POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Politecnico
di Torino

Master's Degree Thesis

Enhancing programs for delay test of microprocessors through fault propagation analysis

Supervisors

Prof. Riccardo Cantoro

Prof. Matteo Sonza Reorda

Dott. Sandro Sartoni

Candidate

Francesco Garau

Academic Year 2020-2021

Saruman! Il tuo bastone è rotto!

Summary

The presented thesis work is centered on the topic of *functional test* of microprocessors. Functional test, in particular using Software Test Library (STL), is becoming a standard solution for the online test of safety-critical systems, for instance in the automotive domain. Commercial tools provide scan test patterns, however frequently they are not able to reach the given target fault coverage. Nevertheless, the development of high-quality test programs is considerably more challenging than using commercial tools, therefore new solutions are required to improve the fault coverage guaranteed by a STL.
The proposed work focuses on the analysis of the propagation of the faults through the paths of the device under test and tries to propose valid solutions to improve the fault coverage of the tests.
First, the thesis describes an approach that identifies the set of sequential cells that captures fault effects before being masked during the propagation towards observable points. In order to reach the target fault coverage, a subset of sequential cells is selected using two different set coverage algorithms, assuming that they can be observed resorting to suitable solutions (for example, existing infrastructures, MISR). Subsequently, the work tries to analyze the execution of the test programs. Particularly, starting from the execution trace of the test program, the source code of the program and a fault dictionary, this approach tries to identify the pieces of source code that generate the *hard-to-test* faults. The goal of this analysis is to give support to the test engineer indicating which parts of the source code should be modified in order to detect those faults and, consequently, allows improving the STLs. The test of this methodology is not the main goal of this thesis, but it is a starting point for a future work.
For the experimental part, the PULPino RISCV architecture has been chosen.

Acknowledgements

Sono grato di aver avuto l'opportunità di poter collaborare con i miei supervisori Riccardo Cantoro, Sandro Sartoni e Matteo Sonza Reorda, e li ringrazio per l'aiuto e il supporto che mi hanno dato durante i mesi della tesi, facendomi sentire parte di un gruppo serio e coeso.
Voglio ringraziare gli amici e tutti coloro che in questi due anni di laurea magistrale, e in generale negli ultimi anni, mi hanno aiutato, ascoltato e apprezzato, rendendo il mio percorso più semplice e meno insidioso. In particolare, ci tengo a ringraziare tutti i miei amici *arburesi*: il gruppo "Le-Plaza". Non posso dimenticare i miei amici *gutturesi*, con i quali ho condiviso più di un decennio di vacanze estive. Nello specifico, vorrei dedicare questo risultato ad Alice, Andrea, Elena e Francesca.
Meriti di questo traguardo spettano ovviamente a tutti quei colleghi che mi hanno aiutato, sopportato e con cui mi sono confrontato in questi due anni intensissimi, in particolare Alessandro, Claudio, Martina e Veronica. Ringrazio caldamente i miei conquilini torinesi, coloro i quali sono diventati la mia seconda famiglia durante questi due anni: Claudia, Federica, Giacomo, Lorenzo e Marika.
Dulcis in fundo, dedico la laurea a coloro che hanno materialmente permesso che io potessi intraprendere questo percorso, grazie ai propri sacrifici e al loro affetto. Ringrazio con tutto il cuore i miei genitori Adriana e Fernando per il supporto costante e per l'opportunità di proseguire i miei studi prima a Cagliari e poi a Torino. Spero di aver ripagato i vostri sacrifici con quest'ultimo traguardo accademico. Oltre ai miei genitori, non posso non ringraziare mio fratello Giacomo e la sua fidanzata Eleonora, per l'aiuto che mi hanno dato in questi due anni a Torino (e per avermi sopportato, scarrozzato di qua e di là e tante altre cose) ma anche nei 3 precedenti a Cagliari durante la mia laurea triennale.
Infine, vorrei concedere un piccolo tributo a me stesso. Nonostante alcuni periodi difficili, difficoltà e talvolta sconforto, non ho mai mollato e sono arrivato all'obiettivo che mi ero preposto.
Questo è solo il punto di arrivo della carriera universitaria. Che sia il punto di partenza per un nuovo percorso e per nuove soddisfazioni.

Table of Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

The aim of the proposed thesis work is to find and develop new solutions for delay testing of SoCs and microprocessors, and, in particular, the work is centered on the topic of functional testing.
Functional test, in particular using Software Test Libraries (STL), is becoming a standard solution for the online test of safety-critical systems, for instance in the automotive domain. Commercial tools provide different solutions, like scan test patterns, however they are frequently not able to reach the given target fault coverage. Nevertheless, the development of high-quality test programs is considerably more challenging than using commercial tools.
Since commercial solutions (ATPG and scan patterns) and STLs that companies provide are able to test and cover a large but not sufficient amount of faults, it is necessary to develop new methodologies to reach higher fault coverage, especially in those domains in which *reliability* and *security* are fundamental.
The proposed work focuses on the analysis of the propagation of the faults through the paths of the device under test and tries to propose valid solutions to improve the fault coverage of the tests. The propagation of faults to primary outputs is one of the most difficult challenges that a test engineer must face. In fact, a notable amount of faulty signals propagate through the integrated circuit of the device under test but stop at an internal gate and can not be observed by the user.
The methodologies developed and presented in the next chapters fit in a dynamic and evolving context in which are inserted several works. Specifically, various studies investigate the possibility to improve test programs, mainly related to *stuck-at* faults. Other works propose the usage of *post-silicon* debug solutions, demonstrating the effectiveness of these ones. All these studies achieved satisfying results but they do not consider *delay faults* and the ones that propose the exploitation of post-silicon methodologies work on the whole core, often requiring a considerable area overhead of the circuit. The study [1] moves the first steps in the development of a methodical strategy for the improvement of the existing STL quality, focusing

on the *transition delay* faults. This work is the starting point of this thesis, which can be considered its natural following step.
In this thesis two approaches are described. First, a hybrid methodology for the improvement of the fault coverage is proposed: this solution has been defined as *hybrid* since at first it requires a software computation and afterwards it uses a hardware infrastructure to perform the task. Starting from a list of *hard-to-test* faults individuated by a simulation analysis performed with STL means, the software defines a list of internal signals that should be monitored in order to observe the propagation of these faults. Subsequently, the selected signals must be physically observed with hardware means (for instance, with MISR) in order to detect the faults. For this methodology, the thesis focuses mainly on the development of two algorithms for the selection of the wires that should be monitored.
In the second part of the work, a methodology for the analysis of the test programs (STLs) execution trace is described. The aim of this section is to create a software infrastructure that can be exploited by the test engineer to store useful information about the test programs (for instance, executed instructions and CPU execution time, set of faults tested), consequently analyse the weaknesses of the programs in order to improve them. Finally, it provides some basic functions for editing the programs. To achieve this goal, a python software has been developed. It provides some functionalities as the insertion into a database of the execution trace information, statistical analysis of the faults and the manipulation of the STLs (for instance, giving the possibility to insert assembly instructions into the source code, according to different strategies). The latter feature is still being developed since it requires several tests, mainly concerning the strategies for the insertion of new instructions into the source code.
The results produced by the first methodology are satisfying and promising since they show the possibility to recover 100% of the hard-to-test faults. The first algorithm, defined as *greedy* since it performs a fixed selection of the wires, has been proved to be inefficient since it requires a notable and almost unfeasible amount of wires to be observed and, sometimes, it can achieve 100% of coverage. On the other hand, the second algorithm, which performs a variable selection of wires, shows the possibility to monitor a smaller number of wires and it reaches 100% of coverage almost in every case.
The test of the second methodology, in particular the manipulation of the STLs, was not the main goal of this part of the thesis, therefore any results have not been produced. Nevertheless, in the sub-chapter 6.3 some graphs concerning the analysis of the execution trace of the test programs are presented.
The chapters 2 and 3 show the *state of art* of testing methodology and the *context* in which this thesis is inserted. Afterwards, in the chapters 4 and 5 the two methodologies developed in the thesis work are presented, and in the chapter 6 the achieved results are described. Eventually, the reader will find the conclusions and

a picture of the potential related works in the chapter 7.

Chapter 2

Testing methodologies and technologies

This chapter will focus on testing methodologies.
Across the years, many testing technologies have been developed in order to find suitable solutions for different necessities. In particular, it is possible to split them according to the part of the circuit they are made to test: *combinational* and *sequential*.
In the first category can be found the *automatic test pattern generation (ATPG)* tools and in the second one the (*scan patterns*) and the functional testing.

2.1 Fundamentals of testing

In general, *Testing* is the process that aims to find faults and misbehaviour in a product. In particular, testing a microprocessor consists in applying a set of stimuli to the integrated circuit in order to excite physical defects and make them observable at one of the IC's output ports.
Testing is a crucial phase in the lifecycle of a product and it is very important for all those application in which safety and reliabilty are indispensable, as in the automotive domain.
During the 90s, the manufactoring cost per transistor has dropped rapidly. On the other hand, testing cost remained fairly steady over the years (figure 2.1). This trend pushes to the necessity of developing new strategies in order to minimize costs but, at the same time, maximize results and guarantee high quality standards.
Physical faults may depend on many different causes and are often difficult to deal with, therefore it is impossible and unfeasible to take each of them into account.
The general solution is to use *logical faults*. These ones model a physical defect (or a set of defects), which is consequently called *fault model*.

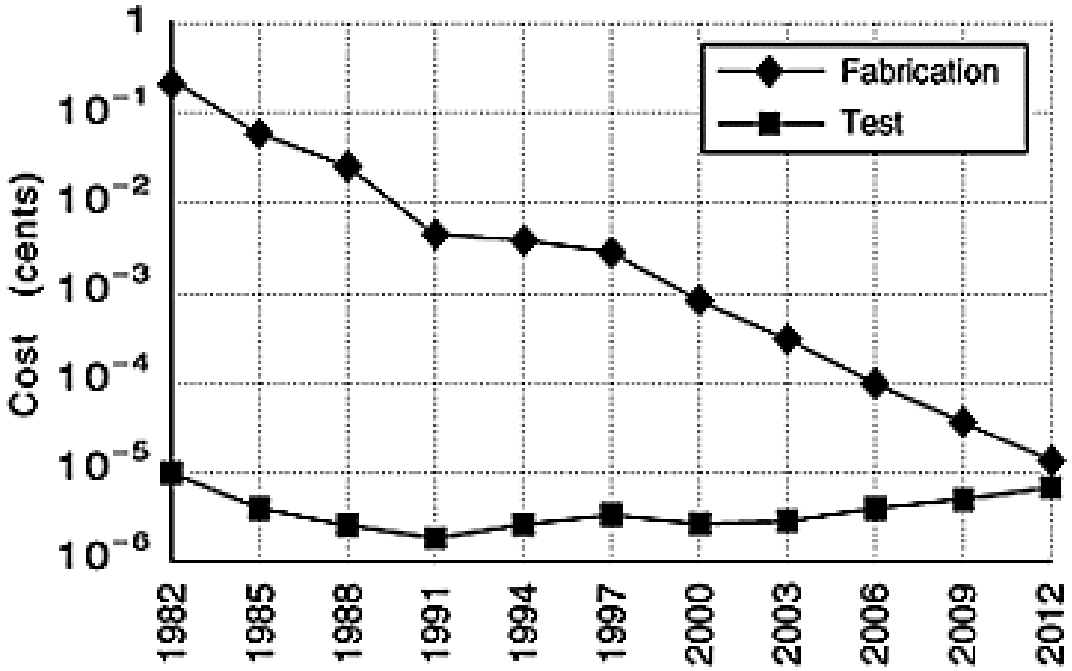

Figure 2.1: Evolution of manufactoring and testing cost per transistor

The most common classes are the *stuck-at* and the *delay* fault models.

2.1.1 Stuck-at fault model

Stuck-at fault (SAF) is a particular fault model used for *post manufactoring test*. The model assumes that individual signals or pins of the circuit are *stuck* at a logical signal: '1', '0'. The former one is called *stuck-at zero* whilst the second one *stuck-at one*.
Figure 2.2 shows a simple example of stuck-at fault. In particular, the E wire is affected by a stack-at one fault. In the case of healthy circuit, the output would be the following:

1. wire A and B are the inputs of a NAND port. 1 NAND 1 = 0, therefore E = 0;
2. wire C and D are the inputs of an OR port. 0 OR 0 = 0, therefore F = 0;
3. wire E and F are the inputs of an OR port. 0 OR 0 = 0, therefore the output is 0

With the stuck-at one fault on the E wire the outputs become:

1. E = 1 because of the fault;
2. wire C and D are the inputs of an OR port. 0 OR 0 = 0, therefore F = 0;
3. wire E and F are the inputs of an OR port. 1 OR 0 = 1, therefore the output is 1

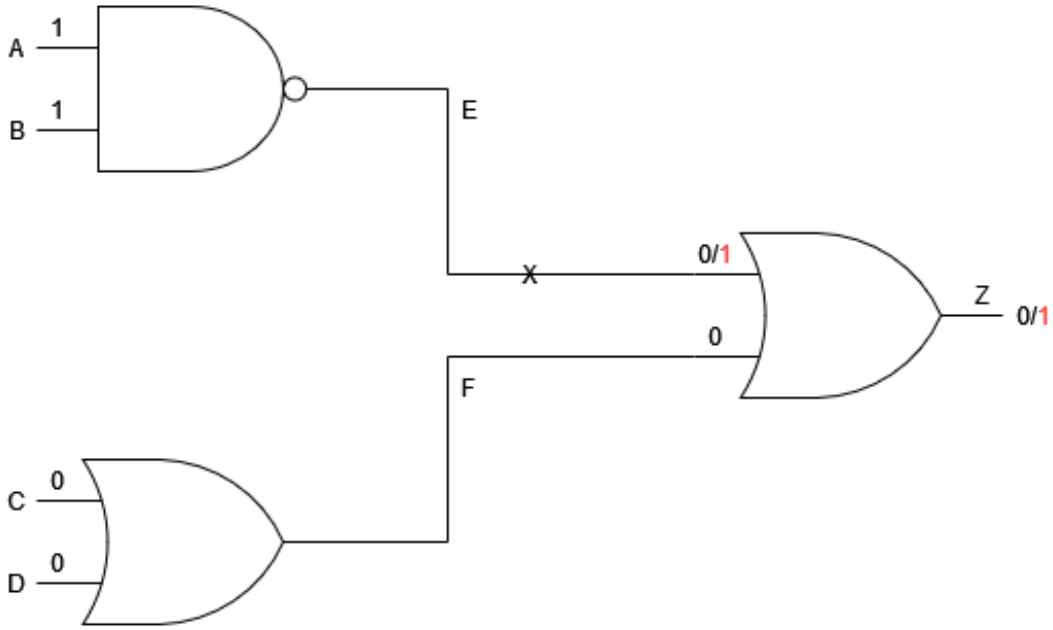

Figure 2.2: Stuck-at example

In order to make the fault observable it is needed an input vector that is capable to excite the fault.
Figure 2.3 shows an example of inefficient input vector. In this case, A NAND B = 1 because B=0 so it is impossible to observe the stuck-at one fault on the E wire because we could not determine if the value is correct or if it depends on a fault.

