Politecnico di Torino

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Computer Engineering (35^{th}cycle)

New Techniques to Detect and Mitigate Aging Effects in Advanced Semiconductor Technologies

By

Sandro Sartoni

******

Supervisor(s):
Prof. Matteo Sonza Reorda
Ing. Riccardo Cantoro, Co-Supervisor

Doctoral Examination Committee:
Prof. Alberto Bosio, INL - École Centrale de Lyon
Prof. Andrea Calimera, Politecnico di Torino
Prof. Giorgio Di Natale, Grenoble INP
Prof. Maksim Jenihhin, Tallinn University of Technology
Prof. Chrysovalantis Kavousianos, University of Ioannina

Politecnico di Torino
2023

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.

Sandro Sartoni
2023

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

I would like to dedicate this thesis to my loving parents

Acknowledgements

I would like to thank all those who have stood by me and supported me throughout this PhD journey. A special thank to my two supervisors, Matteo Sonza Reorda and Riccardo Cantoro, whose knowledge and assistance have been of paramount importance during the last three years. I would also like to acknowledge Michele Portolan and Lorena Anghel, my two supervisors in Grenoble, who have contributed in making my three months in France really fruitful. Thank you, mom and dad, for always believing in me and supporting me no matter what, I wouldn't be here without you. Thank you Emma for your love and patience, I am so happy to have you by my side.

Abstract

New advanced semiconductor technologies are increasingly adopted in emerging applications, as they provide high computational capabilities together with reduced power consumption. Integrated Circuits (ICs) that employ such semiconductor technologies require complex and sophisticated manufacturing processes and feature advanced transistor designs in a highly dense topology, as they require to work at high frequencies to provide the aforementioned advantages. These technologies, however, introduce new issues as they present higher physical defects rates and a reduced lifespan. Some of these defects can also arise during the device lifetime: in most cases they are related to aging effects and overheating-related issues, making systems more sensitive to degradation than older generations with a strong dependence on the adopted workload. Other defects are also generated due to ElectroMagnetic Interference (EMI) or parasitic effects. As a result, ensuring the correct functioning of circuits manufactured with newer technologies within safety-critical applications is becoming more and more crucial, especially when the expected lifetime of the whole electronic system is in the order of at least one decade, such as in the automotive sector. When dealing with the test (both at the end of manufacturing and in field) of ICs manufactured with the most advanced technologies, it is not possible to keep on relying on traditional fault models anymore, e.g., the stuck-at fault model. For this reason, efforts should be made towards adopting more advanced fault models such as delay faults, i.e., faults affecting the timing behavior of the device under test (DUT), that allow to detect the eventual presence of these newly identified issues. When tackling delay faults, it is customary to work with two fault models, namely transition and path delay faults (TDFs and PDFs, respectively). Such fault models, however, are not as popular and widely adopted as the stuck-at fault (SAF) model when in field testing is required, and solutions for dealing with TDFs and PDFs are not as mature, especially when Software-Based Self-Test (SBST) techniques must be adopted. This PhD thesis aims at defining and validating SBST solutions for the

two aforementioned delay fault models, allowing an effective in-the-field test that can ensure the reliability of the device under test for several years.

This manuscript is organized into three parts. The first part is dedicated to the study and development of solutions to improve the effectiveness of Self-Test Libraries (STLs) for transition delay faults. Transition delay faults share similarities with the more mature SAF model, hence why rather than developing test programs from scratch it is more useful to take already existing SAF oriented STLs and find ways to improve the achieved TDF fault coverage by targeting transition delay faults. Two methodologies are presented, one based on a purely software approach, looking for points in the STLs to insert specific pieces of code in order to improve the observability of the aforementioned set of NO faults. The second one, on the other hand, relies on a mixed hardware and software approach that allows the test engineer to reuse post-silicon debug hardware that is already present inside any System on Chip to monitor flip-flops where fault effects propagate and stop inside the DUT. Advantages and achieved results related to these two approaches are presented and discussed in details in this manuscript, showing how the transition delay fault coverage can be increased with a relatively small overhead.

The second part focuses on the more challenging path delay fault model. Functional SBST solutions for this fault model targeting modern complex CPUs are not available to this day. In addition to that, no EDA tool currently supports functional testing for sequential circuits targeting path delay faults, making it difficult to develop STLs and estimate their effectiveness. For this reason, the first step consists in devising a framework capable of performing functional fault simulations for sequential circuits. Thanks to this framework, the effectiveness of already existing STLs developed for other fault models was studied first, showing that they cannot be immediately employed when working with PDFs, followed next by the detailed presentation of techniques for developing STLs for PDFs. This methodology proves to be highly effective, showing significant improvements with respect to the state of the art.

The third and final part of this manuscript tackles the important topic of aging. As circuits age so does their timing behavior, with portions of the circuit that degrade faster than others depending on the workload, voltage and operating temperature. The changes that ensue from the aging process are reflected, from a timing standpoint, in the evolution of the set of critical paths found within a DUT. As circuits

must operate correctly in the field for several years, this third part devotes to the refinement, automation and application of an aging model previously developed at the University of Grenoble-INP in collaboration with STMicroelectronics to modern in-order complex CPUs, so that their path delay faults can be tested through time. An analysis on how paths evolve in time is provided, together with an estimate of how the test coverage changes and how to ensure that appropriate fault coverage levels are achieved throughout the operative lifetime of the DUT.

Contents

List of Figures

List of Tables

Introduction

Semiconductor companies are developing new advanced technologies that require more complex and sophisticated manufacturing processes. The increased complexity is related to, among other aspects, the transistors' shorter channel length that allows high working frequencies and dense designs. This advantage, however, comes at the price of more frequent physical defects and shorter device lifespan. Integrated Circuits are nowadays more prone to manufacturing defects, process variations, aging effects, ElectroMagnetic Interference, parasitic effects and overheating related issues and are much more sensitive to degrading than older generation devices. When dealing with defects stemming from new semiconductor technologies, static fault models, e.g., the stuck-at fault model, are becoming less and less effective in representing the real defect coverage. This is why dynamic fault models such as delay fault models, i.e., fault models that take into account the timing behavior of the device under test, are becoming more and more important. Several delay models have been defined in literature, the two most important and adopted ones being the transition delay and path delay fault models. Both fault models present advantages and disadvantages, the transition delay fault model — a gate-delay fault model — being easier to manage as it models delay faults affecting single points in a circuit, and the path delay fault model being more accurate as it models delay defects distributed along paths — a series of logic gates connected through interconnections — but harder to manage.

Two main solutions are usually adopted when testing delay faults on an integrated circuit, namely structural and functional tests. Structural tests are based on the adoption of a Design for Testability (DfT) approach that is based on the insertion and/or modification of hardware circuitry within the device under test. This is done in order to increase its *controllability*, i.e., the capability of inducing specific states within the device under test, and *observability*, i.e., the capability of observing

the value on specific nodes of the device under test, and it is usually achieved by operating on the flip-flops in the circuit. Scan chains are an example of the features introduced by DfT-based approaches: by modifying flip-flops of the Design Under Test (DUT) it is possible to scan in test vectors, thanks to the increased controllability, and scan out responses to said vectors, thus enhancing the observability, easing the generation and launch of test procedures. Another popular approach based on DfT techniques is the JTAG standard, that requires the insertion of a boundary scan cell per input/output pin to scan in and out test vectors, together with a TAP controller and additional registers needed to perform the test routine. DfT solutions are well supported by commercial tools, and they enable the generation of high quality tests capable of achieving significant fault coverage figures. Nevertheless, they introduce some drawbacks, too: the additional circuitry impacts the area overhead of the whole SoC in a non-negligible way and degrades timing performances. Power consumption during test procedures can be an issue when adopting DfT solutions, and overtesting, i.e., when circuits are discarded for the presence of faults that will not occur under any functional scenario, or functionally untestable faults [4], may occur. In addition to that, routing of clock signals is complicated due to the insertion of scan circuitry. The other popular solution is known as functional test, that is testing without resorting to additional hardware. A commonly adopted approach when opting for functional test is Software-Based Self-Test (SBST)[5, 6]. SBST relies on the execution of a Self-Test Library (STL) by the CPU usually found within any SoC. The test routine usually unfolds as follows: whenever there is an idle time slot where the device to be tested is not performing any critical operation, the STL is launched in order to record the results produced while testing the device. These results are then compacted into a signature through software algorithms; such signature is finally compared against the golden circuit's one, i.e., the fault-free responses that are expected from the device under test, to look for errors. This approach has the advantage of being cheap, as no additional hardware is required, flexible, as the STL can be split into sub-routines based on how long are the idle slots, thus not impacting the functioning of the circuit to be tested, and fast, given that DfT-based routines typically cannot be executed within the aforementioned idle time slots. Test vectors generated through DfT techniques may cause peaks in power consumption during the testing procedure that do not occur under any functional scenario, due to the fact that internal signals may toggle in a way that cannot be otherwise reproduced, as well as overtesting. All these issues can be avoided with an SBST approach, too. Given that test vectors come in

the form of instructions executed at the functional clock speed, SBST solutions have the additional advantage of performing at-speed tests, a crucial feature for effectively tackling delay faults as even small changes in propagation delays are accounted for. Finally, SBST is suitable whenever the reliability of safety-critical devices must be ensured throughout their operative lifetime, as mandated by standards such as the *ISO26262*. This approach has been proven effective both when testing CPUs [7–18], memories [19–21] and peripherals [22–28], and companies provide STLs for their own products [29–35]. At the same time, generally speaking SBST is not capable of achieving the same high fault coverage figures that are obtained through DfT approaches, mainly due to the fact that there are no automatic methodologies to develop STLs capable of testing the totality of faults. This makes SBST not particularly suited for manufacturing testing.

In addition to that, developing SBST solutions oriented at delay faults presents an additional challenge, due to the fact that they are two-cycle faults (hence, two test vectors are required) and, in the case of path delay faults, the defect is distributed along a path. This is why they are not as effective as those developed for the stuck-at fault model, and in some cases are not even extensively supported by EDA tools (especially for path delay faults). This requires the study, definition and validation of new techniques for delay fault-oriented SBST solutions.

In Part I of this thesis, the focus is on addressing the transition delay fault model. This fault model bears similarities to the stuck-at fault model, which is why existing STLs for stuck-at faults serve as a solid foundation for testing transition delay faults. Rather than concentrating on developing STLs from scratch, this section provides two mechanisms to enhance the effectiveness of available STLs in detecting previously unnoticed transition delay faults. These mechanisms aim to increase fault coverage with minimal additions to the existing codebase. To delve into the topic further, Chapter 1 introduces the transition delay fault model, explores the typical testing techniques employed, and provides an extensive overview of related research in the realm of STL development for transition delay faults, STL hardening, and trace buffers. Subsequently, Chapter 3 and Chapter 4 present and elaborate on the techniques proposed in this PhD thesis for bolstering STLs to enhance transition delay fault coverage.

The approach detailed in Chapter 3 is entirely software-based and involves categorizing not-observed transition delay faults into two groups based on the location

of their effects within the device under test. This categorization enables the implementation of two distinct fault detection mechanisms, one for each group. On the other hand, Chapter 4 introduces two algorithms that leverage the post-silicon debug hardware commonly found in modern System-on-Chips (SoCs) to make the effects of the aforementioned transition delay faults observable. Experimental data from both approaches demonstrates the ability to detect previously unnoticed transition delay faults, resulting in a potential increase in overall fault coverage of up to 20.09% for transition delay faults.

Techniques for developing STLs specifically designed for path delay faults are presented in Part II of this thesis. Unlike the transition delay fault model, path delay faults differ significantly from stuck-at faults, and the methods for developing STLs for this fault model are not as well-established. Furthermore, at the time of writing, no commercially available fault simulation tool can perform sequential fault simulation of path delay faults, which means simulating faults using test vectors generated by test programs without utilizing scan chains. To address this limitation, a path delay fault simulation flow, described in detail in Chapter 7, has been developed from scratch. This flow enables sequential fault simulations of path delay faults on integrated circuits. The fault simulation flow consists of several steps, beginning with the synthesis of the DUT and followed by the extraction of critical paths that can be structurally tested using an Automatic Test Pattern Generation (ATPG) engine. Test vectors, obtained as Value Change Dump (VCD) files through logic simulations of an STL executed on the DUT, are then applied at the combinational level to determine the detectability of path delay faults at the primary and pseudo-primary outputs of the DUT. Subsequently, another top-level simulation propagates the effects of this group of faults throughout the DUT, aiming to observe their effects at primary outputs. Chapter 8, on the other hand, proposes a systematic and automated method for developing STLs specifically tailored to path delay faults. The proposed approach involves first extracting a set of critical paths, which will be considered for the presence of path delay faults. An ATPG is then employed to generate test vectors capable of detecting path delay faults at the combinational level. However, since the ATPG is not aware that these test vectors need to be applied functionally through STLs, functional constraints must be provided to ensure the generated vectors can be replicated through programs. Once the test vector generation step is completed, these vectors need to be converted into instructions that form the final STL. Additionally, it is essential to include ‘store’ instructions to make the fault effects observable at

primary outputs. Experimental data demonstrates that all path delay faults within a 32-bit, in-order 5-stage pipelined RISC-V CPU can be detected using the proposed method.

Part III of this thesis delves into aging phenomena and their impact on modern integrated circuits. Detailed in Chapter 9, the mathematical equations necessary for estimating aging effects on integrated circuits are extensively covered. Subsequently, Chapter 11 outlines an automated aging framework capable of generating aged delays for each cell of a design. This is accomplished by utilizing physical data from the technology library used for synthesis, in conjunction with the fresh (VCD) and Switching Activity Interfile Format (SAIF) files of the circuit to be aged. Due to the unavailability of physical data for every possible aging configuration, linear regression models are employed to interpolate data and provide a close estimation of aged propagation delays. Experimental data reveals an increasing number of critical paths over time, with the introduction of several new ones after the 5-year mark. Consequently, fault coverage figures degrade accordingly. Initially, a 32-bit, in-order 5-stage pipelined RISC-V CPU achieves 100% fault coverage, which gradually declines to slightly over 83% after 10 years.

Part I

Transition Delay Fault Oriented Solutions

Chapter 1

Background

Delay faults were first introduced by Melvin Breuer in his "The Effects of Races, Delays, and Delay Faults on Test Generation" [36] published in 1974. The motivations for introducing this new category of faults stemmed from the fact that, in all sequential circuits, logic operations are synchronized by one or more clock signals with the constraint that each and every signal transition must not occur a minimum amount of time after the clock active edge, defined as *hold time*, and settle a minimum amount of time before the following clock active edge, defined as *setup time*. The well-known stuck-at fault model is described as a fault affecting a node in a way such that it is stuck to a specific logic value, may that be a logic 1 (stuck-at-one) or 0 (stuck-at-0). Such description can also be thought as a transition that takes an infinite amount of time to occur on a line, hence why this is not sufficient to effectively model and detect faults affecting the timing behavior of a sequential circuit. To exemplify this concept, Fig. 1.1 shows a basic circuit comprised of three NAND gates that implements a XOR function. It is assumed that all logic gates have the same propagation delay of one time unit.

Physical defects in circuit, however, can impact their timing behavior, leading to unexpected results. Let us consider, for instance, the waveforms reported in Fig. 1.2 where the two inputs `A` and `B` are reported in green, the correct output `F` in cyan and the delayed output `F_del` in orange. As it can be seen, due to the increased delay, the faulty output does not update its value by the end of the simulation, where it should transition from 1 to 0 as both inputs are now equal. If this simple circuit must operate on a clock basis, the sequential elements that are attached at the output of

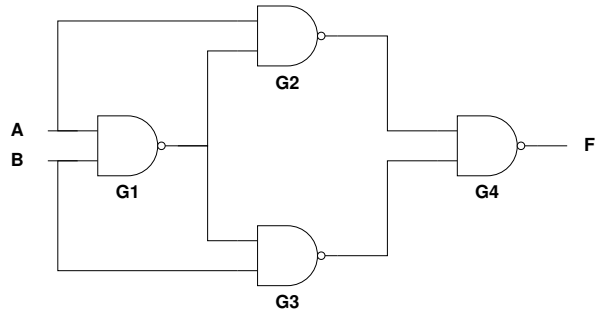


Fig. 1.1 A network of NAND gates implementing a XOR function

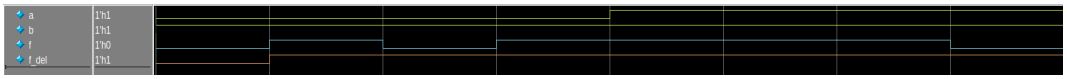


Fig. 1.2 An increased delay causes the circuit's output to not update in time

said circuit might sample the wrong value, leading to an erroneous behavior. It is noted, however, that given enough time the output will eventually be updated to the correct value, differently from the stuck-at fault model where a net is forever tied to a logic value. Given the limitations of the stuck-at fault model, it is obvious that a delay fault model is needed to model this kind of defects. Several delay fault models have been defined ever since, with the two most popular and widely adopted ones being transition delay faults and path delay faults.

In this chapter the transition delay fault model is presented and discussed in Section 1.1, together with techniques on how to test them in Section 1.2. Section 1.3, finally, provides a detailed analysis on related works.

1.1 Transition Delay fault model

The transition delay fault model is one of the two most popular delay fault models. Transition delay faults are described as large additional propagation delays that affect transitions on a single line in a circuit. When a transition delay fault affects rising transitions it is called a slow-to-rise (STR) fault, while faults affecting falling transitions are called slow-to-fall (STF) faults. The increased delay is so large that the propagation delay of all paths containing that line exceeds the clock period, thus making the presence of said fault noticeable. Let us consider, as an example, a simple combinational circuit reported in Fig. 1.3.

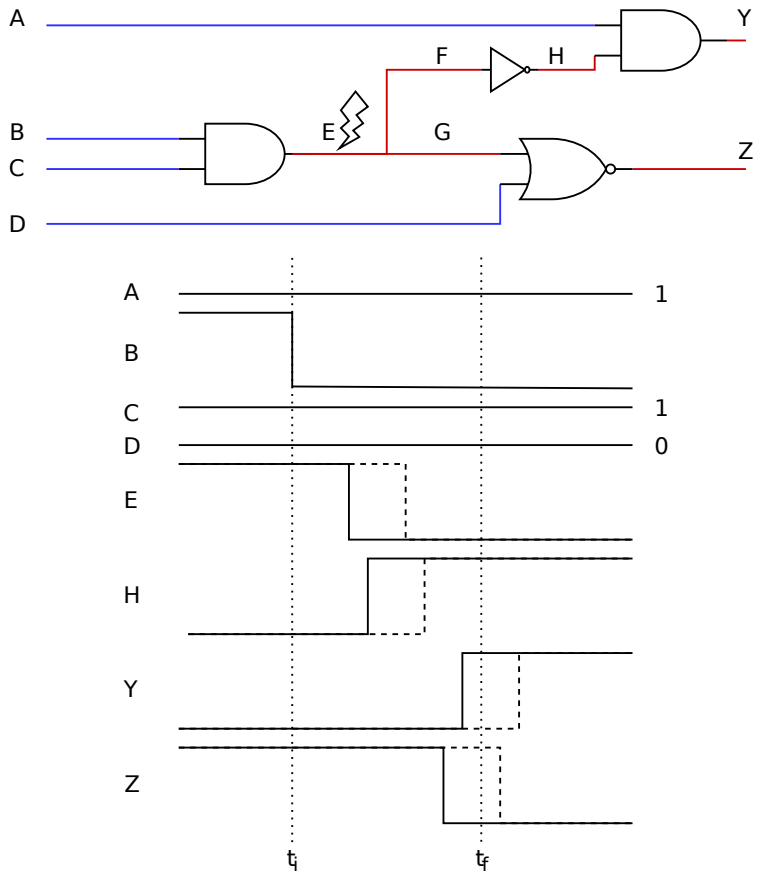


Fig. 1.3 Circuit affected by slow-to-fall fault

In this picture, the line `E` is affected by a STF fault which also affects transitions on lines `F`, `G`, `H` and, ultimately, both outputs `Y` and `Z`. Correct signal transitions are reported with a full line, while delayed transitions are represented with a dashed line. Assuming that the transition on input `B` occurs at the time t_i and any output signal must settle before t_f, the presence of a transition delay fault makes the time required for signal propagation on any path containing line `E` exceeding t_f.

Given the nature of delay faults, testing them requires to apply a pair of test vectors (t_1, t_2), differently from the stuck-at fault model that requires only one test vector per fault. Assuming, for instance, that line E from the previous example is affected by a slow-to-fall fault, the first test vector t_1 must drive the line to a logic 1, followed by a logic 0 in the second test vector t_2, so that the required transition is generated, while other inputs must be set to a non-controlling value so that the examined transition can propagate towards one of the outputs of the circuit. Going

back to Fig. 1.3, an example of test vectors that allows to detect a STF fault on E could be $(1110, 1010)$, but $(0110, 0010)$ is also a suitable choice. The difference among the two pairs is that the first one makes the fault effect observable at both outputs, while the second one propagates the transition to Z, only. Since the extra delay introduced on the line is sufficiently large that any path is affected, both the long and short paths can be chosen to propagate the transition, simplifying the test generation process, although the long path is usually chosen as it allows to detect smaller defects too. Finally, it is noted that testing a slow-to-fall fault implies testing the stuck-at-1 fault on the same line as the second test vector forces a 0 on said line and drives the other inputs so that the fault effect is observed from one output. Similarly, testing a slow-to-rise fault implies testing the correspondent stuck-at-0.

1.2 Techniques for testing transition delay faults

Transition delay fault testing can be performed either through DfT or SBST techniques. This fault model is well supported by EDA tools, also thanks to the similarities it shares with the popular stuck-at fault model. For the sake of simplicity, let us consider a generic implementation of a sequential circuit where the combinational logic has been grouped under one single *combinational block* separated by all flip-flops as in Fig. 1.4.

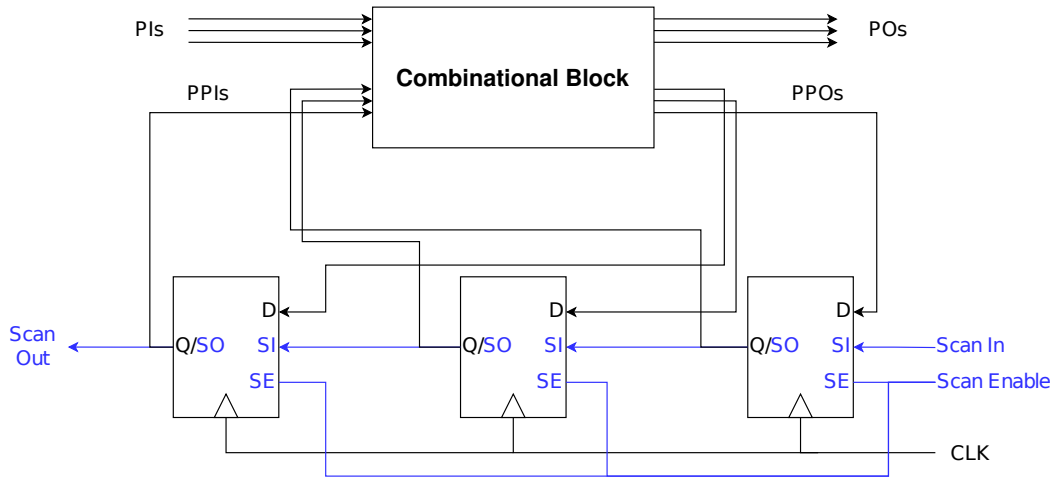


Fig. 1.4 Generic sequential circuit structure with scan chain insertion

In this figure, inputs to the whole circuit, or Primary Inputs (PIs), are directly fed to the combinational block, from where the outputs of the whole circuit, or Primary

Outputs (POs) stem. Two other signal groups are introduced, namely Pseudo-Primary Inputs (PPIs), containing the output of every flip-flop, and Pseudo Primary Outputs (PPOs), the inputs of every flip-flop. In blue, modifications to flip-flops to implement the scan chain functionality, together with two additional input pins, the Scan In (SI) and Scan Enable (SE) signals, and one output pin, the Scan Out (SO) signal, are reported.

Concerning the DfT approach, three of the most notable techniques used to test transition delay faults are Launch on Shift (LOS), Launch on Capture (LOC) and Enhanced Scan Test. All of them require the circuit to be tested to be equipped with scan chains to apply test vectors.

Launch on Shift, also referred to as Skewed Load [37–39], is a testing technique implemented as follows:

1. The first test vector t_1 is fed to the scan chain by driving high the SE signal, while providing the first set of PI values,
2. After t_1 has been scanned in, t_2 is provided by scanning in another bit together with the second set of PI values,
3. POs are observed, and one capture clock cycle is pulsed at nominal speed while driving low the SE pin to latch the PPOs,
4. The content of all PPOs is scanned out while providing the next test vector.

The typical waveforms of a Launch On Shift-based test are reported in Fig. 1.5.

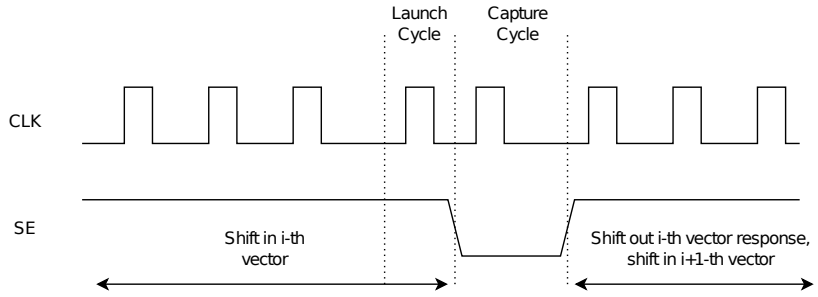


Fig. 1.5 Launch on Shift waveforms[1]

Launch on Capture, also referred to as Broadside test [40, 41], is described as follows:

1. The first test vector t_1 is scanned in by driving high the SE signal, all the while the first set of PI values is applied
2. The SE signal is driven low, and a functional clock cycle is applied to generate the second test vector t_2 from the response of the circuit to the first test vector t_1, while applying the second set of PIs,
3. POs are observed, and another functional clock pulse is given to capture the internal status of the circuit,
4. The internal status of the circuit is scanned out, while providing the new test vector.

Waveforms describing the application of a Launch on Capture-based test are shown in Fig. 1.6.

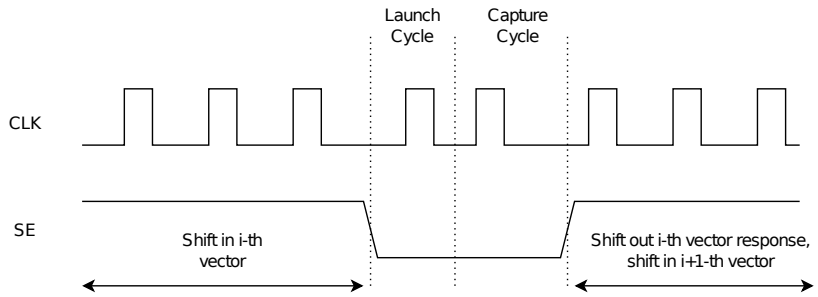


Fig. 1.6 Launch on Capture waveforms[1]

It is noted that, in both Launch On Shift and Launch on Capture, the clock speed at which test vectors are scanned in is slower than the nominal frequency at which the circuit works. Both approaches feature advantages and disadvantages: generating test vectors under the Launch on Capture approach is harder, as the ATPG must take into account that the second test vector is the circuit's internal response to the first test vector, which may lead to lower coverages. Launch On Shift instead requires the second test vector to be a 1-bit shifted version of the first one, with the drawback that the second test vector is applied at speed lower than the nominal one.

Enhanced scan, on the other hand, is the DfT-based approach that allows the highest achievable coverage as it allows to apply any couple of test vectors. In order to do so, however, flip-flops equipped with scan chain circuitry must be further modified to add hold latches together with a hold signal. An enhanced scan test routine unfolds as follows:

1. Shift in the first test vector t_1, while applying the first set of PIs,
2. The hold signal is activated, latching t_1 and giving the chance to load the second test vector t_2,
3. The second set of PIs is applied together with a new application of the hold signal that allows the transition from the first to the second test vector,
4. One capture cycle at nominal speed is executed, so that the internal state is sampled by the flip flops,
5. The internal state of all flip-flops is scanned out.

Hardware modifications introduced in enhanced scan, although easing the test generation process, can thus severely impact the area and timing overheads of the circuit to be tested.

Software-Based Self-Test techniques are a valid alternative to DfT approaches for testing transition delay faults. Differently from their counterpart, such techniques are usually based on two steps, the first one being preparatory to second one that consists in the actual fault simulation. An example of the typical SBST-based fault simulation flow is reported in Fig. 1.7.

For this process to run successfully, two sets of inputs must be provided, namely the post-synthesis netlist of the device to be tested and the technology library with which the netlist has been synthesized, together with the STL intended to test transition delay faults, reported in green in the figure. The first step consists of a logic simulation, usually carried out by means of commercial tools, whose role is to simulate the execution of the suite of test programs by the device under test and record the set of input and outputs applied to it. This set of inputs and outputs represents the stimuli that are then used in the actual fault simulation to check whether they can detect faults, similar to those that the ATPG generates. For this reason, these stimuli are usually recorded into Value Change Dump (VCD) files, which will then be fed to the fault simulator together with the synthesized netlist and technology library to carry out the fault simulation. The fault simulation process, for which several commercial fault simulation tools exist, consists of building an internal model of the device to be tested based on the netlist, followed by the application of input patterns derived from the VCD while observing the POs, looking for differences among the golden, i.e., fault-free, and faulty circuits responses. Once the fault

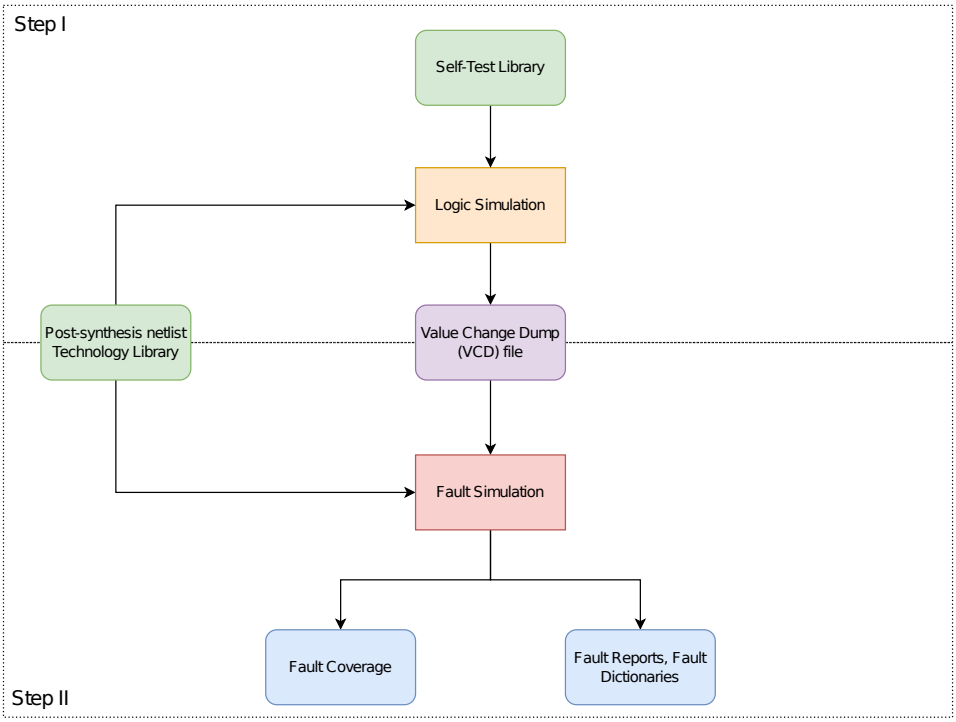


Fig. 1.7 Typical SBST-based fault simulation flow

simulation is concluded, the simulator reports a fault coverage, that is, the percentage of faults detected over the total amount of faults, and a test coverage, evaluated as the percentage of faults detected minus those that are untestable due to the circuit's topology over the total amount of faults. It is noted that in the test coverage figures functionally untestable faults are usually not removed from the active fault list, thus making the test coverage number a worst-case scenario of the STL's effectiveness. In addition to that, fault simulation tools can also report diagnostic data in the form of fault reports, outlining what faults have been detected and what not, or fault dictionaries, useful to understand at what time faults have been detected and, in case they were not, where their effects propagated and stopped inside the device under test.

1.3 Related works

Several works have been proposed in literature on the topic of transition delay fault testing, both resorting to DfT and functional approaches. In this PhD thesis, however,

functional SBST solutions are investigated: for this reason, a detailed description of related works in this area is provided. For the sake of completeness, however, works [42–47] present several techniques aimed at improving launch on shift and launch on capture methodologies for transition delay faults. In the following, related works are divided into three main areas. Section 1.3.1 focuses on works concerning the generation of Self-Test Libraries for transition delay faults, while Section 1.3.2 tackles the topic of hardening, i.e., improvement and refinement of test programs that were developed for other purposes to make them more efficient in terms of fault coverages. Finally, given that one of the approaches proposed in this PhD thesis resorts to the usage of post-silicon debug logic as trace buffers, Section 1.3.3 describes works focusing on the adoption of such circuitry.

1.3.1 STL development for transition delay faults

The development of test programs for delay faults through an SBST approach is a well-studied topic in academia, with several articles describing related methodologies [9, 11, 13]. When dealing with the generation of STLs for transition delay faults, the article [8] describes how to generate test programs for RISC processors without knowing details on their internal implementation. This feature is achieved by first dividing the processor under test into functional blocks, referred to as Modules Under Test (MUT), and representing them as control and data parts separately. Following, two high-level fault models are defined for the control and data parts, namely control and data faults. Data faults are tested through data manipulation functions, bit-wise pseudo-exhausting tests, that allow to achieve high fault coverage of a broad class of faults without the need of knowing implementation details of the module to be tested. As for control faults, the paper introduces a functional fault model that is able to cover a large amount of low-level structural faults, and also a set of traditional high-level functional fault models that are typically used in testing of memories. This approach is initially defined for the stuck-at fault model and then extended to the transition delay fault model, leveraging on the fact that the two fault models share similarities, as highlighted in Section 1.1. Experimental results on the execution and forwarding unit of the MiniMIPS core show high coverages, both when targeting stuck-at and transition delay faults. Authors in [7] tackle the problem of generating test programs for out-of-order superscalar processors. In an in-order processor, since instructions go through the pipeline one after the other,

if one instruction does not have its operands ready or its execution unit free, the processor stalls the instruction and all the following ones until the first one can proceed. An out-of-order (OOO) superscalar processor, while still allowing fetch, decode and writeback (commmit) in order, differs from an in-order processor in the sense that as soon as one instruction is ready to be issued it is executed regardless of other instructions that may come before. This architectural implementation allows for optimization in performance, but introduces challenges as delay faults require specific vector pairs to be applied which in OOO processors may not be replicated since it is not known in advance whether the processor is capable of executing two instructions one after the other. The authors propose a methodology based on the adoption of conditional branch instructions in order to bring the dispatch buffer into a known state and thus apply the test instructions in the intended order. The rationale behind this is that in most cases processors adopt branch prediction techniques so that the throughput of the processor's instruction flow is increased. Through branch prediction, the processor speculates on the outcome of the branch, preloading a set of instructions to be executed in case the outcome matches the prediction, while in the opposite case the pipeline is flushed and new instructions are loaded. Exploiting this particular behavior, the authors take advantage of the misprediction of branches to bring the dispatch buffer into a known state from which the test is launched. Test instructions are obtained through ATPG to which functional constraints are applied. These constraints are obtained by means of Verilog wrappers to be applied to module to be tested, which restrict input values to those that can be obtained by using instructions from the instruction set architecture. This methodology has been defined for execution modules defined as *simple_alu* and *complex_alu*, obtaining high fault coverage figures, the lower being a 98.66% on the *complex_alu* module.

The paper [48] presents a reinforcement learning-based test program generation technique for TDFs for processor cores. This approach makes use of a reinforcement learning algorithm to produce simulation traces geared for fault sensitization, that is, for exciting faults hidden in the inner logic of the processor under test and propagate their effects towards registers, thus allowing to integrate this approach with others that tackle the problem of observing the faulty values stored in registers at primary outputs. The authors of this paper generate fault sensitizing states to be reached in order to excite the faults to be tested by means of a constrained ATPG, and then limit the reinforcement learning search space by selecting only those registers that are relevant for the fault sensitizing states and those that carry control signals. Once the

fault sensitizing states have been screened out for each transition delay fault through the adoption of a constrained ATPG, the goal space and the reinforcement learning state encoding of the processor are determined, followed by the execution of the reinforcement learning algorithm. This is all done with the goal of finally generating the test program, with the possibility of deploying functional constraint extraction to obtain constraints to apply to the ATPG for the next iteration. This approach is applied on a MIPS32 core, with results achieving a 94.94% on the whole core.

Works [49, 50] focus on the generation of temperature-aware SBST solutions for transition delay faults. The idea behind these papers is that delay testing under high temperatures is a critical factor in ensuring the reliability of computer systems, and SBST techniques can be one of the most suitable methods to carry out this task, provided that they are capable of creating and maintaining an appropriate high test temperature throughout the execution of the test program. [49] achieves this task by first adopting an Automatic Test Instruction Generator (ATIG) to which functional constraints are applied. Thanks to this setup, authors produce test programs capable of reaching high transition delay fault coverages on the ALU of processor cores. Experimental results, in fact, show a 97.91% fault coverage on the ALU of a MiniMIPS core. These test programs, however, still need to be modified so that they can heat up the processor core, mainly by trying to reduce any stall which would hinder the switching activity of signals inside the CPU, as the more the signals toggle the higher the power consumption is. Two methods are proposed, either via loop unrolling or by exciting cache misses in load and store instructions. Experimental results show that, besides the high fault coverages, the dissipated power can be increased in a significant way by either implementing one of the two aforementioned strategies on its own or by using them both, at the cost however of an increased code size and execution time. [50] makes use of a Bounded Model Checking (BMC) approach to generate a set of sequential constraints that are used in conjunction with an ATPG to generate functional test patterns for delay faults through which test programs can be autonomously generated. It then follows the same principles as in [49] for making the test program suitable to heat up the circuit under test, producing high fault coverages at test temperatures that fall within safe ranges.

Finally, the work in [22] describes a generic approach to STL development for peripherals embedded in modern System on Chips. The methodology proposed in [22] explains how an STL can be developed starting by a series of small programs that are capable of accessing, both in terms of reading and writing when available,

all peripheral registers and activate all of its functionalities, may them involve the transmission and reception of packets in several working configurations. Each code section usually follows a specific structure, having first a setup or configuration phase that is used to configure the peripheral in a specific working mode, and an operational phase where the peripheral operates under the intended scenario. This article also puts an emphasis on how the targeted fault model impacts the stimuli required to excite the faults and propagate their fault effects, for example, stuck-at faults require one test vector while delay faults require pairs of test vectors. In addition, code written for verification of the device under test can be used as a starting point, but it still needs modifications as the observation of fault effects requires more pervasive data sampling operations. This approach is then validated on a System Power Management Interface (SPSMI) peripheral, dividing the STL into six macros: a *Reset, write 0/1 and read 0/1* on all Control Status Registers, a *Mailboxes* test where all FIFO buffers are tested for rising and falling transitions, a *Bus Access Request* test, an *All Commands* test, going for all commands provided by the peripheral, an *AHB Control Interface* test, thus targeting the AHB interface, and finally a *Counters* test, mainly used for the protocol timing management. Experimental data shows how it is possible to achieve a test coverage figures of 87.83% transition delay faults showing how SBST can be adopted in conjunction with other testing methodologies even after the circuit has been manufactured.

