POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

New techniques for path delay faults functional test

Supervisors

Prof. Matteo SONZA REORDA

Dr. Riccardo CANTORO

Candidate

Dario FOTI

March 2020

Abstract

With the advance in silicon technology, recent electronic devices are becoming deeply embedded in extremely dense silicon dies and are required to comply with increasingly strict timing requirements. Under these circumstances, several faults can be induced by different causes, like defects in the manufacturing processes, process variations as well as ageing, workload, crosstalk and many others.
Some of these defects can only be detected by testing the delay characteristics of the circuit, so delay fault models are assuming a much more relevant position in fault analysis.

However, delay fault models require much more computational effort with respect to simpler and more widespread fault models when producing and evaluating the test. Applying the test is also a challenge due to the cost of ATE and the complexity introduced by BIST techniques, which obliges to search for other options.
Software Based Self Test is desirable in this context because it allows to apply at-speed tests with no hardware overhead and it can be used even in situations where controllability and observability are reduced.
In the context of this thesis, new techniques for delay-oriented online functional testing for embedded systems are proposed, focusing on path delay faults in particular.

The study was conducted using a RISC-V based core as benchmark device and has led to achieve a fast and optimized flow for functional test simulations.
It has been shown that, with the current implementation, in all case studies high percentages of path delay faults detectable with scan techniques can be also detected with functional tests. Additionally, efforts have been put to identify a relationship among stuck-at, transition delay and path delay fault coverage.
Finally, some insights about the functional testability have been extracted by analyzing statistically how many misbehaviours a path delay fault should produce before being functionally observable.

Acknowledgements

After having travelled through the long and winding road that leads to his destination, just before taking the last step forward, the traveler should always stop and take a look back.
Those hills and valleys, those twists and turns, the people met in his way and the marks that they left on him will forever be the means for his mind to relive that journey.
As now I am the traveller about to reach his destination, I look back to remember each step I went through and I recognize I could never take the last step without showing gratitude to those people that helped me along the path.

I want to pay my special regards to my supervisors, Prof. Matteo Sonza Reorda who has given me the opportunity to work on such innovative ideas and has always provided me with the right advice and Dr. Riccardo Cantoro, whose expertise I admire and that has always been an inspiring guide.
My appreciation is also due to the research team of Lab 3, in which this work was developed, who have warmly welcomed me.

I would like to recognize the invaluable assistance that all my colleagues and friends provided during my study and my personal life: Carmelo, Nicola and Stefano, to whom I owe a lot, both personally and professionally, and that I consider irreplaceable; Benito, Dario, Hairo, Kevin, Sandro, Simone, that have made this years brighter and gave me the opportunity to learn a lot from them; Nikos, that has helped me in most of the hard times I experienced during this work; Davide, who has been the best housemate I could ever encounter; Antonio, Benedetto, Gloria, Luciano, Nancy, Paolo, Peppe, Santi and everyone of my friends who are spread around the world, with whom I shared all my life and contributed to make me the person I am.

Nothing of what I achieved through all these years could have been possible without the loving and tireless support of my father Sergio, my mother Enza, my sister Giada and my whole family, to which I wish to express my deepest gratitude for having shared the efforts and the struggles of this long journey .

Lastly, I want to thank my girlfriend, Federica, that deserves part of the credit for this achievement. No words exist to express how much her presence has signified to me and how vital her care was.

"Two roads diverged in a wood, and I
I took the one less traveled by,
And that has made all the difference."

R.Frost, The road not taken

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the recent years the semiconductor industry is evolving very fast and pushing hard on new technologies that allow noticeable improvements in performances of the devices with respect to the past. In particular, the manufacturing phase of the silicon dies is moving towards more complex and sophisticated processes. Most of the added complexity is related to the shrinking of the channel length of the transistors, currently set in the order of few tens of nanometers, that allow very high working frequencies and very compact and dense designs.
This advantage comes at the price of more frequent physical defects and shorter devices lifespan. ICs are nowadays more prone to manufacturing defects, process variations, ageing effects, electromagnetic interference, parasitic effects and overheating-related issues and are in general much more sensitive to degrading than older generation devices.
In figure 1.1 the trend of a typical failure rate curve is represented. With the newest technologies, infant mortality and wear out failures happen with an increasing frequency, causing the curve ends to be narrower, assuming more of a "U" shaped appearance.

This context poses new challenges in the matter of testing. Creating high quality, cost-effective tests is very hard on these devices, for several reasons.
As an example, the cost of *Automated Test Equipment* (ATE) required for production tests is growing exponentially, because of the increased precision, the stricter timing requirements and the larger amount of storage capability required.
The increased density of VLSI devices forces controllability and observability limitations and raises problems in power dissipation, crosstalk and line resistivity that could result in a premature failure of the IC or temporary malfunctioning.
Commonly used fault models (e.g. the stuck-at-0/1) are starting to show their limitations in representing the real defect coverage of a device, while more complex models that are able to represent delay defects in the circuit are becoming far more relevant and popular. Among these, it is worth mentioning:

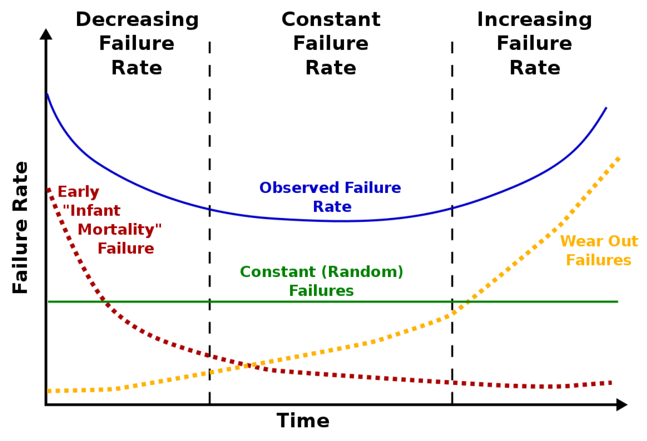

Figure 1.1: Typical failure rate curve, extracted from [2]

1. **Transition delay fault**: it models a delay defect affecting a single logical node, that causes a state transition to be slower than expected.

2. **Path delay fault**: it models a distributed delay defect affecting multiple logical nodes interconnected in sequence, that causes a state transition to traverse those nodes slower than expected.

In particular, the *Path Delay* fault model is considered to be the most accurate since it is able to catch both lumped and distributed delay defects [9].

At-speed testing is crucial for targeting these kinds of timing-related faults, but at the same time it is harder to produce suitable testing patterns. In fact, testing these faults involves finding couples of vectors able to launch a state transition and to capture it, instead than requiring a single vector to force a single value on a specific node, as it happens with the stuck-at fault model.
Moreover, since the number of faults to cover grows very fast with the size of the circuit, it is possible that a very high number of patterns should be produced to achieve high coverages; moreover, the probability of fault masking increases.

As already highlighted, external at-speed testers are not a suitable solution for their cost, while BIST techniques are quite often discarded due to the excessive area overhead, performance degradation and clock issues.
A suitable option to overcome these problems consists in relying on *Software Based Self Test* solutions, that are reliable, affordable and applicable even in situations where accessibility is reduced [9].
Currently, delay faults are usually tested resorting to scan techniques, which allow to have higher controllability and observability of the circuit and to catch a good

portion of these faults even in the case of sequential circuits. By the way, there are several downsides of using these methods, like hardware and area overhead, test-length overhead due to the time spent loading/unloading the scan chains, overtesting.
It is easy to understand that using these delay fault models requires much more efforts and innovative techniques are being developed to adapt to the industry needs.

The work carried out in this thesis falls within this framework, suggesting new ideas in matter of delay-oriented online functional testing for embedded systems.
Online testing, also called in-field test, is a testing technique that does not make use of external testers and it is often performed without removing the system under test from its operational environment. Two types on of online testing can be distinguished: concurrent, when the device is tested without stopping its normal functioning, or non-concurrent when the test is applied while the device is not performing its normal operation.
Therefore, this particular testing technique is useful since it can be applied at any time, monitoring the circuit during the whole lifespan and without requiring any kind of Design for Testability that may affect the circuit behaviour and/or performances.
Taking in consideration *Path Delay faults*, it is proposed a new approach for the fault coverage analysis of *Software Test Libraries* (STLs), that relies on the usage of different commercial software tools widespread in the industrial and academic fields. This is aimed at providing an integrated tool set for path delay functional tests simulations on both combinational and sequential netlists (with special emphasis on processors) that is able to evaluate the fault coverage while cutting down simulation time.

The tool has been evaluated using as benchmark device an open-source single-core SoC platform based on a 32-bit RISC-V core, developed by ETH Zurich and Università di Bologna, named *PULPino*. For the purpose of this thesis, PULPino was configured to use the *RI5CY* core. RI5CY is an in-order, single-issue core with 4 pipeline stages, fully supports the RV32I, RV32C and RV32M instruction sets, which respectively implement integer, compressed and multiplication instructions. It can be configured to have single-precision floating-point instruction set extension (RV32F). It implements several ISA extensions such as: hardware loops, post-incrementing load and store instructions, bit-manipulation instructions, MAC operations, support for fixed-point operations, packed-SIMD instructions and the dot product. It has been designed to increase the energy efficiency in ultra-low-power signal processing applications. PULPino has also been taped-out as an ASIC in UMC 65nm technology in January 2016 [4].

This choice has been made to have the opportunity to evaluate how the flow

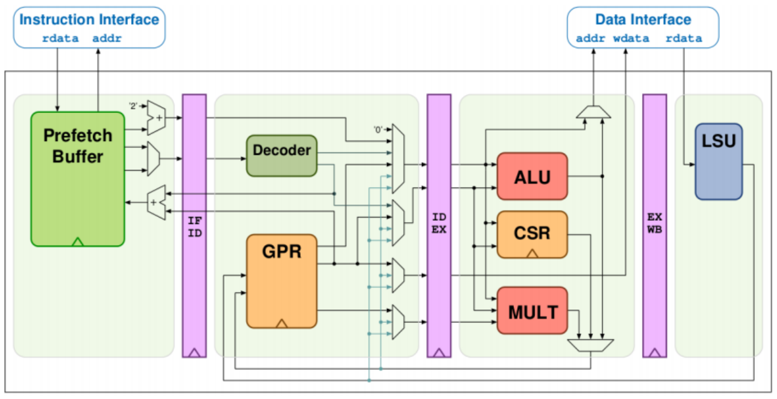

Figure 1.2: PULPino RI5CY core diagram, extracted from [1]

can be applied on a real commercial microcontroller device, whose architecture has been adopted by several companies and academic research centers worldwide. In particular, the main focus was put on testing the core by itself, isolated from other boundary components such as peripherals.