2.1.2 Delay fault models

Delay fault (DF) classes aim to model temporal defects in an integrated circuit. Even though the circuit performs its operation correctly, it may be too slow to propagates signals through paths and gates. Consequently, the circuit outputs could show incorrect logical values. DF testing aims to determine whether the circuit works correctly *at speed* (its specified speed) or not [2].

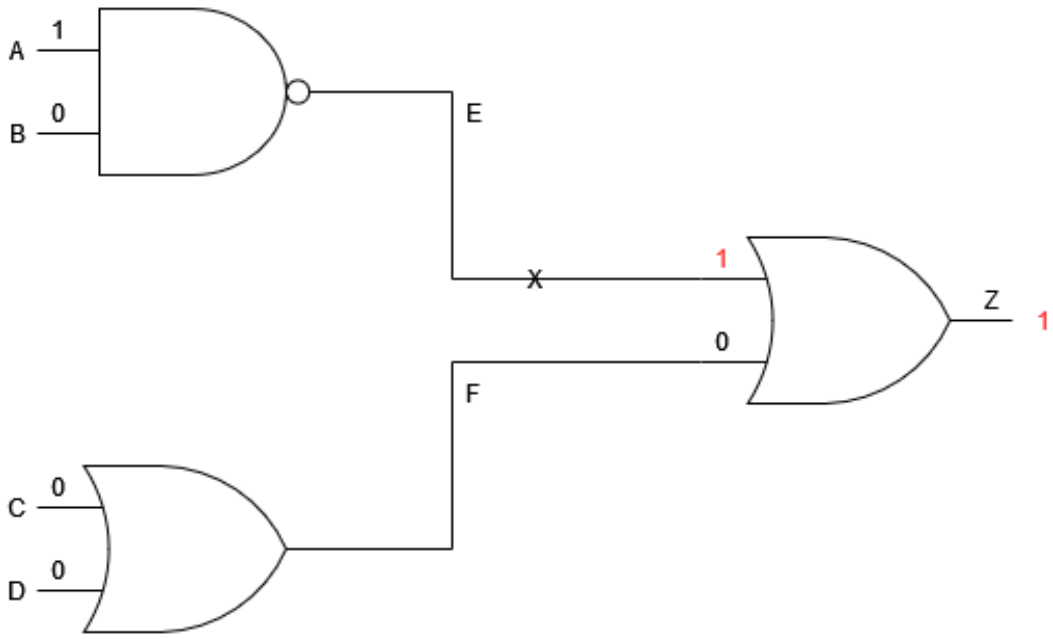

Figure 2.3: Stuck-at bad input vector

Therefore, DFM models failures caused by temporary changes in the circuit characteristics.
According to the Huffman scheme (figure 2.4), a generic sequential circuit can be model with two parts: the combinational logic (it only depends on input and corresponding outputs) and a section able to store logic values, for instance a set of flip-flops.
The maximum operating frequency, and consequently the operating speed of the circuit, depends on the time needed by the combinational circuit to produce stable output values. If the combinational logic produces correct outputs but not within a specific time, the flip-flops will sample incorrect values: this situation lead to delay faults.
The most used delay fault models are the *Transition delay (TDF)* and the *Path delay (PDF)*.

Transition Delay Fault Model

The *Transition delay fault model*, also called *Gate Delay*, models defects as a delay of a single gate to pass from the logical value 0 to 1 or viceversa. The former case is called *slow-to-rise*, the latter *slow-to-fall*.
In comparison to SAFs, testing a transition delay fault requires a *pair of input*

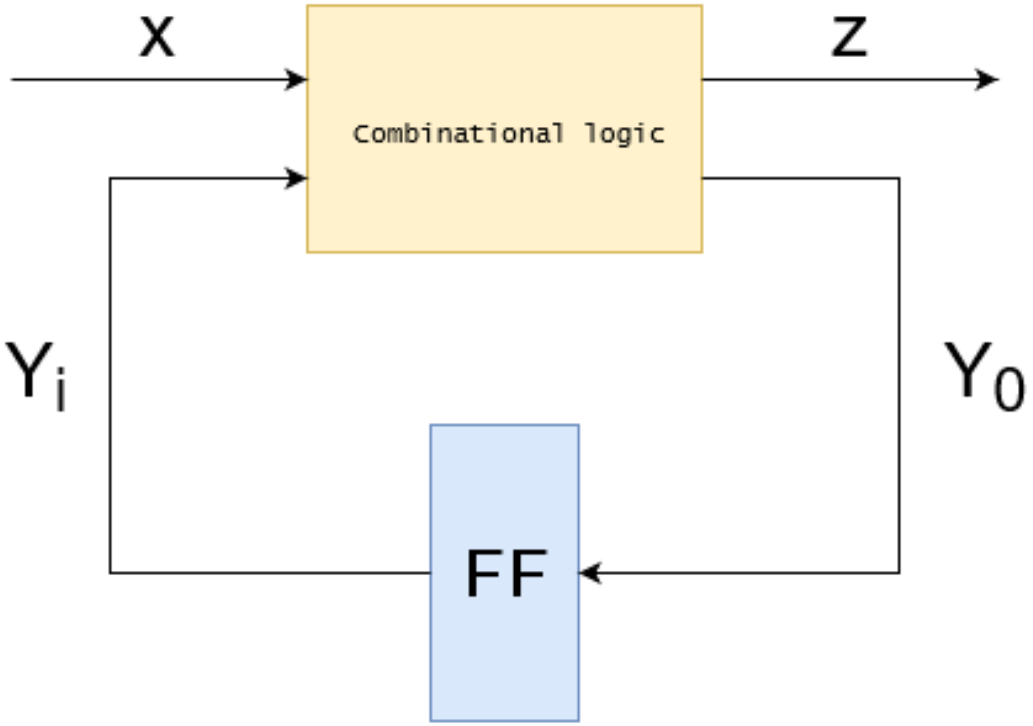

Figure 2.4: Huffman scheme

vectors instead of one.
The first vector must force the output value of the gate to 0 or 1. The second one must force the gate to execute the transition to 1 or 0 (depending on the initial value) and propagate the transition up to the block outputs.
As said before, the input vectors should be applied at *the maximum operating speed* of the circuit.
In figure 2.5, it is shown a transition delay fault example, in particular a *slow-to-fall* case. The input A presents a delay in the transition from 1 to 0 and the fault is observable in the outputs Y and Z.
In order to be observable, a suitable pair of vector is needed. For example, giving as input of wire B two consecutive 0s, the output of the NAND port would have been 1 in any case, making the transition fault on wire A undetectable.
The same applies for the E wire and the main output Z.
When choosing the test vectors, it is also important to consider compensating effects.
In fact, it is common to choose combinations of values that lead to reconvergence when propagating the faulty transition, which inevitably becomes undetectable .

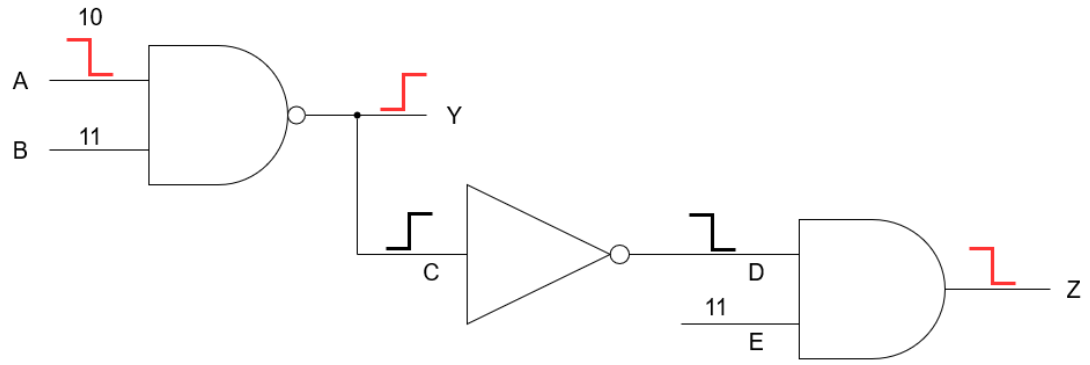

Figure 2.5: Transition Delay fault example

Path Delay Fault Model

The *Path delay fault model* models a distributed delay effect through a given path of the integrated circuit passing from value 0 to 1 or viceversa (as TDF, 0->1 is called *slow-to-rise* and 1->0 *slow-to-fall*). Therefore, this fault class focuses on defects of multiple gates interconnected (from an input or FF output to a main output or an input of another element of the circuit) instead of a single node (TDF).
The number of faults that can be generated with PDF model is proportional to the size of the circuit, in particular is equal to $2m$, with m the number of paths of the IC. Consequently, as the circuit under test grows, a very large set of pattern should be generated in order to reach a satisfying coverage, making this solution not always applicable and feasible.
In order to limit the number of testing vectors, it is common to rank paths according to their length, usually taking into account the sum of delays of the gates that compose the path, and generate the pairs only for the longest ones.
For instance, taking in consideration the simple circuit in figure 2.6, it is possible to notice that the slow-to-fall transition that affects the A wire is propagated to the input C of the second AND port and it is observable on the main output Z.
Also in this case, it is fundamental the choice of a suitable input vector in order to avoid recovergence.

Limitations of delay fault models

As mentioned before, delay testing requires a pair of vectors in order to test a circuit.
Moreover, in the case of path delay faults, the number of faults is double the number of path, therefore reaching a satisfying coverage could be not quickly affordable. In

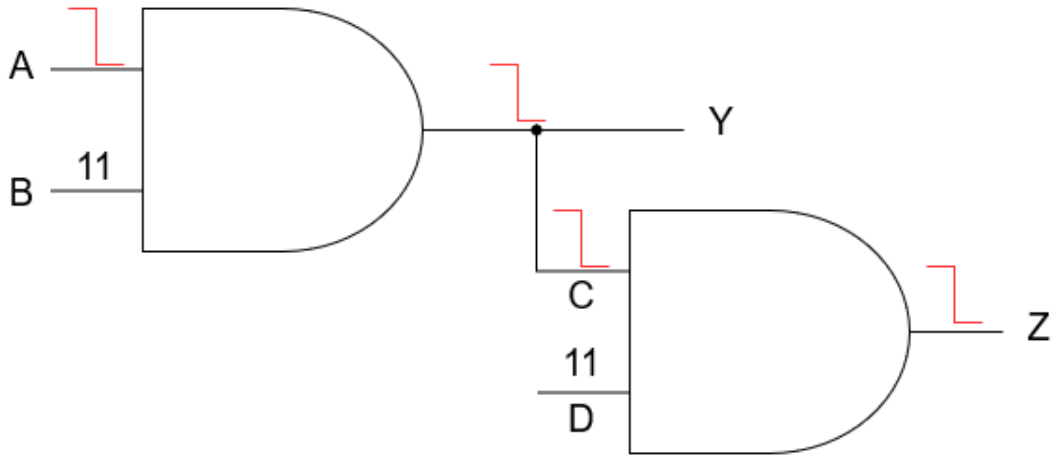

Figure 2.6: Path Delay fault example

addition, not all the paths can be tested.
Untestable faults can be divided in two categories: *structurally* and *functionally* untestable faults:

- ***structurally untestable***: no pair of input vectors applied to the *inputs of a combinational logic block* is able to propagate the fault on the *outputs of the combinational block*;
- ***functionally untestable***: no sequence of stimuli applied to the *inputs of the circuit* is able to propagate the fault on the *outputs of the circuit*. In this case, the fault is *structurally testable* but the input sequence is not able to produce a pair of vectors on the input of the combinational block able to excite the fault and propagate it to the circuit outputs.

The percentage of untestable delay faults may be significant. In particular, functionally untestable may be considerably more than structurally untestable ones.
Figure 2.7 shows the relationship between the whole set of delay faults and testable delay faults. In the next chapters, the reader will find a brief overview of the main testing methodologies and a description of the *context* in which this work operates.

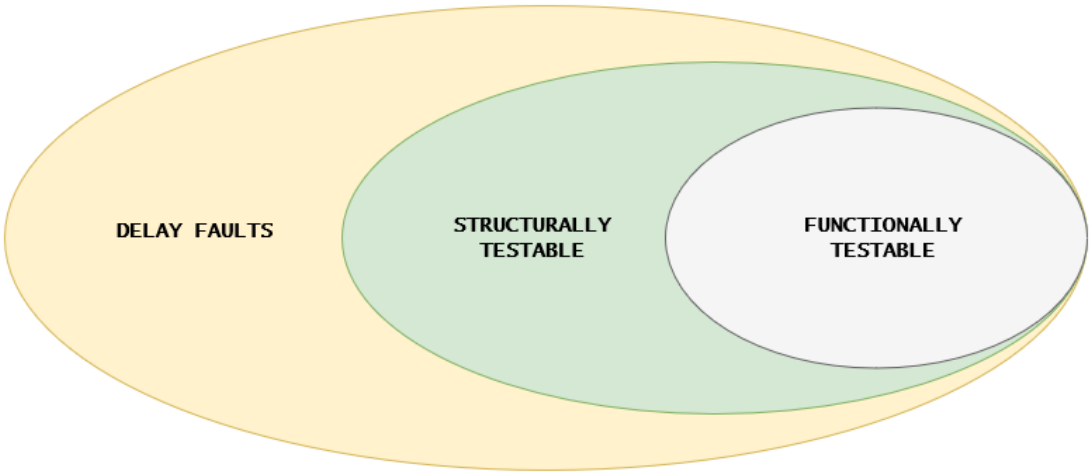

Figure 2.7: Relationship between the whole set of delay faults and testable delay faults

2.2 Automatic Test Pattern Generation

ATPG is a technology used for the generation of test pattern for circuits.
Real circuits are often very large, therefore it is impossible and unfeasible the manual generation of patterns. Hence the need to design and develop an automatic solution to perform this task.
At the same time, ATPG tools can successfully deal with combinational circuits but are inefficient with sequential ones. Therefore, in order to test large circuits different solution are needed.bu

2.2.1 ATPG architecture

As shown in figure 2.8, ATPG system architecture is composed of two main operational blocks (*Fault Manager* and *Test Pattern Generator*) that require specific information about the circuit (*circuit description*) in order to generate the *fault list* and the *test vectors*. Moreover, the tools give information about the simulation and the obtained results (*fault coverage* and *untested faults*).

