test complexity and power-related issues.

Manufactures semiconductor devices undergo a testing flow composed of a combination of structural and functional test approaches for stressing and testing the device under test (DUT) [17, 18]. However, the aforementioned bounding factors

impact the leftover test escapes per generation (or year) of semiconductor devices, e.g., a fault coverage of 99.00 %, independently from the fault model, in absolute numbers with a 20M gates device has approximately 200k possible test escapes; instead, a device with 1B gates results into about 10M test escapes.

From a statistical perspective, the increasing test escapes multiplied with the increasing deployed devices, with high computing workload, temperature-voltage-frequency fluctuations and harsh environments, requires high quality outgoing silicon devices from silicon manufacturers.

The thesis is structured as follows: the following subsections motivate the rising need for System-Level Tests (SLT), their pros and cons, and an analysis of structural test weaknesses. The major Achilles' heel of the System-Level Test is presented: the missing commonly defined high-level quality metric for SLT workloads and its difficulty in grading SLT workloads with mastodontic fault simulations. Lastly, the case study for the thesis work is presented; it is an Automotive System-on-Chip (SoC) manufactured by STMicroelectronics.

Moreover, in Section 2, basic concepts on manufacturing test flow are illustrated, as well as in-depth state-of-the-art analysis for System-Level test; Section 3 describes the main contributions compared to the state-of-the-art; meanwhile Section 4 presents SLT workload development and generation techniques, and possible grading techniques based on indirect or simulation-based methodologies. Section 5 highlights experimental evidence on the case study for the proposed SLT techniques. Section 6 draws some conclusions and future works beyond the Automotive world.

## 1.1 The raising need for System-Level Test

Structural tests, such as those based on Scan Chain and Logic/Memory Built-In Self-Test (BIST) architectures, are compelling automatic approaches for capturing faulty behaviors in Systems-on-Chip (SoCs). Nonetheless, it is known that structural tests may generate test escapes [17, 19], i.e., faults not captured by the structural pattern set may affect devices shipped to market and arise along the chips' mission. The reasons behind this substantial limitation of structural tests are various.

Some generation constraints, such as the non-unlimited time for pattern generation, impact the effectiveness of the Automatic Test Pattern Generator (ATPG)

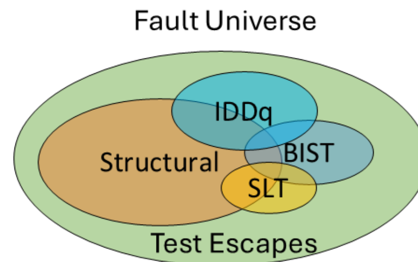



Fig. 1.1: Fault universe with detected faults from different test strategies.

tool. Consequently, the amount and quality of the ATPG-generated patterns are often determined as a balance between the fault coverage and the CPU time.

Automatic Test Equipment (ATE) may also impose extra constraints. ATEs are expensive machinery that apply the test pattern to the SoC during the manufacturing test flow. ATEs are often limited in memory for pattern storage. Hence, ATPG patterns may be further cut out from the final test recipe because of the ATE limits in pattern memory. Current devices show scan chains from hundreds of thousands to millions of flip-flops; in these cases, a single scan pattern occupies hundreds of KB to tens of MB.

Finally, the evaluation of the test cost at large by test engineers may introduce some structural test pattern diseases. For example, if the ATE usage time per chip is contingent and hence limited to a fixed amount of milliseconds or seconds per device, another drop in coverage is highly probable[20].

The combination of the aforementioned factors can result in a significant number of undetected faults, which may ultimately lead to manufacturing test escapes and compromise product reliability, as represented in Figure 1.1.
Therefore, in order to comply with increasing quality requirements, such as *ISO26262*, in the last decade, a new step in manufacturing test flow has been added for filling the leftover gaps of structural tests. The additional test step, called the System-Level Test (SLT), resembles the final application, workload, and environment as much as possible. Therefore, it is a functional test step for exercising the SoC as a whole through functional procedures.

In conclusion, System-Level Testing has become an indispensable part of modern product development and quality assurance, particularly in the semiconductor industry. Its ability to provide comprehensive testing, and ensure product quality

makes it crucial to maintaining competitiveness in today's fast-paced technological landscape. While challenges exist in its implementation, the benefits of SLT far outweigh the difficulties, making it a necessary investment for companies striving for excellence in product quality and reliability.

## 1.2 Structural test weaknesses

Scan-based testing is essential for achieving high fault coverage in large SoCs but presents challenges in the context of ever-growing design complexity. Automatic Test Pattern Generation (ATPG) strategies for scan-based testing in complex designs can be computationally expensive, requiring multi-model incremental generations [21] to reach high fault coverage needed by devices intended for safety-critical sectors.

Although the ATPG seeks to achieve complete fault coverage, many patterns can result in overly broad test patterns, which can configure the circuit in ways not used by the device during its operational mode [22]. This over-testing can increase test time and computational resources [23]. Moreover, in large SoC, ATPG could leave some faults uncovered [17]. Thus, the scan-based pattern set is not always exhaustive.

A possible weakness of scan-based patterns could lay in logic gates that are controlled by scan flops belonging to one clock domain island (or scan chain configuration) and observed by scan flops belonging to another clock domain island (or scan chain configuration). Such a scenario is depicted in Figure 1.2, where the shown gate in yellow is controlled by scan flops belonging to two clock domains that are different from the clock domain where the observer scan flops are. Faults related to gates with the mentioned characteristics could be challenging to test for ATPG in a single run, and they require multiple generations using different combinations of configurations or dedicated strategies [14, 15].

In literature, the algorithms behind the ATPG computation can be varied [24], and typically heuristic-based; one of the most common is the FAN Algorithm [25], which focuses the generation on the gates' fan-out. Its aim is exciting and propagating faults with large input/output logic cones, i.e., it tries to maximize the probability of capturing faults in observable FFs thanks to cascading effects.

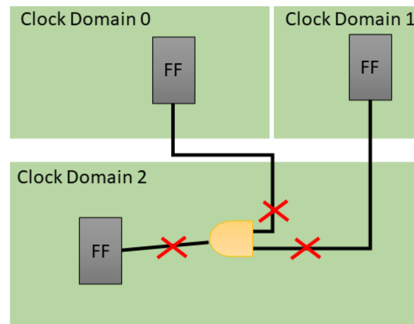



Fig. 1.2: Cross domain faults example.

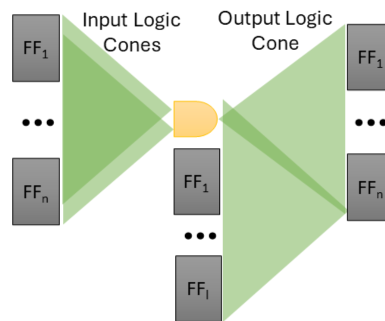



Fig. 1.3: ATPG fault excitement and propagation.

Figure 1.3 represents a possible ATPG scenario. The fault on the gate (in yellow) output must be excited, i.e., the ATPG must create the values to be loaded in the FFs from 1 to $n$. As a second important step, the fault must be observed in the FF of the output logic cone; simultaneously, the ATPG must create the constrained values to be loaded in different FFs to propagate the excited fault to observable FFs. Generally, the output logic cone ends in a high number of FFs, and as a consequence, a fault can be observed as a cascade effect of a different test pattern. On the other hand, if the output logic cone ends into a single observable FF, the ATPG must invest efforts to generate a specific test pattern to excite and propagate a given fault, consequently increasing the computing time and the test pattern set size for a single, specific, fault.

In the last decades, complementary strategies such as functional testing and system-level tests have been used in conjunction with scan-based testing in order to address scan-based weaknesses [18, 17, 19, 11] without any netlist-based structural analysis but instead on empirical observation of failures in faulty devices running in-field like application during production.

### 1.2.1 Enabling stress for SLT phase

To maximize the effectiveness of the SLT step, the previous tests should screen out all failures related to single components. Nevertheless, it is crucial to exacerbate the defectivity of devices engaged in the interaction of system components. Stress generation is a duty of the Burn-In phase, which is supposed to apply a uniform and complete stress to all components of the SoC[26], thus accelerating the evolution into defects of possible weak parts.
Methodologies to improve the Burn-In flow to make the successive test steps more effective in screening out latent defects that could show up in the field are crucial. Complementing, or strengthening, structural stress with functional stress strategies is beneficial thanks to:

- The generated functional stress procedures address the stress of the device more effectively because run-time functionalities are naturally activated.
- The number of transitions per signal provoked by structural methods can be unbalanced (i.e., some signals are stressed more and others less in the same circuit portion, as it happens for the most significant bits in the memory addresses for Memory Built-In Self-Test).
- Different Design for Testability (DfT) architectures can introduce different paths to/from memories or modules, potentially leading to uncovered gates.

### 1.2.2 Communication peripherals

Modern automotive System-on-Chips (SoCs) are equipped with several communication peripherals to facilitate communication between SoCs in vehicles. However, communication with external devices may not be possible during manufacturing tests, which could leave some portions of the logic untested. Additionally, limitations in Automatic Test Pattern Generation (ATPG) may result in pieces of untested logic at the boundaries of modules. Intuitively, weaknesses in communication peripherals can be represented as shown in Figure 4.14.
Arrows and labels are added to the sub-module boxes that can be easily extracted from the netlist. In the resulting visualization, the colors of the arrows indicate the criticality level of specific interactions among modules (e.g., dark colors represent the

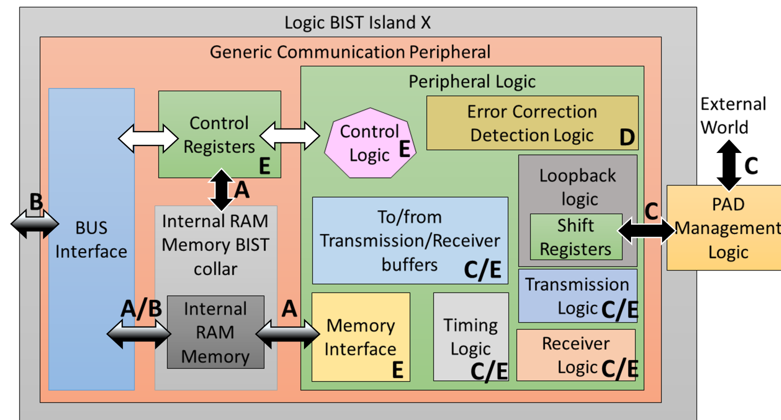



Fig. 1.4: Detail of a Generic Communication Peripheral [1].

most critical interactions, while white indicates low risk), and the alphabetic labels group sub-modules according to their functionality (e.g., all functionalities of the internal RAM are classified under label "A"). Therefore, when developing System-Level Test (SLT) functional programs oriented toward communication peripherals, the following meaningful areas should be targeted:

A) Embedded memory access ports: These may not be completely covered during structural tests due to collars and memory Design-for-Test (DfT) circuits like MBIST.

B) Interfaces to other on-chip components: These may be included in different LBIST or Scan Chain islands, which can introduce testability issues.

C) Transmission/Reception Interfaces to Chip Top: Some signals and pins to and from outside the SoC may never be exercised during manufacturing tests.

D) Detection and correction logic circuits: These usually include large logic functions, resulting in deep circuits that are difficult to target with structural tests.

E) Complex hardware-software functions: Functions such as complex protocol operations and synchronization mechanisms are often not exercised.

Thus, an effective SLT suite for communication peripherals should consist of a combination of programs capable of systematically addressing all the aforementioned considerations.

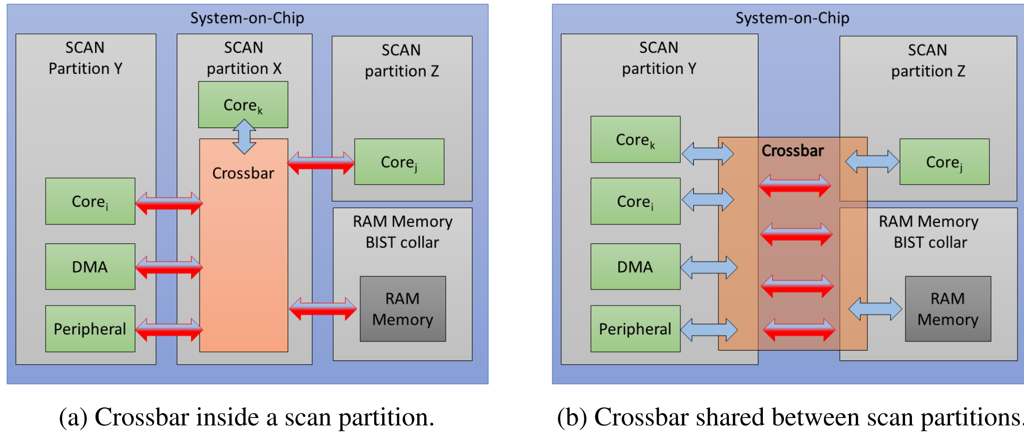
### 1.2.3 Uncore logic

The Design for Testability (DfT) architecture and configuration may introduce additional issues in achieving the desired fault coverage. As the circuit becomes too large to test in one go (e.g., for power reasons), the SoC is often partitioned into multiple groups of modules. Many different scan chains are separately inserted in each group of modules, or distinct Logic Built-In Self-Test (LBIST) islands are created to cover subsets of the fault universe. Such partitioning methods enable a power-efficient test application, but they may also introduce structural untestability [14, 15].

Consequently, scan chain or BIST approaches test the interactions of components to some extent [27]. However, they may fail to check the interaction of modules far apart in the circuit topology. For example, a functional communication path between CPU registers and a specific peripheral may not be tested if the CPU and the peripheral belong to structurally different scan or BIST partitions.

The criticality of interconnection logic, often referred to as "uncore" logic, encompasses a combination of crossbar elements, including routing elements, logic bridges, and communication interfaces, which may conceal DfT issues that can be targeted by System-Level Testing (SLT). Holistically speaking, a good test for the uncore logic modules must activate a large number of the SoC modules. A proper SLT mimics the behavior of an OS boot, which sets up the system by activating the chip's functional units (i.e., CPU, memories, Direct Memory Access (DMA), peripheral cores, etc.). By managing the communication between different on-chip components, an SLT program running on a faulty chip could route compromised data or execute communications incorrectly, such as reaching the wrong modules, corrupting the control signal, or even triggering unintended priority changes when the chip allows complex master/slave bus contention schemes.

The crossbar module can be considered the target module for uncore SLT development due to its criticality regarding the structural test weaknesses illustrated. The crossbar module can reside in a scan partition or an LBIST island, as shown in Figure 1.5.
Figure 1.5 depicts a couple of possible scenarios where the SoC modules are organized into two or three scan partitions. Because the scan partitions operate individually and mutually, e.g., one at a time, it is likely that the glue logic gates pinched between LBIST islands may either not be exercised (driven by signals from an inac-




(a) Crossbar inside a scan partition. (b) Crossbar shared between scan partitions.

Fig. 1.5: SoC Example Architecture [2].

tive island) or their behavior may not be captured because it could only propagate to an inactive island. Similarly, Memory BISTs (MBISTs) may introduce structurally untestable logic between the so-called MBIST collar and the actual RAM array [16]. Figure 1.5a illustrates the case when the crossbar is fully included in a single DfT partition, while Figure 1.5b depicts the case when the crossbar gates are distributed over several DfT partitions. In both cases, DfT partitions may cause some faults to become very hard to detect. Blue/red arrows visualize the intrinsic weaknesses introduced by the DfT partitions. An effective SLT suite for the crossbar is, therefore, a good candidate to cover such shadowed zones of the circuit.

## 1.3 Automatic generation of System-Level Test workload

A crucial aspect of developing an SLT workload is the manual effort of developing and validating the code. It does not exist any automated methodology. A common practice is to start from an already existing application example from the final environment of the SoCs, such as an operating systems (OS) [18]. However, those applications have been developed to achieve a high-level function instead of verifying the correctness of the hardware, and consequently, they may lack effectiveness. Moreover, measuring the complete execution of an OS is impractical in terms of fault simulation time, as the OS operates in an infinite event-driven loop. Consequently,

only the OS bootstrapping process and tasks initialization are measured. The RTOS initializes fewer peripherals and performs fewer operations during its bootstrap phase. However, it may negatively affect the fault coverage achieved by the RTOS in absolute terms. This is because the RTOS only initializes a limited subset of the available SoC peripherals, and its firmware must be manually written.

Therefore, SLT workload development is typically left to the expertise of skilled test engineers who manually assemble the SLT suite or library of programs on iterative efforts.

## 1.4 The Achilles' Heel of System-Level Test: Grading and Quality Metrics

The primary limitation of SLT lies in the difficulty of assessing its fault coverage. The fault simulation process required to evaluate the effectiveness of SLT procedures, such as an operating system boot, can demand substantial CPU resources and extensive time.

Creating functional test programs necessitates manual and/or iterative efforts and is usually assessed using fault simulation tools [28] prior to functional verification. Unfortunately, these fault simulations are exceptionally time-consuming, forcing test engineers to endure long wait times. If the results are unsatisfactory, they must iterate the entire process, even when using automated generation tools [29].

Given the need to exercise the device and the lengthy test execution times, SLT workloads incur incredibly high development costs, often rendering fault simulation campaigns unfeasible. Consequently, this leads to a lack of quality metrics for this class of functional test programs. Indirect methodologies can be employed to guide the development or generation of SLT workloads, ranging from power consumption analysis to trace-based evaluations. Moreover, it is important to note that by correctly generating an SLT workload based on iterative loops of predefined operations, extensive fault simulation campaigns can be parallelized, thereby reducing the overall fault simulation execution time and making feasible the SLT assessment.

## 1.5 The case study: An Automotive System-on-Chip

The case study is a 40 nm Automotive SoC manufactured by STMicroelectronics, compliant with the ISO 26262 ASIL-D standard. This SoC features approximately 20 million logic gates and around 700,000 flip-flops. It has multiple LBIST partitions (or islands) and Scan Chain Domains. The multicore architecture consists of three 32-bit cores that utilize the PowerPC Variable-Length Encoding (VLE) instruction set. Additionally, it includes 6 MB of Flash memory and 128 KB of general-purpose SRAM, along with several peripheral bridges to access a variety of on-chip and off-chip peripherals. For communication between on-chip components, the SoC is interconnected via two fast crossbar switches, AHB-AMBA [30] version 2.0, operating at 64 bits and functioning up to 200 MHz. These two crossbars are linked by a cross-lake, allowing for the connection of one master from one crossbar to two slaves from the other and vice versa. The SoC also supports various communication peripherals, including CAN, LinFlex, SPI, and FlexRay. The evaluation board used for developing the SLT workloads and analysis is shown in Figure 1.6. To

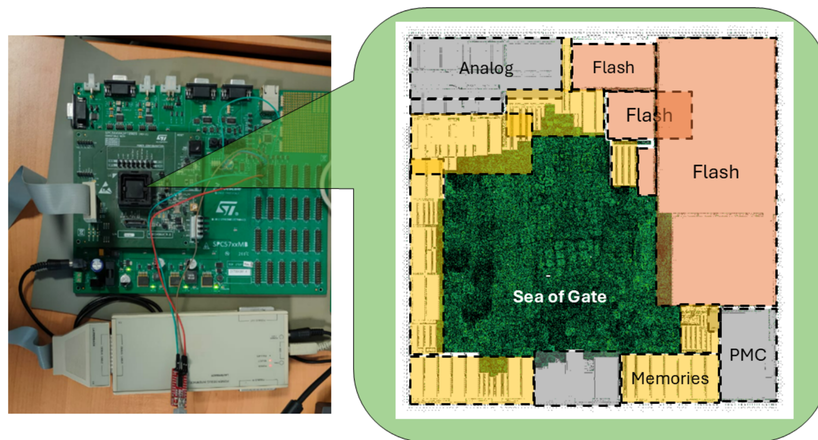



Fig. 1.6: Evaluation board for the case study and its physical layout.

| Fault Model | # Faults | Coverage [%] | # Patterns | # Test Escapes |
|---|---|---|---|---|
| Toggle | ca. 20M | 95.89 | 1k | 822,000 |
| Stuck-at | ca. 40M | 99.21 | 69k | 316k |
| Transition Delay | ca. 40M | 89.65 | 82k | 4M |

Table 1.1: Structural tests coverages, number of patterns, and test escapes.

develop effective SLT workloads and test strategies, information from the RTL,

gate-level netlist, and physical layout of the manufactured automotive SoCs are combined. From manufacturing testing, structural tests (including BISTs and scan-based tests) achieve the coverage levels and corresponding number of patterns depicted in Table 5.1 [20, 31].

Despite the high coverage achieved by ATPG tools for structural tests, the question remains: can SLT further improve these numbers?

# Chapter 2

# Background

*If I have seen further, it is by standing on the shoulders of giants.*

Isaac Newton

## 2.1 Manufacturing Test flow

As transistors scale, the density of transistors per area increases [12]. This significant achievement has led to an exponential rise in the complexity of integrated circuits, making the testing scenario increasingly complex [13, 32]. To address this complexity, structural tests (scan-based) have been introduced to streamline the testing process and enhance the automation of generating test patterns for integrated circuits [33]. Since the introduction of scan-based structural approaches, the generic manufacturing test flow has stabilized, with only minor changes in the order of test steps, which can vary by company, as illustrated in Figure 2.1.

A crucial aspect to highlight is the increased cost associated with discarding faulty devices during the manufacturing test flow. Discarding a device in later test steps incurs a higher cost than doing so in earlier steps. The test flow is divided into different stages, each targeting specific defects with the goal of identifying and discarding faulty devices [34, 18]:

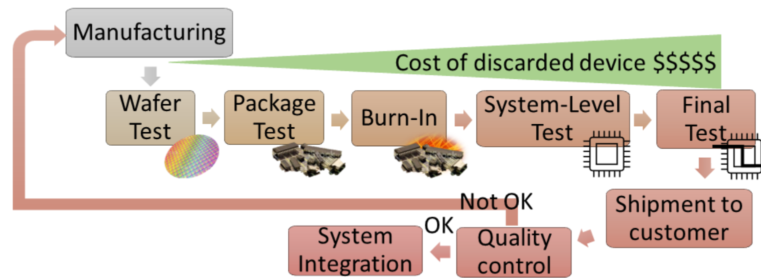



Fig. 2.1: Manufacturing test flow.

- Wafer Test: Conducted at the wafer level, this stage checks the primary electrical functionalities of the chip and executes structural test patterns.
- Package Test: Performed at the package level, this test measures and evaluates the essential electrical characteristics of the pins.
- Burn-In: Primarily for automotive and safety-critical devices, this phase exacerbates latent faults [35, 26]. After the Burn-In phase, devices with weaknesses can be identified in subsequent test steps.
- System-Level Test (SLT): Added as an additional step for safety-critical and automotive devices [17], SLT verifies the correctness of devices through complex functional programs.
- Final Test: This stage detects faults by applying a combination of structural and functional tests [36].

Structural tests cover only a limited spectrum of possible faults and do not account for test escapes from previous steps. Consequently, as the complexity of modern SoCs increases, faults may arise in the field, leading to scenarios where structural tests are re-executed by the manufacturer, resulting in No-Trouble-Found (NTF) outcomes [18].

### 2.1.1 System-Level Test

In the last decade, some companies have begun to introduce an additional test step in the manufacturing test flow, known as System-Level Test (SLT) [11, 19, 10]. Simultaneously, Automatic Test Equipment (ATE) providers have started to support

SLT[37, 38]. SLT has emerged as a critical phase in the semiconductor production cycle, aimed at further reducing test escapes from previous steps for several reasons:

- Increasing Quality Requirements: with more electronic and electrical systems integrated into everyday life, reliability is paramount, especially in safety-critical applications.
- Pushing the Limits of Technology: to meet performance and complexity requirements, aggressive scaling and advanced packaging can introduce new types of defects that traditional test steps may not detect.
- High Number of Leftover Defects: even with a coverage percentage around 95%, the absolute number of untested faults remains significant. For instance, an SoC with 30 million faults and 95% coverage leaves 1.5 million untested faults, likely spread throughout the layout rather than localized.
- Design Verification: modern devices are highly complex, and even after functional verification at the simulation level, gaps in activable functional modes may still exist.
- Quickly Adding Fault Coverage for Escapes: SLT can effectively and rapidly screen faulty devices, reducing the time required to identify the root cause of problems found in structural tests, thus avoiding field returns and overtesting [39].

SLT primarily involves running functional applications that mimic the in-field behavior of the device, including workloads and environmental conditions. A common baseline benchmark is to boot and run a Real-Time Operating System (RTOS) with in-field-like workloads. The SLT phase encompasses various aspects of modern safety-critical devices, as outlined in [11]:

- Hardware/Software Interactions: including clock and power domains [40].
- RTOS Bootstrap Interactions [17].
- Workload-Dependent Aging: errors caused by factors like activity-induced supply voltage droop and cross-talk. When memory is accessed at high rates, these effects are exacerbated, accelerating aging and increasing the likelihood of errors [41].

- Hardware-Implemented Protocols: these protocols cannot be tested through structural tests [42].
- Complex Hardware Resource Management: including dynamic system reconfiguration [42].
- Heavy Workload Exercise: stressing the pipeline of processing units, including architecture-specific operations, floating-point coprocessors, maximum concurrent operations on multiple threads and cores, TLB misses/hits, caches, and their tag fields [42, 43].
- Back-End Assembly Structure Damage: due to aging or thermo-mechanical factors [44].

To effectively screen faulty devices, SLT must incorporate all these aspects, complementing structural tests. It is essential for modern, complex semiconductor devices because scan- and BIST-based test approaches primarily focus on single-component testing without emulating the final device environment.

Consequently, they do not adequately exercise interactions between on-chip and off-chip components or between hardware and software. For instance, hardware-implemented protocols like CAN and SPI bus are not fully tested by structural tests. Loopback strategies are often employed to minimize interaction with the ATE, which can lead to coverage drops in detection and correction logic [42]. Additionally, scan and LBIST may introduce untested logic when the architecture is partitioned into islands or domains. These domains may be activated separately, leaving some glue logic between them structurally untestable. SLT provides better activation stimuli than LBIST, as the device is functionally exercised [14, 11, 19, 10, 18]. However, there is growing concern regarding how SLT applications should be graded [18] due to their long simulation times and the prohibitive nature of fault simulations. New approaches are emerging, ranging from instruction-based analysis [45] to indirect measurements (e.g., power consumption) [46], focusing on extra-functional properties such as temperature, voltage, and frequency.

As Software-Based Self-Tests (SBST) have emerged as an effective technique for online testing of CPU modules [47] or manufacturing [48], the concept has also been applied to system peripherals [49–52]. SBSTs applied to system peripherals have been developed to be transparent to the application for simple protocol-related

testing. SBST and SLT reside under the umbrella of functional tests. On the one hand, SBSTs are mainly adopted as test strategies for online testing capabilities of modules required by safety standards, with the main objective of reaching very high absolute coverage figures, or as a stand-alone functional test strategy focused on a single on-chip component in the SoC, and sometimes isolated during the test between other components. On the other hand, the primary aim of SLT is the detection of faults that escape from previous manufacturing test phases. SLT is typically a holistic strategy that focuses producing heavy functional workloads for generating complex hardware-software interaction between all the system components on and off-chip, complex protocol functions, and requiring additional ATE capabilities, e.g., the capabilities of driving the communication pins from the ATE side.

## 2.2 System-Level Test State-of-the-art

System-Level Test (SLT) has emerged as a critical component in the semiconductor testing ecosystem, addressing the limitations of traditional structural and functional tests [53, 17]. As integrated circuits (ICs) become increasingly complex, SLT plays a vital role in ensuring product quality, reliability, and performance under real-world conditions [10].

Modern SLT approaches incorporate adaptive testing strategies, utilizing predictive models and machine learning techniques to identify high-risk devices for selective SLT application [6]. Integration with other test stages, such as burn-in testing and post-silicon validation, is becoming more common to optimize overall test flow [54].

Despite its advantages, SLT faces several challenges, including high test costs due to specialized hardware requirements, long test durations, and the absence of standardized fault models and coverage metrics [18]. Debugging complexity and integration challenges with other test stages also pose significant hurdles [11].

SLT has found widespread adoption in industries where reliability and performance are critical, such as the automotive sector, mobile and consumer electronics, and server and data center applications [18, 6]. Case studies from major semiconductor companies have demonstrated the effectiveness of SLT in reducing Defective Parts Per Million (DPPM) levels and improving overall test quality [19, 10].

The introduction of machine learning and AI techniques for predicting SLT outcomes, optimizing test coverage, and improving fault diagnosis is gaining traction [55].

### 2.2.1 Enabling stress for SLT phase

The increasing complexity of automotive System-on-Chip (SoC) devices and the stringent reliability requirements for safety-critical applications have necessitated the development of advanced testing methodologies. Among these, the Burn-In (BI) process has emerged as a critical step in identifying early-life latent faults in electronic devices. BI combines thermal and electrical stress to accelerate the aging process of devices, ensuring that defective units are screened before deployment. However, traditional BI processes are associated with high costs and long durations, prompting the need for optimization.

Optimized Test-During-Burn-In (TDBI) integrates functional testing into the BI process, enabling parallel execution of stress and test procedures. For instance, flash memory cycling, RAM stress, and CPU functional stress can be performed concurrently, significantly reducing BI time. Experimental results on automotive SoCs have demonstrated substantial time savings and a reduction in recycled chips, highlighting the effectiveness of TDBI in improving throughput and cost efficiency [56].

Another approach to BI optimization leverages Direct Memory Access (DMA) and cache memory to parallelize stress procedures. By utilizing DMA to perform memory read/write operations independently of the CPU and by storing functional stress programs in the cache, this methodology reduces bus contention and enhances thermal stress. Experimental results on automotive microcontrollers have shown that this approach not only reduces BI duration but also maintains controlled power consumption and improves stress coverage [57].

Low-cost tester architectures have been developed to further reduce the cost of BI. These testers utilize generic microcontroller units (MCUs) equipped with embedded processors and minimal memory resources to generate pseudo-random patterns for stress application. By eliminating the need for extensive memory and leveraging on-the-fly pattern generation, these architectures achieve high toggle coverage and stress metrics comparable to traditional ATPG-based methods. Additionally, the integration of such testers into System-Level Test (SLT) equipment enables seamless

functional testing under operational conditions, further enhancing the reliability of automotive SoCs [31].

Recent advancements have explored the combination of BI and SLT into a unified testing step. This approach leverages the functional capabilities of SLT to detect faults that may not manifest under structural testing while the BI process ensures worst-case operational conditions. By merging these steps, significant cost savings can be achieved through reduced handling and equipment usage while maintaining high defect coverage. Industrial case studies have demonstrated the feasibility of this approach, particularly when combined with high-voltage stress techniques to further reduce BI duration [54].

The evolution of BI methodologies, from traditional processes to optimized TDBI, DMA-based stress, and low-cost tester architectures, reflects the industry's commitment to improving the efficiency and reliability of automotive SoC testing. By integrating these advancements with SLT, manufacturers can achieve higher throughput, reduced costs, and enhanced defect coverage, ensuring the delivery of robust and reliable electronic components for safety-critical applications.

### 2.2.2 State-of-the-art for communication peripherals testing

Since the advent of SoCs, designers have started to pack more and more peripherals in the same die. This trend has increased the complexity of SoCs as well as their peripheral speed and capabilities. As complexity increases, so does the testing effort; thus, different methods have been developed in the literature for effectively testing peripherals. In the scope of the current work, Table 2.1 shows a comparison between different test approaches for communication peripherals.

| Test Nature | Test Approach | Pros | Cons | Target |
|---|---|---|---|---|
| Structural | ATPG | Automated<br>High coverage capabilities | No functional interactions,<br>No online testing, Low Speed Testing | All |
| | BIST [58, 59] | At-speed testing<br>High coverage capabilities<br>Limited online testing (startup) | Area overhead<br>Limited capabilities for TDF<br>No functional interactions | SPI, RS-232 |
| Functional | SBST [49] | At-speed Online testing<br>Deterministic and automated methodologies | Simple protocol operations<br>No off-chip communications | UART, HDLC,<br>ETHERNET |
| | SBST [52] | At-speed Online testing | No off-chip communications | CAN |
| | SBST [50] | At-speed Online testing<br>Off-chip communications | Simple protocols<br>No errors injector | UART, PIA |

Table 2.1: Comparison between different test approaches for communication peripherals testing.

The first commonly used approach for testing communication peripherals is always the Automatic Test Pattern Generator (ATPG), a highly automated engine with high coverage capabilities based on generating Scan-based patterns. However, ATPG-based approaches do not have any online testing capabilities or at-speed testing. Most importantly, Scan-based patterns test the modules without functionally using the communication peripherals, as already underlined in the previous section.

Another partially automated approach is the introduction of BIST in the SoC, paying the overhead of additional area [58, 59]. Although BISTs provide at-speed high coverage capabilities for stuck-at and partially online testing only at the SoC startup, they are area-hungry, and they need to be designed ad-hoc for every communication peripheral present in the SoC, leading to an abnormous area overhead.

### 2.2.3 State-of-the-art for uncore logic/crossbars testing

Crossbars are integral to the communication pathways in VLSI systems. Testing crossbars can ensure that the data paths between uncore components are functioning correctly. This is crucial because uncore components rely heavily on these pathways for data exchange [60].

The testing of crossbars is a very well-known problem [61], and since the advent of SoCs, the problem has shifted from the board level to the SoC level. In the literature, approaches exist for testing crossbars on SoC based on different natures, ranging from pure ATPG-based and BISTs. However, in the state-of-the-art literature, there are no functional approaches for testing a crossbar into a SoC.

| Test Nature | Test approach | Pros | Cons | Metrics |
|---|---|---|---|---|
| Structural | ATPG | Automated<br>High coverage capabilities | Low Speed Testing<br>No online testing | Fault Coverage |
| | BIST | At-speed<br>High coverage capabilities<br>Limited online testing (startup) | Area overhead<br>Limited capabilities for transient<br>partially automated | Fault Coverage |
| | BIST [62][63] | Automated (high-level fault model)<br>At-speed<br>High coverage capabilities<br>Limited online testing (startup) | Sequential and combinatorial elements tested separately<br>Limited to stuck-at fault model<br>Components isolations to/from bus | Fault Coverage<br>High-Level state machine coverage |
| | BIST [64] | At-speed<br>High coverage capabilities<br>Minimal runtime impact | Limited to stuck-at fault model<br>Components isolations to/from bus<br>Area overhead | Fault Coverage |

Table 2.2: Comparison between different test approaches for crossbar testing.

Table 2.2 presents the pros and cons of such test approaches. For example, an ATPG-based flow is highly automated and provides high coverage capabilities but is executed at low speed. It has no online testing capabilities, as they do not exercise functional communication paths between components.

On the other hand, BISTs provide high coverage capabilities and at-speed testing by isolating the crossbar components, which can be executed at the device's startup. On the other hand, it introduces area overhead, limited capabilities for transient faults, and a partially automated design flow. An enhancement of BIST could be seen in [63, 62]. In the latter, an automated generation flow for a high-level fault model based on state machine transition is used to test the sequential and combinatorial parts separately. The combinatorial part, i.e., wires and multiplexers, are tested by test vectors generated from a graph-based system representation similar to the one proposed hereby. On the other hand, component interactions are not tested due to the isolation of the crossbar components.

In [64], authors present an efficient online BIST technique for uncore components in SoCs, which can account for a significant portion of the overall logic area of a multi-core SoC. The technique achieves high test coverage by storing high-quality test patterns in off-chip, non-volatile storage. However, a simple technique that stalls the uncore-component-under-test can result in significant system performance degradation or even visible system unresponsiveness.

To overcome these challenges, the authors propose three special hardware features: resource reallocation and sharing (RRS), no-performance-impact testing, and smart backups that reduce the area overhead and impact on system performance.

## 2.3 State-of-the-art for System-Level Test workload generation

Exploration of fuzzy-based approaches and large language model-based techniques for optimizing SLT program generation is an emerging area of research.

In [65], researchers present an automatic grey box System-Level Test (SLT) program generation method to find code snippets that control the Device Under Test (DUT)'s extra-functional properties. This method aims to achieve better char-

acterization or improve the coverage of emerging defect types. The method does not require structural information and relies solely on simulation results or hardware measurements to guide the generation. The researchers demonstrate that their method outperforms handcrafted snippets on a RISC-V super-scalar processor. The paper also discusses the use of mutation-based grey box fuzzing as the underlying generation method. The researchers conducted an experiment where they aimed to maximize the power consumed during the test using fuzzing as the generation method. The results show that the generated snippets consistently outperform the handcrafted ones. The researchers also found that the length of the snippet only plays a limited role in their performance, whereas the amount of data hazards appears to contribute more to their power consumption.

The optimization of System-Level Test (SLT) program generation using Large Language Models (LLMs) has been proposed [66]. The papers propose the use of LLMs enhanced by Structural Chain of Thought (SCoT) prompting, and a pool of previously generated snippets to generate high-quality code snippets for SLT. The research shows that this approach can automatically generate snippets for SLT that target specific non-functional properties such as Instruction Per Clock (IPC), reducing time and effort. The findings show that this approach improves the quality of the generated snippets compared to unstructured prompts containing only a task description. The paper also discusses the use of LLMs as an iterative optimizer by combining state-of-the-art prompt engineering and optimization techniques. The experiments conducted show that the SCoT prompts outperform the basic prompts in terms of the best score and pass@k metrics. The optimization framework, including the temperature schedule and pool updates, improves the best achievable score of pre-trained LLMs by 12%.

## 2.4 State-of-the-art for System-Level Test grading

Traditionally, structural test methods have dominated manufacturing testing due to their automation capabilities and high fault coverage. However, as modern devices grow increasingly complex and the limitations of test equipment for storing test patterns become apparent, alternative techniques such as functional test programs are gaining traction among companies; for example, Software-Based Self Tests (SBSTs), these tests target specific modules during the manufacturing phase [48] or online

testing [51, 67–69], Test During Burn-In, this approach combines the burn-in phase designed to eliminate infant mortality in devices [70] with functional test and stress programs [71], and, SLT involves executing a functional application that mimics the device's mission mode, such as booting and running a Real-time Operating System (RTOS) [18, 11, 19, 39].

The primary limitation of SLT lies in the difficulty of assessing its fault coverage [18]. The fault simulation process required to evaluate the effectiveness of SLT workloads, such as an operating system boot, can demand substantial CPU resources and extensive time, especially during the early development stages.

Recent advancements in SLT methodologies have explored the use of power consumption as a proxy for detecting defects and optimizing test programs [65]. That higher power consumption correlates with increased stress on the Device Under Test (DUT), enabling the detection of delay faults and other marginal defects. The approaches highlight the importance of non-functional properties, such as power and temperature, in enhancing SLT effectiveness. By focusing on power consumption, test engineers can develop more robust SLT programs that stress the DUT under realistic operating conditions, improving defect detection and possibly the overall test coverage [65].

# Chapter 3

# Main Contributions

*Cogito, ergo sum.*

René Descartes

The System-Level Test (SLT) phase is an additional, expensive, and functionally-oriented test phase in the semiconductor industry.

As Software-Based Self-Tests (SBST) have emerged as an effective technique for online testing of CPU modules [47] or manufacturing [48], the concept has also been applied to system peripherals [49–52]. SBSTs applied to system peripherals have been developed to be transparent to the application for simple protocol-related testing. SBST and SLT reside under the umbrella of functional tests. On the one hand, SBSTs are mainly adopted as test strategies for online testing capabilities of modules required by safety standards, with the main objective of reaching very high absolute coverage figures, or as a stand-alone functional test strategy focused on a single on-chip component in the SoC, and sometimes isolated during the test between other components. On the other hand, the primary aim of SLT is the detection of faults that escape from previous manufacturing test phases. Although SLT is being increasingly adopted without further investigation into how it reduces the defect parts per million (DPPM), it is typically a holistic strategy that focuses producing heavy functional workloads for generating complex hardware-software interaction between all the system components on and off-chip, complex protocol functions, and requiring additional ATE capabilities, e.g., the capabilities of driving the communication pins from the ATE side.

The focus of this study is on traditional fault models used in the industrial manufacturing test flow, as demonstrated by a case study involving an automotive device manufactured by STMicroelectronics. Specifically, the Stuck-at Fault Model and the Transition Fault Model are highlighted as foundational fault models for evaluating the effectiveness of the SLT phase in identifying coverage gaps left by ATPG. These fault models serve as a baseline for understanding how structural tests can leave undetected faults, which may later manifest as test escapes.

It is important to emphasize that this PhD dissertation does not claim to provide a definitive solution to the problem of test escapes in structural testing. As technology evolves, new fault models and faulty behaviors are likely to emerge, which may be better addressed by advanced techniques such as cell-aware testing or small delay fault testing. These methods are designed to capture more intricate defects at the transistor or timing level, but they come with significant challenges. For instance, the complexity of generating test patterns for these advanced fault models using ATPG tools is non-negligible and often results in residual coverage holes. The dissertation acknowledges that while traditional fault models like Stuck-at and Transition Faults remain critical for industrial testing, they are not exhaustive. The continuous advancement of semiconductor technologies will necessitate the development of new fault models and testing methodologies to address increasingly complex defect mechanisms. However, even with these advancements, the inherent limitations of ATPG tools and the complexity of modern devices mean that achieving 100% fault coverage remains an elusive goal, and it could be partially solved by SLT.

The hereby PhD thesis aims to improve the state-of-the-art for the System-Level Test phase in the following main directions:

- Covering structural weaknesses, identifying and addressing test escapes due to DfT logic.
- Automatic generation, by developing techniques for automatically creating effective SLT workloads.
- Assessment, by evaluating the effectiveness of SLT in improving product quality at different abstraction layers.

In the subsequent chapters, each technique for the aforementioned objectives will be described in detail. This will include:

- Novel methods for structural weaknesses in complex systems, such as:
  - The proposed functional test methodology optimizes stress before the SLT phase and offers significant advantages over structural tests, particularly in stressing the interconnection logic to and from embedded memories. It ensures uniform stress application, achieving comprehensive coverage, and matches structural tests in toggle coverage for interconnection logic. Additionally, the methodology operates at speed, making it more effective at detecting timing-related issues that slower structural tests might miss. This is especially critical for modern safety-critical devices. From an absolute perspective, results show that the proposed functional procedures can more uniformly distribute the stress over the memory signals in less execution time than Structural tests (including Memory BIST) by about 118 times higher for Data bus and 35 for Address Bus. On the other hand, in MBISTs, address generation can be achieved by different methods, increasing the toggle distribution. However, this solution has the disadvantage of increasing the silicon area of BISTs and their complexity and reduced flexibility compared to functional methods [16].
  - The proposed SLT methodology for communication peripherals offers key advantages, such as at-speed testing, support for off-chip communications, and the ability to handle complex hardware/software interactions, making it a versatile approach for system-level testing. However, it has limitations, including significant manual effort, and the need for additional tester capabilities, which can increase costs and complexity. It advances the state-of-the-art by moving beyond simplistic assumptions (e.g., relying solely on system boot as a test strategy) and is the first to analyze structural test weaknesses. This analysis informs the development of an SLT suite specifically for communication peripherals, addressing test escapes in areas overlooked by structural tests. The communication peripheral under test is a CAN controller counting about 436K for Stuck-At Fault (SAF) and Transition Delay Fault (TDF). The original coverage guaranteed by the manufacturing test suite of structural tests, including Scan, LBIST, and MBIST tests, is 97.89 % for SAF and 89.38 % for TDF. The synthesized SLT functional test procedure provides increments of

1.12 % and 1.51 %, respectively, reaching 99.01% for SAF and 90.89% for TDF [1].

– The proposed methodology for uncore logic testing addresses the challenges of functional test program development, which often requires skilled engineers and manual effort. While human intervention is needed only to extract the memory map from the SoC user manual, the SLT workload is generated automatically and executed at speed. This workload enforces component interactions by exercising functional bus paths between masters and slaves, either simultaneously or independently, without requiring netlist knowledge. The methodology leverages graph-based models to abstract the system, enabling efficient graph-covering techniques for workload generation by focusing on the crossbar modules, and as ripple effect on the un-core logic. The number of faults for the crossbars are ca. 155k and 111k, respectively. Structural tests include about 150k scan-based test patterns that reach a Stuck-At Fault (SAF) coverages of 95.18 % and 94.28%, meanwhile, the Transition Delay Faults (TDF) coverages are 88.71% and 87.49%, respectively for the crossbars.

  The SAF coverage is increased by 1.9% and 1.67%, bringing the crossbars coverages up to 97.08% and 95.95% respectively. Finally, TDF coverage is raised of 2.4% and 2.27% respectively, and compared to structural tests, the coverages goes up to 91.11% and 89.76%.

  Moreover, to further prove the effectiveness of the generated SLT suite, it has been assessed on different SoC modules such as CPU (1.5M SAF/TDF), peripheral bridge (76k SAF/TDF), Controller Area Network (CAN) peripheral (435k SAF/TDF), and the overall memory controllers and collars(1.4M SAF/TDF). Starting from SAF coverage of 98.03%, 90.88%, 95.73% and 94.51%, respectively for CPU, peripheral bridge, and CAN peripheral, and memory controllers which are already very high fault coverages. The introduction of the SLT suite has been assessed only on the non-detected (residual) faults from structural tests; it covers the 13.96%, 11.60%, 3.98%, 16.69% for CPU, Peripheral Bridge, CAN peripheral interfaces, and memory controllers for the non-detected (residual) faults from structural tests. Meanwhile, the TDF coverages are 95.13%, 68.67%, 87.18%, 84.23%, respectively for CPU, peripheral

bridge, CAN peripheral, and memory controllers; the introduction of the SLT suite covers 1.96%, 0.62%, 0.20% and 1.50% of non-detected faults from structural tests, respectively [2].

- Algorithms and tools for automatically generating SLT workloads based on high-level SoC modeling [2] or genetic frameworks, addressing limitations in current commercial EDA tools. The key advancement lies in an automated flow that significantly improves efficiency by leveraging parallel computing, resulting in faster execution times and substantial human time savings. This approach represents a major step forward in the state of the art, streamlining workload generation while reducing the reliance on manual intervention. The automatic generation of a test program using genetic optimizer for maximizing the power consumption could be done directly using hardware measurements only. However, it is a slow and iterative process taking up to 18 days. Therefore, the generation can be divided into two different steps based on architectural simulation (maximizing the IPC) and register data tuning (maximizing the power consumption). It can provide a significant speed-up, up to 1.81x faster, and more power hungry SLT programs [72].

- Metrics and frameworks for quantitatively assessing the impact of SLT on DPPM and other quality indicators at different levels:

  - As noted by Polian et al. in [18], grading SLT workloads through fault simulations is becoming increasingly impractical due to the growing complexity of SoCs. Toggle activity oriented toolchains offers a practical trade-off between accuracy and computation time for evaluating SLT workloads [26]. Additionally, leveraging high-level SoC models, standardized software architectures (which reduce porting time), and workload regularity allows fault simulations to be partitioned and executed in parallel, significantly reducing computational intensity and improving efficiency. The division into smaller workloads allows the evaluation of the overall generated SLT workloads in less than 15 days, on average, as compared to the estimated month [2].
  - The proposed methodology analyzes data propagation among instructions across cores to observable points, identifying blocking instructions that may override computed results and impact fault coverage. This approach can guide test engineers during the early stages of functional test

program development, reducing repetitive fault simulation campaigns and providing detailed feedback on test programs. It relies on analyzing instruction traces from the actual device, an ISA simulator, or logic simulations. This makes it a cost-effective and versatile solution applicable to various testing scenarios in its early development stages,i.e., seconds of trace analysis vs. hours of fault simulations. By using as fitness proposed metrics to a genetic optimizer, it significantly accelerates the development of functional test programs compared to a genetic optimizer using as fitness the fault coverage from fault simulations. For a multiplier unit, it achieved the targeted fault coverage of 90% in 153 hours (6.4 days) by evaluating 7,868 programs, with a speedup of over 23x compared to fault simulations based approach. Similarly, for the integer divider unit, it reached 75% fault coverage in 71 hours (3 days) after evaluating 3,654 programs, achieving the same result 8.55x faster than the fault simulation-based approach [45, 73].