The study conducted in this thesis has led to devise a fast and optimized flow for path delay functional fault simulations, in which several arrangements have been made in order to properly evaluate the fault coverage while minimizing simulation time. It has been shown that in all case studies a very high percentage of the path delay faults detectable with scan techniques at combinational level can be also detected with functional tests. Additionally, efforts have been put to demonstrate if a correlation between stuck-at, transition delay and path delay fault coverage can be exposed.
Some insights about the functional testability have been extracted, also by analyzing statistically how many times a path delay fault should produce its effects before being observable on the *Primary outputs*.

In the following chapter the reader will find an overview on the delay fault models, with particular emphasis on path delay faults, that outlines the main theoretical aspects, distinctive features and state of the art.
In the third chapter the devised flow will be described, analysing the kind of tools that are needed for each step, the functionalities used and how those are linked to each other.
The fourth chapter is about an example of the actual implementation of the flow: strengths and limitations will be described for each tool, explaining how those have been integrated to automate the process.
The experimental results obtained through the implementation proposed in chapter four are reported in the subsequent chapter. Several test programs are evaluated on the benchmark device to measure the obtained fault coverage.

To summarize what has been obtained and how and where it can be improved, chapter six reports the main conclusions that can be drawn from the practical usage of the flow, together with some proposal on future improvements.

Chapter 2

Delay test

In a generic sequential system, input and output signals are synchronized with respect to a specific periodic signal, often referred as clock. All signals are supposed to assume a steady value within a certain time frame marked by clock signal edges, indicated as clock period. In real applications, signals may experience several transitions within a clock period, that can usually happen within a shorter time frame called transition region. For a manufactured circuit to function correctly, the transition region must not extend beyond the clock period; if this does not happen, the circuit is said to be affected by a delay fault [6].

This chapter is intended to give background on delay faults and context to the discussion that will follow in the next chapters. It is possible to find some fundamental concepts and definitions, an overview of the theoretical aspects of PDF testing, and some insights on the currently most used testing techniques.

2.1 Relevant definitions

Before diving into it, it is worth to define some useful terms that will be recurring during the course of dealing with the matter. To begin with, we can give the following definitions:

1. **Start point**: Either a primary input pin or the output of a sequential element of the design, that defines the beginning of a path

2. **End point**: Either a primary output pin or the input of a sequential element of the design, that defines the ending of a path

3. **Path**: Set of elementary combinational gates connected in a chain, starting from a *start point* and ending on an *end point*.

4. **Propagation Delay**: if referred to a *path*, it indicates the amount of time that a signal transition takes to be propagated from a *start point* to an *end point* traversing a *path*; if referred to a gate, it indicates the amount of time that a signal transition takes to be propagated from the inputs to the output of the gate itself.

5. **Arrival time**: time instant at which a state transition reaches the output of a gate or path and the output state can be considered stable.

6. **Slack**: Difference in time between clock period and *arrival time* for a specific gate or path. For the circuit to work correctly, it must always assume a positive value.

7. **Critical path**: Circuit *path* that shows the slowest propagation delay, or in other words the shortest slack.

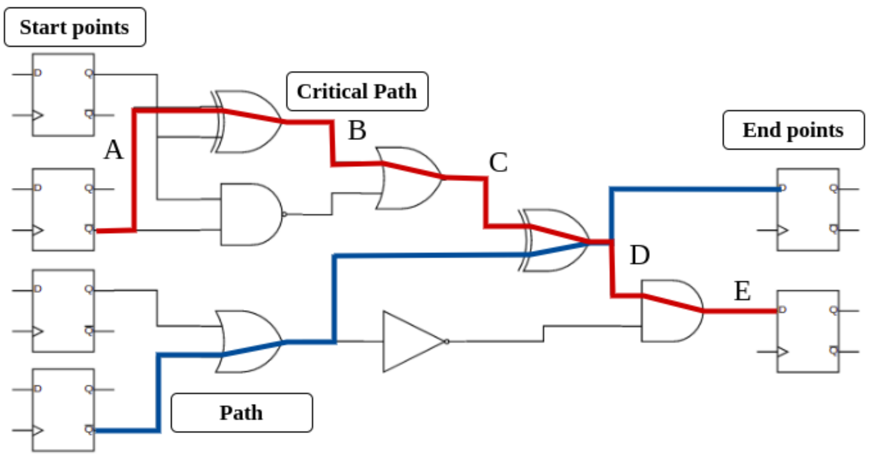

Figure 2.1: Example of sequential circuit

In fig. 2.1 it is possible to see a slice of a sequential circuit. In blue, an example of *path* is highlighted, connecting two flip-flops that act as *start* and *end* point. In red instead it is indicated a *critical path*: since it runs through the highest number of combinational cells in sequence, its *arrival time* will be one of the latest in the represented netlist, exactly equal to the sum of the *propagation delays* of the cells, considering flip-flops and interconnections as ideal.

Having clear these definitions, it is possible to introduce more appropriate definitions of delay fault models:

1. **Transition delay fault**: defect which is lumped on a single gate, that affects the nominal *propagation delay* of that gate causing a specific state transition to

happen later than the sampling edge of the clock. This usually causes incorrect sampling on the sequential elements connected to that gate, preventing the circuit from working at a specific clock frequency. In a circuit composed of n gates, the number of possible transition delay faults is equal to $2n$.

2. **Path delay fault**: defect which is distributed on the set of gates belonging to a *path*, that affects the nominal *propagation delay* of that *path* causing a specific state transition to happen later than the sampling edge of the clock. This usually causes incorrect sampling on the sequential elements connected to that *path*, preventing the circuit from working at a specific clock frequency. In a circuit including m paths, the number of possible path delay faults is equal to $2m$.
3. **Slow to rise**: often shortened in *str*, it indicates a delay fault happening when a rising transition is applied at the input of a gate or a path.
4. **Slow to fall**: often shortened in *stf*, it indicates a delay fault happening when a falling transition is applied at the input of a gate or a path.
5. **Controlling value**: it is an input value that, irrespectively of the other input values, determines the output value of a combinational element.
6. **Non-controlling value**: it indicates a value that, if present as input of a combinational gate, does not determine by itself the output value.
7. **On-path input**: Considering a gate belonging to a path, it indicates its input pin traversed by the path itself.
8. **Off-path input**: Considering a gate belonging to a path, it indicates one or more input pins belonging to the gate that are not traversed by the path itself.

Referring again to the circuit in fig. 2.1, supposing to apply the *path delay fault* model to the *critical path* in red, it is supposed that the *propagation delay* of each of the traversed gates is increased. This prevents the signal transition to be on time and causes the flip-flop to sample the wrong value.
In order to propagate the transition and test the path, it is needed that the *on-path inputs* assume *controlling values*, while the opposite has to happen for *off-path inputs*. In the next section, this concept is formally expressed and analyzed more in depth.

2.2 Path Delay Test criteria

A circuit can pass a delay test if it is capable of producing correct outputs when specific inputs are given and outputs are observed with a specific timing.

Since delay testing requires to generate and catch a state transition, test patterns must be composed of pairs of test vectors to be applied in succession.

As an example, when dealing with transition delay faults the first test vector shall be arranged such that it is able to force a certain value on the output (either a 0 or a 1), while the second vector of the pair shall force the desired gate to execute the transition while ensuring that the former is propagated up to the output. Therefore the second vector is nothing but the same vector that should be used to test a SA1 fault if a *stf* fault has to be tested, or a SA0 in case of a *str* fault.

Concerning *path delay* test in particular, it is said that a test is considered valid for a given fault if the value present at the output of the path at sample time is only controlled by the transition on the input of the path. This means that all *off-path inputs* must assume non-controlling value to make the test possible.
It is possible to define different types of tests, based on how loose the conditions applied for detection are.

2.2.1 Non-robust test

It is a test that guarantees to detect a fault when no other fault is present. This means that all *off-path inputs* assume non-controlling values only during the steady-state following the application of the second vector (a condition known as static sensitization), while during the application of the first vector they can assume whatever value.
This is the loosest requirement for a fault to be considered testable, and unfortunately it can be invalidated by the presence of other faults in the device or by glitches that may occur.

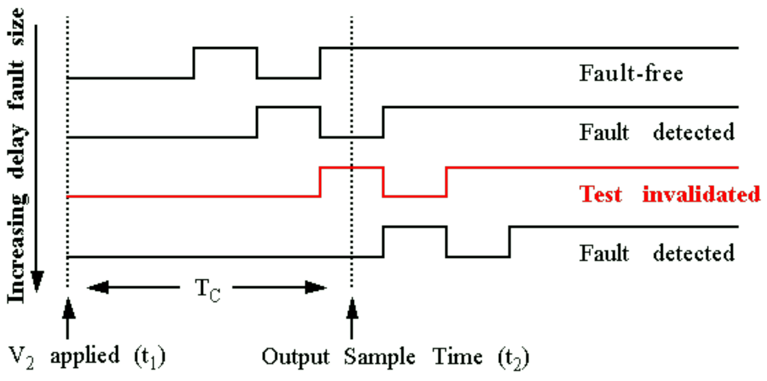

Figure 2.2: Example of invalidation of a test, extracted from [8]

For the test to be an effective measure of the path delay, the expected output value must be uniquely controlled by the transition propagating through the on-path inputs [6].