Fault Manager

The *Fault Manager* block is in charge to perform the following tasks:

- generate the fault list;
- identify untestable faults;

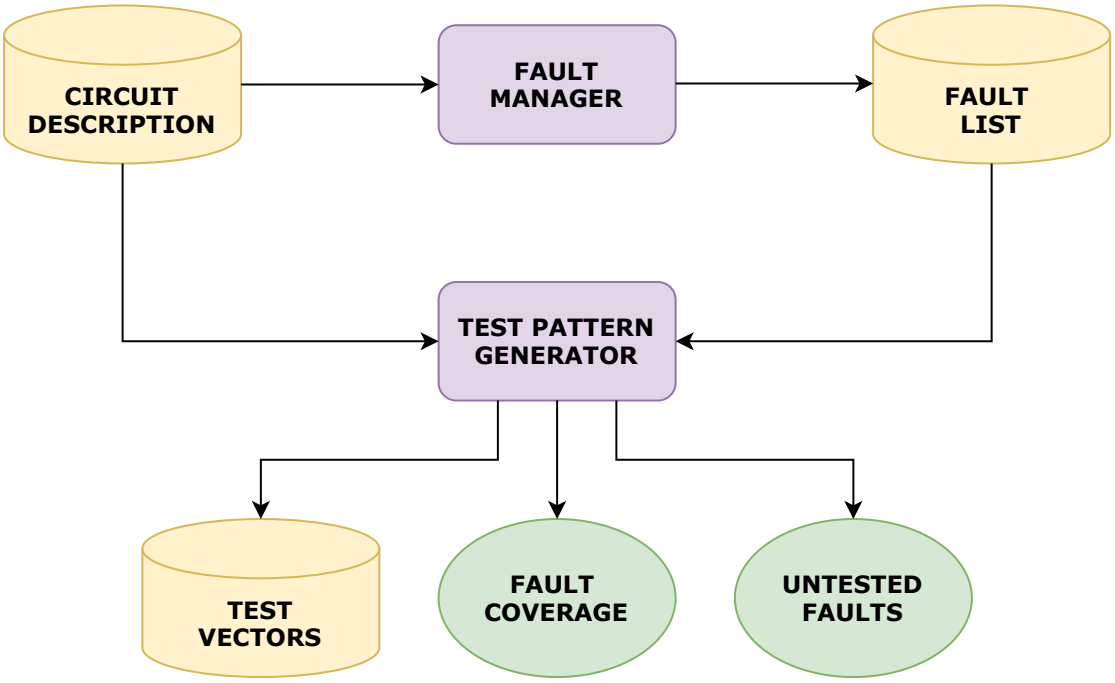

Figure 2.8: ATPG system architecture schema

- perform fault collapsing

Starting from the *circuit description* (for instance, Verilog or VHDL file that describes the circuit), the Fault Manager is able to *generate the fault list*: it produces a list of fault that can affect the circuit and that should be tested.
The second step is *identify untestable faults*. These one are defects that cannot be tested due to physical characteristics of the circuit. Hence the need to exclude them so that the overall number of faults to be tested can be reduced.
The third step aims to reduce once more the number of faults performing the *fault list collapsing*. This operation is based on the assumption that multiple faults can be equivalent, therefore it is unnecessary to test all of them.
Collapsing can be done in some ways [3], for example:

- ***equivalence collapsing***: two or more faults produce the same behaviour for all input patterns. The whole set of faults can be represented by any single fault of the set;
- ***dominance collapsing***: a fault A is called dominant to B if all tests of B detect A. Due to this, A can be removed from the fault set;
- ***functional collapsing***: two faults are functionally equivalent if it is impossible to distinguish them at the primary outputs (PO) with any input vector.

They produce the same faulty functions

Test Pattern Generator

The *Test Pattern Generator* block is composed by an ATPG module and a Fault Simulation module. These modules perform the core steps of the test simulation of the circuit, which follows the next process:

1. fault manager steps (2.2.1, Fault Manager)
2. generation of first test vector step and fault simulation
3. test vector generation and fault simulation

After the steps performed by the Fault Manager module, the tool executes a first fault simulation. The starting point of this test are vectors that could be categorized in two possible ways:

- vectors developed by designers for Design Verification [1];
- vectors generated with random approaches

Starting from this vectors, the tool performs a first fault simulation in order to identify detected faults and eliminate them from the fault list (Fault Dropping). Point 3 is performed by the ATPG module and it is the most time consuming step. As shown in the figure 2.9, the tool selects a fault from the list of *undetected faults* (this step can be performed according to different strategies and these can heavily impact the cost of the whole process), generates a test set for that fault and eventually launches the Fault Simulator to perform a Fault Dropping. The loop stops when the undetected fault list is empty or when the available resources (CPU time, memory) are exhausted. At the beginning of the ATPG steps, all the faults in the fault list are labelled as *untested*. As the process continues, each fault of the list receives a new label according to the result:

- *untestable*: the ATPG proved that the fault cannot be tested. The fault is removed from the fault list used by the fault simulator and the ATPG;
- *tested*: a test vector has been generated for the fault. This is removed from the fault list;

[1] Design Verification Test is a testing program performed to verify that the products respect all the design specifications, interface standards, diagnostic commands and Original Equipment Manufacture requirements [4]

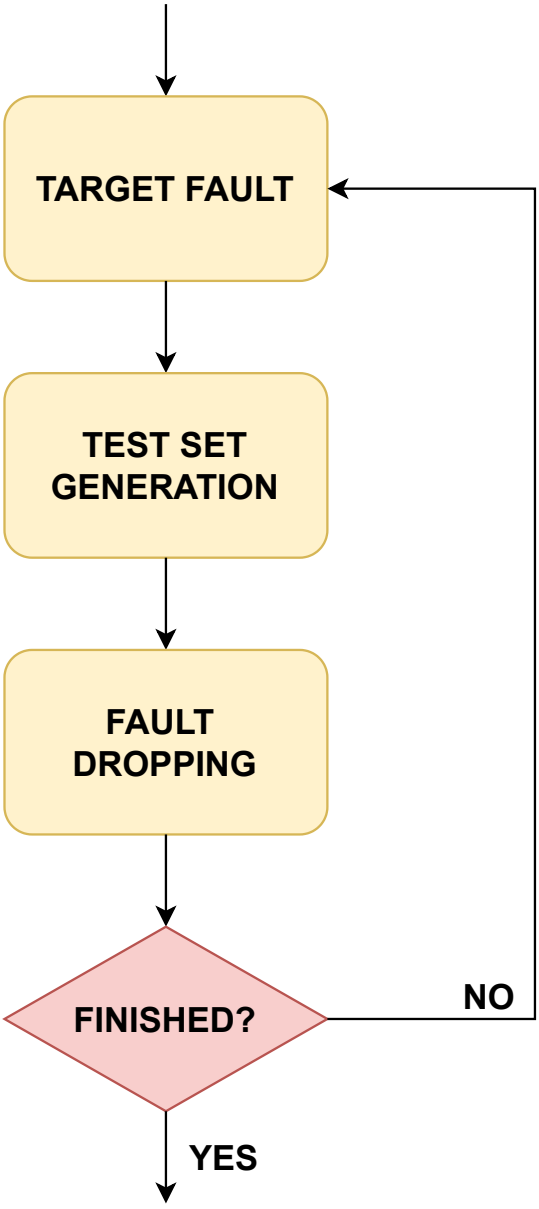

Figure 2.9: ATPG fault test loop

- *aborted*: the ATPG reached a threshold and could not prove the testability or the untestability. The fault is removed from the list of the ATPG but it remains in the Fault Simulator list in order to be tested later.

2.3 Scan techniques

ATPG methodology is inadequate when dealing with sequential circuit testing.
In fact, this approach requires a remarkable amount of time and memory to generate the tests, therefore the reduction in time and memory to perform this task is strongly necessary. Furthermore, a solution for reduce the number of cycles needed to apply the tests to the circuit should be developed. [5].
Nowadays, ATPG tools for sequential circuit testing requires excessive CPU time and the obtained fault coverage is usually unsatisfactory.
A common solution for these concerns are the scan methodologies.
In order to apply scan techniques, the IC is designed to have two different configurations: *functional mode* and *test mode*. The first one is the configuration that allow the circuit to perform its functional operations, the second one is a specific configuration designed to make testing easier. This solution is strongly bound to the *design for test (DFT)* [2] approach.
During the *Test Mode*, scan techniques transform the sequential circuit in a combinational one in order to make the memory elements *observable* and *controllable* as for primary inputs and primary outputs: they are so called *pseudo-primary outputs*.
Between the different scan methodologies, *Full scan* is the simpler and it will be analyzed in the next subsection.

2.3.1 Full scan

As shown in the figure 2.10, during the Test mode, in a Full-scan design all the flip-flop of the integrated circuit are connected in a *scan chain* and they form a *shift register*. This configuration allow observing the internal nodes of the chip without the use of sequential testing techniques. Furthermore, the circuit design has also a special pin (N/T) used to switch between the Normal/Functional mode and the Test mode.
In Test Mode, it is possible to load any pattern in the memory elements via *Scan-In* signal (*controllability*) and any flip-flop can be read via *Scan-out* signal (*observability*). With this solution, it is not necessary the generation of sequential patterns, therefore ATPG tools are commonly used to generate test vectors for the combinational part of the circuit.
For each pattern generated by the ATPG, the test will be applied in the following way:

[2]DFT is an approach that reduces the difficulty and the time needed for testing a circuit. In order to do that, the circuit is designed including testability features

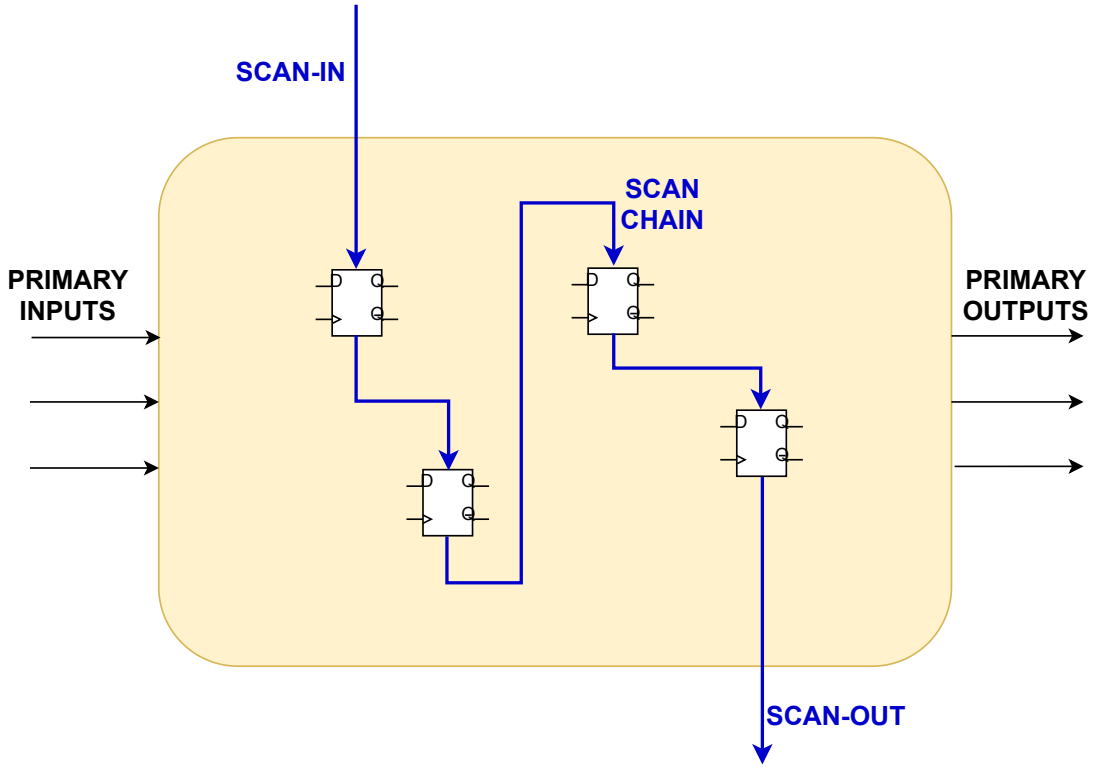

Figure 2.10: Scan chain in Test Mode

1. activate test mode and upload the PPI (pseudo-primary inputs) values (Scan-In);
2. apply the PI (primary inputs) values;
3. deactivate test mode and apply one clock to the circuit;
4. observe PO (primary outputs) values;
5. activate test mode and download the PPO (pseudo-primary outputs) values (Scan-Out) and check if the test passed

This approach is easy to implement, requires a minimal overhead and especially allow testing sequential circuit as combinational ones.