All the works presented in this section tackle the problem of generating test programs for transition delay faults, specifically, by following very diverse approaches. The achieved results are quite high in every case, showing that testing transition delay faults through functional means can lead to satisfactory results and thus validating the usage of SBST techniques for this fault model. Generating test programs starting from scratch, however, is a complex task, and it requires non-negligible time and effort from test engineers that scales with the complexity of the processor to be tested. For instance, the adoption of reinforcement learning algorithms as described in [48] might not be particularly effective when tackling high-complexity cores. The increase in code size and execution time reported in [49, 50] might not be negligible thus leading to limitations depending on the properties of the device to be tested and the main application in which it works. Other works focus more on the execution unit or ALU portions of the processor to be tested, while faults that may arise from other units may still significantly impact the processor and are harder to tests, e.g., those stemming from the control unit. For this reason, the techniques presented in

this PhD thesis focus on how to improve already existing test programs to improve transition delay fault coverages without focusing on single functional units of the device under test and with little timing and size overhead.

1.3.2 Self-Test Libraries hardening

Improving available test programs to achieve higher fault coverages is a thoroughly investigated issue, with several works on this[51–53, 18]. The work in [51] presents a methodology to generate test programs intended for online testing starting from verification-oriented programs. To do that, the authors introduce a two-step approach, namely an Autonomous Systems Safe Faults Classification followed by an On-Line Test Set Generator. The first step aims at generating a fault list from which all Safe Faults, that is, all faults that affect areas that never toggle while running the software application are removed. The idea behind this is that even though a portion of the device under test's logic is affected by faults, if that logic is never used during the operative lifetime the fault cannot affect the system into which the DUT is embedded. This definition is, therefore, strictly bound to the software application, and the list of Safe Faults can thus change from application to another. This task is carried out by recording the switching activity of internal nets in the combinational portion of the device under test, also bearing into mind that there might be signals that must be analyzed carefully, e.g., those handling interrupts from GPIOs or those belonging to status registers, as they can toggle in an unexpected way thus not allowing to constraint them to fixed values. Once this is done, faults belonging to the never toggling signals are identified and removed, so that untestable and undetectable faults are not taken into account in the test generation phase. The second step involves the use of the On-Line Test Set Generator, with the aim of taking code snippets written for verification and turning them into test programs. The verification program provides the skeleton for the final test program and is expanded by adding portions of code, namely a *Stack frame creation*, saving the content of the registers before launching the test routine, a *Register initialization* portion, a *Signature Store* section where the register values affected by errors due to faults are compacted into a signature and stored in memory and finally a *Stack frame destruction*, with the goal of recreating the context prior to the execution of the test program. The authors tested this approach on two modules of a RISCV core embedded in a nanodrone, namely the multiplier and hardware loop control. Experimental results show a stuck-at fault

coverage of 99.26% and 80.41% on the multiplier and hardware control loop module, respectively.

Authors in [52, 53] present an approach based on a tool implementing High-Level Decision Diagrams (HLDDs) used to model microprocessors and their faults, paired with a test generator that produces the final test program starting from already available templates stored in the assembly code library. The High-Level Decision Diagram of the processor under test is produced starting from the ISA of the core to be tested, that has to be appropriately described into specific file formats so that the tool can process it. The user can select how many instructions from the ISA to provide the tool for the HLDD generation, e.g., if only the ALU should be tested then ALU intructions only can be provided. The generated test program can be divided into two parts, the first one being the initialization, where registers are written with specific data, followed by the actual test part. Given these properties, the authors report that the test generation process is heavily influenced by the modeling level implemented in the previous steps: the more instructions can be fed to the tool the better the representation; moreover, specific behaviors of the CPU might not be deducted from the ISA and simple set of instructions could thus not be enough to test them.

Authors in [54] present a technique based on evolutionary algorithms, in the form of Genetic Programming, paired with a hardware-accelerated fault simulation process through which already existing test suites are improved by adding new content to them. The proposed system architecture makes use of a Fault Manager, that has the task of generating the target fault list and supervising the automatic test program generation process, and an Automatic Test Program Generator. The Automatic Test Program Generator is based on μGP, an evolutionary approach for generating programs based on the genetic programming paradigm. With this approach, μGP cultivates a population of candidate test programs. Once generated, each program is simulated with an external tool and then the feedback, i.e., the fault coverage, is used to improve the next generation of candidate test programs. The simulator is+ based on a hardware accelerator that is composed of three modules, namely an FPGA, a PCI bus and a host computer. Through this approach it is possible to increase an initial stuck-at fault coverage of 59% up to a final 99.38% fault coverage on a pipelined SPARCv8 core.

The techniques I worked on regarding STL hardening for transition delay faults have been presented in articles [18, 55] and will be thoroughly detailed in Chapter 3. In summary, all the approaches presented in this section show how already developed programs or code snippets can be improved in order to increase stuck-at fault coverages. Techniques based on HLDDs as in [52, 53], however, may prove hard to adopt when applied to the entire circuiit of complex pipelined CPUs. Moreover, the stuck-at fault coverage on the Integer Unit module of the SPARC core adopted in these works is still low, mainly due to the fact that not every instruction using the Integer Unit was considered when building the HLDD model, and extensions for the model to take into account all instructions needs further research. The approach shown in [54], on the other hand, needs 26 hours to complete, and makes the original test program size 1.4 times larger, posing the question of whether the new test program can fit into the non-volatile memory of the device under test. The work in [51] achieves good results, but the approaches developed in this PhD thesis do not require to heavily modify the original program and focus on the whole CPU. Finally, no work focuses on techniques for hardening STLs for delay faults, specifically.

1.3.3 Trace Buffers

Several works in literature examine in-depth the topic of adopting and optimizing the usage of post-silicon debug logic[56–59]. The authors in [56] introduce a technique for capturing debug data when errors are known to be present by means of a three debug approach, thus overcoming the trace buffers limited capacity by discarding data that is known to be error free. The first debug session is used to estimate the error rate which is then used to extrapolate the maximum observation window size. In the second debug session, during the clock cycles in the window range identified in the previous step, a 2-D compaction algorithm is used to identify a set of suspect clock cycles, where errors can occur. The 2-D algorithm is based on the usage of Multiple-Input Signature-Registers and cycling registers, whose signatures are intersected to extrapolate the aforementioned set of suspect clock cycles. The third and final debug session debug data is captured in the trace buffers during the previously identified suspect clock cycles. The article also describes the hardware architecture of the debug module. Data shows that through the three debug sessions the observation window of trace buffers can be increased by up to two orders of magnitude. The work in [57] is an improvement of the one in [56], aiming at defining

a systematic methodology to identify a set of suspect clock cycles so that erroneous data only is stored inside the trace buffers. The proposed approach is based on a two data debug session, removing one with respect to the previous work. The first session is used to extract the set of suspect clock cycle obtained by means of a 2 dimensional (2-D) compaction algorithm, resorting to an intersection the failing signatures of a MISR and two Cycling Registers followed by the store of such intersection into the trace buffer. Prior to the second debug session, tag bits are generated and stored on the trace buffer with the the goal of facilitating the process of debug data capture. Finally, with the second debug session, final debug data is captured. The article explains in details the hardware architecture used for this approach, and as detailed in the experimental results, it achieves fine-grained error localization with an expansion in temporal visibility.

In the paper [58], authors aim at reducing debug time by proposing an on-chip error detection method capable of increasing the trace buffer capacity, which is usually limited, by identifying time windows in which errors are present. This is done in order to reduce the number of debug sessions by selectively capturing debug data. The article focuses on three main aspects. The first is a technique on how to reuse empty portions of trace buffers to reduce debug time with a limited hardware overhead. Pre-calculated golden responses of the circuit to be tested is stored in trace buffers unused area, allowing for a real time analysis of debug data. The second aspect is a new compaction technique, so that the debug module can compress erroneous intervals in an efficient way. The final aspect is an architectural feature that makes use of Multiple-Input Signature-Registers to perform the compaction mechanism required by the proposed on-chip method. Thanks to this approach, debug time is reduced, with a limited hardware overhead.

Finally, [59] tackles the issue of trace buffers overflowing. Trace buffers are largely adopted in post-silicon validation but usually present a limited size, hence the risk of frequent overflowing. The problem associated to trace buffers overflow is that the chip has to be stalled so that the state data that is stored inside the buffers can be transferred outside of the chip. This obviously hinders an efficient debug procedure. To mitigate this problem, the authors propose a technique to minimize the amount of stalls based on several points. First, they introduce a specification language, called Storage Specification Language (SSL), through which the validation engineer can signal what information must be saved and what can be discarded from the traces to generate summaries. Next, they define techniques to perform spatial

and temporal summaries of traces, i.e., techniques that allow to filter out irrelevant signals by removing unneeded fields or discarding information that is defined as not useful after examining compliance to the specified temporal property. Finally, they present hardware design approaches to implement the proposed spatial and temporal summarizer. Results show how it is possible to achieve a reduction in stalls with a relatively low area overhead. My contribution in this specific field have been presented in [60] and will be described in details in Chapter 4.

The related works show how post-silicon debug logic can be used in an effective way in the validation step on the actual hardware to capture the presence of errors. In my contribution, this hardware is reused for testing purposes, looking for signals where hard-to-observe transition delay faults propagate and stop. This allows for an increase in transition delay fault coverage without area overhead, as only a limited set of signals must be observed as opposed to other applications where trace buffers are at least tens of kilobytes wide.

Chapter 2

Main Contributions

Between the two main delay fault models, transition delay faults are the most adopted fault model and the one for which solutions are more mature, both when looking at DfT and SBST approaches. Most of the approaches in literature focus on the issue of generating from scratch Self-Test Libraries for transition delay faults, and usually mainly tackle fault stemming from the execution unit of processor cores, as thoroughly detailed in Section 1.3.1. By studying and evaluating the solutions available in literature, it is possible to outline two prerequisites that the work related to TDFs presented in this thesis should have:

- Rather than working on another method to develop transition delay fault-oriented STLs, developing techniques to increase the fault coverage of already existing STLs with a minimal overhead is of interest,
- Such techniques must work for faults stemming from the entirety of the device under test.

Improving the effectiveness of STLs for transition delay faults means finding ways for detecting faults that have not been detected by the test program suite. This category of faults, however, can be quite heterogeneous as there can be different reasons why a fault has not been detected in the fault simulation process. As an example, the STL may have been able to excite the location where the fault is present but then fail in propagating the fault effect to any of the observation points, usually POs, or it might have failed in even controlling the fault location, thus not exciting the fault in the first place. Moreover, there is the issue of identifying Functionally

Untestable Faults, that cannot be tested in any way by means of SBST solutions. Faults that were excited but whose effects were not propagated to POs are usually referred to as not-observed (NO) faults, while faults that were not excited in the first place are usually referred to as not-controlled (NC) faults.

Given the absence of solutions for this specific topic, in this PhD thesis the focus will be on not-observed transition delay faults as they usually require the least amount of modifications to be imparted to the already available STL thus allowing for a good tradeoff in terms of amount of faults recovered versus code overhead. As a consequence, any increase in fault coverage is achieved by detecting not-observed faults, only. Test programs enhancement may, as a side effect, cause the detection of not-excited transition delay faults, too, even though they are not the focuse of this thesis.

In order to achieve such results, two works are presented in this PhD thesis, one consisting in a purely software solution capable of easily enhancing the fault coverage with a small increase in code size, the other being a hybrid approach that leverages the post-silicon debug hardware that is already present inside modern System on Chips to observe a subset of flip-flops where effects of not-observed transition delay faults propagate and stop. For these two proposed methodologies to work, no minimum fault coverage from STLs is required, with the exception of the constraint of having many not-observed transition delay faults. This constraint, in practice, is usually satisfied by the large majority of STLs, as it will be highlighted in the following chapter and shown in [18].

Chapter 3

STL hardening techniques for transition delay faults

In this chapter, a systematic methodology for improving STLs previously developed for the stuck-at fault model is presented. The reasons for picking STLs developed for stuck-at faults as the starting point are:

1. methodologies for developing effective SAF-oriented Self-Test Libraries are more mature and capable of achieving high fault coverage when targeting faults from the entirety of the device under test,
2. companies systematically develop STLs for stuck-at faults, thus already having a starting point on top of which the presented methodology can be applied,
3. although being two different fault models, stuck-at and transition delay faults share similarities in the way they are tested, as shown in Section 1.1, further validating the choice of selecting STLs for stuck-at faults.

3.1 Proposed Approach

The proposed approach aims at identifying code regions in the test program where to insert a specific instruction block to increase transition delay fault coverage in complex pipelined processor cores. Once defined the goal, we can formulate the problem as (i) understanding where the code needs to be improved and (ii) finding

a suitable set of instructions that can propagate the effect of NO faults with the minimum overhead possible. Given that test vectors come from the execution of instructions of the test program, understanding where the code should be enhanced is a matter of knowing what instructions cause the fault to be excited but fail in propagating the value to the primary output. To solve this issue, the first step has then been decoupled into two sub-steps as follows:

- Find the time instant at which a fault effect stopped inside the processor under test
- Find what instructions were being executing in a surrounding of that time instant.

By intersecting these two information, one can retrieve the instruction that caused the fault to be not observed, also defining a region of code within which the additional testing instructions can be placed before the faulty value is overwritten in the CPU. A broad overview of the test flow I devised to tackle this problem is outlined in Fig. 3.1.

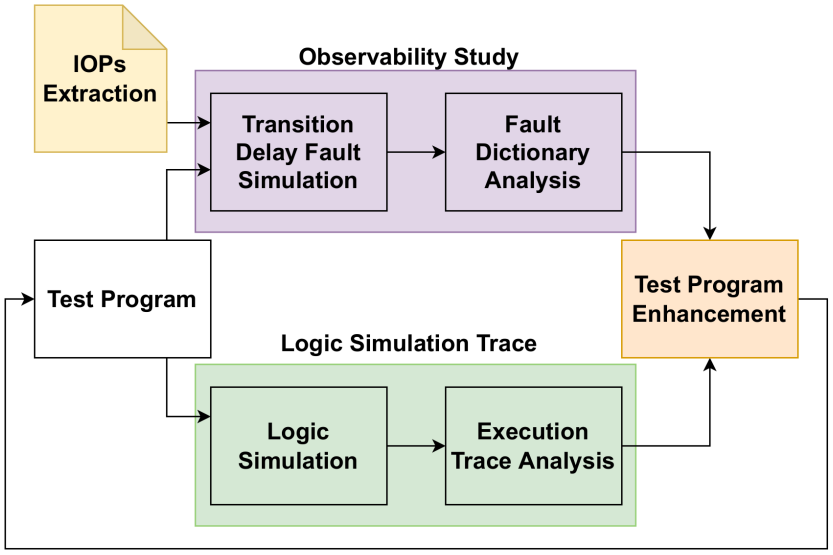


Fig. 3.1 Proposed test flow

Starting from the white block, i.e., the original test program, the test flow articulates into four main processes, namely:

1. Internal Observation Points (IOPs) Extraction: extract a list of signals where faults effect propagate and stop
2. Observability Study: analyse fault simulation data to determine at what time instant fault effects stopped in the DUT
3. Logic Simulation Trace: map instructions with their execution time inside the core
4. Test Program Enhancement: apply a set of techniques to improve test programs based on the information gathered in the previous steps.

In the following, all four processes are described in details.

3.1.1 Internal Observation Points Extraction

When dealing with not-observed transition delay faults, the very first issue to address is understanding where the fault effects propagate and stop. Since this technique is to be applied to processor cores, we can restrict our analysis to the locations within a CPU that can be reached by faults' effects. Moreover, the assumption is that faults originate from nets in the combinational logic and their effects propagate from there towards other sites in the CPU.

Given this premise, an erroneous value stemmed from the presence of a not-observed fault can evolve in three possible ways:

- The value reaches a register that can be directly controlled through instructions from the Instruction Set Architecture,
- The value reaches a register that cannot be directly controlled through instructions from the Instruction Set Architecture,
- The value does not reach any register, and gets stuck somewhere into the combinational logic.

In this PhD thesis, I focus on the first two groups of faults, as they lend themselves to be especially suited for the proposed methodology. The set of faults whose effects reach registers that can be directly controlled through instructions are called User

Accessible Register faults, and the set of signals belonging to such registers are called User Accessible Registers (UARs). An example of UARs are all registers belonging to the register file or control status registers since there are instructions that allow to directly work with such registers, e.g., *load* and *store* instructions allow to read and write into registers as needed. Fig. 3.2 shows an example of UARs in a basic implementation of a pipelined RISC core.

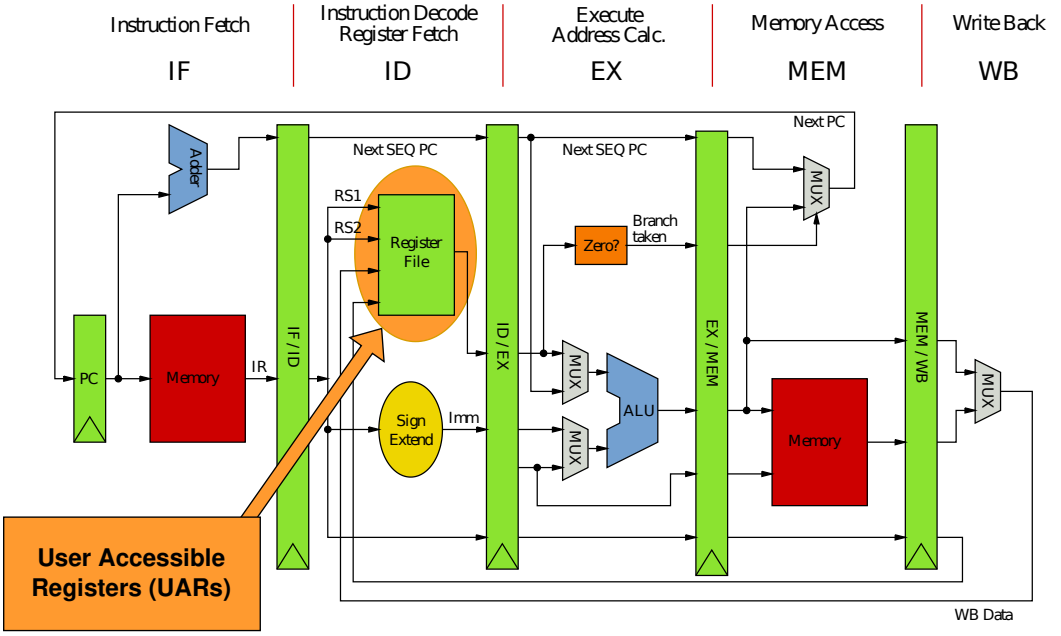


Fig. 3.2 Example of User Accessible Registers in a basic core implementation

The set of faults whose effects reach register that cannot be directly controlled through instructions are called Hidden Register faults, and the set of signals belonging to such registers are called Hidden Registers (HRs). An example of HRs are all registers that can be found in pipelines in every modern pipelined processor core, since no instruction can directly establish the content of such registers, not allowing to directly read from them. Fig. 3.3 shows an example of HRs in a basic implementation of a pipelined RISC core.

Special emphasis is put on the distinction among the two categories of faults because in practice they require different techniques to propagate the faulty value they store towards primary output, as UAR faults tend to be easier in principle to test with respect to HR faults. Having these fault categories introduced and explained, the IOPs Extraction process consists in defining the list of signals belonging to the User Accessible Register and hidden register groups, and extracting their exact name from

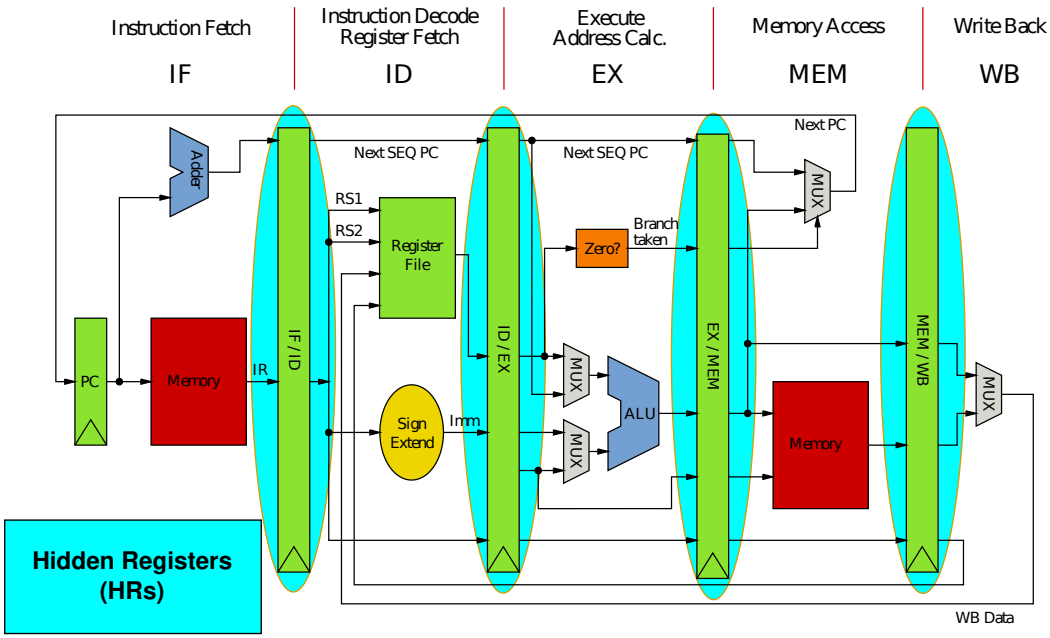


Fig. 3.3 Example of Hidden Registers in a basic core implementation

the post-synthesis level netlist. Performing this task requires the knowledge of the processor's internal structure and its Instruction Set Architecture, considering how tightly connected are these two concepts and the faults' definition previously given. The algorithm used to extract User Accessible Registers and Hidden Registers is quite straightforward, and consists of first extracting all sequential elements, flip-flops and latches, and understanding whether they belong to UARs or HRs based on the functionality implemented by the register they belong to. A graphic representation of such algorithm is presented in Fig. 3.4.

Once the list of UARs and HRs is available, it is possible to move on to the next steps.

3.1.2 Observability Study

Once information on what constitutes User Accessible Registers and what constitutes hidden registers is available, it is possible to start with the Observability Study step. The aim of this process is to give insights on whether a not-observed fault belongs to the UAR or HR category, so that appropriate strategies can be adopted at the STL enhancement level.

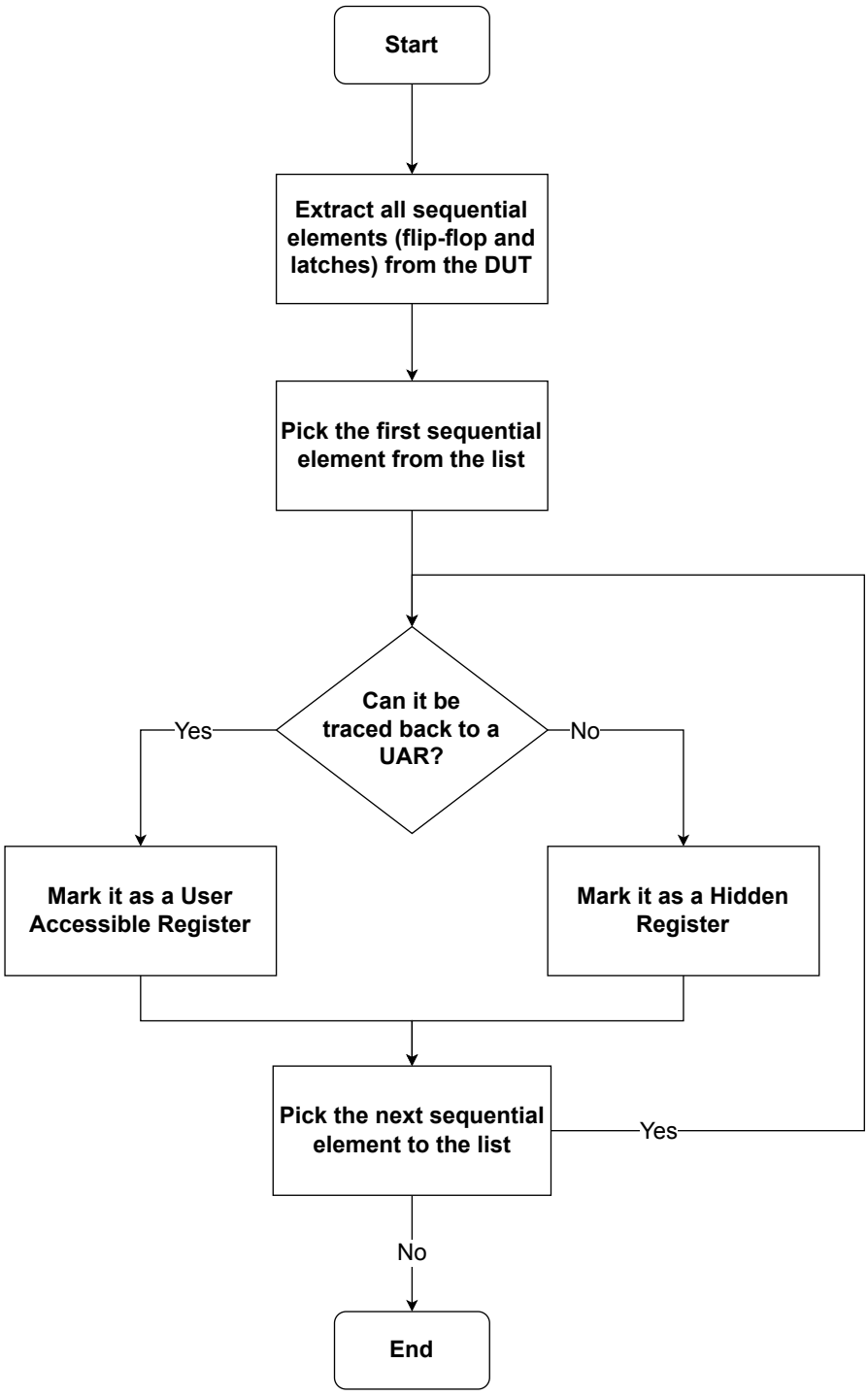


Fig. 3.4 Internal Observation Point extraction algorithm

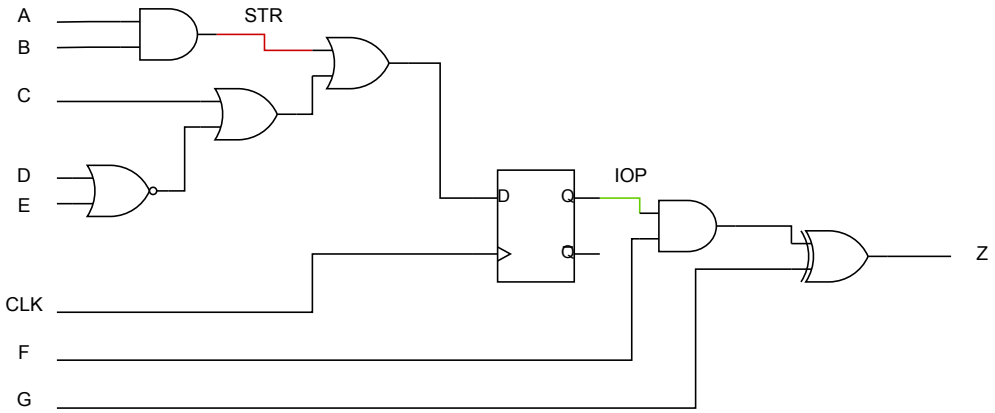


Fig. 3.5 Circuit with internal observation point

The Observability Study consists of two sub-processes, namely a fault simulation followed by an analysis of data produced while performing the simulation. The fault simulation follows the generic approach detailed in Section 1.2, that is, it requires a list of input stimuli stemming from the execution of the STL on the processor under test that will be used while performing the actual fault simulation, so that test vectors deriving from instructions from the test suite can be used to test transition delay faults. In addition to this, however, the information on internal observation point must be applied, so that reports produced after the fault simulation can show what faults belong to what group. In order for the fault simulation tool to acquire the required information on IOPs, the concept of *promotion table* is introduced.

When performing a fault simulation, labels are assigned to faults in order to reflect their status and give the test engineer feedback on the test's effectiveness. Specific names of labels may change from tool to tool, but overall some labels can be found in most commercial tools. For instance, some commonly found labels are those that identify detected faults `DT`, potential detected faults `PT`, not-observed faults `NO`, not-controlled faults `NC` or untestable faults `UN`. However, the test engineer may wish to add other labels that allow to understand whether a fault passed through specific points in the device, so that additional diagnostic information can be gathered. Still, defining new labels is not enough to solve the problem, as it is also necessary to provide the ranking, i.e., the priority list, of all labels. To exemplify the concept, let us consider a simplified example shown in Fig. 3.5.

In this circuit we assume the presence of a slow-to-rise fault on the red line, and define an internal observation point at the output of the flip-flop on the green line. Let

us consider the case in which the STR fault effect reaches the internal observation point at t_1, is detected at the primary output, in this case Z, at time t_2 and is newly observed at the internal observation point at time t_3, where $t_1 < t_2 < t_3$. We assign to the event of the fault being observed at the internal observation point the label `F1`, while the label associated to the fault being detected is the usual `DT`. Based on these assumptions, it would appear that the considered STR fault would be marked as `F1` at time t_1, followed by `DT` at time t_2 and again `F1` at time t_3. This, however, results in an incorrect outcome: once the fault is detected at the primary output, unless differently specified, it should be dropped by the active fault list and no further change in its label must occur, while in the example its status is "downgraded" from being detected to being observed at an internal observation point. The first transition from being observed at an internal observation point to being detected, instead, is perfectly legal, and it shows how the fault effect, once excited, has been propagated at the primary output. From the discussion above it emerges how, together with a set of user-defined labels, it is important to provide a ranking so that fault statuses can evolve in one direction but not in the other. It is noted that creating and assigning labels to points in the circuit does not require the modification of the DUT's hardware as they are a mechanism introduced by commercial fault simulation tools that is simply attached to the netlist in specific points.

A promotion table is a tool introduced by some commercial fault simulation tools that implements the functionalities described above, i.e., it allows to create a set of user-defined labels together with the ranking, so that fault statuses can promote in a coherent way. When working with in-order pipelined processor cores, it is thus possible to define a set of labels as follows:

1. `PP`: a fault effect has been potentially observed at one of the registers of the fetch stage pipeline registers,
2. `FP`: a fault effect has been observed at one of the registers of the fetch stage pipeline registers,
3. `PD`: a fault effect has been potentially observed at one of the registers of the decode pipeline registers,
4. `FD`: a fault effect has been observed at one of the registers of the decode pipeline registers,

5. PE: a fault effect has been potentially observed at one of the registers of the execution stage pipeline registers,
6. FE: a fault effect has been observed at one of the registers of the execution stage pipeline registers,
7. PL: a fault effect has been potentially observed at one of the registers of the memory stage pipeline registers,
8. LS: a fault effect has been observed at one of the registers of the memory stage pipeline registers,
9. PS: a fault effect has been potentially observed at any UAR not in the register file, e.g., control status registers,
10. RS: a fault effect has been observed at any UAR not in the register file, e.g., control status registers,
11. PR: a fault effect has been potentially observed at one of the registers of the register file,
12. RE: a fault effect has been observed at one of the registers of the register file.

It is noted that potentially observed differentiates from observed in the sense that when an effect is potentially observed we are seeing a undefined logic value instead of a logic 1 or 0, e.g., X/1 or X/0, while observed means we are seeing the opposite logic value that we would expect, e.g., 0/1 or 1/0. This set of fault labels is integrated with default labels from the commercial fault simulation tool in a way such that they sit in between the class of not detected faults (NO, NC) and detected faults (PT, DT). Moreover, this labels list has a descending increasing priority, i.e., marking a fault RE can only lead to a fault being eventually detected but it cannot go back to being detected at a pipeline register. The reason for this is that UAR faults tend to be easier to recover with software techniques than HR faults, hence why they have a higher ranking. Together with the promotion table, it is also necessary to provide a strobe list, i.e., a list of internal observation points associated to the user-defined labels. For example, the strobe list will contain all registers belonging to the fetch pipeline registers that are associated with the PP and FP labels, so that the fault simulator tool knows what internal observation points are connected to user labels.

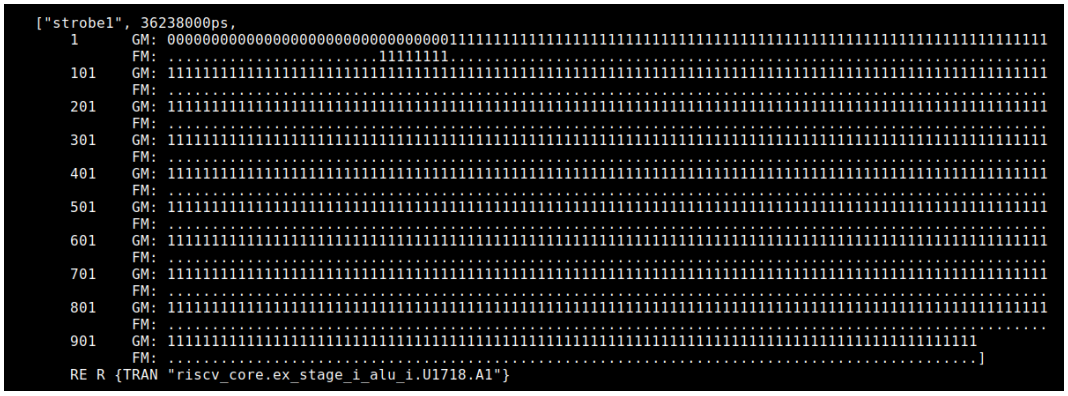


Fig. 3.6 Fault dictionary snippet

Having defined all these additions, the transition delay fault simulation can thus be launched. The information on IOPs, tied together with the data produced while performing the fault simulation, is combined in a special report that is produced at the end of the fault simulation process along the fault and test coverage figures that is called fault dictionary. The fault dictionary is a text file that contains data on where the fault effects have been observed together with their label and time at which the effect reached that internal observation point. The fault dictionary internal structure is reported in Fig. 3.6.

Looking at the figure, three main points can be extracted from the reported fault dictionary entry, namely:

- The time instant at which the fault effect was observed, in this case 36238000ps,
- The internal observation point reached by the fault. The fault dictionary reports the `RE` label, hence we know it is a register from the register file, and provides a strobe list (in this case strobe1) that has an ordered set of signals. The numbered table is interpreted as follows: for every couple of lines, the first one starts with a numeric offset that represents the relative signal in the strobe list and is followed by a set of 1 and 0 that constitute the content of those signals in the golden machine at the time specified above, while the second line constitutes the values found in the faulty machine where the dot stands for no difference with respect to the golden machine and other values, may them be 1, 0 or X, the discrepancies found in the faulty machine. In the reported example, for instance, the fault affects the most significant byte of register 0 of the register file,

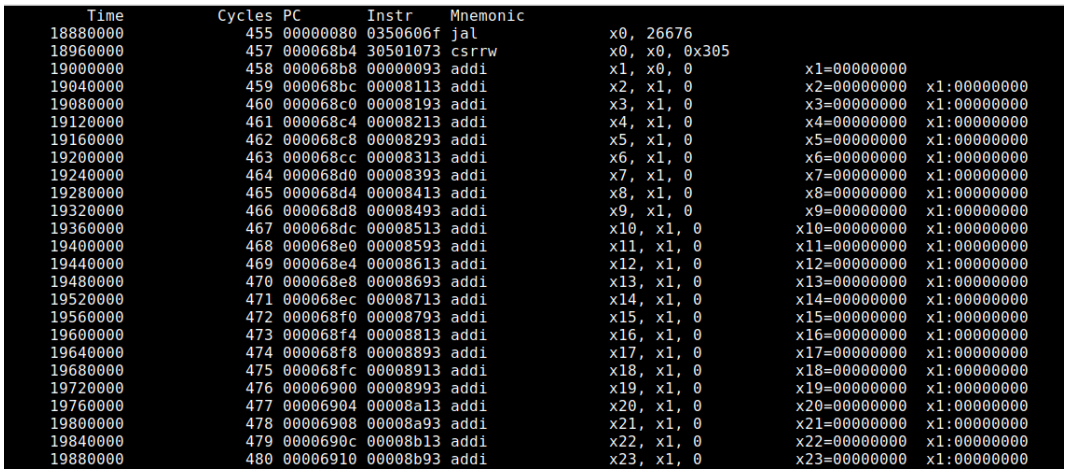
- The name of the fault from which the effect originates and the type of transition, in this case a slow-to-rise on `riscv_core.ex_stage_i_alu_i.U1718.A1`.

The information gathered so far is crucial to the test engineer, as it provides a framework for understanding what portions of the circuit are affected by the faults and the time instant at which such event occurs. Although relevant, this is still insufficient to pinpoint the instruction block inside the STL to be modified. To do that, understanding at what time instructions from the STL are being executed in the DUT is of paramount importance.

3.1.3 Logic Simulation Trace

The goal of the logic simulation trace is to gather information regarding instructions of the STL being executed in the processor under test, mainly the time at which instructions are executed and their operands value. To do so, the first step consists of launching a slightly modified version of the basic logic simulation of the device under test. Generally speaking, a logic simulation involves a device to be simulated plus a testbench that serves the purpose of instantiating the DUT, connecting its signals to other eventual modules, and driving the simulation by applying input stimuli and, optionally, recording output values. Under these conditions, a new module must be included in the testbench, that is, a tracer that has to be attached to the RTL description of the device under test. Such tracer automatically activates as soon as the simulation starts, and closely monitors internal signals of the DUT, like the clock cycle, the program counter (PC), the instruction's opcode coming from the instruction RAM and the instruction operand values. The instruction is reported both in form of hexadecimal value (as described in the documentation of the instruction set architecture) and as a mnemonic, providing full details to the test engineer. While the logic simulation proceeds and the processor under test executes the test program, the tracer stores all of the aforementioned information at each clock cycle, and stores the data into a text file. An example of the tracer's output is presented in Fig. 3.7.

