POLITECNICO DI TORINO

Master's Degree in Electronics Engineering

Politecnico
di Torino

Master's Degree Thesis

Functional test solutions for delay faults in CPUs

Supervisors

Prof. Matteo SONZA REORDA

Prof. Riccardo CANTORO

PhD. Sandro SARTONI

Candidate

NIMA KOLAHIMAHMOUDI

October 18, 2022

"If you don't like where you're heading,
there's no shame in going back
and changing your path!"

Abstract

New advanced semiconductor technologies are increasingly adopted in critical applications. Such technologies are extra complex and sophisticated, leading to more frequent physical defects and reduced operative lifetime. Most of these defects are tested by targeting delay faults, such as transition delay faults (TDFs). By detecting these defects, it is possible to have more reliable applications. The presented thesis work focuses on the functional test for transition delay faults (TDFs) and investigates for possible solutions. In order to perform functional test on integrated circuits, Self-test Libraries (STLs) is one of the widely used technique. These libraries, guarantee that processor behaves correctly, during its period of the operative lifetime. Concerning the development of the STLs for stuck-at faults, a significant amount of the efforts required by test engineers. Moreover, developing libraries for delay faults are even more formidable.

The proposed method, aims automation of the development of the STLs, targeting TDFs, using the available STLs for stuck-at faults (SAFs). Moreover, it relies on identification of the transition delay faults which, are excited but, not observed. Then, adds apt instructions which may observe and detect transition delay faults.

The main target of the proposed method, is the faults reached to flip-flop in the digital circuit. These flip-flops are divided in two main groups:

- User Accessible Registers (UARs): These registers can be accessed directly via CPU's instruction set, therefore faults reach to these locations can be observed through CPU's instruction set. For instance, for the faults propagated to the register file, the proposed approach is to insert a store instruction right after its observation in the register file.

- Hidden Registers (HRs): All other registers that cannot be observed through CPU's instruction set, fall within the Hidden registers. These registers are deeply embedded in the processor core, either belonging to pipeline registers or inner sub-modules, which makes particularly hard to propagate faults observed in these locations to the primary outputs of the digital circuit. In this thesis work, to detect the faults propagated to hidden registers, the applied method is to insert instructions that are in charge of the propagation of the SAFs from HRs to primary outputs, in a specific part of the STL code where TDF is observed in.

The results of the test performed on a RISC-V processor shown in tables 1 and 2, illustrates that, the method is adequate to systematically detect a significant percentage of the targeted faults using reasonable computational effort and test code size increase. Specifically, the proposed method is adequate for faults propagated to the UARs. In a brief gander to table 2, it is observable that, by adding instructions maximum of 6.34 kBytes in size to STL code, it is possible to detect at least 98.76 % of the faults propagate to the UARs. However, in table 1, it is observable that, the applied methods are not as adequate as the employed methods for UARs. It is possible to improve the number of the detected faults up to 15.34 % by adding only 0.92 kBytes in size to the STL code.

Table 1: 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

Table 2: 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

Acknowledgements

In the first place, I'm grateful for the given opportunity by professor Matteo Sonza Reorda. Also, I'm grateful for the comprehensive support from professor Riccardo Cantoro, and Sandro Sartoni during the period of this thesis work.

In the following, I'm happy for getting to know friends in Turin. When I arrived in this beautiful city, I didn't know anyone. Now, after 3 years I have very kind friends and colleagues, both in the place I live and in the CAD group and Lab3 I worked during my thesis activities. I'd like to thank Juan David and Esteban for all their help without any expectations.

I have a special thanks to my parents Ahmad and Marziyeh, my dearest sister Ayda, my grandparents, and my uncles for believing in me, and supporting me endlessly in every single stage of my life.

Finally, I'd like to keep alive memories of my beloved aunt Akram and uncle Morteza, who are no longer with me and I wished them to be alive and make them happy about my achievements. May they rest in peace.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

New advanced semiconductor technologies are increasingly adopted in emerging applications, thanks to their enhanced working frequencies and computational capabilities. Such technologies, however, are extremely complex and sophisticated, leading to more frequent physical defects and reduced operative lifetime. Testing integrated circuits (ICs), hence, is of paramount importance. Most of these defects are tested by targeting not only static, but also dynamic defects, often modeled as delay faults, i.e., faults that affect the timing behavior of the device under test (DUT), such as transition delay faults (TDFs) or path delay faults (PDFs).

Testing integrated circuits can be done using two different approaches. The most common one relies on the adoption of Design-for-Testability (DfT) solutions, which usually require the usage of additional hardware modules such as Logic BIST or scan chains. Such modules are integrated within the DUT and are employed to apply test vectors and monitor the circuit's response to the aforementioned vectors. Although based on mature technology and supported by most EDA tools, such solutions impose non-negligible timing and area overheads that could degrade performances. Moreover, functionally untestable faults[1] (FUFs), i.e., faults whose effects can never be observed within functional scenarios, will possibly be detected, leading to a phenomenon known as *overtesting*, which leads to a yield loss. These issues can be overcome by adopting another testing solution, namely functional testing. In the form of Software-Based Self-Test (SBST), functional testing[2, 3] is based on the execution of a set of Self-Test Libraries (STLs) by the DUT[4].

The results, produced by the test programs are compacted into a signature that is compared against the golden circuit's one to look for the presence of structural faults. This approach has been proved effective both when processor cores[5, 6, 7, 8, 9, 10, 11, 12] and peripherals[13, 14, 15, 16] are tested, and several companies provide STLs for their products [17, 18, 19, 20]. SBST is a desirable solution for *in-field testing*, i.e., when the device's reliability and safety has to be guaranteed throughout the operative lifetime. SBST is reliable, cheap, and flexible — STLs can be developed such that they fit the idle slots of the application run by the DUT, hence avoiding any service interruption. Thanks to these properties, SBST can be successfully used whenever compliance to standards such as the *ISO26262 standard* for automotive systems is required[4].

However, developing STLs from scratch is not a trivial task, more so when targeting delay faults on complex devices. For test programs to achieve high fault coverage figures, they must be able to excite as many faults as possible from the whole DUT and make their effects observable at primary outputs (POs) by using instructions from the system's instruction set architecture, only [21]. Achieving this, requires a non-negligible amount of manual effort by the test engineer. Moreover, when fault reports are available, understanding why certain faults are not detected (possibly isolating the contribution of FUFs) is not always easy. The work in [22] moves the first step in classifying not-observed transition delay faults, giving some insights on where these fault effects propagated and stopped and defining some upper boundaries on how much the final TDF coverage can be increased.

In this thesis work, an automatic and systematic methodology to increase the TDF coverage of STLs is proposed, by detecting faults identified in [22] on complex pipelined processor cores, starting from a set of test programs devised for stuck-at faults (SAFs). This feature is achieved by dividing such faults into two main groups - namely, User Accessible Register (UAR) and Hidden Register (HR) faults - and deploying appropriate strategies. Transition delay fault model is chosen over the path delay one because TDFs are much better supported by both standards and EDA tools than PDFs. The main contributions of this work are:

- A set of techniques able to identify which instructions are capable of detecting not-observed transition delay faults;

- A test flow able to automatically add the previously identified instructions into the right place within existing STLs to improve the final fault coverage;
- Data on how much overhead is added to the original STL after its enhancement.

This approach is validated on a RISC-V core, using available commercial tools and a set of pre-existing STLs targeting SAFs. The reported results show that it is possible to detect most of the aforementioned transition delay faults (increasing the TDF coverage by up to 15%) with a reasonable test time increase (from 15 % to 22%) and computational effort[4].

The thesis work document is organized as follows: in chapter2, a background on the transition delay fault model and related works is outlined, while in chapter3, the proposed approach is described. In chapter4, the experimental results are presented and, finally, in 5 the conclusions are drawn.

Chapter 2

Background

This chapter presents, fundamentals of the testing, different fault models and testing methods for sequential and combinational circuits.

2.1 Testing Fundamentals

A test is a procedure which allows one to distinguish between good and bad parts. This can be done with a test which detects faults. [23]
A fault is present in the system when there is a physical difference between the 'good' or 'correct' system and the current system. [23]
There is an error in the system (the system is in an erroneous state) when its state differs from the state in which it should be in order to deliver the specified service. An error is caused by a fault.[23]
A system failure occurs or is present when the service of the system differs from the specified service, or the service that should have been offered. In other words: the system fails to do what it has to do. A failure is caused by an error.[23]
In order to distinguish a defect, it is crucial to model fault into a tangible parameter. Specifically in digital systems, models can manoeuvre on the binary structure of the digital design. Two of the widely used fault models are: 1)Stuck-at Faults(SAFs) 2)and Delay Faults(DFs).

2.1.1 Stuck-at Fault (SAF) Model

In structural testing, it is necessary to make sure that the interconnections in the given structure are fault-free and are able to carry both logic 0 and 1 signals. The stuck-at fault (SAF) model is directly derived from these requirements. A line is said to be stuck-at 0 (SA0) or stuck-at 1 (SA1) if the line remains fixed at a low or high voltage level, respectively (assuming positive logic).[23] For instance, in figure 2.1, in order to excite and propagate the stuck-at 1 fault in f, which is output of the AND gate and input of the NOR gate to the output which is g and make the fault observable, following considerations need to be taken:

- In order to excite the SA1 fault at f, it is necessary to assume that f has value of 0.
- To have the 0 in f, the output of the and gate with c and d as inputs, has to be 0. Thus, c or d have to be 0 and as it is an AND gate, when one input of the gate is 0, regardless of the other input, the output value is 0.
- For propagation of the fault from f to g, the value of e, must not force the output of the NOR gate. Thus, the value of e has the be 0.
- In order to have the value of the 0 at e, the output of the AND gate which, has a and b as two inputs, has to be 0. Therefore, a or b have to be 0 and as it is an AND gate, when one input of the gate is 0, regardless of the other input, the output value is 0.

Input vector values, play a crucial role in fault propagation. For example, in figure 2.2, it is depicted that, by having the 1 as input values for a and b, the output of the AND gate is 1 and the output of the NOR gate is forced to 0. Thus, no fault is propagated to the output.

2.1.2 Delay Faults

Instead of affecting the logical behavior of the circuit, a fault may affect its temporal behavior only; such faults are called delay faults (DFs). DFs adversely affect the

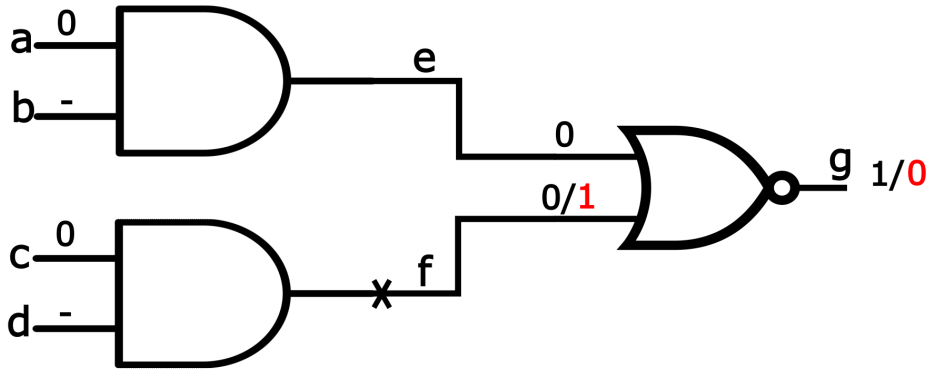

Figure 2.1: Stuck-at fault propagation example

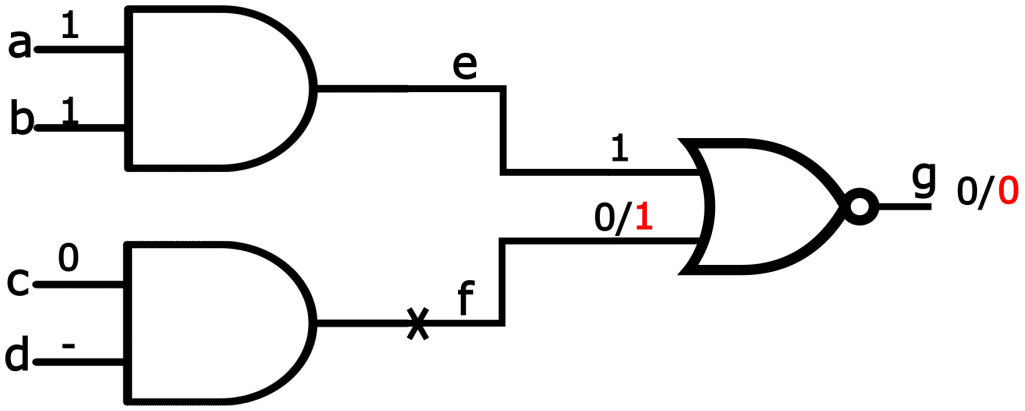

Figure 2.2: Stuck-at fault reconvergence example

propagation delays of signals in the circuit so that an incorrect logic value may be latched at the output. With the increasing emphasis on designing circuits for very high performance, DFs are gaining wide acceptance.[23] Two types of DF models are usually used: 1) Transition Delay faults (TDFs) 2)and Path Delay Faults (PDFs).