2.4 Functional testing

Functional testing is a testing methodology opposed to the usage of the DFT. In fact, this approach aims to test the functionality of the system applying suitable

stimuli to the functional inputs and observing the functional outputs, without resorting to any specific test design infrastructure of the DUT [3]. Functional tests are developed taking into account the functional information of the module therefore it aims to test the functions rather than the faults. This solution is widely adopted and is becoming a standard for *end-of-manufactoring* test and *in-field/on-line* test. It is chiefly developed as *Software-Based Self-Test* [6] [7]: the DUT is forced to run a *Software Test Library (STL)*, for example test programs, and the results produced by the execution are downloaded from the memory and compared to the expected outputs in order to detect faults.
Currently, companies exploit functional testing to test defects in the field not covered by other kinds of test, therefore STLs is considered a Functional Safety (FuSa) solution[4] [8] [9] [10]. SBST solution has several advantages, for example:

- STLs can be schedule in order to not interfere with the device normal operation: this avoids service interruptions
- do not require additional hardware

However the STLs development process is extremely difficult since faults should be propagated through the whole circuit only using the instructions of the instruction set architecture [11].

[3]Device Under Test

[4]In the automotive domain, ISO 26262 is the Functional Safety standard. It focuses on the safety aspects that should be developed in an automotive electronic system

Chapter 3

Context of work

This chapter describes the *state of the art* of the test of integrated circuits. In the last decade, several works [1] [12] [13] [14] investigated the possibility to improve test programs in order to achieve higher fault coverage.
[12] starting from verification-oriented programs[1] presents an approach that focuses on the generation of test patterns for online testing, with improvements on the fault coverage of stuck-at faults on a RISCV core. [13] and [14] propose an approach that exploits HLDDs[2] for modeling microprocessors and faults, and generates the final test program starting from a previously prepared code template. These works show how the quality of the test programs can be improved, even though they refer to the stuck-at fault model only. Other works [15] [16] [17] [18] analyse the post-silicon debug topic in order to find suitable solutions to improve the fault coverage. These works demonstrate the effectiveness of post-silicon solutions but they work on the whole core.
[1] defines the first steps for the development of a systematic methodology to improve the quality of existing STLs, aiming to increase the transition delay faults' observability. This work is the base of the studies proposed in the thesis work, since it has been developed by the DAUIN (Dipartimento di Automatica e Informatica) of Politecnico di Torino. A more detailed description of this work will be proposed in the section 3.1. Eventually, in the following sections the main features of the commercial tools and the PULPino core used in the work thesis will be presented.

[1]Programs that aim to verify that the device is compliant to the design specifications

[2]High-Level Decision Diagrams

3.1 Background works on STL analysis for delay test coverage improvement

As previously mentioned, paper [1] proposes a new approach for improving delay test coverage, starting from the analysis of existing STLs. One of the most common approaches used for the test of integrated circuits is based on Desing-for-Testability (DfT) solutions. These are based on mature and well-tested technologies, but they require area overhead and significant timing that could degrade performances. Furthermore, these solutions are often not able to deal with *functionally untestable faults* (section 2.1.2). Fuctional testing allows avoiding these problems and commonly companies provide STLs for test their products. Since the development of the test programs requires a remarkable amount of effort and moreover they should able to forward fault effects to primary outputs (POs), the work proposes a process flow in order to improve transition-delay faults coverage starting from STLs developed for stuck-at faults (TDFs and SAFs have similarities. For example, detecting a slow-to-rise TDF means detecting a stuck-at zero). To do that, it starts from a set of internal observation points (where faults can be propagated but eventually stopped, not reaching primary outputs) and tries to understand if it is possible to modify any piece of code of the STL in order to propagate the fault. Since the main goal is to find a methodology that allows propagating those fault effects that stopped before reaching the primary outputs, the work proposes to define two different groups of internal observation points (IOPs):

- *User Accessible Registers (UARs)*: registers directly accessible by the user through available instruction;

- *Hidden Registers (HRs)*: registers not directly accessible by the user (i.e. pipeline registers)

In this thesis, a hybrid approach is proposed. Starting from hard-to-test faults, the work aims to re-use the existing debug infrastructure of the core for testing purposes and aims to select specific internal signals in order to test the picked faults. Therefore, this solution combines SBST and post-silicon approaches, leading to satisfying results in terms of fault coverage. Eventually, the work proposes a study of the execution trace of the STLs in order to summarize and gather any useful information needed to develop strategies for the improvement of the test programs. This part is strictly linked to [1] and it could be considered the following step.

3.2 Commercial tool features

For the thesis work, commercial simulation tools like ZO1X [19] and ModelSim [20] has been exploited to perform simulations of microprocessors and retrieve the fault information needed for the development of the methodologies described in the chapters 4 and 5.
As shown in the chapter 2, starting from the circuit description and a list of fault, they are able to *inject* the faults into the circuit, determine which of them can be tested (and which can not) and give information about the fault coverage.

With particular options, Z01X tool can be set in order to produce a *fault dictionary* that gives the possibility to retrieve useful data about each fault: observation time, register involved, detected/not detected/not observed. From this output, with the usage of a simple parser (for the thesis work, Python language was chosen), it is possible to manipulate the data according to the specific necessities. For example, as shown by the sample file in listing 3.1, it is possible to build a json file that summarize for each fault the information about first and last observation time and the register involved. In this case, only the undetected faults have been inserted into the dictionary.

Listing 3.1: Example of fault dictionary parsed with a python script

```
{
  "R riscv_core.ex_stage_i_alu_i.U1718.A1": [
    [
      "36238ns",
      "38758ns",
      "riscv_core.id_stage_i_registers_i.mem_31"
    ],
    [
      "36318ns",
      "42438ns",
      "riscv_core.id_stage_i_registers_i.mem_30"
    ],
    ...
  ],
  "R riscv_core.ex_stage_i_alu_i.U2102.A1": [
    [
      "36238ns",
      "38758ns",
      "riscv_core.id_stage_i_registers_i.mem_31"
    ],
    [
      "36318ns",
      "42438ns",
      "riscv_core.id_stage_i_registers_i.mem_30"
    ],
    ...
```

```
27    ],
28    ...
29 }
```

ModelSim tool can be used to run STL simulation into the circuit. The execution of this task generates two files that has been used for the analysis described in the chapter 5. In particular, it produces an execution tracer of the program (example in listing 3.2: it stores information about the execution time, number of cycles, program counter, opcode and the mnemonic instruction) and an elf file that has been used to disassembly the executable and create an output file that links each executed instruction with the specific line of the source file, as shown in the example 3.3. To perform this operation, the command-line function `riscv32-unknown-elf-objdump -Sld ELF_FILENAME.elf > OUTPUT_FILENAME.txt`.

Listing 3.2: Example of execution tracer

```
1  Time          Cycles PC        Instr     Mnemonic
2  18880000         455 00000080 0350606f jal             x0, 26676
3  18960000         457 000068b4 30501073 csrrw           x0, x0, 0x305
4  19000000         458 000068b8 00000093 addi            x1, x0, 0
5  19040000         459 000068bc 00008113 addi            x2, x1, 0
6  19080000         460 000068c0 00008193 addi            x3, x1, 0
7  19120000         461 000068c4 00008213 addi            x4, x1, 0
8  19160000         462 000068c8 00008293 addi            x5, x1, 0
9  19200000         463 000068cc 00008313 addi            x6, x1, 0
10 19240000         464 000068d0 00008393 addi            x7, x1, 0
```

Listing 3.3: Example of disassembly file

```
1  ...
2  /../simple.S:77
3       190:    f3f290e3            bne t0,t6,b0 <fail>
4  /../simple.S:78
5       194:    fff00293            li  t0,-1
6  /../simple.S:79
7       198:    f1f31ce3            bne t1,t6,b0 <fail>
8  /../simple.S:80
9       19c:    fff00313            li  t1,-1
10 /../simple.S:81
11      1a0:    f1f398e3            bne t2,t6,b0 <fail>
12 ...
```

3.3 PULPino

The methodologies presented in the thesis work have been tested and validated on PULPino [21].

PULPino is a single-core microcontroller system developed by ETH Zurich and Università di Bologna. It is a 32-bit RISCV-based cores: it is configurable to use either the *RISCY* or the *zero-riscy* core.
The first one is the core that has been used in this thesis and it has the following characteristics, as described in [21]:

- in-order;
- single-issue;
- 4 pipeline stages;
- full support for RV32I[3], RV32C[4], RV32M[5] instructions sets
- can be configured for RV32F[6] instruction set

It can implement several ISA[7] extensions and has been designed in order to increase energy efficiency.
The second one, not used in this thesis context, has the following characteristics [21]:

- in-order;
- single-issue;
- 2 pipeline stages;
- full support for RV32I, RV32C;
- can be configured for RV32M and RV32E[8]

This core has been developed for those domains that require ultra-low-power and ultra-low-area constraints.

[3]base integer instruction set

[4]compressed instruction set

[5]multiplication instruction set extension

[6]single-precision floating point instruction set extension

[7]instruction set architecture

[8]reduced number of registers extension

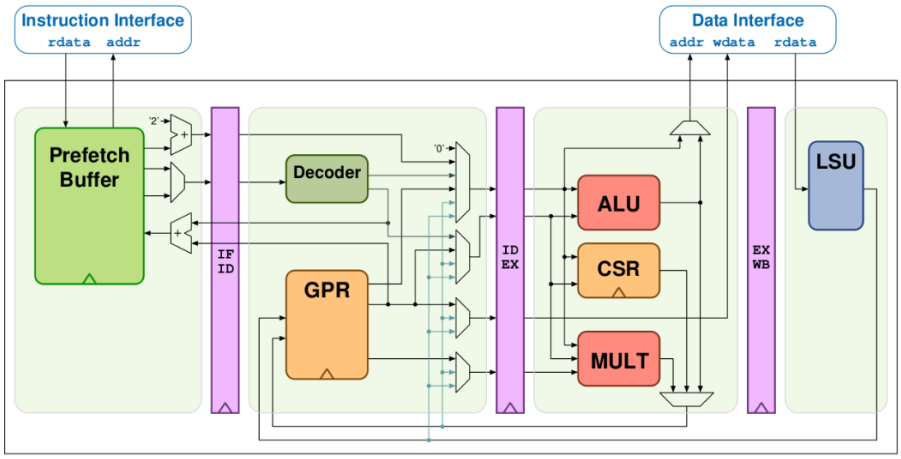

Figure 3.1: Pulpino RISCY core [22]

Chapter 4

Hybrid methodology to improve fault coverage of STLs

4.1 Introduction

As mentioned in the section 2.4, functional testing is a widely adopted solution to implement online test for safety-critical systems. In particular, Software-Based Self-Test (SBST) with STLs libraries is the most used approach thanks to its reliability and flexibility but, since the development of STLs is very expensive and requires a considerable manual effort, it is not convenient to focus on the development of new test code.
Hence the need to develop new solutions to improve the quality of the existing STLs, in order to provide higher fault coverage. In the article [1], new strategies for the improvement of test programs are proposed. Specifically, two groups of *excited* but *not observed (NO)* faults are identified:

1. faults that reach user-visible registers;

2. faults that reach hidden registers

[1] proposes solutions to cover the faults of the first group, but the ones that belong to the second are still considered *hard-to-test* faults and new solution must be elaborate.
The methodology proposed in this chapter presents an approach that allows observing the effects of the hard-to-test faults.

4.2 Methodology description

As mentioned in section 3.1, this work proposes an approach that mixes hardware and software solutions in order to observe those faults that have been previously defined as hard-to-test. In particular, as shown in the figure 4.1, the solution is composed by three fundamental steps:

1. ***identification of hard-to-test faults***: a fault simulation analysis through SBST means is performed in order to individuate the hard-to-test faults. Moreover a set of internal signals where the faults propagate is defined;
2. ***evaluation of faults***: the faults that have been individuated are evaluated in order to determine how many of them can be detected with the minimum resources. This operation is carried out by a previously created algorithm;
3. ***integration with post-silicon mechanisms***: this steps aims to support the test engineer on the development of the STLs, suggesting the signals to be observed in order to keep hardware means at a bare minimum

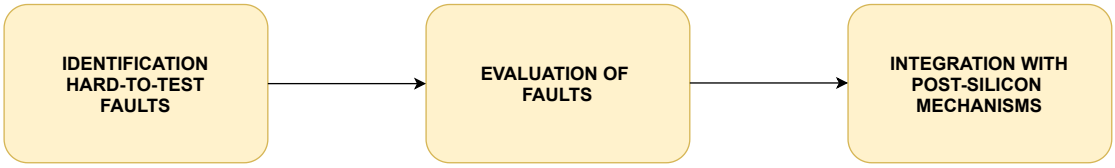

Figure 4.1: Flow of the proposed approach

For the experimental purposes, commercial tools and an already available set of STLs were used, devised for stuck-at faults (SAF) and transition-delay faults (TDF).

4.2.1 Identification of faults steps in details

The goal of the proposed solution is to analyse an STL run and identify a set of flip-flops where the effects of the undetected faults propagate. The figure 4.2 shows in details each step of the approach. In the following subsections, each of them will be analyzed, also taking into account the specific implementation. Data manipulation was developed in Python.

Sequential and Combinational simulations (steps 1 and 2)

The first two step of the process consist in two simulations, that could be runned using commercial tools:

- **sequential simulation**: test simulation performed on the whole DUT. From this, only the undetected faults were extracted;

- **combinational simulation**: test simulation performed on the combinational logic of the DUT only, using the same fault set of the previous simulation

The sequential simulation generates a .csv file in the form shown in the table 4.1. To each detected fault is assigned a progressive number and a specific name composed by ***default***, a number that identifies the fault (e.g. *_12812_)* and a final optional part (in the example in the table 4.1 can be found *stf* or *str* that stand respectively for "slow-to-fall" and "slow-to-rise", since they refer to transition-delay faults).

Fault number	**Fault name**
0	**default**_12991_stf
1	**default**_12820_stf
2	**default**_17535_str
3	**default**_17502_str
4	**default**_17503_str
5	**default**_17505_str
6	**default**_16853_str
7	**default**_16855_str
8	**default**_16856_str
9	**default**_16858_str
...	...

Table 4.1: Sequential .csv file output

The combinational simulation generates a .txt file that contains the list of all the faults that can be tested in the combinational logic of the DUT. As shown in the table 4.2, each line of the file gives information about the specific fault:

- type of fault, in the example table *stf* or *str* since they refer to transition-delay faults;

- information about the detection of the fault during the test simulation. For example, a fault can be NC (*not covered*), DR or DS (*detected*) or can have another status;

- the name of the fault that corresponds to the names of the file produced by the sequential simulation

Fault type	Status	Fault name
stf	NC	**default**_11297
str	DR	**default**_16837
stf	DR	**default**_16836
str	NC	**default**_11297
stf	NC	**default**_11296
str	DR	**default**_16798
str	DS	**default**_16722
...	...	...