After the fault simulation completes and the tracer finishes recording data into the text file, it is necessary to process such data in a way that is easily accessible in an automated way. For this reason, the proposed approach resorts to the adoption of a database where all data is stored. Starting from the data produced by the tracer and



Time	Cycles	PC	Instr	Mnemonic			
18880000	455	00000080	0350606f	jal	x0, 26676		
18960000	457	000068b4	30501073	csrrw	x0, x0, 0x305		
19000000	458	000068b8	00000093	addi	x1, x0, 0	x1=00000000	
19040000	459	000068bc	00008113	addi	x2, x1, 0	x2=00000000	x1:00000000
19080000	460	000068c0	00008193	addi	x3, x1, 0	x3=00000000	x1:00000000
19120000	461	000068c4	00008213	addi	x4, x1, 0	x4=00000000	x1:00000000
19160000	462	000068c8	00008293	addi	x5, x1, 0	x5=00000000	x1:00000000
19200000	463	000068cc	00008313	addi	x6, x1, 0	x6=00000000	x1:00000000
19240000	464	000068d0	00008393	addi	x7, x1, 0	x7=00000000	x1:00000000
19280000	465	000068d4	00008413	addi	x8, x1, 0	x8=00000000	x1:00000000
19320000	466	000068d8	00008493	addi	x9, x1, 0	x9=00000000	x1:00000000
19360000	467	000068dc	00008513	addi	x10, x1, 0	x10=00000000	x1:00000000
19400000	468	000068e0	00008593	addi	x11, x1, 0	x11=00000000	x1:00000000
19440000	469	000068e4	00008613	addi	x12, x1, 0	x12=00000000	x1:00000000
19480000	470	000068e8	00008693	addi	x13, x1, 0	x13=00000000	x1:00000000
19520000	471	000068ec	00008713	addi	x14, x1, 0	x14=00000000	x1:00000000
19560000	472	000068f0	00008793	addi	x15, x1, 0	x15=00000000	x1:00000000
19600000	473	000068f4	00008813	addi	x16, x1, 0	x16=00000000	x1:00000000
19640000	474	000068f8	00008893	addi	x17, x1, 0	x17=00000000	x1:00000000
19680000	475	000068fc	00008913	addi	x18, x1, 0	x18=00000000	x1:00000000
19720000	476	00006900	00008993	addi	x19, x1, 0	x19=00000000	x1:00000000
19760000	477	00006904	00008a13	addi	x20, x1, 0	x20=00000000	x1:00000000
19800000	478	00006908	00008a93	addi	x21, x1, 0	x21=00000000	x1:00000000
19840000	479	0000690c	00008b13	addi	x22, x1, 0	x22=00000000	x1:00000000
19880000	480	00006910	00008b93	addi	x23, x1, 0	x23=00000000	x1:00000000

Fig. 3.7 Data produced by the tracer

the disassembly of the test program, the database is implemented such that, for every instruction found in the STL, the following information is available:

- *id*: an automatic id assigned by the database at the moment of insertion,
- *time*: the time instant in nanoseconds at which the instruction was executed in the processor,
- *opcode*: the operation code of the instruction as per ISA documentation,
- *instruction*: the mnemonic of the instruction executed in the core,
- *output_register*: the destination register of the instruction,
- *input_registers*: the source register(s) of the instruction,
- *PC*: program counter value associated to the instruction,
- *source_file*: the name of the file where the executed instruction comes from,
- *source_line*: the line number of the *source_file* where the instruction is found,
- *source_instruction*: mnemonic of the instruction indicated by the *source_file* and the *source_line* (it may not coincide with the aforementioned *instruction* field due to operations performed by the compiler).

(10) ObjectId("61643780d19e8fdd7e5ce6e3")	{ 10 fields }	Object
_id	ObjectId("61643780d19e8fdd7e5ce6e3")	ObjectId
time	32040	Int32
opcode	f14020f3	String
instruction	csrrs x1, x0, 0xf14	String
output_register	x1	String
input_registers	[1 element]	Array
[0]	x0	String
PC	d0	String
source_file	tests.S	String
source_line	25	String
source_instruction	csrr ra,mhartid	String

Fig. 3.8 Example of database entry

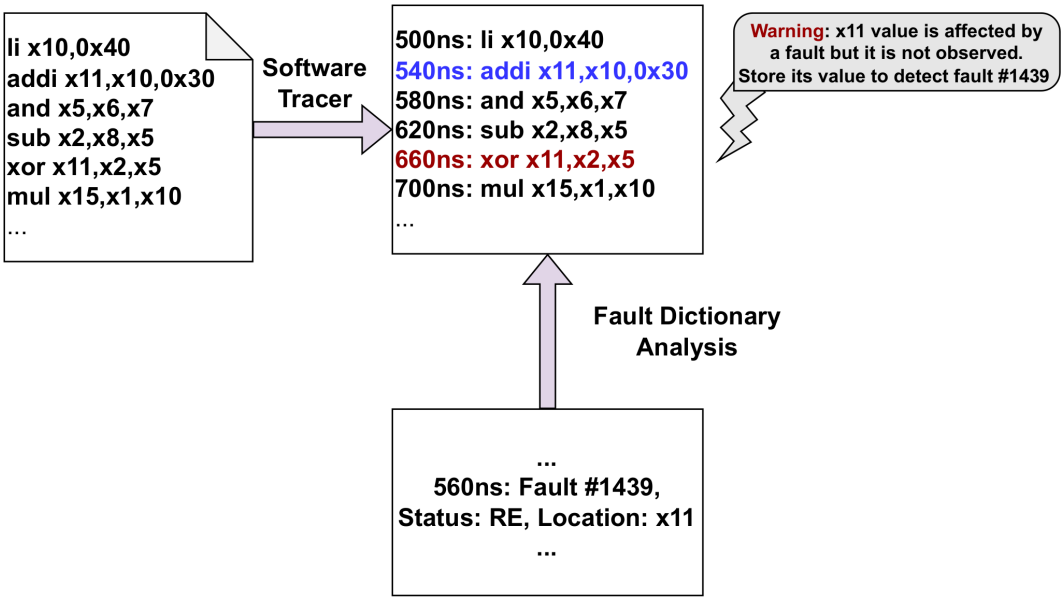


Fig. 3.9 STL improvement strategy

An example of a database entry is reported in Fig. 3.8.

The choice of adopting a database for storing the data generated by the tracer shows its strength when trying to tackle the problem of understanding where in the original code we should insert new instructions to enhance the detection of transition delay faults. Although techniques for generating such instructions are described in the following section, it is useful to visualize this concept by means of an example as reported in Fig. 3.9.

Starting from the test program (block on the left), a simulation trace log is produce by the tracer while running the logic simulation, associating instructions to time instants. After running the fault simulation, the fault dictionary reports that,

at 560ns, a fault reached the register x11 in the register file (RE status), but is never observed. Given this, the following step consists of identifying a range in the original test code where new instructions should be added to observe at primary outputs, and hence detect, the transition delay fault we are considering. Such problem can be modeled as (i) finding the first instruction that operates on register x11 before the time instant where the fault effect was observed in the register file, and (ii) the first instruction following the aforementioned time instant that overwrites the x11 register, that is, where x11 is the destination register. By means of carefully devised scripts, this problem can be fully automated thanks to the adoption of the database, returning not only the instructions that constitute the range into which to operate, reported in Fig. 3.9 in blue and red, but also the source file from which they originate and the source lines.

3.1.4 Test Program Enhancement

Now that information on what faults affect which internal observation points together with data on where to edit the original test program are available, it is possible to start discussing on the actual techniques to detect such faults based on their category. The reason for defining different strategies for User Accessible Register faults and hidden register faults lies in the aforementioned different controllability of such registers through instructions from the instruction set architecture. Section 3.1.4 describes the proposed approach for User Accessible Register faults, while Section 3.1.4 deals with hidden register faults.

User Accessible Register faults

User Accessible Register faults are faults whose effects have been propagated from the original fault site to registers that can be directly written and read through instructions from the instruction set architecture. Being able to directly access these registers' content through instructions helps the test engineer to make faults effects observable at the primary outputs. For instance, if we need to read the content of a register, the store instruction is the best candidate, as it allows to propagate the register's content to the memory from which it can be read at a later time. At a first glance, it would seem that simply putting store instructions after the instruction causing the fault to be not-observed is enough to observe the fault's effect and thus

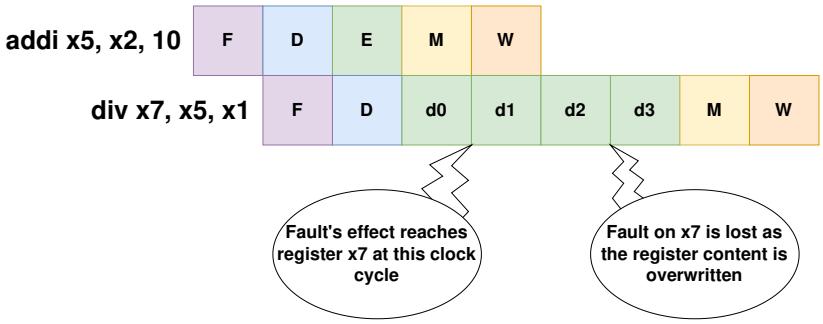


Fig. 3.10 Multi-cycle instructions and fault effects propagation

detect the fault. This, however, is not the case, as a more detailed analysis is required on the instructions that excite the fault and propagate its effects to user accessible registers.

Let us consider the example reported in Fig. 3.10. In this example, the a fault is excited and its effect propagated while a division operation goes through its execution stage. In this example, we consider the case in which the fault's effect reaches the register x7 after the first division cycle. As the division progresses, however, the register is being continuously overwritten, eventually to a point where the faulty value is permanently lost. Placing a store instruction after the division here would be pointless, as the store instruction should sample the register's content while the division is being executed, a feature that is not possible under any circumstance.

For this reason, faults stemming from the execution of these instructions — from hereinafter referred to as *multi-cycle instructions*, as opposed to *single-cycle instructions* that only take one cycle in the execution stage — should be carefully taken into account, as the fault effect might be overwritten during the required execution cycles. Techniques for detecting user accessible register faults are thus differentiated based on the two types of instruction.

When dealing with faults controlled by *single-cycle instructions*, a store instruction placed in the code portion after the fault has been observed at a given register and prior to the instruction that overwrites the aforementioned register is enough to observe the fault's effect at the primary output, hence marking the fault as detected. Additional action must be taken in specific cases when there is no direct match between the instructions found in the source file and those reported by the tracer. When an assembly source file is converted into machine code, the compiler can slightly modify the original program as all alias instructions are mapped into in-

structions from the instruction set architecture. Such mapping is not always ensured to provide a 1:1 conversion, meaning that an alias instruction may be substituted by two or more instructions from the ISA. An example of such feature is given by the pseudo instruction `load immediate, li`, whose effect is to initialize a register to an arbitrary 32-bit value, that is converted to a `load upper immediate, lui`, that takes a 20-bits immediate field that is used to initialize the upper 20 bits of a register, and an `add immediate, addi` instruction, through which the lower 12 bits are initialized. It is highlighted that both instructions write to the same register, given that they split a higher level functionality into two. If we consider the case where the fault effect is propagated to a register by the execution of the `lui` instruction, we can notice how a problem quickly arises: to detect such fault, we should place a store operation strictly in between the `lui` and `addi` instructions, as the `addi` overwrites the content of the register holding the faulty value; those instructions however are not found in the source file as they are the result of the compiling process. This is the reason why in the database implementation provided in Section 3.1.3 both the *instruction* and *source_instruction* fields are provided allowing to trace back the modified instructions generated by the compiler to the original one in the source file. Whenever this scenario occurs, the best course of action is to replace the alias instruction in the source file with the relative instructions and placing the store instruction when required.

If the fault is controlled by a *multi-cycle instruction*, on the other hand, two scenarios can be identified. The first one is quite straightforward, and consists of the case in which the fault effect is still present at the last execution cycle: in this case, the same strategy adopted for *single-cycle instructions* is used. In any other case, we duplicate the *multi-cycle instruction* and modify its operands to ensure that the faulty value reaches the register towards the end of the execution stage, so that it can be observed through a store instruction. In this case, duplicating the instruction rather than modifying the original one is necessary, as the instruction with the original operands might still be instrumental to the detection of other faults. Finding the new set of operands to make the fault effects observabe at the end of the execution stage cycles strictly depends on the instruction itself. It is noted that in most cases instructions that take more than one clock cycle in the execution stage implement either arithmetical or logical operations, thus shifting the issue to understanding how specific operations are implemented, e.g., what algorithm is used to implement divisions. In any case, as it is demonstrated by the experimental results

described in Section 3.2.2 and first presented in [55], identifying suitable operands for this purpose is a feasible task which can often be performed by the test engineer by manual inspection or by mathematically extrapolating them by looking at the adopted algorithm.

Finally, it is noted that in the worst case scenario one instruction must be added to detect a *single-cycle instruction* fault and two instructions must be added to detect a *multi-cycle instruction* fault. In a large amount of cases, however, several different faults happen to propagate their effects to the same register at the same time instant. For this reason, one set of instructions may be capable of detecting more than one not-observed transition delay fault at the same time, thus requiring a minimal code and execution time overhead.

Hidden Register faults

All those faults whose effects reach registers that cannot be directly observed through instructions fall within the hidden register faults group. These registers are deeply embedded inside the processor core, either belonging to pipeline registers or inner sub-modules, which makes particularly hard to propagate values from those locations to either primary outputs or user accessible registers.

The premises on which the strategy I propose to detect these faults is based on is similar to those defined for UARs, that is, it requires the analysis of the fault database to extract information on where faults propagate and stop and at what time instant, i.e., in what portion of the STL, such events occur. Given the nature of hidden registers, however, an additional analysis is needed to understand how to detect these faults.

In order to do so, we observe that the process of exciting a transition delay fault and observing its effects in a pipelined CPU can be decoupled into two sub-tasks. To exemplify this aspect, let us consider the circuit reported in Fig. 3.11. In this circuit, we assume that the net connected to the output of the first AND gate is affected by a slow-to-rise fault, and the goal is to provide a set of test vectors to make the fault effect observable. Given that in the circuit a flip-flop is present, the test vectors required to detect the fault are three, the first two generating the required transition on the net affected by the fault and propagating such transition to the flip-flop, the third vector propagating such transition from the flip-flop to the primary output, in

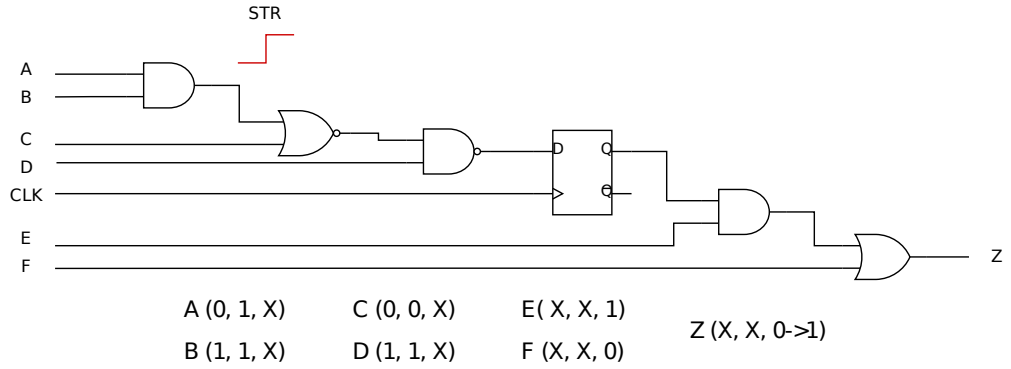


Fig. 3.11 Controlling and observing the effect of a transition delay fault in a sequential circuit

this case Z. In this example, inputs from A to D directly control logic that is placed prior to the flip-flop, while inputs E and F control logic after the flip-flop. For this reason, for the first two cycles E and F are set to a *don't care* value (represented with the letter X), as they do not influence the circuit portion involved with the generation and propagation of the transition to the flip-flop. These two inputs are assigned non-controlling values at the third clock cycle, as they must not block the propagation of the signal to the primary output. Inputs from A to D, on the other hand, are initialized with specific values in the first couple of clock cycles, while in the third one they are all set to a don't care state as the faulty value has crossed the flip-flop. In summary, the way the problem of testing a transition delay fault is decoupled into two sub-tasks is by first finding a specific pair of test vectors to be applied to generate the required transition and propagate it towards an endpoint, may that be a primary output — in which case the fault is marked as detected — or a register within the processor core. Secondly, if the fault's effect reached a register, methodologies to propagate such effect to primary outputs are employed to detect the fault. While the first step obviously depends on the fault model and the transition that we want to generate, the second step does not depend as much on the fault to be excited, and is just a problem of propagating a value from one point to another.

The aim of this section is to define an automatic way to easily increase the transition delay fault coverage for STLs that were previously devised for stuck-at faults. Given this group of faults, hence, we define the algorithm summarized in Algorithm 1.

The presented algorithm is described as follows: in order to test a transition delay fault F_i whose effects propagate and stop to the hidden register H_i at time T_i, it

Algorithm 1: HR faults detection algorithm

input : A list L of triplets (F_i, H_i, T_i) where
F_i is the transition delay fault to be tested
H_i is the HR bit reached by the fault's effect
T_i is the time at which the fault effect reached H_i
An STL S that has been developed for SAFs

output : A set of instructions to propagate transition delay fault effects to primary outputs

```
foreach (F_i, H_i, T_i) in L do
    if S detects H_i's stuck-at-1 and/or stuck-at-0 faults then
        get the time T_s at which the stuck-at fault on H_i is detected;
        extract a block B with the last N instructions before T_s from S;
        check whether B does not contain jump instructions;
        if F_i has been detected by B then
            add B to the original program;
        end
    end
end
```

may be useful to look for pieces of code coming from an STL S already developed for stuck-at-faults, as it may contain blocks of code capable of testing stuck-at-0 and stuck-at-1 faults located in pipeline registers. If that is the case, such block of code can also be used to propagate values from aforementioned locations to primary outputs. This makes for an effortless way to detect transition delay faults, as we just need to find the appropriate chunk of code and put it right next to the one that excites the transition delay fault and propagate its effect up to the relative pipeline register. This operation, however, should not disrupt the overall flow of the original test program: for this reason, jump instructions in the code to be added should be avoided. Implementing this strategy can be easily done in an automatic way by extending the set of functionalities offered by the database storing information on test programs. For instance, in case several STLs are available for stuck-at faults, it is sufficient to gather information on their fault coverage, understanding whether they are capable of detecting stuck-at faults affecting the same line where hidden register faults' effects propagate and stop, and provide a means of extracting the set of instructions responsible for such detection, elaborating it into a format that can be easily elaborated at a later time, e.g., a JSON file. This is achieved by means of scripts that coordinate all this information and all accesses to the database. In the rare case where the transition delay fault effect propagates to a bit in a pipeline register

whose corresponding SAFs are still not detected by the existing STL, methods such as [16] can be used to generate the required chunk of instructions (improving the SAF coverage as well). The size of such block of instructions, in Algorithm 1 reported as N, should be fixed to a specific value, so that the test program enhancement flow can fully operate autonomously. In this approach, few tests can be conducted in order to find the best solution in terms of code overhead and fault coverage improvement, and once such value is experimentally found it is kept for every HR fault.

Similarly to User Accessible Register faults, even for hidden register faults a set of instructions added to the original test program may be capable of detecting more than one TDF at the same time, thus achieving better fault coverages with a smaller test program with respect to having a set of instructions for each fault to be tested.

3.2 Experimental Results

This section is devoted to the presentation and analysis of the experimental results that were collected in order to prove the effectiveness of the proposed approach. The section is divided into a presentation of the core adopted for all tests and the suite of STLs that were taken as starting point in Section 3.2.1, followed by the fault simulation results in Section 3.2.2.

3.2.1 Case study

The approach presented in this chapter has been validated on PULPino[2], an open-hardware single-core system on chip platform based on a 32-bit RISC-V core developed by ETH Zurich and Università di Bologna. PULPino can be configured to use two type of cores, RI5CY or zero-ri5cy core. PULPino is configurable to use either the RISCY or the zero-riscy core. RISCY is an in-order, single-issue core with 4 pipeline stages. This core fully supports the base integer instruction set (RV32I), compressed instructions (RV32C) and multiplication instruction set (RV32M). Eventually, it is also possible to enable the single-precision floating-point instruction set extension (RV32F). It is designed so that it can be used in ultra-low-power signal processing applications, and it implements several additional functionalities such as hardware loops, multiply and accumulate operations. The internal architecture of this core is shown in Fig. 3.12

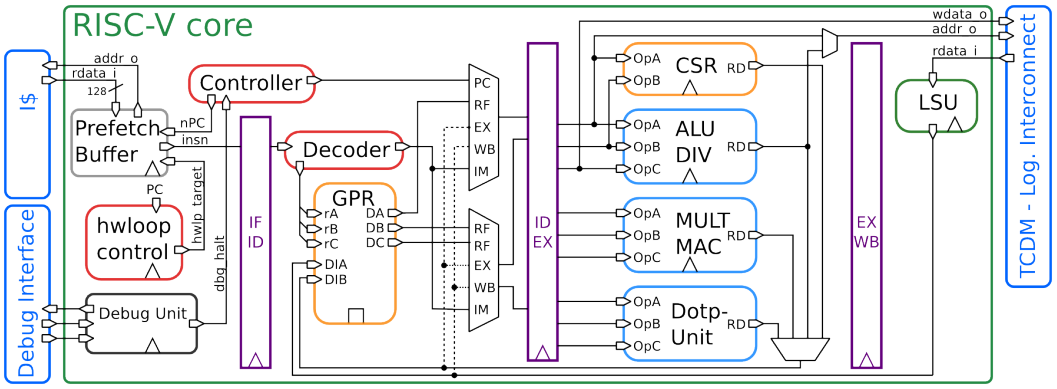


Fig. 3.12 Internal architecture of the RI5CY core[2]

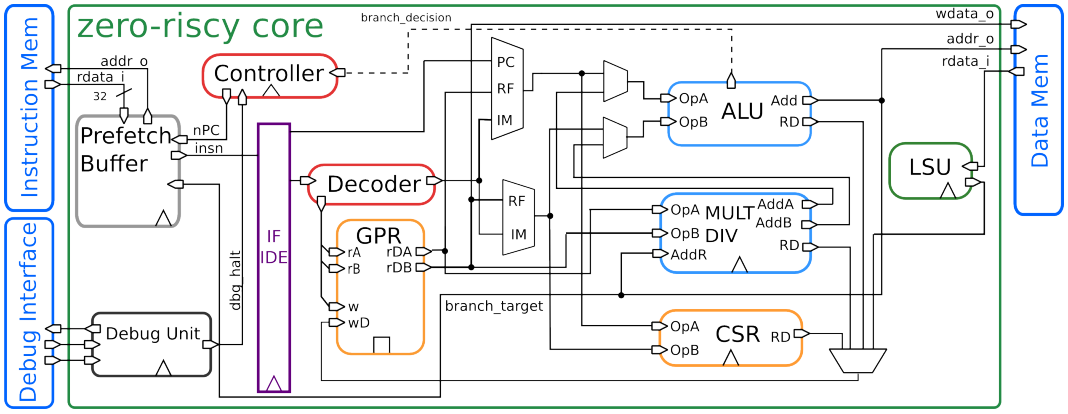


Fig. 3.13 Internal architecture of the zero-ri5cy core[2]

zero-riscy is an in-order, single-issue core with 2 pipeline stages and it fully supports the base integer instruction set (RV32I) and compressed instructions (RV32C). It can be configured to have multiplication instruction set extension (RV32M) and the reduced number of registers extension (RV32E). It has been designed to target ultra-low-power and ultra-low-area constraints. The internal architecture of the zero-ri5cy core is shown in Fig. 3.13.

In these experiments, PULPino was configured to use the RI5CY core, as it is a good trade-off in terms of circuit complexity, size, and amount of offered functionalities. This core has been synthesized using the 45nm Silvaco Open Cell library (former Nangate Open Cell library)[61]. The PULPino system on chip comes with a set of peripherals, e.g., communication peripherals such as UART, SPI, I2C. The target of this approach, however, is the CPU only: for this reason, all other

components have been excluded by the synthesis process and all following steps. Data regarding the synthesized core is reported in Table 3.1.

Table 3.1 Case study general info

Parameter	*Value*
Number of gates	46,850
Total Area (eq. gates)	51,001.65
Clock Period (ns)	40.00
#Transition Delay Faults	159,326

In order to check how well this method performs under different scenarios, I selected a set of three different Self-Test Libraries that were previously developed to test stuck-at faults on the PULPino core, namely STL1, STL2, and STL3. To ensure a diverse and realistic testbench, the three selected test programs have been developed following different implementation strategies. Table 3.2 reports a summary of the most important characteristics of the adopted STLs, namely the execution time (expressed in the total amount of clock cycles), code memory size, and stuck-at fault coverage.

Table 3.2 STLs general information

Test Program	*#Clock cycles*	*Memory size [kB]*	*SAF coverage %*
STL1	17,308	27.32	81.42
STL2	31,158	27.86	81.86
STL3	80,455	16.68	82.18

An important aspect to highlight is that the reported execution time for each STL, that is, the amount of clock cycles for the test routine to execute, has been computed by running STLs in their entirety. The test engineer, however, is free to split them into sub-modules that can be launched separately depending on the situation and the needs, thus creating a set of smaller routines that can fit into small idle time slots with the final cumulative fault coverage.

The fault simulation experiments have been launched on Synopsys Z01X, a commercial tool that was devised specifically for functional safety. Employing Z01X is particularly advantageous, mainly because it provides the promotion table functionalities that are crucial to implement the observability study described in

Section 3.1.2, as well as it features a noticeable fault simulation speed: performing transition delay fault simulations took, for each test program, no longer than 1 day on an Intel Xeon CPU E5-2680 v3 server with a clock frequency up to 3.3GHz.

3.2.2 Achieved results

In this subsection, achieved results are discussed in details both in terms of overall fault coverage improvement and fault coverage improvement related to specific register groups, may them belong to user accessible registers and hidden registers. Starting with data obtained on the whole core, Table 3.3 and Table 3.4 report the most relevant information on the user accessible register and hidden register faults that were detected as a result of the proposed methodology.

Starting with Table 3.3, it is possible to see that our approach is greatly effective as it is capable of detecting almost every fault out of those that are excited but not detected by the existing STL, with the worst case scenario being STL3 with a 98.76% of UAR faults being detected. Given a total amount of 159,326 transition delay faults, through our methodology we can increase the final fault coverage by 4.13% for STL1, 15.01% for STL2, and 1.80% for STL3, respectively. The original transition delay fault coverages of these programs are 61.73% for STL1, 44.19% for STL2 and 62.54% for STL3 respectively; by adding the now detected user accessible faults the fault coverages reach 65.86% for STL1, 59.20% for STL2 and 64.34% for STL3 as reported in Fig. 3.14.

Table 3.3 Analysis on detected UAR faults

	STL1	STL2	STL3
Detected UARs	6,578	23,912	2,864
Total UARs	6,591	23,922	2,900
%Detected UARs	99.80	99.96	98.76
Code size [kB]	6.34	4.17	3.53

This improvement comes with an increase of the final code size that amounts to an additional 22.21% for STL1, 14.97% for STL2, and 21.16% for STL3. This proves that the proposed strategy is able to systematically test not-observed transition

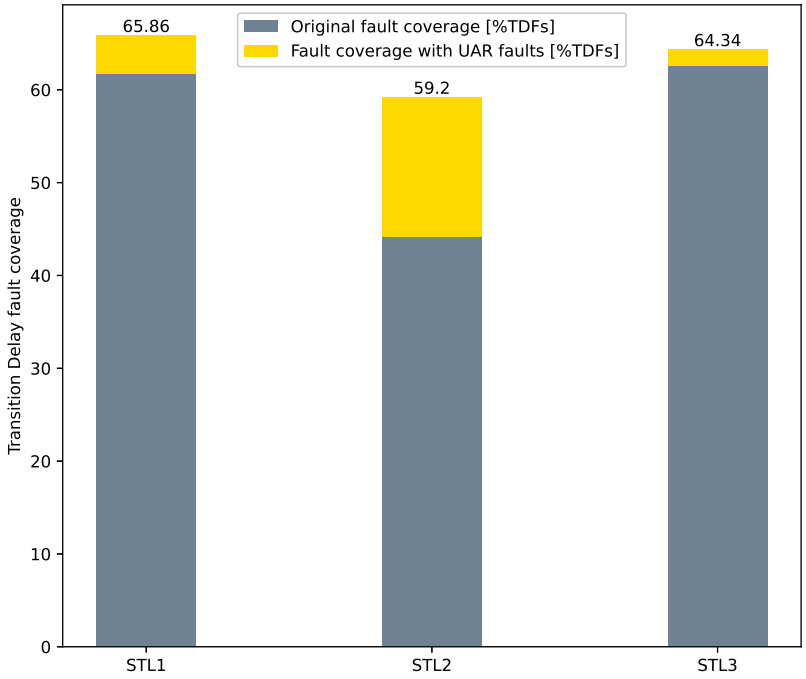


Fig. 3.14 Transition delay fault coverage with UAR faults

delay faults whose effects reached user accessible registers with a reasonable code size increase.

Moving on to data on hidden register faults, Table 3.4 reports information on recovered HR faults in the whole core. In this case, as it can be deducted from the table, the results achieved thanks to the presented methodology are quite dependent on the considered STL.

Table 3.4 Analysis on detected HR faults

	STL1	STL2	STL3
Detected HRs	643	183	608
Total HRs	6,741	3,599	3,955
%Detected HRs	9.54	5.08	15.37
Code size [kB]	2.60	0.92	0.92

For the HR group of faults, the worst case scenario is represented by STL2, for which 5.08% HR faults can be detected, while the best case scenario is represented by STL3, with a total of 15.37% faults detected. Although the results are not as high as for UARs, it is worth mentioning that this methodology does not require any effort from the test engineer, allowing to automatically detect this set of faults. Moreover, experimental data for hidden register faults shows that the increase in code size due to the test program enhancement is rather small, amounting to an additional 9.52% for STL1, 3.30% for STL2, and 5.52% for STL3, respectively. In addition to that, it is also worth mentioning that the data presented in this section does not include a functional untestability analysis. This implies that, among all not-observed transition delay faults belonging to the hidden register group, there is a percentage that cannot be detected in any way with SBST technique, hence making these numbers a worst-case scenario. Increase in the original transition delay fault coverage thanks to the detection of HR faults is shown in Fig. 3.15.

Finally, it is possible to combine data obtained from both user accessible register and hidden register faults, providing a final increased fault coverage that is reported in Fig. 3.16.

In addition to results produced on the whole core, user accessible registers and hidden registers have been divided into blocks of registers belonging to the same functional unit, e.g., user accessible registers have been divided into general purpose

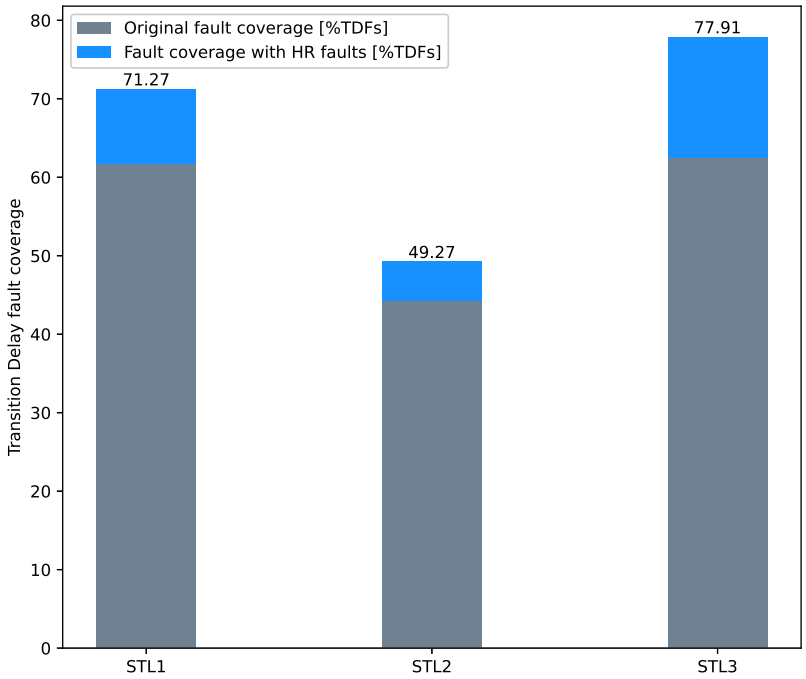


Fig. 3.15 Transition delay fault coverage with HR faults

Table 3.5 Sub-modules analysis for the adopted STLs targeting UAR faults

Test Program		User Accessible Register faults	
		GPRs	SPRs
STL1	Detected faults	4,359	2,219
	Total faults	4,359	2,232
	Added Instructions	1,107	478
STL2	Detected faults	23,814	98
	Total faults	23,814	108
	Added Instructions	1,022	20
STL3	Detected faults	2,853	11
	Total faults	2,853	47
	Added Instructions	877	6

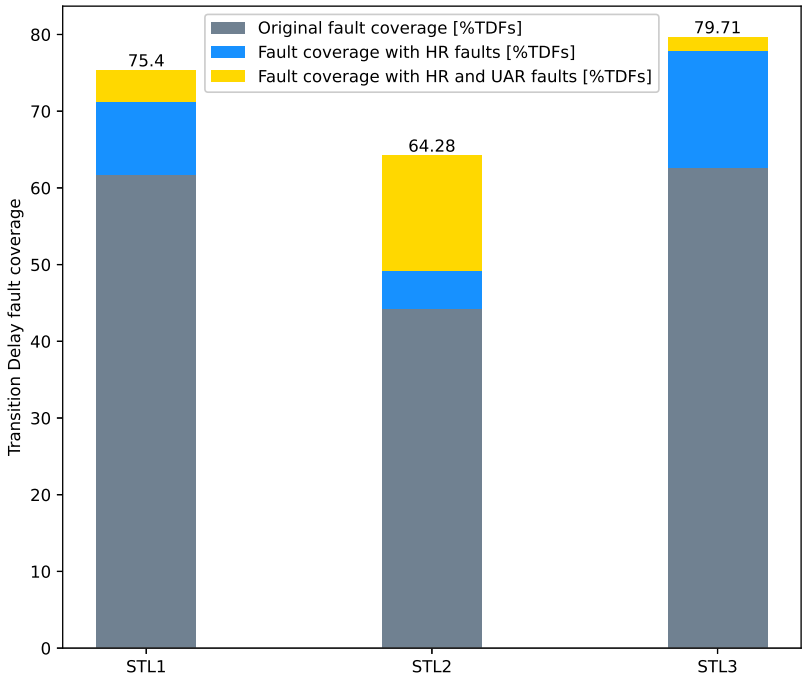


Fig. 3.16 Transition delay fault coverage with UAR and HR faults

Table 3.6 Sub-modules analysis for the adopted STLs targeting HR faults

Test Program		Hidden Register faults			
		IF Stage	ID Stage	EX Stage	MEM Stage
STL1	Detected faults	23	587	32	1
	Total faults	1,109	5,109	388	135
	Added Instructions	45	550	25	5
STL2	Detected faults	52	120	5	6
	Total faults	1,311	1,976	221	91
	Added Instructions	80	115	20	15
STL3	Detected faults	13	595	0	0
	Total faults	1,028	2,463	351	113
	Added Instructions	35	195	0	0

registers and special registers while hidden registers have been divided into all pipeline registers, and data for these blocks has been collected as well. Table 3.5 and Table 3.6 describe the information regarding sub-modules of the tested processor core in details, reporting the contributions in terms of detected faults, total faults and added instructions regarding user accessible registers and hidden registers, respectively. Starting from the UAR group, the table shows how all GPRs have been tested, while only a small minority of SPRs is left undetected. When talking about UAR faults, it is also worth mentioning how many fall within the single-cycle and multi-cycle groups. Concerning STL1, out of all the 4,359 GPR faults 1,366 are related to single-cycle instructions and 2,993 to multi-cycle instructions, while the 2,232 SPR faults are divided into 2,219 single-cycle and 13 multi-cycle related faults. STL2, on the other hand, has a total of 23,814 UAR faults, of which 22,683 are related to single-cycle instructions and 1,131 are related to multi-cycle instructions, and the 108 SPR faults can be grouped into 98 single-cycle and 10 multi-cycle related faults. Finally, STL3 has 2,853 faults of which 1,367 are related to single-cycle instructions and 1,486 multi-cycle instructions; of all 47 SPR faults, 11 are single-cycle and 36 are multi-cycle related faults. The distinction between single-cycle and multi-cycle related faults impacts the number of added instructions required to detect the faults as well. As mentioned in Section 3.1.4, single-cycle related faults only need a store instruction to be detected, with an additional overhead of one instruction for SPR faults consisting in moving the value of the special register into a general purpose register so that it can be stored. Multi-cycle related faults, on the other hand, require to duplicate the related multi-cycle instruction and change its operands to make sure that the fault's effects are propagated towards the final cycles of the aforementioned instruction, plus a store instruction to observe the aforementioned effects at the primary outputs. Most not-detected SPR faults belong to the multi-cycle category, due to the fact that finding the correct operands to propagate the error can be non trivial.

Looking at the HR group, the best results are achieved in the pipeline registers in between the decode and the execute stage, while the other stages pose some challenges. The main reason for having a lower fault coverage stems from the fact that it is not always possible to find the right set of instructions that propagates the values from the pipeline stages to the primary outputs. As for the number of added instructions, experimental data shows that the best results are achieved when adding 5 instructions from the stuck-at fault related test program. It is worth iterating the

fact that not every detected fault needs additional instructions, as some faults may cause errors at the same register in the same time instant, thus requiring only one set of added instructions.

3.3 Chapter Summary

This chapter described a systematic methodology for detecting a set of not-observed transition delay faults by means of a purely software approach. Prior to the description of techniques to improve test programs, the proposed approach introduces some preliminary steps that are required in order to produce the information needed for coming up with STL hardening strategies. Two fault categories have been taken into account, one being all not-observed transition delay faults whose effects propagate and stop to registers that can be directly controlled by instructions of the instruction set architecture, called User Accessible Register faults, the other consisting of not-observed transition delay faults whose effects propagate and stop to register that cannot be directly controlled by instructions, called Hidden Register faults. Starting from these definitions, a first Internal Observation Points extraction is performed, with the goal of collecting information on all user accessible registers and hidden registers. After that, two tasks are performed in parallel, the Observability Study and Logic Simulation Trace. The Observability study is done in order to understand what faults belong to the UAR category and what faults belong to the HR group and the time at which their effects stop at those registers. To do that, the concepts of promotion table and fault dictionary are introduced. A promotion table is a tool supported by some commercial fault simulation tools that allows to create a set of user-defined labels that are attached to faults whenever their effects reach specific points in the circuit. A label is created for each block of registers both for user accessible registers and hidden registers, thus providing an immediate way of categorizing these faults that occurs while performing the fault simulation. These labels are integrated with those that are already used by the fault simulation tool, and for this reason a ranking is provided so that the tool knows how to promote from one label to another, allowing promotions from labels with lower priority to those who have a higher priority only. In this sense, not-dected fault labels (not-controlled, not-observed) have the lowest priority, followed by hidden register faults labels, user accessible register labels (as the related registers are directly controllable through instructions, hence easier to

observe) and finally potentially detected and detected fault labels. The result of a fault simulation process with the adoption of promotion tables can be formatted into a fault dictionary, with the goal of presenting in a compact and easy-to-parse way data on the what register the fault was observed, thanks to the user defined labels, and at what time such internal detection occurs. Concurrently, the Logic Simulation Trace process is launched, with the aim of associating instructions being executed in the processor core with their execution time. This data is crucial, as when intersected with that produced in the fault simulation it is possible what instruction caused what fault to propagate to a specific point in the core, enabling the test engineer to identify the code region that must be modified to detect the not-observed transition delay fault. Information on the test program's instruction is stored into a database, providing ease of access and information retrieval.