By systematically investigating various aspects of SLT, this thesis aims to demonstrate its effectiveness in the semiconductor industry. The introduction of SLT enhances product quality and reliability by addressing limitations of traditional testing methods and ensuring comprehensive fault coverage. This approach not only improves the performance and robustness of semiconductor devices but also aligns with the industry's growing demand for higher reliability in mission-critical applications.

## 3.1 Author's Publication List

[16], F. Angione, P. Bernardi, G. Filipponi, M. Sonza Reorda, D. Appello, V. Tancorre, R. Ugioli, ***An Optimized Burn-In Stress Flow targeting Interconnections logic to Embedded Memories in Automotive Systems-on-Chip***, 2022 IEEE ETS, Barcelona, Spain, 2022, pp. 1-6, doi: 10.1109/ETS54262.2022.9810396.
[1], F. Angione, P. Bernardi, N. di Gruttola Giardino, G. Filipponi, C. Bertani and V. Tancorre, ***A System-Level Test Methodology for Communication Peripherals in System-on-Chips***, in IEEE Transactions on Computers, early access, 2023, doi: 10.1109/TC.2024.3500375.
[2], F. Angione, P. Bernardi, G. Iaria, C. Bertani, V. Tancorre, ***Automatic Generation***

***of System-Level Test for un-core logic of large Automotive SoC***, submitted in IEEE Transactions on Computers, 2024.

[72], D. Schwachhofer, F. Angione, S. Becker, S. Wagner, Stefan, M. Sauer, P. Bernardi, I. Polian, ***Optimizing System-Level Test Program Generation via Genetic Programming****, IEEE European Test Symposium (ETS), The Hague, Netherlands, 2024, pp. 1-4, doi: 10.1109/ETS61313.2024.10567817.*

[26], F. Angione, D. Appello, P. Bernardi, A. Calabrese, S. Quer, M. Sonza Reorda, V. Tancorre, R. Ugioli, ***A Toolchain to Quantify Burn-In Stress Effectiveness on Large Automotive System-on-Chips***, in IEEE Access, vol. 11, pp. 105655-105676, 2023, doi: 10.1109/ACCESS.2023.3316511.

[45], F. Angione, P. Bernardi, A. Calabrese, L. Cardone, A. Niccoletti, D. Piumatti, S. Quer, D. Appello, V. Tancorre, R. Ugioli, ***An innovative Strategy to Quickly Grade Functional Test Programs***, 2022 IEEE International Test Conference (ITC), Anaheim, CA, USA, 2022, pp. 355-364, doi: 10.1109/ITC50671.2022.00044.

[73], F. Angione, P. Bernardi, A. Calabrese, L. Cardone, S. Quer, C. Bertani, V. Tancorre, ***A Novel Indirect Methodology based on Execution Traces for Grading Functional Test Programs***, submitted in IEEE Transactions on Computers, 2024.

# Chapter 4

# System-Level Test Techniques

*Somewhere, something incredible is waiting to be known.*

Carl Sagan

## 4.1 Overview

The hereby chapter is a comprehensive exploration of System-level test (SLT) techniques and tools designed to enhance the testing and validation processes of System-on-Chips (SoCs) due to the growing complexity and the limitations of manual development processes. It delves into various aspects of SLT, including structural weaknesses coverage, automatic generation of SLT workloads, and assessment methods for SLT workloads.

## 4.2 Covering structural weaknesses

### 4.2.1 Enabling stress for SLT phase

The proposed approach aims to integrate the internal stress induced during BI with structural methods (like Logic and Memory BIST) and the adoption of functional

routines. Such routines are automatically generated in a high-level language, complementing the structurally applied stress.

The generated functional programs can address parts of the system that the structural approaches could only partially exercise and are designed to provide more uniform and stronger stress than pure structural approaches. For example, structural stress coverage can be complemented where DfT component interactions may leave some gates not toggled. Moreover, structural methods can provide quite unbalanced stress, making some gates or lines more exercised than others regarding the number of toggles.

The proposed approach examines these deficiencies and proposes functional methods to increase the average toggle activity per gate. In particular, the focus is on the interconnections between CPUs and memories, which are often excised in a poor and non-uniform manner by structural methods.

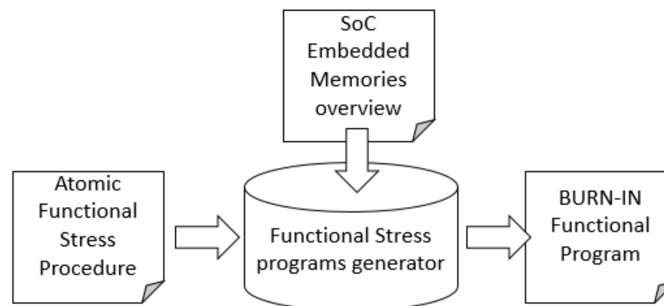



Fig. 4.1: High-level view of the proposed generation approach.

Figure 4.1 illustrates the approach, whose purpose is the creation of bare-metal applications to activate the portion of the circuit immediately surrounding embedded memories. The generated stress-effective application adapts an atomic functional procedure to system components. Portability and flexibility across devices are achieved using an independent library written in a high-level programming language.

In the details, we aim to improve the stress of the logic surrounding embedded memories. Such parts of the overall circuit often include the DfT logic, which is the interface to components implemented between different technologies. Based on our practical experience, this circuit part may be critical for manufacturing tests. Hence, an optimized BI phase that better exacerbates potential latent faults in these SoC

parts is crucial to enable the following test steps, including SLT, to achieve minimal defect level, reduce overkill, and mimic mission-mode applications.

#### 4.2.1.1 Basic Working Principles

This work aims to define effective and systematic solutions to generate functional stimuli that better activate the interconnections between the Sea of Gate, specifically the CPUs and Peripherals SoC cores, and the embedded memories, focusing on RAMs.

As the first step for the approach, a description of memories is manually created and called *memory overview*. The memory overview document contains information on the size, data width, and address range of memory banks. The user manual provides most of the information about the SoC's memories. As seen from the workflow in Figure 4.2, the high-level memory map, from the perspective of the embedded programmer, is enriched by information obtained from the gate netlist of the circuit and its physical layout. During these phases, designers can help produce documentation to speed up the development of the memory overview document, automatizing and, eventually, document generation. As a next step, an offline ad-hoc tool takes the produced memory overview as input and generates a list of targeted memory addresses. This list of addresses is then encapsulated within a bare-metal application that executes access operations to memories.

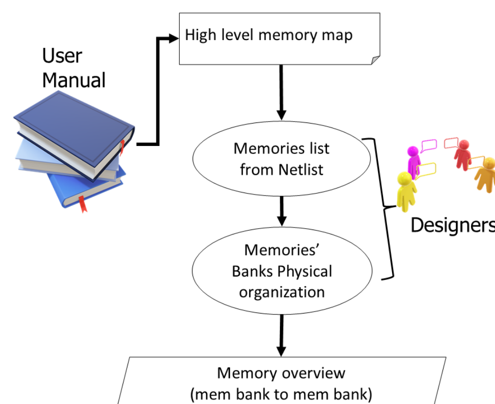



Fig. 4.2: Memory overview.

Memory access operations are read and write operations for different addresses (memory banks) and for different data types. In order to maximize the activity

on the address bus with a minimum number of operations, an already proposed approach [74] is adopted. The approach in [74] exploits the binary arithmetic to fully activate the address calculation unit with few load and store operations. In the stress context, accessing the correct memory addresses permits a very effective activation. Figure 4.3 shows that at half of the address range, it is necessary to add one to the address to switch simultaneously all the bits in the lower address range. We proposed targeting these two complementary address values to reach better stress than scanning the whole memory array like MBISTs do. The stress of the address bits is maximized by repetitively accessing these two memory locations to write and read their contents. Moreover, this approach allows the generation of uniformly distributed stress among address and data bit lines.

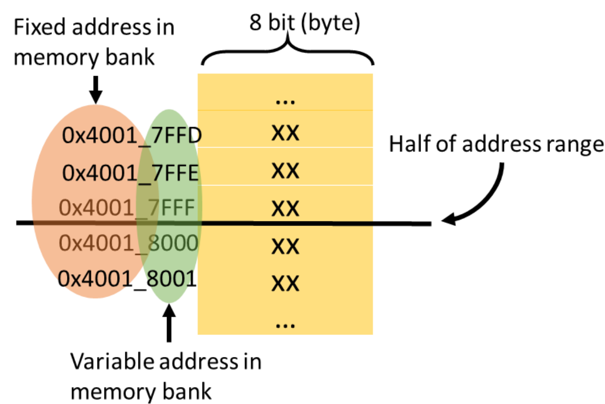



Fig. 4.3: Address toggle example.

In order to implement a flexible access method to the memory, a high-level code header is proposed, which includes the most relevant information about the memory access. Figure 4.4 reports the C language code needed to define a *mem_operation* data structure to support the functional stress code. This declaration contains all the stress-relevant memory addresses of the SoC, e.g., the addresses at half the memory range for the several memory banks that are included in the SoC.

```
1 #define SIZE 2
2 UNION_UINT32_16_8_T *  mem_operations[N][SIZE]= {  //word aligned
3 0x4007fffc, 0x5280fffc, ... };
```

Fig. 4.4: Example of generated code

The presented data structure is based on a user-defined data structure type called *UNION_UINT32_16_8_T*. As Figure 4.5 shows, this type is based on a union artifact,

which allows accessing memories using different access sizes. This data structure is based on the hierarchical decomposition of data types representing a word, two half words, and four bytes simultaneously.

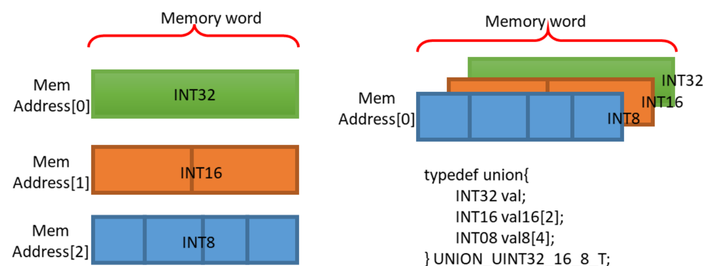



Fig. 4.5: Normal Data Structure vs optimized data structure used for memory operations.

A high-level programming language allows the selection of which data access is done to/from the memories. The optimized data structure is part of the bare-metal application and is used by the functional routines. It uses pointers to the optimized data structure presented in Figure 4.5, where *N* is the number of memory operations to complete and *SIZE* is the size of the vector placed at the given address. This image illustrates two approaches to memory organization and data alignment: separate memory allocation and union-based memory sharing. On the left side, each data type is stored in a separate memory address. Mem Address[0] contains an INT32 (32-bit integer), Mem Address[1] contains an INT16 (16-bit integer), and Mem Address[2] contains an INT8 (8-bit integer). Each data type occupies its own memory word, resulting in separate memory allocations for each variable. On the right side, a union is used to overlay multiple data types within the same memory word. The union allows an INT32, an array of two INT16 values, and an array of four INT8 values to share the same memory space at Mem Address[0]. This approach enables efficient memory usage by allowing different data types to coexist in the same memory location, depending on the context. It highlights the trade-off between simplicity in separate memory allocation and the efficiency of union-based memory sharing.

In such a way, a functional routine parsing the vector can access different data using the same address. This approach drastically reduces the size of the constant section in the code, i.e., it stores one pointer for accessing seven different data instead of having seven different pointers for each value in the non-optimized data structure.

#### 4.2.1.2 Architecture-independent methodology

The main goal is to stimulate gates within the memory collar bypassed by the memory BIST or scan-based approach. Therefore, looking at Figure 4.6 that represents a general architecture of a Memory BIST collar, the approach exacerbates potential latent faults in the red path, stimulated only by functional procedures. On the other hand, light blue, blue, and orange paths are stimulated, respectively, by Memory BIST and scan. Instead, the grey path is stimulated by both functional and memory BIST.

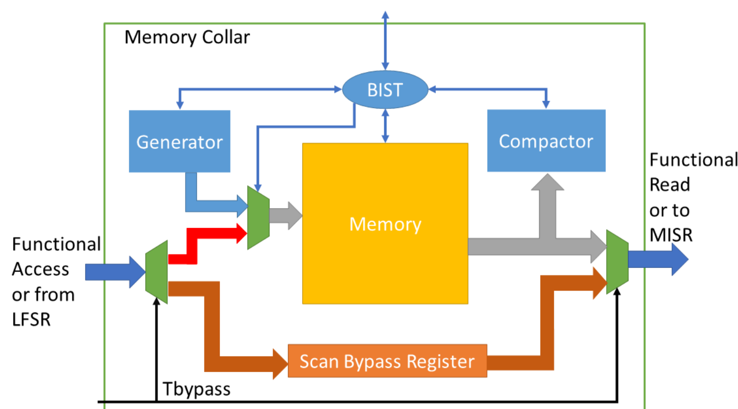



Fig. 4.6: Bypassed register-based DfT for memory collar.

Although different approaches in the architecture presented in Figure 4.6 cover different complementary paths, the architecture can be further optimized to save logic and eliminate any coverage loss. An example can be seen in Figure 4.7, where a multiplexer has been added as a selector between signals tied to zeros or the memory output. Figure 4.8 shows another DfT configuration for a memory collar, where the input multiplexer has been added as a selector of Chip Select of the memory entity between the control signal or a fixed test value. Moreover, the Chip Select freezes the output to a defined value, i.e., during LBIST execution the memory outputs are tied to zero.
Despite these different architectures, the proposed approach is still valid for stimulating paths to and from memories. In particular, runtime functionalities are activated. Therefore, it more effectively stresses the device because it works the same way as during the normal in-field behavior. Moreover, it is independent of the Memory collar architecture and, consequently, is portable across different devices.

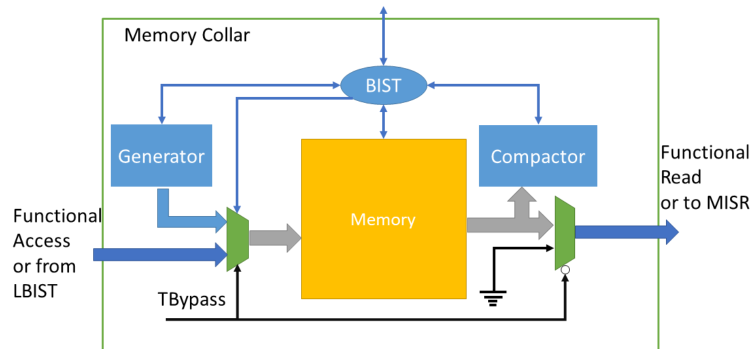



Fig. 4.7: Multiplexer isolation-based DfT for memory collar.

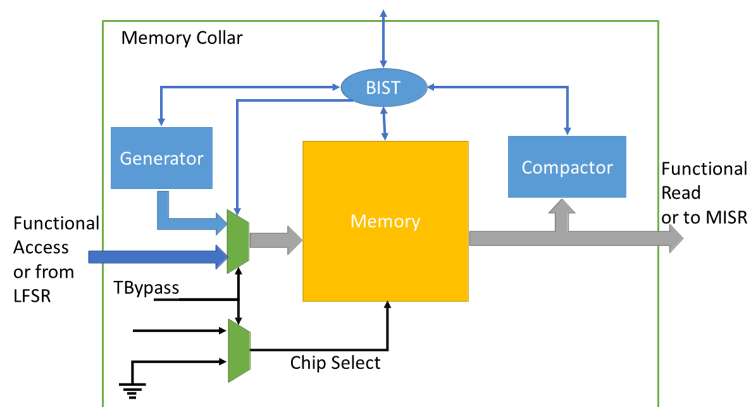



Fig. 4.8: Chip select isolation-based DfT for the memory collar.

Memory BISTs are hardware structures that execute a specific algorithm, hard-coded within the BIST's Control Unit, and are not flexible as a functional approach. The functional procedures depend on the type of memory access that the architecture allows. It depends on the possibility of executing misaligned memory access to the memory. Figure 4.9 shows a routine for architecture where only word access to the memory is allowed. The data structure presented in Figure 4.5 is useless. The provided routine is a low-level memory operation function that interacts with a

```
routine(address){
CPU_INT32U_T * p0,p1;
p0=address;  //half of address range, word aligned
p1=address +1 ; //next word
for i=0 to N/4{
        p0->val32=0x55555555;  p1->val32=0xAAAAAAAA;
        read p0 (word);        read p1 (word);
    }
}
```

Fig. 4.9: Pseudo code of functional routine for architecture where misaligned memory access is illegal.

specific address range in memory. It uses two pointers, p0 and p1, which are derived from the input address. The pointer p0 is set to the base address, while p1 points to the next word-aligned memory location (i.e., address + 1). Within a loop that iterates N/4 times, the routine performs the following operations for each iteration: it writes the 32-bit value 0x55555555 to the memory location pointed to by p0 and the value 0xAAAAAAAA to the location pointed to by p1. After writing these values, it reads back the data from both p0 and p1 as 32-bit words. This process is repeated for the entire loop. The purpose of this routine could be related to testing or initializing memory. Writing alternating bit patterns like 0x55555555 and 0xAAAAAAAA is a common technique used in memory testing to verify that memory cells can store and retrieve data correctly without interference or corruption. The subsequent read operations ensure that the written values are correctly stored and retrieved from memory.

On the other hand, in the literature, architecture also exists that allows misaligned memory access [75]. Therefore, Figure 4.10 presents a routine intended for architecture where misaligned access is allowed. It also partially exploits the data structure presented in Figure 4.5. Since the routine executes read and write operations for a given number of iterations, in both Figure 4.10 and Figure 4.9 the desired toggle

```
routine(address){
CPU_INT32U_T * p0,p1;
p0=address;  //half of address range
p1=address +1 ; //next word
for i=0 to N/4{
        p0->val8[2]=0xff;  p1->val32=0xffffff00;
        read p0 (byte);    read p1 (word);
    }
}
```

Fig. 4.10: Pseudo code of functional routine for misaligned memory access

coverage is divided by four since, within the body of the loop, both the data bus and address bus are uniformly stimulated. For example, referring to Figure 4.10, the same applies to Figure 4.9, the sequence of read and write operations can be ordered for maximizing either the data bus or the address bus within a single execution of the compound statement in Figure 4.10.

| Access Order max ABus | Access order max DBus | Address Bus Lower Part | Data Bus D/Q |
|---|---|---|---|
| 1 | 1 | 0xFFFC | 0x000000FF / 0x000000000 |
| 2 | 3 | 0x0000 | 0xFFFFFF00 / 0x000000000 |
| 3 | 2 | 0xFFFC | 0x00000000 / 0x0000000FF |
| 4 | 4 | 0x0000 | 0x00000000 / 0xFFFFFFF00 |

Table 4.1: Transition on busses for single body loop execution.

The routine for misaligned memory access, described in Figure 4.10, can be further optimized. It is presented in Figure 4.11. It consists of a loop unrolling the previous routines and the use of different data types. It uses two pointers, p0 and p1, which are derived from the input address. The pointer p0 is initialized to the base address, while p1 is set to point to the next memory location (p0 + 1). These pointers are used to access and manipulate memory in a structured way. The loop iterates from i = 0 to i < N / (11 * 2), where N / (11 * 2) determines the total number of iterations. Within the loop body, a sequence of memory operations is performed. These operations include writing specific values to memory locations pointed to by p0 and p1, as well as reading data back from these locations. The writes and reads are performed using different data sizes, such as 8-bit (byte), 16-bit (halfword), and 32-bit (word), to test or manipulate memory at various granularities. The loop body begins with a series of write operations. For example, the value 0xfffff00 is written to p1->val, and 0xff is written to the second byte of p0 (p0->val8[2]). This pattern continues with alternating writes to p0 and p1, using different data sizes and offsets. These writes are likely designed to test memory behavior or initialize it with specific patterns. For instance,

```
1  routine(address){
2  CPU_INT32U_T * p0,p1;
3  p0=address;  // half of address range
4  p1=p+1;
5  for i=0 to N/(11*2){
6      p1->val=0xfffff00;       p0->val8[2]=0xff   /*start body loop*/
7      p0->val=0xfffff00        p1->val8[1]=0xff;
8      p1->val16[1]=0xffff;     p0->val8[2]=0x00;
9      p1->val=0xFFFF00FF;      p0->val16[1]=0xff00;
10     p1->val8[1]=0xff;        p0->val[2]=0x00;
11     p1->val=0xffffffff;
12     read p1 (word);          read p0 (byte);
13     read p1 (word);          read p0 (word);
14     read p1 (byte);          read p1 (halfword);
15     read p0 (byte);          read p1 (word);
16     read p0 (halfword);      read p1 (byte);
17     read p0 (byte);                             /*end body loop*/
18 }
19 }
```

Fig. 4.11: Pseudo code of optimized functional routine for misaligned memory access

alternating patterns like 0xfffff00, 0xFFFF00FF, and 0xffffffff are commonly used in memory testing to detect issues such as bit flips or interference between adjacent memory cells. After the write operations, the routine performs a series of read operations. These reads are also performed using different data sizes and offsets, such as reading a word from p1, a byte from p0, a halfword from p1, and so on. The purpose of these reads is likely to verify that the previously written values were correctly stored in memory and can be retrieved without errors. This combination of writes and reads ensures that the memory is functioning as expected. The loop repeats this sequence of writes and reads for N / (11 * 2) iterations, systematically testing or initializing the memory range accessed by p0 and p1. This routine exploits the full capabilities of the data structure described in Figure 4.5. The increased number of read and write operations allows for reaching a satisfactory toggle activity in fewer iterations. This result follows the same approach in Table 4.1.

#### 4.2.1.3 Analytical Formulas

Regardless of the considered architectures, the number of signal transitions on the Address and Data busses (ABus and Dbus, respectively) can be defined as follows for the Memory BIST, based on the number of Memory Accesses (MA) and March

Elements (ME):

$$TotTrans_{ABus} = MemSize * \sum_{i=1}^{\#ME} MA_i \tag{4.1}$$

$$TotTrans_{DBus} = MemSize * \sum_{i=1}^{\#ME} Rise_i + Fall_i \tag{4.2}$$

$$ExecTime = MemSize * \sum_{i=1}^{\#ME} MA_i \tag{4.3}$$

All the formulas count the number of transitions within every memory access for each March Element by the memory size. On the other side, for functional approaches, the number of transitions can be computed using as parameter $N$ the $Max\{T\{A[i]\}_{MBIST}\}$, where $T$ is defined as the toggle activity of a signal. Therefore, N becomes equal to $\#ME * 2^{ABus-1}$:

$$TotTran_{ABus} = MA_{loopBody} * \frac{N}{MA_{loopBody}} \tag{4.4}$$

$$TotTran_{DBus} = MA_{loopBody} * \sum_{i=1}^{\#N} Rise_i + Fall_i \tag{4.5}$$

$$ExecTime = MA_{loopBody} * \frac{N}{MA_{loopBody}} \tag{4.6}$$

From a functional perspective, the approaches have been developed in such a way as to maintain uniform stress over the busses and are directly proportional to the maximum toggle activity developed on the least significant bit line of the address bus during the memory BIST execution.

### 4.2.2 Communication peripherals

The hereinafter proposed methodology focuses on providing guidelines for developing an SLT suite of functional test programs oriented to complement the testing abilities of structural tests for communication peripherals in SoCs.

The methodology is based on several steps, as Figure 4.12 depicts. Firstly, the potential weaknesses that may arise from structural tests must be highlighted. The previous step polarized the successive System-Level functional tests to test the mentioned structural test shadows. As a solution, a sum of composite software

programs capable of tackling all the aforementioned considerations are generated systematically.

Following the SLT philosophy, off-chip interactions with the external world are crucial for effectively leading to a meaningful functional SLT workload for the SoC, for peer-to-peer communication, or broadcast communication on the communication bus. Therefore, the proposed methodology encompasses the design of a flexible companion module capable of communicating from the ATE side to/from the SoC under test, as Figure 4.13 shows.

The methodology flow is represented in Figure 4.12.

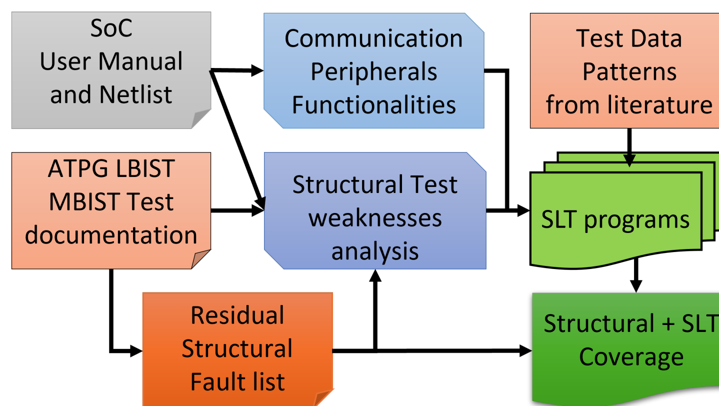



Fig. 4.12: Flow diagram of the proposed methodology.

Firstly, the potential weaknesses that may arise from structural test flow must be highlighted by qualitatively analyzing the fault list after evaluating ATPG, MBIST, and LBIST approaches. This preliminary analysis step enables the polarization of successive efforts in generating SLT functional programs targeting meaningful circuit regions and faulty conditions.

The functional program generation should cover all the possible working modes and parameters that an in-field application could use, as holistically demanded by SLT. Moreover, the proposed method draws more attention to those functionalities that structural weaknesses could threaten.

Stimulating off-chip interactions with the external world through communication channels exposed by the chip is a crucial SLT workload for the SoC. Structural

tests usually force the communication channels to loop-back configurations, as a consequence, the proposed methodology encompasses the design of a flexible companion module capable of communicating from the ATE side to/from the SoC under SLT, as Figure 4.13 shows. The companion module becomes indispensable when detection/correction features are available in the peripheral module under test, as they are not going to be tested since loop-back always transports uncorrupted bitstreams.

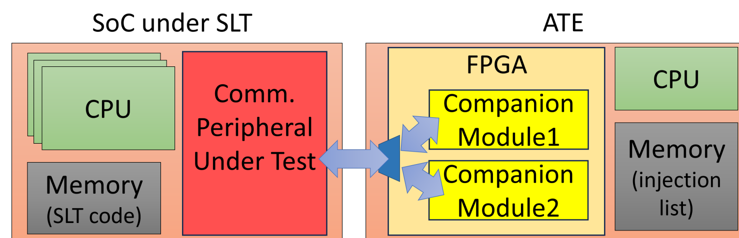



Fig. 4.13: Companion module view with a companion memory storing error injection information.

#### 4.2.2.1 Structural Test Weaknesses Analysis

The structural test weaknesses are analyzed manually, with the help of the user manual, the netlist, and the residual structural fault list. The communication module under investigation needs to be stretched as it is generically done in Figure 4.14. Arrows and labels are added to sub-module boxes that can be extracted easily from the netlist.

In the resulting visualization, the colors of the arrows indicate the criticality level of the specific interactions among modules (e.g., dark color is for the most critical and white for low risk), and the alphabetic labels group sub-modules according to their functionality (e.g., under label "A" are classified all functionalities of the internal RAM) represent the following considerations.

What emerges from crossing functionality and structural test limitations is quite intuitive to understand from Figure 4.14. Therefore, when developing SLT functional programs, meaningful areas to target are the following:

A) **Embedded memory access ports**: they may not be completely covered along structural tests due to collars and memory DfT circuits like MBIST [16].

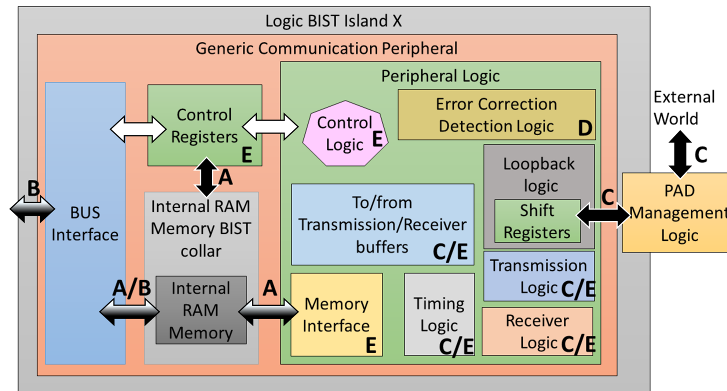



Fig. 4.14: Detail of a Generic communication peripheral.

B) **Interfaces to other on-chip components**: they may be included in different LBIST, or Scan Chain islands can introduce testability issues [14, 15].

C) **Transmission/Reception Interfaces to Chip Top**: Some signals and pins to and from outside the SoC may never be exercised during manufacturing tests.

D) **Detection and correction logic circuits**: they usually include large logic functions resulting in deep circuits that are hard to target by structural tests [18].

E) **Complex hardware-software functions**, like complex protocol functions and synchronization mechanisms, are not exercised [18].

Therefore, an effective SLT suite is a combination of programs capable of systematically tackling all the aforementioned considerations. The next subsections provide guidelines to punctually address functional program generation according to the list of critical circuit elements.

#### 4.2.2.2 SLT functional program suite generation

The generation of the functional SLT suite, addressing the identified critical elements, starts from previously established functional methodologies in the state-of-the-art. State-of-the-art techniques like [52, 51, 50] cover some of the critical functionalities even using standard test configurations, such as exploiting loop-back configurations from different communication channels and typical frame transmissions/receptions. More efforts are required to complement structural coverage for modules not yet

very much mentioned in the literature, such as the Error-Management-Logic (EML), Bus-off sequence (if any), shared bus arbitration, and synchronization logic modules of communication peripherals.

The resulting suite, which encompasses known and novel approaches, has been developed manually by a test engineer using the SoC documentation, including structural test application notes, the netlist itself, and the list of residual faults from structural tests. The suggested suite consists of 6 test programs meticulously assembled to target overlooked areas during structural tests specifically. Out of this set of programs, 2 requires the collaboration of the companion module, which is indispensable to exercise the peripheral by transmitting and receiving packets.

In the following, the functional program specifications are reported in pseudo-code with generic communication peripherals functions that abstract the underlying software.

Moreover regarding the pattern selection, a generic function call $get_pattern()$ is used in the pseudo-code. As such the proposed algorithms can be executed multiple times with various data patterns coming from known literature-based ones such as from [76], where known techniques based on walking bits and checkerboard patterns are illustrated.

The data pattern could also be automatically generated from ATPG-based methodologies [76] or by adopting random methodologies if the cost of fault simulations is not too elevated. In these cases, a generation loop may be set up to refine step by step the pattern set, while a deterministic approach just requires a single evaluation.

In the following subsections, the communication software is described from an algorithm perspective and the companion module for each consideration in Figure 4.14. All the algorithms presented hereafter produce a signature, which is finally compared with the golden one at the end of every test execution.

##### 4.2.2.2.1 Embedded Memory access ports (A)

In SoC testing, embedded memory access ports may not be exercised in structural DfT circuits due to MBIST architecture [16]; when the MBIST is operated, it disconnects the memory from the other SoC elements and then it tests only a part of the connections among memory and the rest of the circuit.

Communication peripherals usually provide a dedicated memory to store data. Its functionality can be thoroughly tested by filling the memory with incoming messages. The proposed test must use all transmission/reception buffers if there are more than one. The CPU running the SLT program sends a burst of messages large enough to fill all buffers.

The test program pseudo-code is represented in Algorithm 1.

**Algorithm 1** Embedded Memory Access Port test.

```
1: signature ← 0                                              ▷ Init signature var
2: while peripheral_rx_buffers_are_full() ≠ True do
3:     data ← get_pattern()
4:     signarure ← signature ⊕ data
5:     send_data(data)
6: end while
7: wait_reception()
8: while peripheral_rx_buffers_are_empty() ≠ True do
9:     data ← receive_data()
10:    signature ← signature ⊕ data
11: end while
```

#### 4.2.2.2.2 Interfaces to other SoC components (B)

Communication interfaces to other SoC components like CPUs, DMAs, etc., are limited during structural tests. They are quite complicated and ATPG-resistant because they are composed of multiplexers, which are typically difficult to test, especially when a large-sized crossbar is included in the system. The situation gets even worse if the SoC modules involved in the communications are far from each other in the layout and fall into different LBIST islands [14] or Scan domains [15]. As the structural tests often need to care about power consumption, patterns are not applied to all gates simultaneously, but island per island or Scan domain by Scan domain.

In addition, complex communication protocols require coordination with digital/analog circuitry in other domains, leading to coverage loss for structural tests, for example, caused by analog IPs being bypassed during Scan tests.

The register interface serves as a collection of registers designed to control the peripheral's behavior, issue commands, store data for transmission/reception, configure additional peripheral units (i.e., timing management logic), and report

information from the peripheral controller, including status or interrupt registers. Reading and writing the registers ensures on-chip bus system usage and functional interactions between LBIST islands. The proposed methodology performs such operations by levering the peripheral under test in an active yet not fully initialized state.

Assuming **R** represents the configuration and control peripheral register file, the test can be succinctly expressed in pseudo-code in Algorithm 2.

**Algorithm 2** Interfaces to other SoC components test.

**Require:** $R$, the set of the peripheral register file.
1: $signature \leftarrow 0$ ▷ Init signature var
2: stop_peripheral()
3: assert_functional_reset_peripheral()
4: peripheral_enable_clock() ▷ Avoid init $r \in \mathscr{R}$
5: **if** $r$ is init_register **then**
6: $r \leftarrow r \wedge (1 \ll INIT_field_pos)$
7: $signature = signature \oplus r$
8: $r \leftarrow r \vee \sim (1 \ll INIT_field_pos)$
9: $signature = signature \oplus r$
10: **else**
11: $r \leftarrow 0$
12: $signature \leftarrow signature \oplus r$
13: $r \leftarrow UINT_MAX$
14: $signature = signature \oplus r$
15: **end if**
16:

Many modern micro-controllers offer additional hardware capable of asserting a reset on modules within the entire SoC to perform a clean communication restart. Therefore, a functional reset test may be necessary, which consists of stopping the peripheral, resetting it, reading back register values, and restarting the peripheral.

#### 4.2.2.2.3 Transmission/Reception Interfaces (C)

Transmission/Reception Interfaces to Chip Top are crucial for the overall functionality of transmitting and receiving messages to and from outside devices. Structural tests are very limited by the test access, as loop-back strategies are almost always used to reduce the number of pins to contact and control from the ATE. Therefore, to functionally exercise the Transmission and Receive logic, the Test Algorithm in 3

is used in conjunction with a companion module for sequencing messages (which details are described in Section 4.2.2.3).

The proposed test program targets the receive/transmission modules by initializing the peripheral to receive/transmit in different configurations. The test program pseudo-code is represented in Algorithm 3.

**Algorithm 3** Transmission/Reception to Chip Top test.

**Require:** $B_{tx}$, the set of transmission configuration modes.
**Require:** $B_{rx}$, the set of reception configuration modes.
**Require:** *shared*, data shared between ATE and the DUT.

```
 1: signature ← 0                                                    ▷ Init signature var
 2: enable_companion_module("message_sequencer") btx ∈ 𝓑tx brx ∈ 𝓑rx
 3: configure_peripheral(btx, brx)
 4: if is_transmission_enabled(btx) then
 5:     data ← get_pattern()
 6:     signature ← signature ⊕ data
 7:     send_data(data)
 8: else
 9:     signature ← signature ⊕ shared
10: end if
11: wait_reception()
12: data ← receive_data()
13: signature ← signature ⊕ data
14:
15:
```

#### 4.2.2.2.4 Detection and correction unit (D)

Many peripheral protocols are designed to be susceptible to faulty behaviors. As such, they implement an Error-Management-Logic (EML), which increases reliability and robustness by introducing detection and correction capabilities.

The detection functionality refers to the ability of the peripheral to assess when the transmitted/received data is wrong. It is usually implemented by protection codes inserted into the transmitted data. In addition to detection, correction logic enables the repair of erroneous data using error correction codes transmitted along the data. The correction is limited as it usually tolerates a maximum number of flipped bits.

Moreover, the bus error detection logic ensures that nodes can detect and report anomalies on the bus. This is achieved by identifying incorrect messages and faulty conditions, thus preventing any node from disrupting the bus with faulty outputs.

Additionally, malfunctioning nodes can recognize their errors and disconnect themselves from the bus, entering a *bus-off* state until they can re-synchronize and safely rejoin communication.

The test program pseudo-code is represented in Algorithm 4. A companion module capable of injecting errors into message frames is needed in this case. As detailed in Section 4.2.2.3, the injection can be done with different strategies.

**Algorithm 4** Detection and correction unit test.

```
1:  signature ← 0                                         ▷ Init signature var
2:  enable_companion_module("pattern_recognizer")
3:  data ← get_pattern()
4:  signature ← signature ⊕ data
5:  send_data(data)
6:  wait_reception_or_errors()
7:  if peripheral_has_errors() then
8:      data ← get_failed_transmitted_data()
9:  else
10:     data ← received_data()
11: end if
12: signature ← signature ⊕ data
13: if support_bus_off() then
14:     wait_bus_off()
15: end if
16: disable_companion_module()
17: data ← get_pattern()
18: signature ← signature ⊕ data
19: send_data(data)
20: wait_reception()
21: data ← receive_data()
22: signature ← signature ⊕ data
```

#### 4.2.2.2.5 Complex hardware-software functions (E)

The transmission logic assumes a pivotal role in handling the transmission of messages, specifically when a priority scheme is part of the peripheral protocol. When the protocol supports message priority, the transmission handler incorporates complex logic for instigating a so-called *transmission scan* to evaluate pending requests and pinpoint the highest priority.

The test program pseudo-code is represented in Algorithm 5.

**Algorithm 5** Complex transmission functions test.

**Require:** $ID_1, ID_2, ID_3$, s.t. $ID_1 > ID_2, ID_3 > ID_2$.

```
 1: signature ← 0                                   ▷ Init signature var
 2: data ← get_pattern()
 3: signature ← signature ⊕ data
 4: send_data(ID1, data)                            ▷ Send data with ID1
 5: data ← get_pattern()
 6: send_data(ID2, data)                            ▷ Send data with ID2
 7: data ← get_pattern()
 8: send_data(ID3, data)                            ▷ Send data with ID3
 9: cancel_transmission(ID2)
10: cancel_transmission(ID3)
11: wait_reception()
12: if rx_contains(ID2) ∨ rx_contains(ID3) then Failure
13: end if
14: data ← receive_data()
15: signature ← signature ⊕ data
```

Communication protocols can be based on a dedicated bus, connecting each node to another separately, or on a shared bus between nodes. The shared bus requires additional logic as nodes must implement functionality to perform the arbitration on the shared bus. Indeed, a node starting a transmission can face cases when the bus is idle (e.g., other nodes are not using the bus) or another node is using the bus. However, in the first case, only the active node will take the bus; bus sensing is not enough when $N$ nodes start transmission simultaneously. Modern protocols introduce an initial phase of transmission called the arbitration phase.

The arbitration logic can be tested by employing the proposed companion module in *message sequencer* mode requiring a shared ID among SLT and ATE. Assuming lower ID has higher priority, the test program pseudo-code is described in Algorithm 6, while the *message sequencer* mode is described in Section 4.2.2.3.

Finally, the timing of communication protocols must be tested. This test requires the implementation of strict synchronization, and the assistance of a companion module able to introduce an arbitrary delay into transmission frames is crucial.

#### 4.2.2.3 Companion module

Different companion modules can be hosted on the ATE FPGA; they can reside on the ATE at the same time with their inputs and outputs multiplexed. The correct

**Algorithm 6** Synchronization functions test.

**Require:** $ID$, predefined ID between ATE and this program.
1: $signature \leftarrow 0$ ▷ Init signature var
2: enable_companion_module(”*message_sequencer*”)
3: $data \leftarrow$ get_pattern()
4: $signature \leftarrow signature \oplus (ID+1)$
5: send_data($ID$, $data$) ▷ Send $data$ with $ID$ Companion senses a frame and transmits ID+1.
6: $data_ID \leftarrow$ receive_data_ID()
7: $signature \leftarrow signature \oplus data_ID$
8: $data \leftarrow$ get_pattern()
9: $signature \leftarrow signature \oplus (ID-1)$
10: send_data($ID$, $data$) ▷ Send $data$ with $ID$ Companion senses a frame and transmits of ID-1.
11: $data_ID \leftarrow$ receive_data_ID()
12: $signature \leftarrow signature \oplus data_ID$

functionality is activated, and communication is supplied along the execution of the SLT functional programs that need external support from the ATE.

The companion module functionalities needed to complement some of the SLT firmware running on the chip under test are several:

- Message reception and transmission from and to the peripheral under test, also called message sequencer.
- Error injection within the content of specific message segments transmitted to the peripheral under test.
- Delay injection during message transmission.

The message sequencer mode transmits predefined messages to the SoC. Pregenerated messages are stored onboard the companion module in the ATE and sent out according to the tested communication protocol. The companion module architecture for this mode consists of a central entity and a memory component. The messages can be hand-crafted, extracted from verification stimulus, or autogenerated pseudo-randomly. Upon receiving an enable signal, i.e., from a digital General Purpose I/O (GPIO), it transmits the message and returns to an idle state upon completion.

Three possibilities can be explored for error injection: error inject errors according to information already provided before running the test (i.e., a list of times to inject at); a pseudo-random injector that injects errors at random times and locations in the message data frame (i.e., a pseudo-random value determines injection times); a communication protocol injector that knows about the protocol characteristics and corrupts them on purpose (i.e., leaving start and stop bit for too long or short time).

Regarding delay injection, a companion module is required to introduce an arbitrary transmission delay. The introduced delay forces a de-synchronization that exercises the communication peripheral modules to adjust the communication timing finely. Moreover, error detection is also exercised when the delay introduces a misbehavior that cannot be corrected.

### 4.2.3 Uncore logic

Due to structural test limitations, this work focuses on developing a System Level Test (SLT) application tailored for testing the crossbar module. The crossbar, serving as the highway between various un-core logic in the System-on-Chip (SoC), is critical for ensuring seamless communication. This approach not only facilitates effective testing of the crossbar itself but also extends to the connected logic [64], including interfaces to/from cores, and un-core logic, such as peripheral bridges, peripherals, and memory controllers, etc. Conversely, the paper doesn't focus on SLT program generation for cores (e.g., CPUs, peripheral cores, user-defined modules, etc.).

In essence, the crossbar logic in a SoC acts as the highway for on-chip communication. A malfunctioning crossbar logic could send data down the wrong road, delivering it to unintended recipients (masters) and potentially changing the order of importance (priority) for different masters. This is why creating SLT workloads targeting the crossbar logic is vital. These applications would simulate real-world interactions between components to verify unintended escaped faults from structural tests. By generating SLT workloads for the crossbar module, it creates a ripple effect on the un-core logic.

In addition, a major challenge in developing these tests, or any general functional test program, is automation. The proposed methodology aims to fill the gap in terms of automatic test generation by providing an automatic generation methodology for

SLT workloads relying on an abstracted model of the SoC.

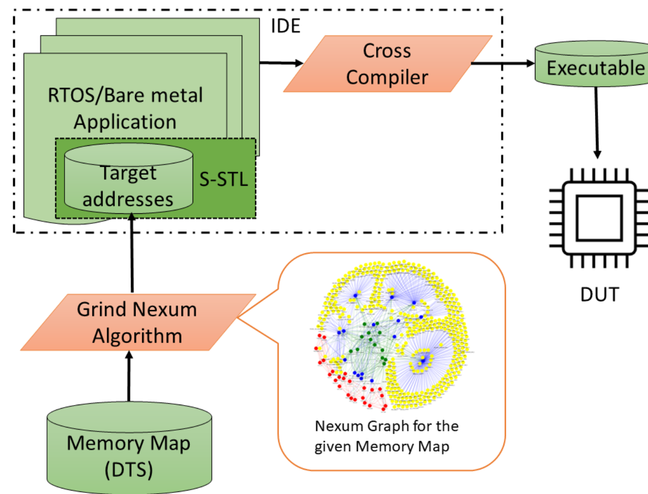



Fig. 4.15: Proposed methodology workflow.

Figure 4.15 shows the generation methodology. The efforts are put especially on the crossbar modules (i.e., the interconnection matrix and the relative bridges).

The proposed method is based on a *Grind Nexum algorithm* that exploits an internal graph representation of all the components in the SoC, including CPUs, DMAs, and peripherals. The graph is built by starting from a memory map description. Meanwhile, crossbar master and slave ports are modeled as virtual vertex to allow a more leisurely visit of the model. In the approach, the devised graph data structure is visited to cover all the edges or nodes (i.e., the connections between each component), and to produce a list of addresses and functions for which masters read and/or write to slaves memory locations.

The targeted addresses are generated as a data structure encapsulated into a library called *System Software Test Library* (S-STL or $S^2$TL). The aforementioned library can be integrated into a Real-Time Operating System or a bare metal application and executed on the Design Under Test (DUT).

The following subsections describe the proposed SoC model, the *Grind Nexum algorithm*, and the $S^2TL$.

#### 4.2.3.1 Proposed System-on-Chip model

The presented subsection focuses on abstracting the SoC's architectural components to a graph's vertexes. Every component of the SoC, master or slave, is represented with a vertex in the graph connected to a virtual vertex representing the master/slave port of the crossbar, where every master is connected to every slave. Eventual internal memory elements in the masters are further added to the model, representing a fully connected internal graph.

More in detail, a CPU may contain a private cache and RAM strongly tied to the CPU registers. Therefore, its graph-based representation has to take into account all those connections.

The proposed SoC model can be analytically represented with the following formulas, where $X$ denotes the bipartite graph between virtual masters and slaves, and $N$ the graph produced by the union of the graph $X$, and the graph representing the peripherals, slaves, and masters.

$$X = (P_{Master}, P_{Slave}, C) \tag{4.7}$$

$$C = \{\forall x \in P_{Master} \exists (x,y), x \neq y, \forall y \in P_{Slave}\} \tag{4.8}$$

Equation 4.7 represents the bipartite graph modelling the crossbar master ports ($P_{Master}$) and the slave ports ($P_{Slaves}$). Meanwhile, Equation 4.8 models all the edges from every master to every slave without any self-loop.

$$N = \cup\{(V,E), X\} \tag{4.9}$$

$$V = \left\{ \bigcup_{i=0}^{M} Master_i, \bigcup_{j=0}^{S} Slave_j, \bigcup_{k=0}^{L} ResRegion_k \right\} \tag{4.10}$$

Equation 4.9 describes the union of the graphs modeling the crossbar and the one modeling the peripherals, slaves, and masters. In equation 4.10, there is the set of all possible vertexes of the SoC, the CPU-addressable reserved region and slaves, and the vertexes capable of acting as a master on the crossbar.

$$Master_i = \{\forall x, y \exists (x,y), \exists (x,y,z) | z \in X\} \tag{4.11}$$

As Equation 4.11 shows, a crossbar master can be composed of several sub-vertexes (*x,y*) such as internal registers, internal memories and internal caches with self-loop capabilities. In addition, internal vertexes of a given master are fully connected, and a vertex exists within a master, which is also present in the graph described by Equation 4.7. Regarding the *Slaves*, or *Reserved Region*, Equation 4.12 shows how the other sets are built.

$$Slave_j = \{\forall x, y \exists (x,y), \exists (z,x,y) | z \in X\} \tag{4.12}$$

$$E = \{\exists (x,z,k,y) | x,y \in V, z,k \in X, \tau(x,y) \neq \emptyset\} \tag{4.13}$$

On the other hand, Equation 4.13 models the edges between a master and a slave, including the virtual vertexes of the master and slave ports of the crossbar and their type of operation described by Equation 4.14 depending on the vertex type.

$$\tau(x,y) = \begin{cases} \text{Cpu} & \text{x a CPU regs, y CPU regs} \\ \text{Peripheral} & \text{x a CPU regs, y a Peripheral} \\ \text{Flash} & \text{x a CPU regs, y a Flash} \\ \text{Ram} & \text{x a CPU regs, y a Ram} \\ \text{Ram_fetch_instr} & \text{x a ICache, y a Ram} \\ \text{Flash_fetch_instr} & \text{x a ICache, y a Flash} \\ \text{Icache} & \text{x a ICache, y generic} \\ \text{Dcache_exception} & \text{x a DCache, y a Reserved/Peripheral} \\ \text{Dcache} & \text{x a DCache, y generic} \\ \text{DMA_32bit_transfer} & \text{x a Ram, y a Peripheral} \\ \text{DMA_No_wait} & \text{x a Ram, y Reserved} \\ \text{DMA} & \text{x a Flash, y a Ram} \end{cases} \tag{4.14}$$

The conditions for the type of operation retrieved from Equation 4.14 are specular to provide the correct operation type, i.e., $\tau(x,y) = \tau(y,x)$. The order implies only a read or write operation. For example, a *Peripheral* type of operation when $x$ is a CPU register and $y$ is a peripheral, it is a write to the peripheral control registers. Meanwhile, when $x$ is a peripheral and $y$ is a CPU register, it is a read to the peripheral control registers. Additional care is needed if a CPU instruction master port is present. In order to be exercised, the executed code must be relocated to the given address.