Transition Delay Faults

A circuit is said to have a Transition delay fault (TDF) in some gate if an input or output of the gate has a lumped DF manifested as a slow $0 \rightarrow 1$ (Slow-to-rise **STR**) or $1 \rightarrow 0$ transition (Slow-to-fall **STF**)[24], [25]. This fault is also called, Gate Delay Fault (GDF).
For example, following the circuit depicted in figure 2.3, STR fault is excited at input of the and gate (a). This fault propagates through the AND gate between a and b inputs. Then, propagates through NOR gate between e and f and, reaches to g which is the output of the NOR gate and circuit.
In aforementioned example, input vectors are important. Because, if they are not chosen correctly, fault may get masked and do not propagate to the output of the circuit.

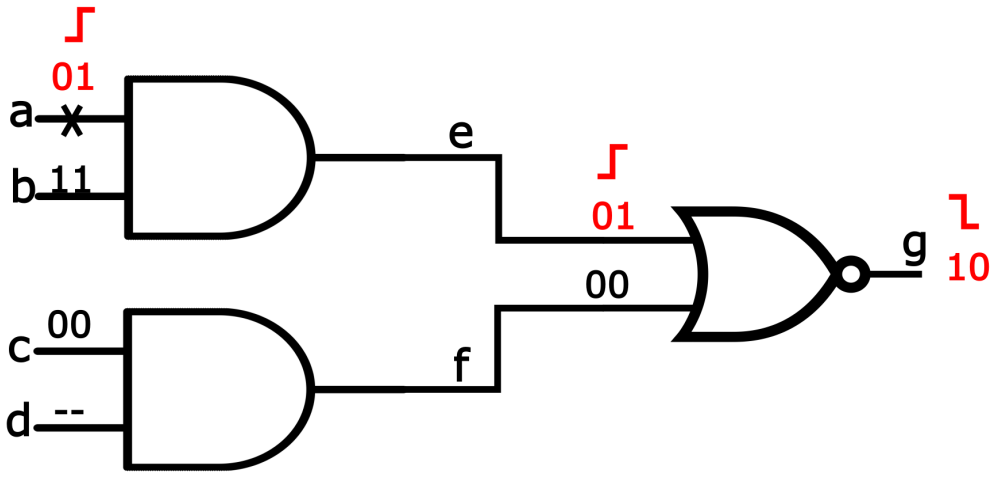

Figure 2.3: Transition Delay Fault example

Path Delay Faults

A circuit is said to have a path delay fault (PDF) if there exists a path from a primary input to a primary output in it which is slow to propagate a $0 \rightarrow 1$(Slow to Rise **STR**) or $1 \rightarrow 0$(Slow to Fall **STF**) transition from its input to its output

[26], [27].
For instance, as it is depicted in figure 2.4, the STR fault is propagated in **a-d-f** path from input to the output. In order to propagate the mentioned fault to the output, in this case, it is necessary to keep both b and c inputs high.
Clearly, the PDF model is the more general of the two models as it models the cumulative effect of the delay variations of the gates and wires along the path. However, because the number of paths in a circuit can be very large, the PDF model may require much more time for test generation and test application than the TDF model. [23]

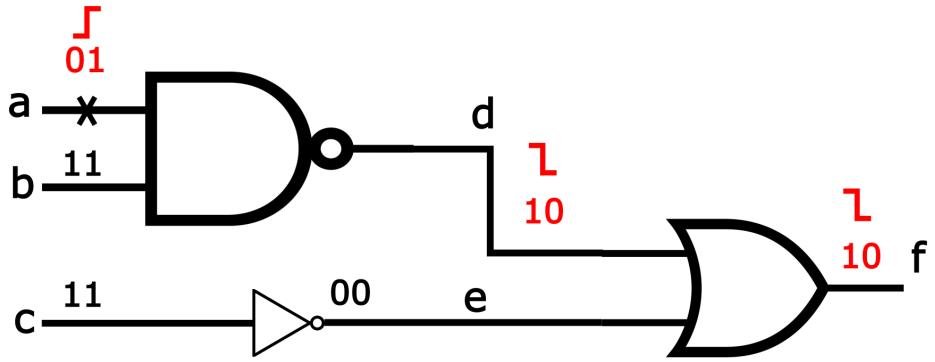

Figure 2.4: Path Delay Fault example

2.2 Automatic Test Pattern Generator

Increasing size and complexity of the digital circuits, makes the test vectors manual generation very difficult and impossible. Thus, having some automatic techniques to generate test vectors is crucial. Automatic Test Pattern Generation (**ATPG**) is the widely used technique for test vector generation. The figure 2.5, shows the architecture of the ATPG. The architecture comprises of two operative blocks which are:1) Fault manager 2) and test pattern generator. Fault manager, requires circuit description in order to generate the fault list. Test pattern generator, needs the fault list generated by fault manager and using the circuit description, it generates the test vectors and reports like **fault coverage** and **Untested fault**.

2.2.1 Fault Management

This block, generates the fault list and performs three following phases to reach to final list: 1) Untestable fault identification 2) and fault collapsing 3) and fault dominance:

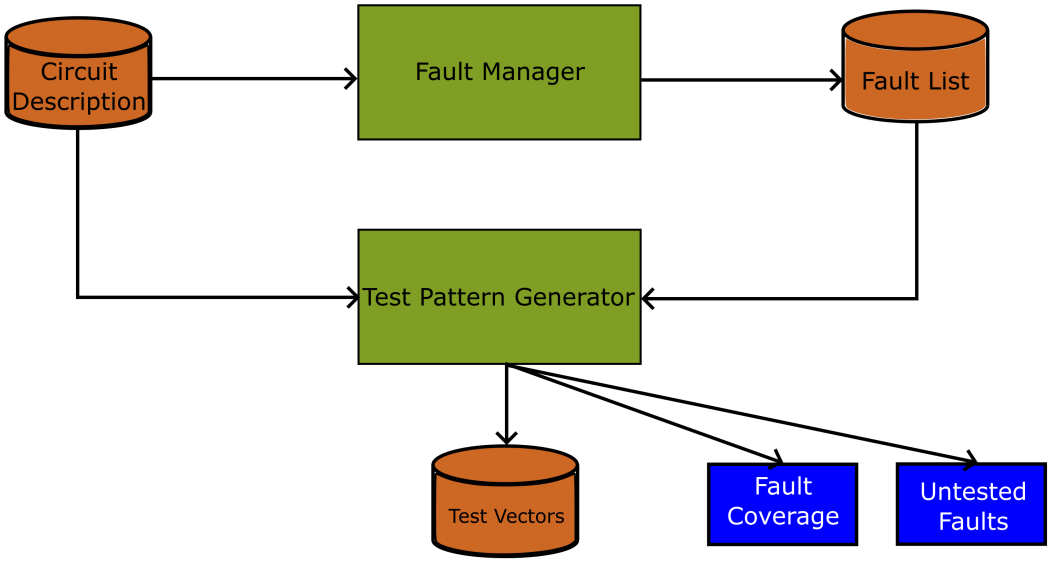

Figure 2.5: ATPG Architecture

Untestable Faults

A fault for which no test can be found is called an untestable fault. There are two classes of untestable faults [28]:

- **Functionally Untestable:** Faults that are redundant, i.e., whose presence does not change the input output behavior of the circuit.
- **Structurally Untestable:** Faults that change the input-output behavior of the circuit but no test can be found by a given method of testing or test generation. Initialization faults of sequential circuits belong to this class.

For instance, in the depicted circuit in figure 2.6, in order to propagate the SA0 fault on c, the value of the b has to be zero. However, this value lets the fault propagate to e, it causes to have 0 on d and get 0 in the output. Thus, the mentioned fault is untestable.

In a digital circuit, there are faults and as the complexity increases, the number of the faults increases too. Among these faults, there are some faults are structurally untestable and some are functionally untestable. In figure 2.7, it is shown that, the functionally untestable faults are subset of the structurally untestable faults.

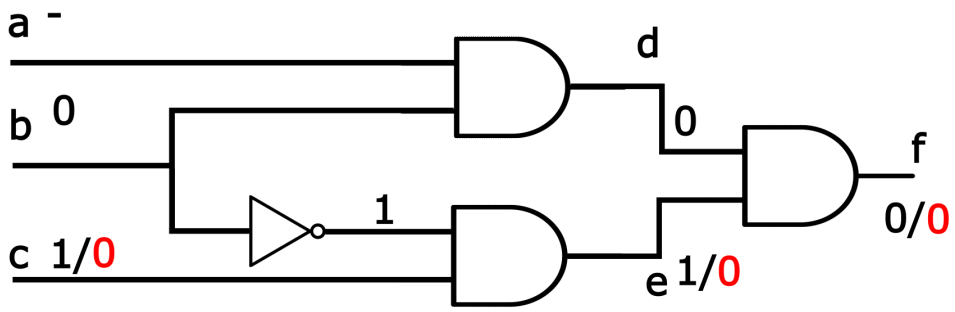

Figure 2.6: Example of Untestable fault

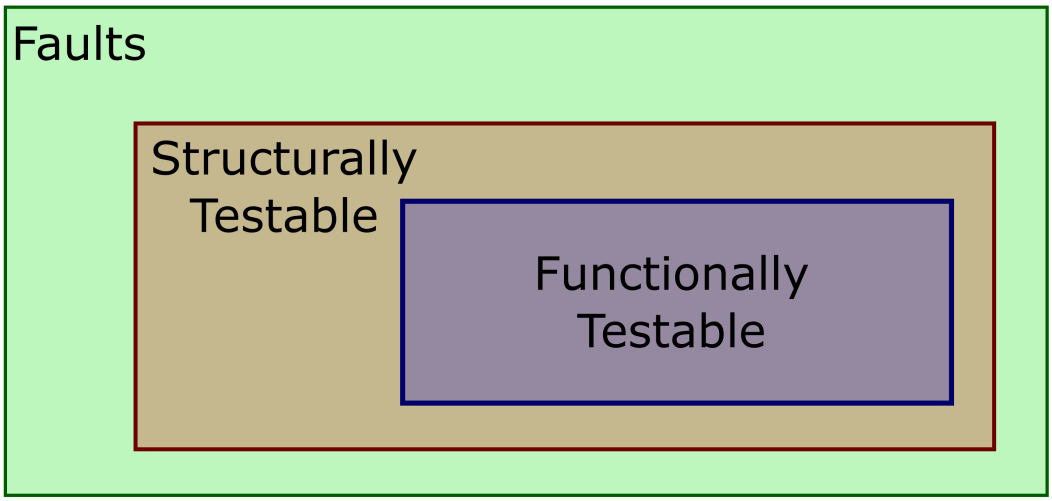

Figure 2.7: Classification of the faults based on their testability

Fault Collapsing

Collapsing is based on equivalence relationships or fault dominance. Two faults of a Boolean circuit are called equivalent iff they transform the circuit such that the two faulty circuits have identical output functions. Equivalent faults are also called indistinguishable and have exactly the same set of tests. [28] For example, in figure 2.8, in an OR gate, SA1 faults of the inputs and output of the gate is the same. Therefore, if only one of these faults get propagated to the output of the circuit, other two fault propagate to the output of the circuit too.