Table 4.2: Combinational .txt file output

Retrieve combinational detected faults and derive undetected on sequential (steps 3 and 4)

The third step aims to select the faults *detected* during the combinational simulation (third step) in order to find the ones that were not detected in the sequential simulation (fourth step).
To perform these operation, some Python functions were developed. Selection of detected faults on the combinational simulation is performed through the function 4.1: it opens the .txt file that contains the list of combinational faults and selects the detected only (marked as *DS* or *DR*). Then it returns a list of faults.

Listing 4.1: Retrieve combinational faults function

```
def retrieve_comb_faults(file_name):
    comb_faults_list = []
    with open(file_name) as comb_file:
        file_lines = comb_file.readlines()
        for line in file_lines:
            fault_type = line.split()[0]
            detected = line.split()[1]
            fault_name = line.split()[2]
            if detected == 'DS' or detected == 'DR':
                fault = fault_name + "_" + fault_type
                comb_faults_list.append(fault)

    return comb_faults_list
```

To find the faults detected on the combinational test simulation but not on the sequential one, first it is necessary to run the function 4.2 and then the function 4.3. The first one takes all the faults contained in the .csv file and inserts them into a python list (only the fault name is retrieved), the latter finds the undetected and inserts them into a list, that will be used for point 7.

Listing 4.2: Retrieve sequential faults function

```
def retrieve_seq_faults(file_name):

    seq_faults_list = []

    with open(file_name) as seq_file:
        reader = csv.reader(seq_file)
        for row in reader:
            seq_faults_list.append(row[1])

    return seq_faults_list
```

Listing 4.3: Find undetected faults on sequential

```
def find_undetected_on_seq(seq_faults_list, comb_faults_list):

    undetected_list = []

    for comb_fault in comb_faults_list:
        if comb_fault not in seq_faults_list:
            undetected_list.append(comb_fault)

    return undetected_list
```

Bind the faults to the path endpoints and retrieve endpoint list (steps 5 and 6)

The steps 5 and 6 are performed in parallel.
Step 5 aims to link each fault to the corresponding path endpoint. In order to do that, the function 4.5 uses a text file that contains the list of paths of the DUT as shown by 4.4 and parses it to retrieve the needed information. Each path has a name that corresponds to the fault name (the fault takes its name from the path) and contains the list of signals of which it is made of. The function 4.5 for each path takes the last signal, which is the *endpoint* of the path, and links it to te corresponding fault. It returns a list of dictionaries, as shown in the example 4.6.

Listing 4.4: DUT paths file

```
$path {
  $name "**default**_1" ;
  $transition {
    "U21688/A2" ^ ; // (NOR2_X1)
    "U12981/A1" v ; // (NAND2_X1)
    "U21763/A1" ^ ; // (NOR2_X1)
    ...
    "U21732/B1" ^ ; // (OAI22_X1)
    "U12947/A" v ; // (AOI21_X1)
```

```
    "U15748/B1" ^ ; // (OAI22_X1)
  }
}
$path {
  $name "**default**_2" ;
  $transition {
    "U21688/A2" ^ ; // (NOR2_X1)
    "U12981/A1" v ; // (NAND2_X1)
    ...
    "U21732/B1" ^ ; // (OAI22_X1)
    "U12947/A" v ; // (AOI21_X1)
    "U16089/B1" ^ ; // (OAI22_X1)
  }
}
...
```

Listing 4.5: Find undetected faults on sequential function

```
def bind_endpoints_fault(file_path):
    paths_list = []
    with open(file_path) as paths_file:
        file_lines = paths_file.readlines()
        index = -1
    for line in file_lines:
        index += 1
        if "$path" in line:
            i_index = index + 4
            name = file_lines[index + 3].split()[1].replace('"', '')
            i_index += 1

            while "}" not in file_lines[i_index]:
                i_index += 1

            paths_list.append({"name": name,
                               "endpoint": file_lines[i_index - 1].
    split()[0].replace('"', '').split('/')[0]
                               })
    return paths_list
```

Listing 4.6: Output of function 4.5

```
{'name': '**default**_1', 'endpoint': 'U15748'}
{'name': '**default**_2', 'endpoint': 'U16089'}
{'name': '**default**_3', 'endpoint': 'U15748'}
{'name': '**default**_4', 'endpoint': 'U16089'}
{'name': '**default**_5', 'endpoint': 'U15748'}
{'name': '**default**_6', 'endpoint': 'U16089'}
{'name': '**default**_7', 'endpoint': 'U16071'}
...
```

The sixth step exploit the same mechanism in order to retrieve a list containing all the endpoints. It will be used in the step 8.

Bind selected faults to endpoints (step 7)

In the step 7, the tool binds each undetected faults selected in the fourth step to the corresponding endpoint (retrieved during step 5). The operation is performed by the function 4.7 and produces the output shown in 4.8. Basically the function adds information to the output of 4.5, specifying the type of the fault (for example, stf or str).

Listing 4.7: Bind undetected to the corresponding endpoint function

```
def find_undetected_and_bind_to_endpoint(paths_list, undetected_list):
    undetected_faults_paths_list = []
    for path in paths_list:
        if path['name'] + "_str" in undetected_list:
            path_tmp = dict(path)
            path_tmp['name'] += '_str'
            undetected_faults_paths_list.append(path_tmp)
        if path['name'] + "_stf" in undetected_list:
            path_tmp = dict(path)
            path_tmp['name'] += '_stf'
            undetected_faults_paths_list.append(path_tmp)

    return undetected_faults_paths_list
```

Listing 4.8: Output of function 4.7

```
{'name': '**default**_11628_str', 'endpoint': 'U18176'}
{'name': '**default**_11630_str', 'endpoint': 'U18176'}
{'name': '**default**_11632_str', 'endpoint': 'U18164'}
{'name': '**default**_11633_str', 'endpoint': 'U18164'}
{'name': '**default**_11635_str', 'endpoint': 'U18173'}
{'name': '**default**_11636_str', 'endpoint': 'U18173'}
{'name': '**default**_11637_str', 'endpoint': 'U18170'}
...
```

Retrieve wires associated to each endpoint (step 8)

This step were performed through the ModelSim commercial tool and its custom language *tcl*. The developed script, shown in the listing 4.9, starting from the list of endpoints previously generated, links each endpoint to the outputs wires of the flip-flops: these are the wires that should be observed to detect the undetected faults. The output of the function is shown by 4.10: for each endpoint is specified a set of corresponding wires.

Listing 4.9: Tcl script

```
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 ../syn_out/riscv_core_top.v
elaborate riscv_core

# Now we have to associate path enpoints to the relative flipflop output...

set endpoints_file [open "undetected_faults_paths_endpoints_file_20.txt" r]
set filename "test20.txt"
set fileId [open $filename "w"]
fconfigure $endpoints_file -buffering line
gets $endpoints_file data
while {$data != ""} {
    redirect -variable val {get_nets -of_object [get_pins -filter {@pin_direction == out} -of_object [get_cells -of_object [get_pins -filter {@pin_direction == in} -of_object [get_net -of_object [get_pins -filter {@pin_direction == out} -of_object Combinational/$data]] -leaf]]]}
    set complete_line [format {%s:%s} $data $val]
    puts $fileId $complete_line
    gets $endpoints_file data
}

close $endpoints_file
close $fileId
quit
```

Listing 4.10: Output of function 4.9

```
U18176:{cs_registers_i_PCCR_q[7] n7673}
U18176:{cs_registers_i_PCCR_q[7] n7673}
U18164:{cs_registers_i_PCCR_q[6] n7672}
U18164:{cs_registers_i_PCCR_q[6] n7672}
U18173:{cs_registers_i_PCCR_q[5] n7671}
U18173:{cs_registers_i_PCCR_q[5] n7671}
U18170:{cs_registers_i_PCCR_q[3] n7669}
U18170:{cs_registers_i_PCCR_q[3] n7669}
U18167:{cs_registers_i_PCCR_q[2] n7668}
U18167:{cs_registers_i_PCCR_q[2] n7668}
...
```

Bind undetected faults to wires (step 9)

The function 4.11 aims to link the wires computed by function 4.9 with the list of undetected faults produced by the function 4.7. The function writes into a python dictionary a collection of wires with associated every faults that they propagate. In this way each wire can be *ranked* in order to select the best set to achieve the required coverage.
In 4.12 can be found an output example of the function 4.11: for each key of the dictionary (fault name) is specified a set of of faults.

Listing 4.11: Bind faults to specific wires function

```
wire_faults_dict = {}
with open(endpoints_wires_file) as f:
    lines = f.readlines()
    for line in lines:
        if line != "\n":
            wires = line.split('{')[1].replace('}', '').split()
            for wire in wires:
                if wire != "...":
                    for u in undetected_faults_paths_list:
                        if u['endpoint'] == line.split('{')[0].
    replace(':', ''):
                            if u['name'] not in wire_faults_dict.
    values():
                                if wire not in wire_faults_dict.keys
    ():
                                    wire_faults_dict[wire] = set()
                                wire_faults_dict[wire].add(u['name'])
```

Listing 4.12: Output of function 4.11

```
{
    'cs_registers_i_PCCR_q[7]': {'**default**_11630_str', '**default
    **_11628_str','**default**_11660_str'}, 'n7673': {'**default**
    _11630_str', '**default**_11628_str', '**default**_11660_str'}, '
    cs_registers_i_PCCR_q[6]': {'**default**_11633_str', '**default**
    _11632_str','**default**_11661_str'},
    ...
}
```

4.2.2 Evaluation of faults and selection of wires

After the steps described in the section 4.2.1, it is necessary to identify a set of wires to be observed during the STL run in order to improve the fault coverage of the test. In the work thesis, two different algorithms were used to reach the goal, as shown by algorithms 1 and 2. The first algorithm (pseudo-code 1) at each

iteration selects the flip-flops which increase the fault coverage the most. Basically, to perform this operation, each time the algorithm takes the biggest remaining set of faults and removes it from the list of set available for the following iteration (during the first iteration it takes the biggest one). This solution is based on the assumption that a non-programmable hardware infrastructure, for example a MISR, can monitor a fixed number of flip-flops (selected at design time).
Being *"greedy"*, this is not the best algorithm (more optimized solutions can be found) but it gives a valid picture of the potentialities of this methodology.

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

Algorithm 1: Greedy algorithm

The second algorithm (pseudo-code 2) presents a different way to extract the needed wires. This algorithm is devise to cover two scenarios: a programmable hardware infrastructure can monitor a number of flip-flops or can monitor a number of them for some clock cycles. It produces a list of configurations, each one composed by a list of flip-flops and the time to reconfigure the hardware infrastructure: in order to cover the first of the two scenarios mentioned above, the user can omit the information about time. The algorithm, starting from a fault dictionary ordered by time, fills a configuration with flip-flops removing the faults observed by them and keeping track of the insertion time. It performs this operations until there is space in the trace buffer. Since it adopts a first-come first-served (FCFS) policy, it discards other faults captured at the same time of the last fill. When a new time is encountered, the algorithm saves the configuration and starts filling another one until it reaches the target fault coverage or the end of the dictionary. This solution is more complex than the first one but reflects the behaviour of post-silicon debug circuitry and provides more efficient and satisfying results.

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 list 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
end
return R

Algorithm 2: Variable flip-flop selection algorithm

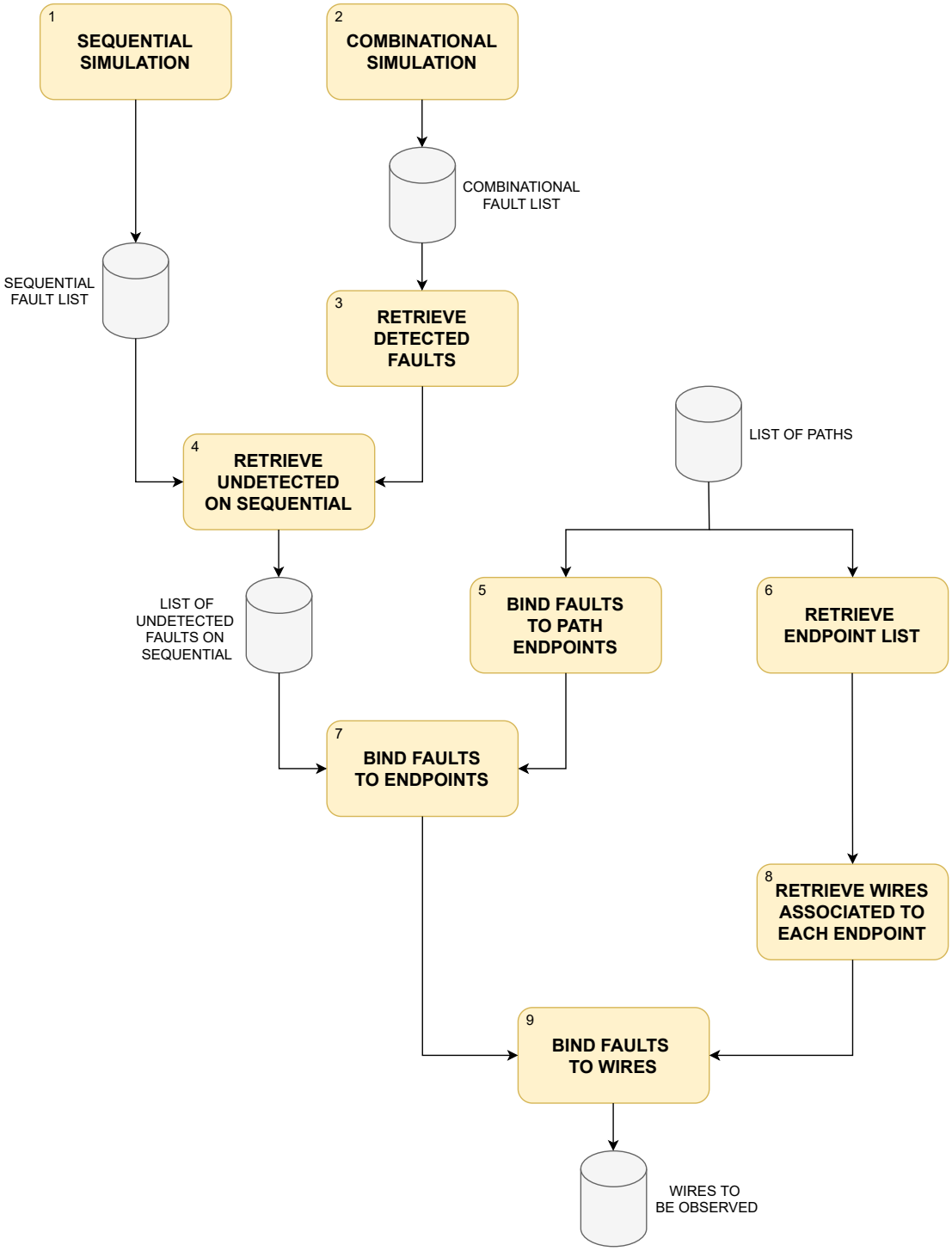

Figure 4.2: Methodology steps in details

Chapter 5

Analysis of the execution trace of test programs

This chapter describes the methodology developed for the analysis of the execution trace of the test programs.
In the context of this work, a python-based software has been built with the aim of create well-structured solution able to store all the useful information about the test program, fault associated to the program and able to propose preliminar approaches for improving the STL's quality. The test of these solutions is not the main goal of this part of the thesis, but it is the starting point of future works.
MongoDB [23] has been chosen as database for the storage of all the information.