For example, suppose to have an SoC Architecture as the one represented Figure 4.16; every Core has an internal private cache and, naturally, CPU registers, and it is connected to the crossbar through a single master port.

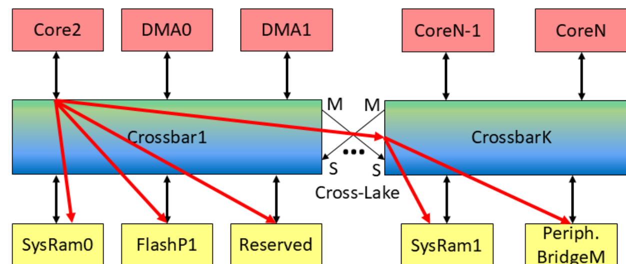



Fig. 4.16: Example architecture with N masters (red), M slaves (yellow), and K crossbars interconnected by a cross-lake.

The presence of virtual vertexes representing the master/slave port of the crossbar enables further analysis as well as the creation of all the possible connected paths from a master to all addressable slaves, as it happens architecturally. For example, in Figure 4.16, red arrows represent all the possible logical paths on the crossbar from master *Core2*. Therefore, the graph model generated from Figure 4.16 is shown in Figure 4.17 using only two crossbars connected by a cross-lake for readability purposes.

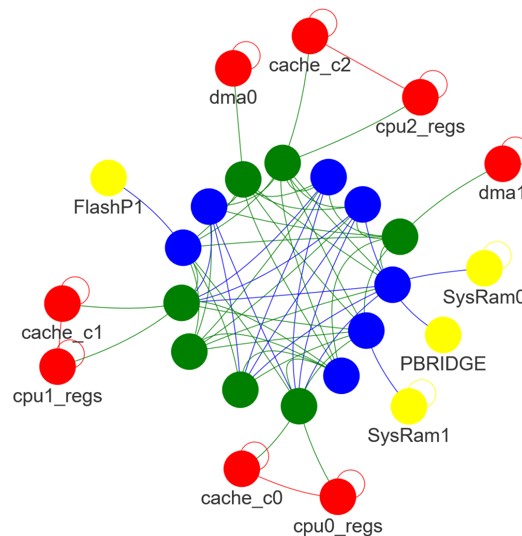



Fig. 4.17: Generated graph model for the example SoC architecture with two linked crossbars, crossbar master ports (green) and slave ports (blue).

For the sake of this work, the generated graph is referred to as *Nexum graph* (in Latin, *Nexum* means connections), representing the connection between every component in the SoC.

Vertexes in Figure 4.17 are colored depending on the nature of the components:

- Red, vertex (component) capable of acting as master on the crossbar.
- Green, virtual vertex modeling the master bus port.
- Blue, virtual vertex modeling the slave bus port.
- Yellow, vertex (component) capable of acting as a slave on the crossbar.

The above reasoning can be easily extended to any number of masters and slaves, independently from the technology or the bus protocol. If a component is simultaneously a master or slave, master color has the predominant role. Moreover, for a given number $M$ of masters and $S$ of slaves, all possible functional paths between each master and slave can be calculated with the following formula:

$$ConnectionPath = M * S \tag{4.15}$$

The proposed SoC model is used in the hereinafter proposed methodology to analyze and generate for each possible master read/write operation on memory addresses (RAM, ROM, peripherals, and reserved partitions).

#### 4.2.3.2 Grind Nexum algorithm

The *Grind Nexum algorithm* is responsible for creating a graph-based representation of the SoC starting from a memory map file. Afterward, it visits the graph to cover all the nodes or edges and generates a list of target addresses and functions to execute on the DUT. In the following subsections, every algorithm step is described in detail.

##### 4.2.3.2.1 Creating the model

An essential aspect of the proposed methodology is the memory map file used as input to the overall generation workflow.

The Memory map file can be generated from the user manual with manual efforts, or exploiting generative AI, to model memory regions (for slaves) and master present in the SoC; the output format can be a custom CSV file. The manual generation of the SoC memory map file is time-consuming and error-prone. To address this challenge, *Device Tree Source* (DTS) files can be used, which are a standardized method for describing SoC hardware in the embedded Linux domain [77]. DTS files are typically already produced and validated as part of the platform development process, making them a reliable foundation for representing SoC information. This approach not only reduces manual effort but also minimizes the risk of errors during the creation process. By utilizing DTS, the representation and usage of SoC information become more streamlined and scalable, offering a robust solution for modeling hardware efficiently.

The Memory map file acts as a comprehensive hardware description document, detailing various components such as CPUs, memory regions, peripherals, and other devices. It includes specific information about address ranges, sizes, access types, clock frequencies, interrupt mappings, and other device-specific settings, such as the attached master or slave ports and other possible communication paths between on-chip components. This level of detail is essential for the proper initialization and operation of the hardware. One of the key functions of a Memory map file is to describe non-discoverable hardware, including custom peripherals and specific memory mappings. This is particularly important for unique hardware configurations that the operating system cannot automatically detect. The file is written in a human-readable format, which allows developers to easily customize and modify the hardware description as needed, facilitating debugging and maintenance of the hardware configuration. This approach simplifies the process of hardware configuration and ensures that the system can correctly interact with all its components. Overall, this Memory map file serves as a crucial blueprint for the modeling an SoC, providing a clear and detailed representation of the system's hardware layout, capabilities, and interconnections.

All the information contained in the Memory map file is consolidated by Algorithm 7 to construct the SoC's undirected graph model, known as the Nexum Graph. The algorithm creates a vertex for each addressable region—both reserved

and non-reserved—embedding key attributes such as address ranges, sizes, etc. (lines 3–17 of Algorithm 7). Afterward, it establishes functional links between on-chip components using defined communication paths (lines 22-32). Since these functional communication paths are bidirectional (with crossbar masters able to both read and write), the resulting SoC graph model is undirected.

**Algorithm 7** Nexum Graph model generation

```
Require: Compliant Memory Map File
 1: xbar_graph = ∅, nexum_graph = ∅
 2: Load memory map from file in mem_map
 3: for each: mem_region ∈ mem_map
 4: if Type(mem_region)=XBAR then
 5:     add_node(mem_region, xbar_graph)
 6: else
 7:     add_node(mem_region, nexum_graph)
 8: end if
 9: if Type(mem_region)=CPU then
10:     cores_edges[mem_region].append(mem_region)
11: end if
12: if Type(mem_region) = no_xbar_connection then
13:     no_xbar_connection.append(mem_region)
14: end if
15: for each: P_master ∈ xbar_graph
16:     for each: P_slave ∈ xbar_graph
17:     xbar_graph.add_edge(P_master, P_slave)
18: for each: v ∈ nexum_graph
19: if Type(v) = RAM then
20:     nexum_graph.add_edge(v, v)
21: end if
22: for each: link ∈ v.masters_slaves_ports
23: links = v.masters_slaves_ports[link]
24: nexum_graph.add_edge(v, links, xbar_graph)
25: for each: core ∈ cores_connection
26:     for each: v_1 ∈ cores_connection[core]
27:         for each: v_2 ∈ cores_connection[core]
28:         nexum_graph.add_edge(v_1, v_2)
29: for each: no_xbar_connection ∈ no_xbar_connection
30:     v_1, v_2 = no_xbar_connections
31:     nexum_graph.add_edge(v_1, v_2)
32: return nexum_graph
```

It is important to emphasize that the master, slave, and crossbar ports of the crossbars are modeled as virtual nodes in the graph, effectively representing an addressable region within the SoC's entire address space. The graph generation process begins with the construction of an internal crossbar graph where every master node is connected to every slave node. Once this initial connectivity is established, the algorithm proceeds to link each master and slave node to their corresponding crossbar ports. This two-step process, starting from the comprehensive master-to-slave connectivity and then integrating the crossbar ports (as shown in lines 15–17 of Algorithm 7), ensures that the final Nexum graph accurately encapsulates both the functional and physical interconnections of the SoC.

Moreover, there are special nodes that require additional care. For example, CPUs may contain additional subunits such as cache, local private memories, and CPU registers. These components are represented as vertices in the graph and are associated with the same CPU master port of the crossbar. They are interconnected by dedicated functional communication paths—bypassing the crossbar—that enable direct data transfer, as illustrated in line 25 of Algorithm 7. Furthermore, these nodes feature self-loops, demonstrating their ability to internally transfer or store data (for instance, moving data from one register to another or between memory words in RAM), as shown in lines 19–21 of Algorithm 7.

Regarding reserved memory regions, they are modeled and visited as vertex, sharing similarities within the memory region where they lay. For example, a reserved region after an addressable RAM region is considered an additional RAM vertex. In addition, in real architecture, there are reserved master and slave ports in the crossbar. They are not modeled without any loss of generality since they will never be exercised in the field. Moreover, certain communication paths between masters and slaves bypass the crossbar and are modeled as additional edges directly in the memory map file (see lines 12–14). These extra edges are then incorporated into the final SoC graph model during its construction (see lines 29–32). This approach ensures that all direct communication routes (those not mediated by the crossbar) are accurately represented, providing a complete depiction of the data flow within the system.

#### 4.2.3.2.2 Visiting the model

Creating the graph models comes with a relatively easy computation because it is just a matter of parsing an input file and creating the associated vertex and their relative edges. When the graph is generated, the tool must visit the graph.

The graph visit is an essential, automated phase of the proposed methodology for generating the target addresses data.

Once the *Nexum graph* has been created, the *Grind Nexum algorithm* can visit the graph model in different ways:

- Visiting all the nodes starting from master nodes as node set and slave nodes as target set using a Deep First Search, independently from the edges (means of transfer), as Algorithm 8 describes.

- Visiting all the edges starting from master nodes using a Deep First Search, as Algorithm 9 describes.
- Visiting all the nodes starting from slave nodes as node set and master nodes as target set using a Deep First Search, independently from the edges (means of transfer), as Algorithm 8 describes.
- Visiting all the edges starting from slave nodes using a Deep First Search, as Algorithm 9 describes.

The performance complexity of visiting all the nodes, or the edges, is $O(|V|+|E|)$, with $V$ and $E$ being, respectively, the number of vertexes and edges in the *Nexum Graph*.

For node visits in Algorithm 8, the traversal is not carried out to explore every possible vertex in the graph in an exhaustive manner, but rather to efficiently locate target nodes (either masters or slaves) and record a corresponding data when a target is found. In this context, the DFS-like approach helps in quickly reaching a target node along one branch before exploring others.

In Algorithm 8, the only substantial difference in starting from master or slave nodes as node set resides in maximizing the bus traffic and contention on the crossbar ports, from masters it results that multiple slaves are exercised from the same master port, from slaves it results that multiple master are exercising at the same time the slave port.

Starting the visit from master nodes is going to maximize the traffic on the master port of the crossbar; meanwhile, starting the visit from slave nodes is going to maximize the traffic on the slave port connected to a given slave, i.e., multiple masters are accessing at the same time the same slave, and as a consequence, they must be properly synchronized.

At the end of a visit, the algorithms generate a list of *Actions*. An *Action* is a defined data structure containing a source address, a destination address, and a function pointer to be applied to the source and destination address on the DUT. The resolution of the function pointer is made accordingly to Equation 4.14 (i.e., it depends on the source and destination node's type). The *Nexum Graph* is an undirected graph. Therefore, for each *Action*, read and write operations are generated between source and destination. Furthermore, every node contains a memory range

of a given size that would be impossible to read and write entirely for every node in the *Nexum Graph*. In order to effectively address this issue, based on the approach proposed in [74], addresses are generated at half of a given address range for each source and destination, and in that interval of words read and write operations are carried out. Regarding peripherals, read and write operations are generated according to their addressable registers.

A detailed view of the algorithm for visiting all the nodes from multiple masters can be seen in Algorithm 8.

**Algorithm 8** Node Visit Function for a Node set (masters or slaves)

```
Require: Nexum_Graph, Xbar_Graph
Require: Node_Set, Target_Set, Action_Direction
 1: for each: node ∈ Node_Set
 2: root = node
 3: if root ≠ Reserved & root ∉ Xbar_Graph then
 4:     actions[root] = []
 5:     visited = set(); sink = None
 6:     visited.add(root)
 7:     connections = Nexum_Graph[root]                ▷ Nodes connected to root
 8:     while connections ≠ ∅ do
 9:         current_node = connections.pop()
10:         if current_node ∈ visited then
11:             continue
12:         end if
13:         visited.add(current_node)
14:         if current_node ∈ Target_Set then
15:             sink = current_node
16:             if Action_Direction == "master_to_slave" then
17:                 actions[root].append(Action(root, sink))
18:             else
19:                 actions[sink].append(Action(sink, root))
20:             end if
21:         end if
22:         connections.extend(Nexum_Graph[current_node])
23:     end while
24: end if
25: return actions
```

First, all the functional paths from the *Root* master to every slave are visited. Afterward, the tool can start the visit from another master with the same approach, visiting all the slaves and generating *Actions* accordingly.

An important aspect to consider is the maximization of concurrency that could happen on the bus. In order to maximize the concurrency, internal paths from the virtual master port to the virtual slave port of the crossbar are exercised multiple times and concurrently by different masters.

Meanwhile, node maximization could ensure that all the functional units are touched at least once. It does not guarantee the functional paths to and from the

bus, especially on those nodes where a self-loop exists, e.g., RAMs. A DMA or a CPU could exploit RAM self-loops to maximize the bus traffic. Self-loops are not considered in the node visit. Instead, they can be visited from the edge visit represented in Algorithm 9.

**Algorithm 9** Combined Edge Visit Function for Masters and Slaves

```
Require: Nexum_Graph, Xbar_Graph
 1: for each: node ∈ (masters ∪ slaves)
 2: root ← node
 3: if root ≠ Reserved ∧ root ∉ Xbar_Graph then
 4:     actions[root] ← []
 5:     visited_nodes ← ∅
 6:     visited_edges ← ∅
 7:     visited ← {root}
 8:     sink ← None
 9:     stack ← [root]
10:     edges ← ∅
11:     while stack ≠ ∅ do
12:         current_node ← stack.pop()
13:         if current_node ∉ visited_nodes then
14:             edges[current_node] ← Nexum_Graph[current_node]    ▷ Retrieve the edges of the current node
15:         end if
16:         if edges[current_node] = ∅ then
17:             stack.pop()                    ▷ No more edges from the current node, so backtrack
18:         else
19:             edge ← next(edges[current_node])
20:             if edge ∉ visited_edges then
21:                 visited_edges.add(edge)
22:             end if
23:             if (root ∈ masters ∧ current_node ∈ slaves) then
24:                 sink ← current_node
25:                 actions[root].append(Action(root, sink))
26:             else if (root ∈ slaves ∧ current_node ∈ masters) then
27:                 sink ← current_node
28:                 actions[sink].append(Action(sink, root))
29:             end if
30:             sink ← current_node
31:             stack.push(edge[0], edge[1])
32:         end if
33:     end while
34: end if
35: return actions
```

More in detail, edge visit algorithms are exploring the *Nexum Graph* until they find a slave or no more edges to push on the stack data structure. In particular, in Algorithm 9, in lines 25, the edge data structure comprises a tuple of two nodes to be added to the stack structure.

By following the approach in Algorithm 9, the idea is to generate the minimum number of trees to touch at least once every edge. In this strategy, the algorithm processes edges sequentially within each node's list, while the DFS stack governs the overall traversal order. From a testing perspective, this means that functional paths, such as a CPU-to-RAM connection and a CPU-to-Peripheral connection are treated

equivalently, even though the DFS approach imposes a specific order. Importantly, the ordering of edge visits can be adjusted as needed to prioritize or specifically isolate certain paths for testing purposes. In this way, the methodology can ensure that all the DUT functional paths and on-chip components are exercised.

#### 4.2.3.3 The System-Software Test Library

A significant concern when developing an SLT workload, or in general, a functional test program, is the lack of automation in generating such test programs. Therefore, their development and management are entrusted with testing engineers, who must understand the SoC architecture and related CPU Instruction Set Architecture (ISA). Consequently, a portion of developing time is dedicated to building a software infrastructure capable of communicating with a bare metal application or an operating system. Eventually, porting the previously developed functional test programs to different operating systems, ISAs, and SoC architectures may become cumbersome due to the missing separation of SoC-independent software.

This section aims to provide a general software architecture for a *System Software Test Library* (S-STL or $S^2$TL) in order to reduce as much as possible the porting time and, at the same time, let test engineers focus only on the functional test program itself. Figure 4.18 shows the organization of the library mentioned above.

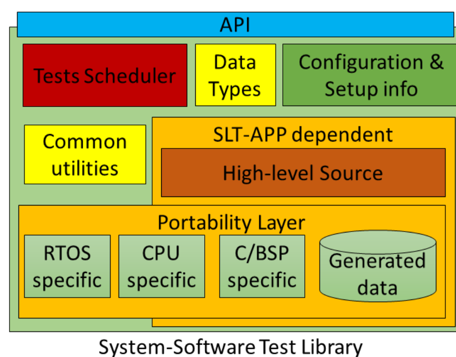



Fig. 4.18: Software modules decomposition of a $S^2$TL.

Figure 4.18 depicts a clear separation between high-level functions and data, data types, and configuration directives, which are all SoC and ISA independent from the SoC and ISA dependent part in the portability layer. The portability layer contains CPU-specific functions, a board/chip support package (C/BSP), and Real-Time

Operating System services. Following this separation, modifying only the portability layer when migrating to a different SoC architecture or ISA is sufficient for an SLT workload.

By organizing the library as Figure 4.18 shows, it is possible to easily integrate further SLT approaches by providing the necessary software without rewriting the functional test program from scratch. The library must encapsulate a runtime test dispatcher for all the available cores. Despite separating internal modules, the $S^2$TL can be accessed only by a bare metal application or an RTOS through the Application programming interface (API).

The $S^2TL$ implements the runtime test dispatcher specific for the generated target address by the aforementioned methodology. It includes parsing the generated data structure in a high-level programming language and calling the proper function pointers. It also synchronizes the cores while executing the tests and preparing the DUT.

The so-called *Actions* are SoC and CPU dependent, and they are implemented between different sources and destinations (such as CPU registers, RAMs, DMAs, Peripherals, and reserved regions) in the portability layer.

**Algorithm 10** Pseudo code for a generic Action.

**Require:** Source, Destination, Error Pointer
**Ensure:** Test data in Source
1: Move Test Data from Source to Destination
2: Read Test Data from Destination
3: **if** Data are not saved correctly **or** (generated Exception **and** Expecting Exception) **then**
4:     **return** notOK
5: **end if**
6: **return** OK

Pseudo Code 10 depicts the general philosophy of an *Action*, i.e., move data from a source to a destination and verify its behavior with the expected one. The actions executed by the system software test library primarily involve data movement between a source and a destination. However, the nature of this data movement depends heavily on the type of destination, which is why a portability layer is implemented to bind the library to the underlying architecture. For instance, data movement between CPU registers and RAM is relatively straightforward and typically does not pose significant challenges. In contrast, data movement between CPU registers and peripheral registers requires careful consideration. If the peripheral

is not properly configured, such operations could trigger exceptions or unintended behavior. Similarly, when using a DMA controller for data movement, it must be correctly configured to avoid issues such as data corruption or system instability. Regarding reserved memory area, the library must capture the generated and expected exception.

Additionally, the choice of data patterns used during data movement can significantly impact fault coverage. Certain patterns may better expose faults in the communication paths or hardware components, while others might fail to trigger specific edge cases (out of scope for this work).

#### 4.2.3.4 Assessment on structural weaknesses

The SLT procedures described so far are generated automatically based on some holistic assumptions. They are intended to detect what remains uncovered after applying comprehensive structural scan-based tests, strongly focusing on the architectural interconnections. To assess analytically whether the holistic generation is effective in bridging the inherent issues of structural testing, an analysis of the residual (non-detected) faults not covered by these tests is illustrated. Indeed, scan-based patterns suffer of some weaknesses in covering specific types of faults. For example, if a fault lays in the intersection of flop logic cones that are triggered by different clock domains, covering that fault would require more scan patterns, specifically generated to target those regions. The same behavior happens for faults laying in the intersection of flop logic cones belonging to different chain configurations.

Upon such considerations, the paper proposes a categorization of those uncovered faults. Specifically, an uncovered fault could fall into one of the following categories:

A. Clock-domain crossing: faults in the intersection of flop logic cones that are not triggered by the same clock domain (Algorithm 11).

B. Chain configurations crossing: faults in the intersection of flop logic cones that do not belong to the same chain configuration(Algorithm 11).

C. ATPG critical: faults that are difficult to observe because they are linked to a few scanned flops (Algorith 12).

D. Functionally untestable: faults that are impossible to excite and observe during a normal functional program execution such as debug circuitry and test enables

for FFs, they are identified with the approach proposed in [78] and introduced as constant constraints in the fault simulator.

The algorithms used to assign the categories are explained below. Algorithm 11 can be used for fault extraction in clock-domain and chain configuration crossing categories. Upstream of these algorithms, it is a requirement to possess the list of circuit gates, with each of them's flop endpoints, along with the information on the clock domains and chain configuration to which they belong. As for the ATPG critical category, Algorithm 12 describes the extraction based on the faults not covered by the structural patterns in use. This way, the algorithm extracts the faults with a higher weight than the average one.

**Algorithm 11** Finding faults between different scan chain configurations or clock domains

```
Require: gates, list of gates including inputs, outputs and corresponding endpoints
 1: faults ← ∅
 2: for each g in gates do
 3:     scan_conf_in ← ∅
 4:     for each inp in g.inputs do
 5:         for each endp in inp.endpoints do
 6:             chain_conf_in.add(endp.scan_conf)
 7:         end for
 8:     end for
 9:     scan_conf_out ← ∅
10:     for each out in g.outputs do
11:         not_found ← FALSE
12:         for each endp in out.endpoints do
13:             scan_conf_out ← endp.scan_conf
14:             if endp.scan_conf not in scan_conf_in then
15:                 not_found ← TRUE
16:             end if
17:         end for
18:         if not_found is TRUE then
19:             faults.add(out)
20:         end if
21:     end for
22:     for each inp in g.inputs do
23:         not_found ← FALSE
24:         for each scan_conf in scan_conf_in do
25:             if scan_conf not in scan_conf_out then
26:                 not_found ← TRUE
27:             end if
28:         end for
29:         if not_found is TRUE then
30:             faults.add(inp)
31:         end if
32:     end for
33: end for
34: return faults
```

Once all the remaining faults not covered by the structural tests are categorized, it is possible to highlight in which categories the SLT was most successful in bridging

the structural weaknesses of the DfT in complex case studies. In this way, the analysis makes it possible to verify that the holistic and automatic generation of SLT programs in the proposed methodology succeeds in its intent to cover more of the critical structural testing faults.

**Algorithm 12** Extracting ATPG critical faults depending on the fan-in/fan-out ratio

**Require:** *gates*, list of gates including inputs, outputs and corresponding endpoints
**Require:** *residual_faults*, set of faults not detected by structural-based tests

1: $atpg_critical_faults \leftarrow \emptyset$
2: **for each** $g$ in $gates$ **do**
3: $\quad fan_in \leftarrow 0$
4: $\quad$ **for each** $inp$ in $g.inputs$ **do**
5: $\quad\quad fan_in \leftarrow fan_in +$ len($inp.endpoints$)
6: $\quad$ **end for**
7: $\quad fan_out \leftarrow 0$
8: $\quad$ **for each** $out$ in $g.outputs$ **do**
9: $\quad\quad fan_out \leftarrow fan_out +$ len($out.endpoints$)
10: $\quad$ **end for**
11: $\quad g.ratio \leftarrow \frac{fan_in}{fan_out}$
12: $\quad faults$.add_with_ratio($g.inputs$, $g.ratio$)
13: $\quad faults$.add_with_ratio($g.outputs$, $g.ratio$)
14: **end for**
15: $unique_ratios \leftarrow$ get_unique_ratios($faults$)
16: $tot_occurrences \leftarrow \emptyset$
17: **for each** $not_det$ in $residual_faults$ **do**
18: $\quad not_det_ratio \leftarrow faults[not_det].ratio$
19: $\quad tot_occurrences[not_det_ratio]$.increment()
20: **end for**
21: $avg_occurrences \leftarrow \frac{tot_occurrences}{\text{len}(tot_occurrences.\text{keys}())}$
22: **for each** $fault$ in $residual_faults$ **do**
23: $\quad$ **if** $fault.ratio \geq avg_occurrences$ **then**
24: $\quad\quad atpg_critical_faults$.add($fault$)
25: $\quad$ **end if**
26: **end for**
27: **return** $atpg_critical_faults$

## 4.3 Automatic generation of System-Level Test workloads

A weak point of SLT, or in general, of functional test programs, is how to automatically generate such applications without investing too much effort and skills from test engineers.

### 4.3.1 Leveraging High-Level Models

High-level models and graph traversal algorithms can automatically generate SLT workloads to verify the uncore logic, as described in previous sections. The proposed methodology abstracts the SoC architecture into a graph-based model called the "Nexum graph," each component (CPU cores, peripherals, memory regions) is represented as a vertex, and the connections between them are modeled as edges. The "Grind Nexum algorithm" is responsible for creating the graph model from a memory map file and then traversing the graph to generate a list of target addresses and functions to be executed on the DUT.

The graph can be traversed in different ways, such as visiting all nodes or edges starting from master or slave nodes to maximize bus traffic and contention on crossbar ports. The generated target addresses and functions are encapsulated in a data structure used by the "System Software Test Library," a standardized software module that can be integrated into a bare-metal application or a Real-Time Operating System (RTOS) and executed on the DUT. The $S^2TL$ provides a clear separation between SoC-independent and SoC-dependent software modules, making it easier to port the test programs to different architectures and Instruction Set Architectures (ISAs).

By using high-level models (enhanced with on-chip components) and graph traversal algorithms, the proposed methodology automates the generation of SLT workloads, reducing the development time and effort required from test engineers.

### 4.3.2 Leveraging non-functional properties

Non-functional properties, e.g., temperature, play a major role in SLT; a test program is executed in such an extreme environment that it can capture defects that could happen during mission mode but not during structural tests [17]. Reaching a target non-functional property, for example, the temperature, is a challenging task as it must respect the operating requirements to avoid overtesting [54]. Therefore, executing structural tests to reach a given temperature could destroy the device due to excessive switching activity, leading to thermal runaway and potentially destroying the DUT. On the other hand, a target temperature can be reached using functional programs, such as during the System-Level Test.

The focus is on the automatic generation of assembly test programs for SLT that aim to indirectly maximize a particular non-functional property, i.e., the temperature, of the DUT, using a genetic programming framework, such as *MicroGP* [79]. The targeted non-functional property is the temperature. The aim is to maximize power consumption instead as it is proportional to temperature. Power consumption is faster and cheaper to measure compared to temperature [80].

The method is based on a two-step generation approach, as Figure 4.19 shows:

- A fast architectural simulation provides a structure for the test programs by maximizing the Instructions Per Cycle (IPC).
- An additional step is done using actual hardware measurements of a processor by fine-tuning the data. The aim is to maximize power consumption by fine-tuning the program's initial register contents.

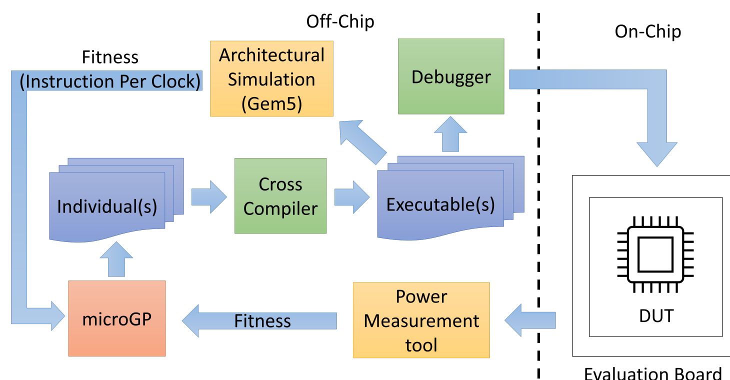



Fig. 4.19: Proposed approach workflow.

The architectural simulation generates the instructions in the `snippet` block; meanwhile, the data loaded in the registers from the data section are tuned in the second phase, exploiting the hardware implementation by executing the previously generated programs.

The following subsections describe the microarchitectural simulation and power measurement generation steps in detail.

#### 4.3.2.1 Microarchitectural Simulation

The microarchitectural simulation is the first generation step of the proposed approach. The architectural simulation is achieved using the `Gem5` [81] simulator. It is a fast and flexible architectural simulator for extracting architectural-related statistics, e.g., number of clock cycles for executing the program, cache misses, wrong branch predictions, Instructions Per Cycle (IPC), etc. The focus of the microarchitectural simulation is to generate the structure of the snippet in Figure 4.21 to maximize the IPC.

The general idea is represented in Figure 4.20.

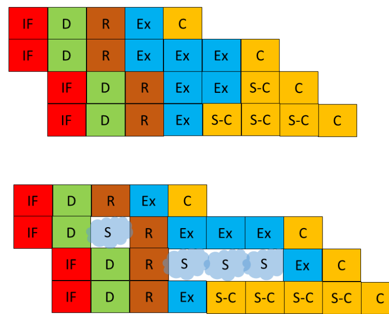



Fig. 4.20: Pipeline example without and with stalls.

The basic assumption is that a CPU pipeline without any stalls (upper part of Figure 4.20) uses more functional units in the core and consumes more power. On the other hand, stalls in the CPU pipeline introduce wait states between every stage of the pipeline (lower part of Figure 4.20), thus lowering the amount of concurrent functional units used and distributing the pipeline workload in time.

As confirmed by [82], a full CPU pipeline consumes more power. From an architectural perspective, a measurement of how much the pipeline and its functional units are used is the IPC. Therefore, the IPC extracted from the architectural simulation could be a good feedback fitness for the genetic framework to maximize power consumption.

This step allows quick generation of programs without involving hardware or hardware measurements which are then used as input for the next step: the initial register content optimization via power consumption.

```
...
.text
_reset_address:
/*startup phase*/
...
_registers_preparation:
/*integer registers*/
/*floating point registers*/
snippet:
/*stress code*/
...
j snippet
.data
/*data for floating point registers*/
/*data for integer registers*/
```

Fig. 4.21: Stress program structure.

#### 4.3.2.2 Tuning the Initial Register Content

The second important step of the proposed approach is fine-tuning the initial register values for the test program, i.e., the data section in Figure 4.21.

Tuning the initial contents for the integer and floating point registers used in the stress programs is a crucial technology and design-dependent step. Therefore, it must be executed by measuring the on-chip power consumption.

This step starts from stress programs generated by the previous step. Afterward, the genetic framework tunes the input data using the fitness of the on-chip power consumption, and at the same time, it maintains the instruction structure. The general idea is to evolve to the best stress programs with the highest power consumption and IPC.

## 4.4 Assessment

The Achilles' heel of SLT is the need for quality metrics to assess the goodness of an SLT workload. The evaluation process of the SLT workload could fall back into the normal grading approach used for developing Software-Based Self Test [28]. However, with the rising complexity of modern automotive devices, the RTOSes, and the applications, normal grading approaches may be prohibitive regarding computing time during the early development stages.

Table 4.2 shows a qualitative comparison of possible proposed assessment techniques for SLT workloads.

| Method | Pros | Cons |
|---|---|---|
| Trace based | Fast grading<br>Extracted from the device/ISA sim | Circumscribed controllability<br>and observability to programmer level registers |
| Toggle analysis | Detailed controllability | Medium time expensive<br>Netlist knowledge |
| Fault simulation [28] | Detailed observability and controllability of faults | Time expensive<br>Netlist/technology knowledge |

(Execution Time ⇓ ; Abstraction ⇑)

Table 4.2: Comparison of different possible evaluation methods for SLT workload.

Fault simulations are the best evaluation method for a detailed understanding of faults behavior at the cost of knowledge of netlist, technology, and time expensiveness. By increasing the abstraction, the evaluation method can resort to toggle analysis focusing only on the controllability details at the medium cost of time expensiveness and netlist knowledge. In order to further reduce the execution time, the evaluation methods can resort to trace-based metrics, which are fast to compute since they are extracted directly from the manufactured devices or an ISA simulator, but at the cost of a circumscribed controllability and observability of programmers' level registers.

Functional test program creation requires manual and/or iterative efforts and is usually graded by using fault simulation tools [28]. Unfortunately, the fault simulations are exceptionally time-consuming [83], forcing test engineers to wait for a significant amount of time and, in case of unsatisfactory results, iterate the entire process, even with automated generation engines [29].

Due to their requirements of exercising the device as a whole, combined with long test execution time, SLT workloads and SDC-oriented functional test programs [84, 85] share the same Achilles' heel: the extremely high cost often leading to the unfeasibility of fault simulation campaigns during their development, as Figure 4.22 shows, and consequently, the lack of a quality metric for such class of functional test programs. Introducing high-level metrics, or faster analysis, compared to the fault simulations (in Figure 4.23) could improve the development flow and reduce the development time and the number of overall fault simulation campaigns.

The following subsections present simulation-based assessment techniques and an indirect methodology based on the execution trace analysis.

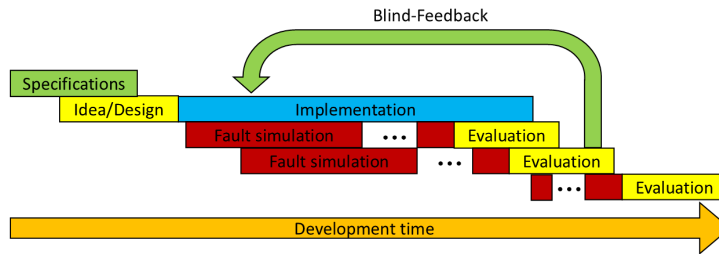



Fig. 4.22: Original development flow for SLT workloads.

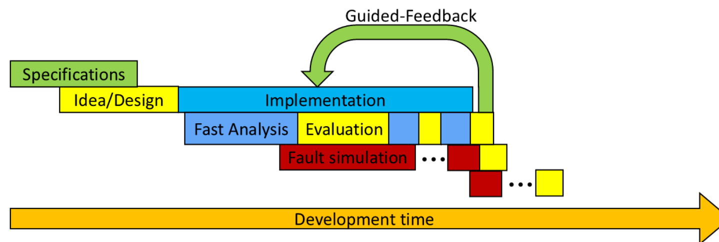



Fig. 4.23: Optimized development flow for SLT workloads.

## 4.4.1 Simulation-based

### 4.4.1.1 Logic Simulation: Toggle coverage analysis

The aim of the proposed toolchain, represented in Figure 4.24, is to abate the computing cost for handling the increased complexity of automotive devices by exploiting thread-based parallel techniques instead of process-based parallel techniques and to overcome the limitation of current EDA tools in terms of stress metrics analyzed per pattern (see Appendix A for implementation details on the tools and algorithms used).

Moreover, it provides test engineers with a visualization of the stress over the layout of the SoC, as well as capabilities to troubleshoot coverage loss by providing fine and coarse stress coverage per gate or module.

As described in Figure 4.24, the toolchain analyzes a VCD file produced by a commercial logic simulator from a functional test program or a structural pattern. In general, test patterns can be developed on the existing silicon version of the device to reduce the prototyping time [86]. Therefore, after the VCD file generation, the toolchain performs the following steps:

- Tool B: VCD analysis and computation of stress metrics.

- Tool C: Layout-aware elaboration for connecting the VCD analysis to the physical device.
- Tool D: Superimposition of the stress results for different stress patterns to provide incremental coverage.
- Tool E: Hierarchical netlist analysis for computing per-module/gate stress metrics.
- Tool F: Visualization of the stress with a heatmap created from the physical layout of the device.

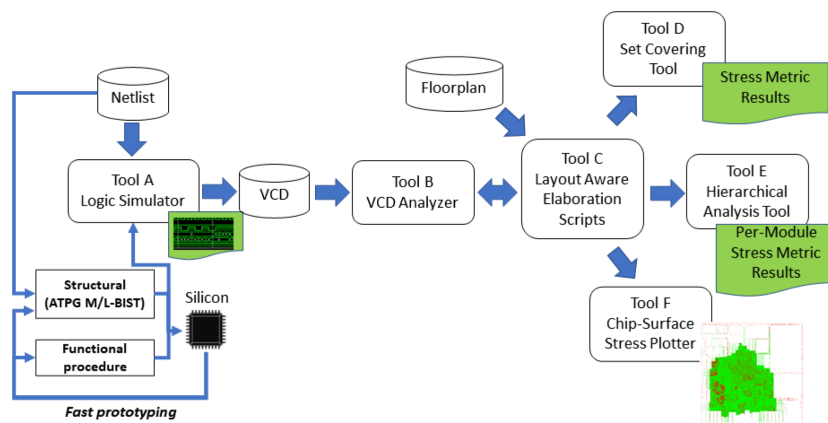



Fig. 4.24: The high-level view of the proposed toolchain for BI stress evaluation.

The stress pattern, either structural or functional, is validated on the silicon implementation of the SoC. Afterward, a logic simulation based on the given stress pattern is performed to provide a VCD file [87]. This file is then analyzed to provide a single or multi-point stress coverage for the stress pattern. The Layout-Aware elaboration links the stress pattern to the SoC layout by weighting the stress metrics with the gate density. There are several options for the final step, from plotting the stress over a layout heatmap to superimposing different stress patterns or presenting stress coverage metrics for each module.

Polian et al. [18] state that grading SLT workloads through fault simulations is becoming increasingly prohibitive due to the rising SoC complexity. Therefore, the toggle activity/coverage could be a clever trade-off between accuracy and computation time for grading SLT workloads. However, SLT-oriented grading is a by-product of the toolchain, mainly conceived for BI purposes.

#### 4.4.1.2 Fault Simulation

Developing System-Level Test (SLT) workloads is inherently complex and requires substantial manual effort and expertise. A common practice among developers is to start from an existing application, such as a Real-Time Operating System (RTOS). This approach can provide a solid foundation, as RTOS frameworks are designed to manage time-critical tasks efficiently. However, this method is not without its challenges.

One of the most significant issues is the long execution time associated with SLT workloads. This extended duration often translates into an abnormal and nearly impossible fault simulation time for grading. In practical terms, this means that the time required to simulate faults in the system can exceed reasonable limits, making it difficult to assess the reliability and robustness of the application effectively. Moreover, the question of design partitioning arises. Developers must consider how to divide the design into manageable segments effectively. Utilizing a residual fault list can be helpful, i.e., leftover undetected faults from structural tests. However, creating and maintaining this list requires a deep understanding of the system architecture and the types of faults that may occur (untestable and functionally untestable [78]), which can be a daunting task for test engineers.

Those limitations necessitate exploring alternative methods that can provide meaningful insights without the exhaustive resource demands of full netlist simulations.

In order to address these challenges, there is a growing interest in the automatic generation of SLT workloads. By leveraging commonly used device descriptions, developers can create SLT workloads that are efficient and tailored to the specific requirements of the DUT. This automation can significantly reduce the manual effort involved in the development process. Furthermore, exploiting the regularity found in applications can lead to the parallelization of fault simulations, as represented in Figure 4.25. By recognizing patterns and commonalities within the application, developers can design concurrent simulations, reducing the overall grading problem and merging the results.

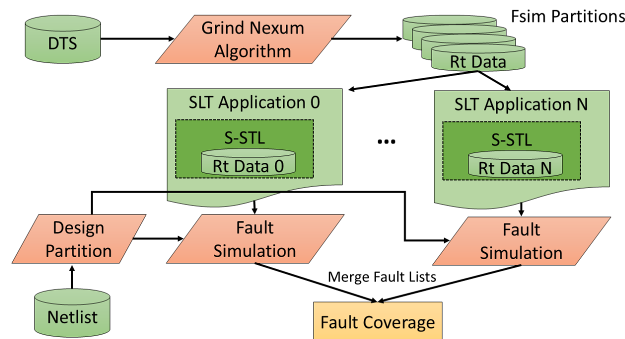



Fig. 4.25: Data partitioning for SLT workloads before the fault simulation campaigns.

### 4.4.2 Indirect methodologies: Execution trace analysis

A fault simulation campaign reports only the fault coverage obtained without indicating the flaws that affected the functional test program. The absence of analysis tools fuels the disappointment in identifying functional test program weaknesses that may prevent reaching a very high fault coverage. The primary objective of this work is not to replace fault simulation campaigns. On the contrary, the target is to prove that the proposed methodology significantly accelerates the development process for functional test programs in the early development stages. The core idea is to provide propagation information at the assembly instructions level, especially for SBSTs. By leveraging this methodology, the efficiency of generating high-quality test programs can be significantly enhanced, reducing the time and effort required by test engineers.

An indirect methodology to quickly analyze a functional test program's quality regarding fault detection capabilities without requiring fault simulation campaigns in early development stages is proposed (see Appendix B for implementation details of basic algorithms).

The methodology loads the instruction trace of a functional test program directly produced by the chip, controlled by a hardware debugger or an Instruction Set Architecture (ISA) simulator. The instruction trace is converted into a Control and Data Flow graph (CDFG) representation. Then, the data and control flows between registers and memory locations are analyzed. The graph representation allows the definition of high-level graph-based metrics, which represent whether the functional

test programs effectively carry their computed value to a diagnostic point, i.e., a software signature in memory or a hardware diagnostic register.

The proposed metrics are approximations of lower-level metrics, such as the one representing toggle and fault coverage. Thus, they indicate the quality of functional test programs in propagating the computed values, exciting the registers' bits, and diversifying data patterns.

Low connectivity of the CDFG indicates that some flaws may affect the program (e.g., previously computed results are overwritten or memory locations are not included in the signatures), potentially reducing its ability to detect faulty behaviors. In order to provide fine-grain feedback to test engineers, the graph analysis is combined with the executable file and the source code to locate code lines affecting the computed metrics.

The proposed indirect methodology is based on the architectural observations reported in the following example.

**Example 1** *Consider the assembly code segment reported in Table 4.3. The table includes three instructions for the RiscV instruction set architecture and illustrates their operation.*

Table 4.3: An example of *RiscV* code segment with the operation performed.

| Cycle | Instruction | Operation |
|---|---|---|
| 1 | addi x19, x18, 192 | x19 = x18 + 192 |
| 2 | sub x6,x6 x18 | x6 = x6 - x18 |
| 3 | addi x6, x19, 35 | x6 = x19 + 35 |

*The data written in x6 by instruction number two is immediately overwritten by instruction number three. This behavior is a* Write-After-Write (WAW) *hazard in computer architectures. Even if there are cases where a WAW behavior may have a specific meaning, they somehow correspond to undesirable situations with information flow disruptions. In principle, these situations should never appear in ASM or compiled C/C++ functional programs and should be prevented by the compiler.*

*Vice-versa, the reading operation of instruction three preserves the write operation on register x19 performed by instruction one; this behavior is named* Read-After-Write (RAW) *hazard. RAW behaviors indicate standard operation flows and are beneficial.*

*Overall, instruction one positively affects the fault coverage if the destination registers are propagated to a software signature in memory during fault simulation, i.e., faults in the*

*ALU potentially propagate to the signature. On the contrary, instruction two is not beneficial to fault coverage as the value of x6 is overwritten and cannot be propagated. Consequently, instruction two can be removed, or the program needs to be modified to unlock a possible propagation of x6.*

An analytical evaluation of the number of potential problems, such as the one shown in the previous example, could indicate potential flaws in the functional test program from a fault detection capabilities perspective.

From the computer security domain, “dynamic taint analysis” runs a program and observes which computations are affected by predefined taint sources, such as the user inputs [88–91]. Following the dynamic taint analysis paradigm, the proposed methodology evaluates a functional test program as illustrated in Figure 4.26.

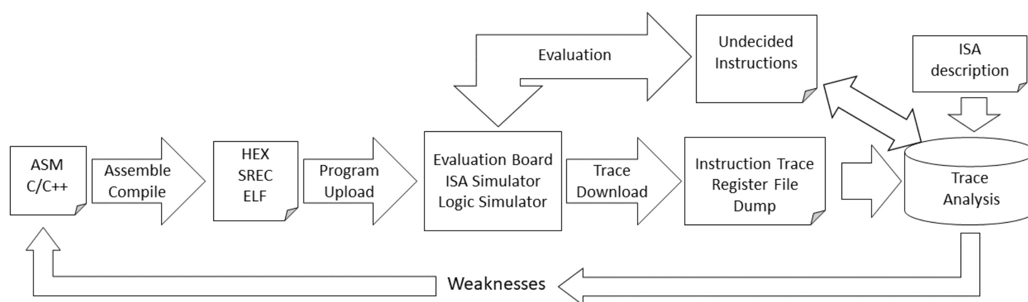



Fig. 4.26: Workflow for the Trace analysis methodology.

As represented by the leftmost block of the chain in Figure 4.26, the input of the methodology is a functional test program written in a high-level language such as C or at low-level assembly code.
Compiled programs can be uploaded on an evaluation board, an ISA simulator (or even a logic simulator), which must provide capabilities of dumping:

- The list of executed instructions with all selected branches and all loops properly unrolled (the instruction trace file).
- All registers that are used the execution of the machine code (the register log for each instruction within the instruction trace file).

The instruction trace file is parsed and transformed into a Control and Data Flow Graph (CDFG) with the proposed trace analysis methodology (represented in the

rightmost block of Figure 4.26). In this CDFG, a vertex represents an instruction, and each direct edge defines the information flow between two instructions. The Instruction Set Architecture (ISA) (i.e., the top-right block of Figure 4.26) is used to provide a specific description of registers and instructions, achieving portability among different ISAs. Instructions are categorized by type (e.g., arithmetic, branch, load, store, etc.), with attributes like byte size, destination positions, operand positions, and memory access information. Registers are divided into general-purpose (e.g., from R0 to R31) and special-purpose, with details including type, permissions (read/write), width, aliases, sub-registers, and initial values. This format provides a detailed schema for comprehensively describing a processor's architecture, covering its instruction set and register file. We generate it starting from the ISA documentation. In order to build the CDFG, the focus is on writing and reading operations and their temporal dependency. Thus, given a data value $d$ in the trace (either a register or a memory location), two types of edges are logically introduced:

- *WAW* edges representing two write operations on $d$, with no reading instructions in between.
- *RAW* edges representing a write operation followed directly by a read operation on $d$.

WAW sequences occurring during the instruction flow over a shared addressable location cause a value to be overwritten, most likely leading to a loss of fault coverage. Conversely, RAW sequences propagate values along with the execution flow, and they can reach an observation point, possibly leading to an increase in fault coverage.