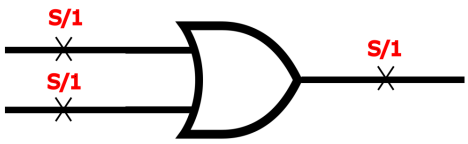

Figure 2.8: Faults equivalence example

Fault Dominance

If all tests of fault F1 detect another fault F2, then F2 is said to dominate F1. The two faults are also called "conditionally" equivalent with respect to the test set of F1. When two faults F1 and F2 dominate each other, then they are equivalent. [28]

2.2.2 Test Pattern Generator

This block is composed of: 1)An ATPG module 2)and a fault simulation module. It is the most crucial and time consuming block of the architecture. It performs following steps:

- Select a target fault among the still undetected ones.
- Launch the ATPG for generating the test set for the target fault.
- Launch the fault simulator to perform fault dropping.

These steps are depicted as a flowchart and it is presented in figure 2.9.

Target fault selection

This step, which is presented in figure 2.9, applies testability measures. Introduction to the testability is provided in section 2.2.1. The utilized technique for target fault selection, affects the cost of the whole ATPG process.

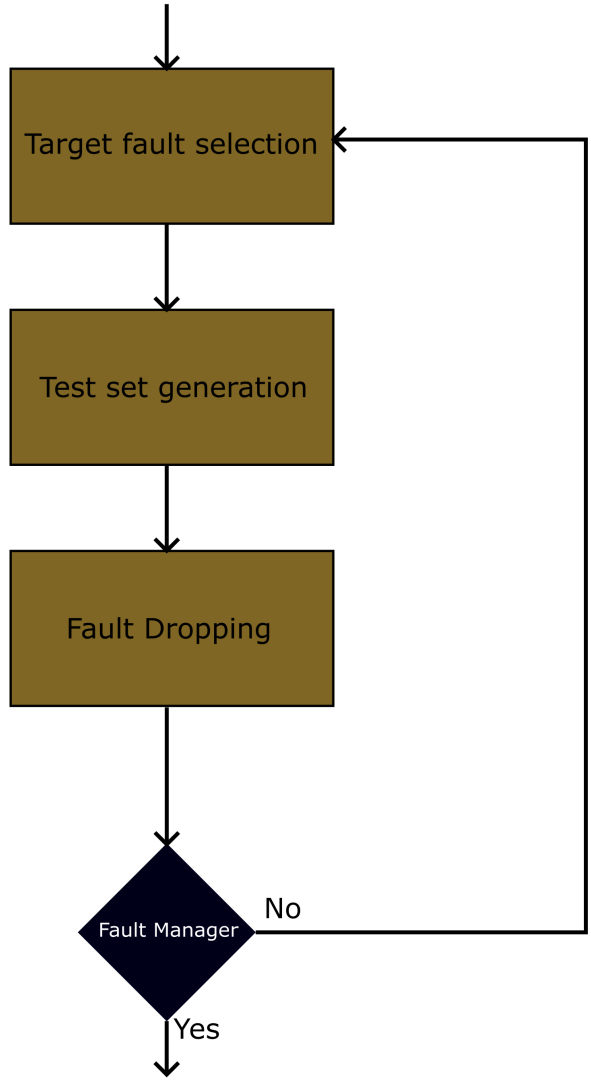

Figure 2.9: Test vector generation, flowchart

Test set generation

This step, is performed by ATPG. Based on the different fault models and type of the circuits, it generates one or more test vectors.

Fault dropping

Thanks to fault dropping, as soon as a fault is detected, it is removed from the fault list. Thus, number of the ATPG process calls and number of the faults which, are simulated each time a new vector is generated, reduces. This step, comprises of 2 sub-steps:

- Fault simulation: It applies the test vector to the circuit and checks if the fault is propagated to the output or not.
- Removal of detected faults from the untested fault list:

At the beginning of the ATPG step, faults in the fault list are labeled as untested. Test vector generation procedure finishes when, the untested fault list becomes empty or the available resources such as, CPU time, memory, etc. are exhausted. In the end of the procedure, based on the results, faults are labeled as:

- Untestable: The ATPG was able to prove the untestablity. in this case the fault is eliminated from the fault list used by the fault simulator and ATPG.
- Tested: A test vector is generated. In this case the fault is removed from the same fault list.
- Aborted: Some computational threshold is reached, without generating a test vector or proving untestability. In this case the fault is removed from the fault list for the ATPG. But, remains in the fault list of the fault simulator. So, it can be possibly detected by the test vectors generated later.

2.3 Scan Design

ATPG is powerful for combinational circuits. But, when it comes to sequential circuits, it fails. Because, observability and controllability through filp-flops require excessive number of time frames. Moreover, keeping the CPU time reasonable, demands reducing the backtrack limit, leading to aborting a huge number of faults. It causes the excessive CPU time for obtaining unsatisfactory fault coverage. The main idea in scan design is to obtain control and observability for flip-flops. This is done by adding a test mode to the circuit such that when the circuit is in this

mode, all flip-flops functionally form one or more shift registers. The inputs and outputs of these shift registers (also known as scan registers) are made into primary inputs and primary outputs. Thus, using the test mode, all flip-flops can be set to any desired states by shifting those logic states into the shift register. Similarly, the states of flip-flops are observed by shifting the contents of the scan register out. All flip-flops can be set or observed in a time (in terms of clock periods) that equals the number of flip-flops in the longest scan register. [23]

2.3.1 Full Scan

Based on the figure 2.10, a sequential circuit can be modeled as combinational logic which is connected to flip-flops.

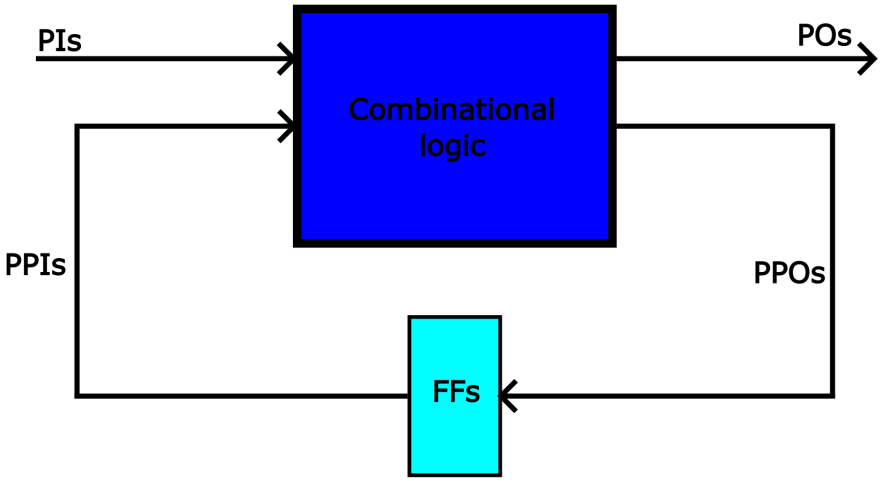

Figure 2.10: Huffman scheme for sequential circuit

After modifying flip-flops into scan flip-flops, the design becomes to a scan design as it is shown in figure 2.11.

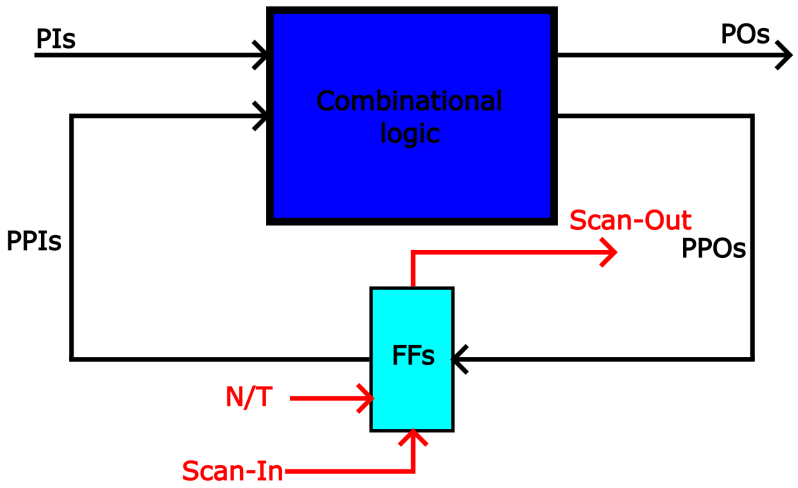

Figure 2.11: Scan circuit

Scan circuit has N/T signal which, changes the circuit mode from Normal mode to Test mode and vice versa. In test mode, as it is depicted in figure 2.12, flip-flops behave as shift register and they get connected in serial manner. However, in normal mode flip-flops behave normally and circuit has its normal behavior.

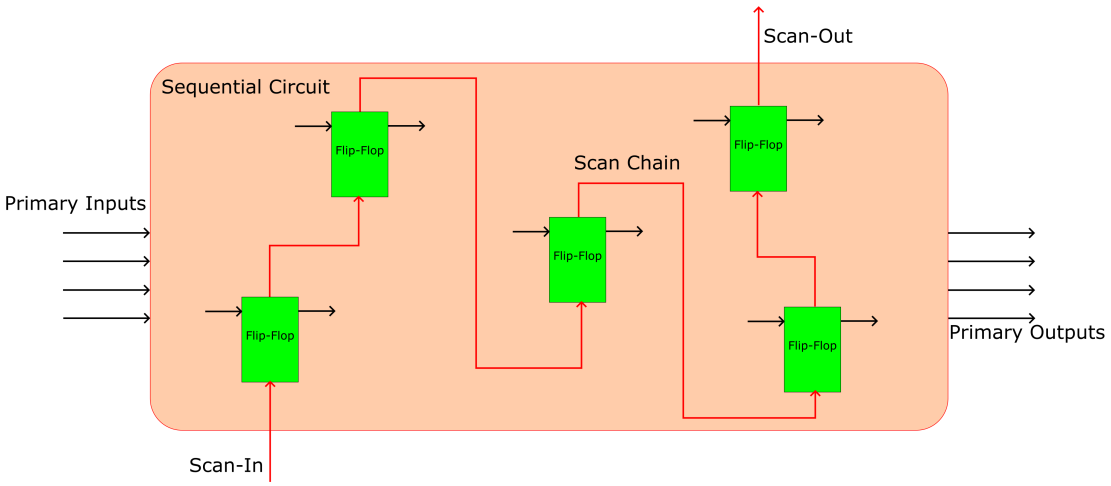

Figure 2.12: Test mode of the scan circuit

In order to apply test on the circuit for each ATPG-generated pattern following steps are taken:

- Uploading serially the PPI values to the circuit in test mode, using the Scan-In pin.
- Applying the PI values
- Changing the circuit mode to normal and clock it
- Observing the PO values
- Downloading and observing serially the PPO values from Scan-Out pin in test mode.

2.3.2 Multiple Chain

When all the FFs are in a single chain, the test time increases with the number of the FFs in the circuit. The majority of the time spent is related to scan operations. But, by using multiple scan chains, test time can be reduced. Thus, in every clock cycle it is possible to load bits concurrently as the number of the scan chains. As it is presented in figure 2.13, FFs are divided in two chains and instead of spending 5 clock cycles to load PPI values, 3 clock cycles is required.