5.1 Creation of a test program collection

Starting from the execution trace of the test program and the disassembly file described in the section 3.2, the software gives the possibility to create a *MongoDB collection* that stores all the information regarding the execution of the program.
In particular, for each executed instruction, as shown by the example in figure 5.1, it creates a document with the following data:

- *id*: automatic id assigned by Mongo to the document;
- *time*: time instant (in ns) in which the instruction was executed;
- *opcode*: operation code of the instruction;
- *instruction*: mnemonic of the executed instruction;
- *output_register*: register written by the instruction;

- *input_registers*: list of registers read by the instruction;
- *PC*: value of the program counter;
- *source_file*: name of the file of the test program that contains the executed instruction;
- *source_line*: number of the line of the *source_file* corresponding to the instruction;
- *source_instruction*: mnemonic of the instruction indicated by the *source_file* and the *source_line* (it may be different from *instruction* due to operations performed by the compiler);

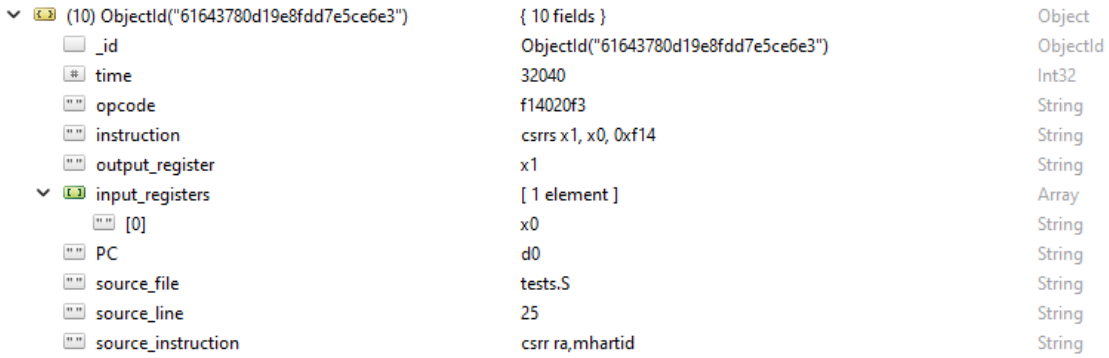

Figure 5.1: Example of document instruction inserted into the MongoDB collection

This operation is made through a python function that parses the two input files (tracer and disassembly file) and merges all the information, eventually storing them into the Mongo database.

5.2 Retrieve information from the DB

Starting from the collection inserted into the database and the fault dictionaries described in the chapter 3.2, it is possible to retrieve some information according to the necessities:

- find blocks of source code editable to cover a fault;
- find which faults *should* be coverable modifying specific pieces of source code;
- find blocks of source code editable to cover a fault and find which part of source code *may* be able to cover that fault;

This three features are interconnected and will be explained in the following subsections.

5.2.1 Find blocks of source code editable to cover a fault

In order to perform this task, the software takes as inputs a fault dictionary (in the format shown by the listing 3.1) and the test program collection previously inserted into the database. For each observation interval of each fault of the fault dictionary, the function find the piece of source code that may be edited in order to improve the test program and consequently cover the associated fault.
In particular, for each interval it search a time interval in which it is possible to operate and finds the instructions linked to the interval boundaries:

- *start time*: execution time of the first instruction executed after the first observation time specified by the interval (of the given fault of the fault dictionary);

- *end time*: execution time of the first instruction executed after the last observation time specified by the interval (of the given fault of the fault dictionary) and that writes the same register;

The choice of the time range is based on the assumption that the fault can be caught from the first detection time (it is chosen the first executed instruction after this time) until the register written by the instruction that causes the fault is overwritten by another instruction (it is chosen the first executed instruction after the *end time* and that writes that register).
For each selected time, it retrieves the information of the associated instructions, fills a list of intervals associated to the fault and creates a document to be inserted into the Mongo database. As shown by figure 5.2, each documents stores the information of the *fault*, that corresponds to the fault name specified in the dictionary, and a list of *intervals*. As mentioned before, each interval contains the following data:

- *end_tracer_instruction*: instruction specified by the execution trace, executed at time *end time*;

- *end_source_instruction*: instruction of the source file that corresponds to *end_tracer_instruction*;

- *end_source_file*: source file that contains the *end_source_instruction*;

- *end_source_line*: line of the source file that contains the *end_source_instruction*;

- *end_time*: int value of *end time* in ns;

- *start_tracer_instruction*: instruction specified by the execution trace, executed at time *start time*;
- *start_source_instruction*: instruction of the source file that corresponds to *start_tracer_instruction*;
- *start_source_file*: source file that contains the *start_source_instruction*;
- *start_source_line*: line of the source file that contains the *start_source_instruction*;
- *start_time*: int value of *start time* in ns;
- *detected_time*: first time of detection that corresponds to the first time specified by the interval in the fault dictionary.

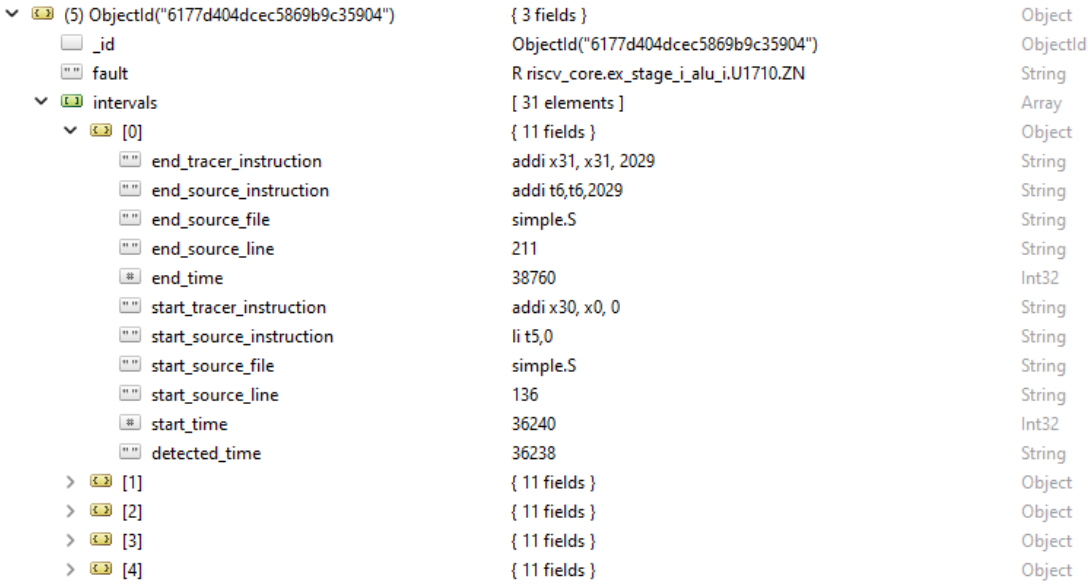

Figure 5.2: Example of stored fault information

This approach considers every fault as *coverable*. After the selection, the function stores the collection into the database and also saves it as json file.
In the next section, it is proposed a solution for the selection of specific faults that can be covered.

5.2.2 Find which faults *should* be coverable modifying specific pieces of source code

In the section 5.2.1 a simple approach for the selection of blocks of code to be modified has been proposed. Starting from these blocks, the software exposes a

function that, for each fault, selects only those intervals that may be edited in order to cover the given fault. The criteria used to perform the task can be summarized as follows:

- if the instruction executed after the *detection time* of the fault is executed within 40 ns (clock of the microprocessor) from the detection time, the interval will be picked;
- otherwise the interval is discarded

This selection is based on the assumption that, if the time of the next instruction after the detection time of the fault is within the clock time of the microprocessor, we are sure that the fault has been observed at the last cycle of the multi-cycle instruction (e.g. div) therefore it is possible to access the produced value inserting a suitable instruction (e.g. sw) immediately after that.

5.2.3 Find blocks of source code editable to cover a fault and find which part of source code *may* be able to cover that fault

This approach is an extension of the one described in the subsection 5.2.1 because it proposes a strategy for the improvement of the test program. In particular, starting from a fault dictionary in the format of the example in listing 5.1, the software searches for blocks of code to be edited in order to cover each fault (with the same process described in the subsection 5.2.1) and proposes blocks of code that can be used to edit the source file.

Listing 5.1: Example of fault dictionary for the selection of suitable blocks of code to be used for the improvement of test programs

```
{
  "F riscv_core.ex_stage_i_alu_i.int_div_div_i.AReg_DP_reg_12_.QN": [
    "53638000",
    [
      "94- riscv_core.ex_stage_i_alu_i.int_div_div_i.n468"
    ],
    [
      "1 ['55278000, G31']"
    ]
  ],
  ...
}
```

This approach tries to find a way to improve the STL for the transition-delay fault detection using pieces of code that are able to cover stuck-at faults. The solution is based on the assumption that TDFs and SAFs have similarities, as described

in the chapter 3.1. As shown in the example, for each transition-delay fault the dictionary gives information about the *detection time* (in ps), the wire in which the fault has been detected and the *time* and the *test program* that was able to detect the corresponding stuck-at fault. Starting from this information, the function finds the pieces of code to be edited and proposes blocks that can be used to modify them. Eventually, the result is saved into a json file.

5.3 Statistical analysis of the execution trace

In addition to the functionalities described in the sections 5.1 and 5.2, the software allows the user to produce statistics about the execution trace of the test programs. In particular, according to three different categories (DIV, MUL, ALU)[1], it allows the generation of statistics about:

- type of the instructions used in the test program;
- type of undetected faults;
- average number of observation intervals per fault

The aim of this feature is to analyze the STL in order to study its characteristics, find weaknesses of the test programs and consequently find new strategies for its improvement.
The python library *numpy* has been used to produce the results in the form of pie and bar charts.

[1]DIV: division instructions, MUL: multiplication instructions, ALU: all the other type of instructions (e.g. add, shift-register, or, xor, jump)

Chapter 6

Experimental results

This chapter covers the experimental results of the methodologies presented in 4 and 5. The chapter is organized as follows: in section 6.1 it is given an overview of the case study, taking into account the experimental setup used to perform the tests. Then in the sections 6.2 and 6.3 the experimental results of the hybrid methodology and the statistic results of the analysis of the execution trace of test programs are presented.

6.1 Case study and experimental setup

The methodologies described in the chapters 4 and 5 have been validated on PULPino (chapter 3.3). The DUT accounts for:

- 51,001 NAND2-equivalent gates;
- 187,857 stuck-at (SAF) and transition-delay faults (TDF);
- 1,207 flip-flops of hidden registers

6.1.1 Hybrid methodology's STLs and setup

For the first methodology, five previously developed STLs have been used: in the section 6.2 they are referenced as STL1 to STL5. These software libraries were developed for SAF detecting, therefore they show similar stuck-at coverage. On the other hand, they do not provide high TDFs coverage. Figure 6.1 presents a table that summarizes information about the STLs used for the test. In particular, the following characteristics are highlighted:

- ***Duration***: test duration in clock cycles;

- ***Size***: memory footprint of the test;
- ***Detected faults***: number of SAFs and TDFs detected by the test;
- ***Recoverable faults***: number of SAFs and TDFs recoverable with the proposed solution;
- ***Fault coverage***: fault coverage of SAFs and TDFs;
- ***Recoverable fault coverage***: recoverable fault coverage of SAFs and TDFs;

STLs general information

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

Figure 6.1: STLs information table

The fault simulations described in the sub-chapter 4.2.1 have been executed using the commercial tool Synopsis Z01X [19], which is widely adopted for FuSa. To perform the simulations, a Linux server with Intel Xeon CPU E5-2680 v3 (clock frequency up to 3.3 GHz) has been used and the task took no longer than 5 hours: in order to reduce the required time and the size of the fault dictionaries, the combinational simulation has been performed dropping each fault after 50 detections. Python post-processing required few seconds to analyse the fault dictionaries.

6.2 Hybrid methodology's results

6.2.1 Greedy algorithm's results

Figures 6.2 and 6.3, for each STL, show respectively the percentage of recovered stuck-at faults and transition-delay faults given the percentage of observed flip-flops:

- *x-axis*: observed flip-flops;
- *y-axis*: recovered faults percentage

The SAF results can be summarized as follows:

- recovering 50% of undetected faults requires to observe:

  - STL2 and STL3: 10% of flip-flops;
  - STL3 and STL4: 18% of flip-flops;
  - STL1: about 22% of flip-flops;

- recovering 75% of undetected faults requires to observe:
  - STL2 and STL3, STL5: 29% of flip-flops;
  - STL4: 38% of flip-flops;
  - STL1: 44% of flip-flops;

- recovering 100% of undetected faults requires to observe:
  - STL2 and STL3, STL5: 58% of flip-flops;
  - STL4: 68% of flip-flops;
  - STL1: 80% of flip-flops;

Recovering all the faults with the STL1 requires 80% of the flip-flops: it means that it is needed to observe 960 FF. This is the worst scenario.