Once the CDFG has been built, it is analyzed to extract graph-based metrics based on the *Connectivity* metric proposed in [45]. The connectivity identifies the program's quality at propagating the computed values, exciting the registers' bits, and diversifying the data patterns.

For fine-grain feedback to test engineers, the proposed methodology provides feedback pinpointing the lines in the source code. Moreover, some cases exist where instructions (mainly control flow instructions) cannot be labeled. Consequently, the proposed methodology permits re-evaluating them by re-executing the functional program and corrupting the registers used by the instructions to evaluate if they affect the control flow of the program [92].

This information can be combined to grade the quality of the functional procedures, rectify their problems, and insert proper signature checks to improve their fault detection capabilities.

#### 4.4.2.1 The Connectivity Algorithms

In this section, the classification for black instruction (blocked) or green (not blocked) is presented, depending on whether it propagates or does not propagate the computed results towards the end of the execution.

It proceeds in three steps: It first creates WAW and RAW edges, essentially building the graph; then, it visits the graph a first time, following WAW edges and coloring nodes as red or green; finally, it visits the graph a second time, following RAW edges, and coloring nodes as green or black. The following example illustrates the overall flow of this procedure.

**Example 2** *Table 4.4 reports a short snapshot of a generic RiscV-like instruction trace file from a test routine for the CPU adder unit. The first column indicates the order of execution; the second and third columns report the address and the mnemonic code of the relative instruction; columns SRC and DST specify the sources and destinations of each instruction.*

Table 4.4: Instruction sequence, source, and destination operands.

| Cycle | Address | Instruction | SRC | DST |
|---|---|---|---|---|
| 1 | 0x0000_0002 | c.sub r19, r6 | r19, r6 | r19 |
| 2 | 0x0000_0004 | addi r6, r19, 35 | r19 | r6 |
| 3 | 0x0000_0008 | addi r6, r19, 192 | r19 | r6 |
| 4 | 0x0000_000C | addi r6, r6, r1 | r6, r1 | r6 |
| 5 | 0x0000_000E | c.sub r19, r6 | r19, r6 | r19 |
| 6 | 0x0000_0010 | c.sub r6, r3 | r6, r3 | r6 |
| 7 | 0x0000_0014 | addi r6, r19, 35 | r19 | r6 |

*Once this information are collected, the most straightforward approach runs through the steps illustrated in Figure 4.27:*

1. *In the first step, represented by Figure 4.29a, we build a CDFG in which each instruction is mapped onto a vertex, and the data flow is represented through edges. We use two types of edges: RAW (reported on the left-hand side of the graph) and WAW (reported on the right-hand side).*

2. *During the second stage, Figure 4.27b, we perform a visit of the CDFG following WAW edges. Vertices are colored either red or green.*

3. *During the third and last step, Figure 4.27c, we perform a visit of the CDFG following RAW edges. Red vertices may become green or black.*

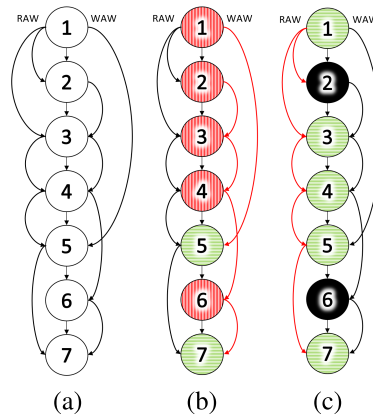



Fig. 4.27: The CDFG of the code snippet of Table B.3 is represented in Figure (a). Figure (b) illustrates the vertex-coloring process obtained following WAW edges. Figure (c) shows the colors obtained at the end of the RAW visit.

The previous process can be optimized by realizing that WAW arcs are unnecessary to obtain the final result. Practically speaking, we perform the WAW visit while building the CDFG and color each instruction in a single backward visit of the CDFG. This process reduces the overall computation costs and improves memory efficiency. Algorithm 13 presents the graph visit, starting from the final instruction and proceeding backward on the edges [93, 94].

In line 1, the algorithm creates an empty set representing the values read along the program. The set must have the property of having only a single instance for each value. In lines 2-7, it stores in this set all observable registers defined by the user and the memory locations in which the system (or the operating system) can verify if the functional test program ends correctly (i.e., all memory location storing the signature or hardware peripherals capable of triggering exception depending on the written value). From line 8, the procedure traverses the graph backward, and in line 9, initializes each node with a black color. When a node has an operand among the destinations read by the observable read set (a memory region or a register) or

**Algorithm 13** A one-step algorithm colors all nodes in a single backward visit of the CDFG of the instruction trace.

**Require:** $G$

```
 1: read_set = {}
 2: for register in ObservableRegs do
 3:     read_set.insert(register)
 4: end for
 5: for address in ObservableMemoryAddresses do
 6:     read_set.insert(address)
 7: end for
 8: for node in G in reverse do
 9:     node_color ← BLACK
10:     for destination in node_dst do
11:         if read_set.contains(destination) then
12:             read_set.remove(destination)
13:             node_color ← GREEN
14:         end if
15:     end for
16:     if node_color = GREEN then
17:         for source in node_src do
18:             read_set.insert(source)
19:         end for
20:     end if
21: end for
```

another operation, it assigns the green color to that node; then, it removes the node from the set of locations that have been read. Since such a register is no longer in the list of read registers, when a new operation tries to write a value in that position, one of the following operations will overwrite it. In lines 17-19, if the information contained in the destination registers is propagated to the observable points, the source registers of the current instruction are added to the list of read values.

The new algorithm does not need to create the graph edges, thanks to the backward visit: at each step, it already knows whether the current information is propagated to the end of the program or it is overwritten, without the necessity to follow the information flow in the forward direction like the original procedure.

For load and store instructions, memory addresses are computed using the actual values of the used registers. Memory addresses are manipulated as virtual registers; thus, they are introduced in the destination for every node in Algorithm 13.

The instruction trace can also include instructions with multiple destinations (either registers or memory addresses) as the new algorithm automatically manipulates a destination list for each node (i.e., instruction) once the ISA correctly describes the instruction (see Appendix B). Data values are propagated if a node has at least one green destination, i.e., the instruction propagates at least some partial result.

To illustrate this process, the following example presents a code snippet with load, store, and arithmetic instructions.

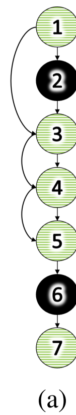
**Example 3** *The code snippet reported in Table 4.5 (and derived from the PowerPC VLE ISA) includes load and store instructions from and to memory locations in addition to arithmetic operations. Figure 4.28 includes the coloring scheme returned by Algorithm 13. For example,*

Table 4.5: Instruction sequence, source, and destination operands.

| **Cycle** | **Instruction** | **SRC** | **DST** |
|---|---|---|---|
| 1 | e_stb r0, 0(r1) | r0, r1 | mem(0+r1) |
| 2 | slli r2, r0, 35 | r0 | r2 |
| 3 | e_lbz r2, 0(r1) | mem(0+r1), r1 | r2 |
| 4 | e_add2i r2, r1, r2 | r1, r2 | r2 |
| 5 | e_stb r2, 0(r1) | r2, r1 | mem(0+r1) |
| 6 | subfze r1, r3 | r1, r3 | r1 |
| 7 | e_add16i r1, r0, 35 | r0 | r1 |

*the first instruction stores a value in the location pointed by register r1. This location is referenced as a source operand by the third instruction. Therefore, instruction one is finally labeled as green (light-gray). Moreover, instruction five is marked as green because the value written in memory is propagated till the last instruction.*

During the process represented by Algorithm 13, conditional branches are left undecided and temporarily colored in orange. After that, the CDFG is visited to find the alternative branch among the instructions following the conditional statement. If the branch's alternative address cannot be found, the branch is colored in black. If the branch alternative can be reached, the visit checks whether the incorrect branch execution leads to a different data flow and control flow and colors the branch in black or green. The following two examples illustrate this procedure.




(a)

Fig. 4.28: An example in which our algorithm is applied to a code section includes arithmetic operations and load/store instructions.

Table 4.6: Instruction sequence, source, and destination operands.

| Cycle | Address | Instruction | SRC | DST |
|---|---|---|---|---|
| 1 | 0x0000_0000 | e_li r0, 0 | | r0 |
| 2 | 0x0000_0004 | cmpl r0, 1 | r0 | cr |
| 3 | 0x0000_0008 | e_beq jump | cr | |
| 4 | 0x0000_000C | cmpl r0, 0 | r0 | cr |
| 5 | 0x0000_0010 | e_add16i r1, r2, r3 | r2, r3 | r1 |
| 6 | 0x0000_0014 | jump: e_add16i r1, r0, r1 | r0, r1 | r1 |

**Example 4** *The code reported in Table 4.6 (PowerPC VLE ISA) includes a branch (line 3) generated by a simple if-then-else construct (line 2).*

*Figure 4.29 reports the CDFG of this code snippet. RAW edges, not explicitly computed by Algorithm 13, are reported to highlight the information flow. Branch nodes are initially left as undecided. Then, a second graph visit evaluates the instruction flow under the hypothesis of taking the wrong branch path. The result is represented in Figure 4.29c. In this case, the instructions that would be reached if the branch decision was wrong are included in the instructions that follow the branch itself. A new orange-colored edge is added to the figure to highlight this situation, and it appears that some instructions would not be executed if the branch was wrongly executed (taken if it was not taken, and vice-versa).*

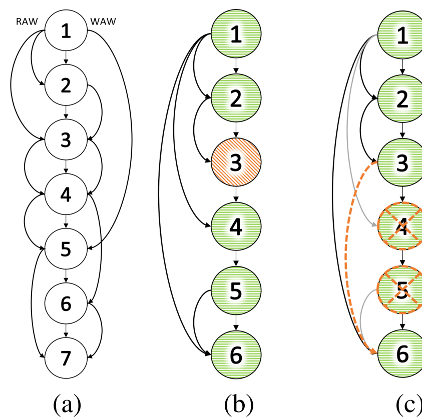



Fig. 4.29: The graph example includes a branch instruction.

*The instructions that may be skipped are green. Therefore, the value of the final registers may change, compromising the execution. Indeed, the branch node is labeled green. Conversely, if all skipped instructions had been labeled black nodes, they would not have contributed to the data flow. The branch would be black.*

**Example 5** *Let us focus on the code reported in Table 4.7 (PowerPC VLE ISA). Following Example 4, the branch is initially left undecided. Then, if the instruction flow does not change under the hypothesis of taking the wrong branch path, it is determined that all skipped instructions are labeled in black. As a consequence, the branch is colored in black. The outcomes of the two visits are illustrated in Figure 4.30c.*

This approach colors branches (in if-the-else statements) only when both paths can be found in the instruction trace. Unfortunately, there are cases in which one of

Table 4.7: Instruction sequence, source, and destination operands.

| Cycle | Address | Instruction | SRC | DST |
|---|---|---|---|---|
| 1 | 0x0000_0000 | e_li r0, 0 | | r0 |
| 2 | 0x0000_0004 | cmpl r0, 1 | r0 | cr |
| 3 | 0x0000_0008 | e_beq jump | cr | |
| 4 | 0x0000_000C | e_add16i r1, r2, 1 | r2 | r1 |
| 5 | 0x0000_0010 | e_add16i r1, r0, 1 | r0 | r1 |
| 6 | 0x0000_0014 | jump: e_add16i r1, r0, r2 | r0, r2 | r1 |

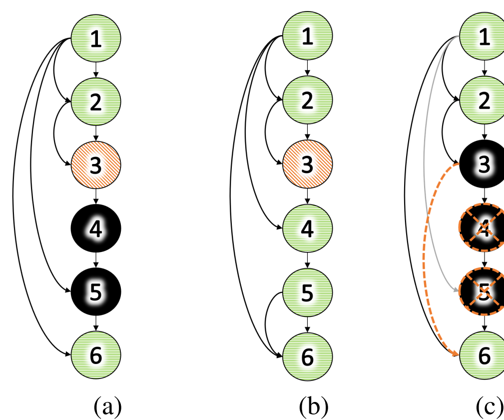



Fig. 4.30: Another branch instructions example.

the paths is not present in the instruction trace. One possible solution to this problem is to adopt a guided fault injector, performing fault injections in undecided branches to force a change in the execution trace [92].

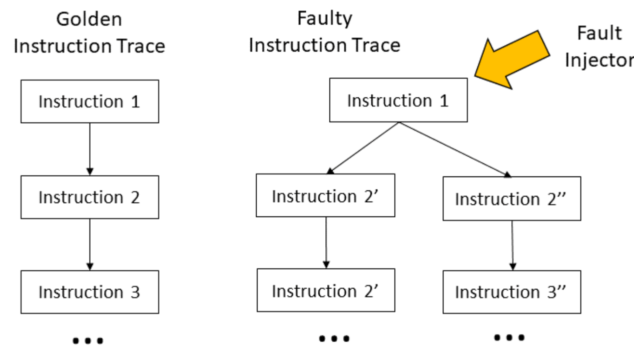



Fig. 4.31: Fault injection in branch-control registers for undecided branches to improve the representativity of the execution trace,

Figure 4.31 graphically represents the action performed by the fault injector. The golden instruction trace is considered on the left-hand side, identifying the undecided branches (in instruction 1). Using a fine-grained fault injector, the condition where the branch is taken is corrupted, forcing the execution of the other branch. By exploring both branches, the whole execution space can be effectively visited and precise information about undecided branches of the original instruction trace can be gathered, i.e., if it changes the CDFG.

#### 4.4.2.2 Graph-based Metrics: The Connectivity Metrics

The CDFG built by the coloring algorithms permits the extraction of graph-based measurements. The idea is to use these measures as indirect quality indicators of the functional test program.

The quality is strictly related to the fault detection capabilities of the code and helps test engineers boost the program development process. We define three different metrics to indicate the program's quality from different aspects.

The *Connectivity* (C) metric, as proposed in [45], for a generic program can be defined as follows:

$$C = \frac{\sum_{i}^{GreenInstr} \left( \frac{1}{Instr_i Dests} \sum_{j}^{Instr_i Dests} GreenDests \right)}{TotInstructions} \tag{4.16}$$

The formula computes the percentage of instructions that propagate data values toward the end of the functional test program (i.e., possibly to an observable point, such as a software signature in memory or a diagnostic register). It proceeds in two steps. As instructions may have multiple destinations in the computation between parenthesis, a connectivity value is assigned to every CDFG node. After that, it evaluates the sum of connectivity values of all green instructions (weighted by multiple destinations), and it divides this contribution by the total number of instructions.

The previous metric does not consider the number of bits that toggle in the destination. For example, suppose a simple *add* instruction on a 32-bit register is executed between R1 and R0, using as destination R2, with R1 equal to 1 and R0 equal to 0. In that case, only one bit is changed in the destination register. Different faults can be excited and observed as different data patterns may activate different circuit logic cones. Therefore, the previous metric can be improved by computing the number of bit flips. As a consequence, we defined a new metric called *Register Bit-Level Connectivity* ($C_{RBL}$). Each instruction propagating its data value (i.e., the basic connectivity metric) represents the number of bits that toggle (i.e., change from 0 to 1 or vice versa). The Register Bit-Level Connectivity can be computed as illustrated by the following equation:

$$C_{RBL} = \frac{1}{N_{regs}} \sum_{i=0}^{N_{regs}} \frac{\sum_{j=0}^{R_{i_{bits}}} \begin{cases} 0 & \text{if } R_{t_{i,j}} = 0 \\ 0.5 & \text{if } R_{t_{i,j}} = 1 \\ 1 & \text{if } R_{t_{i,j}} > 1 \end{cases}}{R_{i_{bits}}} \tag{4.17}$$

Where $R_{t_{i,j}}$ represents the number of total transitions during the whole execution of bit $j$ of the $i^{th}$ register and $R_{i_{bits}}$ represents the total bits of register $i^{th}$. In this way, the formula provides information about registers that undergo two transitions (from 0 to 1 and vice versa), independently from the length of the test program. Notice that we consider each instruction a black box, as only its input and output

states are known. This choice allows the generalization of the metric and relies on the correctness of the instruction trace and the register file. A low value of register bit-level connectivity means that some transitions are not exercised in the registers, and bad test data patterns for capturing faulty behavior are used.

To combine the number of toggles per bit (dependent on the length of the test program) that propagate their value, we introduce the *Register Average Bit-Toggle Connectivity* ($C_{RBT}$):

$$C_{RBT} = \frac{\sum_{i=0}^{N_{regs}} \sum_{j=0}^{R_{i_{bits}}} R_{t_{i,j}}}{\sum_{i=0}^{N_{regs}} R_{i_{bits}}} \tag{4.18}$$

Where $R_{t_{i,j}}$ and $R_{i_{bits}}$ have the same meaning introduced for Equation 4.17. The formula computes the arithmetic mean of the number of toggles among the total register bits. Notice that adding small values or values with many zeroes, using division instructions with a large dividend, and multiplying large numbers together produce uniform outputs, i.e., similar to one of the input operands. These values can produce significantly high $C_{RBT}$ values even when most bits remain unchanged.

The average mean computed in the previous formula can be easily substituted with a geometric mean to evaluate programs with unbalanced toggling activity more appropriately. Thus, we can define an additional metric based on the geometric mean of the toggle values of test programs. We call it *Register Bit-Toggle Tendency Connectivity* ($C_{RT}$), as the following equation shows:

$$C_{RT} = \exp\left(\frac{1}{\sum_{i}^{N_{Regs}} R_{i_{bits}}} \sum_{i}^{N_{Regs}} \sum_{j=1}^{n} \ln R_{t_{i,j}}\right) \tag{4.19}$$

Where $R_{t_{i,j}}$ and $R_{i_{bits}}$ have the same meaning introduced for Equation 4.17. The *Register Bit-Toggle Tendency Connectivity* measures the number of average bit transitions for instructions that propagated their computed values. It is equal to zero when there are no transitions on some bits of registers.

To summarize, the presented metrics allow us to grade the quality of a functional test program in the following directions:

- The *Connectivity metric* represents the flow of data values among the instructions [45].

- The *Register Bit-Level Connectivity* evaluates the flow of bit transitions among registers.
- The *Register Average Bit-Toggle Connectivity* computes the arithmetic average of bit transition among registers.
- The *Register Bit-Toggle Tendency Connectivity* measures the general tendency of bit transitions for data patterns.

#### 4.4.2.3 Weaknesses identification in the source code

For increasing productivity during the development of functional test programs, it is crucial to highlight potential flaws or weaknesses in the source code to prevent fault masking by WAW edges, a scenario that can easily go unnoticed but has significant implications regarding fault coverage, Figure 4.32 shows the weaknesses identification flow.

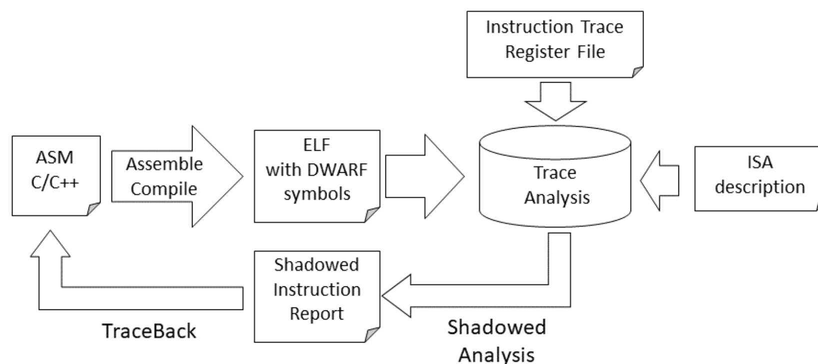



Fig. 4.32: Weaknesses identification flow.

The source code is usually written in assembly or a high-level programming language (such as C or C++) in multiple files linked together by the compiler in the executable. The proposed methodology analyzes the instruction traces and the register files augmented with information from the executable file (in ELF format) and the debugging symbols (in DWARF format [95]) produced by the compiler.

The methodology identifies black instructions that fail to propagate their computed data and produces a report highlighting them. A traceback step allows linking the identified weaknesses to the source code. The decision on rectifying the problem

is left to the test engineer, who can either remove the instructions or add a partial signature computation. Moreover, a black instruction can potentially mask (or shadow) the results of other instructions, further reducing the proposed metrics and impacting the fault coverage. The following subsections briefly describe the traceback and shadowed instruction analysis.

##### 4.4.2.3.1 Traceback From Assembly to Source Code

The reconstruction of the source code from assembly instruction traces is a very well-studied topic in reverse engineering literature [96, 97]. Therefore, the proposed methodology does not provide any innovative methods. However, it relies on providing feedback by tracing the identified weaknesses in the source code, adopting the commonly used dwarf library (libdwarf). This library can access the DWARF debugging information and symbols in executable and object files. The compiler must produce the executable file (in ELF format) with the DWARF debugging information and symbols to allow the proposed methodology to trace the identified weaknesses to the source code. The debugging information contains all the data types, sections, call frame information, and the line numbers in the source code [95]. We use the table storing the line numbers to identify in the source code the weaknesses discovered by the proposed methodology in the assembly code.

##### 4.4.2.3.2 Shadowed Instruction Analysis

The shadowed Instruction analysis identifies the black node (i.e., instructions) overwriting data destinations and disrupting the data flow of other instructions. It parses the CDFG in reverse order to compute how many instructions a given instruction blocks by reconstructing the disrupted data flow from the sink node (the instruction where the dataflow is disrupted) up to the beginning of the CDFG.

Thanks to this analysis, we can find instructions that overshadow many others, producing a cascade effect and turning multiple dependent instructions green with a single change to the code.

#### 4.4.2.4 Multi processor connectivity computation

When an application runs on multiple master cores, such as CPUs, DSPs, DMAs, or AI HW Accelerators, each one manipulates its registers and shared memory locations, which can also be used to communicate with the other cores. This means that the destinations written by an instruction on a core can be read by subsequent instructions executed by the same core or instructions running on other cores.

The two code snippets reported in Figure 4.33 illustrate the difference between the single-core and the multicore algorithms. Destinations are marked in purple and sources in blue for ease of reading. Squared nodes represent load and store instructions, whereas register-based instructions are circle-shaped nodes. Instruction 5, running on core 0, is a load instruction that writes to the memory location 0xF6CA; this location is later read by instruction 6 of core 1. If the algorithm evaluates each trace separately, the instructions are considered "unconnected" and not contributing to fault coverage, as represented on the CDFG on the left-hand side in yellow.

On the contrary, if the evaluation considers multicore behaviors, the instructions working on memory location 0xF6CA become "connected" and can be green-colored as a value is propagated from one core to another. Since that memory destination becomes green, other instructions for the first core become connected, such as instructions 1 and 3 for core 0.

Conversely, the multicore analysis can correctly determine that instruction 3 of core 1 (marked in red), also written to the same memory address 0xF6CA, gets overwritten by core 0.

The order of execution of instructions that modify shared memory is an important factor. Synchronization points implemented in the SLT multicore-oriented workloads (i.e., semaphores) ensure that a core correctly reads information written by another core. Accordingly, information are used to evaluate the correct placement of synchronization points, bringing three pieces of information to the attention of the test engineer:

- Uninitialized memory read: The presence of reads at memory locations that were never explicitly initialized. The delay of a write operation could cause this to move after the corresponding reading phase.

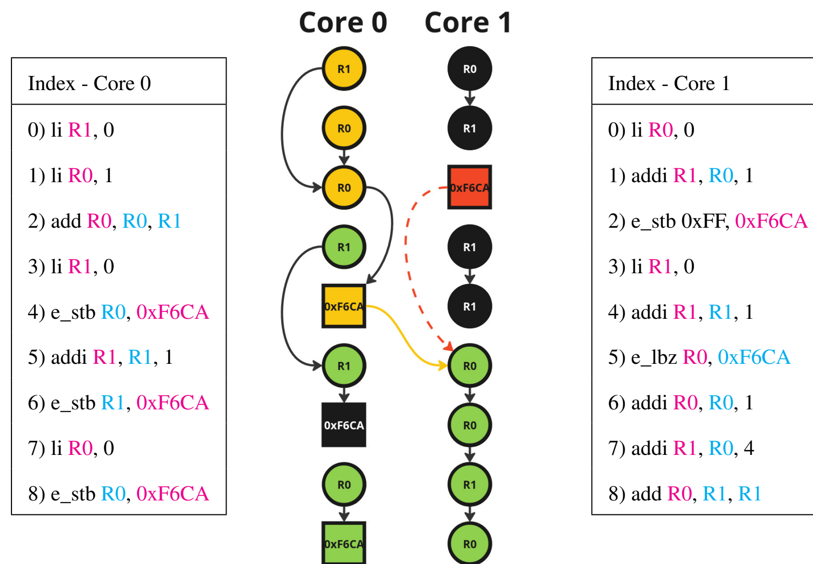



Fig. 4.33: Single and multicore connectivity.

- Missing synchronization between read and write operations: A reading instruction of a data value written by another operation but without synchronization. It represents a read operation delayed long enough to swap its order with a write operation.
- Writes without subsequent reads: The presence of memory writes without an explicit read. This could be caused by a situation similar to the first example.

These three cases represent situations where the code is most likely incorrectly synchronized. This could lead to out-of-order executions of the instructions involved and catastrophically impact the SLT testing effectiveness.

In addition, it is crucial to mention that timing variations in different cores may result in slightly different output traces. Nevertheless, the assumption is that the multicore SLT workloads implement synchronization mechanisms that avoid hazards, such as concurrent execution of critical sections generating indeterminism in the application.

# Chapter 5

# Experimental results

*Extraordinary claims require extraordinary evidence.*

Carl Sagan

## 5.1 Experimental setup

As mentioned in the introduction, this PhD dissertation focuses on a 40 nm Automotive SoC manufactured by STMicroelectronics as a case study. The SoC is compliant with the ISO 26262 ASIL-D standard and features approximately 20 million logic gates and around 700,000 flip-flops. It has multiple LBIST partitions (or islands) and Scan Chain Domains.

The multicore architecture consists of three 32-bit cores that utilize the PowerPC Variable-Length Encoding (VLE) instruction set. Additionally, it includes 6 MB of Flash memory and 128 KB of general-purpose SRAM, along with several peripheral bridges to access a variety of on-chip and off-chip peripherals. For communication between on-chip components, the SoC is interconnected via two fast crossbar switches, AHB-AMBA [30] version 2.0, operating at 64 bits and functioning up to 200 MHz. These two crossbars are linked by a cross-lake, allowing for the connection of one master from one crossbar to two slaves from the other and vice versa. The SoC also supports various communication peripherals, including CAN, LinFlex, SPI, and FlexRay.

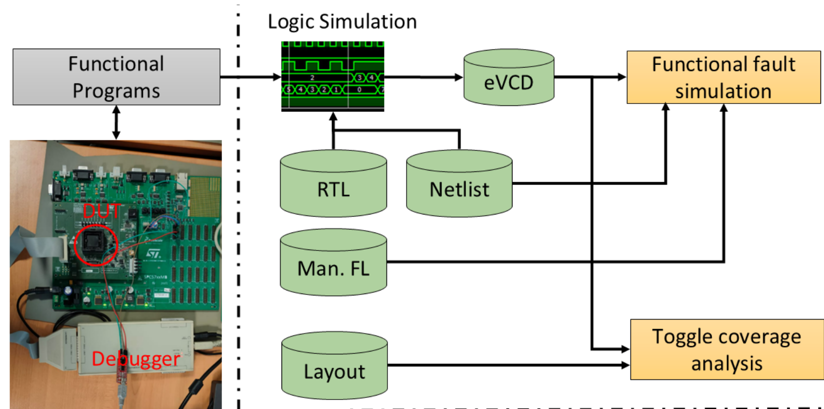



Fig. 5.1: Experimental setup.

In order to develop effective SLT workloads and test strategies, information from the RTL, gate-level netlist, and physical layout of the manufactured automotive SoCs are combined, as the experimental setup shows in Figure 5.1. From manufacturing testing, structural tests (including BISTs and scan-based tests) achieve the coverage levels and corresponding number of patterns depicted in Table 5.1 [20, 31], they are combined with functional fault simulations results ("Man. FL" in Figure 5.1).

| Fault Model | # Faults | Coverage [%] | # Patterns | # Test Escapes |
|---|---|---|---|---|
| Toggle | ca. 20M | 95.89 | 1k | 822,000 |
| Stuck-at | ca. 40M | 99.21 | 69k | 316k |
| Transition Delay | ca. 40M | 89.65 | 82k | 4M |

Table 5.1: Structural tests coverages, number of patterns, and test escapes.

In Figure 5.1, the Logic Simulation is done with the *Incisive Suite*, from *Cadence*. The tool for functional fault simulation is *Z01X*, from *Synopsys*. The toggle analysis is achieved with a toolchain developed in the frame of Politecnico di Torino [26].

The debug environment used for developing and validating the functional test programs (including stress programs and SLT workloads) has been done on an evaluation board for the DUT, using a *Trace 32 PowerDebug*, from *LauterBach*; functional test programs have been compiled with a gnu-based toolchain specifically built for PowerPC VLE.

### 5.1.1 FPGA-based Tester

The main goal of this subsection is to introduce experimental setups (used for some experimental results) based on a flexible FPGA-based tester designed to enhance the testing capabilities of Automotive System-on-Chips (SoCs) in research labs [98] (see Appendix C for more details).

The proposed tester combines structural tests (like scan-based patterns and Built-In Self-Tests) and functional tests (which simulate real-world operating conditions) into a single platform. This integration allows for more efficient screening of faulty devices, reduces the need for expensive industrial Automatic Test Equipment (ATE), and provides a modular design that can adapt to various testing needs. The tester is intended to facilitate research and development in testing strategies while being cost-effective and space-efficient for academic and industrial environments.

The experimental setup for obtaining instruction traces extracted from functional test programs with the case study is presented in Figure 5.2.

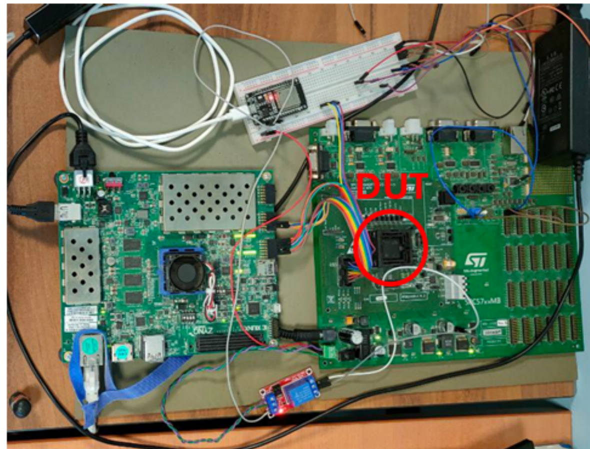



Fig. 5.2: Experimental setup with Xilinx ZCU 104 evaluation board.

The tester has been synthesized for an AMD-Xilinx Zynq Ultrascale+ MPSoC ZCU104 Evaluation Kit [99]. It includes a ZU7EV device with a quad-core ARM Cortex-A53 applications processor, dual-core Cortex-R5 real-time processor, and Mali-400 MP2 graphics processing unit. The FPGA fabric has 504k System Logic Cells and 38Mb of memory. In Figure 5.2, the tester is connected to the TAP Controller of the DUT for debugging purposes.

On top of the hardware architecture in the MPSoC, there is a full Linux-oriented distribution with networking capabilities, there exists a custom software stack resides, which, through the aforementioned instruction to be passed to the hardware module, gives the tester the capabilities to perform the following tasks:

- Access to the on-chip Test Access Port (TAP) controller to extract basic device information.
- Extract information on registers and memories.
- Set on-chip breakpoints.
- Modify registers and memories.
- Inject errors in the memory words and their relative Error Correction Code (ECC).
- Execute Instruction trace download in both single and multicore applications.
- Flash erase, programming, and verify.
- Perform Test-Mode Entry for structural testing for single or multiple scan chains.
- Diagnostics of DUTs through the data retrieved by applying precomputed stimuli through a STIL file [100].

All the above capabilities are device-specific and will be coded for each DUT used. The generic HALs to interface with the debugger and the hardware design are generic and do not require modifications unless required. Using the PYNQ [101] framework (python-based) allows to read, understand, and automatize the operations easily.

Using a System-On-Chip featuring an FPGA, the proposed tester gains an inherent modularity, both on the hardware and software level, allowing the user to create and integrate specialized modules directly on the FPGA fabric. These tailored modules seamlessly integrate with the core design, enabling targeted and comprehensive testing scenarios. This approach offers distinct advantages, including:

- Scalability: Easily add or remove modules, customizing the design for specific tests;

- Flexibility: Effortlessly customize to match requirements;
- Cost-effectiveness: Eliminate the need for expensive dedicated testing hardware, reducing costs.

An application example can be the necessity to test the CAN Module of the DUT functionally. The SLT phase tests the DUT functionally, and peripherals are a fundamental block in modern safety-critical systems. In this case, a generic communication peripheral can be added to the Programmable Logic to communicate with the chip. Moreover, suppose the user needs a protocol error injector to test the answer of the communication peripheral to errors within the frame. In that case, a design can be integrated as part of the tester as Figure 5.3.

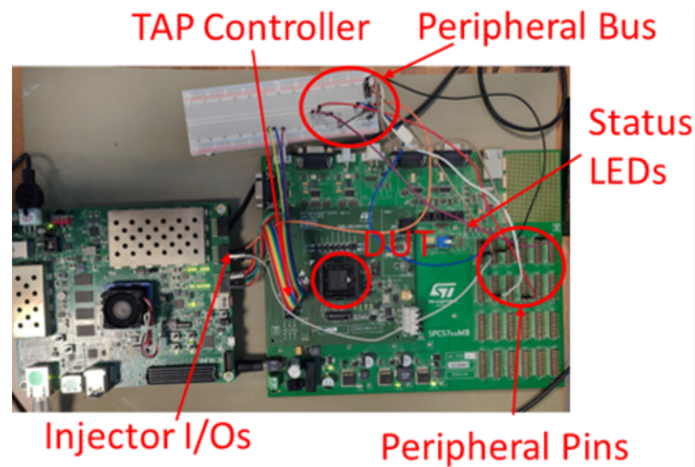



Fig. 5.3: Experimental setup with Xilinx ZCU 104 evaluation board and peripheral bus (CAN).

The proposed FPGA-based tester serves as an enabler for the experimental setup used to obtain the experimental results. While it could be replaced with a commercial debugger without any loss of generality, it was specifically chosen to facilitate the parallel deployment of experimental setups. Additionally, it offers faster execution times by implementing only the basic and necessary functionalities of a commercial hardware debugger, primarily due to the absence of a graphical user interface (GUI). This approach also enables parallel remote connections to the setups in a cost-effective manner, making it a practical and efficient solution for experimental setups in a research lab.

## 5.2 Simulation-based experimental results

### 5.2.1 Toggle coverage analysis

#### 5.2.1.1 Enabling stress for SLT phase

Despite Data and Address busses to embedded memories having a 32-bit width, memory banks use only 16 address bits in the presented case study. The memory isolation collar type inserted in this design is the Chip select-based DfT as in Figure 5.4.

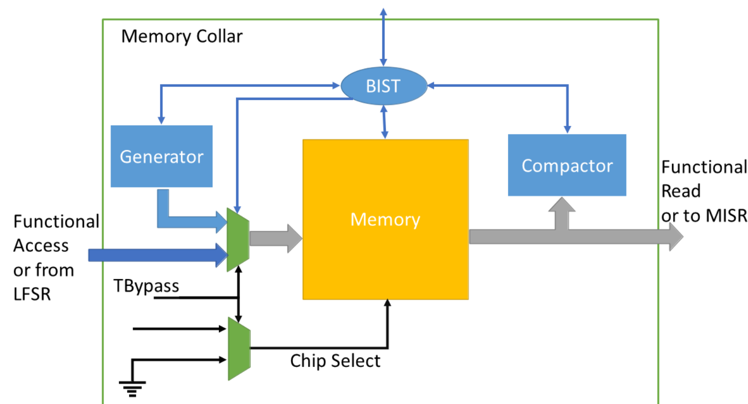



Fig. 5.4: Chip select isolation-based DfT for the memory collar.

Functional logic simulation of the gate-level netlist generates an eVCD file to measure the toggle activity[26]. A Python-based tool generates the evaluated programs, taking as input the *memory overview* and can be customized for different memory access modes, e.g., aligned or not. Other ad-hoc tools are used to analyze the toggle activity from simulation dumps and compare results from the considered stress techniques.

##### 5.2.1.1.1 Toggle Coverage results

A custom tool generates entity-based toggle activity overlapping different approaches, presented in Table 5.2; the three right-most columns show the incremental toggle coverage on a given entity obtained by the overlaps of different approaches. Moreover, the number of entity-related gates around the RAM entity is reported.

| RAM Entity | Size | Gates | Toggle Coverage [%] | | |
|---|---|---|---|---|---|
| | | | LBIST | LBIST + MBIST | LBIST + funct. |
| Core 0 Data | 128KB | 1055 | 75.55% | 96.4% | 96.4% |
| Core 1 Data | 128KB | 1022 | 74.76% | 96.17% | 96.17% |
| Core 2 Data | 128KB | 1126 | 77.09% | 96.63% | 96.63% |
| Core 0 Inst | 32KB | 307 | 94.79% | 98.09% | 98.09% |
| Core 1 Inst | 32KB | 311 | 94.86% | 98.22% | 98.22% |
| Core 2 Inst | 32KB | 311 | 94.86% | 99.07% | 99.07% |
| System | 128KB | 1368 | 97.08% | 97.08% | 97.08% |
| Can | 16KB | 1098 | 96.81% | 97.36% | 97.36% |
| AMU | 32KB | 1663 | 98.74% | 98.89% | 98.89% |

Table 5.2: Incremental toggle coverage per memory collar.

It is important to mention that the stress code is not relocated into instruction RAMs, and the approach treats the instruction RAMs as regular data RAMs. Memory BISTs encapsulate the toggle activity generated by the proposed approach within the Cores' memories. In contrast, other RAMs, even with structural approaches, have already reached a satisfiable toggle coverage,i.e., functional and MBIST approaches stimulate busses similarly from a toggle activity perspective. Therefore, the proposed approach lacks additional stress since the reference Memory Collar architecture is like Figure 4.8. The missing toggle coverage in memories is given by the limitation of stimulating redundancy lines for the memory array by functional means.

#### 5.2.1.1.2 Toggle activity on Memory Busses

Deepening the analysis of the memory busses, it becomes evident that even if Memory BISTs are powerful in toggle coverage, they lack uniform stress distribution, especially on the address bus. As it can be seen from Figure 5.5, the toggle activity of a single March Element of a generic March algorithm implemented by a Memory BIST (in grey) is not uniformly distributed across the bit lines of the address bus. Figure 5.5 results are presented for an address bus of 8 bits; however, it can be easily extended to any address width using the following formula:

$$T\{A[i]\} = \frac{2^{ABus-1} * \#ME}{i+1} \tag{5.1}$$

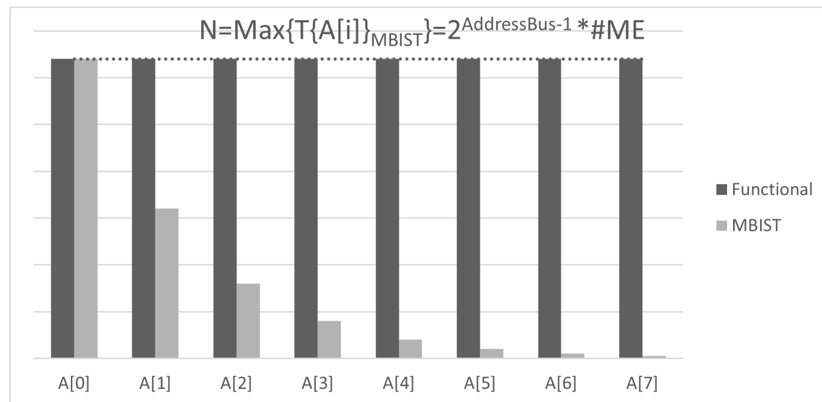



Fig. 5.5: Bit-by-bit comparison between the toggle activity achieved by the functional and BIST approaches.

The proposed approach, in black in Figure 5.5, achieves constant stress over all the bit lines of the address bus, forcing each line to toggle the same number of times N (corresponding to the maximum number of transitions forced by the Memory BIST on the least significant bit).

#### 5.2.1.1.3 Analytical Evaluation

All the previous considerations can be evaluated using analytical formulas. For the March Algorithm, it has been considered the simple and well-known *March C-algorithm*.

| Approach | $TotTrans_{ABus}$ | $TotTrans_{DBus}$ | Clock Cycles |
|---|---|---|---|
| MBIST | 459K | 8.39M | 720K |
| Functional | 5.04M | 322M | 229K |

Table 5.3: Analytical evaluation by approach.

Although it is evident that the number of times a generic bit line within the Data or the Address bus toggles is very high in the Memory BIST approach compared with both functional approaches, the substantial difference is in the execution time. Therefore, a crucial aspect is the normalization concerning the execution time of MBIST, using from (1) to (6).

- Memory BIST:

  - $AvgTrans_{DBus} = \frac{TotTran_{DBus}}{DBus} = 262K$
  - $AvgTrans_{ABus} = \frac{TotTran_{ABus}}{ABus} = 28K$

- Functional approach in Figure 4.11:
  - $AvgTrans_{DBus} = \frac{TotTran_{DBus}}{DBus*ET_{MBIST}} = 31M$
  - $AvgTrans_{ABus} = \frac{TotTran_{ABus}}{ABus*ET_{MBIST}} = 991K$

On the one hand, memory BIST has been conceived for spotting defects within the memory array, not for stimulating circuitry for the BI, even if it still provides acceptable toggle coverage. On the other hand, functional procedures are quite effective because they have been purposely built to maximize the toggling activity over a minimum time. This explains why the average number of transitions for functional approaches normalized to the execution time of Memory BIST is far more significant. On the other hand, the analytical formulas for speedups can be calculated in the following way:

$$SpeedUp = \frac{ExecTime_{MBIST}}{ExecTime_{Func}} = \frac{2 * \sum_{i=0}^{\#ME} MA_i}{\#ME} \quad (5.2)$$

The formula has been calculated using a toggle parameter for functional approaches, the maximum value of the toggle activity generated on the address bus by Memory BIST. Although the speedup is indirectly proportional to the number of March Elements, it becomes evident that the sum of the memory accesses for each March Element, for both functional approaches, is a more critical factor in the formula. In this case, considering *March C-algorithm*, the SpeedUp is 3.14X.

#### 5.2.1.2 Final remarks

The BI phase may not exacerbate the interconnections related to/from memories enough if only logic and memory BISTs are used. The proposed approach for enabling stress for the SLT phase looks at this gap and intends to create complementary functional programs to add to traditional structural tests.

The identification of the non-uniform and unstressed zone was carried out using the toggle coverage analysis toolchain, which pinpointed non-uniform stress in the memory collars.

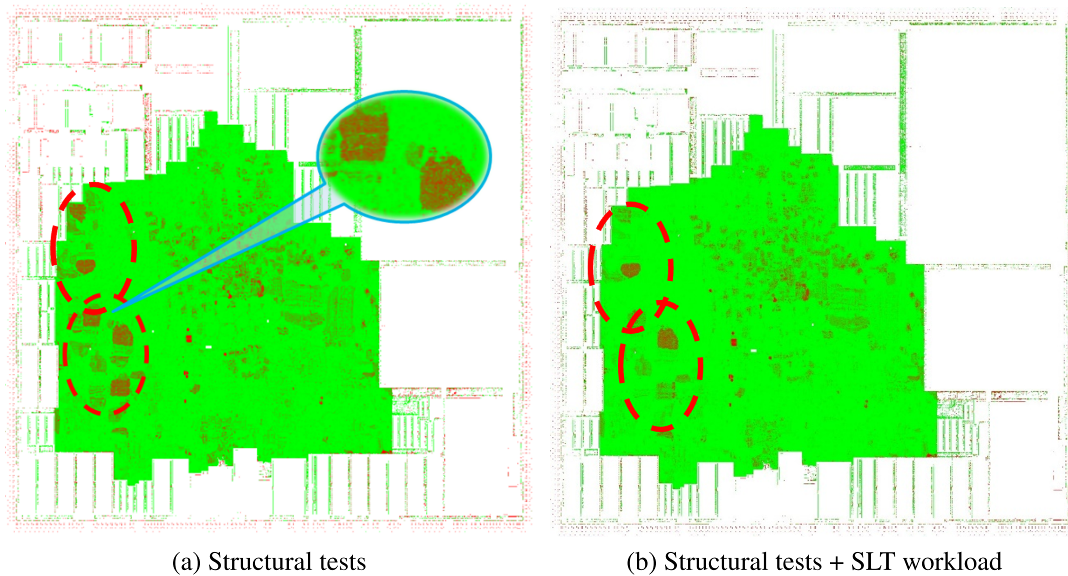
From an absolute perspective, results show that the proposed functional procedures can more uniformly distribute the stress over the memory signals in less execution time than Structural tests (including Memory BIST) by about 118 times higher for the Data bus and 35 for the Address Bus. On the other hand, in MBISTs, address generation can be achieved by different methods, increasing the toggle distribution. However, this solution has the disadvantage of increasing the silicon area of BISTs and their complexity and reduced flexibility compared to functional methods.

The toggle coverage analysis toolchain can be exploited to visualize the stress over the layout and further improve the stress. The BI for safety-critical devices includes different patterns, ranging from structural to functional. For example, Figure 5.7 contains output images representing the stress coverage from different stress patterns. In addition, Figure 5.7 highlights unstressed and stressed regions for different patterns. Figure 5.7a depicts a non-stressed region better strained by the pattern in Figure 5.7b. Similarly, the pattern in Figure 5.7d focuses on a module not well stressed by the pattern represented in Figure 5.7c. On the other hand, the same pattern does not activate enough memory ports and some functional units, i.e., the upper and lower left zooms in Figure 5.7d. Finally, Figure 5.7e and Figure 5.7f show how to adequately stress memory ports and functional units, respectively.

Therefore, single patterns are superimposed and analyzed to understand their weaknesses and abilities. This activity allows the generation of the stress-colored heatmap plot for the overall BI phase displayed in Figure 5.6a. From these images, it is straightforward to distinguish the zones where the stress level is adequate from those needing additional patterns (such as the area inside the dotted red circle).

As pinpointed from Figure 5.6a and confirmed from the hierarchical analysis tool (Tool E in Appendix A), an unstressed zone exists that does not reach an acceptable level of stress coverage. Therefore, we develop an additional functional stress pattern that can provide additional stress as represented in Figure 5.6b.

As shown from Figure 5.6b, the ad-hoc developed functional pattern stresses the region of interest, effectively increasing the stress coverage.



(a) Structural tests (b) Structural tests + SLT workload

Fig. 5.6: Visualization of the overall stress provided by the superimposition of all stress patterns.