After clearing all the preliminary steps, techniques for improving STLs for transition delay faults are presented. Starting with UAR faults, it is noted that two main scenarios can occur: the fault can be excited and its effects propagated by a *single-cycle* instruction, i.e., an instruction that takes only one cycle in the execution stage, or by a *multi-cycle* instruction, i.e., an instruction that takes more than one clock cycle in the execution stage. In the first case, a single store instruction after the instruction exciting the fault and before the next one overwriting the register where the fault effect is propagated and stopped is sufficient. In case the original test program presents alias instructions, extra care must be put in all those situations where the alias instruction is converted by the compiler into two instructions and the fault is excited by the first of the pair: putting a store after the alias instruction originally found in the source program is not sufficient as the fault effect is overwritten by the instruction pair. The best course of action here is to replace the alias instruction with the actual ISA instructions and place the store as needed. HR faults require a completely different approach as hidden registers cannot be directly controlled. The proposed method relies on searching for pieces of code that are capable of testing stuck-at faults on the same location where the fault effect propagated and stopped, with the idea that the same piece of code can propagate the faulty value due to the presence of a transition delay fault to one of the primary outputs. Pieces of code to be added, however, must not disrupt the overall flow of the test program; for this reason code blocks that include jump instructions are discarded.

This approach has been validated on a 32-bit, single issue in-order RISC-V core, starting from a set of three STLs developed for stuck-at faults. Experimental results

show that almost every user accessible register fault is detected, while coverages for HR faults depend on the test program. The approach proves to be effective in terms of faults being detected versus code overhead required to do so, with a final increase in coverage of 22.21% for STL1, 14.97% for STL2 and 21.16% for STL3 if UAR faults only are targeted, and a code increase of 31.73% for STL1, 18.27% for STL2 and 26.68% for STL3 if UAR and HR faults are targeted. Future works will include the improvement of software techniques for detecting HR faults specifically.

Chapter 4

Improving transition delay fault coverage through post-silicon debug logic

The previous section introduces techniques to improve the transition delay fault coverage of Self-Test Libraries developed for the classical stuck-at fault model by targeting User Accessible Register faults and Hidden Register faults. As outlined in Section 3.2.2, almost every User Accessible Register fault is observed by the proposed methodology, while hidden register faults prove to be harder to test, mainly due to the fact that there is no immediate way to propagate values from those registers to the primary output. Most modern System on Chips are equipped with special logic that is used for post-silicon validation purposes. Such logic, however, can also be reused for testing purposes, with the great advantage of allowing increased observability of internal signals that would otherwise be inaccessible from the outside, without additional hardware overhead as the circuitry is already there. In this section, I propose a second technique that makes use of post-silicon debug circuitry to effectively target this group of hard-to-test faults.

4.1 Proposed Approach

The approach proposed in this section aims at defining systematic techniques to analyze Self-Test Libraries that were previously devised for other fault models and

identify all the flip-flops where the effects of not-observed transition delay fault propagate and stop. Starting from that, methods to cleverly select subset of flip-flops to monitor during the STL execution are defined, with the purpose of increasing the final transition delay fault coverage.

This approach can be decoupled into two steps. Section 4.1.1 briefly summarizes the fault dictionary based technique to generate a list of all flip-flops belonging to the hidden register's group that are reached by transition delay fault's effects. Section 4.1.2 defines post-processing procedures to identify a subset of all flip-flops to monitor. This subset can either be fixed or variable in time, based on the features of the hardware that is available to support the test.

4.1.1 Generation of fault dictionary

In order for the approach proposed in this section to extract the subset of flip-flops, a preliminary step involving the generation of a fault dictionary is required. As first described in Section 3.1.2, a fault dictionary is especially useful whenever information on where fault effects were observed together with at what time they were observed is required. Fault dictionaries are generated while performing fault simulations, and require the definition of user-defined labels in conjunction with a promotion table. The way in which the fault dictionary is generated for this approach is similar to Section 3.1.2, with some slight modifications. In particular, a two fault simulation steps is required, described as follows:

1. Run a fault simulation on the whole processor under test using test vectors obtained by the execution of the STL, and extract all not-observed faults,
2. Run without fault-dropping whenever possible, or an *n-detection* fault simulation on the combinational logic of the DUT, using the set of not-observed faults previously identified and observing the *pseudo-primary outputs* connected to hidden register flip-flops (i.e., pipeline registers). Generate a fault dictionary using all gathered information.

The first step is required to obtain a list of not-observed transition delay faults only, and for this reason no promotion table is required, as commercial fault simulation tools already define this fault class. Once the first step is cleared, the second

fault simulation involving the definition of labels for pipeline registers and user accessible registers and the related promotion table can be launched. This simulation however differs from what has been presented so far in the sense that it must be conduced with the no-fault dropping option enabled if possible, or at least with n-detection. When no-fault dropping is enabled, a fault that has been observed at one of the observation points imparted to the fault simulation tool is not dropped from the fault list. Under this scenario, this implies that for every fault the tool returns all possible pairs of time instants and flip-flops affected by the propagation of erroneous values due to the presence of a fault. In this approach this is particularly useful: the more time instants and registers are affected by the propagation of a fault's effects, the better the post-processing procedures perform, allowing for a high percentage of hard-to-test, not-observed transition delay faults to be detected. Nevertheless, depending on the architecture of the design under test, it may lead to computational intensive fault simulations and generate large dictionaries. Such issue is solved by using an *n-detection* fault simulation, i.e., a fault simulation where a fault is dropped from the active fault list after n detections have occurred. This provides the best trade-off in terms of faults recovered and fault simulation time and allocated resources.

4.1.2 Flip-flops selection procedure

In the second step, the fault dictionary previously generated is processed to identify a subset of flip-flops to monitor through trace buffers. Given that the hardware required to observe the value of such registers is already present inside the System on Chip, as this approach reuses the debug infrastructure, critical paths within the processor under test are not affected, thus not impacting the timing performance. Such debug infrastructure is effectively used to increase fault observability in a trasparent manner.

The algorithm used to extract the subset of flip-flops depends on the available hardware and its features. For this reason, three different scenarios are identified, leading to three different selection strategies. It is noted that the goal of this section is not to provide a hardware implementation of the debug infrastructure adopted for testing purposes, as several articles already tackle this issue as outlined in Section 1.3.3, rather to define algorithms that select the minimum amount of flip-flops to effectively increase transition delay fault coverages.

The three scenarios are described as follows:

1. A non-programmable hardware infrastructure can monitor a certain number of flip-flops (e.g., by compacting their values using a Multiple-Input Shift register, or MISR) selected at design time,

2. A programmable hardware infrastructure can monitor a certain number of flip-flops, switching the configuration of monitored flip-flops after an arbitrary amount of clock cycles,

3. A programmable hardware infrastructure can trace a certain number of flip-flops, switching the configuration of monitored flip-flops after a fixed amount of clock cycles.

The first scenario requires identifying the subset of flip-flops to observe during the whole STL run. The algorithm to determine such a subset is described in Algorithm 2.

Algorithm 2: Fixed flip-flop selection

input : A pair $(D, C_{\min})$ where
D is a list of triplets (F_i,P_i,T_i) where
F_i is a fault
P_i is a flip-flop where a F_i is captured
T_i is the time when F_i is captured in P_i
$C_{\min}$ is a target coverage of recoverable faults
output : A set of flip-flops to observe
S := empty list of flip-flops;
while *coverage* $< C_{\min}$ *or D is not empty* **do**
  $P_{\max} \leftarrow$ flip-flop with most distinct faults in D;
  add $P_{\max}$ to S;
  remove all elements with $P_{\max}$ from D;
end
return S

The algorithm starts off with data processed from the fault dictionary, holding a list of hidden register flip-flops reached by erroneous values due to faults and the time at which such events occur. As noted in Section 3.2.2, it is frequent that more than one fault happens to propagate their value at the same register in the same time instant. Selecting those registers thus provides an advantage in terms of amount of recovered faults versus number of selected flip-flops, a parameter that is fixed by

the maximum size of trace buffers. For this reason, at each iteration, the algorithm selects the flip-flop that increases the fault coverage the most, until a target fault coverage $C_{\min}$ is reached or all the flip-flops are selected. A threshold can be included to stop after selecting a certain number of flip-flops. Given that this algorithm is related to the scenario #1, it is noted that it does not give the possibility to switch configurations: once a set of flip-flops has been selected, it is kept statically for the whole test procedure.

The other two scenarios are addressed by the algorithm presented in Algorithm 3, which follows a First-Come First-Served (FCFS) approach. In addition to the input data required in Algorithm 2, i.e., a list D of triplets containing a not-observed transition delay fault F_i whose effects propagated and stopped at P_i at time T_i, this algorithm requires the amount of flip-flops that can be observed in any given *configuration* L_{max}, as well as the amount of time assigned to each configuration T_{max}, if any. Each configuration corresponds to a list of flip-flops and the time to reconfigure the hardware infrastructure. Then, until the fault coverage has not reached the minimum target or D is not empty, the algorithm extracts a triplet from D and looks whether the fault F_{next} is still not tested. If untested, it also looks whether there is room for adding the flip-flop P_{next} to the current configuration and, in case configurations last for fixed amounts of clock cycles, if the time T_{next} is compatible with the current configuration. In such case, the fault is marked as tested, and if the flip-flop P_{next} was not part of the configuration it is added; else, the configuration is complete (no more flip-flops can be observed in a given observation cycle, or a new observation cycle begins in case its size is fixed, faults that are captured at the same time of the last fill by other non-included flip-flops are discarded, as per FCFS policy. At that time, when a new time instant is encountered, the algorithm stores the configuration and moves to the next one, until reaching the target fault coverage or the end of the dictionary.

Scenario #2 and #3 are quite similar, the only difference being that the former allows to switch configuration after an arbitrary amount of time while the other switches configuration after a fixed amount of clock cycles. For this reason, the user can omit timing information, i.e., the maximum observation time T_{max} in clock cycles (highlighted in blue in the pseudo-code), to deal with the second scenario, leading to configurations that can be kept for an arbitrary amount of clock cycles. This algorithm, although more complex, closely reflects the behavior of post-silicon debug circuitry, and is capable of providing more accurate and efficient results.

Algorithm 3: Variable flip-flop selection

input : A quadruplet (D, $L_{\max}$, $T_{\max}$, $C_{\min}$) where
D is a list of triplets (F_i,P_i,T_i) where
F_i is a fault
P_i is a flip-flop where a F_i is captured
T_i is the time when F_i is captured in P_i
$L_{\max}$ is the max number of flip-flops to select
$T_{\max}$ is the max observation time
$C_{\min}$ is a target coverage of recoverable faults

output : A list of pairs (S_j, T_j) where S_j is a set of flip-flops to select at time T_j

S := empty set of flip-flops;
R := empty list of (set of flip-flops, time) ;
$U \leftarrow$ all untested faults in D;
order D by time;
while *coverage* $< C_{\min}$ *or* D *is not empty* **do**
  (P_{next}, F_{next}, T_{next}) $\leftarrow$ extract first el. from D;
  if F_{next} *in* U **then**
    if (`Length` $(S) < L_{\max}$ *or* P_{next} *in* S) *and* $T_{next} - T_{flush} < T_{\max}$ **then**
      remove F_{next} from U;
      if P_{next} *not in* S **then**
        add P_{next} to S;
      end
      $T_{add} \leftarrow T_{next}$;
    else if $T_{next} > T_{add}$ **then**
      add (S, T_{flush}) to R;
      remove F_{next} from U;
      clean S and add P_{next};
      $T_{add} \leftarrow T_{next}$;
      $T_{flush} \leftarrow T_{next}$;
    end
  end
end
return R

4.2 Experimental Results

4.2.1 Case study

Table 4.1 STLs general information

	STL1	STL2	STL3	STL4	STL5
Duration [c.c.]	17,308	31,158	80,455	64,541	118,137
Size [kB]	27.32	27.86	16.68	36.04	35.61
DT SAF faults	151,558	152,269	152,801	160,149	156,038
DT TDF faults	117,758	75,826	118,374	119,367	123,060
RC SAF faults	4,581	2,842	2,327	4,213	3,412
RC TDF faults	7,669	31,338	3,161	5,007	3,562
SAF Coverage	81.66	82.02	82.32	86.22	84.03
TDF Coverage	63.09	40.74	63.41	63.94	65.91
RC SAF Cov.	2.44	1.51	1.24	2.24	1.82
RC TDF Cov.	4.08	16.68	1.68	2.67	1.90

The methodology introduced in this thesis has been validated on PULPino[2], the same processor adopted in Section 3.2.1. The DUT has been synthesized using the 45nm Silvaco Open Cell library[61] and accounts for 51,001 NAND2-equivalent gates, 187,857 stuck-at faults (SAF) and transition delay faults (TDF), and 1,207 flip-flops belonging to hidden registers.

With regards to the adopted test programs, we selected a set of five different STLs developed with the aim of testing stuck-at faults on the adopted core, referenced here as STL1 to STL5. To ensure test diversity, the proposed five test programs have been developed by distinct teams following various implementation and testing strategies. Table 4.1 summarizes the most relevant data of the adopted STLs, i.e., the test *Duration* in clock cycles, the memory footprint of the test program (*Size*), the number of *Detected faults* (DT SAF and TDF) in absolute value,the amount of *Recoverable (RC)* stuck-at and transition delay faults, the *SAF and TDF Fault coverage*, and the *Recoverable fault coverage* for both transitiont delay and stuck-at faults, i.e., how much the fault coverage can be increased by detecting all hidden register faults. Since all STLs were developed targeting the stuck-at fault model, it is possible to notice how stuck-at fault coverages are comparable among all test programs, while transition delay fault coverage figures are less homogeneous, with

STL2 being significantly less effective, with a starting 40.74% fault coverage and STL5 the most effective, with an initial 65.91% fault coverage.

Fault simulations have been carried out using Synopsys Z01X, a commercial tool devised specifically for Functional Safety. As a result, the full flow of top-level and combinational level stuck-at and transition delay fault simulations took no longer than 5 hours on an Intel Xeon CPU E5-2680 v3 server with a clock frequency up to 3.3GHz. The second fault simulation performed on the combinational logic has been conducted by setting the n-detection parameter to 50, i.e., dropping each fault after 50 detections, which led to fault dictionaries not larger than 15MB. The two post-processing algorithms are written in Python and require few seconds to analyze each fault dictionary.

4.2.2 Fixed flip-flop selection

Let us start first by analyzing data obtained when the fixed flip-flop selection algorithm in Algorithm 2 is applied, as showed in Fig. 4.1. Given the relationship existing between transition delay faults and stuck-at faults, i.e., test vectors that are able to detect a transition delay fault affecting a given net on a circuit are capable of testing a stuck-at fault on the same net, improving the transition delay fault coverage allows for an enhancement on the stuck-at fault coverage too. For this reason, Fig. 4.1 reports results for both fault models.

More in details, each graph reports the percentage of *recovered* faults, i.e., faults that become detected (y-axis) when a given percentage of flip-flops are monitored (x-axis). Looking at the performance of the algorithm when tackling transition delay faults, it is possible to notice that STL2 behaves quite differently from the other programs having a quite steep slope at the very beginning, allowing to recover more than 80% of transition delay faults by observing just 2.24% of all flip-flops. It is noted, however, that the transition delay fault coverage of this test program was quite low to begin with as shown in Table 4.1, and the curve, once reached this value, markedly changes its slope, requiring 63.30% of all flip-flops to reach 100%.

Looking at other test programs, detecting 50% of faults requires observing 28% for STL1, 8.5% for STL3, 12% for STL4 and 15% for STL5 of all flip-flops. Moving to the 75% mark, we must monitor a percentage of flip-flops equal to 50% for STL1, 17% for STL3, and about 30% for STL4 and STL5. As for the previous case, trying

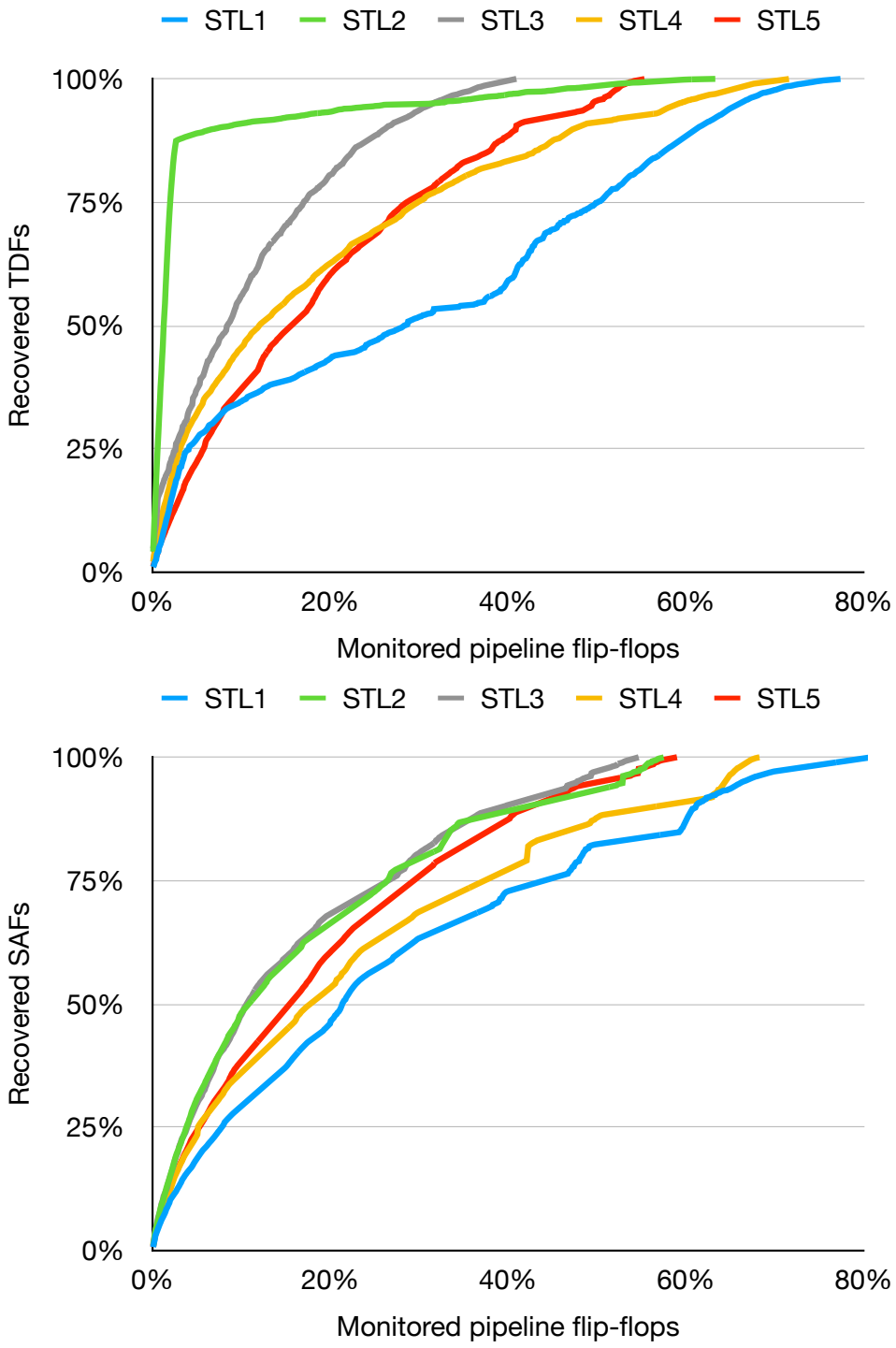


Fig. 4.1 Undetected TDFs and SAFs faults recovered using a fixed selection of pipeline flip-flops to monitor.

to detect all the excited but not detected transition delay faults requires a significant number of flip-flops to be observed, the worst case scenario being 78% for STL1, equal to a total of 942 flip-flops.

Looking at stuck-at fault curves, some of them present significant differences with respect to the transition delay fault case. Such differences can be explained by noting that the available STLs have been developed keeping the stuck-at fault model in mind. Covering 50% of excited but not detected stuck-at faults requires to observe 10% of flip-flops for STL2 and 3, 18% of flip-flops for STL4 and STL5, and more than 20% for STL1. However, if we increase the amount of recovered stuck-at faults to 75%, we must observe 29% of all flip-flops for STL2, STL3, and STL5, 38% of flip-flops for STL4 and 44% for STL1. If we aim at recovering *all* SAF faults, we need to monitor 58% of flip-flops for STL2, STL3, and STL5, 68% for STL4 and 80% for STL1. In the worst case scenario we would have to observe 960 flip-flops for the whole duration of the test procedure.

4.2.3 Variable flip-flop selection

Results of Algorithm 3 are reported for each STL in Figs. 4.2 to 4.6 for both transition delay faults and stuck-at faults. These images report data on the percentage of recoverable faults with respect to the trace buffer width (*Bits*) and the number of clock cycles during which a configuration of observed flip-flops is kept (*Slots*). The value *inf.* defined for *Slots* means that there is no fixed number of clock cycles for the trace buffer to observe, hence the configuration can be kept for as many clock cycles as necessary; this situation implies for example the presence of a MISR to compact the values of the monitored flip-flops.

Although different in absolute values, all figures report very similar trends for all test programs. To get more into details, it is possible to see that the larger the trace buffer width, the higher the amount of recovered faults: providing 128 bits, a size usually adopted with trace buffers, allows to recover all transition delay faults and stuck-at faults in almost every case. Looking at transition delay fault figures, 32-bits trace buffers are required to observe more than 90% of faults, with the sole exception of STL3, where it is possible to recover about 84% of faults with minimal fluctuations due to the different timing slots; when dealing with stuck-at faults, on the other hand, even 16 bits trace buffers allow to observe more than 90% of

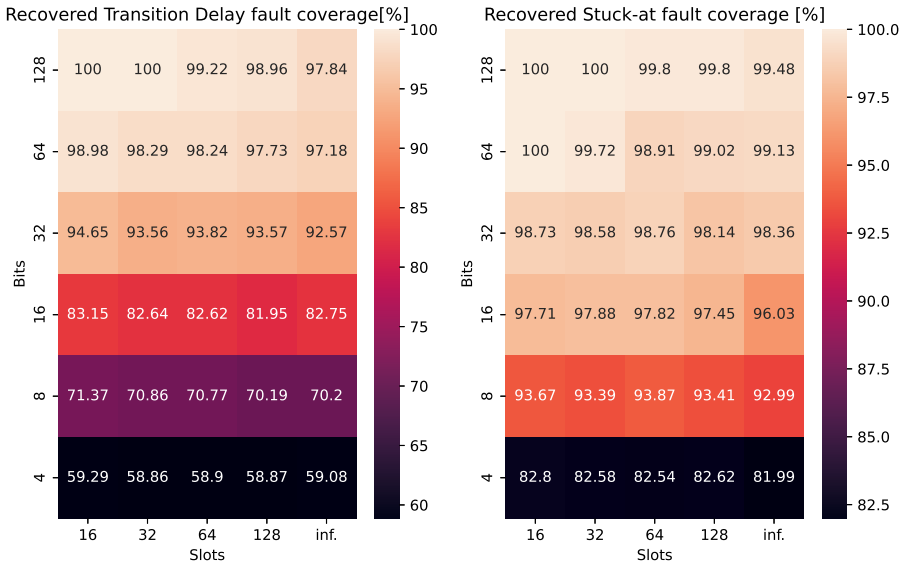


Fig. 4.2 Configurations and percentage of faults recovered for STL1

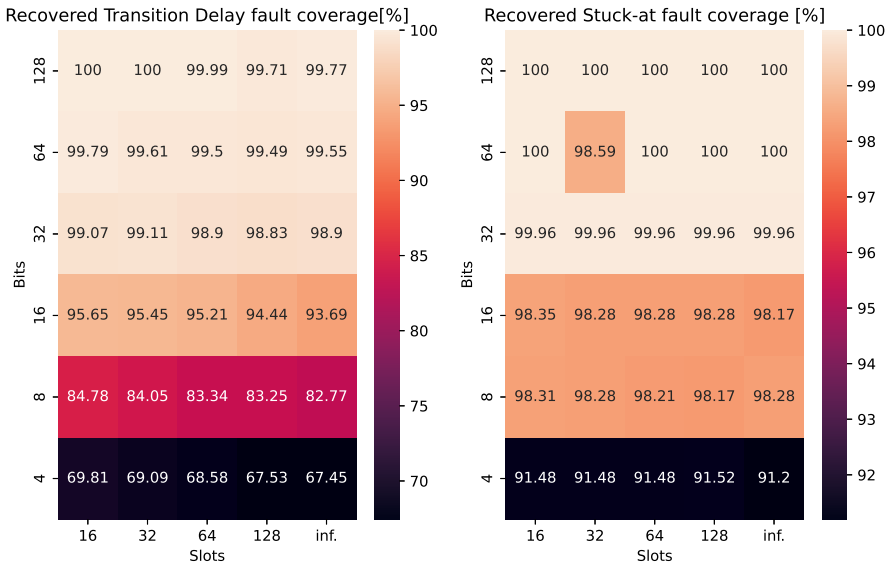


Fig. 4.3 Configurations and percentage of faults recovered for STL2

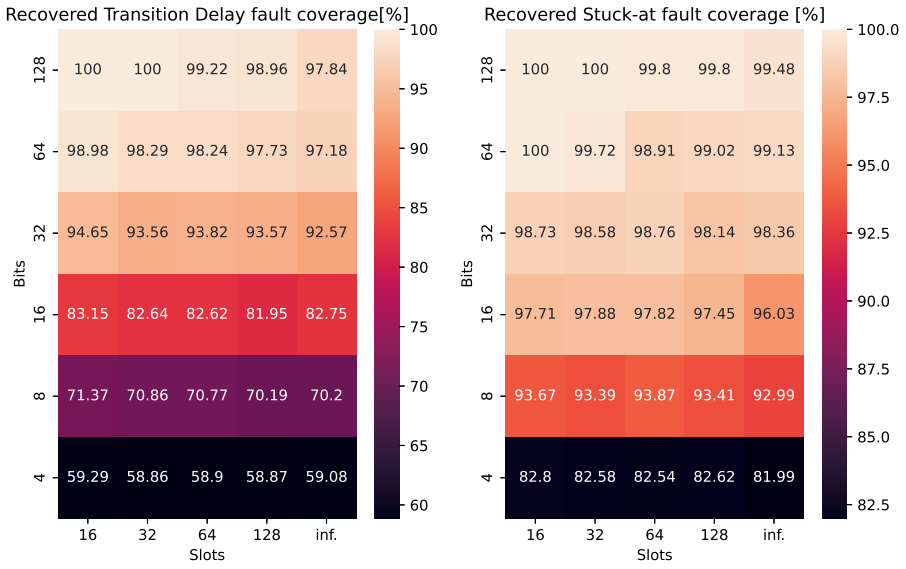


Fig. 4.4 Configurations and percentage of faults recovered for STL3

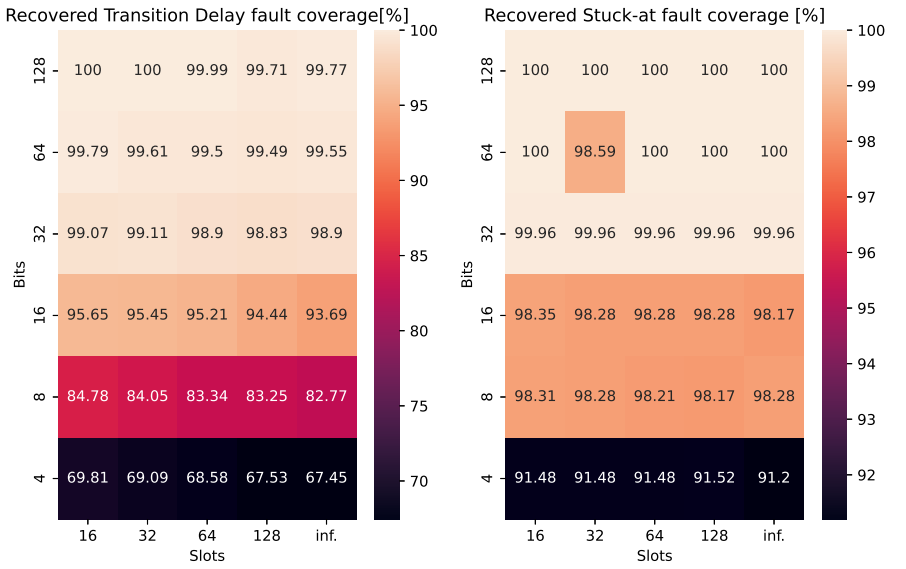


Fig. 4.5 Configurations and percentage of faults recovered for STL4

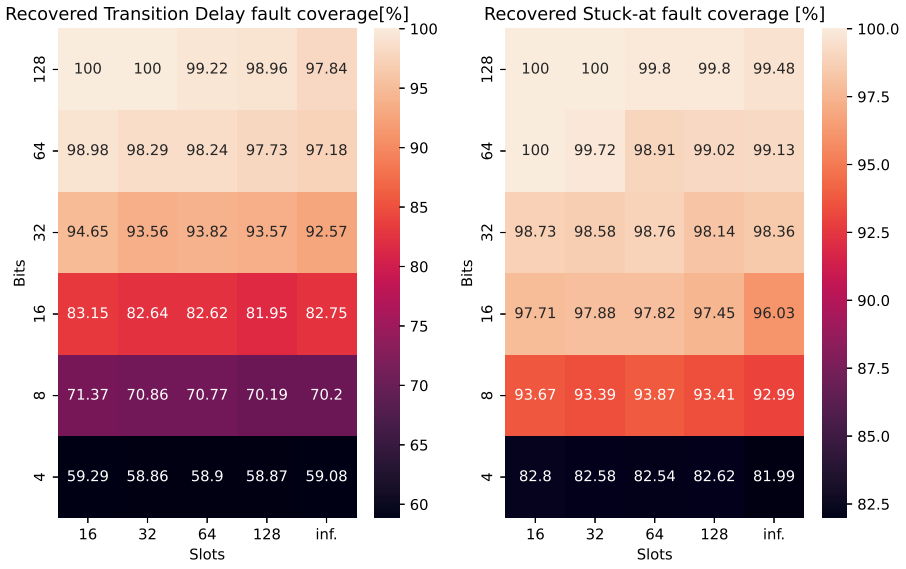


Fig. 4.6 Configurations and percentage of faults recovered for STL5

faults. Out of all five test programs, STL3 is the only one where, even in the best case scenario, it is not possible to recover every not-observed transition delay fault, stopping to a 99.50% of recovered hidden register faults with a trace buffer width of 128 bits and an observation slot 32 clock cycles wide. The reason for this lies in how this STL is implemented: this test program provides a large amount of faults to be observed at the same time, which translates in a large amount of flip-flops to be monitored by the trace buffer. If there are more flip-flops to be monitored than the maximum trace buffer size, this will inevitably lead to the discard of some of them, having some untested faults as a consequence. More sophisticated search algorithms could be implemented to try to recover this situation, or to prove those faults are untestable using a single STL run. Please note that further STL runs would allow the full detection of recoverable faults. Finally, if we take a look at the fault coverages reported in the figures with respect to the amount of observation time slots, we see that the smaller the slot size the higher the final coverage. This is expected, as shorter slots allow for more configurations, hence observing more signals throughout the whole test procedure. Results from the *inf.* column fluctuate and are slightly better or slightly worse than those achieved by having time slots of 128 and, in some cases, 64 clock cycles. This can be traced back to the peculiarities of the single test program. Some STLs might require more frequent configuration changes to achieve

higher coverages, a feature that cannot be achieved when large observation slots are scheduled with no flexibility.

4.3 Chapter Summary

This chapter presented a methodology to detect hard-to-test, not-observed transition delay faults whose effects reach hidden registers by reusing post-silicon debug hardware that is already available in most modern System on Chips. This approach stems from the fact that the hardware infrastructure can be adopted for testing purposes, enhancing the observability of internal points that would not be otherwise visible under any testing approach that does not involve the use of Design for Testability techniques, without impacting the timing performance of the circuit under test. The proposed approach can be divided into two steps: a fault dictionary generation process, where a first fault simulation is launched to extract the set of all not-observed transition delay faults followed by a second fault simulation with no-fault dropping or, if not possible, the n-detection option enabled so that the tool can extract all possible couples of registers affected by faulty values and time instants for each fault. Once that information is produced in a compact way in the fault dictionary, algorithms for extracting a subset of flip-flops to observe through post-silicon debug logic are provided. Such algorithms are defined based on three possible scenarios that derive from the features and capabilities of the hardware, namely (i) non programmable hardware infrastructure is available, hence always observing the same set of flip-flops, (ii) a programmable hardware infrastructure can monitor a configuration of flip-flops, switching configuration after an arbitrary amount of clock cycles and (iii) programmable hardware infrastructure can monitor a configuration of flip-flops, switching configuration after a fixed amount of clock cycles. In the first scenario, the configuration of flip-flops is created by selecting the first n flip-flops where the most fault effects stop, where n is the trace buffer's size. The other two scenarios require an algorithm that, for each configuration, selects the flip-flops to be observed with a first-come first-served policy. When n flip-flops have been selected or in case the next observation time in the fault dictionary exceeds the clock cycles in which the configuration is kept, a new configuration of flip-flops is created. By omitting information related to the amount of clock cycles a configuration must be

kept in case that value is fixed, the latter algorithm can be successfully adopted even in the second scenario.

The approach has been validated on PULPino, a SoC based on a 32-bit single issue in-order RISC-V core, starting with a set of five Self-Test Libraries developed for stuck-at faults. Experimental results show that the algorithm defined for scenario (i) is capable of recovering all hard-to-test, hidden register transition delay faults but requires to observe a large number of flip-flops to do so, the worst case scenario being 942 flip-flops. A similar behavior is obtained when looking at how many not-observed stuck-at faults whose effects reach hidden registers can be detected as a byproduct of applying this technique for transition delay faults, with the worst case scenario requiring to observe 960 flip-flops to recover all not-observed stuck-at faults. The algorithm defined for scenarios (ii) and (iii), although more complex, closely reflects the real behavior of post-silicon debug logic, and allows for a large detection of hidden register transition delay faults with a trace buffer's width of 32 bits, leading to more than 90% of recoverable transition delay faults being detected. Stuck-at fault results are even better, observing more than 90% of recoverable stuck-at faults even with 16 bits trace buffers. If larger trace buffers can be used, 128 bits trace buffers are capable of detecting every recoverable transition delay and stuck-at fault in almost every case. Finally, looking at fault simulation results with respect to the amount of observation clock cycles, it is evident how the smaller the slot assigned to each configuration, i.e., how many clock cycles a configuration is kept, the bigger the final fault coverage, as more configurations can be loaded while the STL is executed.

Part II

Path Delay Fault Oriented Solutions

Chapter 5

Background

5.1 Path Delay Fault Model

The path delay fault model is, together with the transition delay faults, one of the most popular and adopted delay fault models. Path delay faults are described as the defects that cause the cumulative propagation delay of a combinational path to exceed the timing specifications. A combinational path is intended as a sequence of logic gates that starts from a *startpoint*, may that be a primary input or the output of a sequential element, i.e., a pseudo-primary input, and ends on an *endpoint*, either a primary output or the input of a sequential element, i.e., a pseudo-primary output. As for transition delay faults, for each combinational path it is possible to define two path delay faults, a slow-to-rise and a slow-to-fall, where the rising and falling transitions are intended at the startpoint of the path. Let us consider, for example, the circuit in Fig. 5.1.

In this circuit the path `B-D-E-F-H-Z` is assumed to be affected by a slow-to-fall path delay fault. Looking at the waveforms in the figure, where full lines are the waveforms of the fault-free circuit and the dashed lines those of the faulty circuit, the large delay that affects the signal at the end of the path is not the result of a large delay affecting one single net, rather, it is the combination of delays introduced by gates and interconnects. All signals belonging to the path affected by a fault are called *on-path signals*, while the others are called *off-path signals*. Given that we are dealing with a delay fault model, testing such fault requires a couple of test vectors (t_1, t_2). Looking at the circuit, we can deduce that a test vector pair suitable to detect

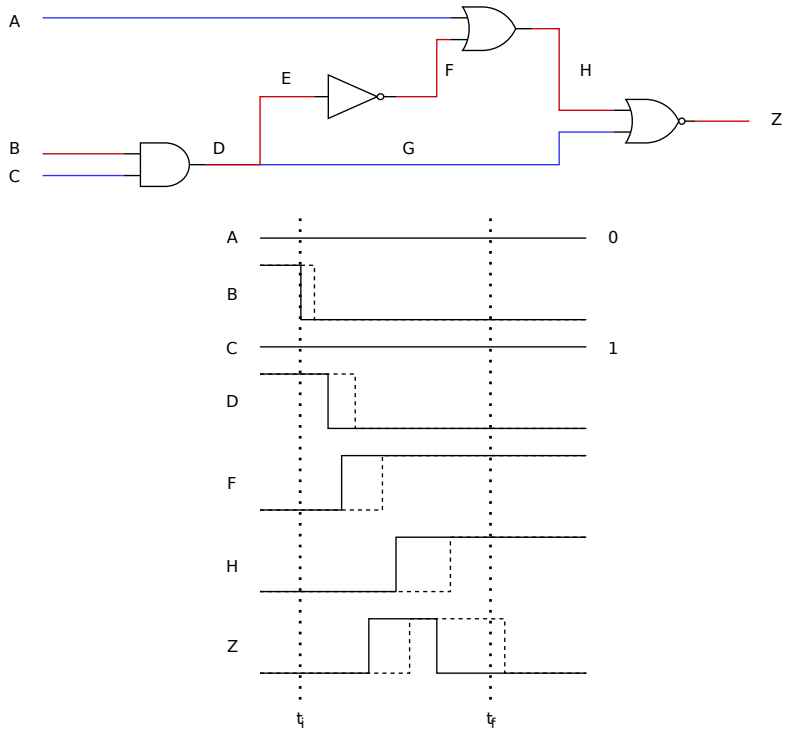


Fig. 5.1 Circuit affected by a STF path delay fault

an STF fault on the considered path is (011, 001): if a slow-to-fall path delay fault affects such path, assuming it is possible to sample the logic value on Z at time t_f, the test vector pair allows to detect the fault as we would observe a logic 1 instead of the correct logic 0. Differently from transition delay faults, when devising test vectors for path delay faults it is possible to introduce the concepts of *non-robust* and *robust* tests. When performing path delay fault simulations, it is customary to consider a single path delay fault at a time, checking whether test vectors are capable of testing it when that fault is the only one present in the circuit. In practice, however, it may occur that there are multiple path delay faults present at the same time in the device under test, and a test vector pair that detects a path delay fault when alone may not be able to do so when other faults are present at the same time. Non-robust path delay tests are tests that are guaranteed to detect the presence of a path delay fault when no other path delay fault is present. An example of a non-robust path delay test can be observed in Fig. 5.2.