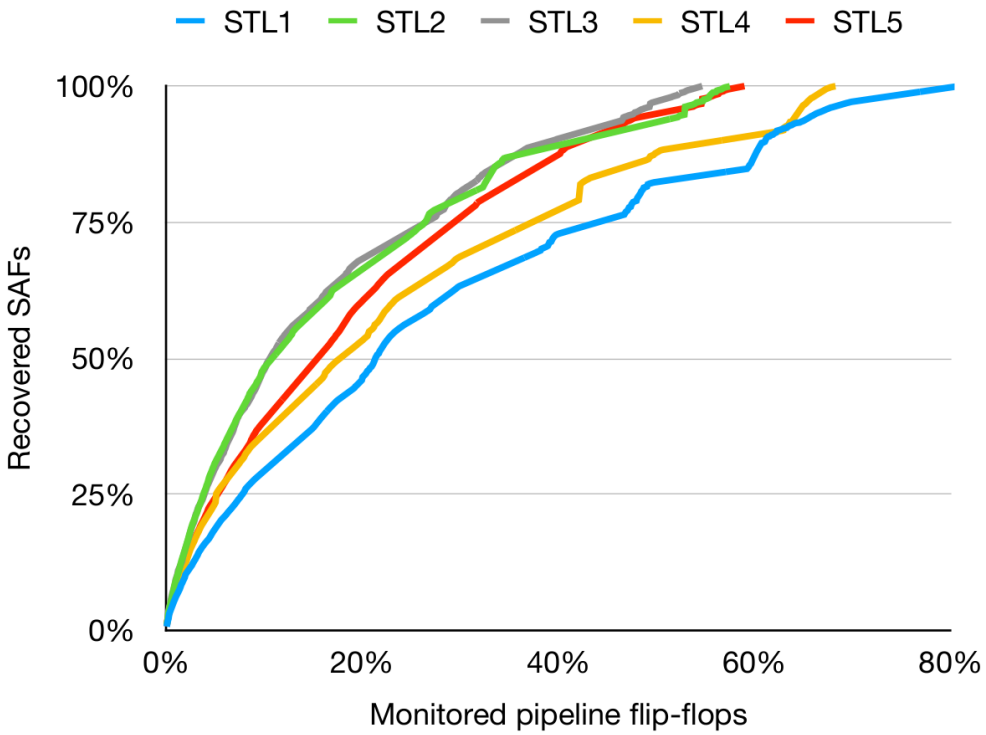

Figure 6.2: Undetected stuck-at faults recovered with greedy algorithm

Since the STLs have been developed to deal with stuck-at faults, the TDFs results, as shown in figure 6.3, are pretty different from the SAF results. They can be summarized as follows:

- recovering 50% of undetected faults requires to observe:
  - STL1: 28% of flip-flops;
  - STL3: 8.5% of flip-flops;
  - STL4: 12% of flip-flops;
  - STL5: 15% of flip-flops;
- recovering 75% of undetected faults requires to observe:
  - STL1: 50% of flip-flops;
  - STL3: 17% of flip-flops;
  - STL4 and STL5: about 30% of flip-flops;
- recovering 100% of undetected faults requires to observe a similar percentage of FF as SAF case.

In this case, STL2 shows a different behaviour. In fact, with 2.24% of flip-flops it is possible to recover about 80% of faults. However, to reach the 100% of covered is required 63.40% of FF. The worst scenario is very similar to SAF's: 78% of FF are required to reach 100% of coverage.

6.2.2 Variable flip-flop selection algorithm's result

Tables 6.1 and 6.2 show the results of the algorithm 2, respectively for SAFs and TDFs. They report the results about recoverable faults with reference to:

- *Slots*: number of cycles during which the configuration has been kept (*inf.* means that there is no fixed amount of clock cycles, therefore the configuration is kept until is necessary);
- *Bits*: trace buffer width in bits

First, it is evident that both SAFs and TDFs results show similar patterns. Especially, the results show that larger trace buffer guarantee higher fault coverage, in both cases: with 128 bits trace buffer, it is possible to recover 100% of faults in every case, except for STL3. Since the debug circuit size must be considered, 128 bits trace buffer might be too large therefore it is needed to look at the results achieved with smaller buffer size. In particular, 16 bits allow recovering more than 90% of SAFs and with 32 bits we can recover more than 90% of TDFs (except for

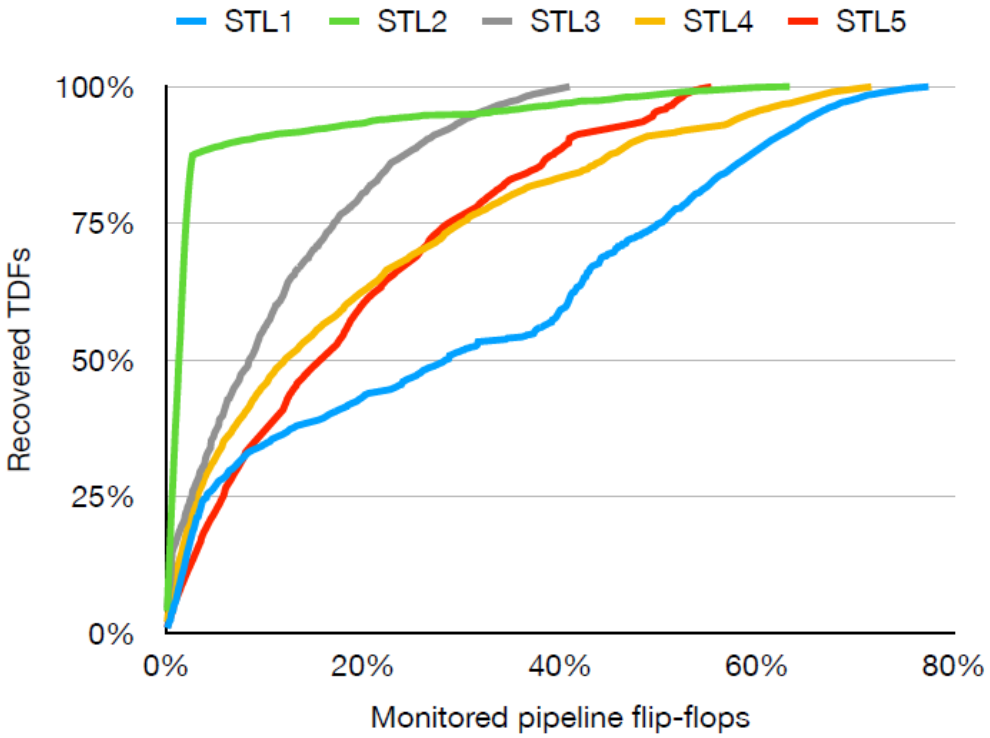

Figure 6.3: Undetected transition-delay faults recovered with greedy algorithm

STL3).
On the other hand, considering the time slots it is possible to notice an opposite trend than the one shown by the trace buffer. In fact, the fault coverage drops with higher time slots: short time slots force to more configurations therefore it is possible to observe a larger number of faults during the test procedure.
For the STL3 program, the algorithm was not able to reach 100% of faults in any case. The reason why this happened can be found on the test program structure: it provides a large amount of faults to be observed at the same time. This means that is necessary to observe a large number of flip-flops at the same time, so that, if this number is larger than the trace buffer, some faults are inevitably discarded.

The methodology presented in this section will be submitted as paper proposal to theme relevant conferences and journals.

Table 6.1: Experimental results on SAFs with variable flip-flop selection

Program	Bits \ Slots	#Configurations					Recovered stuck-at faults					Recovered stuck-at fault coverage %				
		16	32	64	128	inf.	16	32	64	128	inf.	16	32	64	128	inf.
STL1	4	464	452	444	438	429	3,793	3,783	3,781	3,785	3,756	82.80	82.58	82.54	82.62	81.99
	8	302	281	270	258	250	4,291	4,278	4,300	4,279	4,260	93.67	93.39	93.87	93.41	92.99
	16	198	171	155	145	131	4,476	4,484	4,481	4,464	4,399	97.71	97.88	97.82	97.45	96.03
	32	148	118	98	85	66	4,523	4,516	4,524	4,496	4,506	98.73	98.58	98.76	98.14	98.36
	64	126	92	71	56	32	4,581	4,568	4,531	4,536	4,541	100.00	99.72	98.91	99.02	99.13
	128	117	84	58	43	15	4,581	4,581	4,572	4,572	4,557	100.00	100.00	99.80	99.80	99.48
STL2	4	370	362	352	350	334	2,600	2,600	2,600	2,601	2,592	91.48	91.48	91.48	91.52	91.20
	8	236	223	211	203	185	2,794	2,793	2,791	2,790	2,793	98.31	98.28	98.21	98.17	98.28
	16	150	138	127	116	93	2,795	2,793	2,793	2,793	2,790	98.35	98.28	98.28	98.28	98.17
	32	106	97	85	75	46	2,841	2,841	2,841	2,841	2,841	99.96	99.96	99.96	99.96	99.96
	64	90	75	62	53	23	2,842	2,802	2,842	2,842	2,842	100.00	98.59	100.00	100.00	100.00
	128	83	67	54	44	11	2,842	2,842	2,842	2,842	2,842	100.00	100.00	100.00	100.00	100.00
STL3	4	244	243	243	242	242	1,753	1,753	1,753	1,749	1,749	75.33	75.33	75.33	75.16	75.16
	8	155	154	154	153	153	2,111	2,115	2,115	2,111	2,111	90.72	90.89	90.89	90.72	90.72
	16	86	82	81	81	81	2,236	2,212	2,213	2,213	2,223	96.09	95.06	95.10	95.10	95.53
	32	50	44	42	42	42	2,296	2,302	2,302	2,302	2,295	98.67	98.93	98.93	98.93	98.62
	64	31	24	22	21	20	2,315	2,314	2,309	2,309	2,315	99.48	99.44	99.23	99.23	99.48
	128	24	16	12	12	9	2,298	2,297	2,297	2,297	2,295	98.75	98.71	98.71	98.71	98.62
STL4	4	476	461	451	440	417	3,598	3,598	3,603	3,603	3,601	85.40	85.40	85.52	85.52	85.47
	8	319	300	284	267	238	3,983	3,983	3,983	3,978	3,969	94.54	94.54	94.54	94.42	94.21
	16	213	189	169	154	123	4,103	4,106	4,106	4,103	4,114	97.39	97.46	97.46	97.39	97.65
	32	164	140	117	102	59	4,212	4,061	4,061	4,061	4,165	99.98	96.39	96.39	96.39	98.86
	64	140	115	92	78	29	4,213	4,213	4,213	4,213	4,187	100.00	100.00	100.00	100.00	99.38
	128	131	105	82	67	14	4,213	4,213	4,213	4,213	4,213	100.00	100.00	100.00	100.00	100.00
STL5	4	461	445	427	415	385	2,995	2,995	2,995	2,994	2,993	87.78	87.78	87.78	87.75	87.72
	8	322	299	277	257	219	3,278	3,278	3,278	3,278	3,276	96.07	96.07	96.07	96.07	96.01
	16	240	212	186	167	116	3,376	3,376	3,376	3,376	3,379	98.94	98.94	98.94	98.94	99.03
	32	190	158	136	113	58	3,411	3,411	3,411	3,411	3,410	99.97	99.97	99.97	99.97	99.94
	64	168	138	116	92	29	3,412	3,412	3,412	3,412	3,411	100.00	100.00	100.00	100.00	99.97
	128	162	130	107	82	14	3,412	3,412	3,412	3,412	3,412	100.00	100.00	100.00	100.00	100.00

Table 6.2: Experimental results on TDFs with variable flip-flop selection

Program	Bits \ Slots	#Configurations					Recovered transition delay faults					Recovered transition delay fault coverage %				
		16	32	64	128	inf.	16	32	64	128	inf.	16	32	64	128	inf.
STL1	4	551	531	515	510	503	4,547	4,514	4,517	4,515	4,531	59.29	58.86	58.90	58.87	59.08
	8	359	328	311	298	288	5,473	5,434	5,427	5,383	5,384	71.37	70.86	70.77	70.19	70.20
	16	249	221	197	179	164	6,377	6,338	6,336	6,285	6,346	83.15	82.64	82.62	81.95	82.75
	32	198	159	133	112	90	7,259	7,175	7,195	7,176	7,099	94.65	93.56	93.82	93.57	92.57
	64	173	123	90	65	40	7,591	7,538	7,534	7,495	7,453	98.98	98.29	98.24	97.73	97.18
	128	166	112	75	48	16	7,669	7,669	7,609	7,589	7,503	100.00	100.00	99.22	98.96	97.84
STL2	4	936	890	865	847	835	21,877	21,652	21,492	21,163	21,137	69.81	69.09	68.58	67.53	67.45
	8	619	553	498	471	451	26,567	26,341	26,117	26,090	25,938	84.78	84.05	83.34	83.25	82.77
	16	477	395	323	278	234	29,976	29,912	29,837	29,595	29,362	95.65	95.45	95.21	94.44	93.69
	32	400	304	217	165	88	31,048	31,058	30,994	30,971	30,993	99.07	99.11	98.90	98.83	98.90
	64	378	277	191	132	30	31,272	31,215	31,182	31,177	31,197	99.79	99.61	99.50	99.49	99.55
	128	377	273	186	123	13	31,338	31,338	31,336	31,248	31,266	100.00	100.00	99.99	99.71	99.77
STL3	4	161	160	160	159	158	1,652	1,652	1,652	1,648	1,648	52.26	52.26	52.26	52.14	52.14
	8	104	102	102	101	100	2,050	2,049	2,049	2,043	2,043	64.85	64.82	64.82	64.63	64.63
	16	64	61	60	60	58	2,477	2,456	2,456	2,456	2,440	78.36	77.70	77.70	77.70	77.19
	32	44	38	38	37	35	2,797	2,801	2,805	2,805	2,793	88.48	88.61	88.74	88.74	88.36
	64	30	22	20	19	18	3,069	2,986	2,988	2,988	3,021	97.09	94.46	94.53	94.53	95.57
	128	28	16	12	9	8	3,134	3,148	3,139	3,095	3,099	99.15	99.59	99.30	97.91	98.04
STL4	4	587	560	536	507	466	3,614	3,607	3,598	3,599	3,573	72.18	72.04	71.86	71.88	71.36
	8	418	386	358	320	258	4,188	4,180	4,164	4,151	4,088	83.64	83.48	83.16	82.90	81.65
	16	331	289	249	214	136	4,616	4,603	4,581	4,561	4,441	92.19	91.93	91.49	91.09	88.70
	32	286	238	201	155	67	4,816	4,813	4,810	4,864	4,722	96.19	96.13	96.07	97.14	94.31
	64	266	220	176	133	29	5,004	5,005	4,955	4,954	4,927	99.94	99.96	98.96	98.94	98.40
	128	262	216	171	126	13	5,007	5,007	5,004	5,004	4,979	100.00	100.00	99.94	99.94	99.44
STL5	4	669	643	598	570	504	3,380	3,375	3,378	3,378	3,362	94.89	94.75	94.83	94.83	94.39
	8	457	421	365	330	246	3,514	3,497	3,506	3,507	3,489	98.65	98.18	98.43	98.46	97.95
	16	366	319	262	224	114	3,556	3,554	3,554	3,550	3,548	99.83	99.78	99.78	99.66	99.61
	32	328	277	219	180	53	3,562	3,530	3,530	3,530	3,528	100.00	99.10	99.10	99.10	99.05
	64	322	269	204	162	24	3,562	3,562	3,562	3,562	3,558	100.00	100.00	100.00	100.00	99.89
	128	319	265	202	158	11	3,562	3,562	3,562	3,560	3,562	100.00	100.00	100.00	99.94	100.00