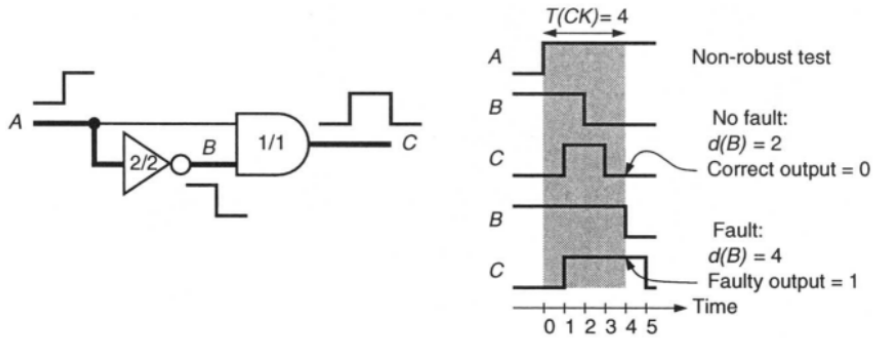

Figure 2.3: Non-robust test example, extracted from [6]

If a non-robust test exists for a specific fault, it is indicated as a *singly-testable path delay fault.*

2.2.2 Robust test

It is a test that guarantees to detect a fault even if multiple faults are present, therefore the detection is guaranteed to be not influenced by the delay distribution in the circuit.
To classify a test as robust, it has to produce real events, which basically means that different steady-state values for first and second vectors should appear on all *on-path inputs* and these events must produce controlling values for *on-path inputs.*

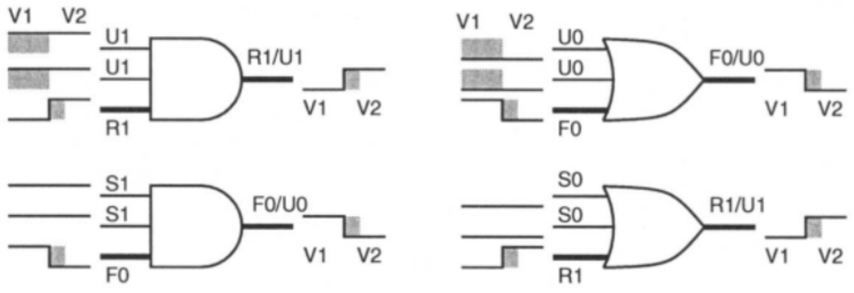

Figure 2.4: Robust test conditions for elementary gates [6]

If a robust test exists for a specific fault, it is indicated as a *robustly-testable path delay fault.*

2.2.3 Untestable faults

If none of the above definition applies to a test for a fault, it is indicated as an *untestable path delay fault*. They can be further divided into *structurally untestable faults* or *functionally untestable faults*. The first definition is used when no vector pair that allows detecting the fault can be generated; the second one is used when no functional input stimuli (e.g. a sequence of two instructions) that allows detecting the fault can be generated.
Therefore, structurally untestable faults appear to be a subset of the functionally untestable ones.

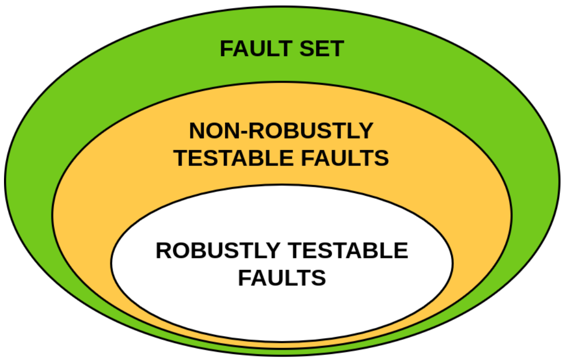

Figure 2.5: Fault classification

Figure 2.5 graphically shows what has been explained in this paragraph.
Only for a subset of faults in the set of all possible path delay faults it is possible to produce a test. Since the non-robust criteria are the least stringent in term of requirements, a large portion of faults can be categorized as non-robustly testable. Robust conditions are more strict, therefore only few faults can be defined as robustly testable. This category of faults can be seen as a subset of the non-robustly testable, since the robust test criteria can be intended as a sufficient (but not necessary) condition for a fault to be non-robustly testable.

2.3 Fault list minimization

A very important aspect in delay fault testing is the generation of the fault list. The whole process revolves around the choice of the paths that need to be tested: a larger fault list results in a wider test set, harder test generation, increased simulation times and efforts and many other side effects.
Since the number of paths in a circuit can be huge in the first place, efforts have been put in the past to find systematic ways to identify a minimum set of faults to test. Moreover, it is not rare to find a very high number of paths having comparable

properties, hence a larger portion of paths should be included in the fault analysis.

It is common to extract a list of critical paths sorted by slack through static timing analysis, but usually static analyzers perform a strongly pessimistic analysis and do not consider the functionality of the design, but just its topology.
This means that a static timing analyzer may take into account paths that exist in the circuit topology but do not allow the propagation of an event. Paths falling into this category are, generally speaking, considered *false paths*. They do not affect the operating working frequency, hence they should not be considered when setting timing constraints on the design and also for testing purposes.
For some devices or modules false paths can represent the vast majority of the paths in the circuit and may represent a great obstacle for tools, as highlighted in the case if the *ISCAS85 C6288* circuit in [15] and for several other *ISCAS* benchmarks in [7].

There exist several methods to prune the list of faults that can be considered, based on the analysis of the functionality of the design. Some of the most well-established are listed in the following.

2.3.1 Static sensitization

This method classifies a path as *statically sensitizable* if it exists at least one set of inputs that allows to propagate a transition on that path from startpoint to endpoint, regardless of the propagation delay of the gates. In other words, the input vectors can sensitize the path if in static conditions all off-path inputs assume non controlling values.

2.3.2 Dynamic sensitization

While it is possible that a path may not meet the criteria for static sensitization, it is also possible that due to propagation delays some statically unsensitizable paths may enter in the condition of propagating an event (glitch) for a short period of time. Hence, paths falling into this category are declared *statically false* paths, but are at the same time also *dynamically true* paths. This shows the limitations of static sensitization: it may underestimate the real timing requirements of a design, hiding the presence of *dynamically true* paths.
From these assumptions, a path is declared *dynamically false* if those events that may occur considering timing properties cannot be propagated through the path and possibly captured by its endpoint. Otherwise, it is declared as *dynamically true*.

2.3.3 False paths

A structural false path is identified if it is not sensitizable by means of static nor dynamic criteria because of some structural properties. An example is proposed in figure 2.6 where a path is highlighted in red. For this path a test can never be produced because no input combination exists for it to be sensitized, since the selection signals show opposite polarity. However, it may be considered critical by a static timing analyzer because the tool analysis is not aware of the functionality of the design.

We define functional false paths, instead, those paths that will never be exercised during the device mission because of some functional properties. As an example, in a processor core some opcodes may happen to be not included in the ISA. Thus, they may never be produced, preventing some structural paths to be excited.

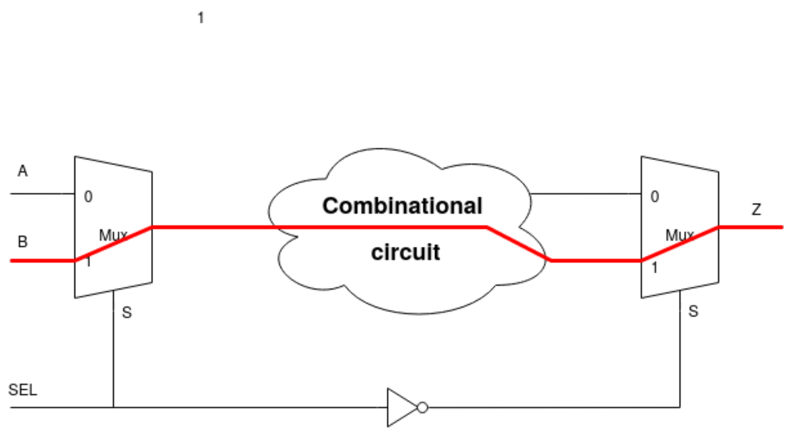

Figure 2.6: Structural false path

2.4 Delay testing techniques

The most well-established testing techniques for delay testing are based on scan. The basic idea of scan techniques is to put the design in a particular mode, often referred as *test mode*, that reconfigures the circuit so as to have more controllability and observability of the circuit.
In fact, it ideally allows to break down a single sequential circuit into several combinational ones, to apply the desired test vector to each of them by shifting it directly inside the sequential element thanks to a dedicated input pin and to extract the result after test application by shifting it out from a designated output. It is worth mentioning three different types of scan-based delay tests, which are explained in detail in the next paragraphs.

2.4.1 Launch-on shift

Often abbreviated as *LOS* it is a test that observes the following procedure:

1. The circuit is put in test mode and the scan chain is enabled; the first vector of the pair is then shifted in the chain at slow clock speed.
2. Once the first vector is fully loaded, while the scan chain is still enabled, one more shifting cycle is generated to produce the second vector of the pair.
3. The scan chain is disabled and an at-speed clock cycle is produced to capture the response of the circuit.
4. The scan chain is enabled again so as to give the opportunity to download the scan chain, while preparing the circuit for the next pattern application uploading the first vector of the next pair.

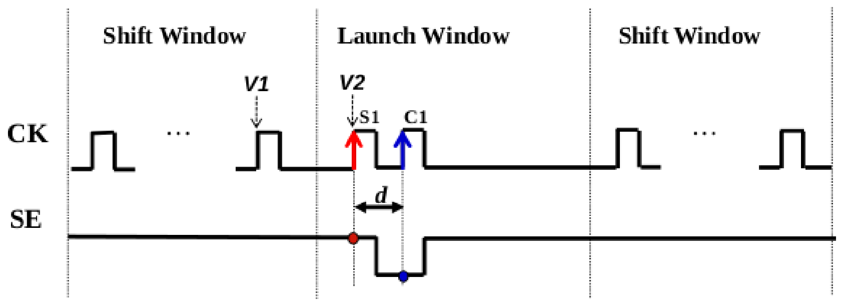

Figure 2.7: LOS procedure, extracted from [5]

It offers the advantage of reusing the same hardware needed for other scan-based tests and of having to store only one test vector per pattern, since the second test vector is just a shifted version of the first one, allowing to cut the memory footprint of the test set.
The disadvantage is that not all possible pairs of vectors can be generated and that is difficult to produce the transitions on the signal that enables the scan chain. Moreover, there is the possibility to produce overtesting, since the second vector may assume values that cannot be generated in any condition by the device.