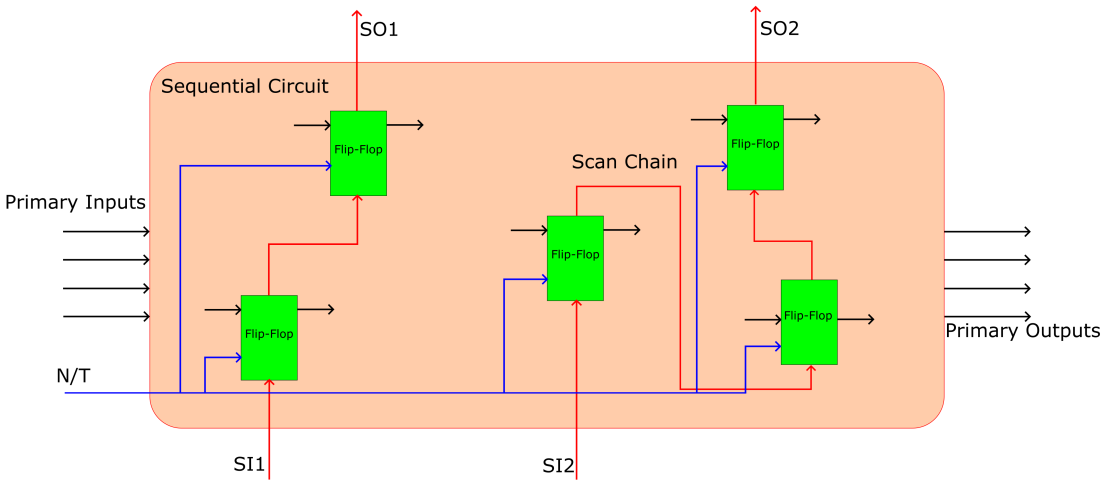

Figure 2.13: Test mode of the multiple chain scan circuit

2.4 Functional Testing

Functional test can be introduced in two different ways [29]:

- A performed test which, acts on the functional inputs and observes the functional outputs and it doesn't resort to any kind of Design for Testability (DfT).

- A developed test, only based on the functional information about the module under test. Thus, it targets testing the functions rather than the faults.(Black box testing)

The mentioned definitions above, indicate how the test is applied and how the test is generated, respectively. Nowadays, functional test is very frequently a used step in the test of the integrated circuits. Because, it covers defects which, are not covered by other kind of tests.
Functional test is used in following fields:

- End of manufacturing test: This test, is performed by Automated Test Equipment (ATE). There are limited constraints on input and output constraints. Moreover, structural information are usually available.

- Incoming inspection: However, for inspection structural data may not be available, the test is performed using an ATE and there are limited constraints on input and output signals.

- In-field (or on-line) test: For In-field test there are several constraints on input and output signals and the test cannot be performed using an ATE. In addition, structural data may not be available.

Functional test, for System on Chips (SoCs) is performed by providing a program to be executed and some data to work on to the internal processor. Then, the ATE forces the processor to execute the program on the input data. As last step, ATE observes the produced results.

2.4.1 Software-Based Self-Test(SBST)

The key idea of SBST is to exploit on-chip programmable resources to run normal programs that test the processor itself. The processor generates and applies functional-test patterns using its native instruction set, virtually eliminating the need for additional test specific hardware. (Test-specific hardware such as scan chains might exist in the chip, but SBST normally does not use this hardware.) Also, the test is applied at the processor's actual operating frequency. A typical flow for an SBST application in a microprocessor chip comprises the following three steps [30]:

- Test code and test data are downloaded into processor memory (i.e., either the on-chip cache or the system memory). A low-cost external tester can perform test code and data loading via a memory load interface running at low speed.
- The processor executes test programs at its own actual speed. These test programs act as test patterns by applying the appropriate native instructions to excite faults. The test code loads two patterns and adds them to excite a fault in the adder module. Finally, the test code stores the test responses back in the processor data memory to propagate the faults. When the test process is supported only on the on-chip cache, the self-test program must be developed so that no cache misses occur during SBST execution. Hence, this is sometimes called cache-resident testing.
- Test responses are uploaded into the tester memory for external evaluation.

The renewed interest in SBST during the past decade was primarily motivated by the existence of two opposite trends: the increasing cost of functional testers was impelling vendors toward structural test techniques such as scan, while the doubts about the effectiveness of the structural-test patterns and the significant yield loss due to overtesting was moving them toward functional testing. Consequently, the emerging approach of SBST (in which normal, functional-test programs use low-cost structural testers to access the microprocessor chip) gained ground as a way to improve processor testing by combining the benefits of the functional- and structural-testing worlds. The question is whether a test program running on

the processor can adequately test its modules by satisfying the industry-standard high-fault-coverage requirements. Achieving this test-quality target requires a composite test-program-generation phase, which is the main subject of most SBST approaches described in the literature during the past decade [30]. The other SBST features that confirmed its role in the microprocessor test flow include the following:

- It is non intrusive. SBST does not need any extra hardware, which sometimes could be unacceptably expensive for carefully optimized circuits such as microprocessors. Moreover, it does not consume any extra power compared to normal operation mode.

- It allows at-speed testing. Test application and response collection are performed at the processor's actual speed, enabling screening of delay defects that are not detectable at lower frequencies.

- It avoids overtesting. SBST avoids test overkill and, thus, detection of defects that will never be manifested during normal processor operation. This leads to significant yield gains.

- It can be applied in the field. Self-test programs from manufacturing test can be reused in the field throughout the product lifetime. For example, for power-up diagnostics to add dependability features to the chip.

2.5 Related Work

The development of test programs for delay faults through a SBST approach has been faced by some works describing methodologies to do so[7, 9, 11]. Regarding TDF specifically, [22] introduces a study on transition delay faults of modern pipelined CPUs that have not been observed throughout the execution of STLs targeting SAFs, focusing on where their effects propagated and stopped inside the DUT.[4] The article, introduces two fault groups:

- **User Accessible Registers *(UARs)*:** Effects reached registers that can be directly observed through instructions from the CPU's instruction set architecture, e.g., the register file [4].

- **Hidden Registers *(HRs)*:** Register which, cannot be accessed directly through available instructions.

The work in [22] provides some useful insights on which faults to target in order to improve the final fault coverage. Moreover, it gives an upper limit on the final transition delay fault coverage improvement.
The article, proposes a systematic methodology that, given test programs already developed for SAFs, allows to pinpoint code regions within the STL to be modified to increase transition delay fault coverage in complex pipelined processor cores.
Given three test programs, namely STL1, STL2 and STL3, [22] presents that it is possible to increase their fault coverage by, relatively, 9.15, 17.85 and 8.96 percentile units. This work, however does not provide strategies to detect them, which is the goal of this thesis[4].
The work [13] describes STL development strategies for peripherals embedded in modern System on Chips, achieving significant fault coverage figures. Figure 2.14, depicts the flow designed and followed for test generation in [13].

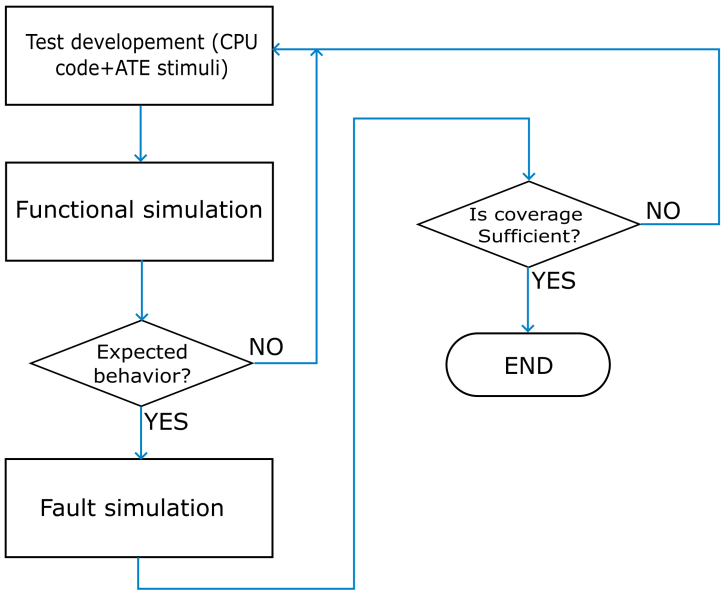

Figure 2.14: Test Generation Flow

Works [5, 6], on the other hand, focus on the development of test programs for

processor cores. [5] describes a methodology to test delay faults on computational blocks within superscalar processors, while [6] aims at testing RISC-like CPUs by dividing them into modules under test and devising test strategies for each of these modules without the need of knowing implementation details[4].
Finally, [31] presents a reinforcement learning-based test program generation technique for TDFs validated on a MIPS32 core. Although effective, showing that it is possible to thoroughly test delay faults through functional means, all these works require the generation of test programs starting from scratch, a task that requires non-negligible time and effort from test engineers. [31], moreover, requires the usage of a reinforcement learning algorithm, which might not be particularly effective when tackling high-complexity cores[4].
Articles like [32, 33, 34] focus on the improvement of available programs to reach high fault coverage values. [32] describes how to derive test patterns intended for online testing starting from programs originally intended for verification purposes, significantly increasing the final coverage of stuck-at faults on a RISCV core. Works [33, 34], on the other hand, present a tool based on High-Level Decision Diagrams (HLDDs) for modeling microprocessors and faults, used in conjunction with previously prepared code templates to generate the final self-test program targeting stuck-at faults.
These works show that methodologies for improving test programs can be successfully devised. Nonetheless, they are developed bearing the classical stuck-at fault model in mind.
The goal of this thesis is to propose some techniques allowing to automate the transformation of existing STLs targeting SAFs so that the resulting TDF coverage is improved[4].

2.6 Commercial Tool Features

In this thesis work, in order to perform simulations and experiments and derive fault information needed to apply algorithms and methodologies defined in section 3, commercial tools such as ZO1X[35], ModelSim[36] and Design Compiler [37] are utilized. Given the introduction in section 2.2.2, for circuit description, three files are required by ZO1X tool:

- Digital circuit description in hardware description language(HDL): Digital circuit design can be presented in any HDLs and the circuit is usually defined in VHDL or Verilog languages. The ZO1X using the circuit design, generates the fault list.
- Value Change Dump (VCD) file: This file, is standard for EDA tools and it is generated by simulating the circuit using the ModelSim. This file contains simulation values of the circuit. It comprises of 5 main sections:
  - Header section with date, simulator and timescale information.
  - Variable definition section.
  - Value change section.

  In listing provided in 2.1, there is a simple example of the VCD file.

Listing 2.1: Basic example of VCD file

```
$date
   Date text. For example: September 11, 2022.
$end
$version
   VCD generator tool version info text.
$end
$comment
   Any comment text.
$end
$timescale 1ps $end
$scope module logic $end
$var wire 8 # data $end
$var wire 1 $ data_valid $end
$var wire 1 ) underrun $end
$upscope $end
$enddefinitions $end
$dumpvars
bxxxxxxxx #
#267
0'
#476
b0 #
1$
#1245
0$
#1893

```

- Strobe file: This file, is used to define observation points in the circuit and based on the locations in the processor where, faults propagate to, faults get labeled. For instance, all faults can be labeled as not detected at the beginning and when, faults propagate to UAR or HR locations, get labeled as HR or UAR accordingly. In the end, if they reach to primary outputs they get labeled as detected. Surely, if a fault propagates to the primary outputs, there is no need to keep the previous label of the fault. This file, is written in systemverilog hardware description language.

The output of the ZO1X tool is the fault dictionary which, reports the label of the fault and time of the observation, the exact location of the fault and the type of the fault (STR or STF). In figure 2.15, one example of the fault dictionary is provided.

```
["strobe2", 438558000ps,
    1       GM: 111 000049e0 00105f61 00ffff00 e
            FM: ... ........ ........ ....00.. . ]
    DD F {TRAN "riscv_core.ex_stage_i_alu_i.U1716.A"}
    -- F {TRAN "riscv_core.ex_stage_i_alu_i.U1715.ZN"}
    DD R {TRAN "riscv_core.ex_stage_i_alu_i.U1715.B2"}
```

Figure 2.15: ZO1X fault dictionary example.

2.7 PULPino

The applied methodologies in this thesis work, are performed, tested and validated on the PULPino architecture. PULPino is an open-source single-core microcontroller system, based on 32-bit RISC-V cores developed at ETH Zurich. PULPino is configurable to use either the RISCY or the zero-riscy core[38]. For the work of this thesis, the RISCY configuration of the PULPino is used. This core has following specifications which are mentioned in [38]:

- In-order execution
- Single issue core
- 4 pipeline stages
- Full support for following instruction sets:
  - base integer instruction set (RV32I)
  - compressed instructions (RV32C)
  - multiplication instruction set extension (RV32M)
- Configurable to have single-precision floating-point instruction set extension (RV32F)

It is able to implement several Instruction Set Architecture (ISA) extensions such as:

- Hardware loops
- Post-incrementing load and store instructions
- bit-manipulation instructions
- MAC operations
- Support fixed-point operations
- Packed-SIMD (Single Instruction stream, Multiple Data stream) instructions
- Dot product

RISCY has been designed to increase the energy efficiency of in ultra-low-power signal processing applications. It also, implements a subset of the 1.9 privileged specification.
On the other side, zero-riscy is an in-order, single-issue core with 2 pipeline stages and it has full support for 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)[38].
For communication with the outside world, PULPino contains a broad set of peripherals, including I2S, I2C, SPI and UART. The platform internal devices can be accessed from outside via JTAG and SPI which allows pre-loading RAMs with executable code. In standalone mode, the platform boots from an internal boot ROM and loads its program from an external SPI flash[38].
Pulpino RISCY core design is depicted in the figure 2.16.