The circuit reported in the figure is a pulse generator, whose pulse width depends on the inverter delay. If the most salient parameters of this pulse, e.g., the width and

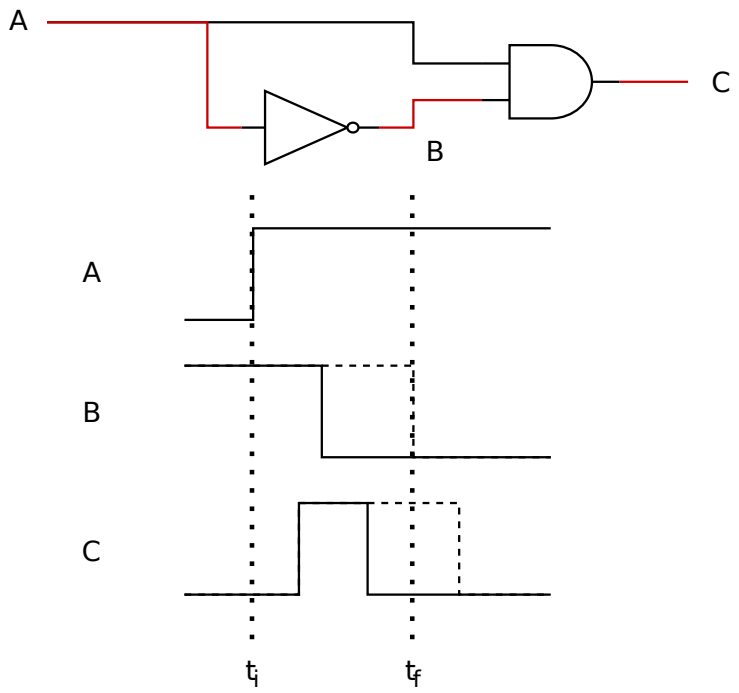


Fig. 5.2 Non-robust test for A-B-C STR fault

position, are important, then analyzing path delay faults on this circuit is necessary. Let us assume that path `A-B-C` is affected by a slow-to-rise fault. A test vector pair capable of detecting such fault is (0, 1), as shown in the figure. If, however, even path `A-C` is faulty, the output C may never toggle, or the pulse could be shifted to the far right, well after the sampling time t_f. Either way, the presence of the path delay fault cannot be noticed, as the output has the correct logic 0 value when sampling is performed, although for wrong reasons. Such test vector pair, hence, is not-robust. Robust tests, on the other hand, are guaranteed to detect the presence of path delay faults even when other faults are present at the same time. Not every path in a circuit is testable. Generating a test vector pair for the path `B-D-E-G-Z` highlighted in red in figure Fig. 5.3 for the slow-to-fall fault is not possible: the second vector of the pair requires the off-path input `C` to be set to a non controlling value, i.e., a logic 0, which however sets the off-path signal `H` to a controlling value, thus masking the transition at the observed output `Z`.

Finally, even though transition delay and path delay fault models are quite different, the same testing techniques can be defined for the two fault models, both when resorting to DfT solutions such as Launch on Shift, Launch on Capture or

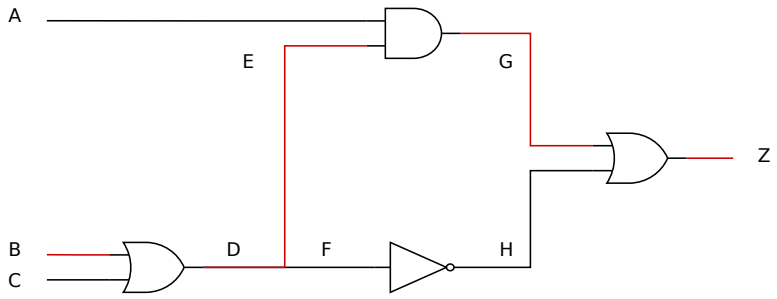


Fig. 5.3 The STF fault affecting the path in red cannot be tested

Enhanced Scan, or functional solutions such as a Software-Based Self-Test approach as first described in Section 1.2. As it will be described in Section 5.2, however, functional solutions for path delay faults are not as mature as DfT ones: an SBST-based fault simulation on sequential circuits targeting path delay faults is currently not supported by any commercial fault simulation tool.

5.2 Related Works

Numerous works on path delay faults have been published in literature. Starting off, the paper [62] reports a comprehensive overview of the state-of-the-art on delay faults, introducing terms and definitions for several delay fault models, together with details on delay fault simulation and test generation and new applications where delay fault testing can be of interest. The work in [63] describes techniques on how to remove long false paths to improve the final path delay fault coverage. This study stems from the fact that paths that are fanout free and not non-robustly testable with respect to at least one transition can be removed from a circuit with no final change in its functional behavior. The authors introduce algorithms based on the adoption of Binary Decision Diagrams (BDDs) to determine whether a path is testable or not. As reported in the experimental results, identifying and removing these paths benefit the overall testability, as it allows to achieve complete non-robust testability of paths and to improve robust testability. This approach has been validated on combinational parts of circuits in the ISCAS-89 benchmark, achieving an increase in path delay fault testability with a limited gate overhead and even a delay reduction for most circuits. The article tackles an important issue, as tools used for extracting most critical paths tend to perform topological analysis on the circuit without considering whether it

is actually possible to propagate transitions from its startpoint to its endpoint, thus artificially lowering fault coverages. However, using BDDs for larger, more complex circuits may not be a feasible task. The methodology presented in this PhD thesis is capable of effectively pruning untestable paths through fault simulations only.

Several works focus on path delay fault testing by resorting to DfT-based approaches. The paper [64] proposes a technique based on the adoption of a single input change test pattern generator (SIC-TPG) that creates test vector pairs, making it suitable for path delay fault detection. The SIC-TPG architecture is described as comprised of a Single Input Change generator, implemented as an n-stage counter followed by a gray encoder circuit, connected to other circuitry that is responsible for shifting by one the output of the SIC generator every time it reaches its final value. This technique is suitable for testing both stuck-at faults and path delay faults, as demonstrated by experimental results on a set of circuits from ISCAS-85 and ISCAS-89 benchmarks, showing fault coverages of at least 90% path delay faults on every circuit, with a limited hardware and timing overhead due to introduction of the SIC-TPG module. Such technique, although effective for small circuits, does not lend to complex circuits such as modern pipelined processor cores. When tackling large circuits, test vectors must be produced in an efficient way either resorting to ATPG-based pattern generation leveraging on scan chains to apply such vectors or by converting them into test programs. The SIC-TPG architecture introduced in the article is rather simplistic and could require large amount of test vectors to reach sufficient fault coverages. The paper [65] focuses on the generation of functional test vectors to be applied through broadside (launch on capture) testing targeting path delay faults. One of the problems associated to DfT techniques is that it can lead to overtesting, i.e., the detection of faults that will never be excited under any functional scenario which may lead to a yield loss. Moreover, paths that cannot be excited with functional vectors do not need to be optimized for speed, having a propagation delay that may even exceed the clock period. In order to generate functional test vectors, this approach first identifies an initial state s_{init} that is a known state prior to the execution of functional operations and then scans it through scan chains. After that, two functional input vectors are applied to primary inputs in two consecutive functional clock cycles while pseudo-primary inputs are taken care of by the circuit itself as it evolves its internal state from the initial state scanned in and the input test vectors, thus completing the application of a pair of test vectors. The test procedure than populates the list of known reachable (thus, functional) states

as test vectors are generated, based on the assumption that if <s_1, s_2> is the pair of internal states that are reached by the circuit while conducting the test and s_1 is a functional state, the functional vector applied to primary inputs that allows the internal state transition from s_1 to s_2 makes the latter reachable as well. In this way, functional broadside tests avoid the detection of functionally untestable path delay faults and deals with overtesting. This approach is validated on a set of circuits from the ISCAS-89 benchmark and supported by experimental results. Finally, the article presented in [66] defines a methodology to test gate-exhaustive path delay faults and introduces a path selection procedure to support the test generation. The author notes that, when dealing with path delay faults, propagating transitions through lines that belong to a path is similar to testing transition delay faults on those lines. Two-cycle defect-aware and cell-aware faults are similar to transition delay faults, in the sense that they model defects that are localized on single lines, but require more complex activation conditions with respect to the transition delay fault model. The premises of this study is that, detecting gate-exhaustive faults, which constitute a superset of cell-aware faults, along a path allows for detecting distributed defects with more complex activation conditions than transition delay faults. The author first proceeds to first define the fault model, than provides techniques for extracting such paths, also addressing the issue of untestable paths by including techniques to identify and remove untestable faults. Experimental data gathered from a set of ISCAS-89 and ITC-99 circuits show that detecting gate-exhaustive path delay faults is a feasible task. DfT-based approaches can achieve significant results on circuits, but present all the downsides mainly related to the problem of power consumption, overtesting if functional vectors are not generated, area and timing overhead and even test application time that may be large enough not to be executed in the idle time slots that may present during the operative lifetime of the device to be tested. In addition to that, processor cores are much more complex than benchmark circuits, thus requiring different testing approaches.

Functional, Software-Based Self-Test solutions for path delay faults have been investigated as well. The work presented in [67] proposes a methodology to generate assembly programs targeting path delay faults on processor cores by exploiting an evolutionary algorithm. Paths are initially grouped into structurally coherent path-delay fault lists, so that the evolutionary algorithm can generate test programs more efficiently: since paths belonging to the same group share the registers from which the startpoint, the endpoint, and the off-path signals originate, it is highly probable that

the same code portion is able to sensitize all those paths. The evolutionary algorithm is initially fed with this information and, starting from a population of randomly generated assembly programs, applies a series of genetic operations, i.e., crossover and mutation, to refine those test programs with the aim of obtaining one test program capable of exciting at least one path from each structurally coherent fault list. For each generation of test programs, besides improving the fitness parameter, i.e., the ability of sensitizing paths, the tool aims at reducing the execution time of each program. The final step consists of running the previously generated test programs on an hardware accelerated fault simulator implemented on an FPGA to refine them so that they can achieve the highest possible path delay fault coverage. Experimental data has been gathered on an open source implementation of an Intel 8051 core, for which the evolutionary algorithm required more than 4 days to generate a set of test programs capable of detecting most of the testable path delay faults. The article in [9] is an extension of [67], including a Binary Decision Diagram step that allows to quickly identify and remove structurally untestable faults. Another approach largely used when developing STLs resorts to the adoption of ATPG for test pattern generation. Work [13] proposes an automatic method to generate self-test programs based on three steps. An initial step consists in the application of a path classification algorithm through which paths whose faults that cannot be tested by instructions from the ISA are identified and not taken into account in subsequent steps. This step is important, as merely checking whether a path is structurally testable is not sufficient due to the fact that functionally testable paths are a subset of structurally testable paths. The second step employs a constrained gate-level ATPG that is launched in order to generate deterministic test vectors for functionally testable paths. In and of itself, an ATPG is not aware whether the test vectors it generates must be used in functional scenarios or not, hence why constraints on the signals that are extracted in the first phase are applied to get vectors that can be translated into instructions. The final step is the conversion of such vectors into instructions, finally obtaining the test program that comprises of a signature generation algorithm so that values are compacted and stored into a non-volatile memory. This technique is validated on two example microprocessors, an 8-bit Parwan processor and a 32-bit DLX processor, both non-pipelined. Results achieved on these two processors show that this technique can detect 99.80% and 96.30% of testable path delay faults on the Parwan and DLX processors, respectively. The work in [10] presents a technique to develop test programs based on the adoption

of virtual constraint circuits (VCCs) targeting path delay faults belonging to critical paths found in the datapath of processor cores. Virtual constraint circuits are small modules that are attached before and after the circuit under test. The authors first employ a set of functional patterns that are used as training tests. By observing and analyzing the input and output values on the interface of the circuit to be tested, input and output VCCs are produced. The input VCC is used to summarize the input constraints on the module under test based on the simulated test patterns, and similarly on the output ports for the output VCC. Once the VCC are ready, they are connected to the input and output ports of the module to be tested, and then a structural ATPG is applied on the module to be tested wrapped by the two VCCs to generate test vectors for the targeted path delay faults. VCCs are also used to prune false paths, thus avoiding any computational effort on faults that cannot be tested in any way. The approach is validated on the Open RISC 1200 (OR1200) core, and is capable of detecting 97.6% of path delay faults that can be structurally, i.e., with the adoption of DfT techniques, faults. The authors also provide a comparison with verification-oriented patterns, showing how they cannot be used as is given that they are intended for a different use and are not optimized neither in terms of fault coverage nor in terms of execution time. The authors in [11] propose a method for developing test programs on pipelined processor cores for path delay faults by using RT and gate level information. The first step of the approach presented in [11] consists of manually generating a Pipelined Instruction Execution (PIE) graph based on the RT level description of the core to be tested and its instruction set architecture. Such graph is used to classify paths into four groups, namely functionally testable (FT), functionally untestable (FUT), potentially functionally testable (PFT), and parity check functionally untestable (PCFUT) paths, and a set of constraints for the subsequent test pattern generation. The constrains generated through this step are then used in conjunction with an ATPG is used to generate test vectors for PFT paths. A Path-Oriented Decision Making (PODEM) based test generation algorithm is used to generate test vectors under functional constraints. Finally, such vectors are converted into instructions and stored in the test program. This approach can be used for paths found in the datapath and controller unit. This technique is tested on two pipelined processors, a 5-stage pipelined, 16-bit RISC core (VPRO) and a 5-stage pipelined, 32-bit DLX processor and is capable of testing all testable path delay faults. [68] presents a technique to generate instructions to test delay faults on pipelined processor cores. This is done in three steps, namely an ATPG-based

delay test generator to be applied to the combinational portion of the device under test, a verification engine based instruction mapper, and a feedback mechanism. The generation of functional instructions to test delay fault is done by means of a set of Verilog properties to be fed to a bounded model checker. This, however, can be quite taxing, as properties are built looking at inputs of paths to be tested, leading to a large number of properties to be fed to the bounded model checker. Moreover, writing all these properties requires a non-negligible amount of manual work. In this paper, we propose a semi-automatic methodology to developing functional constraints, capable of identifying all inputs that cannot be controlled once sub-modules of the DUT are provided. [69] proposes an algorithm, based on formal techniques, that takes the gate-level description of a pipelined processor as input and generates a sequence of assembly instructions able to stress any module within it by maximizing the switching activity. Although effective, this methodology is geared towards the generation of stress-oriented assembly instructions, while our goal is to excite and propagate faults affecting paths inside CPUs. In addition, formal techniques may require a non-negligible amount of time. [70] introduces an approach to generate instruction sequences for SBST, and makes use of a Validity Checker Module to limit test sequences to valid RISC-V instructions and the given environment. This approach deals with the well-known stuck-at fault model, which is quite different with respect to the path delay fault one. Moreover, for larger circuits it was not able to complete the generation of SBST-based routines. My contributions in STL generation for path delay fault testing are presented in [18, 71] and are described in details in Chapter 7 and Chapter 8, respectively.

The functional, SBST approaches described above show that SBST can be effectively used to test path delay faults on processor cores. Papers [67, 9, 13], however, do not deal with the presence of pipeline registers, a feature that most modern processor cores have, thus not tackling the additional complexity introduced by such registers. Moreover, evolutionary algorithms presented in [67, 9] may require a higher amount of time for larger, more complex designs. The methodology discussed in [10] only focuses on faults stemming from paths in the datapath: as shown later, however, faults associated to critical paths do not necessarily all come from the execution unit and require additional care. Finally, deriving graphs in [11] is still a manual task, and may be non-trivial when dealing with larger cores.

Chapter 6

Main Contributions

The path delay fault model is more accurate than the transition delay model in representing the effects of small delay defects along paths in a circuit, but SBST solutions for it are not as mature. The majority of works resorts to DfT approaches on benchmark circuits, a different scenario than testing a processor core through functional methods as test vectors applied through SBST techniques require to be mapped into instructions, i.e., must be *functional* test vectors, a significant constraint that is not present in DfT approaches. Even in the case of works tackling the issue of developing STLs for processor cores targeting path delay faults, they either focus on non-pipelined circuits or require a non-negligible manual effort. Prior to that, it is noted that no commercial fault simulation tool even supports the functional path delay fault simulation on sequential circuits.

The final goal of my PhD thesis in this field consists of developing a systematic methodology to generate Self-Test Libraries for path delay faults in modern in-order pipelined processor cores that require as little manual effort as possible. In order to do so, the work conducted during my PhD mainly focuses on two aspects:

- The development of a test framework that allows to perform path delay fault simulations on sequential circuits when test stimuli come from the execution of Self-Test Libraries by leveraging commercial fault simulation tools,
- The generation of a methodology to generate effective STLs for path delay faults on in-order pipelined processor cores, providing and discussing in details the related fault coverage figures.

Chapter 7 presents the test framework implementation, describing the details and providing the rationale behind all steps. The test framework is validated by launching path delay fault simulations on a processor core based on a set of STLs developed for other fault models, which also provides a reference coverage that can be reached with state-of-the-art STLs for other fault models. Last, Chapter 8 outlines the STL development strategies and presents the achieved results on a modern pipelined CPU, proving its effectiveness.

Chapter 7

Path Delay Fault Simulation Flow

The path delay fault model and fault simulations targeting it are currently supported by commercial fault simulation tools only through the adoption of scan chains, and for manufacturing test, only. This implies that, tests vectors are scanned in and out through scan chains, and only a few functional clock cycles can be applied to the DUT for each test vector. Ideally, a functional fault simulation on sequential circuits for path delay faults would require (i) the definition of a set of paths from which path delay faults are derived, (ii) reading of a set of stimuli applied to primary inputs derived from the execution of an STL, usually in the form of a VCD file, and finally (iii) a fault simulation process where stimuli are fed to the circuit as it evolves for as many functional clock cycles as the test needs, looking for possible mismatches at primary outputs that would signal the detection of faults. What commercial fault simulation tools offer, after extracting paths and generating faults from there, is summarized as follows:

- Load a test vector through scan chains,
- Apply a finite number (usually small) of functional clock cycles for the vector to propagate through the circuit and capture the response,
- Download the response through scan chains, then repeat for all test vectors.

This mode is capable of fault simulating the circuit under test in a functional fashion, provided that such circuit is equipped with scan chains. The amount of functional clock cycles, however, is rather small compared to those needed by a

typical STL; moreover, it requires the presence of DfT hardware. This is not sufficient for a full sequential fault simulation targeting path delay faults. For this reason, prior to the definition of strategies to develop STLs for path delay faults, a detailed description of the architecture of a test flow that allows functional fault simulations on path delay faults is provided.

This flow is devised such that, by providing the RT-level description of the DUT and a test program to be executed, the behaviour produced by each fault is evaluated. This allows identifying faults detected by the test program. It is important to notice that, since this flow is devised for functional testing, the netlist is not required to be equipped with any scan chain architecture as it will not be used for testing purposes. This implies that any fault is only observable, and hence detectable, through at least one primary output. Alternative observation mechanisms for test programs exist, such as checking the memory content at the end of the program execution [72], or observing the response of available safety mechanisms for in-field testing of safety-critical systems.

A schematic representation of the path delay fault simulation flow is given in figure 7.1. The test flow can be divided into a series of preliminary steps, required to prepare all the necessary data, followed by the actual fault simulation. The preliminary steps consist of a synthesis of the DUT, followed by the path extraction by means of a Static Timing Analysis (STA) tool and the generation of input stimuli obtained by performing a logic simulation of the test program. Then, the fault simulation process can be launched targeting path delay faults on the extracted paths. The fault simulation is divided into two steps, the first one is performed on the combinational modules of the DUT, the other one propagates faults observed at the combinational level to the POs through the pipeline stages of the sequential circuit.

In the following sections, a thorough explanation of all different steps is given.

7.1 Synthesis

The first preliminary step required by the proposed test flow is the synthesis of the circuit to be tested based on its RT-level description. The most important aspect of this procedure is that the circuit is synthesized in a way such that the combinational cells of the gate-level netlist are grouped together separately from the sequential

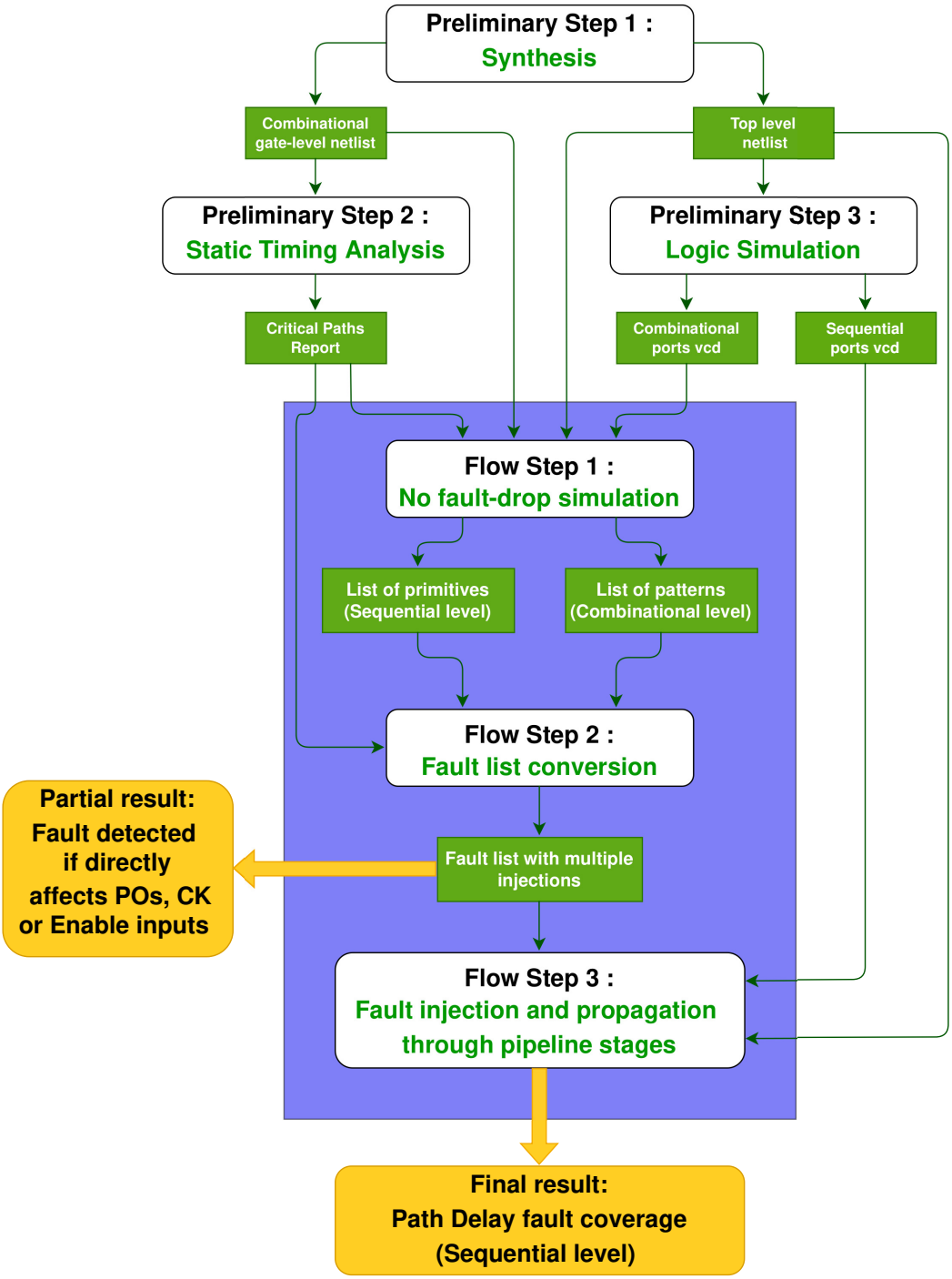


Fig. 7.1 Path delay test flow diagram

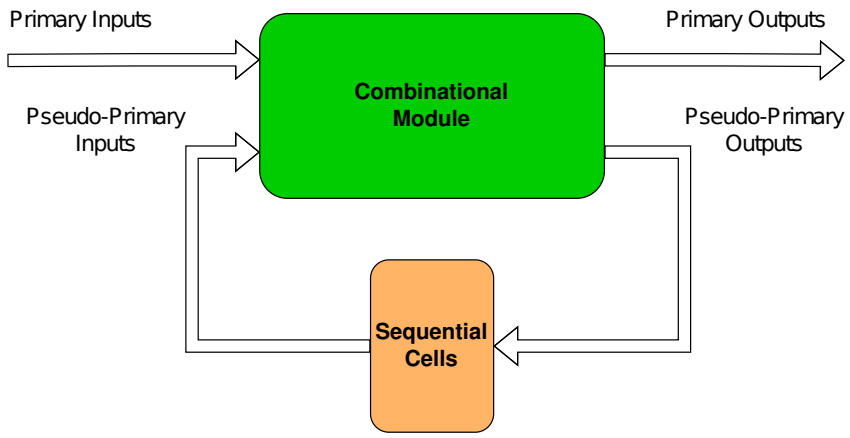


Fig. 7.2 Huffman model for a generic sequential circuit

ones. In this way, we create a large submodule within the CPU that comprises all combinational cells; such submodule is then connected to sequential cells in the processor gate-level netlist, also known as *Huffman model* Fig. 7.2. Besides this aspect, no further constraints are required for the synthesis process.

Two netlists are then produced, one that is specific for the submodule holding all combinational cells, hereinafter referred to as *combinational netlist*, and one for the top-level module including both the combinational submodule and sequential cells, hereinafter referred to as *top-level netlist*. The output signals of the combinational netlist are either POs or PPOs, in case they are connected to the output signals of the top-level netlist or to inputs of sequential cells, respectively. Generating both the combinational and top-level netlists is important, as because in Section 7.4 the combinational netlist will be used as the module under test, while in Section 7.5 the top-level netlist will be the one used for simulations.

7.2 Logic simulation

The second preliminary step consists in performing a logic simulation of test programs using any available logic simulation tool, with the main goal of generating input patterns for the subsequent fault simulation process. This step requires that the top-level netlist is instantiated as a component in the testbench, so that test program golden responses can be recorded both for combinational and top-level circuits, usually in a VCD format. From now on, such golden responses are referred to as

patterns lists. Such lists contain the value held by every combinational/top-level input and output port at any clock cycle during the execution of the test program, and will be used in Section 7.4 as test vectors for the fault simulation.

7.3 Static Timing Analysis

As a last preliminary step, the test flow generates the list of paths to be tested during the fault simulation. As described in Section 5.2, several techniques for extracting paths have been investigated. This test flow uses a Static Timing Analysis (STA) tool to produce a list of combinational paths in ascending order of slack. In this way, in case the amount of extracted paths is too high, only the subset of combinational paths with the most stringent timing requirements is considered. The advantage of using a Static Timing Analysis tool is that it only requires the gate level netlist of the DUT, thus being easily automatable with respect to other solutions that may require manual effort from the test engineer. Nevertheless, it is worth mentioning that STA tools are very pessimistic in performing their analysis and are not able to recognize *false paths*; such paths cannot be sensitized in the final design and would introduce untestable faults in the fault list. Therefore, a subsequent pruning of those paths from the initial path list is needed; this is only partially performed by commercial fault simulators as a preliminary phase of the fault simulation. The benefits of refining the path list are non-trivial: if the list contains paths that cannot be tested by any means, the test engineer will fruitlessly try to activate and detect faults whose effects cannot be observed in any way, and the fault coverage will artificially drop. Pruning of false paths is a topic that has been studied in literature, with several available approaches. As an example, the authors of [73] developed an algorithm to prune untestable paths, taking into account the circuit topology, process variations, and aging effects; remarkably, this reduced the path count by 70.87%.

Path pruning is performed by employing a commercial Static Timing Analysis tool in conjunction with a path delay fault oriented ATPG in an iterative way. The slack range is divided into sub-ranges, then for each sub-range the STA tool extracts a list of paths that is then tested by the ATPG taking the combinational netlist as a device under test. A detailed explanation of this process is reported in Fig. 7.3.

The rationale behind this approach is that there exist a relationship between structurally testable faults, i.e., faults that can be tested by the ATPG without any

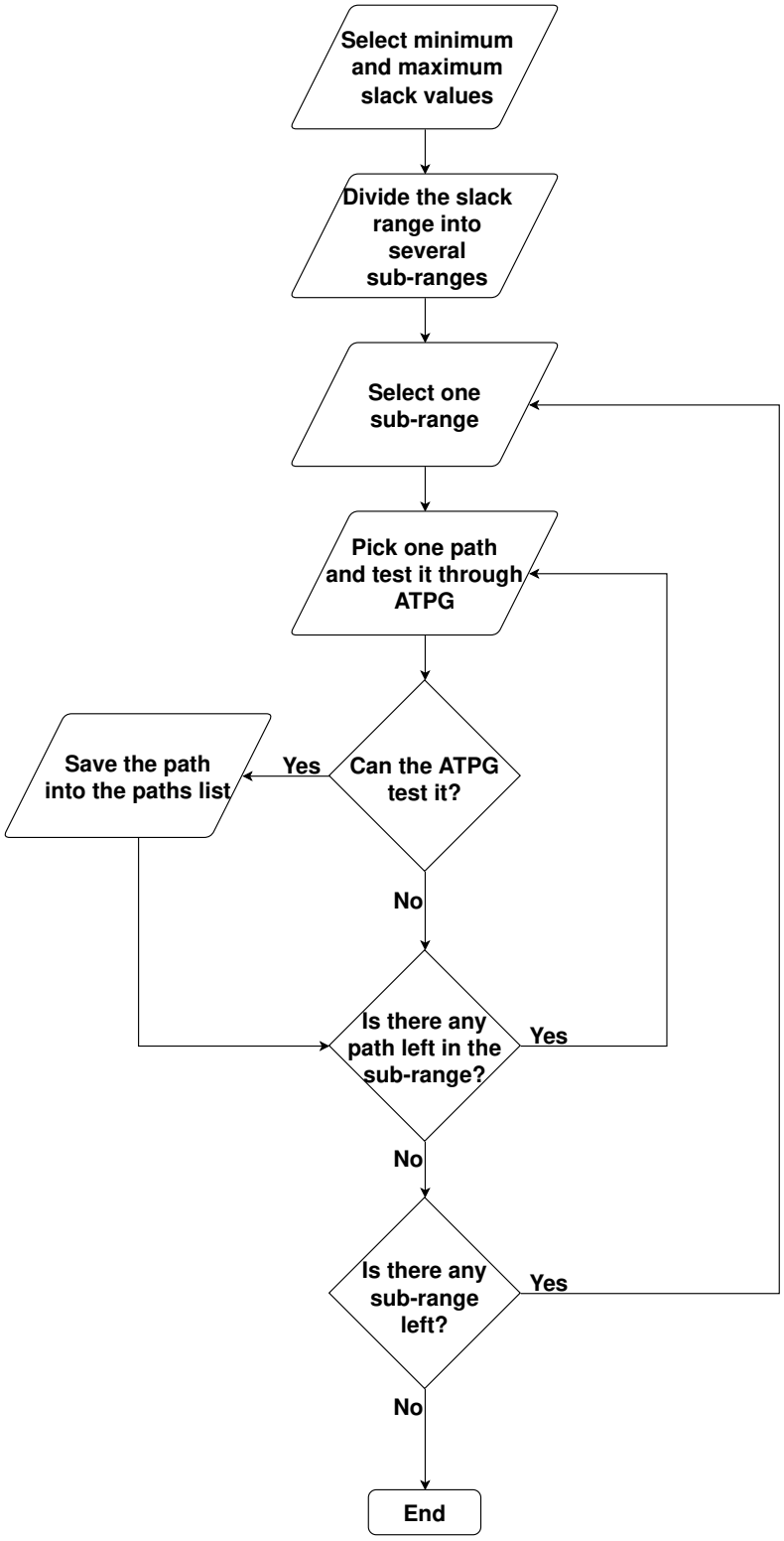


Fig. 7.3 Paths extraction flow

constraint on the type of test vector, and functionally testable faults, i.e., faults that can be tested by test vectors that can be obtained in functional scenarios: if a fault is functionally testable then it is structurally testable too, vice versa, a structurally untestable fault is functionally untestable as well. Iterating through several small slack intervals allows to carefully pick the largest amount of testable paths throughout the whole slack range. In this way, we are sure that structurally untestable faults are excluded. Excluding all structurally untestable faults, however, is not enough when resorting to functional in-field testing approach, e.g., for safety-critical applications, as there may be some structurally testable faults that cannot be properly simulated or observed within functional scenarios and usually identified as functionally untestable. Consequently, it is crucial to identify as many functionally untestable faults as possible, removing them from the list of target faults to be considered.

The path list that is generated in this preliminary step is then used in the following fault simulation steps. Hereinafter, the set of produced paths is referred to as *path definition* list. Once the path definition list has been generated, it can be modified as needed: the proposed flow accepts any path definition list, hence every possible optimization or subset extraction is allowed.

When extracting paths for the subsequent fault simulation, timing information such as the slack associated to the path may be included as well. When performing path delay fault simulations, however, current commercial tools tend to neglect such timing information. For this reason, all data regarding slack or timing behaviors can be safely omitted in the path definition list.

7.4 Combinational-level fault simulation

Once the preparatory steps are cleared, the test flow moves on to the fault simulation process that is divided into two steps. The reason for dividing the fault simulation process in two stems from the fact that, similar to what has been outlined for transition delay faults in Chapter 3, the act of testing a path delay fault can be decoupled in:

- Checking whether a couple of test vectors is capable of sensitizing a path and propagating the required transition towards its endpoint,
- Propagating the faulty value due to the presence of a fault in a path towards one of the primary outputs where it can be observed.

The proposed test flow leverages on the fact that both steps can be carried out with commercial fault simulation tools, taking care of correctly setting and coordinating both simulation steps. In the first step, a fault simulation tool is used to feed, at any clock cycle, the input ports of the combinational netlist with patterns produced by the test program and stored in the Patterns list. This process allows to identify all path delay faults that produce a difference on at least a PO or a PPO when the considered test program is executed. Moreover, this first fault simulation step allows to identify the clock periods when this happens and the specific POs or PPOs affected by each fault. This information will be used at a later time on the top-level netlist. Fault simulators can also report, given any detected fault, which and how many patterns are able to detect it; however, for optimization reasons the simulators might drop, i.e., remove from the active fault list and hence stop considering, some faults that are either already covered in other vector or that are not getting detected in a given time window. This analysis is accurate only when fault simulations are performed without fault dropping, that is, whenever a fault belonging to the active fault list is never dropped from it after being detected by any pattern.

The fault simulator reads the combinational netlist, the library files, the path definition list, and the combinational patterns list. More in detail, netlist and library files are used to build an internal model of the device under test, while the path definition list is analyzed to exclude false paths — hence, untestable faults — from the simulation. Lastly, signals included in the patterns list are interpreted as a list of pairs of vectors to be applied in sequence.

More accurate results in the overall flow can be achieved by running fault simulation without fault dropping; disabling fault dropping, however, significantly increases the fault simulation time. For this reason, by default, fault simulation tools drop every fault after being detected once. As previously mentioned, enabling the no fault-dropping option consists in never deleting faults, even when detected, from the active fault list. As a consequence, for each fault, it is possible to obtain all patterns detecting it at the combinational level, instead of just the first one. This allows us to consider, in the following fault simulation step, the propagation of fault effects through the sequential logic not only for the first pattern, but for the whole test set.

7.5 Sequential-level fault simulation

Once we know which output of the combinational netlist is possibly affected by a certain fault at a specific time step or clock cycle, the final step that is required to complete the fault simulation flow is to propagate the fault effect throughout the sequential logic and check whether it reaches an observable point. The way this final step is implemented in the test flow is by means of bit-flips injected in the sequential elements that capture the fault effect. If the fault simulation tool used in the previous step is capable of performing this task, then it can be used for the second step as well, else another tool must be used and data from the combinational and the sequential fault simulations must be adapted so that data produced from the first tool is understandable from the second one. To the best of my knowledge, no single tool is capable of performing these two steps, hence the need for using a commercial tool different from the one used in the previous step. The detected faults list obtained with previous strategies was made compatible with the tool used for the sequential-level fault simulation. In details, that means that each detected fault, together with its possible propagation endpoint and the time instant at which it reaches the sequential element, were translated into a bit-flip, applied to the faulty path endpoint at the aforementioned time instant.

It is worth noting that it is not obvious that each detected fault from the combinational circuits that could provoke a bit-flip can, in turn, be propagated to the POs. This is why, in the previous step, the fault-dropping option needs to be disabled: by allowing the generation of more patterns for each fault, it is also possible to generate several bit-flips at different time instants, hence increasing the accuracy of fault coverage evaluation. As a consequence, there may be more than one time instant at which a given fault is detected at combinational level.

Due to this reason, for each fault detected in the sequential-level fault simulation, the *minimum number of patterns* required to detect a fault can be defined. This value can be described in terms of the number of patterns generated by the functional fault simulation at the combinational level needed until the effect of the detected fault is propagated to the POs. The smaller this value is, the easier its test generation at sequential level is.

7.6 STL performance evaluation on Path Delay Faults

In order to validate the described test flow, a set of five test programs developed for the stuck-at fault model has been used to test path delay faults on the RISC-V core found in the PULPino system on chip. This serves two purposes, checking the correctness of the proposed path delay fault simulation flow and understanding how well test programs written for other fault models perform for path delay faults. Moreover, in order to make sure that results were not tool dependent, each step has been cross-validated with similar EDA tools from different vendors.

The test programs suite consists of four programs written by test engineers and one test program generated randomly. The test programs written by test engineers have been implemented following diverse approaches, and represent the state-of-the-art for STLs targeting stuck-at faults in modern in-order pipelined CPUs. The presented experiments have been run on 5 cores of an Intel Xeon CPU E5-2680 v3. The whole functional simulation flow required indicatively 48 to 72 hours for each program.

In order to set a reference for the following analysis, an analysis on the combinational logic of the RI5CY core only was performed, i.e., assuming that its input/output signals are fully controllable/observable with an ATPG process. This first analysis was conducted by simply extracting paths with a slack range from 0, i.e., the critical path, to 5ns, the processor's clock period, through a Static Analysis Tool and feeding them to the ATPG, together with the combinational netlist to see how many path delay faults can be detected in the best case scenario. The ATPG engine produced a PDF coverage of 38.79%, with 13,762 detected faults, and 21,714 untestable faults. This is the ideal upper-bound for the achievable fault coverage that can hardly be achieved even when adopting SBST approaches or even scan-based — i.e., LOC — tests, due to constraints imposed by the test procedure. This result can also be read in another way: more than 60% paths that were extracted by the Static Timing Analysis tool in this design were not testable, thus reinforcing the importance of performing path pruning as described in Section 7.3.

Table 7.1 Combinational-level fault simulation results

	Progr. 1	Progr. 2	Progr. 3	Progr. 4	Random	Cumulative
Test patterns	64,502	36,394	17,269	118,098	32,416	268,679
Detected faults	6,816	6,973	6,856	7,554	6,573	8,085
Fault coverage%	49.50	50.67	49.81	54.89	47.76	58.75

7.6.1 Combinational-level fault simulation

Once the reference is set through the ATPG engine, it is possible to evaluate the fault coverage obtained by running already developed STLs. Given that the fault simulation step is divided in combinational and sequential fault simulations, data regarding the combinational fault simulation is discussed first. Fault simulation results of the test programs on the combinational logic are summarized in Table 7.1.

For each test program, the table reports information about the amount of test patterns corresponding to its execution, the number of detected faults and the fault coverage with respect to the reference experiment. The last column, *Cumulative*, reports the aforementioned data in a cumulative fashion, as if all test programs could be collapsed into a single program.

The fault coverage values of Program 1, 2, and 3 differ by at most one percentage point, although Program 3 is faster. This is of particular importance when considering that the test programs have been devised with different techniques and structures being also very heterogeneous in terms of duration and number of instructions. It also suggests that no correlation between SAF and PDF functional coverages may exist.