2.4.2 Launch-on capture

Often abbreviated as *LOC*, it is a test that observes the following procedure:

1. The circuit is put in test mode and the scan chain is enabled; the first vector of the pair is then shifted in the chain at slow clock speed.

2. Once the vector is fully loaded, the scan chain is disabled and a capture cycle is generated to allow the combinational logic to respond to the first vector. This response is sampled by the sequential elements and actually serves as second vector of the pair.

3. An at-speed clock cycle is produced to capture the response of the circuit to the second vector.

4. The scan chain is enabled again so as to give the opportunity to download the scan chain, while preparing the circuit for the next pattern application uploading the first vector of the next pair.

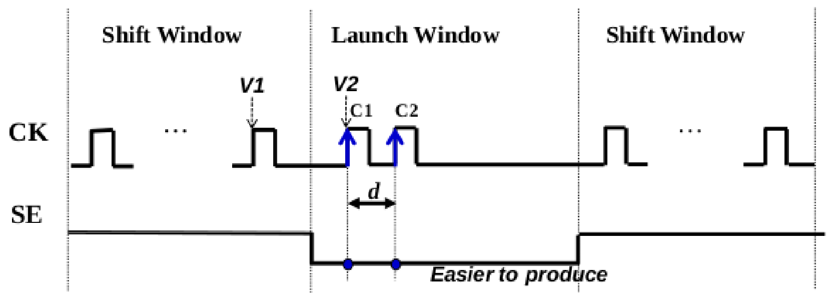

Figure 2.8: LOC procedure, extracted from [5]

It offers the advantage of reusing the same hardware needed in other scan based tests and of having to store only one test vector per pattern, since the second test vector is directly produced by the combinational circuit reacting to the application of the first one, allowing to cut the memory footprint of the test set.
The limitation of this method is that not all possible pairs of vectors can be generated.

2.4.3 Enhanced scan

The singularity of this test method is that it uses a particular hardware arrangement that enables more freedom during the test application.
In addition to the basic scan chain, it introduces a layer of latches used to hold one vector while the other is shifted in. This test observes the following procedure:

1. The circuit is put in test mode and the scan chain is enabled; the first vector of the pair is then shifted in the chain at slow clock speed. It is immediately applied and latched into the additional memory elements.

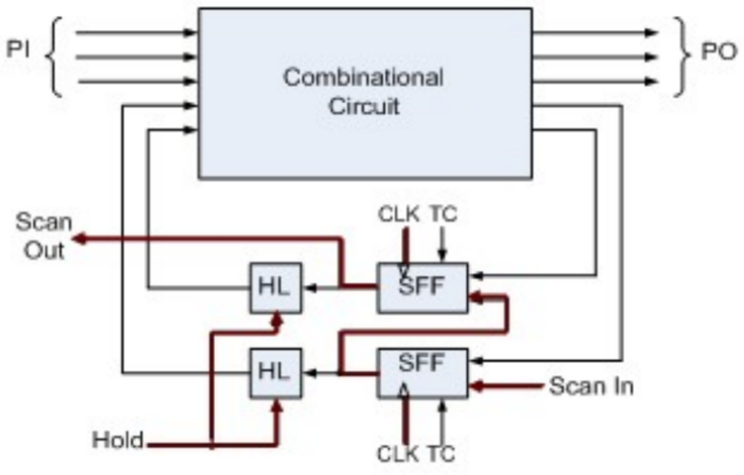

Figure 2.9: Enhanced scan circuit

2. Then, the second vector is scanned in the scan chain. Once the upload phase ends, the scan chain is disabled and an at-speed clock cycle applies the second vector and captures the response
3. The scan chain is enabled again so as to give the opportunity to download the scan chain, while preparing the circuit for the next pattern application uploading the first vector of the next pair.

It offers the advantage of having the possibility to apply any of the possible test patterns, thus maximizing the achievable fault coverage.
The limitation of this method consists in the larger memory footprint, since both vectors need to be stored in memory, the overhead in hardware requirements and the possibility to incur in overtesting, since it is possible to apply any value to the combinational circuit, even those that may never be produced by the device itself.

2.4.4 Functional testing

The state of the art in path delay fault functional testing is still in its early stages. The only consolidated methodologies to apply delay oriented tests are based on the usage of ATE and BIST or other sort of DfT, which by the way are expensive in terms of complexity, costs and performance.

Moreover, there exist articles [11] in literature that question the effectiveness and quality of scan-based delay oriented tests. In the cited article in particular, the author raises the question of what kind of defects LOC and LOS tests applied on transition delay faults can effectively detect. Particular attention should be put in applying LOS because it can potentially lead to overtesting by activating paths from non-functional states, causing unnecessary yield losses. It is then explained how these tests mainly detect open defects, missing resistive defects in vias and interconnections; the author proposes also that these kind of tests should explicitly target open defects.

Software Based Self Test has been recognized as a valid alternative to these techniques, because it can be used in stand-alone modules as well as when the processors are deeply integrated and their accessibility is reduced and implicity avoids overtesting issues, since the test vectors used can only be generated from the subset of values that the circuit could assume during its normal functioning.
The three main issues encountered in this field are the lack of software tools that allow to simulate relatively fast and effectively the test application on this fault model, to which this thesis is proposing a solution, the problem of test program generation and the problem generated by functionally untestable faults.

Some methodologies have been developed for test generation resorting to manual or automated approaches, either deterministic or heuristic, and research is still being performed on that [10].
In [9] both gate and rtl level descriptions are used to generate the test programs. The first one is used to extract the list of paths and to obtain the fault excitation conditions using Binary Decision Diagrams, while the second is used to run an evolutionary algorithm and perform fast rtl simulations useful for the automatic generation of a test program. The device used for evaluating the method was an 8051 microcontroller.
Another approach was proposed in [3], where the rtl description of two different benchmark processors, namely the 16-bit VPRO and the 32-bit DLX processor, were used to build a graph representing the pipeline behaviour with the purpose of finding the right constraints for test generation. The former was then performed resorting to ATPG techniques under those set of constraints.

The problem of test program generation is strictly bonded to the third one: identifying functionally untestable faults and removing them from the fault list may help in producing more effective and optimized self tests carefully tailored on those faults that really need to be tested, as well as their correct fault coverage evaluation.
The challenge comes from the restriction imposed from the Instruction Set Architecture, preventing some transitions to be exercised on some paths. In these cases functionally untestable faults are originated, that never determine the performance in normal operations of the circuit, and if detected during testing may lead to discard functioning chips due to overtest.
An example of applicable methodology proposed in literature is reported in [10].

Here the chosen benchmark device was the i8051 microprocessor, and the methodology was based on the analysis and simulation of the ISA to identify untestability rules (e.g. unreachable states) starting from a list of structurally testable paths and the relative test patterns.

Despite these new ideas in matter of SBST and the advantages it ensures, this technique is still not very widespread in the industry fields, but it has the potential to be employed more often in the next years.

Chapter 3

Functional Delay Testing flow

The proposed flow for path delay functional testing simulation comprises several steps, necessary to set up different aspects of the fault analysis. The steps are reported in the flowchart in figure 3.1. The purpose of this chapter is to describe the approach generically, without referring to a specific implementation or tool. Each of the phases will be explained in detail, considering how and why they should be performed and giving an overview on the required tools.

3.1 Synthesis

Assuming to have available the rtl-level description of the device to analyze, a preliminary step to apply the method is to synthesise it, which means to translate a behavioural description of the circuit into a structural one.

This process can be carried out by using software tools that take the name of *synthesis tools* and are able to produce as a result of the process a gate-level netlist of the circuit to be examined. This category of tools are in fact able to translate the high-level description of the device into a boolean network, that is then simplified by means of technology-independent and technology-dependent methods and mapped to the available set of cells. These are chosen among the ones available in the technology library, that should also be provided to the tool. The standard description languages for synthesis tools are *VHDL*, *Verilog* and *SystemVerilog*, but many tools can also work with *C* and *SystemC*.

This structural description of the target device, properly optimized according to the user constraints in terms of power, timing and area, will be necessary during every step of the applied flow. Moreover, it is useful for simplicity to produce two