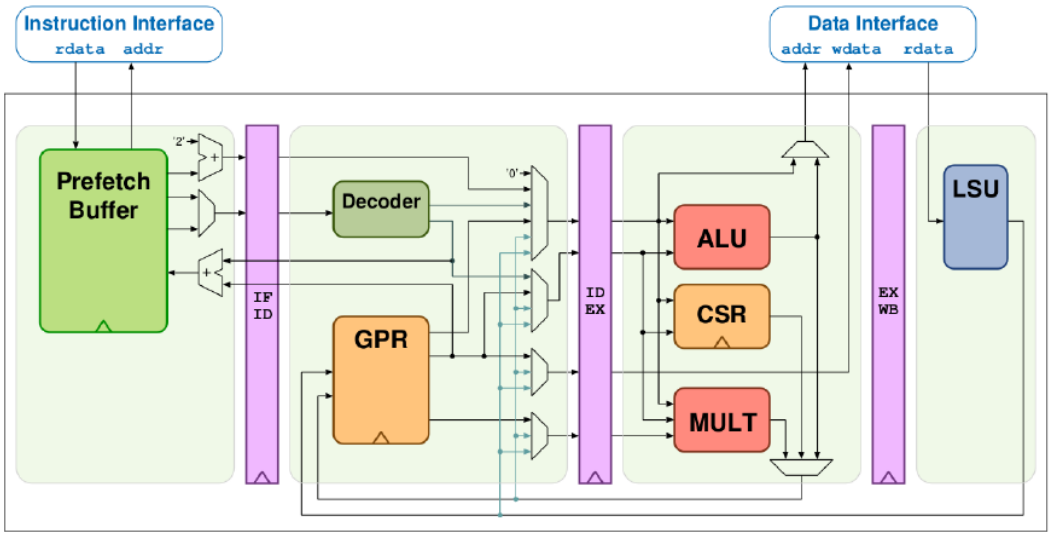

Figure 2.16: PULPino RISCY core[39]

Chapter 3

Methodology

3.1 Introduction

This thesis work mainly stems from an empirical observation: STLs targeting stuck-at faults often fail in achieving a high TDF coverage because of their inability to propagate TDF effects up to some observable point.
By improving the ability of an STL to propagate TDF effects we could significantly increase its TDF coverage. Detecting transition delay faults that were excited but not observed by STLs requires some considerations on where their effects propagated and stopped within the DUT.
The reason for doing so lies in the fact that, in order to detect the aforementioned faults, we need to propagate their effects towards observable points (e.g., memory locations) and strategies for such task may vary based on the functional sub-module from which the propagation occurs[4].

3.2 Description

Given the introduction, in this work the beginning points are two main categories of internal observation points are introduced in [22], i.e., User Accessible Registers (UARs) and Hidden Registers (HRs), and define some internal observation points to be used during simulation to further refine the topological analysis about fault effects. Inserting observation points is done by exploiting capabilities offered

by available commercial fault simulation tools. Such task does not require the modification of the DUT's hardware as they are simply labels that the tool attaches to the netlist. To each label a fault status is associated, and the hierarchy of the statuses can be customized to make sure that locations closer to the primary outputs of the architecture are prioritized. As a further step, this thesis work describes how to introduce suitable instructions in the STL code, so that for each group of faults, a significant percentage of them is made observable, and hence detected[4]. Moreover, this work describes how to extract the information required to extract from fault dictionaries of the commercial tool, in order to extract aforementioned instructions in the STL code. In the following, we discuss strategies for the aforementioned two main fault categories.

3.2.1 User Accessible Register fault

User Accessible Register faults, are faults whose effects have been propagated from the original fault site to registers that, can be directly observed 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. In order to detect such faults, first the fault data base produced at the end of the fault simulation where internal observation points have been added, has to be analyzed. Such data base stores information on which faults have been observed at any given internal observation point at some specific time instant[4].

This helps to obtain topological information, i.e., what register has to be worked on, as well as chronological information, i.e., what portion of the test program has to be improved. The latter is possible since we are able to associate instructions being executed by the CPU to the simulation time reported in the dictionary.

When analyzing the time at which fault effects reached user accessible registers, it is crucial to keep in mind that, in modern in-order pipelined processor cores, there are instructions that take more than one clock cycle to go through the CPU execution stage, e.g., division operations.

These instructions — from hereinafter referred to as multi-cycle instructions, as opposed to single-cycle instructions that only take one cycle in figure 3.2. Multi-cycle instructions and fault effects propagation the execution stage — should be

carefully taken into account, as the fault effect might be overwritten during the required execution cycles.

For instance, given an instruction that takes 4 clock cycles to go through the execution stage, a situation similar to the one reported in figure 3.2, may occur, where the fault effect is propagated at an inner cycle only to be overwritten later, losing the possibility of observing the faulty value. For this reason, two strategies depending on whether single-cycle or multi-cycle instructions are dealt with, are introduced:

- Single-cycle instructions: Detecting these faults is quite easy, as it is sufficient to perform a store operation on the register affected by the faulty value after the time at which said effect reached the register, and before the register is overwritten by another operation[4].

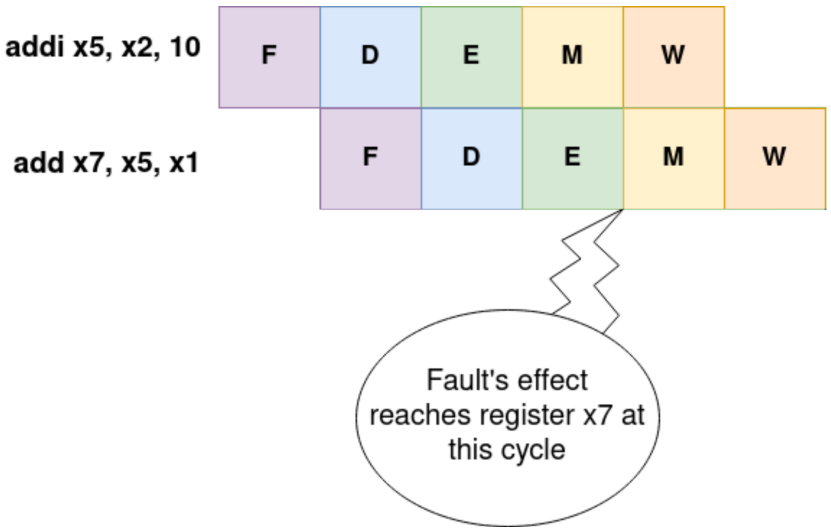

Figure 3.1: Single-cycle instructions and fault effects propagation

- Multi-cycle instructions: If the fault effect is still present at the last execution cycle the same strategy adopted for single-cycle instructions is used, else the operands of the multi-cycle instructions are modified — either arithmetical or logical operations — 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. As it is demonstrated by the gathered experimental results,

identifying suitable operands for this purpose is a feasible task, which can often performed following a try-and-error approach[4].

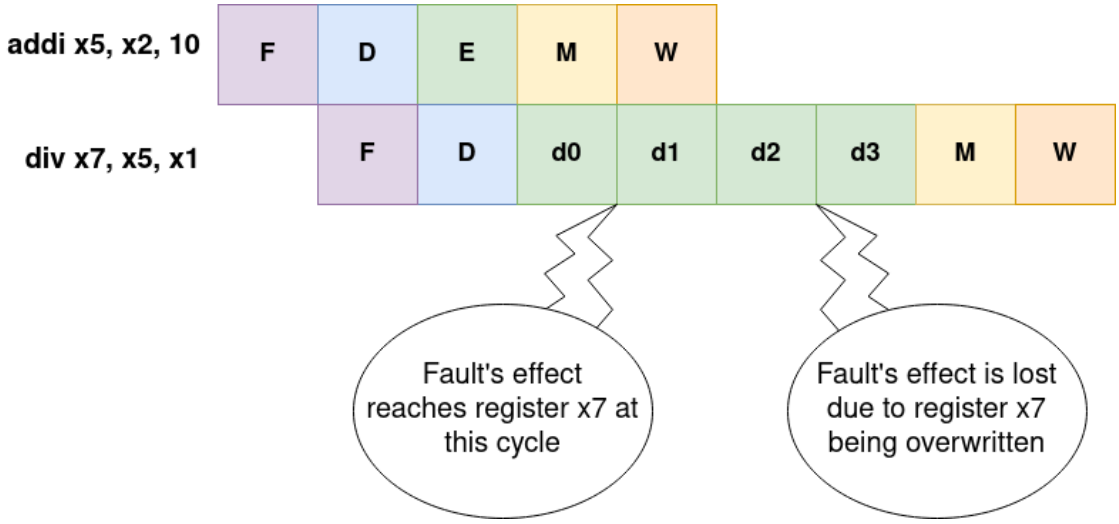

Figure 3.2: Multi-cycle instructions and fault effects propagation

3.2.2 Hidden Register faults

All those faults whose effects reach registers that cannot 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 (in this case the techniques described in the previous sub-section can then be adopted).

The proposed strategy to detect these faults, starts off similarly to Section 3.2.1, that is, by analyzing the fault data base 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, additional analysis are 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-processes .

First, a specific pair of test vectors must be applied to generate the required

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
```

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 work is to define an automatic way to easily increase the transition delay fault coverage for STLs that were previously devised for stuck-at faults[4].

Given this group of faults, hence, we define the algorithm summarized in Algorithm 1.

The basic idea behind this algorithm is that the available STLs may already be able to test stuck-at-0 and stuck-at-1 faults located in pipeline registers. The code that serves that purpose, however, can also be used to propagate values from said 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. In the rare case when the TDF propagated to a bit in a pipeline register, whose corresponding SAFs were still not detected by the existing STL, methods such as [40] can be used to generate the required chunk of instructions (improving the SAF coverage as well). A final point that applies to both User Accessible Register and Hidden Register faults is that, given the right premises, a set of instructions added to the original test program may be capable of detecting more than one TDF at the same time. This is only possible for all those transition delay faults whose effects propagate to the same register at the same time. Thanks to this feature, it is possible to achieve better fault coverages with a smaller test program with respect to having a set of instructions for each fault to be tested.

3.3 Analyzing Fault Dictionaries

As mentioned in the section 2.6, one of the output files of the commercial tool (ZO1X), is the ***Fault Dictionary***. This file, provides crucial information about each fault in the digital circuit. These information are: 1)Fault location 2)Fault observation location which, extracted using strobe and the change in **Faulty** values position, 3)fault detection time, 4)label of the fault 5)and type of the fault (STR or STF).

In order to apply algorithms proposed in section 3.2, it is necessary to parse this dictionary and extract all required information.

As it is described in section 3.2, the main idea of this methodology is to detect transition delay faults for the cases in which, as depicted in figure 3.3, a transition delay fault propagates to an observation point. In this case, observation points are UARs and HRs therefore, during parsing the fault dictionary, information of faults which are propagated to target observation points (HRs and UARs), need to be extracted. As the next step, SAFs of target observation points which are propagated to the primary outputs of the digital circuit, need to be extracted and using the detection time, it is possible to find instructions which, possible propagated the fault to the primary outputs.

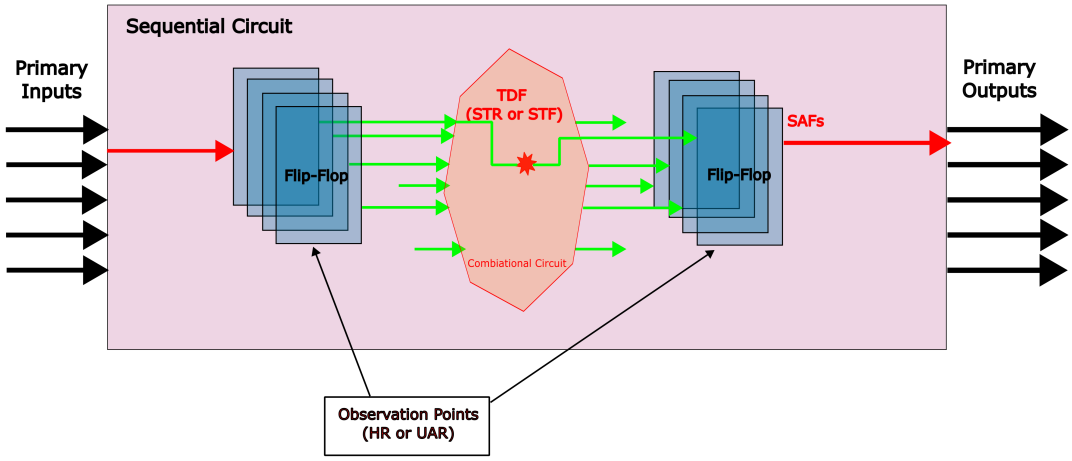

Figure 3.3: Transition Delay Faults propagation to the observation points

3.3.1 Parsing fault dictionary

In order to parse the fault dictionary, and apply the previously mentioned algorithms in section 3.2, following steps need to be taken:

- First step of parsing fault dictionary, generated for TDFs is to extract the strobe number of each label. These strobe numbers is given to the tool which, is written in python, as a dictionary and for each label. For instance, as it is shown in figure 3.4, in this fault dictionary the strobe3 is allocated to the Special Registers (SR). Both are shown in green in figure 3.4, also R or F shown in blue indicates the slow-to-rise or slow-to-fall type of the TDF. The fault locations are shown in gray.