### 5.2.2 Fault Simulation

#### 5.2.2.1 Communication peripherals

In the proposed case study, there is a cluster of four CAN modules. Figure 5.8 presents a layout heatmap (extracted from the actual physical implementation) for the CAN peripheral in the DUT, where the 4 different controllers can be observed. Figure 5.8a shows a color-driven layout map of the CAN peripheral logic depending on the critical regions highlighted by the structural test weaknesses analysis, as previously detailed (see the legend in the figure). This map has at least two very localized hot spots located in the error detection/correction logic and in the Transmission/Reception interface to the chip top in the southern part of the heat map. This experimental evidence confirms the hypothesis about structural test weaknesses for communication peripherals. As a preview of the final results fully reported later, Figure 5.8b provides the layout heatmap of the CAN peripheral, the untested SAF faults in red, and in green the tested ones by all the structural tests. Meanwhile, Figure 5.8c presents the effect of applying the proposed SLT suite complementing

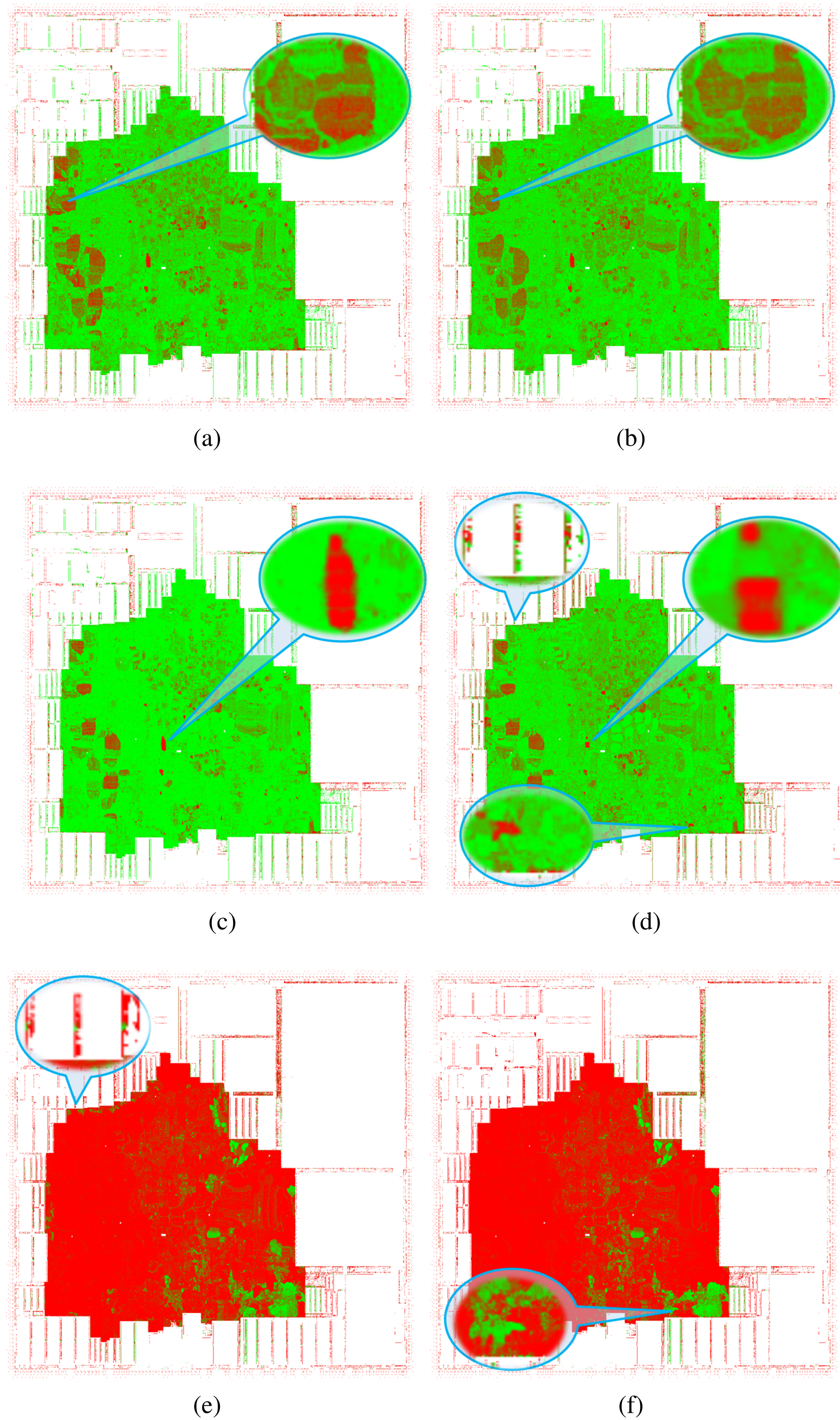
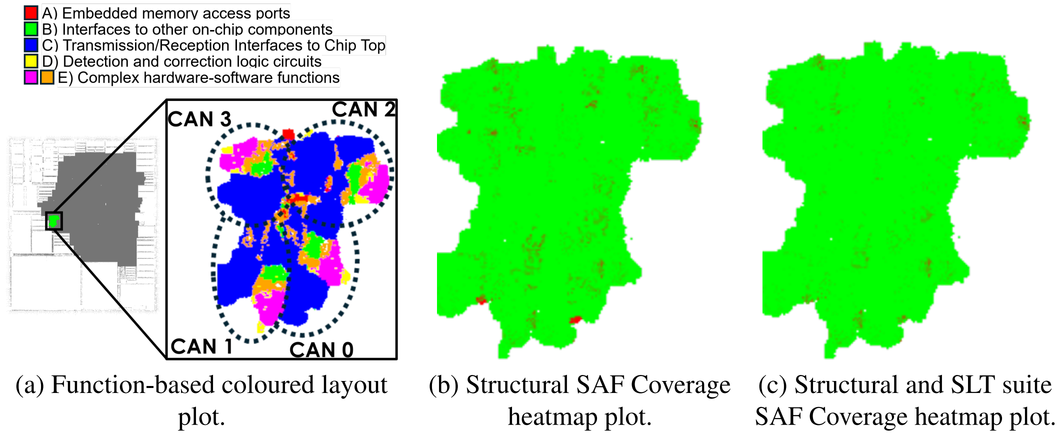



Fig. 5.7: Stress-colored heatmaps in terms of toggle coverage with detailed zoom on some regions. Figure 5.7a: 32 scan based ATPG patterns, 5.7b: 12 Selective ATPG patterns; 5.7c: 1024 Scan based pseudo-random patterns; 5.7d: LBIST patterns; 5.7e: MBIST patterns; 5.7f: RTOS boot.

structural tests. Many red spots disappear, or the areas colored red become much lighter after SLT.




(a) Function-based coloured layout plot. (b) Structural SAF Coverage heatmap plot. (c) Structural and SLT suite SAF Coverage heatmap plot.

Fig. 5.8: Layout view of the four CAN controllers in the CAN peripheral of the DUT.

The algorithms presented in section III.B have been implemented for the cluster of CAN modules. Test patterns are provided according to the deterministic approach shown in [76]. The resulting suite of 6 functional test programs is presented in

| Test Name | Algorithm Number | Mode[1] | Targeted Weakness | Execution Time [cc] | Mem. Foot-print [KB] | Fault Sim. Time[2] [h] | Develop. Time [h] |
|---|---|---|---|---|---|---|---|
| Embedded Memory Access Port | Algorithm 1 | LPBK | **A** | 2,522,475 | 9.96 | 13.96 | 20 |
| Interfaces to other SoC components | Algorithm 2 | NA | **B** | 23,870 | 12.55 | 0.13 | 26 |
| Transmission/Reception to Chip Top | Algorithm 3 | LPBK, CMP | **C** | 332,096 | 6.42 | 1.84 | 22 |
| Detection and correction unit | Algorithm 4 | CMP | **D** | 7,294,466 | 6.82 | 40.39 | 34 |
| Complex transmission functions | Algorithm 5 | LPBK | **E** | 1,002,251 | 6.92 | 5.55 | 10 |
| Synchronization functions | Algorithm 6 | CMP | **E** | 447,579 | 6.77 | 2.48 | 34 |
| | | | **Total** | **11,622,737** | **49.44** | **64.35** | **136** |

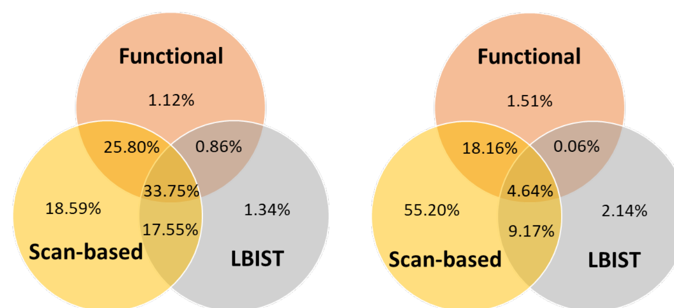
Table 5.4: Characteristics of SLT suite for CAN peripheral.
[1] LPBK = Loopback, CMP = Companion Module. [2] Stuck-at + Transition Delay fault models for a total of ca. 900k faults.

Table 5.4; it reports, for each program, the targeted weakness, as well as the execution time in terms of clock cycles, the memory footprint (code and data), their fault simulation grading time, and their approximate development time by

one person for both program development and verification and companion module development and verification. The functional SLT suite requires 11,622,737 clock cycles, translating into 145 ms with a clock frequency of 80 MHz (the maximum reachable clock for the CAN peripherals in the DUT). Two test engineers developed the suite in 136 hours or 8.5 days.

Meanwhile, the companion modules are synthesized and implemented in a Xilinx MPSoC ZCU 104 Ultrascale Plus+ equipped with 4 Arm A53 cores running a Linux-based operating system and connected to a host computer through Ethernet. The FPGA is directly controllable by a Linux operating system through the PYNQ framework. The laboratory setup is shown in Figure 5.3. This allows the connection of an external CAN node to the chip top, allowing off-chip by the DUT, including all features related to the error injections. The instantiated companion modules occupy 13,071 LUTS, 6,441 FlipFlops, 2 BRAM cores, and three external I/O pins of the programmable logic; it was synthesized and implemented with a clock of 2 MHz.

To grade the System-Level Test functional programs, functional fault simulations [102] are performed using a commercial fault simulator *Z01X (Synopsys)*, for Stuck-At-Faults (SAF) and Transition Delay Faults (SDF) fault models. Table 5.5 presents SLT fault coverage results, including "Single," "Incremental," and "delta" Δ coverages, which represent the individual program coverage, the incremental value to previous tests, and the increment to previous ones, respectively.




(a) Stuck-At fault model. (b) Transition Delay fault model.

Fig. 5.9: Venn diagrams between Structural, LBIST, and functional (SLT) test approaches.

The fault coverage achieved by the structural tests (Scan-based, LBIST, and MBIST) reaches 97.89% for 435,967 SAFs and 89.38% for 435,966 TDF. The

proposed SLT permits reaching up to 99.01% for SAFs and 90.89% for TDFs. Figure 5.9a and Figure 5.9b show that the functional SLT programs add up to 1.12% and 1.51% of fault coverage.

LBIST and Scan-based approaches are also quantified in Figure 6, showing that Scan-based approaches cover most of the faults. Functional SLT emerges to be more powerful than LBIST (e.g., covering more faults of the CAN cluster), but LBIST covers more faults "uniquely" than SLT. MBIST adds no unique coverage and is not reported in the diagrams.

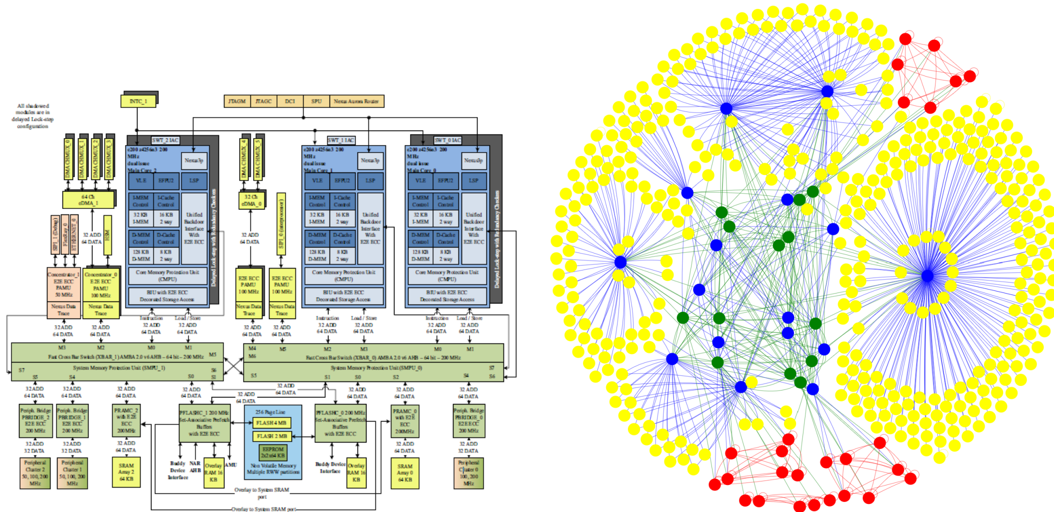
| **Test Nature** | **Test Name** | **Algorithm** | **Fault coverage SAF [%]** | | | **Fault coverage TDF [%]** | | |
|---|---|---|---|---|---|---|---|---|
| | | | **Single** | **Incre-mental** | **Δ** | **Single** | **Incre-mental** | **Δ** |
| Structural | Scan-based | NA | 95.69 | 95.69 | NA | 87.17 | 87.17 | NA |
| | LBIST | NA | 53.46 | 97.89 | 2.2 | 16 | 89.38 | 2.15 |
| | MBIST | NA | 1.13 | **97.89** | 0 | 0.03 | **89.38** | 0 |
| Functional | Transmission/Reception to Chip Top | Alg. 3 | 40.30 | 98.45 | 0.56 | 12.55 | 89.65 | 0.27 |
| | Embedded Memory Access Port | Alg. 1 | 41.79 | 98.65 | 0.2 | 12.77 | 89.73 | 0.08 |
| | Complex transmission functions | Alg. 5 | 25.01 | 98.66 | 0.01 | 5.79 | 89.74 | 0.01 |
| | Interfaces to other SoC components | Alg. 2 | 13.36 | 98.96 | 0.3 | 5.11 | 90.63 | 0.89 |
| | Detection and correction unit | Alg. 4 | 39.94 | 98.99 | 0.03 | 11.82 | 90.67 | 0.04 |
| | Synchronization functions | Alg. 6 | 35.39 | 99.01 | 0.02 | 14.24 | 90.89 | 0.22 |
| | | | **Total** | **99.01** | **1.12** | **Total** | **90.89** | **1.51** |

Table 5.5: Fault coverage for Stuck-at fault model (435,967 faults) and Transition delay fault model (435,966 faults).

The relative improvement by the SLT suite for SAF is about 50% concerning untested faults from structural tests. About TDFs, despite the incremental improvement being higher than for SAFs, the improvement over untested faults is 15%. Given the relatively low coverage for TDF achieved by structural methods, many TDF faults look to be functionally untestable.

### 5.2.2.2 Uncore logic

The general SoC architecture of the case study is presented in Figure 5.10a.



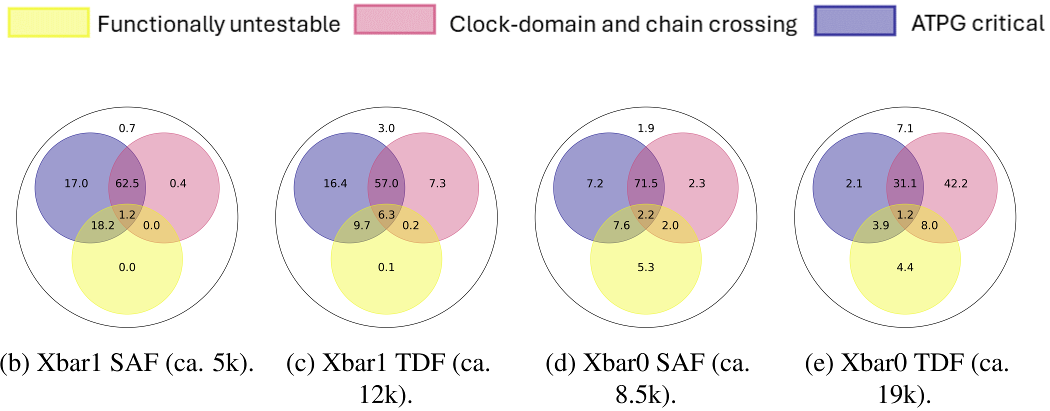
(a) SoC Architecture with cores (blue), peripherals, crossbars and peripherals bridges (green), and additional crossbar masters and memories (yellow).

(b) SoC Nexum Graph Model with 29 masters (red), 397 slaves (yellow), 15 virtual crossbar masters (green), and 14 virtual crossbar slaves (blue).

Fig. 5.10: A qualitative contrast between the SoC Architecture from the User Manual with the SoC Graph model.

Figure 5.10 contrasts the SoC architecture from the User Manual and the generated *Nexum graph* model. Figure 5.10b shows a medium-high number of crossbar masters combined with a very high number of crossbar slaves. Meanwhile, all the edges (possible communication paths) between masters and slaves are 827. According to Equation 4.15, all the possible paths are $M * S = 29 * 397 = 11,513$. The manual creation of the memory map file in CSV format required a total of 10 hours by a single person, including the time spent on verification and debugging. In contrast, by leveraging generative AI, the same memory map file was generated using the DTS description in just 1 hour, which included verification and refining the input prompt. This demonstrates a significant reduction in time and effort, and possibly reducing the risk of errors due to the manual generation.

For the presented case of study, the undetected faults from structural tests are analyzed and labeled following the considerations presented in the previous section in Figure 5.11. For readability reasons, the categories as *clock-domain crossing* and *chain configurations crossing* are merged into a single category, *clock-domain and*




(b) Xbar1 SAF (ca. 5k). (c) Xbar1 TDF (ca. 12k). (d) Xbar0 SAF (ca. 8.5k). (e) Xbar0 TDF (ca. 19k).

Fig. 5.11: Classification of Undetected faults from structural tests for different fault models. All the values illustrated are shown in percentages.

*chain crossing*, in all the figures. Figure 5.11 shows that most undetected stuck-at faults reside in the ATPG critical class. On the other hand, the number of undetected faults for the Transition Delay fault models is bigger than undetected stuck-at, and they spread among different classes.

The $S^2TL$, running on the DUT, implements the runtime test dispatcher; it parses the generated data structure in a high-level programming language and calls the proper function pointers; it is based on an iterative loop over the produced data by the *Grind Nexum algorithm*. Therefore, they can be parallelized during the fault simulation, drastically reducing the computing time. The *Grind Nexum algorithm* can be instructed to split the generated data into multiple partitions to parallelize the fault simulation process. As the last step, fault simulation results are merged to obtain the overall fault coverage of the SLT workload, and they are further merged with fault coverage from structural tests (scan patterns and LBIST).

#### 5.2.2.2.1 The functional SLT suite

Table 5.6 presents the characteristics of the generated SLT functional suite, generated in seconds from the proposed tool, from different perspectives. From a *Grind Nexum algorithm* perspective, it presents for each generated workload the visiting algorithms, the number of masters involved (e.g., five masters accounting for three CPUs, two DMA engines, and two cache controllers for instruction and data

| Name | Visit algorithm | # masters | Tot. # Actions | Execution Time [cc] | Memory Footprint [Kbyte] | Fault Sim. Time [days][3] (est.) | # Fault Sim. Partitions | Avg. Fault Sim. Time per Partition [days] |
|---|---|---|---|---|---|---|---|---|
| App1 | Node covering from masters | 7 | 3,707 | 26,481,222 | 695 | 129.3 | 5 | 12.9 |
| App2 | Edge covering from masters | 7 | 6,536 | 51,071,225 | 870 | 249.3 | 7 | 9.9 |
| App3 | Node covering from slaves | 7 | 2,755 | 20,279,995 | 497 | 99.0 | 5 | 3.45 |
| App4 | Edge covering from slaves | 7 | 24,051 | 161,110,659 | 1,894 | 786.5 | 21 | 11.4 |
| RTOS | NA | NA | 4 | 495,858 | 164,850 | 1.5 | NA | NA |

Table 5.6: Generated SLT workload suite and RTOS characteristics targeting the crossbar module.

[1] Stuck-at + Transition Delay fault models for a total of ca. 270k faults.

cache) and the total number of generated actions. Moreover, it presents the execution time and memory footprint for each SLT workload. The *Grind Nexum algorithm* can split the SLT workload into smaller ones, leveraging on the program's regularity. This is done based on the number of total actions and user-defined requirements, generating n-th data structures integrated into the runtime library ($S^2TL$). This allows for the execution of parallel, independent fault simulation campaigns despite the estimation of an impossible fault campaign. The division into smaller workloads allows the evaluation of the overall SLT workloads in less than 15 days, on average, as the last column in Table 5.6 shows (compared to the estimated months).

A reduction in the number of transactions in the system software test library can impact the coverage, as it limits the exploration of functional communication paths between on-chip components. On one hand, applications generated using the node-covering algorithm (from masters or slaves) focus on reaching specific on-chip components. This approach does not exhaustively traverse the graph model but instead generates transactions aimed at covering nodes. As a result, the SLT applications created are smaller and faster, which can reduce test time but may compromise coverage. On the other hand, applications generated using the edge-covering algorithm aim to exhaustively visit all edges in the graph model; for example a direct communication path between a RAM port and Flash exists outside of the crossbar module by means of a DMA. This ensures that all possible functional communication paths between on-chip components are exercised. While this approach can

significantly improve fault coverage, it also increases the size and complexity of the SLT applications, leading to longer test times.

The RTOS used as the baseline for comparison with the generated SLT suite is a customized version of Micrium µC/OS-III [103] with a software canary in the context switching for maximizing the fault detection. This RTOS is considered a medium-complexity real-time operating system, designed for embedded systems, and mathematical application tasks are added. Its selection as a baseline is justified by its balance of complexity and functionality, providing a reliable reference point for evaluating the effectiveness of the proposed SLT suite. As suggested in prior research [18], using an RTOS like Micrium µC/OS-III ensures a meaningful and practical comparison.

In Table 5.6, the RTOS exhibits a higher memory footprint compared to the SLT application suite due to the inclusion of various kernel modules and complex data structures. However, in terms of execution time, the RTOS shows very low clock cycles and fault simulation time compared to the SLT suite. This is primarily because the measurement is limited to the RTOS bootstrap process, which involves initializing tasks, configuring a subset of peripherals required by the RTOS, and bootstrapping the tasks themselves. This contrast highlights the trade-off between the RTOS's broader functionality and the lightweight, task-specific design of the SLT application suite, allowing a further integration of the SLT application suite in the RTOS.

##### 5.2.2.2.2 Effects on the crossbar logic

The functional programs within the proposed SLT suite have been simulated for the Stuck-at fault model (270k faults among the two crossbars). Experimental results are presented in Table 5.7, in which the column "Single" represents the coverage of a single test approach, and the column "Incr" represents the cumulative coverage of a given test approach with the previous approaches. The column "delta" (Δ) represents the increment for a given test approach concerning the previous one. Moreover, ATPG untestable identified faults have been added in the fault coverage of structural test patterns; meanwhile, functionally untestable faults following the approach identified in [78] are added to functional approaches.

The Stuck-at fault coverage achieved by the structural tests (scan-based and LBIST) reaches 95.18% for crossbar 1 (XBAR1) and 94.28% for crossbar 0 (XBAR0),

| Test Nature | Test Approach | Stuck-at Fault coverage [%] | | | | | |
|---|---|---|---|---|---|---|---|
| | | XBAR1 | | | XBAR0 | | |
| | | Single | Incr | Δ | Single | Incr | Δ |
| Structural | Scan-based | 94.07 | 94.07 | NA | 93.19 | 93.19 | NA |
| | LBIST | 51.99 | 95.18 | 1.11 | 50.44 | 94.28 | 1.09 |
| Functional | App1+App2+ App3+App4 | 49.7 | 97.08 | 1.9 | 53.66 | 95.95 | 1.67 |
| | RTOS boot | 25.31 | 97.08 | 0.0 | 19.13 | 95.95 | 0.00 |
| | | **Total** | **97.08** | **1.9** | **Total** | **95.95** | **1.67** |

Table 5.7: Fault coverage for Stuck-at fault model (ca. 270k faults).

removing identified redundant faults. Combining functional programs in the SLT suite with visiting algorithms provides additional fault coverage for high results. It adds up to 97.08% for crossbar 1 (XBAR1) and 95.95% for crossbar 0 (XBAR0).

Moreover, the proposed SLT suite has been fault simulated also for the Transition Delay fault model (a total of 270k faults among the two crossbars), and experimental results are presented in Table 5.8.

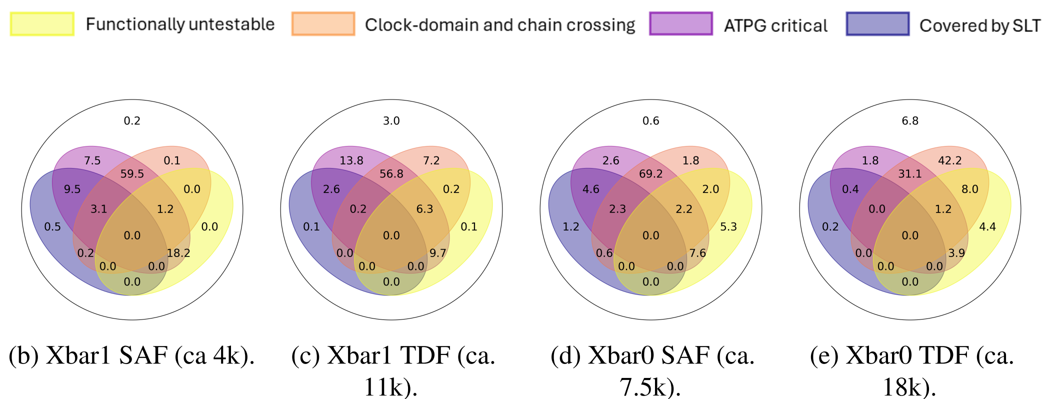
| Test Nature | Test Approach | Transition Delay Fault coverage [%] | | | | | |
|---|---|---|---|---|---|---|---|
| | | XBAR1 | | | XBAR0 | | |
| | | Single | Incr | Δ | Single | Incr | Δ |
| Structural | Scan-based | 87.28 | 87.28 | NA | 87.33 | 87.33 | NA |
| | LBIST | 4.08 | 88.71 | 1.43 | 3.15 | 87.49 | 0.16 |
| Functional | App1+App2+ App3+App4 | 29.32 | 91.11 | 2.4 | 31.61 | 89.76 | 2.27 |
| | RTOS boot | 13.09 | 91.11 | 0.0 | 2.99 | 89.76 | 0.0 |
| | | **Total** | **91.11** | **2.4** | **Total** | **89.76** | **2.27** |

Table 5.8: Fault coverage for Transition Delay fault model (ca. 270k faults).

The Transition Delay fault coverage achieved by the structural tests (scan-based and LBIST) reaches 88.71% for crossbar one and 87.49% for crossbar zero, removing identified redundant faults. Combining functional programs in the SLT suite with visiting algorithms provides additional fault coverage for high results. It adds up to 91.11 % for crossbar one (XBAR1) and 89.76% for crossbar zero (XBAR0). Measuring the complete execution of an RTOS is impractical in terms of fault simulation time, as the RTOS operates in an infinite event-driven loop. Consequently, only the RTOS bootstrapping process and tasks initialization are measured. This approach positively impacts execution time compared to the SLT suite application, as the RTOS initializes fewer peripherals and performs fewer operations during its bootstrap phase. However, it negatively affects the fault coverage achieved by the RTOS in absolute terms. This is because the RTOS only initializes a limited

subset of the available SoC peripherals, and its firmware must be manually written, unlike the proposed SLT methodology, which exhaustively utilizes all available SoC components. As a result, the increment of fault coverage obtained during the RTOS bootstrap compared to structural tests is a subset of the fault coverage achieved by the SLT suite, ca. 0.37% for SAF, and ca. 0.48% for TDF in crossbar 1; and ca. 1.41% for SAF, and ca. 0.00% for TDF in crossbar 0; the proposed SLT suite results into a more comprehensive testing approach compared to the RTOS boot.

Additionally, Figure 5.12 shows the classification of undetected faults from structural tests and the percentage for each labeled class of faults detected by the SLT suite. On the one hand, some faults detected by SLT are still unclassified; on the other hand, other faults detected by SLT reside in different classes (i.e., structural pattern weaknesses).




(b) Xbar1 SAF (ca 4k). (c) Xbar1 TDF (ca. 11k). (d) Xbar0 SAF (ca. 7.5k). (e) Xbar0 TDF (ca. 18k).

Fig. 5.12: Classification of Undetected faults from structural tests and SLT covered faults. All the values illustrated are shown in percentages.

The total increase of fault coverage for the TDF model is more significant than the increase for the SAF model. The difference is because scan-based test patterns for modules between different cross-domain clocks are hard to detect [15], and they would require a big structural pattern set. Instead, experimental results show how SLT can detect such faults in a functional and in-field-like manner without adding a considerable set of structural patterns and that the presence of functionally untestable faults is considerable, as Figure 5.12 shows.

#### 5.2.2.2.3 Effects on the un-core logic

The SLT suite suite is capable of activating the DUT as a whole. Therefore, the

effects of the suite on a small subset of on-chip components are presented to prove its effectiveness further. It is important to highlight that in order to have fault simulations in a reasonable amount of time, the chosen fault lists are the undetected faults (ND), or residual faults, from structural test patterns. The undetected fault lists are obtained by removing the ATPG untestable and functionally untestable faults. In Table 5.9, the effects of SLT suite on a CPU, a peripheral bridge, a CAN peripheral and the top-level entity containing all the memory collars for all the memory controllers (outside any LBIST partitions, the MBIST is not considered for structural tests since it focuses on testing the memory array) are presented for the SAF model.

| Module | Structural Coverage [%] | # faults | ND faults | ND covered [%] | |
|---|---|---|---|---|---|
| | | | | RTOS boot | SLT |
| CPU | 98.03 | 1.5 M | ca. 29 k | 6.46 | 13.96 |
| Periph. Bridge | 90.88 | 76k | ca. 6.7k | 6.94 | 11.60 |
| CAN Periph. | 95.73 | 435k | ca. 19k | 3.97 | 3.98 |
| Memory Controllers | 94.51 | 1.4M | ca. 78.8k | 2.60 | 16.69 |

Table 5.9: Effects of SLT suite on different on-chip modules for SAF model.

The last column represents the percentage of undetected faults by structural test patterns covered uniquely by the hereby proposed SLT suite, compared with the boot of an RTOS.

In Table 5.10, the effects of the SLT suite on a CPU, a peripheral bridge, a CAN peripheral and the top-level entity containing all the memory collars for all the memory controllers (outside any LBIST partitions, the MBIST is not considered for structural tests since it focuses on testing the memory array) are presented for the TDF model.

| Module | Structural Coverage [%] | # faults | ND faults | ND covered [%] | |
|---|---|---|---|---|---|
| | | | | RTOS boot | SLT |
| CPU | 95.13 | 1.5 M | ca. 29k | 0.28 | 1.96 |
| Periph. Bridge | 68.67 | 76k | ca. 6.7k | 0.09 | 0.62 |
| CAN Periph. | 87.18 | 435k | ca. 19k | 0.00 | 0.20 |
| Memory Controllers | 84.23 | 1.4M | ca. 226.5k | 0.14 | 1.50 |

Table 5.10: Effects of SLT suite on different on-chip modules for TDF model.

A further investigation showed that most detected faults reside in the interfaces of the un-core logic to/from the crossbars. Due to limitations in peripheral configuration and overall SoC utilization, the RTOS demonstrates a relatively low fault coverage for undetected faults compared to the proposed SLT suite. It is important to note

that the fault detection capabilities of the RTOS could be significantly improved by extensively utilizing the entire SoC. However, achieving this would require substantial manual effort, which would increase development time and complexity. In contrast, the proposed methodology automatically generates a comprehensive SLT suite that exhaustively tests all available SoC components, providing broader fault coverage without the need for manual intervention. This highlights the efficiency and scalability of the proposed approach compared to the more resource-intensive manual efforts required to enhance RTOS fault detection.

#### 5.2.2.3 Final remarks

A crucial aspect of developing an SLT workload is the manual effort of developing and validating the code. There are no automated methodologies or development guidelines.

For communication peripherals, the proposed technique illustrates how to create a general recipe for developing an effective functional SLT suite based on the identification of structural test weaknesses in communication peripherals. Support for companion modules has been added to address off-chip communications and error injections. In order to validate the proposed methodology, the CAN peripheral has been selected. However, the methodology is peripheral, independent, and portable to other communication modules like UART, SPI, and others.

The crossbar logic is the principal infrastructure and the most critical for handling the communication between different un-core logic components. A new methodology for testing such infrastructure is devised to overcome the limitations of structural tests, and as ripple effect the testing of the un-core components.

The proposed automatic generation of SLT workloads focuses on automatically generating a functional SLT application for on-chip interaction between components, providing the following contributions:

- A graph-based model of the SoC modules described by a DTS file, modeling their means of communication as edges between nodes representing the modules, i.e., the CPUs, DMAs, and peripherals.
- A flexible automated generation methodology, leveraging the abstracted graph-based model for System-Level functional test programs.

- A method to analyze the netlist and its DfT, aimed to classify structural test escapes and assess the capabilities of the SLT suite.

Utilizing standardized DTS files streamlines the generation of memory map files with linear complexity relative to the number of on-chip components, ensuring accurate representation of even complex SoCs with minimal manual effort. Once the graph is generated, a DFS-like traversal is employed to efficiently locate target nodes (masters or slaves) by quickly descending a branch rather than exhaustively exploring every vertex. With a DFS-based complexity of $O(|V|+|E|)$, both graph generation and traversal complete in seconds, making the proposed approach scalable and practical even for SoCs with thousands of components. On one hand, using a node-visit approach, the Grind Nexum Algorithm traverses every on-chip component, generating only one action for each master-slave pair. On the other hand, using an edge-visit approach, the algorithm ensures that all functional communication paths between on-chip components are touched exactly once.

Experimental results obtained on a large automotive SoC by STMicroelectronics demonstrate the effectiveness of the proposed generation methodology by detecting, primarily, test escapes from structural test patterns on well-known fault models (stuck-at and transition-delay faults).

The automatic generation methodology relies on a graph-oriented model of the SoC and its exhaustive traversal. However, a visit algorithm based on AI techniques can be effectively implemented to leverage the graph for generating test data actions. These algorithms may require training with production data to achieve optimal performance. Deploying AI-based approaches could be an effective solution to tailor test data generation efforts more precisely. By integrating production data and customer data, these methods can focus on generating test actions specifically targeted at parts of the SoC that are more prone to failure. This approach not only enhances the efficiency of the testing process but also ensures that critical areas of the system receive greater attention during fault detection and validation.

Fault simulating an SLT workload is computationally intensive since it has an execution time in the order of seconds. Therefore, even if possible, it would require months of fault simulations with all the risks of losing data due to power grid or server-related problems. By exploiting the regularity of System Software Test Library, and the Grind Nexum algorithm, fault simulations can be partitioned and executed.

An important aspect to consider for fault simulation execution time is the number of faults to be simulated, directly correlated to the complexity of the module under test. They strongly impact the execution time.

Therefore, to have a fair comparison of the original generated SLT workload for the DUT and their estimated fault simulation execution time as represented in Table 5.11, the crossbar module has been selected.

```
/* test setup */
....
for (i = 0; i < MODULES_TO_TEST && cpu_ops[i]!=NULL ; i++)
  {
    /* module configuration */
    length=cpu_ops_length[i];
    for (j = 0; j < length && length!=0 ; j++) {
    *err = ERROR_NONE;
    cpu_ops[i][j].func(cpu_ops[i][j].src,cpu_ops[i][j].dest,err);
    /*error check*/
    /*collect partial signature*/
    }
  }
/* test clean up and signature collection */
....
```

Fig. 5.13: $S^2TL$ Test Scheduler.

The $S^2TL$ implements the runtime test dispatcher specific for the generated data by the automatic generation methodology. It includes parsing the generated data structure in a high-level programming language and calling the proper function pointers. An example of the library for a given module under test can be seen in Figure 5.13.

The Test Dispatcher from the $S^2TL$ is structured into three main phases: test setup, module testing, and test cleanup. In the test setup phase, the environment is prepared, and variables are initialized to configure the modules and operations that will be tested. The module testing phase involves iterating through a list of modules that need to be tested. Each module is associated with a set of operations, which are executed sequentially. For each module, the code retrieves the number of operations and iterates through them. Each operation is executed using a function pointer that takes source data, destination data, and an error pointer as arguments. Before executing each operation, the error variable is reset to indicate no errors. After execution, the code checks for errors and collects partial results, referred to as "partial signatures." Finally, the test cleanup phase finalizes the process by

releasing resources and consolidating the collected signatures. This framework is modular and flexible, allowing for the testing of multiple modules and operations while systematically handling errors and collecting results.

It becomes evident that the nested loops are independent of each other, the action $i-th$ is independent from action $i-th+1$ making them suitable for parallelization during fault simulation. Partitioning the workload in sub-workloads can drastically reduce computing time by distributing the sub-workloads across multiple fault simulation campaigns. The Grind Nexum Algorithm can be instructed to partition test data actions, tailored for parallelizing the fault simulation campaigns.

| SLT workload Name | Targeted Module | Execution Time [cc] | Fault Simulation (SAF) Time (est.) [h] | Fault Simulation (TDF) Time (est.) [h] |
|---|---|---|---|---|
| App 1 | Crossbar | 2540006 | 8,314.66 | 9,541,58 |
| App 2 | Crossbar | 4,378,897 | 11,056.87 | 12,159.01 |
| App 3 | Crossbar | 26,481,222 | 86,685.76 | 99,477.20 |
| App 4 | Crossbar | 51,071,225 | 167,180.65 | 191,850.01 |
| App 5 | Crossbar | 20,279,995 | 66,386.16 | 76,182.18 |
| App 6 | Crossbar | 161,110,659 | 527,392.59 | 605,215.20 |

Table 5.11: Original SLT workload suite.

The partitioned workloads are presented in Table 5.12, demonstrating that the proposed generation methodology can effectively partition the workloads and create feasible, parallel fault simulation campaigns.

On the other hand, the fault simulation execution time still depends on the number of faults, the complexity of the module under test, the SLT workload, and the fault model. The proposed methodology aims to lighten the grading process of SLT workload in a reasonable time, as results in Table 5.12 depicts.

| SLT workload Name | Partition Number | Execution Time [cc] | Fault Simulation (SAF) Time (est.) [h] | Fault Simulation (TDF) Time (est.) [h] |
|---|---|---|---|---|
| App 1 | 0 | 820,018 | 2,684.313 | 3,80.41 |
| | 1 | 820,018 | 2,684.313 | 3,80.41 |
| | 2 | 1,051,673 | 3,442.63 | 3,950.62 |
| App 2 | 0 | 820,018 | 2,161.04 | 2,367.44 |
| | 1 | 820,018 | 2,161.04 | 2,367.44 |
| | 2 | 820,018 | 2,161.04 | 2,367.44 |
| | 3 | 820,018 | 2,161.04 | 2,367.44 |
| | 4 | 1,051,673 | 2,771.54 | 3,036.24 |
| App 3 | 0 | 1,160,120 | 2,113.49 | 2,107.19 |
| | 1 | 2,024,011 | 3,266.1 | 3,827.91 |
| | 2 | 597,894 | 1,385.77 | 1,372.68 |
| | 3 | 4,859,545 | 71,348.92 | 14,180.33 |
| | 4 | 18,010,516 | 41,177.87 | 45,495.44 |
| App 4 | 0 | 2,305,619 | 3,740.88 | 7,431.18 |
| | 1 | 3,433,347 | 5,038.49 | 6,445.9 |
| | 2 | 1,179,055 | 2,409.14 | 2,740.06 |
| | 3 | 9,240,142 | 14,292.72 | 18,950.68 |
| | 4 | 12,336,873 | 22,001.14 | 27,385.47 |
| | 5 | 9,839,071 | 15,919.74 | 23,421.76 |
| | 6 | 12,993,414 | 21,973.48 | 27,826.69 |
| App 5 | 0 | 13,393,897 | 12,560.78 | 10,628.94 |
| | 1 | 3,594,906 | 6,825.98 | 7,108.69 |
| | 2 | 554,650 | 1,229.59 | 1,073.56 |
| | 3 | 1,453,703 | 2,357.31 | 2,686.64 |
| | 4 | 1,453,703 | 2,519.41 | 2,718.64 |
| App 6 | 0 | 2,881,120 | 5,716.26 | 8,056.31 |
| | 1 | 4,041,848 | 7,837.82 | 10,355.62 |
| | 2 | 2,794,327 | 4,352.93 | 5,359.01 |
| | 3 | 3,861,129 | 6,047.35 | 8,135.6 |
| | 4 | 7,143,753 | 12,501.85 | 17,432.64 |
| | 5 | 7,075,729 | 13,228.62 | 17,183.05 |
| | 6 | 9,627,088 | 15,987.04 | 22,255.12 |
| | 7 | 7,080,465 | 13,057.51 | 19,783.51 |
| | 8 | 7,019,809 | 12,850.24 | 19,536.61 |
| | 9 | 14,451,751 | 17,955.53 | 17,955.53 |
| | 10 | 9,979,946 | 19,112.99 | 25,565.72 |
| | 11 | 9,979,912 | 17,463.92 | 25,479.63 |
| | 12 | 14,593,459 | 68,114.51 | 28,643.7 |
| | 13 | 9,981,140 | 16,959.73 | 22,847.8 |
| | 14 | 10,485,569 | 17,884.5 | 23,841.41 |
| | 15 | 16,024,926 | 25,797.17 | 34,146.38 |
| | 16 | 7,223,949 | 11,845.78 | 14,761.96 |
| | 17 | 9,547,238 | 15,731.65 | 18,459.72 |
| | 18 | 1,275,203 | 2,176.85 | 3,580.76 |
| | 19 | 2,881,930 | 5,300.82 | 7,696.99 |
| | 20 | 4,014,688 | 7,067.18 | 9,303.7 |

Table 5.12: Partitioned SLT workload suite for Crossbar Module.

## 5.3 Indirect methodologies experimental results

### 5.3.1 Execution trace analysis

Experimental results are provided in different contexts. First, the proposed methodology has been used to analyze already developed Software Test Libraries for core online testing [47]. Then, it evaluates SLT workloads based on different versions of an existing Real-Time Operating System (RTOS). Finally, two SBST programs have been developed from scratch using an Evolutionary Optimizer called $\mu GP$ [29] to estimate the boost in the development process. The SBST development time is compared between a flow using the proposed metrics and a flow based on fault simulation results. The experimental results demonstrate that low connectivity of the CDFG always correlates to low fault coverage values. In all cases, the newly proposed metrics have been extremely efficient in developing and improving functional routines and avoiding extraordinarily long and repetitive fault simulation campaigns.

For comparison, functional test programs are also evaluated on the gate-level netlist description of the DUT for the stuck-at-fault model using Synopsys's commercial fault simulator *Z01X*. When these times are excessively long, CPU time estimations are provided only.

#### 5.3.1.1 Evaluation of Functional Test Programs

During the first set of experiments, we analyze SBST programs belonging to a Core Self Test library written in assembly code (avoiding compiler optimizations) [47, 69] developed in about 6 months. SBST programs are used during online testing to target all potential stuck-at-faults (SAF) in CPU modules.

The first section (the one on top) of Table 5.13 reports findings in this area. To understand the complexity of each program, their size (in bytes), execution time (in clock cycles), and the number of instructions executed are reported. Columns "Connectivity", "$C_{RBL}$ ", "$C_{RBT}$", and "$C_{Rt}$" indicate the connectivity metrics previously defined, i.e., the *Connectivity metric*, the *Register Bit-Level Connectivity*, the *Register Average Bit-Toggle Connectivity*, and the *Register Bit-Toggle Tendency Connectivity*. Column "Grading Time" reports the execution time. This is mainly due to the extraction of the instruction trace. The last three columns are dedicated to fault-

related information and report the number of stuck-at faults (column "SAF") for the module under test, the fault coverage (column "SAF cov."), and the single-threaded fault simulation time for each program.

| Program name | | Code size [bytes] | Execution Time [cc] | Executed instructions | Connectivity [%] | $C_{RBL}$ [%] | $C_{RBT}$ | $C_{RT}$ | Grading Time [s] | SAF | SAF cov. [%] | Fault simulation CPU time [h] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Adder | | 2,708 | 3,388 | 1,082 | 95.67 | 97.66 | 7.32 | 6.25 | 40 | 20k | 92.56 | 24.39 |
| Multiplier | original | 5,612 | 7,002 | 1,332 | 82.20 | 89.26 | 12.17 | 7.50 | 33 | 64k | 92.30 | 46.09 |
| | improved | 5,584 | 6,981 | 1,320 | 82.65 | 91.21 | 12.27 | 7.80 | 33 | 64k | 92.30 | 43.14 |
| Floating Point | | 13,948 | 78,286 | 17,645 | 97.53 | 75.78 | 183.28 | 8.90 | 680 | 70k | 90.78 | 1,407.11 |
| Shifter | original | 6,344 | 7,226 | 2,079 | 96.28 | 87.66 | 16.81 | 6.55 | 80 | 17k | 86.19 | 169.61 |
| | improved | 6,344 | 7,226 | 2,078 | 96.54 | 89.22 | 16.92 | 6.84 | 80 | 17k | 89.48 | 39.86 |
| Count-zeros | | 1,408 | 3,823 | 1,112 | 94.30 | 80.47 | 4.92 | 2.61 | 43 | 3k | 86.81 | 12.19 |
| Bit-wise Logical | | 680 | 513 | 146 | 100.00 | 90.62 | 6.73 | 4.65 | 5 | 3k | 95.00 | 5.65 |
| Load-Store | | 3,016 | 4,228 | 1,052 | 49.81 | 61.72 | 11.72 | 3.86 | 40 | 13k | 52.54 | 11.19 |
| BTB | | 4,476 | 31,135 | 3,794 | 45.41 | 63.44 | 4.97 | 2.73 | 133 | 20k | 71.16 | 66.86 |
| RTOS V1 | original | 130,024 | 210,816 | 16,715 | 24.78 | 12.94 | 1.28 | 1.19 | 1,108.43 | 1.5M | NA | 7,154.92 (est.) |
| | enhanced | 131,906 | 467,840 | 19,895 | 35.62 | 15.43 | 2.29 | 1.30 | 1,349.97 | 1.5M | NA | 10,655.26 (est.) |
| RTOS V2 | original | 162,795 | 294,134 | 37,094 | 14.41 | 15.28 | 1.27 | 1.25 | 1,430.82 | 1.5M | NA | 9,982.64 (est.) |
| | enhanced | 164,850 | 495,858 | 40,094 | 20.31 | 15.97 | 1.89 | 1.31 | 1,546.50 | 1.5M | NA | 11,293.39 (est.) |
| RTOS V2 enhanced opt. | Loop opt. | 164,936 | 537,391 | 40,201 | 20.29 | 16.02 | 1.90 | 1.31 | 1,550.67 | 1.5M | NA | 12,239.33 (est.) |
| | L/S opt. | 164,872 | 535,961 | 40,094 | 20.31 | 16.11 | 1.91 | 1.31 | 1,546.54 | 1.5M | NA | 12,205.75 (est.) |
| | Loop unroll. | 170,168 | 408,525 | 30,487 | 20.91 | 17.38 | 0.78 | 1.22 | 1,175.93 | 1.5M | NA | 9,281.56 (est.) |
| | Regs Live | 166,028 | 534,611 | 39,993 | 20.39 | 14.04 | 0.84 | 1.21 | 1,542.64 | 1.5M | NA | 12,176.00 (est.) |
| | Min dist L/S | 164,854 | 534,948 | 40,094 | 20.30 | 15.97 | 1.89 | 1.31 | 1,546.50 | 1.5M | NA | 12,206.45 (est.) |
| | all opt. | 170,348 | 439,072 | 32,849 | 13.24 | 14.75 | 0.59 | 1.16 | 1,266.96 | 1.5M | NA | 10,000.07 (est.) |

Table 5.13: Functional test programs evaluation for benchmark applications: STL for online testing and RTOSs for SLT.

Although each program has its intrinsic characteristics, the grading time is proportional to the number of executed assembly instructions. Each assembly instruction's structural complexity has a significantly lower influence on the grading time thanks to a clever data structure containing all the necessary information. The shadowed analysis for identifying weaknesses in the source code allowed us to improve some of the test programs. In particular, we identified a lack of connectivity for the *Shifter* test program due to a single instruction affected by a WAW hazard. Once this issue was rectified, the *Connectivity metric* increased from 96.28% (line "original") to 96.54% (line "improved"). Similar improvements can be seen on the other connectivity-related metrics, and its fault coverage increased by more than 3%, from 86.19% to 89.48%. For the *Multiplier* test program, a few instructions did not propagate their output value to an observable point and were thus marked as useless. In our test case, this observable point was a software signature in memory. Therefore, the useless instructions have been removed from the specific Multiplier SBST. In this way, the execution time and the memory footprint of the program were reduced (ca. 0.5% for ROM footprint and ca. 0.3% for the execution time) while the fault coverage remained 92.3%.