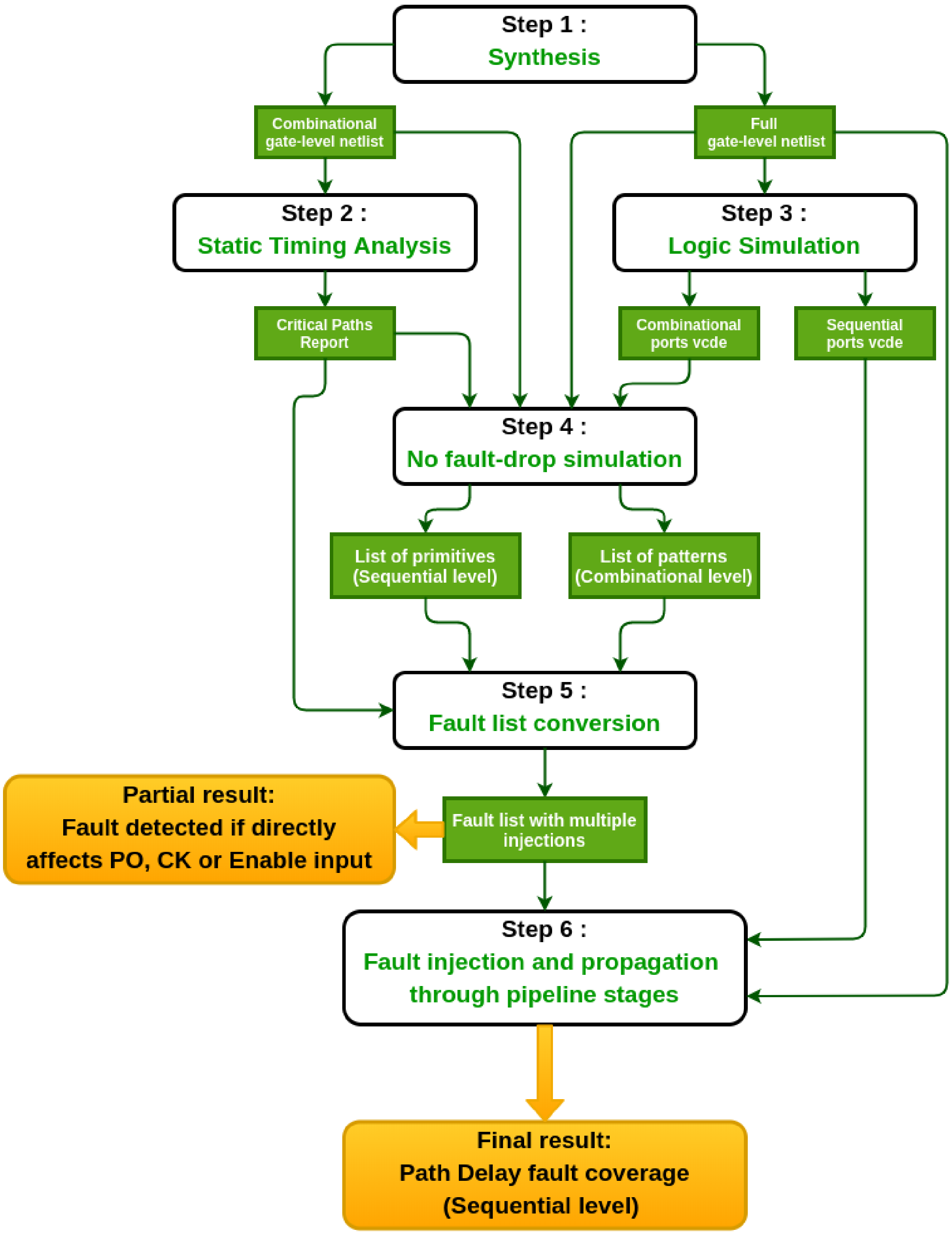
Step 1 :
Synthesis
Combinational gate-level netlist
Full gate-level netlist
Step 2 :
Static Timing Analysis
Step 3 :
Logic Simulation
Critical Paths Report
Combinational ports vcde
Sequential ports vcde
Step 4 :
No fault-drop simulation
List of primitives (Sequential level)
List of patterns (Combinational level)
Step 5 :
Fault list conversion
Partial result: Fault detected if directly affects PO, CK or Enable input
Fault list with multiple injections
Step 6 :
Fault injection and propagation through pipeline stages
Final result: Path Delay fault coverage (Sequential level)

Figure 3.1: Flowchart of the tool

netlist files: one that groups purely combinational cells, the second containing all components of the device, hence both combinational and sequential logic. The former should then replicate what's represented in figure 3.2, while the first netlist should only comprehend the section of the picture highlighted in light green. In fact, some steps of the presented approach require to work directly at the combinational level, while other take in consideration the full functionality of the IC.

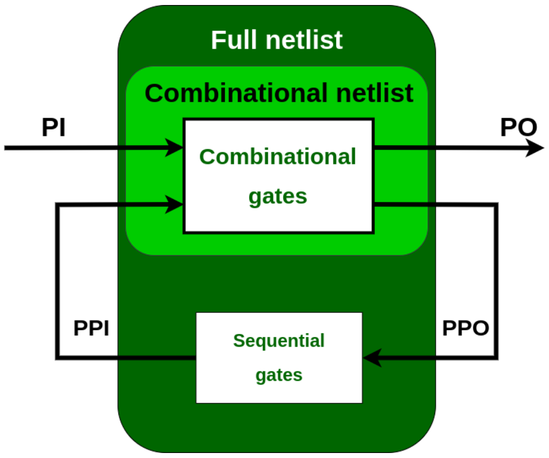

Figure 3.2: Huffman model

During the synthesis phase, once the tool has mapped the design to the standard cells available in the library meeting the specified criteria, it is possible to create a new cell group that collects all the combinational cells. Then, this can be used as a lower-level hierarchical component in the design.

3.2 Static timing analysis

The generation of the fault list is related to the topological properties of the target device. More specifically, for each path identified in the netlist, two faults will be originated, described by the imposed transition on the startpoint of the path (either *stf* or *str*).

The fault list is usually generated by running a Static Timing Analysis (STA) in order to find which paths are the most critical in the design, or in other words, which ones are to be considered in the analysis.
Software tools that serve this jobs are called Static timing analysers and are capable of analysing the design with respect to the applied timing constraints and finding the most critical sections of the design with respect to that specific metric.
The result of the STA will be a text file where topological properties of each path are described: gates traversed by the path are listed in sequence, together with

their start point and end point and the relative timing information. Paths are usually sorted by slack in increasing order.

Keeping timing information such as clock period or slack is optional during fault simulation, since the process can ignore this information to verify if the required transition can be generated and propagated through the path.
As mentioned in section 2.3, a subsequent path list reduction may be needed to minimize the number of faults that should be tested.

3.3 Logic simulation

Assuming to have the availability of different STL programs, either produced "by hand" or automatically generated by using ATPG tools or any other means, these need to be translated into patterns for the fault simulator.

Doing so requires to simulate the response of the fault-free circuit to the test program execution, and annotate the state of internal registers, input and output ports at each clock cycle. Suitable tools for this task are logic simulators, that have the possibility to dump user specified signals on text files using *Extended Value Change Dump* format (.vcde).
This kind of feature, more in detail, enables the user to record information about the value assumed by the signals at specific simulation times: in fact, each time a signal changes state, this is dumped on the file along with the time instant at which the transition occurred. VCDE files can be easily interpreted by fault simulators and translated into patterns.
Each pattern extracted from the file by the fault simulator will be no other than the result of the execution of two neighbour instructions of the test program.

For this purpose then, the gate-level netlist of the circuit must be instantiated within the testbench and signals of interest should be monitored while the simulation is running and included on the .vcde file.
Also in this case it could be handy to produce two separate .vcde files, one including all ports from for the combinational netlist and the other including the ports of the full netlist, since the flow expects to perform fault simulations at both levels.

3.4 Combinational level fault simulation

Having performed all the preliminary steps just discussed, it is possible to move on and perform the actual fault simulation. Specific software tools are required, which are referred as fault simulators. These make it possible to simulate the behaviour of the device responding to the input patterns in presence of a fault and to compare it to the expected behaviour of the fault-free IC, to identify potential mismatches.

Taking as input a structural description of the circuit, they build their own internal representation which is replicated many times to parallelize the simulation effort. For each fault it will be created one faulty replica of the device, such that only one fault at a time is active during each simulation.
The signal transitions listed in the .vcde file are read by the tool and translated into couples of test vectors; the path definition files is also read by the tool and transposed into a fault list.
The tool will then classify each of the added faults according to the outcome of the simulation. The most common classes to which a fault can be assigned to are:

1. **Detected** : the fault has been excited and has generated a detectable difference at an observation point with respect to the fault-free circuit;

2. **Not detected** : the fault has been excited but has not generated a detectable difference at an observation point with respect to the fault-free circuit;

3. **Not controlled** : the fault has not been excited by the test vectors used, but it could possibly exist a combination of vectors able to excite and detect it;

4. **Undetectable** : the fault cannot be excited or propagated to the observation point due to structural reasons.

In order to find all possible test patterns for the analyzed faults, it is useful to enable the *no fault-dropping* feature during this fault simulation: enabling it will ensure that, once a detecting pattern for a specific fault is found, it will not be removed from the fault list, but it will be still considered in the analysis. In case this option is not available, it is possible to implement some workarounds, as shown in the implementation described further in this thesis.
It is important to point out the main purposes of the combinational level fault simulation:

1. **Identify whose paths can be sensitized by the test program**.

2. **Verify the fault coverage** provided by the test program in use at combinational level.

3. **Identify which and how many patterns are able to detect each fault**.

Achieving a detection, at this stage of the process, signifies that a mismatch has been successfully propagated to one of the PPO primitives (e.g. a pipeline register of the processor core) and that this has been captured by the former.
Unfortunately, this is not sufficient to classify the fault detected also at sequential level, or in other words functionally detected. In consideration of the fact that this test flow is structured for functional testing, the netlist is not supposed to be

provided with scan chains at all, or at very least it is supposed not to use those. This implies that the only opportunity to detect and observe faults is to experience the misbehaviour caused by the fault on one of the PO (e.g. the main memory port).

Since it is not given that a fault causing misbehaviour on a given PPO can produce similar occurrences also when observing PO only, a further effort is necessary to complete the process. It consists of propagating the fault through the sequential elements, which in the case of a processor core means traversing the pipeline stages, until reaching an observation point, that could be represented by the main memory data or address port.
The reason for a fault to never be observable at *PO* can be found in how the test program is devised (e.g. absence of load/store instructions, such that the fault can never be observed in memory; data dependencies that cause fault masking) or to device structural properties (e.g. structural fault masking).

3.5 Sequential level fault simulation

From what has been described in the previous paragraph, it is possible to understand that propagating the fault effect until reaching the POs plays a very important role in the functional fault simulation process. This can be done either with the same fault simulator tool used at combinational level if possible, or with other fault injecting methods.

Most of the currently available commercial fault simulators provide just a small support over path delay fault testing, in particular when the target devices are not provided with scan functionalities, since this model is not as popular as others. It may be possible to work with a combinational slice of the circuit, but in most of the cases the analysis boundaries can't be expanded further.
Two working modes are generally supported: *Basic Scan* mode which implies the use of scan-based procedures and the *Fast Sequential* mode, an emulation of sequential circuits working just for small time windows composed of a limited amount of clock cycles. In *Fast Sequential* mode, once both vectors have been uploaded and applied, the circuit is able to work in operating mode for a fixed amount of cycles; after that, data is download from the scan chain. Unfortunately, this solution could result impractical when a full test program has to be simulated, since their duration can reach hundreds of thousands of clock cycles.
At the current state of the art, no tool supports true *Full Sequential* mode, that is a purely functional simulation in which every consecutive couple of instructions $(inst_i, inst_{i+1})$ constitutes the vectors couple necessary to test path delay faults.