```
["strobe2", 718000ps,
    1       GM: 100101 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
            FM: 01.... . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
    54      GM: 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
            FM: . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
    105     GM: 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1 0 1 001 0 1 0 1 0 1 0 1
            FM: . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
    157     GM: 0 1 0 1 0 1 0 1 0 1 0 1 0 10101 0 1 0 1
            FM: . . . . . . . . . . . . . . . . ..... . . . .]
    SR R {TRAN "riscv_core.cs_registers_i.id_valid_q_reg.CK"}
    SR F {TRAN "riscv_core.cs_registers_i.id_valid_q_reg.CK"}
    SR R {TRAN "riscv_core.cs_registers_i.id_valid_q_reg.RN"}
```

Figure 3.4: Fault dictionary analysis

- After knowing the fault label, type, detection time and location, it is necessary

to extract also data related to the fault observation location. The exact location of the observation point can be extracted from the matrix of *Faulty Machine*(**FM**) and *Good Machine*(**GM**) which, is provided above the fault locations in figure 3.4. The observation locations are the locations in the faulty machine which are different from the good machine. The locations remained fault free are indicated in dots and the rest are the faulty ones which are marked in red in the figure 3.4.

For instance, as depicted in figure 3.5, the fault from location in grey is observed in 19 and 20 locations of the strobe8 at time in orange color. This fault is labeled as Load Store (LS) and based on the location, they have to be on pins 19 and 20. In strobe pins provided in fault dictionary figure 3.6, the strobe8 list contains 76 pins. However, in the FM and GM matrix, there are 80 elements and these numbers don't add up.

The main reason behind this problem, is the fact that the strobe list provided by tool, contains only connected pins while, there are pins in the design, which are not connected.

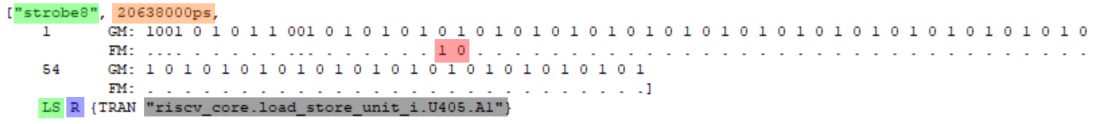

Figure 3.5: Fault dictionary analysis for finding observation location.

```
StrobeList{"strobe8"
    Location{"$fs_set_status, ./bin/strobe_v3_2impro.sv : line 110"}
    Pins{
        1 "riscv_core.load_store_unit_i.data_we_q";
        2 "riscv_core.load_store_unit_i.data_we_q_reg.QN";
        3 "riscv_core.load_store_unit_i.data_sign_ext_q";
        4 "riscv_core.load_store_unit_i.data_sign_ext_q_reg.QN";
        5 "riscv_core.load_store_unit_i.rdata_q[15]";
        6 "riscv_core.load_store_unit_i.rdata_q_reg_15_.QN";
        7 "riscv_core.load_store_unit_i.rdata_q[7]";
                                                    ----
        74 "riscv_core.load_store_unit_i.rdata_q[1]";
        75 "riscv_core.load_store_unit_i.n340";
        76 "riscv_core.load_store_unit_i.n339";
    }
}
```

Figure 3.6: Strobe List example.

The solution proposed in this work is to ignore this strobe list and use the components locations written in strobe file and retrieve all pins using the Design Compiler tool. In this way, all pins locations can be found and it is possible to use these observation locations for the next step. The tcl code written to retrieve these locations is provided in script 3.1.

Listing 3.1: TCL script for retrieving the strobe pins

```
# read ports from file extracted from strobe file
set input_file [open "./../ports_list.txt" r]
set file_data [read $input_file]
set data [split $file_data "\n"]
set net [list]
# First, initialize required variables and set libraries
set search_path [list ./../../asic/synopsys/bin ./../../asic/
    techlib/ [getenv "SYNOPSYS"]]
set synthetic_library dw_foundation.sldb
set target_library NangateOpenCellLibrary_fast.db
set link_library [list $target_library $synthetic_library]

# Next, read synthesized core and elaborate it
analyze -f verilog ./../../gate/riscv_core.gate.v
elaborate riscv_core
set output_file [open "./../nets_list.txt" w]
```

```
set n 0
foreach line $data {
# Now we have to associate path endpoints to the relative
    flipflop output and put NC (Not connected) if it is not
    connected
if {$line == ""} {break}
incr n 1
set pin [get_object_name [get_pins -filter {@pin_direction == out
    } -of_object $line]]
set pin_Q [lindex $pin 0]
set Q [get_object_name [get_net -of_object $pin_Q]]
if {$Q == ""} {set str_out1 "$n- NC"}
set pin_QN [lindex $pin 1]
set QN [get_object_name [get_net -of_object $pin_QN]]
if {$Q == ""} {set str_out1 "$n- NC"} else {
set str_out11 [string map {/ .} $Q]
set str_out1 "$n- riscv_core.$str_out11"}
puts -nonewline $output_file "$str_out1"
puts -nonewline $output_file "\n"

incr n 1
if {$QN == ""} {set str_out2 "$n- NC"} else {
set str_out21 [string map {/ .} $QN]
set str_out2 "$n- riscv_core.$str_out21" }
#Write the output data in the output file
puts -nonewline $output_file "$str_out2"
puts -nonewline $output_file "\n"
}

close $input_file
close $output_file

exit
```

- Strobe file: This file, is used to define observation points in the circuit and based on the locations in the processor where, faults propagate to, faults get labeled. For instance, all faults can be labeled as not detected at the beginning and when, faults propagate to UAR or HR locations, get labeled as

HR or UAR accordingly. In the end, if they reach to primary outputs they get labeled as detected. Surely, if a fault propagates to the primary outputs, there is no need to keep the previous label of the fault. This file, is written in systemverilog hardware description language.

- In this step, given the fault observation locations, fault dictionary for stuck-at faults are searched in the fault dictionary. If the SA0 or SA1 faults related to mentioned locations are detected, the observation time is extracted. In the end of this step, the outcome is a dictionary depicted in figure 3.7.

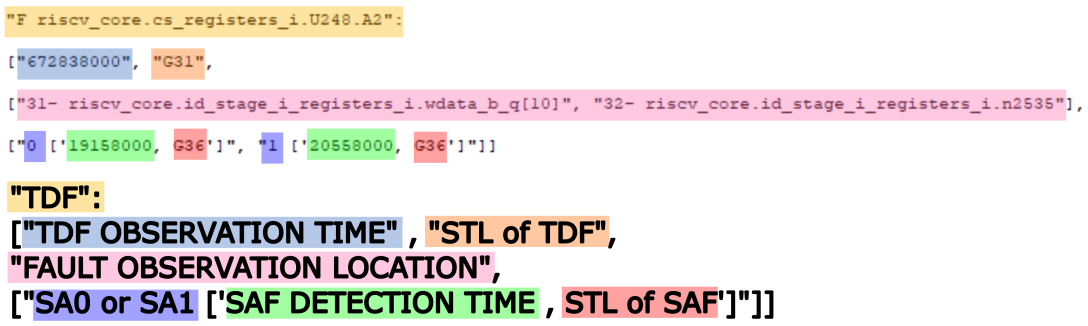

Figure 3.7: Dictionary made by the written tool and its format

In figure 3.7, the format of the output dictionary is shown below the example of the output and each highlighted part of the example, corresponds to the same color in the format. For instance, the TDF is highlighted in orange.

- In this step, after having the observation times for SAF and TDF, it is possible to track the instructions in the STLs and choose possible blocks for fault coverage improvement and insert them in the part STL which, the TDF is observed in. In order to check the trace of STL execution, the tool provided in [41], is used. In this tool, there is a database comprised of execution tracer(Example in 3.2) and disassembly files(Example in 3.3) of the STLs. By merging data of two files, tool receives the observation times and provides the instruction blocks which possibly detects the SAFs and provides the location in STLs which these produced instruction blocks can be inserted. The example of the output dictionary is provided in script 3.4.

Listing 3.2: Example of execution tracer [41]

```
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
19040000  459    000068bc 00008113  addi    x2 , x1 , 0
19080000  460    000068c0 00008193  addi    x3 , x1 , 0
19120000  461    000068c4 00008213  addi    x4 , x1 , 0
19160000  462    000068c8 00008293  addi    x5 , x1 , 0
19200000  463    000068cc 00008313  addi    x6 , x1 , 0
19240000  464    000068d0 00008393  addi    x7 , x1 , 0
```

Listing 3.3: Example of Disassembly file [41]

```
...
/../ simple.S:77
      190:    f3f290e3        bne t0,t6,b0<fail>
/../ simple.S:78
      194:    fff00293        li t0,-1
/../ simple.S:79
      198:    f1f31ce3        bne t1,t6,b0<fail>
/../ simple.S:80
      19c:    fff00313        li t1 ,-1
/../ simple.S:81
      1a0:    f1f398e3        bne t2,t6,b0 <fail>
...
```

Listing 3.4: Example of output dictionary of tracer tool.