Notice that the above improvements have been obtained with small computational efforts: In all the cases, the time required by the proposed methodology is in the

order of seconds, whereas fault simulation requires hours of CPU time. The proposed connectivity metrics provide valuable insights into the data patterns used by the functional test programs depending on the nature of the unit under test. For a pipeline sequential module such as the floating point unit, to effectively test the module, the characteristics of the data patterns are high values of registers bit toggle and tendency mean, repetitively applied to allow the sequential logic cones to propagate faults to the outputs. For arithmetic combinational modules like the adder, multiplier, and shifter, efficient test data patterns require only a few transitions among registers to achieve an acceptable level of fault coverage. Regarding combinational input-constrained modules, such as the load and store unit, count-zeros, and bit-wise logical unit, they do not accept the whole representable range of values. On the contrary, they must use specific test data patterns leading to low register-oriented connectivity metrics, but they must be very effective. Therefore, depending on the nature of the unit under test for SBSTs, register-oriented connectivity metrics can progress in different acceptable ranges for high-quality functional test programs.

The second section of Table 5.13 (the one marked as “RTOS”) reports an analysis of an RTOS used for SLT and written in a high-level programming language. The device can run a basic and an advanced version of the Micrium-C Operating System III [103] and different application tasks mainly related to mathematical operations. The RTOS code has been compiled using the optimization level 2 as baseline (“RTOS V1/V2” in the second section of Table 5.13), and with different optimization strategies that affect the generated machine code in the executable, e.g., loop optimizations, load/store optimizations, loop unrolling, control the register live range, control the distance between load and store of a variable, respectively “Loop Opt.”, “L/S Opt.”, “Loop Unroll.”, “Regs Live”, “Min dist L/S” and all the previous optimization in “RTOS V2 enhanced Opt.” in the second section of Table 5.13. Our methodology is motivated by the fact that the fault simulation of these applications and, more generally, of complex system-level test programs is practically unfeasible [18], due to the enormous computation times, estimated to be at least equal to one year.

As the table shows, the RTOS’s first version (V1), with a small set of application tasks, has very low connectivity. When this version of the RTOS is enhanced with a signature computation inserted in every context switching between tasks, the basic connectivity increases by roughly 10%. With longer application tasks, version V2 of the RTOS dramatically pushes down the overall basic connectivity metric while slightly increasing the register-oriented connectivity metrics. Enhancing the RTOS

with a signature computation in every context switch increases our connectivity metrics.

Compiler optimizations are crucial in determining the number of weak instructions and their positions in the execution trace when grading functional test programs written in high-level languages. Unlike SBSTs (test routines written in assembly code), where code can be precisely modified, high-level code cannot be adjusted as effectively. As expected, experimenting with different optimization strategies (e.g., "RTOS V2 enhanced Opt." in the second section of Table 5.13) leads to changes in connectivity metrics, which follow the variations in the generated assembly code in different directions. For instance, compared to the baseline "RTOS V2 enhanced," compiled with the optimization level O2 (as shown in the second section of Table 5.13), introducing different optimization techniques can reduce the code size and/or the number of executed instructions. The latter directly impacts connectivity-related metrics. On one hand, reducing the distance between the load and store of a variable and optimizing loops and load/store instructions can slightly affect these metrics. On the other hand, optimization strategies that increase code size and the number of executed instructions tend to produce more uniform machine code with higher data flow among instructions, thereby increasing connectivity metrics. However, it is important to note that excessive optimization may reduce connectivity-related metrics, as represented by the "RTOS V2 enhanced Opt. - all opt." row. Additionally, it is critical to observe that the generated assembly instructions strongly depend on the structure of the high-level code.

Generally, a high-quality SLT workload should exercise the DUT and effectively collect computed results with various data patterns. Consequently, by applying our methodology to different versions of the RTOS, it is easy to understand that the RTOS may fail to capture faults at the system level. Since they strongly rely on control verification points, the BTB and RTOS programs include different undecided branches (i.e., if-then-else statements). As previously discussed, undecided branches without the alternative flow in the instruction trace are marked as undecided. To correctly evaluate them, the functional test programs must be executed twice with the fault injector active to force a change in the control and data flow of the program, using the methodology proposed in [92].

The last column of Table 5.14 presents the connectivity results after the fault injection in undecided branches. The table shows that the RTOS and the BTB can

| Program name | | Connectivity [%] | # Undecided branches | Connectivity post Fault Injection [%] |
|---|---|---|---|---|
| BTB | | 45.41 | 676 | 62.54 |
| RTOS V1 | original | 24.78 | 1,052 | 31.53 |
| | enhanced | 35.62 | 1,215 | 39.66 |
| RTOS V2 | original | 14.41 | 9,481 | 18.64 |
| | enhanced | 20.31 | 9,481 | 23.77 |
| RTOS V2 enhanced | Loop opt. | 20.29 | 9,496 | 22.26 |
| | L/S opt. | 20.31 | 9,481 | 22.25 |
| | Loop unroll. | 20.91 | 2,242 | 23.09 |
| | Regs Live | 20.39 | 9,428 | 22.24 |
| | Min dist L/S | 20.30 | 9,481 | 22.24 |
| | all opt. | 13.24 | 1,192 | 15.37 |

Table 5.14: Connectivity metrics post-fault injection for undecided branches (the average injection time is about 2.0 s).

effectively capture changes in the control flow as the connectivity metric increases after the fault injection. Notice that all metrics evolve like the basic connectivity; however, they are not reported in the table for space.

#### 5.3.1.2 Developing New Functional Test Programs

To further prove the effectiveness of the proposed methodology for developing functional test programs, an Evolutionary Optimizer $\mu GP$ [29] is used. The same experimental setup introduced in the previous section is adopted to automatically extract the instruction traces and register files for the proposed methodology 5.2. The connectivity metrics are fitness values for the evolutionary optimizer $\mu GP$.

The targeted fault coverages for generating functional test programs from scratch for both the multiplier and integer divider units were deliberately chosen to meet or exceed the fault coverages achieved by previously developed test programs [47]. These earlier test programs were created through a combination of multiple evolutionary execution and handwritten methods over a development period of approximately six months. The primary objective of this section is to present how the proposed methodology significantly accelerates the development process for functional test programs while maintaining or surpassing the fault coverage benchmarks of prior approaches. By leveraging this methodology, the efficiency of generating high-quality test programs can be significantly enhanced, reducing the time and effort required.

Figure 5.14 shows the evolution of a test program for a multiplier unit in terms of connectivity and fault coverage.

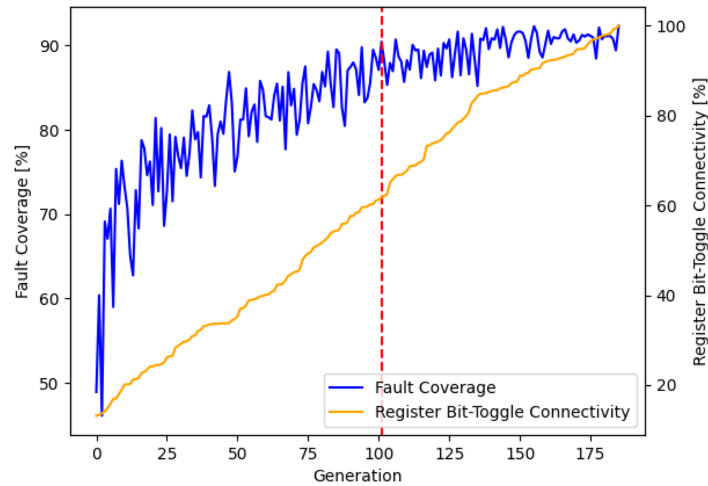



Fig. 5.14: Evolutionary generation of SBST for the multiplier unit.

The plot illustrates the characteristics of the best programs in each population during its evolution generation after generation (along the x-axis). Due to the structure of the generated test programs, the values of the connectivity metrics are always extremely high; however, they are not reported for the sake of readability. On the contrary, the register bit-toggle connectivity grows even when the other connectivity metrics have already reached a saturation point. The plot shows its variation from a bit over 0% (i.e., no toggles at all) to about 100% (i.e., all bit toggles for the best individual of the last population). The best test programs are also fault simulated (using the SAF model), and the evolution is plotted together with the connectivity (in blue). After 101 generations, we reached a fault coverage of about 90% (the targeted fault coverage), as illustrated by the vertical and dotted red line. As the generations progress, fault coverage shows a growing trend with reduced oscillations between best-identified test programs. The $\mu GP$ tool evaluated a total of 7868 distinct programs, with an evaluation runtime of about 70 seconds each and 153 hours (about 6.4 days) of execution time to achieve the targeted fault coverage. Nonetheless, by letting the experiments run to see if it could be further improved, it reached a fault coverage of 92.34% at generation 185, a slight increase compared to the same SBST in Table 5.13. Overall, $\mu GP$ evaluated 14242 programs in 277 hours (i.e., about 11.5 days). With the current setup, the time taken by the on-chip evaluation is mainly used to generate the instruction trace; the trace analysis takes less than one second for each program. Thus, if a faster trace generation

were implemented, the runtime would significantly reduce. In contrast, running a single-thread fault simulation campaign of the given programs would take up to 27 minutes each, over 23x longer than the proposed development flow based on the proposed methodology.

To further substantiate the claims regarding the methodology's effectiveness, additional experiments on a different device unit, the integer divider unit, are performed. Figure 5.15 shows the evolution of a test program for an integer divider unit.

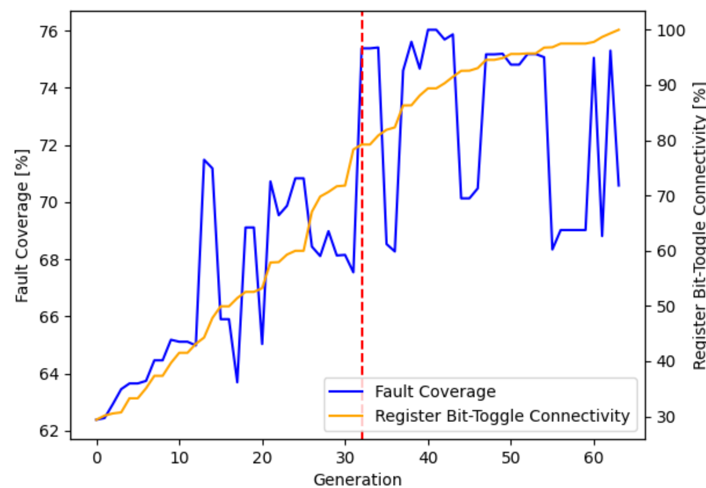



Fig. 5.15: Evolutionary generation of SBST for the integer divider unit.

Figure 5.15 shows (generation after generation) the evolution of the best programs for the population in terms of fault coverage (blue line) and the register bit-toggle connectivity. Figure 5.15 shows that the register bit-toggle connectivity monotonically increases with a certain regularity. On the contrary, the fault coverage oscillates in a range of about 10% even if it follows an increasing overall trend. This behavior is mainly due to the sequential pipeline nature of the module and the generated data pattern by $\mu GP$. As for the previous experiment, the plot shows that in merely 32 generations (highlighted by the vertical and red dotted line), we reach a fault coverage of about 75% (the targeted fault coverage), after evaluating 3654 programs. The previous experiments evaluated each program in about 70 seconds, requiring a total evaluation time of approximately 71 hours (i.e., 3 days). By allowing

the evolution to run for more generations, the best program achieved 76.03% fault coverage in only eight generations and about 800 evaluated programs.

| **Generation Strategy** | **Execution Time [h]** | **Number of individuals** | **Best Program Fault Coverage [%]** |
|---|---|---|---|
| Baseline (FSIM) | 830 | 1855 | 75.07 |
| Proposed methodology | 71 | 3654 | 75.38 |

Table 5.15: Comparison of different generation strategies for developing the SBST for the divider unit.

The same evolution was performed using fault coverage as a target metric to compare the achieved result with a more traditional generation of test programs. Table 5.15 shows the results of the fault simulation-based and the connectivity-based evolutions side by side. The two methodologies reach very similar results, and the fault-simulation-based approach requires around half of the generated programs to reach the desired fault coverage. Nonetheless, the proposed methodology requires only a fraction of the time to evaluate each individual, thus reaching the expected result much faster than the counterpart with a speed-up of 8.55x. As a final consideration, it is worth highlighting the possibility of fine-tuning the result reached with the connectivity metric. By using $\mu GP$, the evaluator function can be replaced to switch from the connectivity metric to the fault coverage, thus evolving a population composed of already promising individuals instead of starting from a population of random individuals.

#### 5.3.1.3 Multicore Connectivity analysis

The case study is a multicore device that includes three PowerPC VLE ISA CPUs. Therefore, multiple execution traces must be analyzed to evaluate the test programs correctly.

The execution trace is produced while each core of the SoC under test is running contemporarily. To correctly align the traces of execution, each trace dump containing the information about the currently executed instruction is complemented by a progressive number functioning as a timestamp. Two instructions with the same timestamp are considered to be executed simultaneously on different cores.

The set of SLT Functional Programs listed in table 5.16 was evaluated using the multicore trace analysis.

The rows of the table are subdivided according to the type of SLT workload considered. The table starts with analyzing a Single Core Real-Time Operating System (RTOS), using as a starting point Micrium C OS-III [103], then extended with a custom multicore version. Asymmetric multiprocessing (AMP) applications exercise two CPUs, one running a bare-metal application while the second executes a single-core version of the RTOS. At the same time, the Symmetric MultiProcessing (SMP) RTOS treats all three cores equally with a multicore scheduler. Ad-hoc developed SLT workloads are reported, too. All programs are written in high-level C language and compiled using optimization level two (basic) as a baseline. Table 5.16 reports information for each SLT workload.

The column Executed Instructions reports the number of low-level assembly instructions retrieved along the trace phase. The SLT suite shows application flows ranging from a hundred thousand to fourteen million traced instructions. Timing-wise, the SLT program execution duration spans from a few hundred milliseconds to some seconds.

Connectivity metrics results for single and multicore SoC are reported. The columns named Core 0, Core 1, and Core 2 report the connectivity metric values computed with the single core metric described in [45]; the Multicore column reports the connectivity value measured with the multicore approach proposed in this paper. Such figures highlight how the single-core connectivity may be significantly far from the most correct one computed over all cores.

The estimated fault simulation CPU time in hours is reported in the FSIM column. With a medium-sized system like the one used in this work, being able to fault-simulate SLT has become way more utopic than in the past.

The execution time of the methodology is finally illustrated. The execution time is divided into trace generation, file transfer, and algorithm computation on the host PC. Trace generation is the most time-consuming activity, costing about 20ms to dump a single instruction.

In the experimental cases, AMP and SMP RTOS variations span from a basic version to more cured ones, enhanced by tuning compile options or fixing evident deficiencies of the original code emerging from the results analysis, as Table 5.16 shows for XBAR applications that were developed in sequence and addressing each a specific weakness that was highlighted by connectivity drops.

| SLT Functional Programs | Executed Instructions | Est. Execution Time FSIM [h] | Connectivity metric [%] | | | | Execution Time[s] | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Core 0 | Core 1 | Core 2 | Multicore | Dump Trace | Transfer | Analysis | Total |
| SingleCore RTOS | 108,235 | 7,154.92 | 72.4 | NA | NA | 72.4 | 3,720 | 152 | 66 | **3,938** |
| AMP RTOS | 145,848 | 9,641.34 | 75.19 | 23.71 | NA | 76.79 | 3,360 | 208 | 90 | **3,658** |
| AMP RTOS - fixed | 145,829 | 9,640.09 | 73.89 | 70.15 | NA | 74.87 | 4,680 | 204 | 93 | **4,977** |
| SMP RTOS | 154,493 | 10,411.14 | 73.93 | 79.15 | 79.15 | 75.57 | 6,660 | 219.2 | 92 | **6,971.2** |
| SMP RTOS - O1 | 175,227 | 11,583.45 | 70.71 | 82.87 | 82.87 | 72.83 | 5,040 | 243.2 | 99 | **5,382.2** |
| SMP RTOS - Conserve stack | 157,642 | 10,420.99 | 73.89 | 79.15 | 79.15 | 75.53 | 3,120 | 220.8 | 96 | **3,436.8** |
| SMP RTOS - Address anchor | 157,648 | 10,421.38 | 73.89 | 79.15 | 79.15 | 75.53 | 3,540 | 222.4 | 96 | **3,858.4** |
| SMP RTOS - No omit frame pointer | 165,299 | 10,927.15 | 76.72 | 88.36 | 88.36 | 78.08 | 4,500 | 233.6 | 100 | **4,833.6** |
| XBAR SLT app 1 | 364,293 | 24,081,74 | 90.14 | 85.15 | 84.76 | 87.74 | 13,440 | 500 | 816 | **14,756** |
| XBAR SLT app 2 | 376,580 | 24,893,99 | 89.72 | 84.33 | 84.13 | 86.97 | 12,840 | 516.8 | 2,051 | **15,407.8** |
| XBAR SLT app 3 | 664,265 | 43,911.51 | 91.13 | 91.88 | 92.06 | 91.65 | 22,831 | 411.2 | 722 | **23,963.7** |
| XBAR SLT app 4 | 14,196,439 | 938,461.54 | 83.07 | 83.21 | 83.17 | 83.15 | 487,927 | 8,560 | 4,218 | **500,704.7** |

Table 5.16: Experimental results for different SLT workloads.

#### 5.3.1.4 Final remarks

The proposed methodology for analyzing execution traces and computing connectivity metrics has proven to be effective for evaluating and improving functional test programs for both core and system-level testing. The connectivity metrics provide valuable insights into the quality of the test programs and help identify weaknesses in the source code that can be addressed to enhance fault coverage.

The experimental results demonstrate that the proposed methodology can be applied to various types of functional test programs, including Software Test Libraries (STLs) for online testing, Real-Time Operating Systems (RTOSs) for SLT, and automatically generated test programs using evolutionary optimization techniques like $\mu GP$. In all cases, the connectivity metrics correlate well with the fault coverage for SAF model, allowing faster development and improvement of test programs without time-consuming fault simulation campaigns.

The methodology has also been extended to support multicore devices, enabling the analysis of execution traces from multiple cores running simultaneously. The multicore connectivity metric provides a more accurate assessment of the overall test program quality than single-core metrics. The case study involving a multicore device with three PowerPC VLE ISA CPUs demonstrates the effectiveness of the multicore trace analysis in evaluating various types of SLT workloads, including single-core RTOS, asymmetric multiprocessing (AMP), and symmetric multiprocessing (SMP) applications.

In terms of evaluation, the proposed methodology exhibits parallels to both debugger-based [104] and microarchitectural fault injection techniques [105]. However, debugger-based fault injection evaluates the resilience of the functional test program to perturbations in user register values throughout its execution. While it is

technically possible to check the system state after every instruction, doing so incurs a high computational cost due to the need to compare against a golden model during functional testing. In contrast, micro-architectural fault injection provides control and observability over all micro-architectural registers and the application output. However, it does not offer feedback on which parts of the functional test program are essential for capturing faulty behavior. Like debugger-based fault injection, it grades the test program over its entire execution without pinpointing specific areas for improvement. The proposed methodology, on the other hand, grades functional test programs from a test program fault detection perspective by unrolling and analyzing their execution. It evaluates the quality of the test program by relying solely on the user's view of the registers and the punctual data propagation among them for each instruction, independently from the fault model. This approach highlights whether a fault may be masked by an instruction or propagated toward the end of the test program, providing valuable feedback at the instruction level. This fine-grained feedback makes the proposed methodology particularly useful for improving the quality of functional test programs.

The execution time of the proposed methodology is significantly faster than fault simulation, making it a practical and efficient approach for developing and improving functional test programs. Most of the execution time is spent on generating the instruction traces, while the actual analysis and computation of connectivity metrics take only a fraction of the total time.

In conclusion, the proposed methodology based on execution trace analysis and connectivity metrics offers a powerful and efficient approach for evaluating and improving functional test programs from a fault detection capabilities point fo view.

### 5.3.2 Generating power hungry System-Level Test workloads

A super-scalar, out-of-order RISC-V processor, called Berkeley Out-of-Order Machine (BOOM) [106], has been used for this experimental setup. The BOOM core is running on an FPGA to obtain the power consumption of the generated snippets. Meanwhile, it is also modeled in the `Gem5` [81] architectural simulator to extract the IPC. The power consumption and IPC are fitness values for the genetic framework *MicroGP*.

Experimental results show that the two-step generation is more effective in converging than the single-step approach, which uses only power consumption as fitness. Moreover, to prove the effectiveness of the microarchitectural simulations, the first step is substituted with a random fitness generator.

#### 5.3.2.1 Experimental setup

The experimental setup is shown in Figure 5.16. The Design-Under-Test (DUT) is an out-of-order, super-scalar CPU called *Berkeley Out of Order Machine (BOOM)* [106] based on the *RISC-V* instruction set architecture. The DUT is synthesized for the Xilinx VC707 FPGA evaluation board and runs at 25 MHz. A Stanford Research SR560 Low-Noise Voltage Preamplifier is connected to a 0.5 mΩ shunt resistor on the FPGA board. The SR560 has been set to a gain of 500 and we enabled the low-pass filter with a 12 dB/oct roll-off and a 1 kHz cutoff frequency. The inputs to the SR560 are DC coupled. The oscilloscope is set up to show 5 ms/div on the x-axis and 1 A/div on the y-axis. The oscilloscope measures the power consumption from the average current.

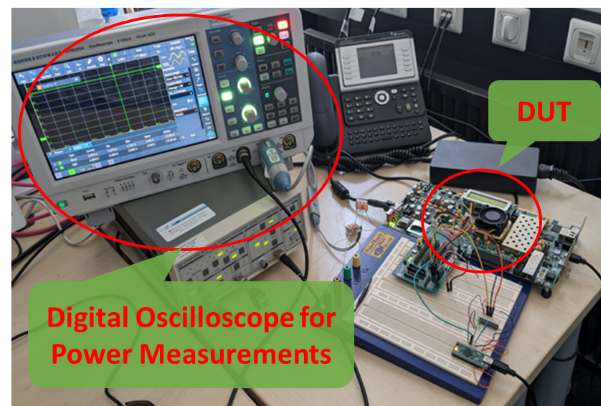



Fig. 5.16: Experimental setup for power consumption measurements.

Microarchitectural simulations are executed off-chip to speed up the evaluation process using *Gem5* [81]. The BOOM core is modeled and evaluates only the stress code for extracting the microarchitectural measurements. The microarchitectural simulations last in seconds, which is a significant speed-up compared to logic simulations that go in the order of minutes for short SLT workloads.

The evolutionary tool used in this technique is *MicroGP* (or *$\mu$GP*) [79]. Its genetic parameters affecting the individuals are reported in Table 5.17.

| Parameter | Value |
|---|---|
| Initial Population Size | 250 |
| Genetic operators | Mutation & Crossover |
| Max Generations | 10 |
| Generated instructions | Min 18 & Max 550 |

Table 5.17: *MicroGP* genetic parameters.

The initial register contents of the integer and floating point registers for the first step in the approach and the direct power measurement run is *0xCAFECAFECAFE-CAFE*. The contents only get optimized in the second step of the approach. This has been done to reduce the time *MicroGP* needs to generate the initial population, as that seems to be the slowest step overall.

As baseline, *MicroGP* run for nine generations, solely using power consumption as fitness value. It took about 18 days to complete and generated 6,264 snippets. The best snippet in the final generation has a power consumption of 0.6823 W. This run is represented by *Baseline* in Table 5.18.

#### 5.3.2.2 Two Step Generation

Table 5.18: Comparison of different generation strategies

| Generation strategy | Architectural Simulation | | On-chip Measurements | | Speed-up | Number of generated individuals | Power consumption best individual [W] |
|---|---|---|---|---|---|---|---|
| | Fitness | Execution Time | Fitness | Execution Time | | | |
| Baseline | NA | NA | Power Consumption | 18 days | – | 6,462 | 0.6823 |
| Two-step | Random | 8.39 days | Power Consumption | 9.2 h | 2.05x | 6570 | 0.045 |
| Two step | IPC | 9.6 days | Power Consumption | 8 h | 1.81x | 4,621 | 0.7142 |

##### 5.3.2.2.1 Random Evaluator

As shown in Table 5.18 (second row), the first step with the random evaluator took 8.39 days, generating 5,705 individuals. Afterward, a snippet from the pool of individuals is randomly picked. After the first step, this snippet achieves a power consumption of 0.0044 W and 0.045 01 W after the second generation step, with 865 additional generated individuals. The second step ran for 10 generations and took 9 h20 min. The generated individuals have very low power consumption mainly because the program structure is generated purely based on an arbitrary fitness value.

Although it generates individuals quickly during the architectural simulation phase, it does not restrict the solution space. Overall, the best individual from this run is significantly worse than the baseline regarding power consumption.

##### 5.3.2.2.2 Microarchitectural Simulation

For this evaluation, *MicroGP* using the IPC value extracted from *Gem5* simulations as feedback has been used. Table 5.18 (last row) took about 9.6 days, generating 3,670 individuals. The snippet with the highest IPC, 0.64, is obtained from this step.

Then, the second step was executed for another ten generations, which took around 8 h, generating only 851 additional individuals. The highest power consumption after the second step is 0.7142 W. As a comparison, the power consumption of the snippet with the fixed register data is 0.6520 W. The comparison of power consumption with fixed data value and tuned register data shows how the initial register contents impact the overall power consumption.

In summary, as it can be seen from Table 5.18, the two-step generation exploiting the IPC and the on-chip measurements for the register data tuning performs better in three aspects:

- Execution Time: the two-step generation obtains a speed-up of 1.81x compared to the baseline.
- Number of generated individuals: a guided multi-step generation generates fewer individuals to evaluate because it guides the genetic framework (and simultaneously restricts the solutions space).
- Achieved Power consumption: the proposed guided generation performs better regarding final power consumption than the baseline.

In addition, Figure 5.17 shows the average, best, and worst fitness values per generation of the baseline (dashed lines) vs the power consumption for tuning the register data after programs are generated using the architectural simulation maximizing the IPC (solid lines).

The register data tuning from two-step generation starts from the better initial population than the baseline. Additionally, it performs better during evolution by restricting the distance between the worst and the best individual.

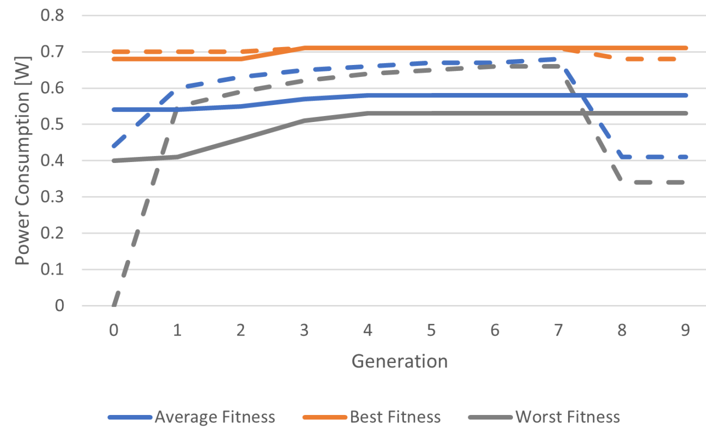



Fig. 5.17: Average, Best, and Worst fitness values per generation of Baseline (dashed line) and register data tuning for the two-step generation based on microarchitectural simulation.

One of the main reasons is the enormous space of solutions to explore from the genetic framework perspective. Therefore, it needs to be guided from some aspects; otherwise, the time to converge to an optimal solution may be infinite. On the other hand, if the solutions space is too big, the genetic framework may apply genetic operators that could affect the fitness (the case of the last two generations for the baseline) and, consequently, increase the execution time for converging to a solution.

#### 5.3.2.3 Final remarks

In conclusion, the study demonstrates the effectiveness of a two-step generation approach that combines architectural simulations and on-chip power measurements for generating stress test programs using genetic algorithms. The proposed approach outperforms the single-step approach regarding execution time, number of generated individuals, and achieved power consumption, highlighting the benefits of guiding the genetic framework with architectural insights.

In addition, the study shows an alternative methodology to grade SLT workload: measuring power consumption.

# Chapter 6

# Conclusions

*Remember to look up at the stars and not down at your feet. Try to make sense of what you see and wonder about what makes the universe exist. Be curious. And however difficult life may seem, there is always something you can do and succeed at.*
*It matters that you don't just give up.*

Stephen Hawking

System-Level Test has become an indispensable component of modern semiconductor testing workflows. As the semiconductor industry continues to advance, SLT methodologies are expected to evolve, addressing current challenges and leveraging emerging technologies. The integration of AI, machine learning, and advanced analytics promises to enhance the efficiency and effectiveness of SLT, further solidifying its role in ensuring the quality and reliability of next-generation integrated circuits.

While developing SLT applications presents numerous challenges, particularly regarding execution time and fault simulation, there are viable strategies to mitigate these issues. By embracing automation and leveraging the inherent regularity of applications, developers can streamline the testing process, reduce manual efforts, and improve the reliability of SLT outcomes.

This dissertation establishes the critical role of System-Level Testing (SLT) in bridging the gaps of traditional structural testing methodologies for modern automotive SoCs. The research highlights that while structural tests provide a robust foundation for fault detection, they are insufficient for addressing faults in system-

level interactions, particularly in safety-critical applications adhering to ISO 26262 standards. SLT emerges as a complementary approach capable of achieving higher fault coverage and reliability. The proposed contributions address critical challenges in SLT implementation:

- Stress Optimization for SLT: integrating functional stress methodologies into the BI phase ensures uniform and effective stress across untested regions, particularly in embedded memories and communication peripherals, enabling the stress for successive test phases.
- Structural test weaknesses: shadowed untested zones can be pinpointed by analyzing structural test fault lists and the netlist. Afterward, the development of SLT workloads uses shadowed untested zone information as a starting point to produce an effective SLT suite.
- Automated Workload Generation: by leveraging graph-based SoC representations and Device Tree Source (DTS) files or genetic framework, the thesis demonstrates scalable and automated workload generation techniques, significantly reducing manual development efforts.
- Grading and Metrics for SLT: the development of connectivity-based grading methodologies facilitates early feedback on SLT workload effectiveness, enabling iterative improvement without exhaustive fault simulation campaigns.

Moreover, as additional secondary contributions, it is worth to mention the design and implementation of a low-cost, FPGA-based modular tester demonstrate the feasibility of executing both structural and functional tests in a cost-effective and scalable manner and the development of a toolchain for BI stress evaluation that could be used as intermediate analysis prior the fault simulation campaigns while developing SLT workloads.

Furthermore, the modular and automated nature of the proposed techniques ensures adaptability to diverse SoC architectures, reducing development time and cost. The use of standardized SoC descriptors (such as the DTS file), the ISA description file, and a high-level programming language makes the proposed methodologies highly portable across different architectures (e.g., RISC-V-based automotive SoCs). This portability is achieved in a cost-effective manner by simply adapting the existing ISA description file, updating the DTS, and aligning the programming language

with the target toolchain. Moreover, the $S^2TL$ ensures that a foundational skeleton is already in place for porting the hardware abstraction layer of the library to a different SoC. This is accomplished by linking the library with the Low-Level Driver provided by the silicon vendor.

Experimental results can now answer the question, "can SLT further improve structural test coverage?". The experimental results underscore the effectiveness of SLT in improving fault coverage, with small, but significant, gains observed in stuck-at and transition-delay fault models by using proposed SLT workloads.

The key finding in using in-field like SLT workload to improve ATPG coverage is crucial to highlight the importance of introducing SLT as an additional test phase for increasing the quality of semiconductor devices. The feasibility of introducing the proposed methodologies into an industrial manufacturing test flow depends on the time and cost budget, as well as the trade-offs between test time and achieved coverage. Nonetheless, different SLT suites can be provided to meet specific industrial needs. Regarding the grading methodologies, they can be seamlessly integrated into the design flow since they rely on a combination of commercial tools and custom-developed tools. These tools and toolchains are easily adaptable to different designs or ISAs by simply providing the necessary input information. This adaptability ensures that the methodologies remain flexible and practical for a wide range of industrial applications.

It is crucial to mention that this PhD dissertation is not the final solution for test escapes from structural tests. New fault models and new behavior may arise with newer technologies. However, it sheds some light on the mysteries of SLT and demonstrates that SLT is not a whim, and it could be an effective solution for increasing the reliability of modern devices.

## 6.1 Beyond Automotive

As semiconductor complexity continues to grow, SLT will remain indispensable in achieving the quality and reliability demanded by next-generation integrated circuits [6, 11, 17, 19]. SLT for automotive applications could be beneficial for devices beyond the automotive industry, such as high-end processors in data center fleets of different CSP [3–5].

The increasing insurgence of small delay defects and other subtle misbehaviors makes the testing work complicated and expensive for manufacturers [107]. Growing attention is being placed on the so-called Silent Data Corruption (SDCs) [108, 9], also known as Silent Data Errors (SDEs), which are of both permanent and transient nature [109].

Focusing on permanent faults, Silent Data Errors (SDEs) are a subset of Defective Parts per Million (DPPM) test escapes. Some defects that could manifest as SDEs are screened during the SLT phase. A common example of SDEs is [4]:

$$Int(1.1^{53}) = 0 \tag{6.1}$$

In this case, the computation returns a wrong value from an application perspective; however, it does not cause the system to hang or generate errors in the application.

Many papers deal with the characterization of SDC impacts at application level [107, 110–112], such as using high-level emulators/architectural models to inject and evaluate system behavior, studying the impact of transient fault on systems reliability [109, 112], and also studying permanent fault effects [113, 114].

Recent studies attribute the nature of SDEs to small delay faults [107, 115–117] which are hard to excite and propagate using scan-oriented test approaches; they can be only be captured by functional test programs [85].

Moreover, functional test programs oriented to detect SDEs share similarities with SLT, such as long execution times, the absence of quality metrics, and the crafting of such test programs by skilled test engineers [85].

Therefore, SLT techniques could look beyond the automotive industry, finding applications in other contexts.

# References


[1] F. Angione, P. Bernardi, N. di Gruttola Giardino, G. Filipponi, C. Bertani, and V. Tancorre, “A system-level test methodology for communication peripherals in system-on-chips,” *IEEE Transactions on Computers*, pp. 1–8, 2024.

[2] F. Angione, P. Bernardi, G. Iaria, C. Bertani, and V. Tancorre, “Automatic generation of system-level test for un-core logic of large automotive soc,” *Submitted to IEEE Transactions on Computers*, pp. 1–2, 2024.

[3] P. H. Hochschild, P. Turner, J. C. Mogul, R. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat, “Cores that don’t count,” in *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS ’21, (New York, NY, USA), p. 9–16, Association for Computing Machinery, 2021.

[4] H. D. Dixit, L. Boyle, G. Vunnam, S. Pendharkar, M. Beadon, and S. Sankar, “Detecting silent data corruptions in the wild,” 2022.

[5] S. Wang, G. Zhang, J. Wei, Y. Wang, J. Wu, and Q. Luo, “Understanding silent data corruptions in a large production cpu population,” in *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP ’23, (New York, NY, USA), p. 216–230, Association for Computing Machinery, 2023.

[6] D. P. Lerner, B. Inkley, S. H. Sahasrabudhe, E. Hansen, L. D. R. Munoz, and A. v. de Ven, “Optimization of tests for managing silicon defects in data centers,” in *2022 IEEE International Test Conference (ITC)*, pp. 578–582, 2022.

[7] S. K. S. Hari, P. Rech, T. Tsai, M. Stephenson, A. Zulfiqar, M. Sullivan, P. Shirvani, P. Racunas, J. Emer, and S. W. Keckler, “Estimating silent data corruption rates using a two-level model,” 2020.

[8] V. Chickermane, N. Mukherjee, A. Meixner, S. Bhatia, S. Gurumurthy, D. P. Lerner, J. Dworak, J. Rajski, and N. Brousard, “Will silent data errors give a new lease on life to semiconductor test?,” in *IEEE 2023 International Test Conference Panel*.

[9] R. S. Chappell (Microsoft), L. Warnes (NVIDIA), and V. Sridharan (AMD), “Open compute project - silent data corruption: Specification overview and metrics.” https://drive.google.com/file/d/1Y8FtMxMXQJ1lco1PeI0q2fJMRB-Z8iHC/view?usp=drive_link. Accessed: 3 March 2024.

[10] S. Biswas and B. Cory, "An industrial study of system-level test," *IEEE Design Test of Computers*, vol. 29, no. 1, pp. 19–27, 2012.

[11] D. Appello et al., "System-level test: State of the art and challenges," in *IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2021.

[12] G. E. Moore, "Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff.," *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 3, pp. 33–35, 2006.

[13] M. Campbell, "Plenary presentations: Keynote: The product complexity and test — how product complexity impacts test industry," in *2010 15th IEEE European Test Symposium*, pp. 9–9, 2010.

[14] D. Tille *et al.*, "Towards an automated flow for implementation ofdedicated lbist scan chains for functional safety," *TUZ*, 2020.

[15] N. Karimi, K. Chakrabarty, P. Gupta, and S. Patil, "Test generation for clock-domain crossing faults in integrated circuits," in *IEEE DATE*, pp. 406–411, 2012.

[16] F. Angione, P. Bernardi, G. Filipponi, M. S. Reorda, D. Appello, V. Tancorre, and R. Ugioli, "An optimized burn-in stress flow targeting interconnections logic to embedded memories in automotive systems-on-chip," in *2022 IEEE European Test Symposium (ETS)*, pp. 1–6, May 2022.

[17] H. H. Chen, "Beyond structural test, the rising need for system-level test," in *International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, 2018.

[18] I. Polian, J. Anders, S. Becker, P. Bernardi, K. Chakrabarty, N. ElHamawy, M. Sauer, A. Singh, M. S. Reorda, and S. Wagner, "Exploring the mysteries of system-level test," in *2020 IEEE 29th Asian Test Symposium (ATS)*, pp. 1–6, 2020.

[19] D. K. R. Tipparthi and K. K. Kumar, "Concurrent system level test (cslt) methodology for complex system-on-chip," in *2014 IEEE 16th Electronics Packaging Technology Conference (EPTC)*, pp. 196–199, 2014.

[20] G. Iaria, F. Angione, P. Bernardi, M. S. Reorda, D. Appello, G. Garozzo, and V. Tancorre, "A novel pattern selection algorithm to reduce the test cost of large automotive systems-on-chip," in *2022 IEEE 23rd Latin American Test Symposium (LATS)*, pp. 1–6, Sep. 2022.

[21] M. Fujita, N. Taguchi, K. Iwata, and A. Mishchenko, "Incremental atpg methods for multiple faults under multiple fault models," in *ISQED*, 2015.

[22] C. Hobeika, C. Thibeault, and J. F. Boland, "Illegal state extraction from register transfer level," in *Proceedings of the 8th IEEE International NEWCAS Conference 2010*, pp. 245–248, 2010.

[23] I. Pomeranz, "Built-in generation of functional broadside tests using a fixed hardware structure," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 1, pp. 124–132, 2013.

[24] M. Bushnell and V. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Springer Publishing Company, Incorporated, 2013.

[25] Fujiwara and Shimono, "On the acceleration of test generation algorithms," *IEEE Transactions on Computers*, 1983.

[26] F. Angione, D. Appello, P. Bernardi, A. Calabrese, S. Quer, M. S. Reorda, V. Tancorre, and R. Ugioli, "A toolchain to quantify burn-in stress effectiveness on large automotive system-on-chips," *IEEE Access*, pp. 1–1, 2023.

[27] W. K. Al-Assadi, M. V. Joshi, and G. M. Chaudhry, "A bist technique for configurable nanofabric arrays," in *2008 IEEE International Workshop on Design and Test of Nano Devices, Circuits and Systems*, pp. 63–66, Sep. 2008.

[28] P. Bernardi, M. Grosso, E. Sanchez, and O. Ballan, "Fault grading of software-based self-test procedures for dependable automotive applications," in *2011 Design, Automation & Test in Europe*, pp. 1–2, March 2011.

[29] F. Corno, G. Cumani, M. Sonza Reorda, and G. Squillero, "Efficient machine-code test-program induction," in *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*, vol. 2, pp. 1486–1491 vol.2, May 2002.

[30] Arm, "Amba ahb protocol specification." https://developer.arm.com/documentation/ihi0033/latest/.

[31] F. Angione, D. Appello, P. Bernardi, C. Bertani, G. Gallo, S. Littardi, G. Pollaccia, W. Ruggeri, M. S. Reorda, V. Tancorre, and R. Ugioli, "A low-cost burn-in tester architecture to supply effective electrical stress," *IEEE Transactions on Computers*, pp. 1–14, 2022.

[32] L.-T. Wang, C.-W. Wu, and X. Wen, *VLSI Test Principles and Architectures: Design for Testability*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[33] "Ieee draft standard test access port and boundary scan architecture," *IEEE P1149.1/D2012.e27, September 2012*, pp. 1–434, 2012.

[34] C. He and Y. Yu, "Wafer level stress: Enabling zero defect quality for automotive microcontrollers without package burn-in," *2020 IEEE International Test Conference (ITC)*, pp. 1–10, 2020.

[35] A. Benso, A. Bosio, S. D. Carlo, G. D. Natale, and P. Prinetto, "Atpg for dynamic burn-in test in full-scan circuits," in *2006 15th Asian Test Symposium*, pp. 75–82, 2006.

[36] A. Birolini, “Reliability engineering theory and practice,” *Springer*, 2017.

[37] P. Reichert, “System Level Test,” *Teradyne*.

[38] D. Armstrong, “Shifting Left = More Wafer Probe Real-World Implications,” *SWTEST*, 2022.

[39] P. Bernardi, M. Restifo, M. S. Reorda, D. Appello, C. Bertani, and D. Petrali, “Applicative system level test introduction to increase confidence on screening quality,” in *2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pp. 1–6, 2020.

[40] H. Amrouch, V. M. van Santen, and J. Henkel, “Interdependencies of degradation effects and their impact on computing,” *IEEE Design & Test*, vol. 34, pp. 59–67, June 2017.

[41] B. Eklow, *On Microprocessor Reliability*. IEEE Computer Society, Computing Now, 1 ed., 2013.

[42] P. Aggarwal, “Cost effective manufacturing test using mission mode tests,” in *2007 IEEE International Test Conference*, pp. 1–8, Oct 2007.

[43] D. Appello, P. Bernardi, G. Giacopelli, A. Motta, A. Pagani, G. Pollaccia, C. Rabbi, M. Restifo, P. Ruberg, E. Sanchez, C. Villa, and F. Venini, “A comprehensive methodology for stress procedures evaluation and comparison for burn-in of automotive soc,” in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pp. 646–649, 2017.

[44] F. X. Che, J.-K. Lin, K. Y. Au, and X. Zhang, “Comprehensive study on reliability of chip-package interaction using cu pillar joint onto low k chip,” in *2014 IEEE 16th Electronics Packaging Technology Conference (EPTC)*, pp. 288–293, Dec 2014.

[45] F. Angione, D. Appello, P. Bernardi, A. Calabrese, L. Cardone, A. Niccoletti, D. Piumatti, S. Quer, V. Tancorre, and R. Ugioli, “An innovative strategy to quickly grade functional test program,” 2022.

[46] H. Amrouch *et al.*, “Intelligent methods for test and reliability,” in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 969–974, March 2022.

[47] P. Bernardi and et al., “Development flow for on-line core self-test of automotive microcontrollers,” *IEEE Transactions on Computers*, vol. 65, pp. 744–754, March 2016.

[48] V. Gangaram, D. Bhan, and J. K. Caldwell, “Functional test selection for high volume manufacturing,” in *Seventh International Workshop on Microprocessor Test and Verification (MTV’06)*, pp. 15–19, 2006.

[49] A. Apostolakis, D. Gizopoulos, M. Psarakis, D. Ravotto, and M. S. Reorda, "Test program generation for communication peripherals in processor-based soc devices," *IEEE Design & Test of Computers*, vol. 26, no. 2, pp. 52–63, 2009.

[50] P. Bernardi, L. Bolzani, A. Manzone, M. Osella, M. Violante, and M. S. Reorda, "Software-based on-line test of communication peripherals in processor-based systems for automotive applications," in *Seventh International Workshop on Microprocessor Test and Verification (MTV'06)*, pp. 3–8, 2006.

[51] M. Grosso, W. J. H. Perez, D. Ravotto, E. Sanchez, M. S. Reorda, and J. V. Medina, "A software-based self-test methodology for system peripherals," in *2010 15th IEEE European Test Symposium*, pp. 195–200, 2010.

[52] F. A. da Silva, R. Cantoro, S. Hamdioui, S. Sartoni, C. Sauer, and M. Sonza Reorda, "A systematic method to generate effective stls for the in-field test of can bus controllers," *Electronics*, vol. 11, no. 16, 2022.

[53] A. Jutman, M. S. Reorda, and H.-J. Wunderlich, "High quality system level test and diagnosis," in *2014 IEEE 23rd Asian Test Symposium*, pp. 298–305, Nov 2014.

[54] F. Almeida et al., "Effective screening of automotive socs by combining burn-in and system level test," in *IEEE International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, 2019.

[55] A. D. Singh, "An adaptive approach to minimize system level tests targeting low voltage dvfs failures," in *2019 IEEE International Test Conference (ITC)*, pp. 1–10, 2019.

[56] P. Bernardi, A. Bosio, G. Di Natale, A. Guerriero, E. Sanchez, and F. Venini, "Improving Stress Quality for SoC Using Faster-than-At-Speed Execution of Functional Programs," in *VLSI-SoC: System-on-Chip in the Nanoscale Era – Design, Verification and Reliability* (T. Hollstein, J. Raik, S. Kostin, A. Tšertov, I. O'Connor, and R. Reis, eds.), vol. AICT-508 of *IFIP Advances in Information and Communication Technology*, (Tallinn, Estonia), pp. 130–151, Springer International Publishing, Sept. 2016.

[57] P. Bernardi, R. Cantoro, L. Gianotto, M. Restifo, E. Sanchez, F. Venini, and D. Appello, "A dma and cache-based stress schema for burn-in of automotive microcontroller," in *2017 18th IEEE Latin American Test Symposium (LATS)*, pp. 1–6, 2017.

[58] B. Jose and J. S. Immanuel, "Design of bist(built-in-self-test)embedded master-slave communication using spi protocol," in *2021 3rd International Conference on Signal Processing and Communication (ICPSC)*, pp. 581–585, 2021.

[59] S. Saha, M. A. Rahman, and A. Thakur, "Design and implementation of a bist embedded high speed rs-422 utilized uart over fpga," in *2013 Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT)*, pp. 1–5, 2013.

[60] D. Lin, S. Eswaran, S. Kumar, E. Rentschler, and S. Mitra, "Quick error detection tests with fast runtimes for effective post-silicon validation and debug," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1168–1173, 2015.

[61] J. Sosnowski, "Testing crossbar switch interconnection networks," in *Proceedings ETC 93 Third European Test Conference*, pp. 540–542, April 1993.

[62] E. Karimi, M.-H. Haghbayan, A.-M. Rahmani, M. Tabandeh, P. Liljeberg, and Z. Navabi, "Accelerated on-chip communication test methodology using a novel high-level fault model," in *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, pp. 283–288, 2015.

[63] E. Karimi, M. H. Haghbayan, A. Maleki, and M. Tabandeh, "Functional fault model definition for bus testing," in *East-West Design & Test Symposium (EWDTS 2013)*, pp. 1–4, 2013.