In case a different fault injection method is chosen, the fault detecting patterns that have been individuated in the previous step can be used for this purpose:

focusing on those faults that have been detected at combinational level, patterns can be used for setting up multiple fault injections at several time instants, adding up multiple fault effects and maximizing the probability to detect the faults on the primary outputs. Different solutions can be used, based on software (fault simulators) or hardware (FPGA) tools.

Chapter 4

Flow implementation

In order to disclose a proof of concept of the devised flow, it has been decided to implement it using well-established software tools that are cutting edge in their field of application and are considered a standard in most of the cases.
This choice has given the opportunity to validate and optimize the flow obtaining robust and trusted results. Moreover, it ensures the repeatability of the results.
The tool suite used is based on *Synopsys* softwares, which is one of the leading companies in EDA tools when dealing with advanced IC design and verification. Unless differently specified, in this chapter it is assumed that the mentioned tools are produced by it.

The functional testing flow has been automated by integrating all these software by means of *Python* and *Bash* scripts, which effectively allowed to wrap everything as it was a single tool aimed at functional test of path delay faults. Several measures have been taken in order to give a flexible and expandable shape to the implementation.
As stated in chapter 1, the benchmark device chosen for validating the implementation is a single-issue in-order RISC-V based processor core, composed of 4 pipeline stages.

In the paragraphs below some insights about the tools and how the implementation was done are given.

4.1 Synthesis : Design Vision

The *PULPino* core has been synthesized by means of *Design Vision*, a very popular and de-facto standard synthesis tool. The design was mapped to standard cells taken from the *Nangate open cell library*, which is an open-source standard cell library produced by *Silvaco*. This technology library is one of the most used libraries for independent EDA flow testing and academic research.

The synthesis process follows this structure:

1. **The design is analyzed** by the tool, which checks for syntax errors in the *HDL* description and translates each module in an intermediate representation.
2. **The design is elaborated**, meaning that all the analyzed modules are built and connected together, mapping it to a generic gate library.
3. **Timing constraint are applied to the design**, setting the clock period, uncertainty, flip-flop internal delays, cell loads and so on.
4. **The netlist is ungrouped to remove the hierarchy levels and synthesised**. This is where the design is mapped to the *Nangate open cell library* cells.
5. **The netlist is then regrouped to create a module containing all the combinational cells of the design**, which is at the bottom of the hierarchy, instantiated as a cell inside the top level module, that will comprehend also sequential cells.
6. **The verilog gate-level netlist is extracted** for both combinational and top level modules, together with .sdf and .sdc files.

A graphic representation of the obtained hierarchy can be evinced by figure 3.2. The *Standard Delay Format* file (.sdf) and the *Synopsys Design Constraint* file (.sdc) are used to annotate cell delays and the set of constraints that were imposed during the synthesis.

4.2 Static Timing Analysis : Primetime

To perform static timing analysis on the core, it was chosen to adopt *PrimeTime*, that is a suite of tools used not only for timing, but also for signal integrity, power-aware and variation-aware analysis.

The flow used is fairly simple and linear:

1. **The combinational gate-level netlist is read**, together with the technology library.
2. **The .sdc and .sdf are read**, to allow the tool to apply the same conditions and constraints applied during the synthesis process.
3. **A report of the most critical paths is generated** for each path group in the design.

Since these reports have to be imported after by the fault simulator, a special command is used to produce them in a compatible format for the fault simulator without the need for post-processing, which is included in the *PrimeTime* distribution.
The list of paths has been taken as it stands, so no pruning or other advanced techniques have been applied to recognize the presence of false paths. In any case, the proposed flow accepts any paths list, hence every possible optimization or subset extraction is allowed.

No specific target module was identified when performing the STA, but instead the STA parameters have been set in order to have an homogeneus distribution of paths through the whole core, ranging from the most critical to the least critical ones.

4.3 Logic simulation : Questasim

Questasim is an HDL simulation environment produced by *Mentor Graphics.* It is part of the *Questa Advanced Functional Verification Platform.* It supports multiple HDL languages as *VHDL, Verilog, SystemVerilog* and provides very powerful and advanced verification solutions.

For the purpose of this thesis, it was just used to simulate the core behaviour when subject to the execution of various test programs and to dump the state of the circuit at each clock cycle. The process is based on these steps:

1. **Compile** the test program.
2. **Read the netlist and testbench files** and build a model for the simulation.
3. **Run the simulation** until a stop condition is reached in the testbench. Meanwhile, dump the input and output port values at both combinational and sequential level.

This process has been performed each time a new test program had to be analyzed, simply loading a new assembly file into the testbench and repeating the previous steps.

4.4 Combinational fault simulation : Tetramax

For the fault simulation, it has been chosen to rely on *Tetramax*, which is a fault simulator that provides different functionalities for this fault model. It is in fact designed to work with scan-based devices; it also supports the different test procedures described in the previous chapters. It is able to provide very fast path delay fault simulations with accurate results, and gives the opportunity to produce several report that allow a deep analysis of the results.

Since no scan chains were included into the benchmark device, Tetramax was used only to perform path delay fault simulations at combinational level. Unfortunately, for this fault model the native no fault-dropping feature was not activable. This limitation has been addressed applying the following strategy:

1. **Perform a first fault simulation at combinational level** in order to identify which faults can be detected by the test program in use.
2. **Divide the test program into sections**, each of them containing a subset of the whole patterns set, according to the detecting patterns found in the previous step. This can be achieved by setting a breakpoint into the pattern list for each detecting pattern found in the first simulation. Then, each section will be identified by two successive breakpoints.
3. **Implement a simulation manager in Python** that, given each section, re-simulates a subset of the detected faults within the range of patterns included in the section. The set of faults to be included is based on the first pattern of the section to simulate. In particular, only faults that have been detected by a pattern previous to the start of the section in the first place are selected, because later ones are certain to produce no detection within that range. If detection is found, a new smaller section is produced and appended to the list of simulations to perform. If no more detection is found, the process reaches its end. This basically replicates the no fault-drop functionality.
4. **After section simulation, collect all the detecting patterns** and associate them to faults in an appropriate data structure for future post-processing.

In algorithm 1 it is reported the pseudo-code for this phase that better explains the procedure.
Since the no fault-drop feature plays such an important role in this context, efforts have been put to highly optimize it and provide useful functionalities to the final user.

In fact, the algorithm can be ideally portrayed as an exhaustive search based on a *divide and conquer* approach. Even if the simulations are performed on relatively few patterns, the algorithm may need thousands of those simulations before completion.
To speed up the process, the user has the ability to spawn several concurrent processes internally managed by the simulation manager script. Each of those can access to shared memory objects and can run simulations independently from each

Algorithm 1 No drop simulation

```
if $dump\_state\_file$ is $not(empty)$ then
    $current\_state \Leftarrow restore\_state(dump\_state\_file)$
    $flist \Leftarrow current\_state.flist()$
    $sim\_set \Leftarrow current\_state.sim\_set()$
    $next\_sim\_set \Leftarrow current\_state.next\_sim\_set()$
    $fault\_lut \Leftarrow current\_state.fault\_lut()$
else
    $flist \Leftarrow read\_paths(paths\_definition\_file)$
    $first\_results \Leftarrow run\_comb\_sim(all\_patterns)$
    $splits \Leftarrow list\_detecting\_patterns(first\_results)$
    $sim\_set \Leftarrow create\_sections(splits)$
    $fault\_lut \Leftarrow create\_lut(flist, splits)$
end if
$child\_procs \Leftarrow create\_procs(max\_processes)$
while $sim\_set$ is $not(empty)$ do
    for $section$ in $sim\_set$ do
        $results \Leftarrow child\_procs.simulate(section, fault\_lut)$
        if $detection$ in $results$ then
            $flist \Leftarrow update\_fault\_list(results.detected())$
            $splits \Leftarrow list\_detecting\_patterns(results.new\_sections())$
            $next\_sim\_set \Leftarrow create\_sections(splits)$
        end if
        if $elapsed\_time() > seq\_sim\_interval$ then
            $seq\_results \Leftarrow run\_seq\_sim(flist)$
            $flist \Leftarrow drop\_faults(seq\_results.detected())$
        end if
        $dump\_state\_file \Leftarrow dump(flist, sim\_set, next\_sim\_set, fault\_lut)$
    end for
    $sim\_set \Leftarrow next\_sim\_set$
end while
```

other, thus shaving a huge slice of simulation time from the total, proportional to the number of processes used.
In order to minimize the overhead introduced by the management of large fault lists, internal data are represented and organized as *Pandas* data frame objects. Pandas is a python module dedicated to data frames management and in the context of this thesis has allowed to use several optimized methods and functions needed for searching, merging, updating and in general keeping track of the fault record modifications throughout the simulations.

Moreover, since a huge part of the simulation time is spent in reading the path definition files and the fault definition files, a strategy has been implemented to add the least amount of paths and faults in the fastest way.
This is based on the lower boundary of the set of patterns that has to be simulated: when the simulation manager starts its analysis, it builds a lookup table where each fault is matched to the pattern that has first detected it. This avoids to read files at each simulation in favour of using internal variables, which is much faster. Then, this table is used to add only those faults that have been first detected by a pattern that precedes the start pattern of the current simulation, as outlined at page 29.
Another interesting feature is that the algorithm is implementing checkpointing: it is able to capture the state of some crucial internal variables and to store it at regular intervals. Again, being such a time consuming task, this feature allows to stop the simulation and recover it anytime automatically after restarting the tool, both if the stop is intended or if it happens for some unwanted events.
Since as a general rule the more patterns are collected during this phase, the more fault injections can be performed next, the higher the probability to detect the fault at sequential (functional) level, faults are never dropped from the fault simulation. Instead, a very peculiar strategy has been adopted.
The simulation manager script is capable of launching the fault injection of faults detected at combinational level at regular intervals of time, parameter that can be tuned by the user, and to drop faults from the combinational simulation in case of functional detection. This allows to shorten the simulation time by avoiding unnecessary overhead caused by resimulation of faults, dropping those at the first occurence of functional detection.