```
"8": {"faults":
 ["R riscv_core.ex_stage_i_alu_i.int_div_div_i.U329.B1",
 "F riscv_core.ex_stage_i_alu_i.int_div_div_i.U294.A1",
 "F riscv_core.ex_stage_i_alu_i.int_div_div_i.U327.B2"],
 "saf_blocks": ["li      x31,     0x106a9e
                 sw      x14,     0(x31)
                 li      x18,     0xcc8109c3
                 li      x19,     0x3579b1b0
                 remu    x22,     x18,     x19",
                "li      x31,     0x102036
                 sw      x8,      0(x31)
                 li      x23,     0xadc7eee3
                 li      x1,      0xfdf66728
                 rem     x22,     x23,     x1"],
"destination": {"tdf_program": "G31", "source_tdf_file": "simple.
   S", "start_tdf_line": 439, "end_tdf_line": 441}},
```

In script 3.4, it is shown that the format of the output dictionary is as following:

- Key: It is used to count the number of the destinations in the STLs.
- faults: It contains the TDFs.
- saf_blocks: It comprises of the blocks of the STLs which, may be able to detect the TDF faults mentioned above.

– destination: It contains details about the line range of specific file in specific STL, which saf_blocks can be inserted in to detect the TDF faults. It is possible that, by inserting some of these saf_blocks in the code, several TDFs get detected. Therefore, all TDFs are gathered to optimize the number of the simulations.

The steps provided above are used to make a final dictionary in order to perform simulations for HRs. However, for UARs the steps are more simple. Instead of finding saf_blocks, it is only required to have a store instruction in the destination. In the strobe related to the UARs, the bit of the related register can be found. In the end it is only a matter of generating an instruction in format of **sw *Faulty register*, 0(sp)**.

Chapter 4

Experimental Results

4.1 Case Study and Setup of the Experiments

This chapter, contains information about the case study, setup of the experiments of the thesis work and results of the applied algorithms.

4.1.1 Case Study

The methodology introduced in this thesis work has been validated on PULPino[38], a 32-bit RISC-V-based SoC platform developed by ETH Zurich and Università di Bologna.

As this approach focuses on CPUs, this SoC has been configured to solely include the RI5CY core, an in-order, single-issue core with 4 pipeline stages; peripherals and other boundary components, on the other hand, have been left out. Details about PULPino are mentioned in 2.7. The DUT has been synthesized using the 45nm Silvaco Open Cell library[42] and accounts for 51,001 NAND2-equivalent gates, 159,326 stuck-at faults (SAFs) and transition delay faults (TDFs), and 1,207 flip-flops belonging to hidden registers[4].

As for the test programs, three different STLs that were originally specialized in order to test SAFs on the PULPino core, namely STL1, STL2, and STL3, are adopted. In order to ensure a diverse and realistic testbench, the three selected test programs have been developed following different implementation strategies, by different test engineers. A summary of the most important characteristics of

the adopted STLs, namely the execution time (expressed in the total amount of clock cycles), memory size (in kB), and SAF coverage, is reported in 4.1.

Table 4.1: 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

It is noted that the reported amount of clock cycles is obtained by executing all STLs completely; depending on the situation, the test engineer can then decide to split them into sub-modules that can be launched separately, each requiring a fraction of the overall time with the same final fault coverage. Fault simulations have been carried out using Synopsys Z01X, a commercial tool devised specifically for functional safety.(Details in 2.6) Experiments have been conducted by means of Python scripts, with the goals of collecting information from fault dictionaries(for more details refer to 3.3), improving test programs according to the methodologies described in 3, and launching the actual fault simulations. As a result, the full flow of STL improvement and fault simulation for transition delay faults took no longer than 4 days on an Intel Xeon CPU E5-2680 v3 server with a clock frequency up to 3.3GHz.

4.1.2 Setup of the Experiments:

In order to perform fault simulations, it is necessary to design a tool chain to perform simulations in an automatized approach to optimize the time required for performing fault simulations. The designed tool chain comprises of following steps:

- **Step1:** As first step, the output dictionary generated in 3.4 is taken and TDFs are inserted to the fault list of the fault simulator. The python function defined in 4.1, is used for this reason. It reads the TDFs and modify them in order to adapt to the format used in the tool.

Listing 4.1: Function for inserting TDFs.

```
def insert_faults(list_faults):
    with open('./zoix_SAF_TRF/bin/user_pipe_tmp.sff') as fd :
        new_lines= fd.readlines()
    for fault in list_faults:
        line = "NA " + fault.split(" ")[0] + ' { PORT "' + fault.
    split(" ")[1] + '" }'
        new_lines.append(line+'\n')
    new_lines.append("}")
    output_file = open('./zoix_SAF_TRF/bin/user_pipe.sff',"w")
    output_file.writelines(new_lines)
    output_file.close()
```

- **Step2:** In the next step, after inserting of the TDFs, code blocks used for detecting the SAFs, are inserted in the destination range. Based on the performed manual experiments, this range is chosen as 5 lines. For instance, if the starting line in the STL is line 200, the code block can be inserted in line range of 200 to 205.

- **Step3:** In this step, after inserting the code blocks, the fault simulation is performed. In order to perform the fault simulation, first a tool chain is executed to generate the VCD file required by ZO1X. In order to generate the VCD file it is necessary to run simulation in the ModelSim. However, during insertation of the code blocks, there is the possibility of the infinte loop generation in the modified STL. Hence, to avoid this problem a time threshold based on the performed manual experiments is chosen which, considers the worst case of execution time and in case, simulation continues more than this period, it is identified as an infinite loop case.

- **Step4:** As fourth step, the fault simulation is performed using ZO1X. In order to be able to run the ZO1X tool, following steps have to be followed:

  - **Step1:** First step is removal of the log files and unnecessary files from previous simulations. The script 4.2, is the bash script used for this step.

Listing 4.2: Bash script to remove redundant files.

```
#!/bin/csh -fe
```

```
2
3 # Remove old files to ensure clean directory
4 set outputFileList=(flops.txt \
5   sim.src \
6   zoix.log \
7   sim.zdb \
8   zoix.sim \
9   __ddbfiles__ \
10  fault_report.log \
11  __fmdict__ \
12  fmsh.log \
13  fr2fdef.log \
14  __fubs__ \
15  __globfiles__ \
16  sim.fdef \
17  user_coverage.sff \
18  testability.txt \
19  __tests__ \
20  __tmp__ \
21 )
22
23 foreach f ($outputFileList)
24   if ( -e $f ) then
25     rm -rf $f
26   endif
27 end
```

– **Step2:**In this step, using another bash script the ZO1X environment is opened and *transition_v3.fmsh* is executed by suggestion of the ZO1X tool documentation. The bash script is provided in 4.3.

Listing 4.3: Bash script to run ZO1X simulation.

```
1 #!/bin/bash
2 #First step
3 ./clean.csh
4
5 rm -rf zoix.sim sim.zdb faults.fdef simout-N0* zoix.log zoix.
    progress* zoix_rt.log *.log* *.cdf
6 #Second step
```

```
zoix -f ./bin/read_design.f ./bin/strobe_v3_2.sv +timescale+
    override+1ps/1ps +top+riscv_core+strobe +sv +
    notimingchecks +define+ZOIX +suppress+cell +
    delay_mode_fault -l log/zoix_compile.log

fmsh -load bin/transition_v3.fmsh
```

The *transition_v3.fmsh* script is also provided in 4.4. This script is created according to the official documentation of the ZO1X tool. In this script, some parameters such as fault statuses and used VCD file address, are passed to the tool. Moreover, there is a file required by the tool which contains the list of the faults needed to be simulated by the tool and in optional cases, a promotion table is provided to in order to merge the faults when multiple tests are run which, an example of it is presented in 4.5.

Listing 4.4: Script required to configure and run ZO1X simulation.

```

set(var=[resources], messages=[all])
set(var=[defines], format=[standard])
set(var=[defines], dictionary.enable=[1])
set(var=[fsim], dictionary.values=[all])
set(var=[defines], trans.delay=[40ns])
set(var=[fdef], method=[fr], fr.fr=[bin/user_v3.sff],fr.
    transition=[1],abort=[error])
set (var=[coats],status=[NA,NX,PP,FP,PD,FD,PE,FE,PL,LS,PS,RS,
    PR,RE,PX,DX])
set (var=[fsim],hyperfault=[0])
design()

addtst(test=[riscv_core], stimtype=[vcd], dictionary.enable
    =[1], stim=[../tmax/dumpports_rtl.riscv_core.vcde],dut.
    stim=[riscv_core,tb.top_i.core_region_i.CORE.RISCV_CORE],
    stim_options=[+TESTNAME=riscv_core] )
fsim()

coverage(type=[coverage],file=[TDF_coverage_V3.sff],
    collapseoff=[1])
```

```
16 coverage (type=[dictionary],file=[tdf_dic_short_V3.txt],test
    =[riscv_core], style=[short])
```

Listing 4.5: Example script to pass fault list to ZO1X simulator[35].

```
1  StatusDefinitions
2  {
3  Redefine ND NX "Not Detected"
4  NN "Not Observed Not Diagnosed";
5  NP "Not Observed Potential Diagnosed";
6  ND "Not Observed Diagnosed";
7  PN "Potential Observed Not Diagnosed";
8  OP "Observed Potentially Diagnosed";
9  ON "Observed Not Diagnosed";
10 OD "Observed Diagnosed";
11 DefaultStatus (NN)
12 Selected (NA, NN, NP, PN, OP, ON)
13 PromotionTable
14 {
15 StatusLabels (NN,NP,ND,PN,OP,ON,OD)
16 #    NN NP ND PN OP ON OD
17 [    -    |    |    |    ON   |    |    ; # NN
18      -    -    |    |    |    |    |    ; # NP
19      -    -    -    |    OD   |    |    ; # ND
20      -    -    -    -    ON   |    |    ; # PN
21      ON   -    OD   ON   OD   |    |    ; # OP
22      -    -    -    -    -    -    |    ; # ON
23      -    -    -    -    -    -    -    ; # OD
24 ]
25 }
26 }
27 FaultList {
28 NN F { PORT "riscv_core.ex_stage_i_alu_i.int_div_div_i.
    ResReg_DP_reg_28_.Q" }}
```

After running each simulation, a fault dictionary is generated. Using the python function provided in 4.6, detected faults are extracted and removed from the fault list for next fault simulations. The reason behind removing detected faults is to reduce the fault simulation time.

- **Step5:** In this step, steps 2,3,4 are repeated in a loop until all the TDFs are detected or all code blocks are tried in the destination and there is no code block left. Also, results are stored as a dictionary which, shows detected TDFs by each code block in each position in STL.

The mentioned steps, are depicted as a flowchart in figure 4.1.

Listing 4.6: Function for checking TDF detection.