[64] Y. Li, O. Mutlu, D. S. Gardner, and S. Mitra, "Concurrent autonomous self-test for uncore components in system-on-chips," in *2010 28th VLSI Test Symposium (VTS)*, pp. 232–237, 2010.

[65] D. Schwachhofer, M. Betka, S. Becker, S. Wagner, M. Sauer, and I. Polian, "Automating greybox system-level test generation," in *2023 IEEE European Test Symposium (ETS)*, pp. 1–4, 2023.

[66] D. Schwachhofer, P. Domanski, S. Becker, S. Wagner, M. Sauer, D. Pflüger, and I. Polian, "Large language model-based optimization for system-level test program generation," in *2024 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pp. 1–6, 2024.

[67] A. Paschalis and et al., "Effective software-based self-test strategies for on-line periodic testing of embedded processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, pp. 88–99, Jan 2005.

[68] P. K. Parvathala and et al., "Functional random instruction testing (frits) method for complex devices such as microprocessors," *United States Patent 6948096*, 2005.

[69] D. Piumatti and et al., "An efficient strategy for the development of software test libraries for an automotive microcontroller family," *Microelectronics Reliability vol. 115*, vol. 115, dec 2020.

[70] T. Mak, "Infant mortality–the lesser known reliability issue," in *13th IEEE International On-Line Testing Symposium (IOLTS 2007)*, pp. 122–122, 2007.

[71] D. Appello, C. Bugeja, G. Pollaccia, P. Bernardi, R. Cantoro, M. Restifo, E. Sanchez, and F. Venini, "An optimized test during burn-in for automotive soc," *IEEE Design Test*, vol. 35, no. 3, pp. 46–53, 2018.

[72] D. Schwachhofer, F. Angione, S. Becker, S. Wagner, M. Sauer, P. Bernardi, and I. Polian, "Optimizing system-level test program generation via genetic programming," in *2024 IEEE European Test Symposium (ETS)*, pp. 1–4, 2024.

[73] F. Angione, P. Bernardi, A. Calabrese, L. Cardone, S. Quer, C. Bertani, and V. Tancorre, "A novel indirect methodology based on execution traces for grading functional test programs," *Submitted to IEEE Transactions on Computers*, pp. 1–2, 2024.

[74] P. Bernardi et al., "On-line software-based self-test of the address calculation unit in risc processors," in *IEEE European Test Symposium*, 2012.

[75] J. Hennessy et al., "Computer architecture: a quantitative approach," 1990.

[76] P. Bernardi *et al.*, "On the in-field testing of spare modules in automotive microprocessors," in *IEEE VLSI-SoC*, 2017.

[77] "Devicetree specification - release v0.3." https://github.com/devicetree-org/devicetree-specification/releases/download/v0.3/devicetree-specification-v0.3.pdf. Accessed: 20 October 2023.

[78] P. Bernardi, M. Bonazza, E. Sanchez, M. Sonza Reorda, and O. Ballan, "On-line functionally untestable fault identification in embedded processor cores," in *IEEE DATE*, pp. 1462–1467, 2013.

[79] E. Sánchez *et al.*, "New evolutionary techniques for test-program generation for complex microprocessor cores," in *GECCO*, June 2005.

[80] K. DeVogeleer *et al.*, "Modeling the temperature bias of power consumption for nanometer-scale cpus in application processors," in *SAMOS XIV*, 2014.

[81] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and Éder F. Zulian, "The gem5 simulator: Version 20.0+," 2020.

[82] Z. Hadjilambrou *et al.*, "Gest: An automatic framework for generating CPU stress-tests," in *IEEE ISPASS*, 2019.

[83] D. Appello, P. Bernardi, A. Calabrese, G. Pollaccia, S. Quer, V. Tancorre, and R. Ugioli, "Parallel multithread analysis of extremely large simulation traces," *IEEE Access*, vol. 10, pp. 56440–56457, 2022.

[84] Intel, "Opendcdiag." https://github.com/opendcdiag/opendcdiag. Accessed: 26 February 2024.

[85] T. Macieira, S. Gurumurthy, S. Gurumurthi, A. Haggag, G. Papadimitriou, and D. Gizopoulos, "Silent data corruptions in computing: Understand and quantify," in *2024 IEEE 30th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pp. 1–7, 2024.

[86] F. Angione *et al.*, "Test, Reliability and Functional Safety trends for Automotive System-on-Chip," European Test Symposium, 2022.

[87] D. Appello, P. Bernardi, A. Calabrese, S. Littardi, G. Pollaccia, S. Quer, V. Tancorre, and R. Ugioli, "Accelerated analysis of simulation dumps through parallelization on multicore architectures," in *2021 24th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pp. 69–74, 2021.

[88] M. Hassan *et al.*, "Early soc security validation by vp-based static information flow analysis," in *IEEE/ACM ICCAD*, 2017.

[89] R. Drechlser *et al.*, "Ensuring correctness of next generation devices: From reconfigurable to self-learning systems," in *IEEE ATS*, 2019.

[90] W. Hu *et al.*, "An overview of hardware security and trust: Threats, countermeasures, and design tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[91] K. M. Alatoun *et al.*, "Efficient methods for soc trust validation using information flow verification," in *IEEE ICCD*, 2021.

[92] F. Angione, P. Bernardi, N. di Gruttola Giardino, D. Appello, C. Bertani, and V. Tancorre, "A guided debugger-based fault injection methodology for assessing functional test programs," 2023.

[93] G. Cabodi, P. Camurati, and S. Quer, "Symbolic Exploration of Large Circuits with Enhanced Forward/Backward Traversals," in *Proc. IEEE EURO–DAC'94*, (Grenoble, France), pp. 22–27, IEEE Computer Society, sep 1994.

[94] G. Cabodi, S. Nocco, and S. Quer, "Mixing Forward and Backward Traversals in Guided-Prioritized BDD-Based Verification," in *Proc. Computer Aided Verification* (E. M. Clarke and R. P. Kurshan, eds.), vol. 2102 of *Lecture Notes in Computer Science*, (Copenhagen, Denmark), pp. 471–484, Springer-Verlag, jul 2002.

[95] E. C. Michael J. Eager, "Introduction to the dwarf debugging format." https://dwarfstd.org/doc/Debugging%20using%20DWARF-2012.pdf. Accessed: 18 July 2024.

[96] A. Ressel and R. Schmidt-Vollus, "Reverse engineering in process automation," in *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA )*, pp. 1–4, 2021.

[97] A. K. Dwivedi, S. K. Rath, S. M. Satapathy, L. S. Chakravarthy, and P. K. S. Rao, "Applying reverse engineering techniques to analyze design patterns in source code," in *2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pp. 1398–1404, 2018.

[98] N. di Gruttola Giardino, F. Angione, P. Bernardi, T. Foscale, C. Bertani, and V. Tancorre, "A flexible fpga-based test equipment for enabling out-of-production manufacturing test flow of digital systems," in *2024 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pp. 1–6, 2024.

[99] Xilinx, "Zynq ultrascale+ mpsoc zcu104 evaluation kit." https://www.xilinx.com/products/boards-and-kits/zcu104.html/.

[100] "Ieee standard test interface language (stil) for digital test vector data," *IEEE Std 1450-1999*, pp. 1–140, Sep. 1999.

[101] Xilinx, "Pynq." https://github.com/xilinx/pynq.

[102] P. Bernardi, M. Grosso, E. Sanchez, and O. Ballan, "Fault grading of software-based self-test procedures for dependable automotive applications," in *2011 Design, Automation & Test in Europe*, pp. 1–2, March 2011.

[103] J. J. Labrosse, *UC/OS-III, The Real-Time Kernel, or a High Performance, Scalable, ROMable, Preemptive, Multitasking Kernel for Microprocessors, Microcontrollers amp; DSPs*. Weston, FL, USA: Micrium Press, 2009.

[104] A. Hanneman, J. Gava, V. Bandeira, R. Garibotti, R. Reis, and L. Ost, "Debate-fi: A debugger-based fault injector infrastructure for iot soft error reliability assessment," in *2023 IEEE 9th World Forum on Internet of Things (WF-IoT)*, pp. 1–6, 2023.

[105] G. Papadimitriou and D. Gizopoulos, "Avgi: Microarchitecture-driven, fast and accurate vulnerability assessment," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 935–948, 2023.

[106] J. Zhao *et al.*, "SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine," *Fourth Workshop on Computer Architecture Research with RISC-V*, May 2020.

[107] A. Singh, S. Chakravarty, G. Papadimitriou, and D. Gizopoulos, "Silent data errors: Sources, detection, and modeling," in *2023 IEEE 41st VLSI Test Symposium (VTS)*, pp. 1–12, 2023.

[108] B. Parthasarathy (Google), A. Huffman (Google), H. Dattatraya Dixit (Meta Platforms Inc), T. Chakravarthy (Meta Platforms Inc), R. Chappell (Microsoft), V. Sridharan (AMD), S. Gurumurthy (AMD), T. Macieira (Intel), R. Jeyapaul (ARM), L. Minwell (ARM), and L. Warnes (NVIDIA), "Open compute project - server component resilience." https://www.opencompute.org/contributions?contributions%5Bquery%5D=resilience. Accessed: 3 March 2024.

[109] G. Papadimitriou and D. Gizopoulos, “Silent data corruptions: Microarchitectural perspectives,” *IEEE Transactions on Computers*, vol. 72, no. 11, pp. 3072–3085, 2023.

[110] P. Omland, A. Netti, Y. Peng, A. Baldovin, M. Paulitsch, G. Espinosa, J. Parra, G. Hinz, and A. Knoll, “Hpc hardware design reliability benchmarking with hdfit,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 3, pp. 995–1006, 2023.

[111] A. Netti, Y. Peng, P. Omland, M. Paulitsch, J. Parra, G. Espinosa, U. Agarwal, A. Chan, and K. Pattabiraman, “Mixed precision support in hpc applications: What about reliability?,” *Journal of Parallel and Distributed Computing*, vol. 181, p. 104746, 2023.

[112] G. Papadimitriou, D. Gizopoulos, H. D. Dixit, and S. Sankar, “Silent data corruptions: The stealthy saboteurs of digital integrity,” in *2023 IEEE 29th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pp. 1–7, 2023.

[113] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, “Understanding the propagation of hard errors to software and implications for resilient system design,” *SIGARCH Comput. Archit. News*, vol. 36, p. 265–276, mar 2008.

[114] D. Gizopoulos, G. Papadimitriou, and O. Chatzopoulos, “Estimating the failures and silent errors rates of cpus across isas and microarchitectures,” in *2023 IEEE International Test Conference (ITC)*, pp. 377–382, 2023.

[115] V. Sridharan, “Addressing emerging fault modes with testing and reliability,” in *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems - Keynote*, 2024.

[116] M. Shamsa and D. Lerner, “Defect mechanisms responsible for silent data errors,” in *2024 IEEE International Reliability Physics Symposium (IRPS)*, pp. 1–5, 2024.

[117] P. W. Deutsch, V. Sridharan, V. Q. Ulitzsch, J. S. Emer, S. Gurumurthi, and M. Yan, “Delayavf: Calculating architectural vulnerability factors for delay faults,” 2024.

[118] M. Zakaria, Z. Kassim, M.-L. Ooi, and S. Demidenko, “Reducing burn-in time through high-voltage stress test and weibull statistical analysis,” *IEEE Design Test of Computers*, vol. 23, no. 2, pp. 88–98, 2006.

[119] X. Guo, W. Burleson, and M. Stan, “Modeling and experimental demonstration of accelerated self-healing techniques,” in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2014.

[120] W. Ruggeri, P. Bernardi, S. Littardi, M. S. Reorda, D. Appello, C. Bertani, G. Pollaccia, V. Tancorre, and R. Ugioli, "Innovative methods for burn-in related stress metrics computation," in *2021 16th International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, pp. 1–6, 2021.

[121] M. L. Bailey, J. V. Briner, and R. D. Chamberlain, "Parallel logic simulation of vlsi systems," *ACM Comput. Surv.*, vol. 26, p. 255–294, sep 1994.

[122] "Sqlite." https://sqlite.org, 2022. Online; accessed 21 November 2022.

[123] M. Hahsler, M. Piekenbrock, and D. Doran, "dbscan: Fast density-based clustering with R," *Journal of Statistical Software*, vol. 91, no. 1, pp. 1–30, 2019.

[124] "Sdl libraries." https://www.libsdl.org, 2022. Online; accessed 21 November 2022.

[125] "Openmp." https://www.openmp.org, 2022. Online; accessed 21 November 2022.

[126] F. F. d. Santos, J. E. R. Condia, L. Carro, M. S. Reorda, and P. Rech, "Revealing gpus vulnerabilities by combining register-transfer and software-level fault injection," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 292–304, 2021.

[127] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Sonza Reorda, "Microprocessor software-based self-testing," *IEEE Design & Test of Computers*, vol. 27, no. 3, pp. 4–19, 2010.

[128] S. Malik *et al.*, "Specification and modeling for systems-on-chip security verification," in *Proceedings of Design Automation Conference*, 2016.

[129] L. Ciganda, F. Abate, P. Bernardi, M. Bruno, and M. Sonza Reorda, "An enhanced fpga-based low-cost tester platform exploiting effective test data compression for socs," in *2009 12th International Symposium on Design and Diagnostics of Electronic Circuits Systems*, pp. 258–263, April 2009.

[130] A. Patel, V. Gosain, R. S. Mohal, and J. Kumar, "Fpga based low-cost portable tester with on-board supplies," in *2020 4th International Conference on Electronics, Communication and Aerospace Technology (ICECA)*, pp. 301–306, Nov 2020.

[131] L. Mostardini, L. Bacciarelli, L. Fanucci, L. Bertini, M. Tonarelli, A. Giambastiani, and M. De Marinis, "Fpga-based low-cost system for automatic tests on digital circuits," in *2007 14th IEEE International Conference on Electronics, Circuits and Systems*, pp. 911–914, Dec 2007.

[132] B. Rabakavi and S. Siddamal, "Design of high speed, reconfigurable multiple ics tester using fpga platform," in *2018 International Conference on Electrical, Electronics, Communication, Computer, and Optimization Techniques (ICEECCOT)*, pp. 909–914, 2018.

[133] T. Lyons, G. Conner, J. Aslanian, and S. Sullivan, "The implementation and application of a protocol aware architecture," in *2013 IEEE International Test Conference (ITC)*, pp. 1–10, 2013.

[134] A. A. Bayrakci, "Elate: Embedded low cost automatic test equipment for fpga based testing of digital circuits," in *2017 10th International Conference on Electrical and Electronics Engineering (ELECO)*, pp. 1281–1285, 2017.

[135] D. de Carvalho, B. Sanches, M. De Carvalho, and W. Van Noije, "A flexible stand-alone fpga-based ate for asic manufacturing tests," in *2018 IEEE 19th Latin-American Test Symposium (LATS)*, pp. 1–6, 2018.

[136] Y. Fan and Z. Zilic, "Ber testing of communication interfaces," *IEEE Transactions on Instrumentation and Measurement*, vol. 57, no. 5, pp. 897–906, 2008.

[137] L. Manual, "mmap." https://man7.org/linux/man-pages/man2/mmap.2.html.

# Appendix A

# The toolchain for evaluating Burn-In related stress metrics

## A.1 State-of-the-art tools for BI metrics

Overexertion must stimulate as many device nodes as possible [118, 43, 119], and evaluating BI metrics is crucial for grading the quality of stress patterns. Figure A.1 illustrates two possible high-level workflows for evaluating BI stress metrics: An ATPG and a VCD (Value Change Dump) based approach.

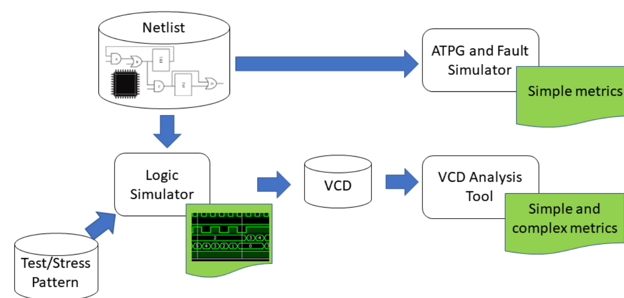



Fig. A.1: Possible evaluation flows for stress metrics.

ATPG-based methods (the upper path of Figure A.1) focus on toggle coverage metrics based on the fault simulation of the netlist. Unfortunately, they cannot focus on more specific coverage metrics for the BI phase, as stated in Table A.1. Moreover, since a single core may execute a single fault simulation, ATPG-based methods may require prohibitive computational times for large devices. VCD-based methods

(the lower path of Figure A.1) rely on the standard VCD format to evaluate the BI metrics. They often run ad-hoc tools in a post-processing phase without the necessity of performing fault simulation [83].

| Tool | Metrics | | | Parallelization | Superimpose capability | Coverage-loss Troubleshoot |
|---|---|---|---|---|---|---|
| | Single-point | Multi-point | Layout-aware | | | |
| TestMax *Synopsys* | Toggle coverage | N.A. | N.A. | Process based | Yes | Per module/gate coverage |
| Tessent *Siemens* | Toggle coverage | N.A. | N.A. | Process/thread based | Yes | Per module/gate coverage |
| Modus *Cadence* | Toggle coverage | N.A. | N.A. | Process/thread based | Yes | Per module/gate coverage |
| Proposed toolchain | Toggle Coverage Toggle Activity + Avg. | Neighborhood static stress | Yes | Thread based | Yes | Per module/gate, layout heatmap |

Table A.1: The table compares our toolchain with state-of-the-art EDA tools. The Comparison is made only from the analysis perspective of stress metrics for BI.

Table A.1 compares the proposed toolchain with some main state-of-the-art EDA tools. The table reports the capabilities of the different tools in terms of the following: adopted metrics (single point, multi-point, layout-aware), the use of parallel techniques to improve time efficiency (notice that process-based techniques are more expensive than thread-based parallelization), the capability to superimpose different stress patterns, and the troubleshooting capability for coverage-loss (coarse or fine coverage, per module or gate coverage, or by plotting the stress over a layout heatmap). Overall, it could drastically reduce development time and improve the quality of the stress patterns.

## A.2 The proposed Toolchain

As the complexity of automotive devices is dramatically increasing, so are the requirements on memory and CPU time from tools to compute metrics for grading stress patterns. Therefore, a common trade-off is to relax the constraints of the final coverage and relief of the tools from time-consuming analysis and simulations.

Moreover, the increasing complexity of automotive devices is also impacting the stress metrics, requiring more advanced metrics [120]. More in detail, the distribution of gates among the SoC floorplan could play a crucial role in distributing uniformly the stress, as well as the mutual interaction between neighbor gates. The latter considerations are not handled by commercial tools in the metrics, as presented in Table A.1.

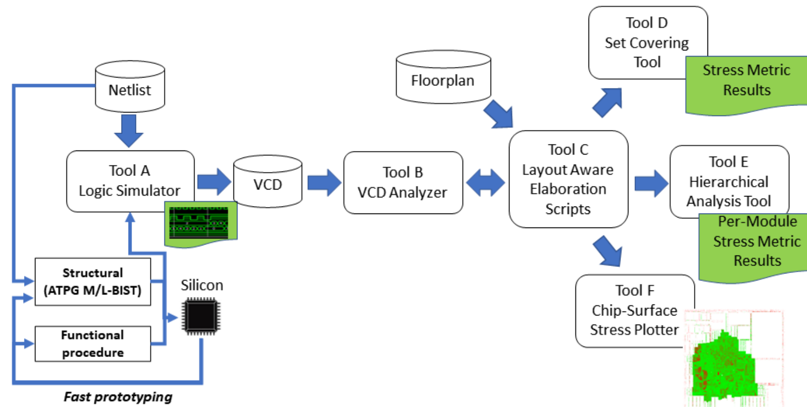



Fig. A.2: The picture represents a high-level view of the proposed toolchain. The stress pattern, either structural or functional, is validated on the silicon implementation of the SoC. Afterwards, a logic simulation based on the given stress pattern is performed to provide a VCD file. This file is then analyzed to provide a single or multi-point stress coverage for the stress pattern. The Layout-Aware elaboration links the stress pattern to the SoC layout by weighting the stress metrics with the gate density. There are several options for the final step, from plotting the stress over a layout heatmap to superimposing different stress patterns or presenting stress coverage metrics for each module.

The aim of the proposed toolchain, represented in Figure A.2, is to abate the computing cost for handling the increased complexity of automotive devices by exploiting thread-based parallel techniques instead of process-based parallel techniques and to overcome the limitation of current EDA tools in terms of stress metrics analyzed per pattern.

Moreover, it provides test engineers with a visualization of the stress over the layout of the SoC, as well as capabilities to troubleshoot coverage loss by providing fine and coarse stress coverage per gate or module.

As described in Figure A.2, the toolchain analyzes a VCD file produced by a commercial logic simulator from a functional test program or a structural pattern. In general, test patterns can be developed on the existing silicon version of the device to reduce the prototyping time [86]. Therefore, after the VCD file generation, the toolchain performs the following steps:

- Tool B: VCD analysis and computation of stress metrics.

- Tool C: Layout-aware elaboration for connecting the VCD analysis to the physical device.
- Tool D: Superimposition of the stress results for different stress patterns to provide incremental coverage.
- Tool E: Hierarchical netlist analysis for computing per-module/gate stress metrics.
- Tool F: Visualization of the stress with a heatmap created from the physical layout of the device.

An automated flow for overcoming the limitation of current commercial EDA tools is the major progress concerning the state of the art that this article is proposing. The concrete benefit is a conjunction of human time cost saving derived from the quick execution times achieved by parallel computing and the clear indications that the results return, either in percentages or visually. In the following sections, we describe every tool in detail from both the theoretical and implementation point of view.

### A.2.1 Tool A: The Logic Simulator

During the BI phase, both structural and functional tests are executed. Consequently, the netlist-based logic simulation phase of the SoC is based on a commercial tool capable of generating a VCD file for different stress approaches and then analyzed by the proposed toolchain.

In the testing, verification, and validation steps, the logic simulation phase is one of the most time-consuming phases. Although, during the years, attempts to parallelize logic simulation have been published [121]. These strategies are based on netlist analysis, which would make the approach unfeasible today due to the rising complexity of modern SoC because they are based on time-consuming formal methods. Therefore, a logic simulation is commonly a single-thread process reproducing the device behavior correctly. Occasionally, a second thread can be used to write the VCD file, providing a slight speedup.

Functional stress programs are developed as small programs in terms of executed instructions because, in this case, the simulated netlist needs to behave exactly as

the actual implementation of the device, e.g., the power-up operations. Under those circumstances, a logic simulation based on functional patterns cannot be reduced in terms of execution time. Consequently, the functional programs are based on a defined sequence of instructions to be executed for reaching a specific state, and they cannot be further boosted except by changing the hardware on which the logic simulator runs. Nonetheless, a slight speedup can be achieved by preloading the memories before the actual logic simulation starts.

Meanwhile, for logic simulations based on structural patterns, the most straightforward solution is to simulate the whole shifting phase of the test vectors inside the scan chain exhaustively to evaluate the effectiveness of a scan pattern in terms of stressing capabilities. This method can replicate what happens at the hardware level; it could be defined as an "exhaustive" approach because it measures every activity produced by the device. Exhaustive methods can succeed in reproducing the device behavior in a reasonable time if part of the device is stimulated, making a limited number of modules work simultaneously. Conversely, when a scan chain is used, exhaustive methods are not recommended. The shift phase can require the simulation of millions of clock cycles while simulating the entire SoC. Such a combination of duration and activation abilities leads to a very long simulation time (estimated in months for the case study described below).

To overcome problems related to scan-chain-based stress and to find a compromise between computational effort and accuracy during the simulation phase, ATPG engines use a "deductive" approach. It consists in loading the corresponding value in each flip-flop of the scan chain in parallel. Such a parallel load approach is based on the following simplified assumption: given a set of test patterns to be applied to the SoC, only the final configuration of the scan chain after the entire shift of every single pattern is considered.

Theoretically, it is true that if applying two consecutive final configurations to the combinational part cause transitions, then the same transitions will show up during the shift operations. It means that:

$$t(deductive) \subseteq t(exhaustive) \tag{A.1}$$

Where t stands for transitions caused by the method within round brackets. This approach is called "deductive" as the recorded transitions constitute a subset of the transitions that take place when applying the exhaustive approach, and a lower

bound on the possible transition coverage is deduced. When we simulate two consecutive final configurations, as in the deductive approach, if a gate toggles, we can demonstrate that this gate is guaranteed to toggle in the exhaustive approach. Hence, the gate is guaranteed to toggle. On the other hand, if the gate does not toggle according to the deductive approach, we cannot guarantee that it does not toggle. In conclusion, the results obtained through the deductive approach are approximated but conservative. More in detail, the deductive approach is implemented as an additional simulated set of *System Verilog* tasks. *System Verilog* tasks load a given structural stress pattern in all the scan-chain registers in parallel.

The article [120] illustrates in more detail how the deductive approach works. Even though the deductive approach provides an approximation, its use is valuable not only for its efficiency but also because it is conservative and can guide the test engineer to develop good patterns, leaving the shift phase out of consideration. It is also appropriate for a high switching frequency, driven by a PLL, during the apply phase but less significant during the low switching frequency of the shift phase.

### A.2.2 Tool B: The VCD File Analyzer

The VCD File analyzer is a crucial tool within the toolchain, and it is based on the parallel analysis of typical data format [83].

#### A.2.2.1 The VCD File Format

A general Value Change Dump (VCD) file produced by the logic simulation is divided into three main sections, as depicted in Figure A.3:

- **Signal header**. Each gate or bus is located within a hierarchical structure, and an ID identifies it; the main difference is that gates hold one logic value, while buses hold many, hence the need for a *sub-ID* to identify each of those values. We identify as signals both gates and buses. This section also provides the initial value for all signals. Every signal is guaranteed to appear in it, together with the initial time of the recorded simulation.
- **Signal change dump**. Each change list is introduced by the current time of the simulation; the list contains only the signals that changed state at the

previously declared time. The signal initialization and the change lists share the same format, with the difference that the first is ensured to contain all the signals previously declared. The list goes on until the end of the file.



Fig. A.3: The VCD file format.

#### A.2.2.2 The VCD File Analyzer

The VCD tends to be very large (typically between hundreds of GBytes and few TBytes) due to the high number of changes. Because of this, the signal change dump analysis is the most critical in terms of computation time. As described in Section A.2, we consider three types of analysis: the full and statistical single-point stress metrics, and the multiple-point stress metric. To speed up the process, we implement a parallel pipeline consisting of 5 stages, each managed by one or more concurrent threads, as shown in Figure A.4:

- The **Discovery** stage divides the file into logical sections, dividing the file for the next stage.
- The **Reading** stage is in charge of reading the logical sections and loading them from the disk to the main memory.

- The **Parsing** stage uses the Reading stage results to transform the text data into logical data; this stage may also update the signal list in the main memory.
- The **Elaboration** stage performs calculations based on the data provided by the Parsing stage, updating the signal list in the main memory.
- The **Write-Back** stage stores the analysis results on disk.

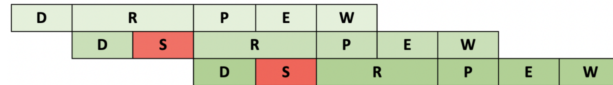



Fig. A.4: The different stages of the pipeline: [D]iscovery, [R]eading, [P]arsing, [E]laboration, [W]rite-Back. In a realistic scenario, some [S]talls might be present.

In Figure A.4, we have a visual representation of the pipeline stages. In a pipeline, the elapsed time strongly depends on the slowest stage. As the pipeline may need to be more perfectly balanced, stalls may appear when waiting for stages that take longer than others. For example, reading from a disk is the slowest operation; because of this, the Parsing, Elaboration, and Write-Back stages are delayed. In this case, we put more effort into reducing the reading stage overhead as much as possible.

Using Equation A.2 we can describe the elapsed time $T_{proc}$ based on the knowledge of the slowest stage $T_{slowest}$ and the number of times $N$ that we make a pipeline call; the other stages, as they are executed in parallel with each other (and with the slowest stage), so the elapsed time for each stage $T_s$ only appears once.

$$T_{proc} = T_{slowest} \cdot N + \sum_{s}^{\{all\ stages \setminus slowest\}} T_s \tag{A.2}$$

Communication between each stage occurs through thread-safe queues with limited, predefined size; as a stage fills in a queue with its results, the next one empties it until the end of the computation occurs. Please notice that this is a general idea, during implementations for each type of analysis slightly differs. In particular:

- In the **full single-point metric** (toggle analysis), the Parsing and Elaboration stages are merged, as the overhead of data passing would be much higher than the duration of each operation. Moreover, since the Write-Back stage is only needed at the end of the elaboration, it starts after every other stage is finished.

- In the **statistical single-point stress metric**, we have all the stages in the pipeline. Since the results may be huge on disk, the Write-Back stage works parallel with the others. We have two versions for the output file: One similar to the standard file, with the difference that we only save the toggle information, and one using the widely used SQLite [122].

- In the **multiple-point stress metrics**, the Parse and the Elaboration stage may read and write from the same queue. We designed a mechanism that needs subsequent circuit states to be read. Parallelism in the Parsing stage may lead to an issue where two elements in the queue may not be subsequent. We solved this issue by providing a unique increasing identifier for each element in a queue; in this case, the queue is a priority queue, where the element identifier is the key, and elements are removed in ascending order; if we extract two elements with no consecutive identifiers, those elements are put back in the queue.

Low-level optimizations are also a critical factor in speeding up the process:

- We created a low-level buffered reader that performs file reading and it provides essential tools for parsing. A buffered reader reduces the number of file readings by loading file chunks of approximately 8 MBytes; the choice of chunk size is based on experimental data on our system.

- We minimize the number of dynamic allocations as much as possible, resorting to fixed-size structures allocated before the signal change analysis. Queues are also pre-allocated using their maximum size available.

- We use an $O(1)$ access table for accessing the signals through the ID and sub-ID keys, as signals are frequently accessed and updated by different threads.

- We use data structures optimized for the needed analysis; this comes at the expense of their flexibility. However, as future devices may be larger, we want to reduce memory usage and computation time as much as possible.

Considering the large spectrum of manipulation that the stress information should endure, and as we would like all tools to be compatible, we designed an intermediate human-readable text-based file format to pass information among the various components of the proposed toolchain. This intermediate formatted source allows the

toolchain to be improved with less effort, possibly introducing intermediate steps if needed, only sacrificing a small amount of disk memory. Tools are also built to read and write files that must be coherent with the format specified; we divide the values by spaces, while each record is on a separate line, meanwhile comments starts with "#"; comments usually include traceability information such as the metric coverage.

### A.2.3 Tool C: Layout-Aware Elaboration Scripts

In this phase of the proposed toolchain, a set of scripts is included to provide layout-awareness capabilities. This toolchain elaboration step is exploited to link the stress evaluation with the actual SoC physical characteristics.

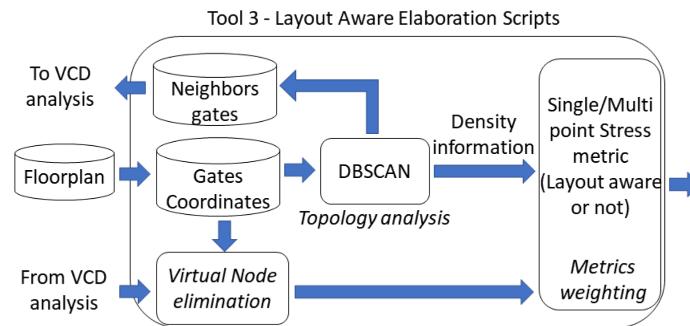



Fig. A.5: Layout-Aware elaboration substeps.

The substeps are summarized in Figure A.5, and three main components can be individuated:

- Virtual node elimination: it eliminates virtual nodes and signals that do not have a physical implementation.
- Topology analysis: it analyzes the floorplan of a given SoC to generate helpful information on the neighbor gates used for multi-point stress metric analysis in the VCD file analyzer.
- Metrics Weighting: the stress metrics can be weighted using a bi-dimensional density by exploiting layout information.

In the following subsection, every component of the Layout-aware elaboration scripts is described in detail.

#### A.2.3.1 Virtual Node Elimination

A significant issue in the stress coverage evaluation process comes from the pure VCD analysis of the simulation dump. As a matter of fact, by looking at the VCD header part, the number of signals saved in the dump includes many replications. As shown in Figure A.6, a circuit connection traversing hierarchies is memorized several times in the VCD.

More in detail, the path to and from the not gate ports includes physical circuits points “a” to “d” and “f” to “i”, but the VCD dump also includes points “b”, “c”, “g”, and “f”, which are not corresponding to any real circuit point but are inherited as simulation artifacts. Including extra points leads to longer computations and affects the stress metric value because an unexcited gate may reflect in many VCD signals, thus polluting the final stress metric value.

In order to eliminate VCD over information and not affect stress metrics, a tool is used to filter useless VCD signals. Such a method is similar to a collapsing strategy, but it is not based on the netlist analysis.

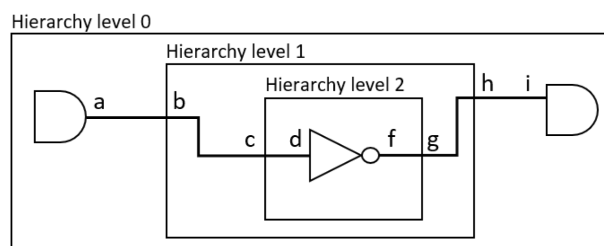



Fig. A.6: An example of the filtering method result.

The signal pruning is obtained by matching the VCD information with the list of real SoC gates, i.e., gates with a physical implementation in the layout. This operation is based on two steps:

1. The coherent loading of a list of gates extracted from the layout.
2. The linear search of VCD signals in the layout gates.

An example of the operation performed in this step can be seen in Algorithm 14.

In terms of complexity, the method leads to an overall $O(n)$ algorithmic complexity.

**Algorithm 14** Virtual node elimination pseudo code.

**Input:** VCD file.

**Require:** Layout information.

```
1: Read and save physical gates into a hash table.
2: for signal in VCD do
3:     if signal in the hash table then
4:         Preserve the signal.
5:     end if
6: end for
```

#### A.2.3.2 Topology Analysis

When dealing with a complex System-on-Chip, it is fundamental first to understand its topology, i.e., how gates are physically placed across the layout to gather valuable insights that can help understand the meaning of the computed stress metrics and devise tests to cover all the parts of the SoC adequately and uniformly.

Modern SoCs do not show a uniform distribution of gates on the layout front-end [120]. Traditional stress metrics are usually "gate-based", i.e., they consider the behavior of a gate or a set of gates regardless of how the SoC is structured; they are also "unweighted", i.e., they consider each gate to yield the same contribution to the metric. The stress per unit of the area varies across the layout, and it may lead to different aging scenarios depending on the density of gates in a given area. Therefore, knowledge of the device topology is crucial to assessing the quality and uniformity of the stress.

For the reasons above, we provide layout-aware elaboration steps to enhance the computed stress metrics.

More in detail, the purposes of crossing information extracted from the VCD analysis with layout data permits to:

1. Introduce gates aggregation when measuring multi-point stress metrics.
2. Reach a high level of accuracy by weighting the stress measurement according to the SoC density.
3. Generate SoC stress heatmap plot by providing information about gate coordinates.

The core aspect of the topology analysis is based on a heuristic method capable of generating density information starting from the layout front-end.

It is essential to mention that exhaustive methods exist for computing the bidimensional density. However, when we use exhaustive methods, the computing time grows with the dimension of gates, and it explodes for larger SoC. For the sake of this work, to overcome computing time limitations, Machine Learning approaches are used. In particular, the *Density-based spatial clustering of applications with noise* (DBSCAN) [123] approach is used. The idea of DBSCAN is to generate a set of clustering, i.e., grouping data in the same group with some similarities.

DBSCAN is a density-based clustering algorithm. Given a set of points, it groups closely packed points (points with many nearby neighbors), highlighting outliers in the low-density region.

With a fixed inter-gate distance, all the logic gates of the SoC can be organized into tuples of neighbors for performing multiple-point analysis. In the toolchain flow, a gate pair located at a distance smaller than a selected threshold on the layout is considered of interest and extracted to feed the VCD analysis tools for the multi-point analysis.

In particular, the computation time costs to extract a couple of gates close enough to each other are traded off with accuracy; a classification method that implements clustering is proposed to abate the timing costs while guaranteeing sufficient accuracy in the selection.

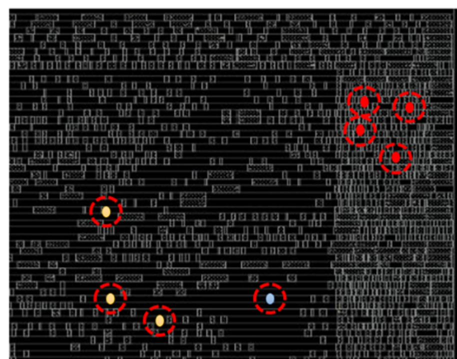


Fig. A.7: An example of the clustering method (DBSCAN) on layout front-end.

Figure A.7 illustrates with an example how the algorithm works on a generic front-end layout: The classification process divides SoC gates into core points

(in red), border points (in yellow), and noise points (in blue), and it analyzes the neighborhood within a fixed inter-gate distance (red dashed circle). Once the clusters are identified over the SoC surface, the list of couples (or tuples) can be performed in a little computation time. Furthermore, the method is helpful for successive steps of the flow, particularly for weighting the stress activity based on the SoC density, which can vary from one region to another in the SoC layout.

#### A.2.3.3 Metrics Weighting

As stated in the previous subsection regarding the topology analysis, the stress per unit of area differs across the layout in modern SoC. Depending on the density of gates in a given area, it may lead to different aging scenarios. Therefore, knowledge of the SoC topology is crucial to assessing the quality and uniformity of the stress and enhancing the stress metric with density awareness as proposed in [120]. This substep of the toolchain provides formulas and considerations presented in [120].

Consequently, a layout-aware stress metric is provided for the single and multi-point stress metric to weigh the stress over denser areas of the SoC instead of considering all the gates equal among them.

### A.2.4 Tool D: The Set Covering Tool

During the BI, different stress approaches of various natures can be used. For example, a BI phase can run scan-based patterns, logic or memory BISTs procedures, and functional programs. Distinguishing which coverage is granted by which method (i.e., which pattern covers a specific section of the SoC) is extremely important, as it allows an understanding of which approaches provide more remarkable improvements.

Therefore, the proposed toolchain includes a tool for implementing a set covering analysis. This tool receives the stress results collected by adopting patterns of different natures, generating all possible set interactions, and providing insights about the stress percentage for each stress approach.

In particular, the set tool provides the following:

- A confusion matrix of toggle coverages among different stress approaches.

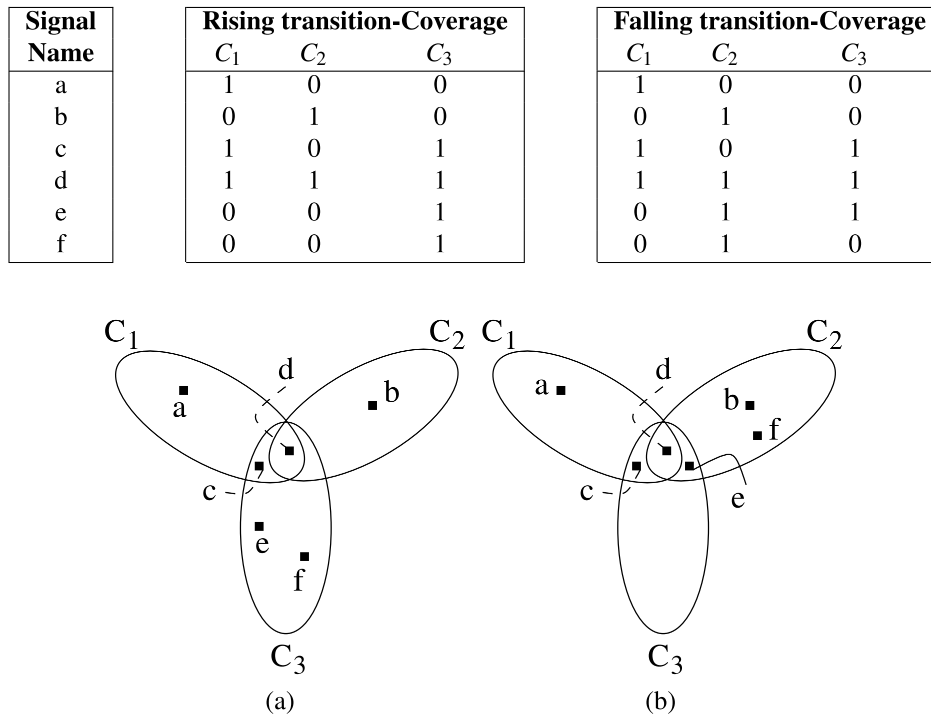
- The list of unique toggle coverages for each stress approach.
- The number of times that a signal toggles.
- The possibility to merge different results, using the different coverages as a subsequent superimposition of stress approaches.
- The possibility to identify raising or falling transitions of a given gate for each stress approach.

More in detail, for inputs and outputs of gates, we store all the raising and falling signal transitions in the table.

As signal names are unique and immutable and tend to be referenced by many files, we use a technique called "string pooling" or "string interning" to avoid having equal strings in memory as strings would be memory hungry. Signals in different files share references to their names instead of having more instances of the same string. However, reading the input files is the slowest part of the process, string pooling is enabled when moving the signals to an appropriate data structure; in this way, we avoid slowing down the file reading stage. In the beginning, the application reads all covering sets directly from the file and stores them in hash tables using the identifiers of the signal as a hash key. Later, as we identify the sequence of signals, we move them to a dynamic array.

Figure A.8 show an example of the entire process. The upper table shows the original set covering representation, which stores each rising and falling transition covering for each signal. The bottom Venn diagrams show the set interpretation and highlighting per-pattern subsets.

Suppose to have $N$ signals (with $N$ potentially very large) and $M$ set covering (with $M$ limited by the number of different covers), the application has an $O(N \cdot M^2)$ time complexity and memory complexity. As the value of $M$ is usually limited to a few units, the time required by the entire process is restricted to a few tens of seconds in the worst case. Moreover, using a single bit for each signal value reduces the memory usage to a few GBytes.



| Signal Name | Rising transition-Coverage | | | Falling transition-Coverage | | |
|---|---|---|---|---|---|---|
| | $C_1$ | $C_2$ | $C_3$ | $C_1$ | $C_2$ | $C_3$ |
| a | 1 | 0 | 0 | 1 | 0 | 0 |
| b | 0 | 1 | 0 | 0 | 1 | 0 |
| c | 1 | 0 | 1 | 1 | 0 | 1 |
| d | 1 | 1 | 1 | 1 | 1 | 1 |
| e | 0 | 0 | 1 | 0 | 1 | 1 |
| f | 0 | 0 | 1 | 0 | 1 | 0 |



Fig. A.8: Rising and Falling transition set cover: From file to set interacts.

### A.2.5 Tool E: The Hierarchical Analysis Tool

The "Divide et impera" approach in digital design has increased the overall complexity of SoCs, allowing teams to focus more on a design of a single entity instead of the whole SoC. Following this approach, today's SoCs comprise several subunits with a variable number of other nested subunits up to the leaves, i.e., the logic gates. Therefore, considering metrics on SoCs with millions of gates, it becomes evident that coarse metrics on the overall device have less meaning that thorough computed metrics on modules below the average coverage of the whole SoC.

The stress analysis frequently has to concentrate on some specific module, e.g., the ones showing low coverage. In order to support this analysis, the flow includes a selection process implemented as a tool that extracts critical modules below a given threshold and a per-module stress coverage.

The hierarchical analysis tool analyzes a stress pattern, or a set of them, in order to produce a module-based coverage file where the module and sub-modules have their coverage within the design hierarchy.

The hierarchical analysis starts from the standard human readable text-based input file of coverage. Table A.2 presents an example of a coverage file, where signal names contain the hierarchy with the associated stress coverage.

Table A.2: A simplified view of the coverage file.

| **Signal Name** | ... | **Coverage** | ... |
|---|---|---|---|
| a/b/c | ... | 1.0 | ... |
| a/b/d | ... | 0.5 | ... |
| a/b/f/g | ... | 0.5 | ... |
| a/h | ... | 1.0 | ... |

The tool is independent of the SoC and analyzes the input file sequentially to avoid non-deterministic access to data structures. Consequently, it analyzes only the coverage file recreating the hierarchy by decomposing the path. In other words, using as an example the design represented in Table A.2, the top-level unit *a* is decomposed into a subunit called *b* and *h*; *b* is further decomposed into its children, i.e., *c*, *d*, *f*, and the leaf *g*.

Internally, it works on parsing strings in such a way as to create a tree of modules and sub-modules, starting from the top entity; for each node of the tree, it calculates and saves the list of signals and their coverages in the internal data structure. The tool creates the tree-coverage structure depicted in Figure A.9.

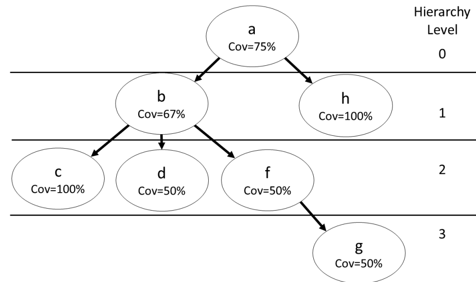



Fig. A.9: A high-level view of the tree data structure containing the coverage for each node.

An important aspect to mention is that the coverage of a general parents node in the tree hierarchy follows the recursive formula:

$$Cov(node) = \frac{\sum_{i=0}^{\#children(node)-1} Cov(i)}{n} \tag{A.3}$$

Where n is the number of leaves in the sub-tree, if the number of children of the node is one, then the node coverage is returned. The formula can generate, given a module, its related stress coverage by computing the average on all the children nodes.

More in detail, it generates warnings on those modules in the hierarchy that does not reach the acceptable level of coverage, given as an input parameter. Moreover, it produces a file containing every module with its associated level in the hierarchy and the related stress coverage. Instead of searching for a not satisfied coverage module, the tool also accepts a module name (a string) to extract the stress coverage and generate a coverage file within the module hierarchy.

A hierarchical decomposition of SoC modules eventually allows for test engineers to focus on the module, which would more likely give a more significant step up into the coverage than modules that have already reached an acceptable level.

### A.2.6 Tool F: The Chip-Surface Stress Plotter

Although an essential aspect of the stress approach is a quantitative information provided by the hierarchical analysis tool and the set covering tool, a qualitative visualization of stress plays a crucial role. Qualitative visualization of stress over the SoC layout allows locating stress pattern weaknesses and easily highlighting the coverage abilities of different stress patterns.

The plot tool uses as input a pure outcome of the VCD analysis or the superimposition of different stress approaches from the set covering tool to generate a heatmap of the stress over the SoC layout.

Figure A.10 presents an example of such a heatmap over a generic SoC.

All signals are mapped into pixels that summarize the stress applied to the device gates, while white zones are embedded memories that are not part of the BI grading. Gate stress coverage resides between 0% (red pixel) and 100% (green pixel), with values in the middle represented by a color gradient.

As it can be seen from Figure A.10, there exist portion of red that correspond to non-stressed modules, portions of green (stressed modules) in different shades depending on the resolution of the image, and islands of white that correspond to memories, analog and power modules (outside the scope of BI).

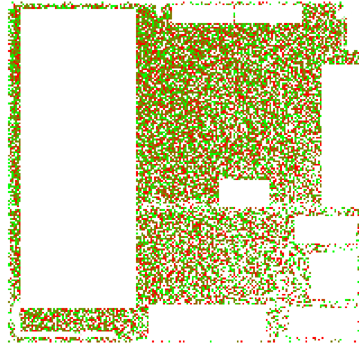


Fig. A.10: An example of stress heatmap over a generic SoC layout.