4.5 Sequential simulation : Z01X

As mentioned in the previous chapter, the last task to perform to achieve a functional coverage consists in propagating the fault effects to the PO. This task has been implemented thanks to the adoption of *Z01X* fault simulator.

This is one of the newest software tool provided by *Synopsys* and it is designed for functional fault simulations on safety-critical automotive electronic systems. Its purpose is to enable its users to meet the fault injection testing requirements listed in several safety standard, such as the ISO 26262 for automotive and the IEC 61508 in the industrial field. It ensures very fast fault simulations, also when dealing with thousands of multiple fault injections [13].
Its limitation in this context comes from the fact that it doesn't support the path delay fault model. Fault effects can, however, be replicated using the transient fault model.

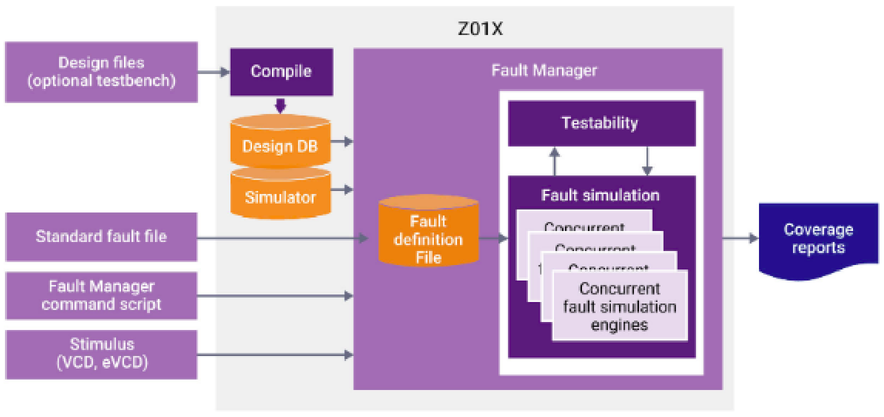

Figure 4.1: Z01X flowchart, extracted from [13]

Assuming that the increase in delay resulting from the presence of a path delay fault causes the endpoint of that path to sample the opposite value with respect to the fault-free circuit, the fault effect has been modeled as a bit flip of the duration of one clock cycle affecting the output of that specific endpoint. This replicates the lack of transition at the end of the path, basically forcing the sequential element to keep the value sampled at the previous clock cycle.

Applying this strategy has a downside: faults are described in a very different manner and some post-processing needs to be applied to keep coherence between combinational and sequential fault simulation.
In fact, a path delay fault in Tetramax is represented by the path topology by itself, and could be derived in two possible transitions, namely slow-to-rise and slow-to-fall. As indicated at page 29, to each fault is associated a set of functional patterns, that represents a set of couples of vectors that test the fault. On the other hand, a faultin *Z01X*, according to the implementation proposed, has a specific syntax that is explained in the following:

<FS> <FT> (<IT>) { [<PT>] "<PN>.Q" + [<PT>] "<PN>.QN" }

More in details, regarding the syntax:

- **FS: fault status.** Within the fault list file has to be set as NA (not attempted). Its status will change after simulation according to the results.
- **FT: fault type.** To specify a toggle transient fault, the syntax requires a "∼" character.
- **IT: injection times.** It is a list of time instants in which injections will be performed.

- **PT: primitive type.** Sequential elements are identified by the keyword "FLOP".
- **PN: primitive name.** The name of the sequential element on which injections are performed, according to what indicated in the netlist.

The part of the string that follows the "+" symbol is optional, and it is used only in the case the sequential element has both direct and inverted output pins.

The fault list conversion requires to:

1. **Associate paths to respective endpoints**. An endpoint can either be a PO, a latch or a flip-flop. It is important to also identify which input pin of the endpoint is affected by the path: this is helpful in reducing the fault list. For example, if the fault is affecting a PO of the circuit, it can be considered functionally detected without simulating it, so happens if the path is feeding an enable signal or a clock input.
2. **Convert the patterns into time instants, since fault injection is time based**. This can be obtained by knowing the clock period applied in simulation and the number that identifies the pattern. Beware that the injection is performed on the output of a sequential element, so if the detection at combinational level has occurred at clock cycle *i* corresponding to time *j*, injection should be done at clock cycle *i+1* corresponding to time *j + clock period.*
3. **Identify equivalent faults after conversion**. It may happen, in fact, that two paths share the same endpoint and are detected by the same patterns. This produces two identical faults during the conversion, even if the faults were originally distinguishable at combinational level.
4. **Remove equivalent fault from the fault list**. It can be assumed that the detection of one of the replicas implies also the detection of all the others.

Therefore, the whole sequential simulation phase is conducted as follows:

1. **Post-process the fault list inherited from the no fault-drop simulation** individuating the correct endpoints for injection and associate patterns to time instants; then eliminate equivalent faults. Only faults detected at combinational level are considered.
2. **Create a new fault list according to the *Z01X* format**, where a single transient fault (bit flip) is injected multiple times on the output of each primitive for the duration of one clock cycle.

3. **Simulate the faults** on the target *Top Level netlist* monitoring POs only. The top level .vcde file is needed for this fault simulation.

4. **Post-process the output reports** in order to match detected faults with path delay faults.

In algorithm 2 it is reported the pseudo-code of the algorithm.

Algorithm 2 Sequential simulation

```
$prim\_list \Leftarrow get\_prim(Top\_Level\_Netlist)$
$timing\_list \Leftarrow get\_timing(Report\_pattern\_file)$
$comb\_flist \Leftarrow read\_comb\_flist()$
for $fault$ in $comb\_flist$ do
    $endp \Leftarrow prim\_list(fault.path())$
    $inj\_times \Leftarrow timing\_list(fault.patterns())$
    $seq\_flist \Leftarrow append\_transient\_fault(endp, inj\_times)$
end for
$sequential\_results \Leftarrow run\_simulation(seq\_flist)$
$comb\_flist \Leftarrow drop\_faults(sequential\_results.detected())$
```

4.6 Software tools integration

Since the flow expects to use such diverse tools and to perform many time-consuming operations, as already briefly said it was needed to automate the process through software integration.
This has been achieved by adopting scripting in Python language predominantly, mostly used when dealing with large files, complex data structures and process handling, but also in Bash language, used instead for text parsing and result analysis.

The advantage of the integration comes also in form of modularity and reconfigurability, allowing the user to skip some sections of the flow if needed, to change some process parameters very rapidly and to expand the flow fairly easy complying with different netlists and tools.
Reconfigurability has been achieved through the use of *YAML* files. The former is a human-readable data-serialization language which is widely used for writing configuration files and in other kind of applications. It is very well supported with dedicated libraries and packages by the most used programming languages [14].
Using Python, the module named *PyYAML* can be imported, which is a YAML parser and emitter.

Each phase of the flow can be managed as a separate thread, providing that it receives the right inputs from the other sections of the flow and on its turn gives the right outputs to the next ones. This is possible because a configuration file has been structured for each phase, in addition to a global one that serves the purpose of "master" configuration file.
To comply with the structure of the global configuration file it is needed to specify three fields for each phase:

1. **Enable**: by setting or clearing this flag it is possible to perform the job or skip it if not necessary.
2. **Run folder**: specifies which folder will be used as root folder for that specific tool, so as to have separate work-spaces for each of those.
3. **Scripts**: specify a list of commands or scripts to be launched for that specific phase of the flow. This allows not only to run tool scripts, but also user made scripts for post-processing or result analysis and eventually to expand the flow functionalities even further.

Algorithm 3 Root script

```
for $flow\_section$ in $glob\_set$ do
  if $flow\_section.isenabled()$ then
    $section\_var \Leftarrow import\_config(section\_file)$
    $curr\_dir \Leftarrow chdir(flow\_section.run\_dir())$
    $env \Leftarrow set\_environ(section\_var)$
    for $script$ in $flow\_section.scripts()$ do
      $run(script)$
    end for
    $env\_var \Leftarrow unset\_environ(section\_var)$
    $curr\_dir \Leftarrow chdir(root\_dir)$
  end if
end for
```

Then, for each phase, it is possible to specify in a separate .yaml file all the internal variables used by the scripts, either if they are meant for use in tool-specific scripts or other kind of scripts. These will be set as environment variables before the execution of the commands or scripts required.

A root script written in Python is in charge to parse the global configuration file and the tool-specific configuration file, according to algorithm 3. Then it instances a separate process for each of the enabled features of the flow, executing scripts one at a time.

It is important to highlight that tool specific scripts are written in *tcl* language, which is native (or in any case supported) for the EDA softwares used in the implementation. Each of them has its own set of commands that can be issued to perform tasks, produce report about the activities as well as output files written according to specific formats.

Chapter 5

Experimental results

In order to prove the efficiency and the effectiveness of the proposed flow, a small set of self-test programs has been selected to be used as a source of input functional patterns. These test programs were meant to be targeted at functional stuck-at fault testing and were able to achieve a high fault coverage in that regard. One of the test program used, instead, was obtained by random code generation targeting the alu functionalities. Their coverage with respect of transition delay fault has been evaluated as well, since this fault model is more similar to path delay, in order to make comparisons among the results. The structure of each code will be briefly described in the dedicated paragraphs, in order to highlight the main characteristics and features.
By performing these fault simulations, the following objectives were then pursued:

1. **Prove the functioning of the implemented flow**.

2. **Prove a correlation between stuck-at fault coverage and path delay fault coverage**.

3. **Statistically derive insights on functional testability** of path delay faults.

4. **Give indication in the matter of test generation**.

Before listing the results extracted by the test programs simulation, some general data and characteristics of the design are reported. In fact, in order to allow comparisons among them, the test programs were run on the same netlist and with the same fault list.