7.6.2 Detected by Implication faults

Once the list of path delay faults detected at combinational level is obtained as a result from the previous step, it is possible to categorize them based on the path's endpoints they are related to. This is important, because depending on the endpoint type some faults might belong to the *Detected by Implication* group. When adopting SBST techniques, primary outputs are usually set to be the observation points. A fault directly affecting a primary output is hence automatically marked as detected. The same, however, can be concluded for faults affecting very sensitive signals such

Table 7.2 Number of detected faults per endpoint type

Endpoint Type	Program 1	Program 2	Program 3	Program 4	Random
PO	298	167	193	453	206
FF/D	440	409	209	623	188
FF/Clock	1	1	1	1	1
Latch/D	6,044	6,363	6,339	6,444	6,145
Latch/Enable	33	33	33	33	33

as clock gating or enable signals: a delay fault affecting those signals could provoke serious synchronization issues that would most likely cause the whole circuit to fail. Hence, these faults are labeled as *Detected by Implication* in the following, avoiding their further explicit fault simulation.

Table 7.2 reports data on how many faults affect each endpoint type that can be found in the design under test. The endpoints reported in the table are the top-level netlist's primary outputs and clock/enable and data pins of sequential elements, like flip-flops and latches. These data are reported for each test program. As shown in Table 7.2, the program that achieved the best results in every field was Program 4, as it detected 453 faults at POs, 623 faults at FF/D pins and $6,444$ faults at Latch/D pins. On the other hand, program random achieved the worst result in terms of faults detected at FF/D pins (188) while program 2 was the worst in terms of faults detected at POs (167) and program 1 the worst in terms of faults detected at Latch/D pins ($6,044$).

From this data, it is possible to conclude that, among all faults detected at combinational level, 332 faults are detected by implication in Program 1, 201 faults are detected by implication in Program 2, 227 faults are detected by implication in Program 3, 487 faults are detected by implication in Program 4 and 206 faults are detected by implication in the Random program, thus not needing any further simulation.

7.6.3 Sequential-level fault simulation

All faults that do not belong to the detected by implication group must now be checked to see if their effects are noticeable at the primary outputs of the whole device under test, thus constituting the set of faults to be tested in the sequential level

Table 7.3 Functional fault simulation results

Parameter	Program 1	Program 2	Program 3	Program 4	Random
Injected	6,484	6,772	6,629	7,067	6,333
Det. by Simulation	4,797	5,420	4,781	6,660	5,011
Det. by Implication	332	201	227	487	206
Fault Coverage%	37.27	40.84	36.39	51.93	37.91
Prop. Coefficient%	75.25	80.61	73.04	94.61	76.23

fault simulation. In order to decrease the fault simulation cost, all faults producing a bit-flip on the same pseudo-primary output during the same clock cycle have been grouped together. This is done because, even though faults may stem from different portions of the device under test, in the sequential-level fault simulation they are equivalent since they affect the same flip-flop at the same time. Table 7.3 presents results of the sequential-level fault simulation.

Row *Injected* reports the amount of injected faults in this last fault simulation process which is equal, for each program, to the total amount of faults detectd at the combinational level minus the faults detected by implication, reported in the *Det. by Implication* row. The absolute value of faults detected at the sequential level as a result of the fault simulation process is reported in row *Det. by Simulation*. *Fault Coverage%* row gives the fault coverage calculated as the sum of *Det. by Simulation* and *Det. by Implication* faults over the total amount of faults. Finally, the *Propagation Coefficient* is the percentage of faults that have been successfully propagated to a primary output among those faults detected at combinational level.

It is possible to see that program 4 achieved the best results in terms of both faults detected at combinational level ($7,554$) and propagation coefficient (94.61%). This result may be partly due to the longer duration of the program 4. Program Random shows very interesting results as well. Despite the large gap in terms of fault coverage for SAF and TDF, the functional test results are comparable to what has been achieved with other test programs. This brings us again to the conclusion that no correlation among the considered fault models coverage exists.

Table 7.4 Most critical detected faults per program

Program	Scope	Path ID	# Gates	Slack (ns)	Fault
Prog. 1	Comb.	4,323	95	1.776	str
	Seq.	4,323	95	1.776	str
Prog. 2	Comb.	4,321	95	1.770	str
	Seq.	4,321	95	1.770	str
Prog. 3	Comb.	10,873	23	4.199	str
	Seq.	10,873	23	4.199	str
Prog. 4	Comb.	4,321	95	1.770	str
	Seq.	4,321	95	1.770	str
Random	Comb.	7,158	38	3.542	str
	Seq.	11,330	20	4.305	stf

7.6.4 Test programs effectiveness

Finally, an analysis of the effectiveness of the test programs was performed, with the goal of understanding how delay faults on paths with different slacks are covered. Moreover, with this analysis it is possible to see if there is a correlation between detected faults and the associated path slack value.

The results of this analysis are shown in Table 7.4. This table shows, for each test program, the most critical PDF — i.e., the fault associated to the path with the smallest slack — together with its gate counts and slack. Faults were ranked in ascending order according to the slack of the paths they are related to, i.e., the path ranked first has te smallest slack, and an ID was assigned to each fault. For example, the fault with ID #10 is associated to the path with the tenth smallest slack in the ranking.

Programs 2 and 4 were those that achieved the best results as they both detected the same longest faulty path, #4,321, followed by Program 1, with the longest detected faulty path being #4,323. Program 3, on the other hand, did not perform that well, as the longest detected faulty path was ranked #10,873. Remarkably, the random program performed differently between the combinational and the sequential-level fault simulations, where the longest detected faulty paths were #7,158 and #11,330, respectively. Looking at the netlist of the device under test, paths #4,323, #4,321, and #7,158 are located in one of the arithmetic modules instantiated in the

Table 7.5 Fault Coverage per slack range

Slack intervals [ns]	Total faults	Comb. ATPG FC%	Functional test programs FC%
[0.0 - 0.5]	4,672	0.0	0.0
[0.5 - 1.0]	3,228	0.0	0.0
[1.0 - 1.5]	640	0.0	0.0
[1.5 - 2.0]	248	79.4	7.2
[2.0 - 2.5]	388	66.0	7.0
[2.5 - 3.0]	422	64.6	6.6
[3.0 - 3.5]	4,670	5.8	0.5
[3.5 - 4.0]	7,176	12.7	0.3
[4.0 - 4.5]	3,968	56.1	3.6
[4.5 - 5.0]	10,064	95.0	75.1

divider circuit featured inside the ALU; paths #10,873 and #11,330, on the other hand, go through two different arithmetic modules, both being inside the logic circuit used to address the memory stage.

The distribution of the considered PDFs with respect to slack intervals is presented in Table 7.5. Each line takes into account a 0.5ns step interval; if a path owns a slack ranging in that interval, it will be counted in the row. This table reports how many faults were extracted by the Static Timing Analysis tool in the first column, while the second column reports the percentage of faults detected by the ATPG and the third column the percentage of faults cumulatively detected by the test programs. Interestingly, all faults belonging to the slack interval [0.0-1.5]ns were not detected by any test method. These numbers are in accordance with data shown in previous publications [74, 75].

In order to identify the most critical faults and how they are covered by means of functional SBST methods (test programs) and through the adoption of the ATPG, a further analysis on the faults affecting the arithmetic blocks within the processor core under test was performed. Table 7.6 reports such information. It is noted that not all faults from the fault list traverse one of these modules; in some cases, one path could traverse more than one arithmetic block.

Furthermore, it is of interest to understand how much effort is required to propagate each fault from a pseudo-primary output to a primary output, i.e., how many

Table 7.6 Fault Coverage per module

Module name	Total faults	ATPG comb. FC%	Functional fault sim. FC%
id_stage_i_add_531	160	100.0	0.0
alu_i_int_div_div_i_sub_100	12	50.0	33.3
alu_i_int_div_div_i_add_100	446	25.8	25.8
ex_stage_i_mult_i_add_109_2	1,580	0.0	0.0
ex_stage_i_mult_i_mult_109	1,580	0.0	0.0
cs_registers_i_add_775	132	100.0	0.0
load_store_unit_i_mult_add_463_aco	1,884	72.4	21.8
load_store_unit_i_add_463_aco	1,994	73.5	24.6
r1589	868	100.0	0.0
ex_stage_i_alu_i_add_168	6,960	0.0	0.0
ex_stage_i_alu_i_add_182	6,960	0.0	0.0

detections are required for a fault to be propagated from the combinational to the sequential level. As described in Section 7.4, ideally a no-fault dropping simulation would yield the most accurate results, as the more pairs of flip-flops reached by faulty values and time instant at which such event occurs are produced for each fault, the higher the chance that the fault is detected at the end of the sequential fault simulation as well. This however may lead to very time and resource consuming simulations, hence why resorting to a *n-detect* fault simulation, i.e., a fault simulation where a fault is dropped after being detected n times, is often the adopted solution. The aim of this last analysis is understanding how many detections are required for faults that were detected at a combinational level to be detected at the sequential level too. Table 7.7 reports such information focusing on Program 4 specifically. Most of the faults (nearly 80%) are immediately detected, as soon as they produce a difference on a PPO. Only a subset composed of about 20% of the faults require a significant number of differences on PPOs at different clock cycles before being detected. Currently, we are unable to state whether we could speed-up the detection of the latter subset of faults by a more careful design of the test programs, or whether their detection strictly requires longer test programs.

Finally, to summarize, Table 7.8 collects the fault coverage percentages for each program on the three fault models mentioned in this chapter.

Table 7.7 Number of detected faults vs. number of detections on PPOs

Differences on PPOs	1	10	30	64
Fault detected	5,712	6,552	6,918	7,146
(%)	79.9	91.7	96.8	100.0

Table 7.8 Fault coverage summary

Parameter	Program 1	Program 2	Program 3	Program 4	Random
Clock cycles	64,527	36,500	17,308	181,370	32,455
SAF FC%	86.77	81.79	81.37	82.97	59.44
TDF FC%	41.90	44.21	63.16	61.90	24.41
PDF FC%	37.27	40.84	36.39	51.93	37.91

7.7 Chapter Summary

This chapter presents and describes in details a fault simulation flow that is capable of performing a path delay fault simulation on a generic sequential circuit when resorting to a functional, SBST approach. The reason for developing such flow is that, at the time of writing, there is no way to generate fault coverage figures for functional tests of path delay faults through the adoption of commercial fault simulation tools. The goal of this PhD thesis for this part is to provide techniques to test path delay faults on processor cores; nevertheless, the same approach described in this thesis can be used to perform functional tests on any sequential circuit.

The presented fault simulation test flow is divided among three preliminary steps, required for generating all the required data, followed by the actual fault simulation process that has been divided into two sub-steps. The first preliminary process consists of synthesizing the core to be tested, making sure that the combinational logic and the sequential elements are grouped separately, thus creating a combinational netlist where only combinational elements are present and a top-level netlist that encompasses the combinational module connected to the sequential elements. Following, the second preliminary step consists of running a logic simulation of the core to be tested executing the STL, with the goal of recording the input stimuli into a pattern list that will later be used in the fault simulation process as source of test vectors to test path delay faults. Finally, the last preliminary step requires the extraction of paths on which STR and STF path delay faults will be tested. This is done by means

of an iterative approach that features a Static Timing Analysis tool paired with an ATPG. Since the Static Timing Analysis tool is not capable of identifying untestable paths, the path extraction process first produces a list of paths through the STA tool and then discards all paths whose faults cannot be tested through ATPG, following the idea that a fault that cannot be tested with the ATPG cannot be tested through SBST methods, too. Once the preliminary steps are cleared, a first combinational-level fault simulation is performed. The aim of this fault simulation step is to test path delay faults generated from paths in the path list in the combinational level only, thus obtaining a first indication of how many path delay faults are observed at the pseudo primary outputs and primary outputs of the combinational logic. Faults observed at primary outputs are marked as detected, while all the other faults are to be simulated in the sequential level fault simulation, whose goal is to check whether the test program is capable of propagating a fault effect from the pseudo-primary outputs to the primary output. The only set of faults observed at pseudo-primary outputs that are exempted from the latter fault simulation are faults whose values affect sensitive signals, e.g., clock-gating or enable signals, that may cause obvious malfunctioning if reached by a faulty value. Such faults are defined as *Detected by Implication*. In order to maximize the probability of a fault detected at combinational level to be detected in the sequential fault simulation too, the no-fault dropping, or an n-detect if the no-fault dropping is too time and resource consuming, option should be enabled while performing the combinational fault simulation.

The proposed test flow has been validated with a set of test programs previously devised for stuck-at faults on the PULPino core, a 32-bit in-order pipelined PCU. The set of test programs is comprised of four programs developed by test engineers and one program randomly generated for comparison. Experimental results show that the proposed test flow is capable of correctly performing fault simulation for path delay faults and demonstrates how, even in the best case scenario, 51.93% of path delay faults are detected by already available test programs, thus reinforcing the need for defining techniques for developing STLs that target path delay faults specifically.

Before concluding the discussion on the path delay fault simulation flow presented in this PhD thesis, some final considerations should be made. When transitioning from the combinational to the sequential fault level simulations, the test flow records the effects of path delay faults on registers which constitute the endpoint of paths they are associated to, and then propagates those effects by means of bit-flips

in the top-level circuit. It is to be noted, however, that when a path delay fault is present, its effects may compromise values of flip-flops other than the endpoint of the path affected by the fault. The path delay fault simulation flow presented in this PhD thesis constitutes a first approach in developing a flow to tackle path delay faults in sequential circuits, and will include such mechanism in future developments.

Chapter 8

STL Development for Path Delay Faults

8.1 Proposed Approach

This chapter is devoted to the description of an STL development methodology for processor cores targeting path delay faults. In order to validate the effectiveness of this methodology, the previously presented fault simulation flow is used. Such STL development strategy is divided into three steps: functional constraint identification, test patterns generation using ATPG, and conversion of those patterns into instructions. This approach is summarized in Fig. 8.1.

In the following subsections, the three steps are described in detail.

8.1.1 ATPG pattern extraction

To test PDFs, test patterns must be able to generate and propagate specific transitions through the targeted paths. This task, however, cannot be fulfilled by any generic couple of vectors: very few patterns are capable of driving all PIs and PPIs properly. This is especially true when dealing with long paths: using random programs or even programs developed for other fault models, such as stuck-at faults (SAFs) or transition delay faults (TDFs), does not work effectively, and leads to a very small

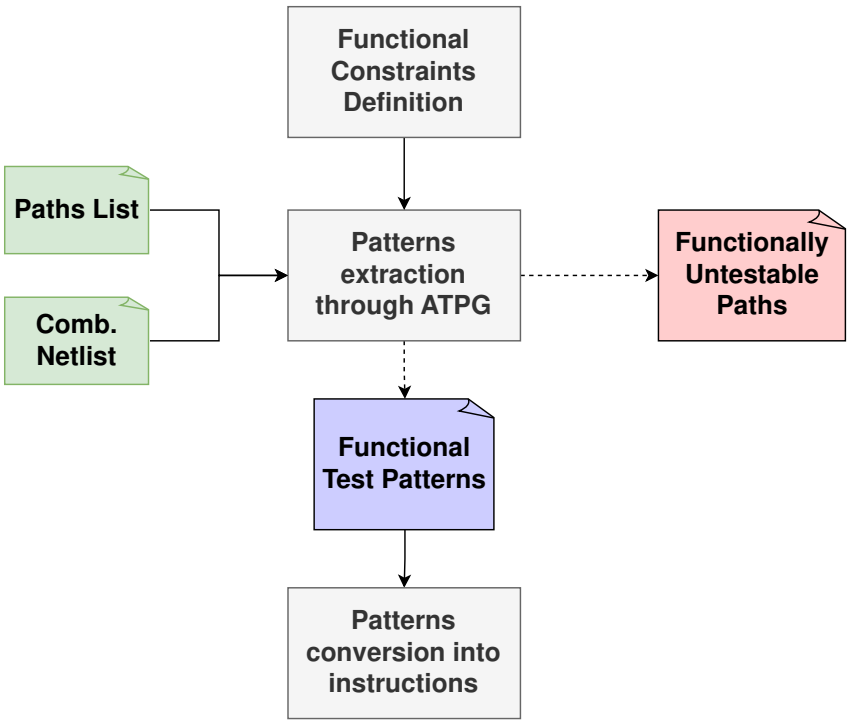


Fig. 8.1 Patterns generation flow

coverage. For this reason, special emphasis on the test pattern generation step should be placed.

The adopted strategy for generating test vectors is summarized in Fig. 8.2.

The pattern generation task is managed by an ATPG, that requires the DUT's combinational netlist and the path list to produce the aforementioned test patterns. This allows producing effective and reliable patterns in a relatively short amount of time. Test vectors generated by ATPG usually feature some don't-care values. Typically, the number of don't care signals is reversely proportional to the path's length. Moreover, it is noted that the more don't care values are found within the test vectors, the easier it is to convert test patterns into instructions, as more degrees of freedom are provided when looking for instructions that can drive signals as specified by the ATPG. For this reason, it would be inconvenient to use fully specified test vectors. If the ATPG supports the generation of multiple test patterns for each fault, enabling such option is beneficial as it increases the pattern-to-instruction conversion rate. Having multiple test patterns provides a higher probability that one test vector can be successfully converted into an instruction. If such an option is available, multiple candidate test patterns for each fault are generated, allowing to select the

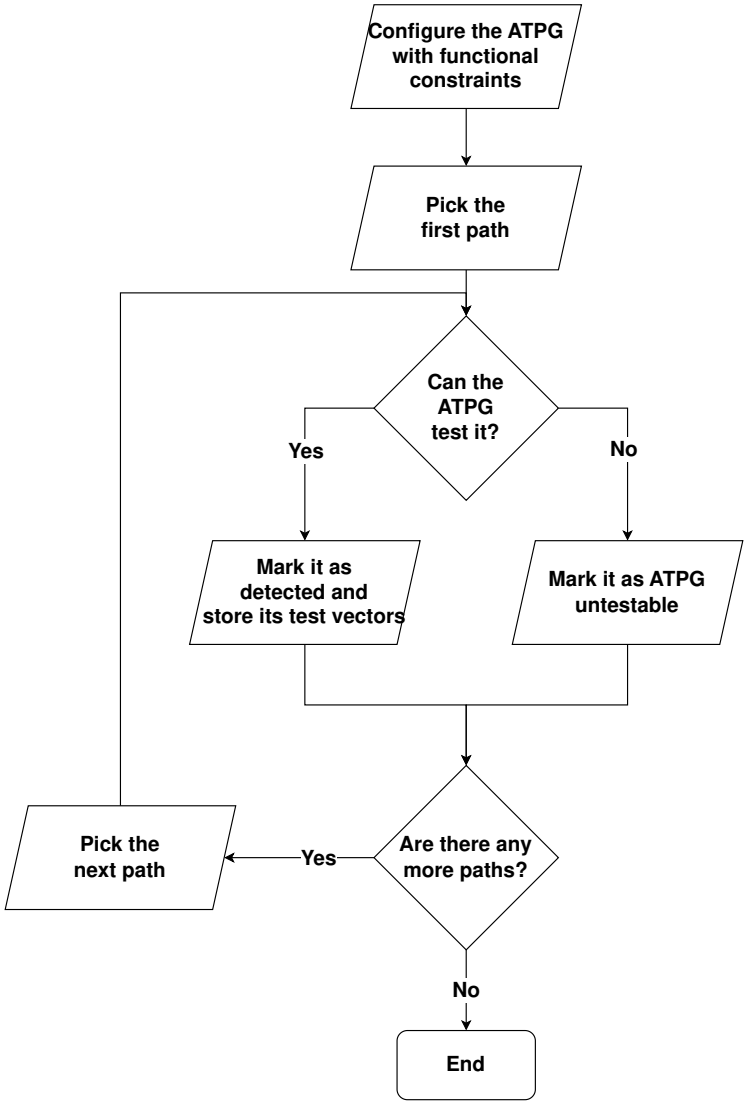


Fig. 8.2 ATPG-based test vectors generation

most convenient vector to convert, e.g., the one showing the most don't-care values. In case the ATPG does not support this feature and is not capable of providing a vector that can be translated into an instruction, other techniques can be used, e.g., refining functional constraints so that a new, easier to convert, pattern is generated. As test vectors are generated through the ATPG, their properties, e.g., whether they are robust or non-robust tests for path delay faults, depend on the ATPG capabilities. In most cases, the ATPG tries to generate robust tests and, if not possible, resorts to non-robust test vectors. The test engineer can, eventually, configure the ATPG so that it discards non-robust test vectors, at the expense of the final fault coverage.

It is however important to highlight that the ATPG is not aware of the fact that these patterns have to be functionally applicable: this could lead to the generation of test patterns that cannot be translated into instructions. To mitigate this problem, a set of functional constraints must be applied to the ATPG in order to promote the generation of test patterns that can be mapped into instructions belonging to the CPU's instruction set architecture (ISA). Such feature is described thoroughly in Section 8.1.2.

Faults managed by the ATPG following this approach may belong to one of the following categories:

- *Detected* group: the fault has been detected and the relative couple of test vectors have been produced,
- *ATPG Untestable* group: the ATPG could not generate test vectors capable of testing the fault under the specified constraints.

The latter category requires some observations. Based on what is presented in Section 7.3, faults produced by this flow are known to be testable by the ATPG when no constraint is applied. As a consequence, if a fault is marked as ATPG untestable it means that this fault is also functionally untestable, as the ATPG cannot produce test vectors when functional constraints are applied. In this way, the flow is able to remove a portion of FUFs from the fault list, obtaining a list of faults whose patterns can be converted into instructions. Nevertheless, it is still possible that there might be a non-negligible subset of functionally untestable faults among the detected ones. The reason for this lies in the architecture of the DUT, but also on how instructions are issued and executed by the processor core. To give an example, let us consider a fault that affects the arithmetic unit of a pipelined, in-order processor and the test

pair produced by the ATPG requires launching a division followed by another ALU operation. Given this premise, the targeted fault is a FUF: divisions require more than one clock cycle to complete, and their execution stalls the whole CPU, hence it will be impossible to execute an ALU operation at the clock cycle following the issuing of the division. It is worth reiterating, however, that functionally untestable faults cannot be detected by SBST means, due to the fact that they cannot be excited under functional scenarios.

8.1.2 Functional constraints identification

Functional constraints are a crucial component of the ATPG pattern extraction process, as they ensure that the produced test vectors can be effectively translated into instructions belonging to the DUT's instruction set architecture (ISA). For this reason, identifying and defining a set of functional constraints is a key task in our methodology, and is presented separately. The functional constraint identification algorithm that is used in this thesis is briefly summarized in Algorithm 4.

In order to describe this algorithm, it is first important to highlight that, in this PhD thesis, the set of functional constraints needed to generate valid test patterns is directly applied to the ATPG. Such constraints come in the form of values applied to the PIs and PPIs of the DUT: for this reason, prior to any further step, we perform an *off-path inputs analysis*, i.e., for each path we analyze the path's input cone logic, starting from off-path inputs and moving towards the *cone of influence inputs*. This, together with a list of sub-modules into which the DUT is divided, are the input of our functional constraint identification algorithm.

The presented algorithm adopts a semi-automatic approach to tackle the problem of generating functional constraints. For each sub-module S_i, all paths within the sub-module and the input cone logic for each path are identified first and grouped into a set D of paths P_i and its input cone logic C_i. Next, an automatic tool capable of performing logic simulations of a suite of carefully devised programs is employed, each program targeting a specific sub-module of the device under test. While simulating, the automatic tool records all input values. Once all simulations are completed, it identifies all the input pins that cannot be controlled when running test programs. Once this is cleared, the last step consists of annotating the set of input signals, together with their value, into a list of constraints to be fed to the ATPG. In

Algorithm 4: Functional Constraint Identification

```
Input  : A set S:=(S_i), where S_i is a sub-module of the device under test
Input  : A set D:=(P_i,C_i), where P_i is a path to be tested and C_i is the list of the i-th
         path's input cone logic
Output : A list of functional constraints to be applied to the ATPG
begin
    F := empty list of functional constraints
    /* Repeat for every sub-module                                          */
    foreach S_i in S do
        NC:= empty set of non-controllable input signals;
        select all paths P_i belonging to S_i;
        extract all C_i related to paths P_i;
        run logic simulation of ad-hoc programs;
        store non-controllable input signals into NC;
        /* Check whether non controllabe inputs belong to the C_i
           of a P_i within the considered S_i                               */
        foreach signal in NC do
            if signal in C_i then
                add signal with its tied value to F;
            end
        end
    end
    return F
end
```

this way, it is possible to make sure that the produced test patterns only involve those pins that can be driven through SBST means with replicable values. The proposed functional constraints identification method is conservative in the sense that, given its nature, there still may be a set of signals that can be constrained that are not identified. This can be further refined by means of other techniques, e.g., by using SAT solvers.

The time complexity of Algorithm 4 scales linearly with the number of sub-modules that are found within a CPU, as each sub-module requires one logic simulation to be launched, plus some processing time to extract information about signals that cannot be driven with any functional program, which is however negligible with respect to the logic simulation time. Moreover, since simulators allow dumping information on all signals in a single file, it is also possible to run a single simulation and then post-process such file to extract constraints. In this case, the time scales linearly with the product of signals to monitor and number of clock cycles required for the logic simulation to execute.

8.1.3 Patterns-to-instructions mapping

Once the test pairs have been generated, it is necessary to translate them into instructions. The very first step in performing such conversion consists of mapping the values stored in the test patterns into signal groups, e.g., the *opcode*, *ALU operands*, *registers*, etc. Afterwards, the test engineer must choose a set of instructions that is capable of replicating the generated test pair. This process is quite complex for two reasons. First, for each available instruction, it is necessary to understand which signals are controllable and how the instruction affects them; this could be non-trivial depending on the signal group. The second reason is due to the fact that values from the test vector must be applied concurrently on CPU pipeline stages. As a consequence, a single vector has to be mapped to several instructions, each one controlling signals in one pipeline stage. This has to be carefully selected such that they reproduce a test vector in a given clock cycle. In this methodology, this is done employing a semi-automated approach: a parser maps the test vectors into the aforementioned signal groups, while the test engineer defines a sequence of instructions and simulates them to make sure that they reproduce the required values.

A generic sequence of instruction is divided into three blocks:

1. Initialization instructions: mainly in the form of *load* instructions, they are used to initialize the DUT so that the test can be effectively applied.
2. Test instructions: once the DUT is prepared, these instructions generate and propagate the intended transition through the targeted path.
3. Store instructions: to propagate data affected by errors to POs, where mismatches due to faults are finally observed.

To better clarify, in the following an example of pattern conversion of a real test vector couple is reported, depicted in a simplified version in Fig. 8.3. To give some reference, the path targeted by the ATPG in the figure belongs to an adder embedded in the jump address sub-module. To start off, in the upper table we reported a portion of the two test vectors, namely *First Vector* and *Second Vector*, whose values have already been mapped into their respective signal groups. Among all groups, those that can easily be identified with modules that are found in the processor core are reported. The first 8 groups refer to portions of registers in the register file, while the latter 6 groups refer to fields of fetched instructions. The dash symbol, '-', has been

used to represent the "don't-care" value, while in all other cases hexadecimal values are reported. The different fields into which the instruction is divided are strictly dependent on the processor's Instruction Set Architecture: in this specific example, we identify the following fields:

- id_instruction[6:0]: contains the opcode of the instruction (bold black text),
- id_instruction[14:12]: contains a special function field (func3) or the lower portion of the immediate field depending on the type of instruction (bold pink text),
- id_instruction[19:15]: contains the register source 1 field or a portion of the immediate field depending on the type of instruction (bold green text),
- id_instruction[24:20]: contains the register source 2 field or a portion of the immediate field depending on the type of instruction (bold red text),
- id_instruction[31:25]: contains a special function field (func7) or the upper portion of the immediate field depending on the type of instruction (bold brown text).

Looking at the values reported for the first and second vectors, it is possible to notice that numerous don't-care are associated to signal groups. Let us start by considering signals mapped to registers, first. It is possible to see that for each register, one of the two vectors allows for a don't-care value while the other is fully specified. This means that those registers can be initialized to the required values in the aforementioned block of initialization instructions, without needing to further drive them in the test instructions block as they already store values that satisfy the test vectors. Instruction fields, on the other hand, are for the most part fully specified both in the first and second test vector, hence why we need the two test instructions to drive the appropriate fields accordingly. The opcode points to what couple instructions must be generated, namely a *load byte* instruction followed by a *jump and link register*. The remaining instruction fields identify immediate values and registers that must be employed with these two instructions, and allow to fully determine what couple of instructions must be placed in the test instructions block. Finally, it is necessary to introduce a store instruction to propagate the effects of the previously excited faults towards one of the POs, hence observing data affected by the targeted path delay fault. This is done so that the faulty value can be stored

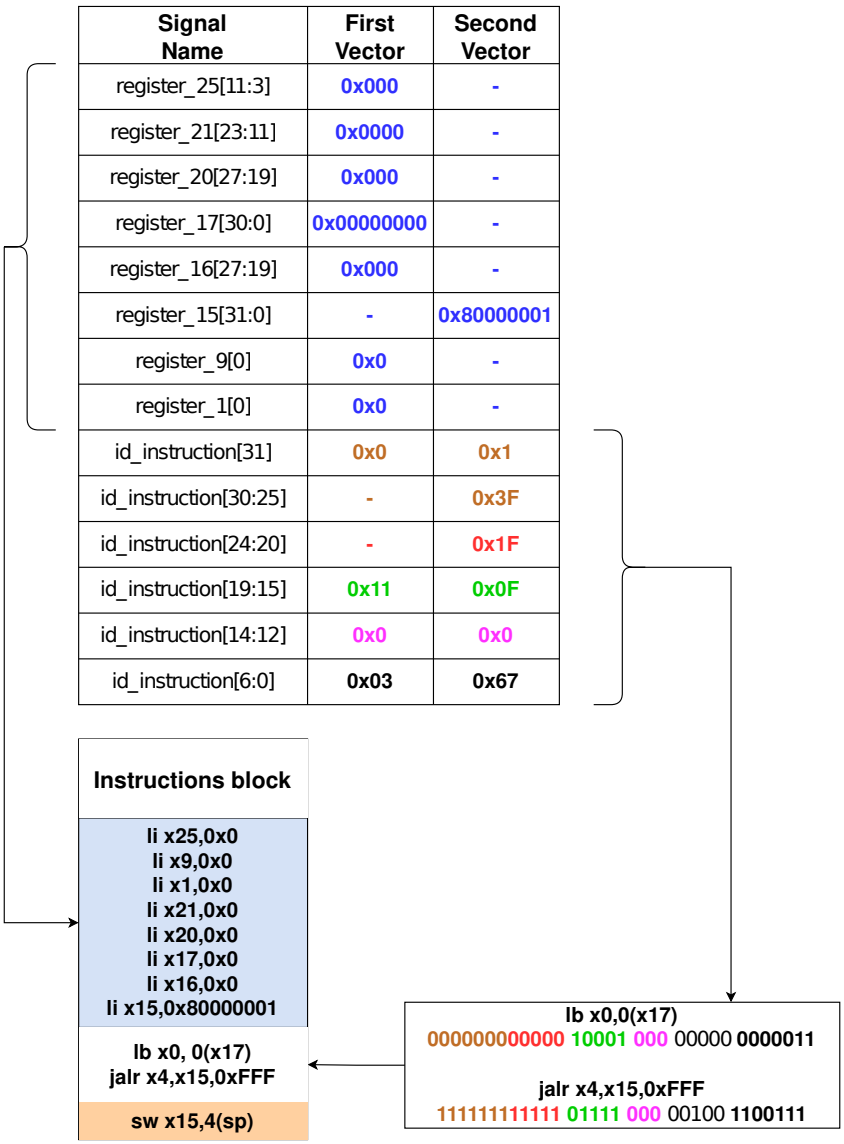


Fig. 8.3 Example of test program

into a non-volatile memory, possibly after being compacted into a signature, to be compared against the golden circuit's one. In this sense, observing a faulty value at the primary output equals to detecting its relative fault. In this example, instructions able to test a path delay fault affecting the jump mechanism of the CPU were generated quite easily thanks to the presence of several don't-care values. In general, however, choosing the right instructions is not trivial as they must be carefully picked to replicate all the signals in the test vectors without any mismatch, bearing into mind how they behave across all pipeline stages. This is why, in this PhD thesis, the process of converting patterns to instructions still requires some manual effort. It is possible, however, to make this process more automatic, e.g., by identifying all those signals that have a don't-care value in one of the two vectors so that instructions for them can easily be generated for the initialization block, as well as implementing mechanisms to store the instructions format defined in the Instruction Set Architecture so that the tool can come up with instructions without manual intervention.

Finally, the STL consists of every sequence of instructions generated for each path. One of the strengths of this approach consists in being modular: depending on the length of the test slot, i.e., the time slot in which the DUT can be tested, the test engineer can decide whether to run the test program as a whole, or split it into several submodules. The only rule to be observed is not to split a single instruction block that must always be run as a whole, since the ability of testing a path strictly depends on the execution of the instruction sequence without interruptions.

8.2 Experimental Results

8.2.1 Case Study

To validate the STL development methodology presented in the previous chapter, the choice fell on PULPino, a 32-bit RISC-V core developed by ETH Zurich and Università di Bologna. For these experiments, PULPino was configured to use the RI5CY core, an in-order, single-issue core with 4 pipeline stages, capable of supporting the RV32ICM instruction set which includes integer, compressed, and multiplication instructions, as thoroughly described in Section 3.2.1.

Table 8.1 Case study general info

Parameter	*Value*
Number of gates	46,850
Total Area (eq. gates)	51,001.65
Clock Period (ns)	5.00
Number of seq. elements	2,325

Table 8.2 Paths report

Paths class	*#Paths extracted*	*Slack range (ns)*
Long paths	5,009	[1.3 : 2.5]
Short paths	5,137	[4.8 : 4.97]

This processor core was synthesized using the open hardware 45nm *Nangate OpenCell Library* provided by Silvaco[61]. Data regarding the synthesized core can be found in Table 8.1.

Row *Number of seq. elements* include all flip-flops and latches found in the design. Table 8.2 reports additional information on the extracted paths. These paths are the result of the extraction process described in Section 7.3; as a consequence, only paths that can be tested through ATPG are reported here and targeted in our fault simulation experiments. Two types of paths were extracted: long paths, whose slack is small, and short paths, with a larger slack. Paths from both of these categories have been targeted in our fault simulation experiments.

Longer paths have been extracted by looking at all paths in the slack range $[0 : 2.5]$ns: the reason for not having any path in the $[0 : 1.3]$ns slack range lies in the fact that the ATPG did not find any testable path among those. From a topological point of view, long paths are divided into three groups:

1. 45 paths in an adder of the divider module of the ALU,
2. 540 paths in an adder of the load/store unit,
3. $4,424$ paths in an adder inside the jump address module of the decode stage.

Shorter paths, on the other hand, have been limited to those belonging to the $[4.8 : 5.0]$ns slack range. As there are plenty of short paths in a circuit, the extracted

ones do not belong to single modules of the CPU, rather they are scattered throughout the whole processor core.

As for the test programs, since we are working with two different classes of paths – long paths and short paths – we decided to develop two separate STLs.

Starting with the long paths STL, this set of procedures was developed completely from scratch, and it can be thought of as three independent procedures that aim at testing faults in the three aforementioned submodules. For the vast majority of faults, this methodology was capable of producing appropriate testing instructions that excite faults and propagate their effects towards primary outputs. For a very small percentage of all faults (45 out of $10,018$, to be found in the divider unit), however, the test vectors could not be replicated due to physical constraints: such couples required a division promptly followed by another ALU-related instruction, which cannot be executed at the immediately following clock cycle as divisions take more than one clock cycle to complete, making those 45 faults functionally untestable.

As for the short paths STL, experimental data from [17] shows that programs developed for SAFs and TDFs are somehow capable, although not with high coverages, to test short paths. For this reason, we decided to start from the already available SAF and TDF-oriented programs. In this way, we could discard the faults that are already covered by them, and focus on the remaining ones through the proposed methodology. Thanks to this approach, we were able to easily identify a large number of FUFs, as well as to further increase the number of detected faults. These results validate the proposed approach, as it succeeded in producing effective STLs even though long and short paths are quite different in terms of topological distribution and functionalities.

8.2.2 Achieved Results

The experimental results reported in this thesis have been obtained through the adoption of several different commercial tools coordinated through the usage of Bash and Python scripts as described in Chapter 7. Regarding the preliminary steps *Design Vision*, a tool provided by *Synopsys*, was used for the synthesis step, together with *Questasim* by *Mentor Graphics* for logic simulations and *PrimeTime* for path extraction in conjunction with *TetraMAX*, both by *Synopsys*, for path refinement. The combinational-level fault simulation has been performed using *TetraMAX*, while

the sequential-level fault simulation makes use of *Z01X*, a fault simulator specifically designed for functional safety by *Synopsys*. It is reported that TetraMAX ATPG preferably generates robust test vectors and resorts to non-robust vectors if and only if robust vectors cannot be generated for the fault. In this thesis the tool has been configured so that non-robust test vectors are discarded, thus producing robust test vectors, only.

The experimental results we gathered are referred to the processor core and synthesis library described in Section 8.2.1, and are reported separately for long and short paths. The STLs we generated were capable of covering 100% of testable long paths and 87.31% of testable short paths of the chosen DUT. In both cases, the STL generation required about 8 days of ATPG time, using 5 cores of an Intel Xeon CPU E5-2680 v3 server.

Starting with the long paths STL, the whole test program requires 26kB of memory space, and a total amount of $45,605$ clock cycles to execute. The amount of clock cycles has been computed by executing the whole STL in a single simulation; depending on the situation, the test engineer can then split the STL into several shorter testing sub-routines to adapt them to the available idle time slots, matching the strict time constraints of in-field test. The generation and simulation of the whole test program required no more than four days worth of CPU time to complete. Since no methodologies to develop STLs for all PDFs in a processor core are currently available in literature, to assess the validity of the proposed methodology we decided to compute the fault coverage figures for STLs intended for other fault models – namely, SAFs and TDFs – used as a reference case. Stuck-at fault oriented test vectors are not compatible with delay faults when generating test patterns for scan chains through ATPG. Nevertheless, throughout this thesis the SBST paradigm is always adopted, thus ensuring that test vectors are applied through functional at-speed clock cycles throughout the whole STL duration. For this reason, it is appropriate to compare our results with those achieved by SAF-oriented test programs. In particular, we run 5 different SAF programs and 1 TDF program, able to achieve 90.91% (cumulatively) and 74.25% wrt their target faults, respectively. Test programs for stuck-at faults and transition delay faults target the same fault list (one for stuck-at and the other for transition delay faults). In both cases, all faults from the processor core are targeted. In order to ensure a diverse test suite, even though the fault list is the same for any given fault model, the test programs were developed using different approaches by multiple test engineers, thus consisting in a different set of test vectors

Table 8.3 Long path fault coverage

Program	*#Clock Cycles*	*Combinational Fault Coverage*	*Final Fault Coverage*
SAF Program 1	64,502	0.33%	0.32%
SAF Program 2	36,394	0.27%	0.27%
SAF Program 3	42,970	0.27%	0.22%
SAF Program 4	118,098	0.40%	0.32%
SAF Program 5	17,269	0.09%	0.08%
TDF Program	23,451	0.27%	0.23%
PDF Program	45,605	99.50%	99.50%

Table 8.4 Long path fault coverage per module

Module	*#Faults*	*Combinational Fault Coverage*	*Final Fault Coverage*	*Testable Fault Coverage*
ALU_Div Adder	90	50.00%	50.00%	100.00%
LoadStore Adder	1,080	100.00%	100.00%	100.00%
Jump_Addr Adder	8,848	100.00%	100.00%	100.00%
Total	10,018	99.50%	99.50%	100.00%

for every program. We decided to pick five test programs for stuck-at faults as this fault model is the most widely adopted in industry and the most mature in terms of SBST solutions. All these data are reported in Table 8.3. The *Combinational Fault Coverage* column shows the path delay FC achieved on the combinational portion of the DUT while *Final Fault Coverage* reports the fault coverage for the whole CPU. The last row reports information about the test program generated by the proposed method.