```
def check_detected(input_file, name):
    dd_line =[]
    with open(input_file) as fd:
        for line in fd:
            if line.strip().startswith('DD') or line.strip().
    startswith('--'):
                dd_search=re.search(r'DD ([R/F]) {TRAN "(.*?)"}',line
    )
                if dd_search:
                    dd_line.append(dd_search.group(1)+' '+dd_search.
    group(2))
                eq_search=re.search(r'DD ([R/F]) {TRAN "(.*?)"}',line
    )
                if eq_search:
                    dd_line.append(eq_search.group(1)+' '+eq_search.
    group(2))
    return {
        name: dd_line
    }
```

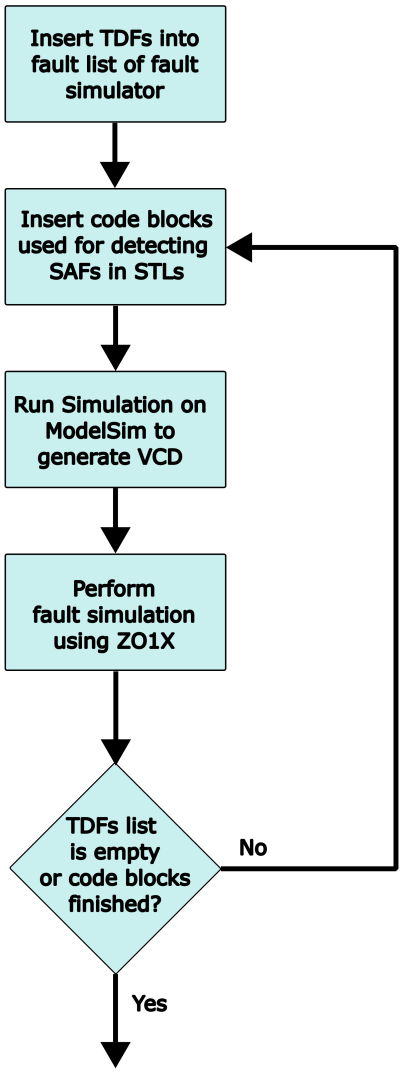

Figure 4.1: Simulations flowchart.

4.2 Results of Algorithms

In this section the achieved results in details is described. 4.2 and 4.3 show summary data on the user accessible register (UAR) and hidden register (HR) faults that were detected as a result of the proposed methodology.

Starting with 4.2, it is possible to see that the proposed 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.

Table 4.2: 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, which amounts to an additional 22.21% for STL1, 14.97% for STL2, and 21.16% for STL3. This proves that the proposed strategy in this work is able to systematically test not-observed transition delay faults whose effects reached user accessible registers.

Moving on to 4.3, it is possible to see that the results we achieved thanks to our methodology are quite dependent on the considered STL.

Table 4.3: 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 still worth mentioning that, the proposed methodology allows to automatically detect these faults, thus not requiring any manual effort from the test engineer. For this latter group, the increase in the code size is rather small, amounting to an additional 9.52% for STL1, 3.30% for STL2, and 5.52% for

STL3, respectively. It is also worth mentioning that some of the undetected faults may belong to the group of FUFs.

Table 4.4: Sub-modules analysis for the adopted STLs

Test Program		Hidden Register faults				User Accessible Register faults	
		Fetch Stage	Decode Stage	Execute Stage	Memory Stage	GPRs	SPRs
STL1	Detected faults	23	587	32	1	4,359	2,219
	Total faults	1,109	5,109	388	135	4,359	2,232
	Added Instructions	45	550	25	5	1,107	478
STL2	Detected faults	52	120	5	6	23,814	98
	Total faults	1,311	1,976	221	91	23,814	108
	Added Instructions	80	115	20	15	1,022	20
STL3	Detected faults	13	595	0	0	2,853	11
	Total faults	1,028	2,463	351	113	2,853	47
	Added Instructions	35	195	0	0	877	6

Table 4.4, describes 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 for each sub-module and STL. All the pipeline stages columns belong to the hidden registers category, while general purpose registers (GPRs) and special purpose registers (SPRs) are user accessible registers. 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 3.2.1, 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 said 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.

Chapter 5

Conclusions

This work introduces an automated and systematic methodology to detect transition delay faults whose effects have been excited but not observed by already available STLs. Starting from a library of self-test programs developed for stuck-at faults, the approach defines strategies to detect faults based on where their effects propagated and stopped inside the DUT, dividing them into user accessible registers and hidden register groups. Experimental results gathered on a RISC-V test case show that almost every fault affecting UARs is detectable, with the worst case scenario being a 98.76% UAR fault coverage. Data on HR faults, on the other hand, show that we are capable of detecting from 5% to more than 15% of all HR faults. Such increase in fault coverage comes with a reasonably small increase of the code size, with the worst case scenario consisting in about 22% added code size for UAR faults, while the contribution for HR faults is practically negligible. The main strength of this work resides in the fact that it is completely automated, hence not requiring any effort from the test engineer, and can drastically enhance the quality of the available STL. Future works will include the refinement of strategies to test HR faults, in order to match as closely as possible the upper bounds in recoverable fault coverage presented in [22].

Bibliography

[1] 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)*. 2013, pp. 1462–1467. DOI: 10.7873/DATE.2013.298 (cit. on p. 1).

[2] 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)*. 2006, pp. 393–398. DOI: 10.1145/1146909.1147014 (cit. on p. 1).

[3] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Sonza Reorda. «Microprocessor Software-Based Self-Testing». In: *IEEE Design & Test of Computers* 27.3 (2010), pp. 4–19 (cit. on p. 1).

[4] Riccardo Cantoro, Francesco Garau, Patrick Girard, Nima Kolahimahmoudi, and Sandro Sartoni et al. «Effective techniques for automatically improving the transition delay fault coverage of Self-Test Libraries.» In: *ETS 2022 - IEEE 27th European Test Symposium* 27 (May 2022), pp. 1–2 (cit. on pp. 1–3, 20–22, 28–32, 42).

[5] N. Hage, R. Gulve, M. Fujita, and V. Singh. «On Testing of Superscalar Processors in Functional Mode for Delay Faults». In: *International Conference on VLSI Design and International Conference on Embedded Systems (VLSID)*. 2017, pp. 397–402. DOI: 10.1109/VLSID.2017.58 (cit. on pp. 2, 21, 22).

[6] 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)*. 2020, pp. 646–650. DOI: 10.1109/DSD51259.2020.00105 (cit. on pp. 2, 21, 22).

[7] K. Christou, M.K. Michael, P. Bernardi, M. Grosso, E. Sanchez, and M. Sonza 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)*. 2008, pp. 389–394. DOI: 10.1109/VTS.2008.37 (cit. on pp. 2, 20).

[8] 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: *IEEE VTS*. 2005, pp. 107–113 (cit. on p. 2).

[9] Virendra Singh, Michiko Inoue, Kewal K. Saluja, and Hideo Fujiwara. «Instruction-Based Self-Testing of Delay Faults in Pipelined Processors». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 14.11 (2006), pp. 1203–1215. DOI: 10.1109/TVLSI.2006.886412 (cit. on pp. 2, 20).

[10] P. Bernardi, R. Cantoro, S. De Luca, E. Sánchez, and A. Sansonetti. «Development Flow for On-Line Core Self-Test of Automotive Microcontrollers». In: *IEEE Transactions on Computers* 65.3 (2016), pp. 744–754 (cit. on p. 2).

[11] Wei-Cheng Lai, A. Krstic, and Kwang-Ting Cheng. «Test program synthesis for path delay faults in microprocessor cores». In: *Proceedings International Test Conference 2000 (IEEE Cat. No.00CH37159)*. 2000, pp. 1080–1089. DOI: 10.1109/TEST.2000.894321 (cit. on pp. 2, 20).

[12] P. Bernardi, M. Grosso, E. Sanchez, and M. Sonza Reorda. «A Deterministic Methodology for Identifying Functionally Untestable Path-Delay Faults in Microprocessor Cores». In: *International Workshop on MTV*. Dec. 2008, pp. 103–108. DOI: 10.1109/MTV.2008.9 (cit. on p. 2).

[13] 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)*. 2019, pp. 76–81. DOI: 10.1109/VLSI-SoC.2019.8920306 (cit. on pp. 2, 21).

[14] R. Cantoro, S. Sartoni, and M. Sonza Reorda. «In-field Functional Test of CAN Bus Controllers». In: *IEEE VTS*. 2020, pp. 1–6. DOI: 10.1109/VTS48691.2020.9107628 (cit. on p. 2).

[15] A. Apostolakis, D. Gizopoulos, M. Psarakis, D. Ravotto, and M. Sonza Reorda. «Test Program Generation for Communication Peripherals in Processor-Based SoC Devices». In: *IEEE Design & Test of Computers* 26.2 (2009), pp. 52–63. DOI: 10.1109/MDT.2009.43 (cit. on p. 2).

[16] A. van de Goor, G. Gaydadjiev, and S. Hamdioui. «Memory testing with a RISC microcontroller». In: *DATE*. 2010, pp. 214–219. DOI: 10.1109/DATE.2010.5457210 (cit. on p. 2).

[17] Hitex. *Microcontroller self-test libraries*. URL: https://www.hitex.com/tools-components/software-components/selftest-libraries-safety-libs/pro-sil-safetlib/ (cit. on p. 2).

[18] ARM. *Enabling Our Partnership to Bring Safer Solutions to the Market Faster*. URL: https://developer.arm.com/technologies/functional-safety (visited on 06/26/2019) (cit. on p. 2).

[19] Microchip Technology Inc. *16-bit CPU Self-Test Library User's Guide*. 2012. URL: http://ww1.microchip.com/downloads/en/DeviceDoc/52076a.pdf (visited on 06/26/2019) (cit. on p. 2).

[20] STMicroelectronics. *Guidelines for obtaining IEC 60335 Class B certification for any STM32 application*. Mar. 2016. URL: http://www.st.com/content/ccc/resource/technical/document/application%5C_note/02/1a/91/78/e4/15/4d/35/CD00290100.pdf/files/CD00290100.pdf/jcr:content/translations/en.CD00290100.pdf (cit. on p. 2).

[21] J. Perez Acle, R. Cantoro, E. Sanchez, M. Sonza Reorda, and G. Squillero. «Observability Solutions for In-Field Functional Test of Processor-Based Systems». In: *Microprocessors and Micros.* (2016), pp. 392–403. ISSN: 0141-9331. DOI: 10.1016/j.micpro.2016.09.002 (cit. on p. 2).

[22] 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)*. 2021, pp. 1–4. DOI: 10.1109/IOLTS52814.2021.9486711 (cit. on pp. 2, 20, 21, 28, 53).

[23] N. K. Jha and S. Gupta. *Testing of Digital Systems*. Cambridge University Press, New York: Cambridge University Press, 2003 (cit. on pp. 4–6, 8, 15).

[24] E.P. Hsieh, Rasmussen, Vidunas R.A., L.J., and W.T. Davis. «Delay test generation.» In: *Proc. Design Automation Conference* (1977), pp. 486–491 (cit. on p. 7).

[25] T.M. Storey and J.W. Barry. «Delay test simulation». In: *Proc. Design Automation Conference* (1977), pp. 492–494 (cit. on p. 7).

[26] J.P. Lesser and J.J. Shedletsky. «An experimental delay test generator for LSI logic». In: *IEEE Trans. on Computers* (1980), pp. 235–248 (cit. on p. 8).

[27] G.L. Smith. «A model for delay faults based on paths.» In: *Proc. Int. Test Conference* (1985), pp. 342–349 (cit. on p. 8).

[28] Michael L. Bushnell and Vishwani D. Agrawal. *Essential of Electronic Testing*. New York, Boston, Dordrecht, London, Moscow: Kluwer Academic Publishers, 2002 (cit. on pp. 10–12).

[29] Matteo Sonza Reorda. *Testing and Fault Tolerance course*. 2021 (cit. on p. 18).

[30] Mihalis Psarakis, Dimitris Gizopoulos, Ernesto Sanchez, and Matteo Sonza Reorda. «Microprocessor software-based self-testing». In: *IEEE Design & Test of Computers* 27 (2010), pp. 4–19 (cit. on pp. 19, 20).

[31] C. -Y. Chen and J. -L. Huang. «Reinforcement-Learning-Based Test Program Generation for Software-Based Self-Test». In: *IEEE Asian Test Symposium (ATS)*. 2019, pp. 73–735. DOI: 10.1109/ATS47505.2019.00013 (cit. on p. 22).

[32] 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)*. 2019, pp. 1–6. DOI: 10.1109/DFT.2019.8875345 (cit. on p. 22).

[33] A. Jasnetski, R. Ubar, and A. Tsertov. «On automatic software-based self-test program generation based on high-level decision diagrams». In: *IEEE LATS*. 177-177. 2016. DOI: 10.1109/LATW.2016.7483357 (cit. on p. 22).

[34] A. Jasnetski, R. Ubar, and A. Tsertov. «Automated software-based self-test generation for microprocessors». In: *International Conference MIXDES*. 2017, pp. 453–458. DOI: 10.23919/MIXDES.2017.8005252 (cit. on p. 22).

[35] ZO1X. *Functional Safety Assurance*. URL: https://www.synopsys.com/verification/simulation/z01x-functional-safety.html (cit. on pp. 22, 47).

[36] ModelSim. URL: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/model-sim.html (cit. on p. 22).

[37] Synopsis. *Design Compiler: Concurrent Timing, Area, Power, and Test Optimization*. URL: https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html (cit. on p. 22).

[38] ETH Zurich and Universita di Bologna. *PULPino microcontroller system*. URL: https://github.com/pulp-platform/pulpino (cit. on pp. 25, 26, 42).

[39] Andreas Traber, Florian Zaruba, Sven Stucki, Antonio Pullini; Germain Haugou, Eric Flamand, Frank K. Gürkaynak, and Luca Benini. *PULPino: A small single-core RISC-V SoC*. URL: https://riscv.org/wp-content/uploads/2016/01/Wed1315-PULP-riscv3_noanim.pdf (cit. on p. 27).

[40] Andreas Riefert, Riccardo Cantoro, Matthias Sauer, Matteo Sonza Reorda, and Bernd Becker. «A Flexible Framework for the Automatic Generation of SBST Programs». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24.10 (2016), pp. 3055–3066. DOI: 10.1109/TVLSI.2016.2538800 (cit. on p. 33).

[41] Francesco Garau. *Enhancing programs for delay test of microprocessors through fault propagation analysis*. URL: http://webthesis.biblio.polito.it/id/eprint/21298 (cit. on pp. 38–40).

[42] Silvaco. *Silvaco 45nm Open Cell Library*. URL: https://www.silvaco.com/products/nangate/FreePDK45_Open_Cell_Library/ (cit. on p. 42).