Since images of arbitrary resolutions can be generated, it is important to speed up the drawing process depending on the level of detail we need in parallel. GPUs are usually the way to go for image elaboration, as their parallel computation capabilities vastly surpass the ones of the CPU. However, to guarantee compatibility with every device, we perform computations on CPU resorting to the widely used SDL2 libraries[124] for saving the image into a standard format. SDL2 libraries provide APIs for image creation and manipulation on the CPU and GPU. It also provides methods to color the image pixel-by-pixel. Since every pixel is independent, this tool assigns each pixel to a different thread. In this case, we use OpenMP[125], a widely used library for easing parallel patterns implementation to perform image generation.

# Appendix B

# Trace Analysis

## B.1 State-of-the-art for functional test programs evaluation

Table B.1 presents a qualitative evaluation of the state-of-the-art assessment methodologies based on different aspects, ranging from the necessity of an experimental setup, its cost in terms of equipment and resources, the abstraction level at which the assessment is carried out, and the target fault model. The last two columns are the most significant as they represent the controllability and observability obtained by the methodology, i.e., the capability of controlling where a fault is inserted and where its effect can be observed.

The beam experiments have been used to access functional test programs in a real-case scenario (neutron beams). However, it is a fast but expensive methodology targeting only transient faults without knowing where the fault is located.

A debugger-based fault injection lowers the abstraction to the assembly level, is capable of injecting transient faults in the user register, and can observe whether the application crashes or not. It requires the physical actual device, as the beam experiments methodology, and a debugger capable of corrupting register values; thus, even if it has an acceptable execution time, it remains an expensive technique.

Micro-architectural fault injection relies on the architectural description of the SoC, having a medium cost, including the setup of the micro-architectural simulation and description on a medium-performance server. It can inject transient and perma-

Table B.1: State-of-the-art methodologies for functional test program assessment for hardware testing of CPU-based SoCs.

| Method | Experimental Setup | Cost | Abstraction | Execution Time | Fault Model | Controllability | Observability |
|---|---|---|---|---|---|---|---|
| Beam experiments [109] | Real Device | Very High | Application Level | Fast | Transient | Random | Application Error |
| Debugger-Based Fault Injection [92, 104] | Real Device | High | Assembly Level | Medium | Transient | User Register | Application Error |
| Proposed methodology [45] | Real Device or ISA simulator | Medium | Assembly Level | Medium | Transient Permanent | User Register | Data Propagation |
| Micro-Architectural Fault Injection [109] | Architectural description | Medium | Micro-Architectural Level | Medium | Transient Permanent | Arch. Registers Modules | Arch. Registers and Application errors |
| Toggle coverage analysis [26] | Logic Sim. | High | Signal Level | High | Toggle | Signal | Signal |
| RTL Fault Simulation [126, 110] | Fault Sim. | High | Behavioural level | High | Transient Permanent | RTL Signal | RTL Outputs |
| Gate-level Fault Simulation [102, 127] | Fault Sim. | Very High | Gate level | Very High | Transient Permanent | Real Signal | Real Outputs |

nent faults in memorization elements at a micro-architectural level in a reasonable execution time. The fault can be controlled in the architectural registers or a specific module, and it is observed as an application error or a mismatch in the architectural registers.

Netlist-based methodologies can rely on different types of fault models, and they are very accurate when controlling the fault location and observing its effect on the outputs. On the one hand, toggle coverage analysis can be used before gate or RTL fault simulation to check if a fault can be controlled in a specific line. Thus, controllability and observability are bound to the signal level. Meanwhile, RTL fault simulation can inject the fault in the RTL signal and observe the RTL outputs to understand if the fault is excited and propagated to the outputs; gate-level fault simulation can inject the fault on the real inputs (i.e., the actual signal present in the physical device) and observe if it can be observed on the outputs.

The Trace Analysis methodology analyzes how data propagates among instructions to observable points, identifying blocking instruction overriding computed results and potentially affecting the final fault coverage. This approach can be added in the early phases of functional test program developments to guide test engineers, avoid repetitive fault simulation campaigns, and, at the same time, provide fine-grained feedback on the developed functional test program. The methodology shares similarities with the micro-architectural fault injection methodology. However, a micro-architectural description of the SoC is not always available, and it has a cost to configure it properly. On the contrary, the proposed methodology relies on analyzing the instruction trace that can be extracted from the actual device, an Instruction Set Architecture (ISA) simulator, or a logic simulation (often integrated into test benches capable of executing functional programs). This approach makes the proposed trace analysis methodology cost-effective and viable for many testing scenarios.

## B.2 The Trace Analysis Methodology

Logic and fault simulation are the most expensive phase of creating high-quality functional test programs. Therefore, reducing the number of fault simulations is very important in the testing area. The Trace Analysis methodology addresses these aspects.

Two of the most adopted methodologies in the security research domain are "symbolic execution" and "dynamic taint analysis" [88–91]. Both these strategies analyze some software piece of code and grade it from the security perspective. Symbolic execution automatically builds a logical formula describing a program execution path, and it reduces the problem of reasoning about a program to a logic domain. Dynamic taint analysis runs a program and observes which computations are affected by predefined taint sources, such as the user inputs.

Following the dynamic taint analysis paradigm, a technique to perform a preliminary evaluation of a functional test program without the necessity of a logic or fault simulation step in the early development phases is proposed. The flow of the methodology is illustrated in Figure B.1, which major product is the computation of a novel metric called "connectivity". The connectivity value is fast to compute and can effectively guide the development, especially during the early stages.

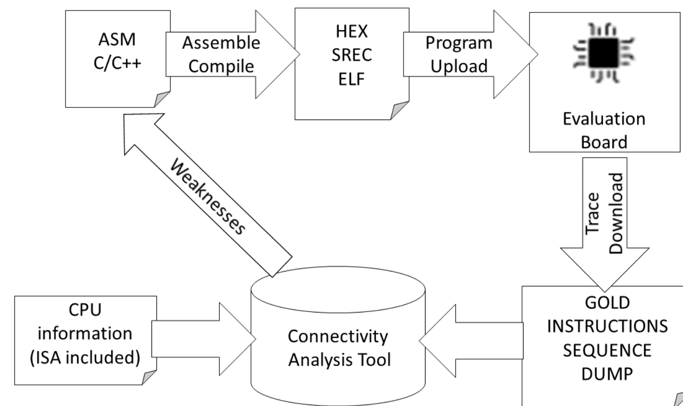



Fig. B.1: The logic flow of our testing framework.

Previously designed programs in C language or Assembly code [47, 69] are used as benchmark tests (top-left corner of Figure B.1). These test programs are compiled and executed on the silicon device and the instruction traces are generated by adopting advanced debugging tools. Each trace contains the list of executed instructions, with all selected branches, all loops properly unrolled, and all actual registers and memory values generated by the chip during the execution of the machine code.

Test programs are developed to exercise functional blocks of the device and their characteristics have a direct relationship with the verified functionalities and, as

a consequence, with their final coverage. Their instruction traces are considered the gold functional executions (bottom-right block of Figure B.1). Once they are produced, each trace is parsed and transformed into a Control and Data Flow Graph (CDFG) by the connectivity tool (bottom-center block of Figure B.1). Portability between different architectures and ISA is achieved by the CPU information file (bottom-left block). An important aspect to highlight is the independence of the proposed approach from single and multi-core executions of the functional test programs. The connectivity analysis is performed from a CPU register perspective, meaning that we investigate how the data propagates within CPU registers and across instructions of test programs. Therefore, even with a multi-core execution, the functional routines can be divided and analyzed based on the CPU running them. Moreover, a multi-core program shares common resources and data, and consequently, it needs some synchronization mechanism and atomic operations in order to avoid non-deterministic values in memory which are taken into account by the CPU information file. In the connectivity step, every instruction is represented by a vertex and the information flow is represented by directed edges. We focus on writing and reading operations and their temporal dependency, independently from the sequence of operations and the program complexity in terms of lines of code. Thus, given a data value $d$ in the trace (either a register or a memory location), two types of edges are introduced:

- *WAW* (Write-After-Write) edges represent two write operations on $d$ back-to-back with no reading instructions in between.
- *RAW* (Read-After-Write) edges represent a write operation followed directly by a read operation on $d$.

Once it has been built, the CDFG is analyzed to extract the “connectivity” metric. The connectivity measures potential code weaknesses and guides test engineers to grade the quality of the functional procedures, rectify their problems, and insert proper signature checks to improve their quality. The main advantage of this procedure is that the entire process is extremely fast (up to seconds) when compared to a standard logic and fault simulation phase (requiring up to hours or days).

The analysis relies on the following observation.

**Observation 1** *Write-after-Write (WAW) instruction sequences occurring during the instruction flow over a shared addressable location causes the previously computed values to be overwritten and most likely lead to a loss of fault coverage. Conversely, Read-after-Write (RAW) instruction sequences propagate values along with the execution flow, and they can reach an observation point, possibly leading to an increase in fault coverage.*

The following example illustrates the main concepts of our process.

**Example 6** *Let us consider the code segment reported in Table B.2. The code includes three instructions belonging to PowerPC VLE Instruction Set Architecture and the operation they perform.*

Table B.2: An example of instruction sequence with the operation performed.

| Cycle | Instruction | Operation |
|---|---|---|
| 1 | e_add2i r19, r18, 192 | r19 = r18 + 192 |
| 2 | subfme r6, r18 | r6 = r6 - r18 |
| 3 | e_add16i r6, r19, 35 | r6 = r19 + 35 |

*For the sake of simplicity, we may suppose to represent each instruction with a vertex node as shown in Figure B.2a. The data written in r6 by instruction number 2 is immediately overwritten by instruction number 3. In the representation, this is represented by a WAW edge incident from vertex 2 and in vertex 3 (Figure B.2b). Vice-versa, the write operation on register r19 performed by instruction one is preserved by the reading operation of instruction 3 and this is represented by a RAW edge incident from node one and in node 3 (Figure B.2b). Even if there are cases in which WAW edges may have a specific meaning, they somehow correspond to undesirable situations with information flow disruptions. These situations should never appear in ASM or compiled C/C++ functional programs. On the contrary, RAW edges indicate the standard operation flow and are beneficial.*

*From a testing point of view, instructions 1 and 3 have may provide a positive effect of the fault coverage when the destination registers is propagated to an observable point. On the contrary, instruction 2 is not beneficial to fault coverage as the value of r6 is overwritten and cannot be propagated. As a consequence, instruction 2 is useless and can be canceled, or the program needs to be modified to unlock a possible propagation of r6. To sum up, WAW and RAW edges allow us to color the vertices of the graph as illustrated in Figure B.2c and described in the following paragraphs.*

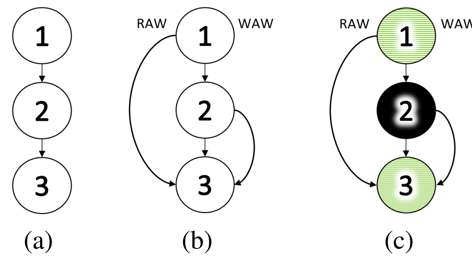



Fig. B.2: A graphical representation of Example 6.

The analysis performed on the golden instruction dump verifies the presence of blocking situations similar to the one illustrated in the previous example. Analyzing the presence of WAW and RAW edges is essential, at least in the following contexts:

1. Randomization and evolutionary methods to automatically generate functional programs may naturally introduce WAW edges. Therefore, it would be beneficial to tune the generation of programs to minimize WAW edges and reach a high coverage faster.

2. Bugs may constantly be introduced in assembly software, i.e., even simple typos may introduce unwanted WAW sequences when writing low-level ASM code. Discovering an error after a fault simulation is frustrating, and a quick preliminary check would significantly reduce time loss.

3. It is often unfeasible to grade the fault coverage achieved by very long programs, such as benchmarks directly compiled in C/C++ language or OS boot procedures. In this case, checking potential weaknesses can be one of the few feasible measures in short time.

As shown in the following subsection, every instruction node of the CDFG is classified as blocked (black) or not blocked (green), depending on whether it propagates, or not, the result of its execution according to Observation 1. The *connectivity* metric is computed as the fraction of non-blocked instructions over the total number of investigated instructions. A program with very high connectivity can be considered promising from the perspective of its potential fault coverage. Conversely, if the program shows a lot of blocked instructions and a low connectivity, it needs to be

revised to avoid useless and time-expensive fault simulation stages. Furthermore, the connectivity measure locates the blocked instructions and may guide the designer to fix the proper components along the development process.

### B.2.1 The Basic Algorithm

Although static and dynamic code analysis is performed in other fields [128, 88–91], in the proposed approach, we process the code generated with a hardware debugger during the execution of functional programs on real silicon. Similarly to the taint analysis, the approach statically elaborates the instructions flow executed by the program to identify blocking situations within the golden instruction trace. Therefore, the proposed methodology recognizes critical edges in the CDFG, i.e., it identifies WAW edges preventing information to be forwarded along the program flow.

Table B.3: Instruction sequence, source, and destination operands.

| Cycle | Address | Instruction | SRC | DST |
|---|---|---|---|---|
| 1 | 0x0000_0000 | subfme r19, r6 | r19, r6 | r19 |
| 2 | 0x0000_0004 | e_add16i r6, r19, 35 | r19 | r6 |
| 3 | 0x0000_0008 | e_add2i r6, r19, 192 | r19 | r6 |
| 4 | 0x0000_000C | e_add2i r6, r6, r1 | r6, r1 | r6 |
| 5 | 0x0000_0010 | subfme r19, r6 | r19, r6 | r19 |
| 6 | 0x0000_0014 | subfze r6, r3 | r6, r3 | r6 |
| 7 | 0x0000_0018 | e_add16i r6, r19, 35 | r19 | r6 |

Table B.3 reports a short snapshot of a golden instruction dump extracted from a manually developed test routine for the CPU adder unit. The first column indicates the order of execution; the second and third columns report the address and the mnemonic code of the relative instruction; columns SRC and DST specify the sources and destinations of each instruction.

Based on this information, the basic algorithm runs through the steps illustrated in Figure B.3:

1. In the first step, the one represented by Figure B.3a, we build a CDFG in which each instruction is mapped onto a vertex and the data flow is represented through edges. We use two types of edges, namely RAW (reported on the

left-hand side of the graph) and WAW (reported on the right-hand side of the graph).

2. During the second stage, Figure B.3b, we perform a visit of the CDFG following WAW edges. Vertices are colored either in red or green color.

3. During the third and last step, Figure B.3c, we perform a visit of the CDFG following RAW edges. Red vertices may become green or black.

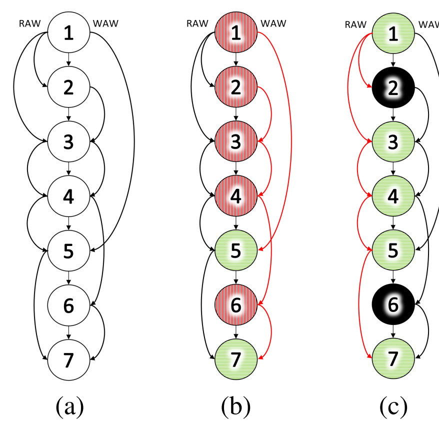



Fig. B.3: The CDFG of the code snippet of Table B.3 in represented in Figure (a). Figure (b) illustrates the vertex-coloring process obtained following WAW edges, and Figure (c) the colors obtained at the end of the RAW visit.

To build the CDFG, we use the function reported in Algorithm 15. For each node (i.e., instruction) of the graph (line 1), we visit all subsequent nodes (instructions) of the graph (function Search) looking for the destination of the current node as source or destination of one instruction. WAW edges are inserted to connect "victim" instructions to "aggressor" instructions (line 6). Aggressor instructions overwrite the result of victim instructions. RAW edges are inserted to connect a predecessor and a successor instruction (line 8). Building the graph has a cost that is linear in the number of instructions of the gold model.

Function Build_Graph is followed by Algorithm 16 which visits the graph for the first time, considering only WAW edges. For each node of the graph $G$ (line 1), we check the existence of an outgoing WAW edge: Each vertex with an outgoing WAW edged is colored in red (i.e., dark gray in the black-and-white reproduction of the picture) or in green (i.e., light-gray) if they do not present an outgoing WAW

**Algorithm 15** Phase 1 of Figure B.3a: From the gold trace to the CDFG. The procedure elaborates the execution dump and creates the graph with all WAW and RAW edges.

```
1:  Build_Graph ()
2:  for (node in G) do
3:      Search (G, node)
4:  end for
5:  Search (G, node)
6:  for (n in G > node) do
7:      if (node_dst = n_src) then
8:          node_RAW = n
9:      elseif (node_dst = n_dst)
10:         node_WAW = n
11:         break
12:     end if
13: end for
14: return
```

edge. For example, in Figure B.3b instruction 4 is red (dark-gray) because it has an aggressor in instruction 6. On the contrary, instruction 5 is green (light-gray) because its computed value will reach the end of the program. Function WAW_Visit has a linear cost in the number of vertices of the graph.

**Algorithm 16** Phase 2 of Figure 4.27b: The WAW visit.

```
1: WAW_Visit (G)
2: for (node in G) do
3:     if (node_WAW ≠ ∅) then
4:         node_color = RED
5:     else
6:         node_color = GREEN
7:     end if
8: end for
```

The second visit essentially traverses RAW edges to finalize the color of all red nodes. The corresponding pseudo-code is reported in Algorithm 17. The structure of the code is similar to the one of the function Build_Graph in Algorithm 15. During the visit, a red node can turn green (line 8) if there is a RAW edge connecting it ahead to a green vertex (line 7). On the contrary, the block suspect is confirmed, and the red color is updated to black (line 12). For example, in Figure B.3b, instruction 4 is a case of a red node evolving into a green one, whereas instruction 6 is confirmed

as a blocking vertex, and it becomes black. Function RAW_Visit, as all the previous procedures, has a linear cost in the number of vertices of our CDFG.

**Algorithm 17** Phase 3 of Figure B.3c: The RAW visit.

```
1: Visit (G)
2: for (node in G) do
3:     RAW_Visit (node)
4: end for
5: RAW_Visit (node)
6: if (node_color = RED) then
7:     for (RAW_destination in RAW_edges of node) do
8:         ret = RAW_Visit (RAW_destination)
9:         if (ret = GREEN) then
10:            node_color = GREEN
11:            return GREEN
12:        end if
13:    end for
14:    node_color = BLACK
15: end if
16: return node_color
```

Once all graph nodes are colored, the connectivity value is computed as the percentage of green (light-gray) nodes over all nodes in the CDFG, and it indicates the percentage of the instructions that are beneficial in terms of fault coverage. For example, in Figure B.3, as 5 out of 7 instructions are finally green, the connectivity is 71.42%. The remaining two black vertices bring no valuable information to the assertion/check part of the code. Indeed, instructions 2 and 6 will never contribute to the fault coverage since their results are overwritten before propagating elsewhere.

### B.2.2 Optimized Algorithm

The algorithm previously described can be optimized by removing the WAW edges and the WAW visit. The method previously illustrated creates WAW arcs to find problematic instructions. However, WAW arcs are not necessary to obtain the final result, and we can perform the WAW visit while we build the CDFG, reducing the computation cost and improving the memory efficiency.

More specifically, when building the CDFG, instead of linking two nodes with a WAW edge, we evaluate an initial coloring of the node from which the edge

leaves. To detect this preliminary color we check whether the node which would be overwritten also contains a RAW edge. In the positive case the assigned color is red, otherwise black. For example, in Figure B.3, instruction 6 could be set to black already during the construction of the CDFG as the destination is overwritten, and none of the following nodes has a RAW dependency from it.

The optimized procedure building the CDFG, i.e., Algorithms 15, is reported in Algorithm 18.

**Algorithm 18** The optimized algorithm for phase 1 (originally, Algorithm 15). Several nodes are colored up-front, saving subsequent computation time.

1: Build_Graph ($G$)
2: **for** (node in $G$) **do**
3:     Search ($G$, node)
4: **end for**
5: Search ($G$, node)
6: **for** (nextNode in $G >$ node) **do**
7:     **if** (nextNode$_{src}$ = node$_{dst}$) **then**
8:         node$_{RAW}$ = nextNode
9:     **end if**
10:     **if** (nextNode$_{dst}$ = node$_{dst}$) **then**
11:         **if** (node$_{RAW} \neq \emptyset$) **then**
12:             node$_{color}$ = RED
13:         **else**
14:             node$_{color}$ = BLACK
15:         **end if**
16:         break
17:     **else**
18:         node$_{color}$ = GREEN
19:     **end if**
20: **end for**

## B.2.3 Load/Store Instructions

Memory values are manipulated as virtual registers. The golden instruction sequence is first enriched with information related to registers to locate memory locations. Then, the current values of these registers is used to compute the virtual register address of the memory location. The following example clarifies this strategy.

**Example 7** *The code snippet reported in Table B.4 includes load and store instructions from and to memory locations in addition to arithmetic operations. For the sake of simplicity, all memory locations have a size of 1 byte.*

Table B.4: Instruction sequence, source and destination operands.

| Cycle | Instruction | SRC | DST |
|---|---|---|---|
| 1 | e_stb r0, 0(r1) | r0, r1 | mem(0+r1) |
| 2 | e_add16i r2, r0, 35 | r0 | r2 |
| 3 | e_lbz r2, 0(r1) | mem(0+r1), r1 | r2 |
| 4 | e_add2i r2, r1, r2 | r1, r2 | r2 |
| 5 | e_stb r2, 0(r1) | r2, r1 | mem(0+r1) |
| 6 | subfze r1, r3 | r1, r3 | r1 |
| 7 | e_add16i r1, r0, 35 | r0 | r1 |

*Figure B.4a reports the corresponding graph and Figure B.4b the final coloring scheme returned by the RAW visit. The example shows how to handle memory addresses as registers. For example, instruction one stores a value in the location pointed by register r1. A load instruction accesses the same location at position 3. Therefore, instruction one is finally labeled as green (light-gray). Moreover, instruction 5 is marked as green because the value written in memory is propagated until the end.*

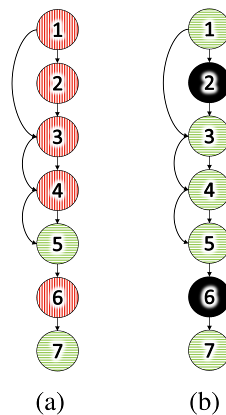



Fig. B.4: An example including arithmetic operations and load/store instructions.

### B.2.4 Branch Instructions

Another extension is required to manage branch instructions. In order to analyze and classify branches as green or black, the basic algorithm is modified in the following directions:

- During the regular visit, conditional branches are left undecided and temporarily colored in orange.
- An additional visit is performed to find the alternative branch destination among the instructions executed after the conditional statement.
  - If the branch's alternative destination address cannot be found, then, the branch is colored in black.
  - Conversely, when the branch alternative destination is in the traced instruction flow, the visit checks whether the incorrect branch execution leads to a different signature or not and color the branch in black or green.

However, notice that just some cases can be resolved by this strategy as the algorithm can color the branches in green or black only when the alternative branch can be found in the code.

**Example 8** *The code reported in Table B.5 includes a branch generated by a simple if-then-else construct.*

Table B.5: Instruction sequence, source and destination operands.

| Cycle | Address | Instruction | SRC | DST |
|---|---|---|---|---|
| 1 | 0x0000_0000 | e_li r0, 0 | | r0 |
| 2 | 0x0000_0004 | cmpl r0, 1 | r0 | cr |
| 3 | 0x0000_0008 | e_beq jump | cr | |
| 4 | 0x0000_000C | cmpl r0, 0 | r0 | cr |
| 5 | 0x0000_0010 | e_add16i r1, r2, r3 | r2, r3 | r1 |
| 6 | 0x0000_0014 | jump: e_add16i r1, r0, r1 | r0, r1 | r1 |

*Figure B.5a reports the result of a first visit, leaving the branch node as undecided. The second visit, whose result is represented in Figure B.5b, evaluates the signature under the*

*hypothesis of taking the wrong branch path. In this specific case, the instruction that would be reached if the branch decision was wrong is included within the instructions that follow the branch itself. A new orange-colored edge is added to the figure to highlight this situation, and it appears that some instructions would not be executed if the branch was wrongly executed (taken if it was not taken, and vice-versa).*

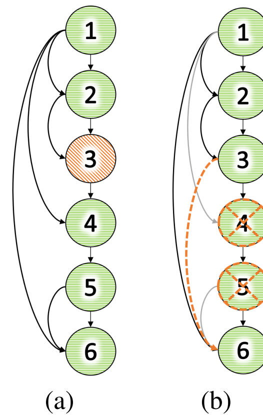



Fig. B.5: A graph example including a branch instruction.

*The instructions that may be skipped are green. Therefore, the value of the final register may change, and the signature value may potentially be compromised. Indeed, the branch node is labeled with a green color. Conversely, if all skipped instructions had been labeled as black nodes, they would not contribute to the signature value. In this case, the branch would be colored black, as illustrated in Example 9.*

**Example 9** *Let focus on the code reported in Table B.6. Following Example 8, the first visit leaves the branch node undecided. The second visit determines that the signature would not change under the hypothesis of taking the wrong branch path, i.e., all skipped instructions are labeled as black nodes. As a consequence, the branch is colored black. The outcomes of the two visits are illustrated in Figure B.6.*

### B.2.5 Multiple Destination Instructions

To complete the functionalities of the proposed methodology, a last extension has to be considered, as the code can include instructions with multiple destinations. In

Table B.6: Instruction sequence, source and destination operands.

| Cycle | Address | Instruction | SRC | DST |
|---|---|---|---|---|
| 1 | 0x0000_0000 | e_li r0, 0 | | r0 |
| 2 | 0x0000_0004 | cmpl r0, 1 | r0 | cr |
| 3 | 0x0000_0008 | e_beq jump | cr | |
| 4 | 0x0000_000C | e_add16i r1, r2, 1 | r2 | r1 |
| 5 | 0x0000_0010 | e_add16i r1, r0, 1 | r0 | r1 |
| 6 | 0x0000_0014 | jump: e_add16i r1, r0, r2 | r0, r2 | r1 |

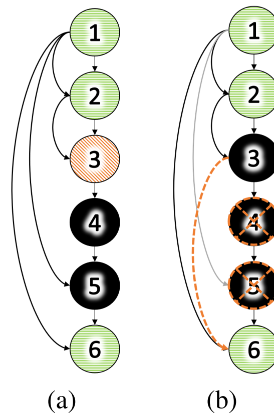



Fig. B.6: Another branch instructions example.

this case, the analysis is extended to target destinations instead of instructions as illustrated in Example 10.

**Example 10** *Table B.7 reports a code snippet where several instructions have more than one operand destination.*

Table B.7: Instruction sequence with multiple destinations.

| Cycle | Address | Instruction | SRC | DST |
|---|---|---|---|---|
| 1 | 0x0000_0000 | subf r0, r1 | r0, r1 | cr, r0 |
| 2 | 0x0000_0004 | e_add16i r1, r0, 35 | r0 | r1 |
| 3 | 0x0000_0008 | e_add2i r1, r0, 192 | r0 | cr, r1 |
| 4 | 0x0000_000C | e_add2i r1, r1, r1 | r1 | cr, r1 |
| 5 | 0x0000_0010 | subf r0, r1 | r0, r1 | cr, r0 |
| 6 | 0x0000_0014 | subf r1, r2 | r1, r2 | r1 |
| 7 | 0x0000_0018 | e_add16i r1, r0, 35 | r0 | r1 |

*For the sake of completeness, Figure B.7 shows the initial graph (Figure B.7a), the one with WAW and RAW edges, and the one obtained after the final coloring phase (Figure B.7c). Coloring is performed considering "single" destinations. Consequently, each graph vertex includes a color for each of the destination fields considered by the instruction.*

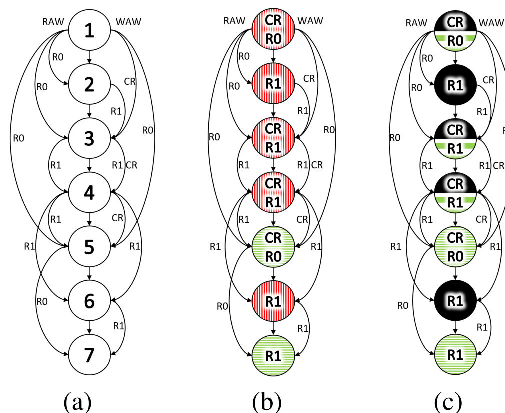



Fig. B.7: A code snippet with instructions with multiple destinations: Our instruction-oriented analysis is modified to be destination-based.

Algorithms 19 and 20 report the modified functions to build and visit the CDFG in case of multiple destinations. The data structure of a node has been changed to

**Algorithm 19** Build graph function that elaborates the execution dump considering multiple destinations.

```
1:  Build_Graph ()
2:  for (node in G) do
3:      for (dst in node) do Search (G, node, dst)
4:      end for
5:  end for
6:  Search (G, node, dst)
7:  for (n in G > node) do
8:      if (n_src = dst) then
9:          n_srcRAW = dst
10:     end if
11:     if (n_dst = dst) then
12:         if (dst is never read) then
13:             dst_color = BLACK
14:         else(if dst is read)
15:             dst_color = RED
16:         end if
17:         return
18:     end if
19: end for
20: if (dst never found) then
21:     dst_color = GREEN
22: end if
```

**Algorithm 20** RAW visit considering multiple destinations.

```
1:  Visit (G)
2:  for node ∈ G) do
3:      RAW_visit (node)
4:  end for
5:
6:  RAW_visit(node)
7:  for (dst in node) do
8:      if (dst_color = RED) then
9:          for (RAW edge i) do
10:             if (dst_color ≠ GREEN) then
11:                 dst_color = RAW_visit (RAW edge[i])
12:             end if
13:         end for
14:     end if
15: end for
16: if (Any of dst in node = GREEN) then
17:     return GREEN
18: end if
19: return BLACK
```

manage multiple destinations. Moreover, if a node has at least one green destination, i.e., the instruction propagates at least some partial result, the data is considered propagated. When considering multiple destinations, the connectivity metric needs to be computed in a slightly different way. Firstly, a connectivity percentage to every node is assigned. Then, the overall connectivity value is computed as the average of the connectivity of all nodes.

For instance, in Example 10, node 1, 3 and 4 are 50% connected, instruction 5 is 100% connected, and instructions 2 and 5 are not connected at all. Therefore, the overall connectivity metric value is computed as $(3 \cdot 50\% + 2 \cdot 100\%)/7 = 50\%$.

# Appendix C

# FPGA-based tester

## C.1 State-of-the-art for FPGA-based tester

In order to provide placement of the proposed tester into the state-of-the-art FGPA-based tester, Table C.1 is presented. The Comparison is done from the perspective of FPGA-based tester for digital systems. Moreover, in the literature there is a lack of FPGA-based testers, advanced and non, in the last years. Industrial ATEs were not taken into account because the purpose of this work is not to compete with ATEs on the market, but to allow research centres with low budgets to perform their activities on in-production devices, at affordable prices. Table C.1 compares different FPGA-based testers from a structural and functional perspective.

In [129], a tester capable of compressing test programs is presented, but it has no enhanced feature for functional test programs. Furthermore, in [130–132], different testers are presented for applying test patterns on the IOs of SoCs, by either generating on-the-fly or applying precomputed patterns.

In [133], an FPGA-based tester is presented for only functionally verifying a communication. Meanwhile, in [134], it focuses on testing GPIO and verifying propagation delay and power consumption.

On the other hand, authors in [135] presents a tester capable of feeding precomputed test patterns (in multi-chain configuration) at 50GHz, thanks to a high-speed interface between a Socket-specific test board and custom IPs. However, it does not provide any functional capabilities.

Table C.1: Comparison between different FPGA-based tester.

| Tester | Description | Host PC | Hardware requirements | Structural capabilities | | | Functional capabilities | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Max Pattern feeding speed | Pattern Memory requirements | Single/Multi chain | Executable flashing | Executable debugging | Instruction Trace capabilities | Protocol-based communication |
| [129] | FPGA-based with data compression | Required | FPGA design | 50MHz | 5MB | NA | yes | NA | NA | NA |
| [130] | FPGA-based with on-board power supplies | Required | FPGA custom board | NA | NA | NA | NA | NA | NA | NA |
| [131] | FPGA-based applying simulation-extracted patterns | Required | FPGA design | 6 MHz | 576KB | NA | NA | NA | NA | NA |
| [132] | FPGA-based for multi-IC, on-chip pattern generation | Required | FPGA design and multiple hw connections | 1 MHz | NA | NA | NA | NA | NA | NA |
| [133, 136] | FPGA-based functional protocol | Required | FPGA design | NA | NA | NA | NA | NA | NA | Yes |
| [134] | FPGA-based functional GPIO test propagation delay and power consumption test | Required | FPGA design Power measurements HW | 200 MHz (funct.) | 32KB | NA | NA | NA | NA | Yes |
| [135] | FPGA-based with precomputed scan pattern, IOs stimulus | No | FPGA design Socket-Specific test board | 50 GHz | 16GB | Multi | NA | NA | NA | NA |
| [86] | FPGA-based with on-chip pattern generation (SW) precomputed scan pattern sequencer | Required | FPGA-design | 6.25 MHz | Max 2GB | Single | NA | NA | NA | NA |
| Proposed Tester | FPGA-based with on-chip pattern generation (HW) precomputed scan pattern sequencer and full functional capabilities | No | FPGA-design SW development | 100 MHz[1] | Max 2GB | Single/Multi | yes | yes | yes | yes |

[1] The DUT was capable of reaching only 20MHz due to the device's I/O limitation on the maximum frequency.

The proposed tester aims to enhance a preliminary version (with only limited structural capabilities) presented in [86]. However, none of the aforementioned FPGA-Based Testers provide a combination of structural and advanced functional capabilities as well as design modularity, functional test program information extraction, and protocol-based communication.

## C.2 Proposed Hardware architecture

It is challenging for research labs, in companies or universities, to develop new test strategies since semiconductor ATE may not be affordable, and unnecessary. This greatly limits the ability of research labs.

The hereinafter proposed flexible FPGA-based tester aims at overcoming the aforementioned limitations. It requires an affordable cost and space from research labs, compared to industrial ATE. The hardware architecture of the proposed tester is shown in Figure C.1, it is a flexible FPGA-based tester tailored for SLT, with the additional capabilities to structurally test Devices Under Test (DUTs), addressing issues of adaptability, modularity, and cost-effectiveness found in conventional testers. This has been achieved using the capabilities of a System on Chip which hosts both a Processing System and a Programmable Logic, in which the digital design has been synthesized, capable of communicating with each other.

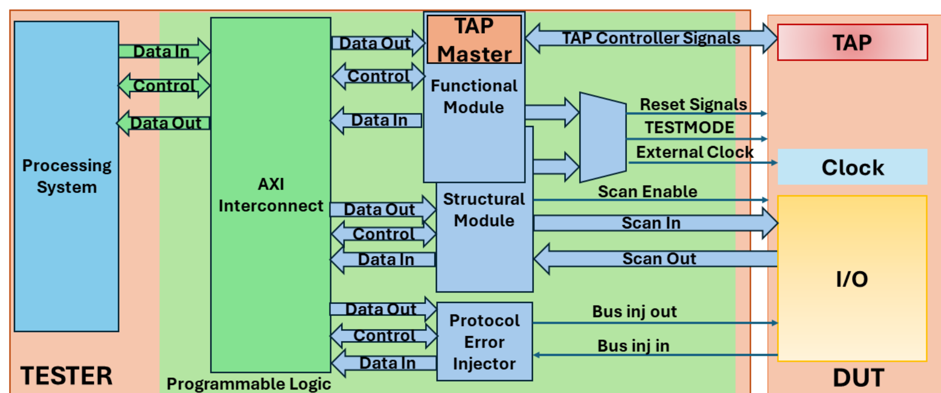



Fig. C.1: FPGA-based Tester Architecture.

In the following subsections, the two main hardware blocks of the proposed tester are presented. The first one focuses on functional testing by exploiting the JTAG

Standard [33]. The structural module is capable of executing the reconfiguration of the circuit for structural tests.

### C.2.1 Functional Module

The IEEE 1149.1 JTAG standard defines the Test Access Port (TAP) [33] as the interface to access and control the device in debug mode. All standardized devices integrate a slave TAP Controller which, through the five required pads (TCK, TDI, TDO, TMS and TRST) can be controlled via a master implemented in the tester.

Following this principle, the functional module for the proposed tester, in Figure C.2, was developed as a block composed of a Master TAP Controller, which provides full access to the internal TAP FSM of the DUT. To access the registers and the overall debugging features, an IP has been designed around this module.

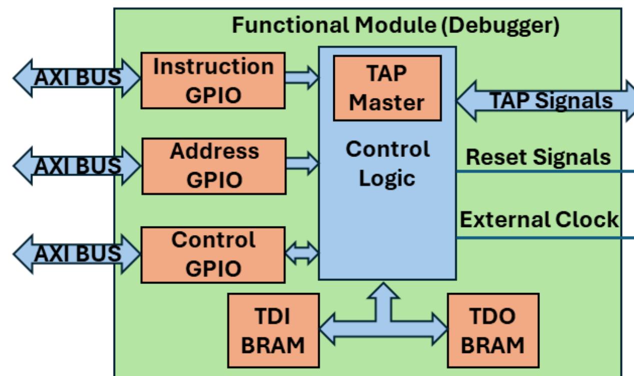



Fig. C.2: Functional Module Block Design

The module can be controlled through an AXI bus by the PS. It is attached to two double port Block RAMs, dedicated to store the data exchanged on the JTAG TDI and TDO lines. The address to read and write are provided to the module through two GPIOs, and can be accessed, not concurrently, by the PS.

A third GPIO is used to activate the module and wait for the completion of the execution, acting as a hardware mutex, avoiding concurrent access from different users. In Figure C.1 the main structure can be observed.

The operations to be executed by the functional module, in compliance with the JTAG standard, are provided through an instruction encoded to accept the following information:

- Reset status of the TAP.
- A bit to reach the Test Logic Reset state.
- A bit to decide whether a write operation needs to be performed on an Instruction or Data Register.
- Number of bits to shift out through TDI.
- One bit encoding whether the read operation needs to be performed on an Instruction or Data Register.
- Number of bits to shift in through TDO.
- Bits controlling the status of the Reset signals of the DUT (Power-On Reset and External Reset pads).

### C.2.2 Structural Module

The structural block is designed starting from the previously designed functional module. The functional module can be used to execute the Test Mode Entry, by just adding minor features. Furthermore, the tester requires the ability to sequence patterns into a scan chain.

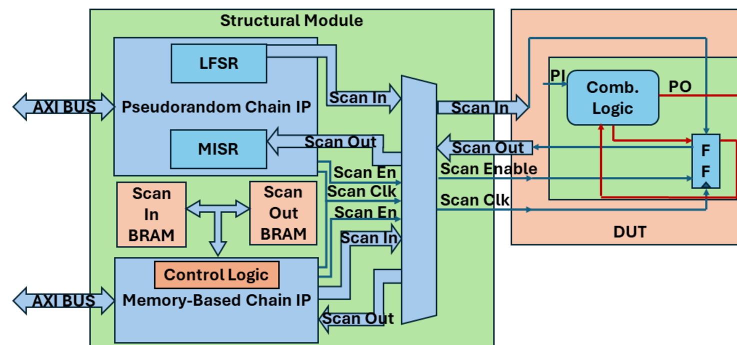



Fig. C.3: Structural Module Block Design

To do this, a module has been designed (in Figure C.3) to take as input the patterns and output them to the scan in pins, while reading the scan outs. The design has two Block RAMs, one for the patterns to scan in, and the other with the data read from the chain. To keep it modular, the number of scan chains can be set up

before synthesis thanks to a configurable IP. It works by reading the first word of the BRAM, which contains the number of bits to shift in the chain, then reading $N$ words (with $N$ being the number of chains) of 32 bits, shifting them in, and repeating the last two steps until the number of bits requested has been scanned into the chain. Reading the chain is done the same way. Additionally, the module is able to automatically generate patterns through a Pseudo-Random Pattern Generator (PRPG), and collecting the output signature, a condensed representation of a circuit's output response, through a Multiple Input Shift Register (MISR), which compresses the data read from the chains.

## C.3 Proposed Software Framework

The Hardware Architecture needs to be addressed from the Processing System via software, to exploit as much as possible the Hardware to create complex functions. To make sure that the tester's software kept both the highest modularity possible and a user-friendly design, scripting via the Python language is used, via linux drivers to access the design on the FPGA.

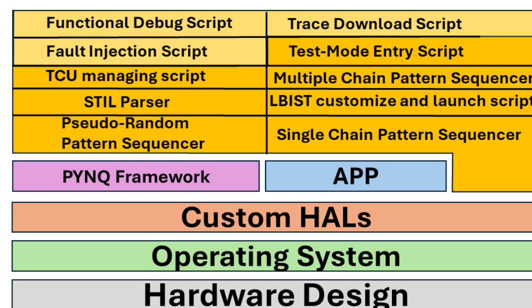



Fig. C.4: Software stack

All modules in the PL are accessible and controllable by the PS through the AXI. On the latter, a customized version of Linux runs, the Pynq OS [101]. The usage of an operating system comes with all the features it supports, most notably the *mmap* library [137], which was used to create a Hardware Abstraction Layer for the peripherals connected to the AXI. This way, the hardware keeps just the basic features to access the DfT (scan chain, LBIST engine) and debug infrastructure of the device, while the software is customizable.
On top of the HALs, a software stack resides (Fig. C.4), which, through the afore-

mentioned instruction to be passed to the hardware module, gives the tester the capabilities to perform the following tasks:

- Access to the on-chip Test Access Port (TAP) controller to extract basic device information.
- Extract information on registers and memories.
- Set on-chip breakpoints.
- Modify registers and memories.
- Inject errors in the memory words and their relative Error Correction Code (ECC).
- Execute Instruction trace download in both single and multi-core applications.
- Flash erase, programming and verify.
- Perform Test-Mode Entry for structural testing for single or multiple scan chains.
- Structural Testing for different fault models.
- Diagnostics of DUTs through the data retrieved by applying precomputed stimuli through a STIL file [100].

All the above capabilities are device specific, and are to be coded for each Device Under Test (DUT) used. The generic HALs to interface with the debugger, as well as the hardware design, are generic and do not necessitate modifications, unless required. The usage of PYNQ framework allows to easily read, understand and script the operations.

## C.4 Enhancing Modularity

By using a System-On-Chip featuring an FPGA, the proposed tester gains an inherent modularity, both on the hardware and software level, which gives the user the ability to create and integrate specialized modules directly on the FPGA fabric. These tailored modules seamlessly integrate with the core design, enabling targeted and comprehensive testing scenarios. This approach offers distinct advantages, including:

- Scalability: Easily add or remove modules, customizing the design for specific tests;
- Flexibility: Effortlessly customize to match requirements;
- Cost-effectiveness: Eliminate the need for expensive dedicated testing hardware, reducing costs.

An application example can be the necessity to functionally test the CAN Module of the DUT. SLT are used to functionally test the DUT, and peripherals are a fundamental block in modern safety-critical systems. In this case, a generic communication peripheral can be added in the Programmable Logic to communicate with the chip. Moreover, if the user needs a protocol error injector to test the answer of the communication peripheral to errors within the frame, a design can be integrated as part of the tester [1], as Figure C.5 shows.

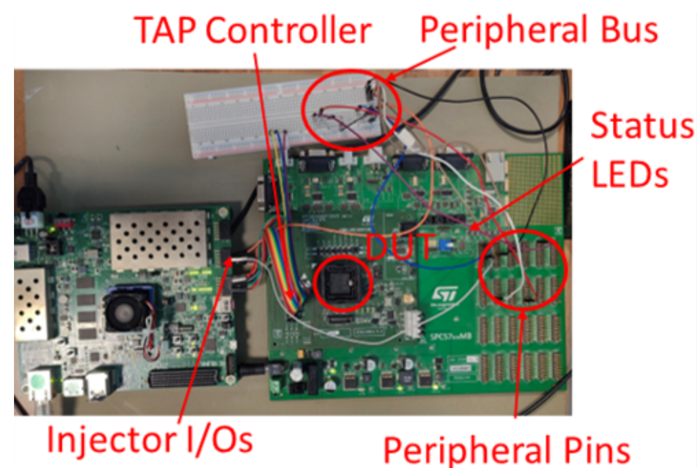



Fig. C.5: Experimental setup for SLT of communication peripherals (CAN).

## C.5 Final Considerations

Companies need System-Level Testers who can seamlessly blend functional and structural testing [54]. However, integrating these test steps demands significant investment in collaboration, expertise, and tools. Finding Test Equipment that combines these diverse test methods remains a complex and costly hurdle. While

industrial semiconductor testers exist, no FPGA-based tester on the market unifies advanced SLT and structural testing.

The proposed tester aims to help solve these problems by using a standard architecture, with modular capabilities. It aims to give the users the ability to run multiple typologies of tests, ranging from functional to structural tests, leaving the space to enhance its features without additional external hardware requirements (or minimal).

The proposed approach enables, at low costs, researchers to perform:

- General device debugging.
- Program tracing as in [45, 92].
- Load programs into the DUT and test communications with external peripherals [1].
- Execute Test-Mode Entry and use of scan chains (single/multiple) using pre-computed or pseudo random patterns.
- Customization and launch of Logic BISTs engine.
- Diagnosis of DUTs through DfT.

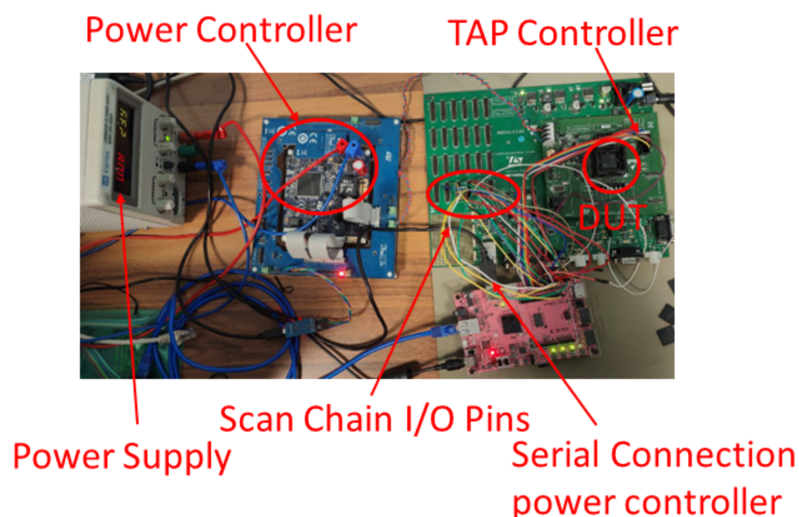



Fig. C.6: Experimental setup with power controller board.

The tester's abilities can be enhanced by adding the capabilities to control the input voltage of the DUT (using a PCTRL), with an experimental setup like the one shown in Figure C.6. The idea is to execute diagnostics (functional and structural) and tests, forcing different voltages for characterizing faults due to process variations, at the cost of an additional custom board for voltage regulation.

Moreover, as Figure C.7 represents, the tester can be exploited for running functional test programs at different voltages and measuring the power consumption (using an ADC).

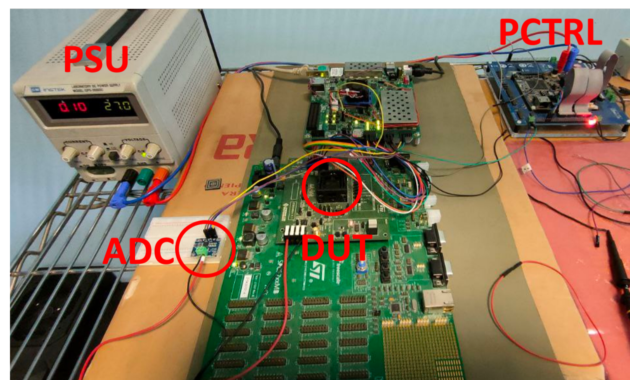



Fig. C.7: Experimental setup for power measurements.