The table clearly shows that the method dramatically improves the FC figures of existing test programs. The reason for this significant difference is that in our test program's instructions and values are carefully chosen to test specific paths; other test programs, instead, can be approximated to a random – and thus ineffective – approach due to the significant differences between the three fault models. We also analyzed the FC figures on the three submodules of the CPU into which the longest paths are found, as reported in Table 8.4.

Table 8.5 Short path fault coverage

Program	*#Clock Cycles*	*Combinational Fault Coverage*	*Final Fault Coverage*
SAF Program 1	64,502	71.98%	50.10%
SAF Program 2	36,394	73.76%	58.00%
SAF Program 3	42,970	74.78%	55.70%
SAF Program 4	118,098	76.51%	67.60%
SAF Program 5	17,269	73.16%	51.25%
TDF Program	23,451	73.60%	56.70%
PDF Program	279,253	83.77%	77.15%

We achieved a 100% FC in both the load/store unit and the jump address adder, while we were able to cover 45 out of the 90 faults of the divider. The remaining 45 faults have been identified as FUFs, as previously explained. Consequently, if we exclude such FUFs from the final fault coverage, we obtain a testable fault coverage for all three modules – and, thus, for all long paths – equal to 100%. It is important to notice that the long paths we test are not the longest path from a topological standpoint in this DUT. Looking at Table 8.2, it is possible to notice that the set of long paths we are targeting falls within the [1.3-2.5] ns range slack, and this is because all paths with smaller slack have been deemed structurally untestable by the ATPG. Our approach tests all path delay faults stemming from those paths. In principle, the same approach could be used with short paths (those whose range is almost equal to the clock period), but as discussed in Section 8.2.1, test programs written for other fault models constitute a good starting point for testing these faults, hence why it would be more efficient and timesaving to harden those test programs to make them more suitable to test path delay faults.

Table 8.5, on the other hand, reports data for the STL developed for short paths (last row). This program requires 18kB of memory space and a total amount of $279,253$ clock cycles to execute completely. Since short paths are distributed among the whole CPU, we decided to associate them to the module, or set of PIs, from which the path's startpoint stems, also reporting the relative FC. Table 8.6 reports such data.

For each module, we show the total amount of faults, the combinational and final fault coverages, as well as the testable fault coverage achieved by removing FUFs

Table 8.6 Short path faults coverage per module

Startpoint	*#Faults*	*Combinational Fault Coverage*	*Final Fault Coverage*	*Testable Fault Coverage*
Debug_PIs	666	0.00%	0.00%	100.00%
Other_PIs	299	92.31%	81.60%	81.60%
CS_Registers	290	69.56%	34.48%	41.38%
Debug_Module	287	0.00%	0.00%	100.00%
ID_Stage	7,714	97.54%	94.28%	94.82%
Controller	80	0.00%	0.00%	50.00%
Pipeline_Regs	744	94.35%	69.22%	69.49%
Registers	6,890	99.01%	98.08%	98.08%
EX_Stage	652	57.05%	22.38%	22.65%
ALU	182	11.53%	11.53%	17.37%
Multiplier	48	33.33%	33.33%	43.33%
Sparse_logic	422	79.38%	25.82%	25.82%
LoadStore_Unit	366	81.15%	81.15%	84.70%
Total	10,274	83.77%	77.15%	87.31%

from the final FC. Most paths belong to the *ID_Stage*, which is well covered by our STL. Two blocks of faults are marked as completely untestable, namely those that stem from debug-related circuitry, as they are not controllable by functional programs. Faults that originate from non-debug PIs are easily controllable – hence a 92.31% combinational coverage – but not always easily propagated to primary outputs, with a final 81.60% coverage. Moving to faults related to *CS_Registers*, 20 of them are also affected by clock gating circuitry and, hence, untestable. As for faults belonging to the *EX_Stage*, in most cases they are found in control-related logic, making it hard to either control or observe them. Lastly, looking at the *LoadStore_Unit*, some faults could not be properly excited due to the presence of off-path inputs from other modules and PIs; observability however is quite easy to achieve, as every fault detected at the combinational level is also observed at the POs.

Comparing the path delay fault coverages achieved with our test programs against those obtained by test programs for stuck-at and transition delay fault, it is possible to see that the latter are generally much worse than the former, especially for long paths. This is to be expected, given the significant difference between the fault models. This could lead to the possibility of adopting this approach for the generation of functional test programs for other fault models, too. If we exclude the tasks that

are strictly related to path delay faults, e.g., the path generation process that is not required for stuck-at and transition delay faults, the constrained, functional ATPG test pattern generation process followed by test vectors conversion into instructions can be used for other fault models. However, given the vast number of methodologies for developing stuck-at and transition delay fault oriented Self-Test Libraries, rather than developing test programs from scratch it might be interesting to apply such technique to harden STLs [76, 77, 60], so that faults that were not detected by the test programs can be covered, too.

The fault list we produced and used throughout all fault simulation steps is a generic one, not depending on the application the DUT is executing. This detail is relevant as, based on the application, some working modes, and hence circuitry, of the DUT might never be used. Consequently, faults located in the unused circuitry can be effectively marked as Safe Faults (i.e., FUFs), as per ISO26262 standard. The results achieved in this work can thus be enhanced by a careful analysis of those working modes that we can neglect, e.g., identifying a set of constraints to be fed to a formal verification tool that identifies the aforementioned FUFs, as described in [78].

8.3 Chapter Summary

This chapter presented a novel methodology on STL development for processor cores targeting path delay faults. This methodology makes use of an ATPG engine to generate a set of test vectors which is then converted into instructions to be put into the final test programs by means of a semi-automated tool. Generating test vectors through ATPG allows to (i) produce effective and reliable vectors in a timely manner and (ii), remove untestable faults from the fault list, thus focusing the effort on those faults for which it is possible to generate test programs. Moreover, in most cases ATPGs are capable of generating vectors that are not fully specified, i.e., that contain some don't-care values, thus easing the vector to instruction conversion, as more degrees of freedom are left for the tool to find suitable instructions. If possible, the ATPG can be configured to discard non-robust test vectors, thus leaving us with a final STL that is comprised of robust test vectors, only. The main drawback of using an ATPG consists in the fact that, by default, it produces non-functional vectors, i.e., vectors that are assumed to be applied through the usage of scan chains and cannot

be replicated with instructions from the processor's ISA. For this reason, the chapter introduces an algorithm to derive functional constraints to be applied to the ATPG, so that test vectors can be effectively mapped into instructions. Once patterns are obtained, the final step consists in mapping them into instructions. This is done by first adopting an automatic tool that is capable of identifying and categorizing signals into groups, e.g., opcode, ALU operands, registers, so that instructions that satisfy the groups' values can be chosen. Then, for each fault, a test engineer must define a sequence of instructions capable of testing the aforementioned fault. Such sequence is divided into three main blocks, the first being responsible for the initialization phase where the DUT is brought to a state where the test vector pair can be effectively applied, followed by the actual test instructions that apply the transition required to excite and propagate towards the endpoint the path delay fault, with a final store instruction that makes the fault effect observable at the primary outputs of the DUT. This approach has been tested on PULPino, a 32-bit pipelined RISC-V core, showing that it is capable of testing 100% of testable faults affecting long paths, and it has been used to improve the coverage on path delay faults affecting short paths too, where a set of previously devised test programs developed for other fault models, namely stuck-at and transition delay faults constituted the starting point, reaching a final fault coverage of 87.31% of path delay faults affecting short paths. This proves that the proposed methodology can produce effective STLs for modern processor cores targeting path delay faults.

As a last note, I would like to focus on limitations and areas for improvement of the proposed STL development methodology. Most of the proposed approach is carried out in an automatic fashion, namely the path generation, ATPG patterns extraction and part of the instruction conversion processes. With the due timing overhead related to the complexity of the device under test, I believe that it can be extended to larger designs. Currently, a task to be improved to scale this method for larger cores is the functional constraint generation process, as it requires some manual effort from the test engineer and a batch of logic simulations to identify what signals cannot be controlled under functional constraints. Future works will tackle this issue, so that it can be automated and improved to support more complex designs.

Part III

Testing an aged integrated circuit

Chapter 9

Background

Modern digital integrated circuits are designed and manufactured leveraging on advanced semiconductor technology and nodes. An empiric law introduced by Gordon Moore describes how, at a first approximation, the number of transistors found on microchips doubles every two years circa, showing the trend described in Fig. 9.1.

Increasing the number of transistors on an integrated circuits not only greatly affects the physical size and geometry of each transistor, but it also has repercussions on the power supply at which the circuit operates and parasitic capacitance and resistances, thus impacting the power consumption and operating frequency of the integrated circuit. While this enabled the production of very large scale integrated circuits that are capable of operating at high frequencies with limited power consumption, it also impacted the reliability of such devices. Physical phenomenon that were negligible on larger technology nodes became more and more relevant with newer technologies, and parameters shift in time influence the behavior of the device as it ages, possibly leading to its failure as the circuit is stressed over time. Given that many fields require safety-critical devices to correctly behave on a large period of time, e.g., the automotive field where replacing a faulty device many times over the operative lifetime of the vehicle is not feasible, it is crucial to tackle the issue of ensuring the safety and reliability of integrated circuits as they age.

Indeed, there are several works in literature that focus on the study of aging in circuits and how it affects their timing behavior [79–84]. Aging and performance degradation of an integrated circuit is the consequence of several physical phenomena

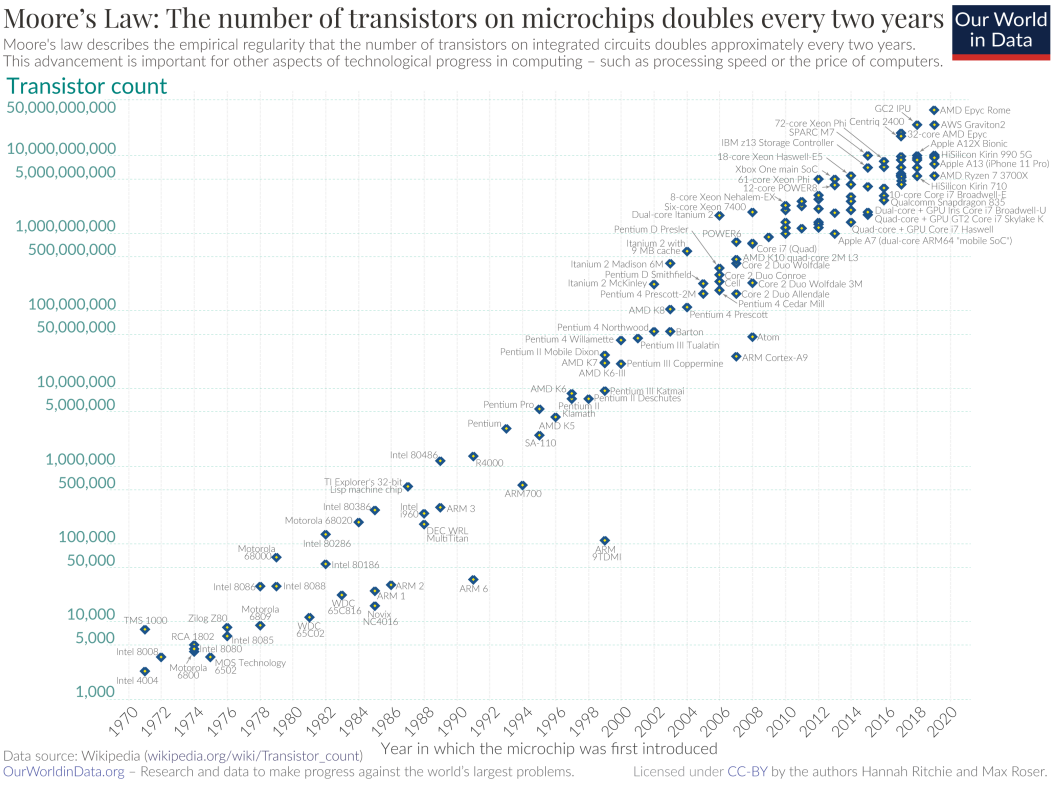


Fig. 9.1 Number of transistors on microchips over time [3]

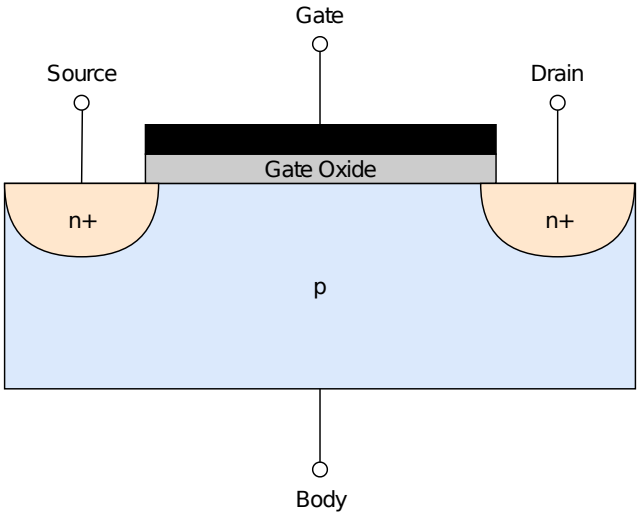


Fig. 9.2 Silicon model of a MOSFET

that occur concurrently, impacting the device in different ways. Although the scope of this thesis does not include the study of the physics of semiconductors, to provide an overview on this topic it is still worth introducing and discussing the effect of Bias Temperature Instability (BTI), Hot Carrier Injection (HCI), and Time-Dependent Dielectric Breakdown (TDDB) as the main causes for degradation over time in modern integrated circuits. BTI is a destructive phenomenon that can be further divided into Negative Bias Temperature Instability (NBTI) [85] and Positive Bias Temperature Instability (PBTI). The former affects PMOS transistors mostly, and it induces a more significant degradation with respect to the latter which affects NMOS transistors mostly. When a transistor is affected by BTI, many of its physical properties are impacted, e.g., its threshold voltage V_{th}, saturation current $I_{d,sat}$ and transconductance g_m. HCI occurs whenever an electron or a hole gains sufficient kinetic energy to overcome the potential barrier, thus penetrating the dielectric oxide layer that is found below the gate plane in a MOSFET [86–88], as showed in Fig. 9.2. As transistor size scales, the voltage at which they operate does not scale accordingly which causes large electrical fields inside the device. The larger such fields, the higher the probability a hot carrier, may it be electron or hole, is injected in dielectric films, thus degrading the device over prolonged periods and impacting its physical properties.

Finally, TDDB affects the dielectric film of transistors, and it is described as a change in properties of the dielectric due to the presence of electric fields, shifting

from a material with insulating properties to one with more conductive features [89]. This affects leakage currents when the device is supposed to be off and threshold voltage too, and it is a major reliability issue in MOSFETS [90].

In order to tackle these issues, researchers have worked on producing a mathematical model that could predict the shift in propagation delay of any given cell in a circuit. Such model can be obtained by integrating formulas described in literature and actual data gathered on logic gates implemented on silicon wafers. Let us start by considering a model for delay estimation on MOSFET devices first introduced and described in [79], also known as the alpha-power law. Such model states that the propagation delay is proportional to the output capacitance the cell is subjected to times the voltage supply at which it operates over its drain current as follows:

$$d_{cell} \propto C_{out} \frac{V}{I_d} \tag{9.1}$$

Starting from Eq. (9.1), it is possible to derive an equation that describes the propagation delay of an inverter gate as follows:

$$d_{inv}(V,T) \propto C_{out} \frac{V}{\mu(T)(V - V_{th}(T))^{\alpha}} \tag{9.2}$$

where $\mu(T)$ is the carrier mobility that depends on the temperature at which the gate operates, V the supply voltage, $V_{th}(T)$ the threshold voltage depending on the temperature and α a positive constant (carrier velocity saturation). Such model effectively describes how the inverter delay changes with the temperature and supply voltage, but it does not account for the time effects. For this reason, Eq. (9.2) has been further developed in [83] so that time is factored in as well. The final formula allows to define the delay of an inverter cell as:

$$d_{inv}(V,T,t) = p_{\beta} + p_{\mu^{-1}}(T) \frac{V}{(V - (p_{V_{th}}(T) + \Delta p_{V_{th}}(V,T,t)))^{p_{\alpha}}} \tag{9.3}$$

where p_{β} and p_{α} are constants, while $p_{\mu^{-1}}(T)$ is related to the transistors mobility and $p_{V_{th}}(T)$, $\Delta p_{V_{th}}(V,T,t)$ are factors related to the threshold voltage, the latter describing a shift in threshold voltage depending, among the other parameters, to time. The transistors mobility and threshold voltage factors can be described by

means of equations with an exponential dependence to temperature:

$$p_{\mu^{-1}}(T) = C_\mu + k_\mu T^{n_\mu} \tag{9.4}$$

$$p_{V_{th}}(T) = C_{V_{th}} - k_{V_{th}} T^{n_{V_{th}}} \tag{9.5}$$

while the $\Delta p_{V_{th}}(V,T,t)$ contribution can be expanded as:

$$\Delta p_{V_{th}}(V,T,t) = (c_1 \cdot t^{N_1 + a_1 log(V)} + c_2 \cdot t^{N_2 + a_2 log(V)}) \cdot V^\gamma \cdot e^{-E_a/kT} \tag{9.6}$$

With the exception of γ (voltage acceleration factor), E_a (temperature activation energy) and k (Boltzmann's constant), all the other parameters defined in Eq. (9.4), Eq. (9.5) and Eq. (9.6), namely, C_μ, k_μ, n_μ, $C_{V_{th}}$, $k_{V_{th}}$, nV_{th}, c_1, N_1, a_1, c_2, N_2 and a_2 are fit parameters defined for a given technology, e.g., FDSOI 28nm, that are obtained with a high degree of confidence.

Eq. (9.3), although effective, has some important limitations. First, it only describes the delay of an inverter gate, while any integrated circuit worth discussing has several different gates within its design. Second, it does not factor in the effect of switching activity, i.e., how much the cell toggles throughout the operative lifetime of the device under test. If certain portions of a circuit have a higher switching activity, that is, they toggle more as a consequence of a more intensive usage, it follows that those portions will age faster than other, less-used ones. Finally, even though this is not an actual limitation per se, it is worth reiterating the fact that the physical parameters that are used in the aforementioned equations strictly depend on the technology on top of which a library of cell gates is defined. This implies that each library produced by different manufacturers, or even two libraries defined with different technologies belonging to the same manufacturer, will have its own set of parameters. While the last observation is tied to how these set of equations are defined and cannot be dealt with without changing the underlying model, extending Eq. (9.3) so that other gates are defined too, together with weighting in the effect of the switching activity, is possible and has been investigated in [84].

Let us start by tackling the task of extending the delay equation to a generic cell. In order to generalize Eq. (9.3) from the delay of an inverter to that of a generic cell, it is necessary to introduce the concept of logical effort. As extensively described in [91], the logical effort of a logic gate is defined as the ratio of its input capacitance

to that of an inverter that delivers equal output current. If a standard cell library has been used to synthesize a design, calculating the logical effort for a given gate is quite straightforward. For each gate, several parameters are defined. Among them three are needed to extract the logical effort, namely the *input capacitance* C_{in}, usually expressed in pF, the *intrinsic delay* t_i, usually expressed in ns, and the *propagation delay with respect to the load capacitance* K_{load} which is usually expressed in ns/pF. With the exception of the input capacitance, these parameters are defined for each input of the logic gate and for both rising and falling transitions from the input to the output. For instance, in a 2-input AND gate where the two inputs are `A` and `B` and the output is `Y`, the standard cell library documentation provides an intrinsic delay associated to the rising transition from `A` to `Y` and another associated to the falling transition from `A` to `Y`, and similarly for `B` and `Y`.

The average propagation delay of a logic gate from an input `X` to the output `Y` driving a fanout of h, calculated by averaging the propagation delay for the rising and falling transitions, can be defined as:

$$t_{pd(X,Y)} = \frac{t_{i,rise} + t_{i,fall}}{2} + C_{in} \cdot \frac{K_{load,rise} + K_{load,fall}}{2} (\text{h gates}) \tag{9.7}$$

The second term of this equation, namely $C_{in} \cdot \frac{K_{load,rise}+K_{load,fall}}{2}$ is also referred to as τ. Then, the logical effort for a generic logic gate associated to a pair of input-output pins (`X`, `Y`) is defined as the ratio of the gate's τ over the inverter's:

$$LE_{gate} = \frac{\tau_{gate}}{\tau_{inv}} \tag{9.8}$$

To clarify the concept, Table 9.1 reports the aforementioned physical parameters for an INV and AND gate. In both cases, they have a drive strength of 1 and, for the sake of simplicity, only the A input is considered in the AND gate.

The average propagation delay of both gates is:

$$t_{pdINV(A,Y)} = \left(\frac{0.0253 + 0.0146}{2} + 0.0036 \cdot \frac{4.5257 + 2.3675}{2} (\text{h gates}) \right) \text{ns}$$
$$t_{pdAND(A,Y)} = \left(\frac{0.0313 + 0.0195}{2} + 0.0042 \cdot \frac{4.5288 + 2.8429}{2} (\text{h gates}) \right) \text{ns}$$

Table 9.1 Physical parameters of an INV and AND gate

	INV_X1	AND2_X1
t_i (A $\rightarrow$ Y $\uparrow$) [ns]	0.0253	0.0313
t_i (A $\rightarrow$ Y $\downarrow$) [ns]	0.0146	0.0195
C_{in} [pF]	0.0036	0.0042
K_{load} (A $\rightarrow$ Y $\uparrow$) [ns/pF]	4.5257	4.5288
K_{load} (A $\rightarrow$ Y $\downarrow$) [ns/pF]	2.3675	2.8429

which is equal to:

$$t_{pdINV(A,Y)} = (0.0200 + 0.0124\text{h})\text{ns}$$
$$t_{pdAND(A,Y)} = (0.0254 + 0.0155\text{h})\text{ns}$$

We can finally obtain the logical effort for the AND2_X1 gate related to the A input, which is equal to:

$$LE_{AND} = \frac{\tau_{AND}}{\tau_{INV}} = \frac{0.0155}{0.0124} = 1.25$$

Now that the logical effort has been introduced and defined, extending the delay formula to a generic gate through the following identity:

$$d_{gate}(V,T,t) = d_{inv}(V,T,t) \cdot LE_{gate} \tag{9.9}$$

where $d_{inv}(V,T,t)$ is Eq. (9.3).

In order to include the switching activity contribution, the authors in [92] introduce a formula that is described as follows:

$$d(SA) = d(0.5) \cdot \frac{tanh(x^{\alpha})}{tanh(1)} \tag{9.10}$$

where SA is the switching activity, d(0.5) is the 50% switching activity delay value, x is $SA/(1-SA)$ and α is a cell dependent fit parameter. This formula can be integrated in Eq. (9.9) to get the final equation:

$$d_{gate}(V,T,t,SA) = d_{inv}(V,T,t) \cdot d(SA) \cdot LE_{gate} \tag{9.11}$$

Through Eq. (9.11) it is finally possible to estimate how the delay of a gate degrades in time with respect to voltage, temperature, time and switching activity, closely emulating to what happens in an actual circuit implemented on silicon. Although effective, this approach has a main drawback as it needs large quantities of data that can only be gathered by means of experiments on devices implemented on silicon. Taking Eq. (9.10) as an example, the cell fit parameter α varies with the duty cycle and it cannot be calculated in an exact way by means of mathematical formulas. As a consequence, the available options are to either launch a massive amount of experiments on silicon devices and record the value of all physical parameters accordingly, or employ some mathematical model to interpolate the delay whenever real data is not available. The work in [84] proposes a multiple linear regression algorithm through which it is possible to create a continuous function [93], i.e., the delay of a gate cell, in the form of:

$$y = \alpha + \beta_1 \cdot x_1 + \beta_2 \cdot x_2 + ... + \beta_n \cdot x_n \tag{9.12}$$

where y is the delay of a given gate cell, α is the y-intercept value, (x_1, x_2, ..., x_n) is the set of gate features (voltage, time, switching activity, temperature), and (β_1, β_2, ..., β_n) the coefficients of gate parameters. The idea behind this approach is that, once the parameters are tuned so that the error sum of squared errors (SSE) between observed and predicted results is minimized [94], such model can be used to generate reliable delay values for aged cells, thus aging the whole circuit.

In order to validate the proposed methodology, the aging framework is applied to two different circuits, namely a Finite Impulse Response (FIR) filter and an Advanced Encryption Standard (AES) crypto-processor that performs a set of encryption and decryption operations. For the FIR filter, the author provides data on how the most critical path ages in time over the course of 10 years, together with information on the delay degradation over 6 months of a set of 21 nearly critical paths. Data for the AES device includes information on how the propagation delay of 150 critical paths evolves after 6 months and 1 year of use. Results achieved in these two devices under test are quite different: while the delay degradation on paths belonging to the FIR filter amounts, on average, to a 1% increase with respect to the original value, the AES ages much faster, with an increase equal to, on average, an additional 50% of the original propagation delay on all critical paths after one year of use. Both circuits have been synthesized on a proprietary FDSOI 28nm library provided

by STMicroelectronics, that also provided all the physical parameters required to generate the aging model. Such results show that this aging framework can be effectively used to age integrated circuits, providing a means to extract aged critical paths that takes into account the fact that portions of the circuit toggle, and hence age, more than others.

Chapter 10

Main Contributions

The aging framework presented in Chapter 9 has proven effective in evaluating how critical paths' propagation delay evolve in time, thus providing a mean to easily age an integrated circuit. Nevertheless, it is possible to expand on some areas that are still not covered by the work presented in [84], thus strengthening this tool so that it can become a key aspect in delivering SBST solutions that are capable of ensuring the safety and reliability of integrated circuits over long periods of time. As a matter of facts, thanks to the proposed aging framework, it is possible to analyze how the path delay fault coverage changes as critical paths evolve in time, thus proving the importance of taking aging into account when writing test procedures.

The work carried out on the field of aging in this PhD thesis can be summarized into three main points:

1. The generation of an automatic tool that is capable of aging an integrated circuit with a limited amount of input data,
2. The definition of a flow that allows to generate a list of critical paths given an aged circuit and compare the list of critical paths of the aged circuit with respect to those found in the fresh circuit,
3. The identification of strategies that ensure that high path delay fault coverages are achieved throughout the operative lifetime of the device under test.

The aging tool proposed in this PhD thesis has been developed so that it can be used in conjunction with any commercial synthesis and static timing analysis tool, thus making it easily reusable.

Chapter 11 presents the automatic aging tool that allows to obtain a list of aged critical paths, describing in details each step of the aging process. The tool is validated by gathering data on a RISC-V processor core, showing how critical paths, and hence path delay fault coverage, change over time. This provides test engineers a metric on how STLs for path delay faults should be developed so that they can be effectively used over long time periods.

Chapter 11

Automatic Aging Tool

The automatic aging tool developed in this PhD thesis aims at providing a unified and automatic device that is capable of generating aged delays for each cell in a circuit starting from data generated by commercial tools currently used in research and industry. This tool has been developed in collaboration with the Grenoble INP university, where the studies [84, 95, 83] were conducted. At its core, the proposed aging tool implements the set of equations and linear regression model presented in [84] and provides a wrapper that acts as an interface capable of parsing input information and generating output information seamlessly. Fig. 11.1 summarizes the structure of the proposed automatic aging tool, and shows some necessary preliminary steps that are required for the tool to function correctly.

The very first step that is required for this tool to work is performing a synthesis of the device under test with a technology library for which physical parameters are known or available. This is an important aspect, as the whole mathematical model on top of which this approach is built requires that these parameters are known for calculating the propagation delay for each cell. No constraint is placed on the synthesis process, thus leaving the test engineer full freedom in customizing such step as required. The synthesis step is required in order to obtain information on the propagation delay of each cell of design at time 0, also referred to as fresh delay information, which will constitute the starting point for calculating the aged delay. Usually, such information is automatically generated at the end of the synthesis process in the form of a Standard Delay Format (SDF) file, where information on the cell name, i.e., the name of the cell found in the syntesized circuit, the cell type, i.e.,

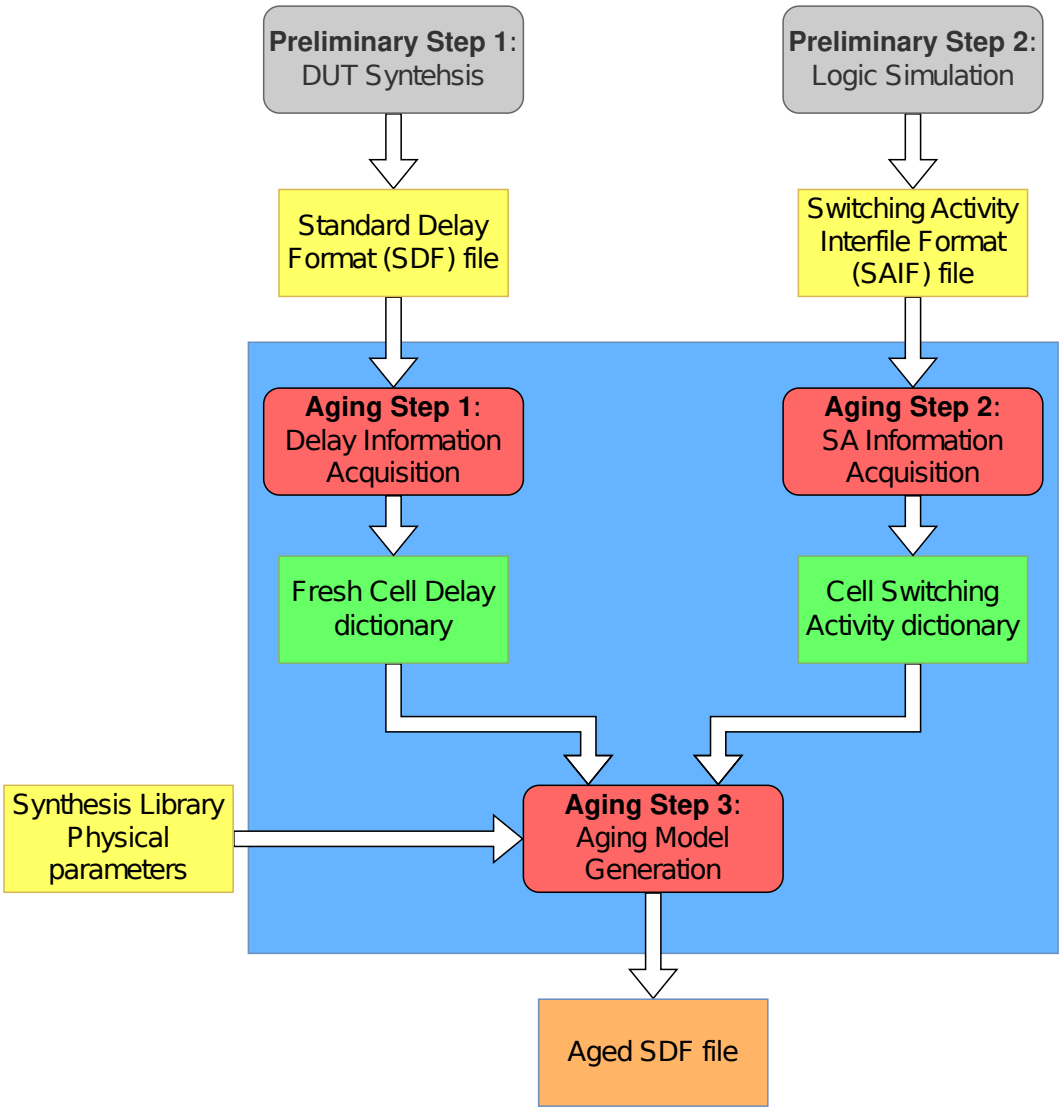


Fig. 11.1 Automatic Aging Tool flow diagram

```
(CELL
  (CELLTYPE "XOR2_X1")
  (INSTANCE cs_registers_i_add_775/U17)
  (DELAY
    (ABSOLUTE
    (COND (B == 1'b0) (IOPATH A Z (0.025:0.025:0.025) (0.026:0.026:0.026)))
    (COND (B == 1'b1) (IOPATH A Z (0.013:0.013:0.013) (0.008:0.008:0.008)))
    (COND (A == 1'b0) (IOPATH B Z (0.030:0.030:0.030) (0.031:0.031:0.031)))
    (COND (A == 1'b1) (IOPATH B Z (0.017:0.018:0.018) (0.012:0.012:0.012)))
    )
  )
)
(CELL
  (CELLTYPE "INV_X1")
  (INSTANCE cs_registers_i_add_775/U16)
  (DELAY
    (ABSOLUTE
    (IOPATH A ZN (0.009:0.009:0.009) (0.006:0.006:0.006))
    )
  )
)
```

Fig. 11.2 An example of SDF syntax for two cells

whether it is an AND, INV, OR gate, and slow, typical and fast propagation delays from each input to each output are reported, also reporting the logic values of other inputs. A snippet of an SDF file is reported in Fig. 11.2.

The second preliminary step is the execution of a logic simulation where the device under test performs its tasks and activities. This is achieved by compiling all the source files of the main application the device under test executes in its operative state and feeding the compiled binaries to the processor during a logic simulation. In this way, it is possible to recreate the working environment where the DUT is supposed to operate throughout its operative lifetime The reason for doing so lies in the fact that aging depends, among other factors, on how much the circuit is stressed, i.e., on how much its internal nodes toggle, which can be easily tracked by evaluating the switching activity on each cell. Two identical circuits, if applied in different scenarios, may show different aging patterns over time, hence why this step is required. While the logic simulation unfolds, a Value Change Dump file is recorded, storing information on the value stored by every cell of the device under test at each time instant. Such file is then converted into a Switching Activity Interfile Format (SAIF) file, that reports for each cell how many toggles have occurred. A snippet from a SAIF file is shown in Fig. 11.3.

Once the preliminary steps are cleared, the tool can proceed in aging the DUT. Initially, the tool parses the information stored in the SDF and SAIF files so that it

```
(INSTANCE U21668
  (NET
    (A
      (T0 25760000) (T1 1977060000) (TX 0)
      (TC 1286) (IG 0)
    )
    (B
      (T0 294440000) (T1 1708380000) (TX 0)
      (TC 10454) (IG 0)
    )
    (C1
      (T0 0) (T1 2002820000) (TX 0)
      (TC 0) (IG 0)
    )
    (C2
      (T0 199440000) (T1 1803380000) (TX 0)
      (TC 7942) (IG 0)
    )
    (ZN
      (T0 1682620000) (T1 320200000) (TX 0)
      (TC 11006) (IG 0)
    )
    (i_12
      (T0 320200000) (T1 1682620000) (TX 0)
      (TC 11006) (IG 0)
    )
    (i_13
      (T0 25760000) (T1 1977060000) (TX 0)
      (TC 1286) (IG 0)
    )
    (i_14
      (T0 0) (T1 2002820000) (TX 0)
      (TC 0) (IG 0)
    )
  )
)
```

Fig. 11.3 Snippet from a SAIF file

Fig. 11.4 An example of an entry from the fresh cell delay dictionary

can use that data as a starting point to calculate the aging delay. First, the SDF file is read, transposing the data stored into the SDF into a fresh cell delay dictionary where each instance name is associated to the cell type and the delay data, a sub-dictionary that maps conditions on input ports to two triples of slow, typical and fast delays for rising and falling transitions from an input to the output. An example of the structure of the dictionary is reported in Fig. 11.4, where only one condition is shown for simplicity.

In a similar fashion, data from the SAIF file too is parsed into a cell switching activity dictionary, where for each cell in the design the correspondent switching activity is recorded. As the switching activity value in Eq. (9.10) ranges in the interval $[0,1]$, rather than using the absolute value reported in the SAIF file, for each cell the tool calculates the normalized switching activity value with respect to a reference signal, e.g., the clock signal.

Next, the tool proceeds to generate an aging model for each cell in the design based on the mathematical equations introduced in Chapter 9 and the switching activity information stored in the related dictionary. Such models are then used to calculate the relative increment in delay (RID) for each cell, defined as:

$$RID = 1 + (AD - FD)/FD \tag{11.1}$$

where AD is the aged delay and FD is the fresh delay obtained by applying Eq. (9.11). The RID is multiplied to the fresh delay reported in the SDF file so that the final aged propagation delay for each cell is obtained. Finally, the tool writes an aged version of the SDF file so that it can be used by commercial tools to generate a list of aged critical paths. Given that the automatic aging tool is designed with the main goal of being easily integrable with any flow that uses commercial tools, the synthesis process required as input can overlap with that required in Chapter 7, thus saving time and computational resources. The logic simulation and critical path extraction processes too can be carried out by following the general ideas described in Section 7.2 and Section 7.3 respectively, but they need to be launched from scratch as the logic simulation defined in Chapter 7 requires the STL to be executed while here the main application must be run, and the critical path extraction process defined here is used to generate a list of aged paths rather than fresh ones.

In conclusion, it is noted that the reason for introducing and using the relative increment in delay rather than the aged delay generated through the mathematical formulas itself lies in the fact that in this way it is possible to achieve more accurate results in predicting the aged delay. Even though the mathematical model is accurate, errors are introduced by formulas as they are obtained by fitting processes over experimental data. Such errors however are mitigated by using the relative values rather than the absolute ones.

11.1 Experimental Results

11.1.1 Case Study

In order to analyze how aging affects a processor core, the automatic aging tool has been validated using PULPino, the 32-bit RISC-V core that has been adopted throughout this PhD thesis. As in other cases, the core was configured to use the RI5CY core, that provides an in-order, 4 pipeline stages core that supports integer, complex and multiplication instructions. Constructing an aging model requires, as a starting block, the knowledge of physical parameters that can only be obtained by characterizing silicon devices. For this reason, it was not possible to use the Silvaco 45nm that has been adopted in all other experiments as it lacks this kind of information. Given that this research activity has been conducted in collaboration

Table 11.1 Paths distribution per modules in fresh circuit

Module	*#Paths*
ALU_Div Adder	51
LoadStore Adder	2,004
Jump_Addr Adder	4,628
Total	6,683

with the University of Grenoble in cooperation with STMicroelectronics, the library used for synthesizing the device under test and conducting experiments is the FDSOI 28nm ST proprietary library that has also been used in [84].

Using a different library to create a post-synthesis netlist of a circuit inevitably leads to generating a different set of critical paths. This is why, even though the processor core was synthesized using the exact same set of configurations used in Section 8.2.1, a total of 6,683 critical paths was obtained, with a slack ranging from 0 to 2.5ns, while the clock signal period is 5ns. Although different, the critical paths can still be grouped in the three main functional groups described in Table 8.4, i.e., paths belonging to an adder found within the divider unit of the ALU, those belonging to an adder belonging to the LoadStore unit and those belonging to an adder that calculates the address that should be taken after a jump instruction. Table 11.1 shows how many paths belong to each group. Thanks to this similarities, it was possible to easily regenerate an STL capable of testing all path delay faults stemming from these paths, following the methodology described in Chapter 8.