6.3 Results of the analysis of the execution trace of test programs

The test of the solutions proposed in the section 5.2 was not the main goal of the thesis, since they are the starting point for future works.
However, as shown by the figures 6.4, 6.5 and 6.6, some sample statistics about a test program have been produced.
Going into details, figure 6.4 reports a pie chart that shows the percentage of the instruction categories of which the test program is made, according to the division mentioned in the section 5.3.
As shown by the picture, 99.15% of the executed instructions are related to the ALU unit, while the other 1% regards DIV and MUL operations, respectively 0.13% and 0.72%.

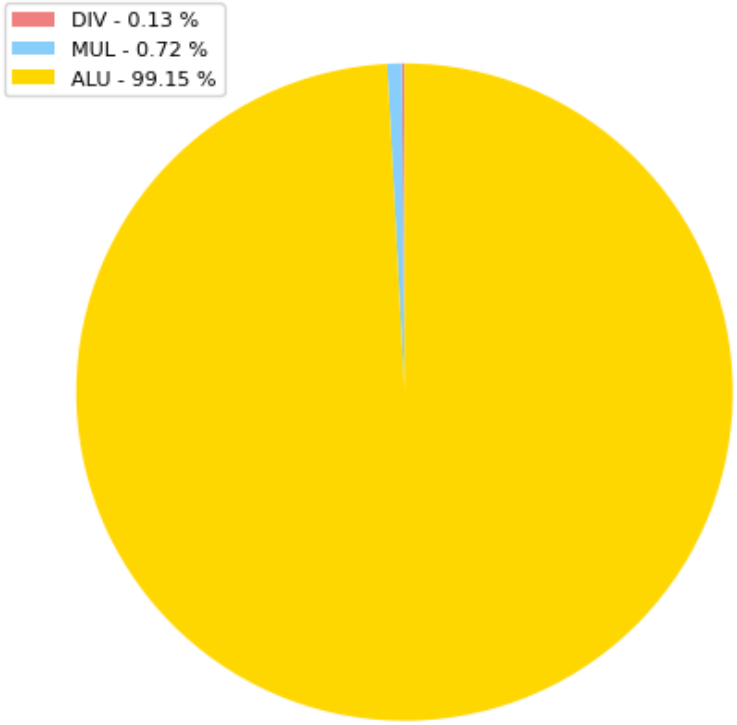

Figure 6.4: Example of percentage of instruction divided per categories

On the other hand, as shown by figure 6.5, most of the faults are linked to DIV instructions. In fact, it is possible to notice that, for the test program that have been analyzed, the faults related to DIV unit cover 81.18% of total, while ALU faults are 18.82% and no faults affect the MUL unit.

This is an interesting result since it highlights the criticality of division operations and the difficulty to observe and detect the faults that affects this unit. In fact, the majority of the DIV faults affects the unit during an internal stage of the division. This means that, in order to actually observe and detect the fault, it is needed to stop the dividing operation to observe the intermediate signal that causes the misbehaviour: this operation can not be performed, therefore a different strategy must be investigated.
However, as shown by figure 6.6, on average DIV faults can be observed in a larger number of intervals (about 27 for DIV and 3 for ALU). This characteristic can be highly useful for the test engineer since if the fault can be observed in multiple intervals, there are better chances to find a block of code to be edited in order to allow the fault detection. The test program manipulation, for example with the

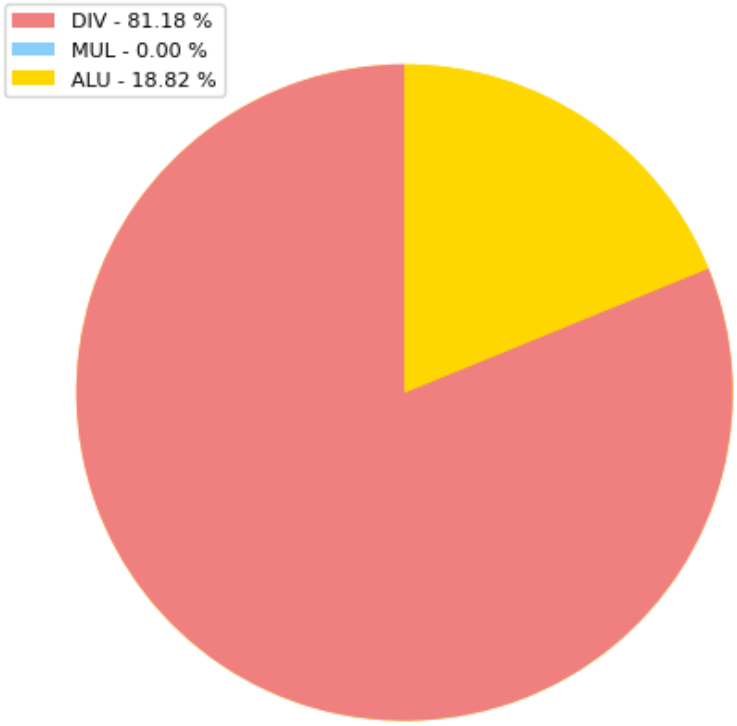

Figure 6.5: Example of percentage of faults divided per categories

insertion of new assembly instructions, might be a possible approach to bypass some of the obstacles previously mentioned.
Moreover, during the last weeks of the thesis period, several tests of this methodology have been conducted. In particular, the source code editing has been tested for the detection of faults that do not refer to the DIV unit and some promising results have been produced. Since this methodology should be improved and tested

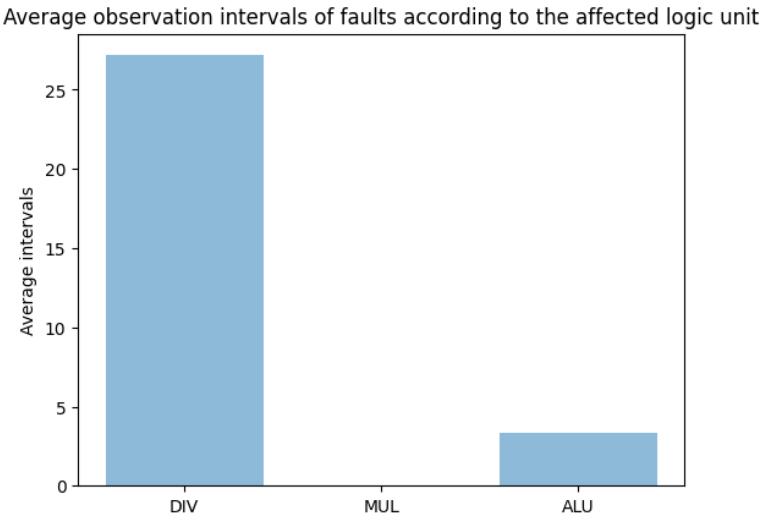

Figure 6.6: Example of average number of intervals divided per categories

with different test programs and since simulations require on average one day to complete, the results have not been reported in this work.

Chapter 7

Conclusions and related work

The thesis work proposed a systematic approach for the detection of *hard-to-test* and *not observed* faults in pipelined processors, and the development of structured software for the analysis of the execution trace of test programs in order to improve the fault coverage of STLs.
The methodology described in the chapter 4 assumes that testing is made with the usage of STLs and, starting from a set of recoverable faults (identified through fault simulation performed by commercial tools), it exploits specific algorithms in order to produce a configuration schedule for already available monitoring features (for example, trace buffers). In the sub-chapter 6.2 satisfying and promising experimental results are shown. In particular, it has been shown that a *greedy algorithm* does not allow recovering efficiently the hard-to-test faults, since it needs to monitor a massive amount of flip-flops and, sometimes, it can not reach 100% of coverage. On the other hand, an algorithm that performs a variable selection of flip-flops is able to recover 100% of SAF and TDF in almost every case and during a single STL run, using 128 bits trace buffer and a suitable number of configurations. If the size of the trace buffer is not affordable, results show that 32 bits trace buffer achieves satisfying fault coverage and it can be the best trade-off *buffer size - coverage*. Moreover, with multiple STL runs it is possible to achieve 100% of coverage in every case using smaller trace buffers.
The second part of the thesis aims to develop a systematic approach for studying the execution trace of the test programs and proposes some preliminar solutions for the improvement of the STLs. In particular, storing the information about the trace into a database has been proved to be highly useful for the workflow, since it allows scaling the software and add features avoiding the repetition of previous step.

The generation of statistic is particularly interesting since it shows the weaknesses of the STL and it can help the test engineer in the development of programs. In fact, for example, as shown in the subsection 6.3, most of the hard-to-test faults (in the example about 81%) are linked to *division* instructions and, at the same time, these faults can be observed in 27 intervals on average: this aspect can be pretty interesting since the programmer has the possibility to find an interval in which it is possible to operate and eventually detect the fault (one "working" interval is enough).
Moreover, the approaches proposed for the improvement of the test programs are a solid base for future work. First of all, the solutions proposed need to be tested in order to validate the effectiveness of the methodology, since it was not the main goal of this part of the work. Subsequently, according to the results, new solutions can be developed or the proposed ones can be improved.

Bibliography

[1] R. Cantoro *et al.* «Self-Test Libraries Analysis for Pipelined Processors Transition Fault Coverage Improvement». In: *IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS)* (2021) (cit. on pp. 1, 18, 19, 24).

[2] N. K. Jha and S. Gupta. *Testing of Digital Systems*. Cambridge University Press, 2012, pp. 382–444 (cit. on p. 6).

[3] *Wikipedia - Fault model*. URL: `https://en.wikipedia.org/wiki/Fault_model` (cit. on p. 12).

[4] *Wikipedia - Engineering validation test*. URL: `https://en.wikipedia.org/wiki/Engineering_validation_test` (cit. on p. 13).

[5] N. K. Jha and S. Gupta. *Testing of Digital Systems*. Cambridge University Press, 2012, pp. 266–313 (cit. on p. 15).

[6] M. Psarakis *et al.* «Systematic software-based self-test for pipelined processors». In: *ACM/IEEE Design Automatic Conference (DAC)* (2006) (cit. on p. 17).

[7] ——. «Microprocessor Software-Based Self-Testing». In: *IEEE Design and Test of Computers* (2010) (cit. on p. 17).

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

[9] ARM. *Enabling Our Partnership to Bring Safer Solutions to the Market Faster*. URL: `https://developer.arm.com/technologies/functional-safety` (cit. on p. 17).

[10] Microchip Technology Inc. *16-bit CPU Self-Test Library User's Guide*. 2012. URL: `http://ww1.microchip.com/downloads/en/DeviceDoc/52076a.pdf` (cit. on p. 17).

[11] J. Perez Acle *et al.* «Observability Solutions for In-Field Functional Test of Processor-Based Systems». In: *Microprocessors and Micros.* (2016) (cit. on p. 17).

[12] A. Ruospo *et al.* «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) (cit. on p. 18).

[13] A. Jasnetski *et al.* «On automatic software-based self-test program generation based on high-level decision diagrams». In: *IEEE LATS* (2016) (cit. on p. 18).

[14] ——. «Automated software-based self-test generation for microprocessors». In: *International Conference MIXDES* (2017) (cit. on p. 18).

[15] B. Kumar *et al.* «A methodology to capture fine-grained internal visibility during multisession silicon debug». In: *IEEE Transactions on VLSI Systems* (2020) (cit. on p. 18).

[16] J.-S. Yang *et al.* «Improved trace buffer observation via selective data capture using 2-d compaction for post-silicon debug». In: *IEEE Transactions on VLSI Systems* (2013) (cit. on p. 18).

[17] H. Oh *et al.* «An on-chip error detection method to reduce the postsilicon debug time». In: *IEEE Transactions on Computers* (2017) (cit. on p. 18).

[18] S. Chandran *et al.* «"Managing trace summaries to minimize stalls during postsilicon validation». In: *IEEE Transactions on VLSI Systems* (2017) (cit. on p. 18).

[19] *Z01X Functional Safety Assurance*. URL: `https://www.synopsys.com/verification/simulation/z01x-functional-safety.html` (cit. on pp. 20, 43).

[20] *ModelSim*. URL: `https://en.wikipedia.org/wiki/ModelSim` (cit. on p. 20).

[21] ETH Zurich and Università di Bologna. *PULPino microcontroller system*. URL: `https://github.com/pulp-platform/pulpino` (cit. on pp. 21, 22).

[22] Andreas Traber; Florian Zaruba; Sven Stucki; Antonio Pullini; Germain Haugou; Eric Flamand Frank K. Gürkaynak; 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. 23).

[23] *MongoDB*. URL: `https://www.mongodb.com/it-it` (cit. on p. 36).