Aging a circuit requires, among other aspects, to provide a SAIF file that contains switching activity data for each cell in the device under test. For this reason, two programs were chosen so that switching activity data could be extracted from performing a logic simulation, giving an insight on how much aging depends on the switching activity parameter. The two programs are *basicmath_small* and *qsort* from the *automotive* section of the MiBench-Embedded benchmark [96], as other programs are too large to fit in the device under test's memory. While *basicmath_small* can be run as is, *qsort* required a slight change in the source C code to remove all instances related to `FILE` variables, as they take a considerable amount of instruction RAM: vectors to sort were hence declared and defined in the code rather than read from a file as originally intended. *Basicmath-small* requires 14.73kB of memory

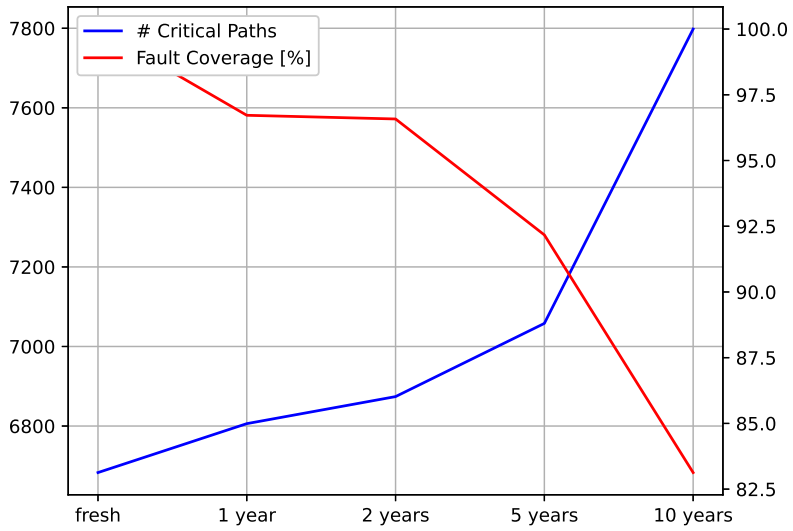


Fig. 11.5 Critical path and path delay fault coverage time evolution for basicmath_small program

and $1,743,500$ clock cycles to execute completely, while *qsort* requires 16.18kB of memory and $2,021,548$ clock cycles to execute.

11.1.2 Achieved Results

All the experiments whose results are reported in this section have been launched on 5 cores of an Intel Xeon CPU E5-2680 v3 machine. The Automatic Aging Tool presented in Chapter 11 has been implemented as a Python class with a set of methods each implementing all the functionalities listed above. Generating the aged SDF file takes no longer than a couple of minutes, provided that all other input files are already available at the start of the experimental session. The extraction of aged critical paths, on the other hand, requires about 12 hours to complete.

Experimental results for the *basicmath_small* and *qsort* programs are reported in Fig. 11.5 and Fig. 11.6, respectively. In both figures the fault coverage trend over time, reported as the percentage of detected faults, is shown in red, while the absolute value of critical paths trend over time is reported in blue.

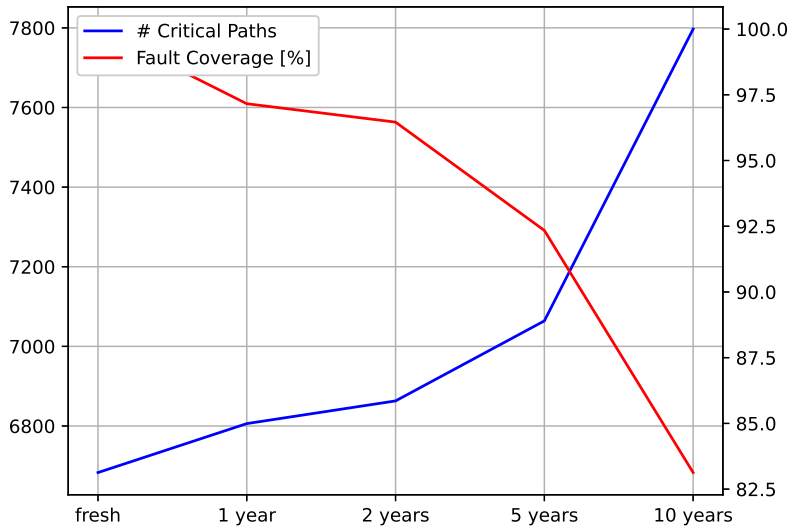


Fig. 11.6 Critical path and path delay fault coverage time evolution for qsort program

In both cases it is possible to notice that the total amount of critical paths grows in time: this can be explained noting that most of the paths that are sub-critical at time zero, i.e., paths whose slack is quite large but not enough for them to be considered critical, slow with time, thus becoming critical paths. This phenomenon becomes more accentuated with time, with a steep growth past the 5 years mark. This reflects on the fault coverage figures as well, showing an overall decrease from the initial 100% fault coverage down to 83.14% for the *basicmath_small* program and 83.12% for the *qsort* program.

In order to better understand how critical paths evolve in time, an additional analysis concerning how many critical paths in the fresh circuit can still be found after aging the device under test has been performed, together with one investigating how many new paths are introduced with aging. Such data is reported in Fig. 11.7 and Fig. 11.8 for the *basicmath_small* program and the *qsort* program, respectively. As for the previous set of data, the two programs show a similar trend over time, both in terms of how many paths remain unchanged and how many new ones are introduced. With the exception of the two years mark, the number of critical paths that never changes over time decreases, losing more than 200 paths with respect to the fresh circuit, while the number of new paths grows, with slightly more than 1300 paths added after 10 years. Focusing on the new paths introduced by aging, it is

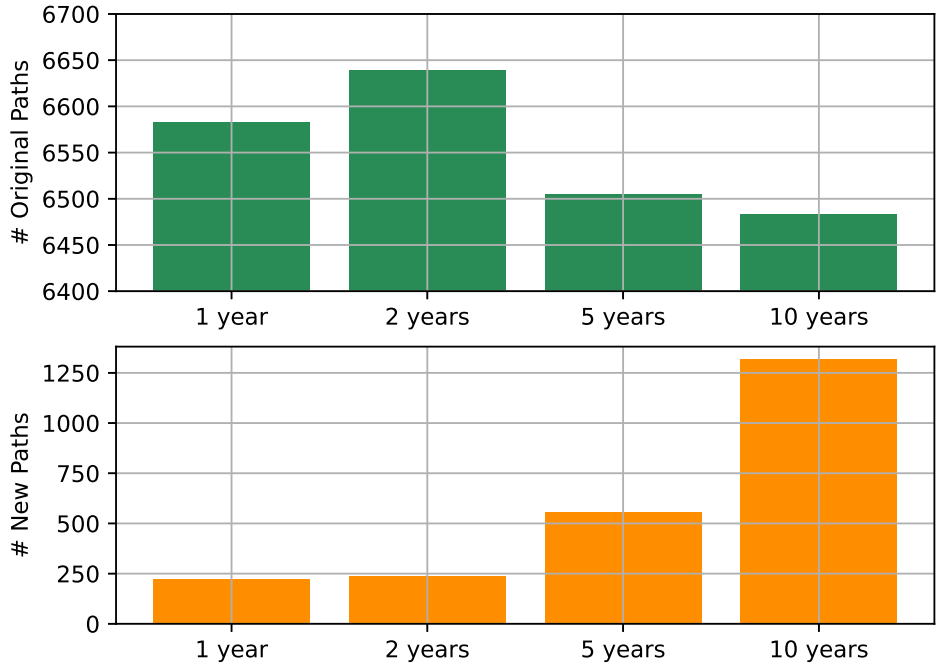


Fig. 11.7 Original and new critical paths evolution for basicmath_small program

interesting to investigate their ranking, i.e., whether they are particularly slow or not. After ten years, 10% of the new paths fall within the top 30% slowest paths for both programs. This shows that, although the vast majority of paths introduced by aging are not the slowest ones, there is still a non-negligible amount of paths whose slack is small enough for them to be among the slowest critical paths. Finally, it is noted that, for both programs, every set of new paths introduced by aging includes that of the antecedent time mark, e.g., all new paths introduced after one year are found after two years, all paths introduced after two years are found after five years and all paths introduced after five years are found after ten years. Moreover, all paths introduced by aging can still be grouped in the three categories reported in Table 11.1, that is, an adder in the ALU divider, an adder in the Load/Store unit and an adder used to calculate the address if a jump instruction is to be taken.

In conclusion, it is possible to state that aging an integrated circuit introduces a non-negligible amount of critical paths over time, with a steep increase after five years of use. Such paths are currently not detected through STLs that are developed

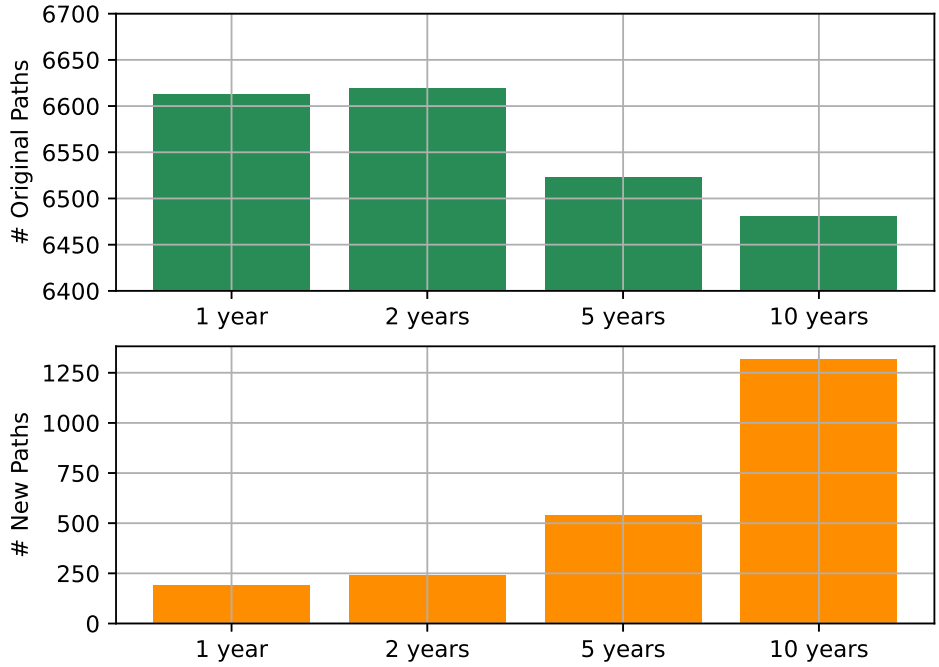


Fig. 11.8 Original and new critical paths evolution for qsort program

tackling faults stemming from paths found in the fresh circuit, thus posing a problem when it comes to ensuring the safety and reliability of the device under test over long periods of time. For this reason, test engineers should develop the STL also taking into account the set of critical paths that are to be tested as the circuit ages, making sure that it can ensure a satisfactory fault coverage throughout the operative lifetime of the device, choosing between the possibility of having (i) an all-encompassive STL since time zero, capable of detecting failures stemming from faults in all critical paths including those due to aging, or (ii) a modular STL that is capable of enabling chunks of code, adapting to the paths as they age. Whatever the choice, such STL can be effectively developed by following the methodology presented in Chapter 8.

11.2 Chapter Summary

This chapter describes a methodology on how to age an integrated circuit, with the goal of ensuring its safety and reliability over a long period of time. This methodology complements the work presented in Part II, allowing test engineers to develop testing solutions, e.g., STLs, that are capable to ensure the safety of the DUT over its lifetime.

In order to age an integrated circuit few input data are required, namely, physical parameters of the technology library used to synthesize the circuit to be aged, a SDF file containing propagation delays for each cell in the fresh circuit, and a SAIF file storing switching activity information for each cell of the design. Given such preliminary data, the aging tool implements the set of equations and linear regression models presented in [84, 95], allowing to automatically generate an aged SDF for the device under test. The aged SDF has exactly the same structure and syntax of the first one, with the exception of aged propagation delays for each cell. Thanks to that, it is possible to generate new lists of aged critical paths, understanding how critical paths, and thus the path delay fault coverage, change over time. Experimental data gathered on two programs from the MiBench-Embedded benchmark show that as the circuit age, more and more paths that once were sub-critical, i.e., their slack was large but not enough for them to be considered critical, become critical, leading to a decrease in fault coverage that after 10 years drop from an initial 100% to about 83% for both programs. Moreover, although most of the new critical paths introduced by aging are among the fastest of the whole set, there is still a non-negligible amount of

new critical paths that are quite large, with a 10% of the new paths falling within the top 30% slowest paths for both programs. Thanks to this information, test engineers can harden STLs for path delay faults so that they include test vectors for path delay faults that may originate with time, ensuring that strict levels of reliability are met throughout the operative lifetime of the device and proving the effectiveness of this methodology.

Chapter 12

Conclusions and Achievements

This PhD thesis aims at presenting and validating new techniques for detecting and mitigating the issues that stem in integrated circuits when affected from delay faults, both in the form of transition and path delay faults, together with an analysis on how aging affects path delay faults covering. The reason for focusing on this class of faults resides in the fact that dynamic fault models, such as delay faults, better represent the actual defects that can be found in modern, state-of-the-art integrated circuits thanks to the fact that they take into account the DUT's timing behavior, differently from static fault models, e.g., the stuck-at fault one. Throughout this PhD thesis, functional testing solutions in the form of SBST techniques have been used, as they are cheap, reliable and allow for at-speed tests that are crucial when targeting delay faults, being a great solutions to be used in conjunction with DfT techniques to ensure the DUT's reliability and safety throughout its operative lifetime.

Part I of this thesis is dedicated to solutions for the transition delay fault model. This fault model shares similarities with the stuck-at one, hence why STLs for stuck-at faults already provide a good basis for testing transition delay faults. Rather than focusing on techniques to develop STLs from scratch, two mechanisms to harden already available STLs are provided in Chapter 3 and Chapter 4 respectively, so that not-observed transition delay faults are detected, providing an increase in fault coverage with minimal code additions. The approach in Chapter 3 is purely software-based, relying on a subdivision of not-observed transition delay faults into two categories based on where their effects propagate and stop inside the device under test leading to two different fault detection mechanisms, while the one in

Chapter 4 proposes two algorithms that leverage the post-silicon debug hardware that is typically found within modern SOCs to make the effects of the aforementioned transition delay faults observable. Experimental results show that the methodology presented in Chapter 3 is capable of recovering up to 99.96% of UAR faults and up to 15.37% HR faults with a limited code size increase, the worst case scenario requiring an additional 31.74% of the original code size. HR faults can be easily detected through the approach presented in Chapter 4, with experimental data that shows how a 32-bit trace buffer helps detecting more than 90% of recoverable transition delay faults, while larger trace buffers, e.g. 128-bit wide buffers, allow for detecting all recoverable transition delay faults.

Part II of this thesis presents techniques for developing STLs geared towards path delay faults. Differently from the transition delay fault model, path delay faults are quite different to stuck-at faults, and methods to develop STLs for this fault model are not as mature. Moreover, no commercial fault simulation tool is capable of performing a sequential fault simulation of path delay faults, that is, a fault simulation where test vectors are provided by the execution of test programs, without the usage of scan chains. For this reason, Chapter 7 describes a path delay fault simulation flow that has been developed from scratch that allows to perform sequential fault simulations of path delay faults on integrated circuits. The fault simulation flow is comprised of several steps, starting from a synthesis of the DUT followed by the extraction of critical paths that can be structurally tested, i.e., by means of an ATPG engine. Test vectors recorded in the form of VCD files obtained by performing logic simulations of an STL being executed on the device under test are then applied first at a combinational level to see how many path delay faults can be detected at the primary and pseudo primary outputs of the DUT, followed by another top-level simulation that aims at propagating the effects of the latter group of faults throughout the DUT, looking to see if it is possible to observe their effects at primary outputs. The path delay fault simulation tool is used to evaluate how well STLs written for other fault models perform with path delay faults, showing how in the best case scenario only 51.93% of path delay faults can be detected. For this reason, Chapter 8 proposes a systematic and automatic method on how to develop STLs for path delay faults. To do so, once a set of critical paths — from which path delay faults will be considered — are extracted, an ATPG is launched to extract test vectors that are capable of detecting path delay faults at a combinational level. Since the ATPG is not aware that these test vectors must be applied through

functional means, that is, by embedding them in STLs, it is necessary to provide functional constraints to it so that the produced vectors can be replicated through programs. Once the test vectors generation step is completed, such vectors must be converted in instructions that will constitute the final STL, also making sure to add *store* instructions to make the effects observable at primary outputs. Experimental data shows that all path delay faults found within a 32 bit, in-order 5 stages pipelined RISC-V CPU can be detected.

Lastly, Part III of this thesis discusses about aging phenomena and how they affect modern integrated circuits. Chapter 9 provides an extensive background on the mathematical equations through which it is possible to estimate aging effects on integrated circuits. Chapter 11 describes the automatic aging framework that is capable of generating aged delays for each cell of the design, provided physical data on the technology library used for synthesis together with the fresh VCD and SAIF file of the circuit to be aged. Since it is not possible to have physical data for each aging configuration possible, linear regression models are used to interpolate data and obtain a close estimate of aged propagation delays. Experimental data show that the number of critical paths increase over time, with several new ones that are introduced after the 5 years mark. The fault coverage figures degrade accordingly, starting from an initial 100% on a 32 bit, in-order 5 stages pipelined RISC-V CPU that evolves to slightly more than 83% fault coverage after 10 years. Such faults should hence be tested by additional code portions to be added to the original STL and enable either at time zero or gradually enabled with time, as the circuit ages.

12.1 Future Directions

The work presented in this PhD thesis and its achievements open the way for new discussions and investigations. Starting from Part I, the techniques presented in Chapter 3 can be used as a starting point to further improve the fault coverage of transition delay faults belonging to the Hidden Registers group, while the methodology described in Chapter 4 could be tested on an actual SoC with post-silicon debug circuitry to assess its performance. As for Part II, there is room for improving the degree of automation in generating STLs for path delay faults as described in Chapter 8, e.g., in identifying the set of functional constraints to be fed to the ATPG to generate functional test vectors. Moreover, all the techniques described so far for

transition and path delay faults can be applied to larger designs, so that they can be further improved and adapted to more complex circuits. Finally, the automatic aging tool is strictly dependent on the technology library adopted and it currently only takes into account the propagation delay of cells, without taking into account the interconnections between gates. New research can be done in this area, either collecting new data on other libraries and checking if aging occurs differently based on the library and understanding how interconnections affect the aging of propagation delay through paths.

I strongly hope, and believe, that the aforementioned research points will be of interest for new works and investigations.

References

[1] Mohammad Tehranipoor, Ke Peng, and Krishnendu Chakrabarty. *Delay Test and Small-Delay Defects*, pages 21–36. Springer New York, New York, NY, 2012.

[2] ETH Zurich and Università di Bologna. PULPino microcontroller system. https://github.com/pulp-platform/pulpino, 2022.

[3] Hannah Ritchie Max Roser. Moore's law: The number of transistors on microchips doubles every two years. https://ourworldindata.org/uploads/2020/11/Transistor-Count-over-time.png, 2023.

[4] P. Bernardi, M. Bonazza, E. Sanchez, M. Sonza Reorda, and O. Ballan. On-line functionally untestable fault identification in embedded processor cores. In *Design, Automation & Test in Europe Conference Exhibition (DATE)*, pages 1462–1467, 2013.

[5] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Sonza Reorda. Microprocessor software-based self-testing. *IEEE Design Test of Computers*, 27(3):4–19, 2010.

[6] M. Psarakis, D. Gizopoulos, M. Hatzimihail, A. Paschalis, A. Raghunathan, and S. Ravi. Systematic software-based self-test for pipelined processors. In *ACM/IEEE Design Automation Conference (DAC)*, pages 393–398, 2006.

[7] N. Hage, R. Gulve, M. Fujita, and V. Singh. On testing of superscalar processors in functional mode for delay faults. In *30th IEEE Intl. Conference on VLSI Design and 16th IEEE Intl. Conference on Embedded Systems (VLSID)*, pages 397–402, 2017.

[8] A. S. Oyeniran, R. Ubar, M. Jenihhin, and J. Raik. Implementation-independent functional test for transition delay faults in microprocessors. In *Euromicro Conference on Digital System Design (DSD)*, pages 646–650, 2020.

[9] K. Christou, M. K. Michael, P. Bernardi, M. Grosso, E. Sanchez, and M. S. Reorda. A novel sbst generation technique for path-delay faults in microprocessors exploiting gate- and rt-level descriptions. In *26th IEEE VLSI Test Symposium (vts 2008)*, pages 389–394, April 2008.

[10] C. H. . Wen, L. . Wang, Kwang-Ting Cheng, Kai Yang, Wei-Ting Liu, and Ji-Jan Chen. On a software-based self-test methodology and its application. In *23rd IEEE VLSI Test Symposium (VTS'05)*, pages 107–113, 2005.

[11] V. Singh, M. Inoue, K. K. Saluja, and H. Fujiwara. Instruction-based self-testing of delay faults in pipelined processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(11):1203–1215, Nov 2006.

[12] P. Bernardi, R. Cantoro, S. De Luca, E. Sánchez, and A. Sansonetti. Development flow for on-line core self-test of automotive microcontrollers. *IEEE Transactions on Computers*, 65(3):744–754, 2016.

[13] Wei-Cheng Lai, A. Krstic, and Kwang-Ting Cheng. Test program synthesis for path delay faults in microprocessor cores. In *IEEE Intl. Test Conference*, pages 1080–1089, 2000.

[14] P. Bernardi, M. Grosso, E. Sanchez, and M. S. Reorda. A deterministic methodology for identifying functionally untestable path-delay faults in microprocessor cores. In *2008 Ninth Intl. Workshop on Microprocessor Test and Verification*, pages 103–108, Dec 2008.

[15] R. Cantoro, S. Carbonara, A. Floridia, E. Sanchez, M. Sonza Reorda, and J. Mess. An analysis of test solutions for COTS-based systems in space applications. In *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 59–64, Oct 2018.

[16] Andreas Riefert, Riccardo Cantoro, Matthias Sauer, Matteo Sonza Reorda, and Bernd Becker. A flexible framework for the automatic generation of sbst programs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(10):3055–3066, 2016.

[17] R. Cantoro, D. Foti, S. Sartoni, M. S. Reorda, L. Anghel, and M. Portolan. New perspectives on core in-field path delay test. In *2020 IEEE International Test Conference (ITC)*, pages 1–5, 2020.

[18] Riccardo Cantoro, Patrick Girard, Riccardo Masante, Sandro Sartoni, Matteo Sonza Reorda, and Arnaud Virazel. Self-test libraries analysis for pipelined processors transition fault coverage improvement. In *2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 1–4, 2021.

[19] Felipe Augusto da Silva, Ahmet Cagri Bagbaba, Said Hamdioui, and Christian Sauer. Combining fault analysis technologies for iso26262 functional safety verification. In *2019 IEEE 28th Asian Test Symposium (ATS)*, pages 129–1295, 2019.

[20] Daniel Kraak, Mottaqiallah Taouil, Innocent Agbo, Said Hamdioui, Pieter Weckx, Stefan Cosemans, and Francky Catthoor. Parametric and functional degradation analysis of complete 14-nm finfet sram. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(6):1308–1321, 2019.

[21] Sandra Irobi, Zaid Al-Ars, and Said Hamdioui. Memory test optimization for parasitic bit line coupling in srams. In *2011 Sixteenth IEEE European Test Symposium*, pages 205–205, 2011.

[22] M. Grosso, S. Rinaudo, A. Casalino, and M. Sonza Reorda. Software-Based Self-Test for Transition Faults: a Case Study. In *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 76–81, 2019.

[23] A. Apostolakis, D. Gizopoulos, M. Psarakis, D. Ravotto, and M. Sonza Reorda. Test program generation for communication peripherals in processor-based soc devices. *IEEE Design Test of Computers*, 26(2):52–63, March 2009.

[24] Riccardo Cantoro, Sandro Sartoni, and Matteo Sonza Reorda. In-field functional test of can bus controllers. In *2020 IEEE 38th VLSI Test Symposium (VTS)*, pages 1–6, 2020.

[25] L. Bolzani, E. Sanchez, M. Schillaci, M. Sonza Reorda, and G. Squillero. An automated methodology for cogeneration of test blocks for peripheral cores. In *13th IEEE International On-Line Testing Symposium (IOLTS 2007)*, pages 265–270, July 2007.

[26] A. van de Goor, G. Gaydadjiev, and S. Hamdioui. Memory testing with a risc microcontroller. In *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pages 214–219, March 2010.

[27] A. Cook, D. Ull, M. Elm, H. Wunderlich, H. Randoll, and S. Döhren. Reuse of structural volume test methods for in-system testing of automotive asics. In *2012 IEEE 21st Asian Test Symposium*, pages 214–219, 2012.

[28] Felipe Augusto da Silva, Riccardo Cantoro, Said Hamdioui, Sandro Sartoni, Christian Sauer, and Matteo Sonza Reorda. A systematic method to generate effective stls for the in-field test of can bus controllers. *Electronics*, 11(16), 2022.

[29] Hitex. Microcontroller self-test libraries. https://www.hitex.com/tools-components/software-components/selftest-libraries-safety-libs/, 2022.

[30] STMicroelectronics. Guidelines for obtaining IEC 60335 Class B certification for any STM32 application. http://www.st.com/content/ccc/resource/technical/document/application_note/02/1a/91/78/e4/15/4d/35/CD00290100.pdf/files/CD00290100.pdf/jcr:content/translations/en.CD00290100.pdf, Mar 2016.

[31] Cypress Semiconductor. FM3 and FM4 Family, IEC61508 SIL2 Self-Test Library. https://www.cypress.com/file/249196/download, 2022.

[32] Renesas Electronics. SSP Supplemental Add-Ons. https://www.renesas.com/en-eu/products/synergy/software/add-ons.html, 2022.

[33] Microchip Technology Inc. 16-bit CPU Self-Test Library User's Guide. http://ww1.microchip.com/downloads/en/DeviceDoc/52076a.pdf, 2022.

[34] ARM. What is functional safety? https://www.arm.com/technologies/safety, 2022.

[35] NXP Semiconductors. S32 SDK for S32K1 microcontrollers. https://www.nxp.com/support/developer-resources/run-time-software/s32-sdk/s32-sdk-for-s32k1-microcontrollers:S32SDK-ARMK1, 2022.

[36] M.A. Breuer. The effects of races, delays, and delay faults on test generation. *IEEE Transactions on Computers*, C-23(10):1078–1092, 1974.

[37] Irith Pomeranz. Skewed-load tests for transition and stuck-at faults. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(10):1969–1973, 2019.

[38] J. Savir. Skewed-load transition test: Part i, calculus. In *Proceedings International Test Conference 1992*, pages 705–, 1992.

[39] S. Patil and J. Savir. Skewed-load transition test: Part ii, coverage. In *Proceedings International Test Conference 1992*, pages 714–, 1992.

[40] J. Savir and S. Patil. Broad-side delay test. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(8):1057–1064, 1994.

[41] J. Savir and S. Patil. On broad-side delay test. In *Proceedings of IEEE VLSI Test Symposium*, pages 284–290, 1994.

[42] Irith Pomeranz. Partitioning functional test sequences into multicycle functional broadside tests. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 29(1):89–99, 2021.

[43] Irith Pomeranz and Sudhakar M. Reddy. Forming multi-cycle tests for delay faults by concatenating broadside tests. In *2010 28th VLSI Test Symposium (VTS)*, pages 51–56, 2010.

[44] Irith Pomeranz. Skewed-load test cubes based on functional broadside tests for a low-power test set. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23(3):593–597, 2015.

[45] Irith Pomeranz. Multicycle broadside and skewed-load tests for test compaction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(1):262–266, 2020.

[46] Irith Pomeranz. Generation of mixed broadside and skewed-load diagnostic test sets for transition faults. In *2011 IEEE 17th Pacific Rim International Symposium on Dependable Computing*, pages 45–52, 2011.

[47] Irith Pomeranz. Direct computation of lfsr-based stored tests for broadside and skewed-load tests. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(12):5238–5246, 2020.

[48] C. Y. Chen and J. L. Huang. Reinforcement-Learning-Based Test Program Generation for Software-Based Self-Test. In *IEEE Asian Test Symposium (ATS)*, pages 73–735, 2019.

[49] Ying Zhang, Zebo Peng, Jianhui Jiang, Huawei Li, and Masamro Fujita. Temperature-aware software-based self-testing for delay faults. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 423–428, 2015.

[50] Ying Zhang, Yi Ding, Zebo Peng, Huawei Li, Masahiro Fujita, and Jianhui Jiang. Bmc-based temperature-aware sbst for worst-case delay fault testing under high temperature. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 30(11):1677–1690, 2022.

[51] A. Ruospo, R. Cantoro, E. Sanchez, P. D. Schiavone, A. Garofalo, and L. Benini. On-line testing for autonomous systems driven by risc-v processor design verification. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–6, 2019.

[52] A. Jasnetski, R. Ubar, and A. Tsertov. On automatic software-based self-test program generation based on high-level decision diagrams. In *IEEE Latin-American Test Symposium (LATS)*, pages 177–177, 2016.

[53] A. Jasnetski, R. Ubar, and A. Tsertov. Automated software-based self-test generation for microprocessors. In *International Conference Mixed Design of Integrated Circuits and Systems (MIXDES)*, pages 453–458, 2017.

[54] E. Sanchez, M. Sonza Reorda, G. Squillero, and M. Violante. Automatic generation of test sets for sbst of microprocessor ip cores. In *2005 18th Symposium on Integrated Circuits and Systems Design*, pages 74–79, 2005.

[55] Riccardo Cantoro, Francesco Garau, Patrick Girard, Nima Kolahimahmoudi, Sandro Sartoni, Matteo Sonza Reorda, and Arnaud Virazel. Effective techniques for automatically improving the transition delay fault coverage of self-test libraries. In *2022 IEEE European Test Symposium (ETS)*, pages 1–2, 2022.

[56] Joon-Sung Yang and Nur A. Touba. Improved trace buffer observation via selective data capture using 2-d compaction for post-silicon debug. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(2):320–328, 2013.

[57] Binod Kumar, Jay Adhaduk, Kanad Basu, Masahiro Fujita, and Virendra Singh. A methodology to capture fine-grained internal visibility during multisession silicon debug. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(4):1002–1015, 2020.

[58] Hyunggoy Oh, Taewoo Han, Inhyuk Choi, and Sungho Kang. An on-chip error detection method to reduce the post-silicon debug time. *IEEE Transactions on Computers*, 66(1):38–44, 2017.

[59] Sandeep Chandran, Preeti Ranjan Panda, Smruti R. Sarangi, Ayan Bhattacharyya, Deepak Chauhan, and Sharad Kumar. Managing trace summaries to minimize stalls during postsilicon validation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(6):1881–1894, 2017.

[60] Riccardo Cantoro, Francesco Garau, Riccardo Masante, Sandro Sartoni, Virendra Singh, and Matteo Sonza Reorda. Exploiting post-silicon debug hardware to improve the fault coverage of software test libraries. In *2022 IEEE 40th VLSI Test Symposium (VTS)*, pages 1–7, 2022.

[61] Silvaco. Silvaco 45nm open cell library.

[62] J. Mahmod, S. Millican, U. Guin, and V. Agrawal. Special session: Delay fault testing - present and future. In *2019 IEEE 37th VLSI Test Symposium (VTS)*, pages 1–10, 2019.

[63] U.E. Sparmann and L. Koller. Improving path delay fault testability by path removal. In *Proceedings. 16th IEEE VLSI Test Symposium (Cat. No.98TB100231)*, pages 200–208, 1998.

[64] Sabir Hussain, M A Raheem, and Afaq Ahmed. Sic-tpg for path delay fault detection in vlsi circuits using scan insertion method. In *2021 Devices for Integrated Circuit (DevIC)*, pages 1–5, 2021.

[65] Irith Pomeranz. On the detection of path delay faults by functional broadside tests. In *2012 17th IEEE European Test Symposium (ETS)*, pages 1–6, 2012.

[66] Irith Pomeranz. Gepdfs: Path delay faults based on two-cycle gate-exhaustive faults. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(7):2315–2322, 2022.

[67] P. Bernardi, M. Grosso, E. Sanchez, and M. Sonza Reorda. On the automatic generation of test programs for path-delay faults in microprocessor cores. In *12th IEEE European Test Symposium (ETS'07)*, pages 179–184, May 2007.

[68] Sankar Gurumurthy, Ramtilak Vemu, Jacob A. Abraham, and Daniel G. Saab. Automatic generation of instructions to robustly test delay defects in processors. In *12th IEEE European Test Symposium (ETS'07)*, pages 173–178, 2007.

[69] Nikolaos I. Deligiannis, Riccardo Cantoro, Tobias Faller, Tobias Paxian, Bernd Becker, and Matteo Sonza Reorda. Effective sat-based solutions for generating functional sequences maximizing the sustained switching activity in a pipelined processor. In *2021 IEEE 30th Asian Test Symposium (ATS)*, pages 73–78, 2021.

[70] Tobias Faller, Philipp Scholl, Tobias Paxian, and Bernd Becker. Towards sat-based sbst generation for risc-v cores. In *2021 IEEE 22nd Latin American Test Symposium (LATS)*, pages 1–2, 2021.

[71] Lorena Anghel, Riccardo Cantoro, Riccardo Masante, Michele Portolan, Sandro Sartoni, and Matteo Sonza Reorda. Self-test library generation for in-field test of path delay faults. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2023.

[72] J. Perez Acle, R. Cantoro, E. Sanchez, M. Sonza Reorda, and G. Squillero. Observability solutions for in-field functional test of processor-based systems: A survey and quantitative test case evaluation. *Microprocessors and Microsystems*, 47:392 – 403, 2016.

[73] J. Chen, L. Winemberg, and M. Tehranipoor. Identification of testable representative paths for low-cost verification of circuit performance during manufacturing and in-field tests. In *32nd IEEE VLSI Test Symposium (VTS)*, pages 1–6, April 2014.

[74] N. Ahmed, M. Tehranipoor, and V. Jayaram. Timing-based delay test for screening small delay defects. In *2006 43rd ACM/IEEE Design Automation Conference*, pages 320–325, 2006.

[75] X. Fu, H. Li, and X. Li. Testable path selection and grouping for faster than at-speed testing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(2):236–247, 2012.

[76] Riccardo Cantoro, Patrick Girard, Riccardo Masante, Sandro Sartoni, Matteo Sonza Reorda, and Arnaud Virazel. Self-test libraries analysis for pipelined processors transition fault coverage improvement. In *2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 1–4, 2021.

[77] Riccardo Cantoro, Patrick Girard, Riccardo Masante, Sandro Sartoni, Matteo Sonza Reorda, and Arnaud Virazel. Effective techniques for automatically improving the transition delay fault coverage of self-test libraries. In *2022 IEEE European Test Symposium (ETS)*, 2022 [In press].

[78] F. A. da Silva, A. C. Bagbaba, S. Sartoni, R. Cantoro, M. Sonza Reorda, S. Hamdioui, and C. Sauer. Determined-Safe Faults Identification: A step towards ISO26262 hardware compliant designs. In *IEEE European Test Symposium (ETS)*, pages 1–6, 2020.

[79] T. Sakurai and A.R. Newton. Alpha-power law mosfet model and its applications to cmos inverter delay and other formulas. *IEEE Journal of Solid-State Circuits*, 25(2):584–594, 1990.

[80] Sachin S. Sapatnekar. What happens when circuits grow old: Aging issues in cmos design. In *2013 International Symposium onVLSI Design, Automation, and Test (VLSI-DAT)*, pages 1–2, 2013.

[81] Basel Halak, Vasileios Tenentes, and Daniele Rossi. The impact of bti aging on the reliability of level shifters in nano-scale cmos technology. *Microelectronics Reliability*, 67:74–81, 2016.

[82] Yong Zhao and Hans G. Kerkhoff. Highly dependable multi-processor socs employing lifetime prediction based on health monitors. In *2016 IEEE 25th Asian Test Symposium (ATS)*, pages 228–233, 2016.

[83] M. Altieri, S. Lesecq, E. Beigne, and O. Heron. Towards on-line estimation of bti/hci-induced frequency degradation. In *2017 IEEE International Reliability Physics Symposium (IRPS)*, pages CR–6.1–CR–6.6, 2017.

[84] Kalpana Senthamarai Kannan. *Management des performances de sûreté et de sécurité pour les applications automotives et IoT*. PhD thesis, Université Grenoble Alpes, 2021. Thèse de doctorat dirigée par Anghel, Lorena et Portolan, Michele Nanoélectronique et nanotechnologie Université Grenoble Alpes 2021.

[85] Dieter K. Schroder and Jeff A. Babcock. Negative bias temperature instability: Road to cross in deep submicron silicon semiconductor manufacturing. *Journal of Applied Physics*, 94(1):1–18, 2003.

[86] T.H. Ning. Hot-electron emission from silicon into silicon dioxide. *Solid-State Electronics*, 21(1):273–282, 1978.

[87] Ho Joon Lee and Kyung Ki Kim. Analysis of time dependent dielectric breakdown in nanoscale cmos circuits. In *2011 International SoC Design Conference*, pages 440–443, 2011.

[88] J.W. McPherson. Time dependent dielectric breakdown physics – models revisited. *Microelectronics Reliability*, 52(9):1753–1760, 2012. Special Issue 23rd European Symposium on the Reliability of Electron Devices, Failure Physics and Analysis.

[89] Tony Tae-Hyoung Kim, Pong-Fei Lu, Keith A. Jenkins, and Chris H. Kim. A ring-oscillator-based reliability monitor for isolated measurement of nbti and pbti in high-k/metal gate technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23(7):1360–1364, 2015.

[90] John Keane, Xiaofei Wang, Devin Persaud, and Chris H. Kim. An all-in-one silicon odometer for separately monitoring hci, bti, and tddb. *IEEE Journal of Solid-State Circuits*, 45(4):817–829, 2010.

[91] Neil Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison-Wesley Publishing Company, USA, 4th edition, 2010.

[92] Ajith Sivadasan, S. Mhira, Armelle Notin, A. Benhassain, V. Huard, Etienne Maurin, F. Cacho, L. Anghel, and A. Bravaix. Architecture- and workload-dependent digital failure rate. In *2017 IEEE International Reliability Physics Symposium (IRPS)*, pages CR–8.1–CR–8.4, 2017.

[93] Jeremy Watt, Reza Borhani, and Aggelos K. Katsaggelos. *Machine Learning Refined: Foundations, Algorithms, and Applications*. Cambridge University Press, USA, 1st edition, 2016.

[94] Max Kuhn and Kjell Johnson. *Applied Predictive Modeling*. Springer, 2013.

[95] Mauricio Altieri Scarpato. *Estimation de la performance des circuits numériques sous variations PVT et vieillissement*. PhD thesis, Université Grenoble Alpes, 2017. Thèse de doctorat dirigée par Beigné, Édith et Lesecq, Suzanne Nano electronique et nano technologies Université Grenoble Alpes (ComUE) 2017.

[96] Jeremy Bennett. Mibench-embedded benchmark. https://github.com/embecosm/mibench/tree/master, 2023.