5.1 Netlist data

In order to give an overview of the size and complexity of the synthesised *RI5CY* core of the *PULPino*, in the table below it is presented a brief summary of the most relevant characteristic of its gate-level representation.

Characteristic	**Value**
Number of ports	9,076
Number of nets	34,258
Number of cells	25,144
Number of combinational cells	22,749
Number of sequential cells	2,325
Number of buffers/inverters	4,974
Number of gates	46,850
Combinational area	27,843.81 μm^2
Buffers/inverters area	3,213.01 μm^2
Noncombinational area	9,642.50 μm^2

Table 5.1: Top level core info

It is important to highlight that the synthesised netlist has been obtained by first flattening the design hierarchy, then regrouping the combinational cells avter the issue of the *compile* command. This was necessary to apply the devised flow, and has added the benefit of allowing the synthesis tool to have more space for optimizations.

5.2 Fault list data

The fault list was generated through static timing analysis, imposing a constraint on the clock period of 5.0 ns, and features the characteristics reported in table 5.2:

Characteristic	**Value**
Total number of paths	17,738
Clock period	5.00 ns
Minimum slack	0.37 ns
Maximum slack	4.95 ns

Table 5.2: Report timing information

Since the purpose was to evaluate the coverage achieved across the whole core, a large number of paths has been extracted, such that it could evenly represent the device.

In order to give an idea of how representative the fault list is and which are the main targeted design modules, an analysis on where the paths are located has been performed.
A total of 17,738 paths have been extracted, distributed as indicated in table 5.3 across the whole core. Note that each path could either traverse more than a single module, not be entirely contained inside the traversed modules and, especially in case of very short paths, it may not even go through a module but just through combinational cells.

Design module	N° of paths
ex_stage_i_alu_i_add_168	3,480
ex_stage_i_alu_i_add_182	3,480
load_store_unit_i_add_463_aco	997
load_store_unit_i_mult_add_463_aco	942
ex_stage_i_mult_i_add_109_2	790
ex_stage_i_mult_i_mult_109	790
r1589	434
ex_stage_i_alu_i_int_div_div_i_add_100	223
id_stage_i_add_531	80
cs_registers_i_add_775	66
ex_stage_i_alu_i_int_div_div_i_sub_100	6

Table 5.3: Number of paths per module

It is important to underline that the fault list used for the reported experiments has been accepted as valid without further investigation on the nature of the paths contained in it. Fault list minimization was not among the objectives of this thesis, so no pruning or analysis with respect to functionally untestable faults has been performed. Only the set of structurally untestable faults has been identified by the fault simulator tool automatically during fault simulations.

5.3 Test programs information

In this paragraph the most relevant data about the test programs used are reported, together with a brief overview of their structure. A total of five different test programs have been used: four of those have been previously created for functional stuck-at fault simulation specifically for the benchmark device (numbered from 1 to 4) while one has been produced by random generation specifically for the purpose of the presented work, from now on referred as *program random*.
The coverage of these programs will be evaluated with respect to path delay faults, so the effort here is to suggest a possible correlation between stuck-at and path delay faults. This could be helpful also to give insights in regards of test program

generation.

Program 1 is a medium-sized test program of the duration of 64502 clock cycles. It is composed by a series of macros, each one targeting a separate functionality of the core. From a brief look at the code, it can be seen that there are dedicated macros for testing the alu, multiplier, load and store unit, together with those meant for register file testing, branches and compressed instructions. Specific macros to test the call of invalid instructions and the relative interrupt handling are also included.
The macros were built using a similar structure, composed by multiple instructions in series followed by multiple store in series.

Program 2 instead is a quite short test program, composed by only 36,500 clock cycles.
In this case the code was organized for the most part as a series of couples of instructions where arithmetic operations were immediately followed by a store operation. The multiplier has been tested loading two registers with checkerboard patterns and other random values and performing the same loop of operations after each load. A similar approach has been used to test the alu, while the register file has been tested resorting to march algorithms. Other part of the design like hardware loops and csr registers have been tested with custom algorithms.

Program 3 is a very short test program, that lasts just for 17,269 clock cycles. Again, basically all the functionalities of the processor core are targeted, with a different strategy with respect to other programs. In this case also instructions for vector operation, hardware interrupts and for reading the performance counter have been used.
The code is structured for the most part as a series of groups of instructions, composed by two loads on different registers, one operation based on the targeted functionality, then one store instruction.

Program 4 has the characteristic to be the longest of those considered for this comparison. It lasts for 181,370 clock cycles, approximately 10 times more than program 3.
The approach applied in this program is similar to program 2 but with some differences: at the beginning of each procedure the register file is cleared with a series of xor operations; then, the program features couples of operations composed by an instruction followed by a store. In this case also instructions for vector operation, hardware interrupts and for reading the performance counter have been used.
A distinctive feature of this program is that in each of the procedures several nested loops are inserted, while in others loops were unrolled.

Program random, as the name suggest, was randomly generated targeting the alu only, using a very small subset of assembly instructions. Its structure is based on groups of instructions repeated several times. Each of this groups follows the same structure:

1. **All the registers in the register file are loaded** with random immediate numbers.
2. **A random number of arithmetic instructions are issued**, chosen among additions, multiplications and divisions. A subset of registers is dedicated to each type of instruction. Operand and destination registers are chosen randomly within the reserved subset.
3. **All the registers values in the register file are stored** in the main memory.

In addition to that, a small portion of the test program was built reusing some of the procedures featured in program 1, to test the hardware loops, exceptions and jumps.
The structure was pretty linear and simplistic, since the purpose of evaluating this program coverage was to demonstrate if a correlation among the path delay fault model and the other two models considered could be emphasised. It was in fact expected that this program would not perform well also for path delay faults.
It resulted in one of the shortest program, with only 32,455 clock cycles and the observed performances towards stuck-at and transition delay are the weakest in the set, since many functionalities have been deliberately neglected.

The functional fault coverage obtained by the test programs with respect to stuck-at and transition delay faults are listed in table 5.4. This table will be also reported at the end of the chapter to summarize what has been witnessed after the functional path delay fault simulation, so that is possible to easily compare the functional coverage on this fault models with the path delay one.

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

Table 5.4: Test programs functional fault coverages

As reported in table 5.4, all the programs show a similar fault coverage with respect to stuck-at faults, with results ranging from 81.37% to 86.77%. Program random is the only one out of this range, achieving a 59.44% across the whole design.
With regard to transition fault instead, it is noticeable how less consistent results

were obtained, with very low coverage in particular for the random test.

5.4 Fault simulation data

This section will be dedicated to the results obtained from the different fault simulations performed. Intermediate results that have lead to the final functional defect coverage will be also reported, explaining their importance and commenting on what can be extracted from them.

5.4.1 Combinational level fault simulation

The flow has been executed with every of these files as input. The first data to be reported are those relative to the initial fault simulation performed at combinational level. The coverage obtained here represents the ideal coverage that could be reached with functional techniques, if all faults could present the relative misbehaviours on the PO.
Again, the fault coverage reported here is not to be intended as the true achievable fault coverage, since that would require to remove untestable faults from the fault list.

Parameter	Program 1	Program 2	Program 3	Program 4	Pr. Random
Number of Patterns	64,502	36,394	17,269	118,098	32,416
Total faults	35,476	35,476	35,476	35,476	35,476
Undetectable	2,662	2,662	2,662	2,662	2,662
Not controlled	25,998	25,841	25,958	25,260	26,241
Total of detected	6,816	6,973	6,856	7,554	6,573
Detected by simulation	1,648	1,909	1,288	1,673	1,861
Robustly detected	5,348	5,064	5,568	5,881	4,712
Fault coverage	19.21%	19.66%	19.33%	21,29%	18,53%

Table 5.5: Combinational fault simulation results

As shown in table 5.5, program 1, 2 and 3 have performed almost identically in terms of achieved fault coverage, differing by just few decimal digits.
This is of particular importance when considering that the test program have been devised with different techniques and structures and are very heterogeneous in terms of duration.

The number of robustly detected faults may pose questions on what has been described in the previous chapter, appearing slightly contradictory.
It is important to highlight that faults indicated in table 5.5 as *Detected by simulation* and as *Robustly detected* are classified by Tetramax respectively as class

DS and *DR*.
A section of the tool manual [12] is quoted verbatim here in order to clarify the definition of faults assigned to these fault classes.

*" **DT (Detected) = DR + DS + DI + D2 + TP***
The "detected" fault class is comprised of faults which have been identified as "hard" detected. A hard detection guarantees a detectable difference between the expected value and the fault effect value. The detection identification can be performed by simulation or implication analysis."

*" **DR (Detected Robustly)***
DR faults are hard detected by the fault simulator using weak non-robust (WNR), robust (ROB), or hazard-free robust (HFR) testing criteria to mark path delay faults. During ATPG, at least one pattern that caused the fault to be placed in this class is retained. This classification applies only to Path Delay ATPG."

*" **DS (Detected by Simulation)***
DS faults are hard detected by explicit simulation of patterns. During ATPG, at least one pattern that caused the fault to be placed in this class is retained."

Therefore we can justify this result by considering that the definition of the criteria for which a fault is classified as *DR* is comprehensive of both the robust and non-robust criteria listed in section 2.2.

Comparing the results of program random with the other ones, an important consideration can be done: even if the test programs achieved very different coverages on stuck-at and transition delay faults, for what concerns path delay faults these differences are much more compressed and limited to more or less one percentage point in three out of four cases.
This hints that no correlation may exist between the fault models. In fact, if it existed, because of the large variation in coverage on stuck-at it was expected to see a proportional variation also on path delays. This conclusion, by the way, can only be confirmed after having evaluated the functional fault coverage.

This idea is also reinforced by the result achieved by program 4. In fact, it was revealed to be the most well performing on path delays, even if it is only the second best performing with respect to stuck-at.
These considerations, by the way, have to be observed also at sequential level in order to give a better understanding of the kind of relation that intervenes between the fault models.

5.4.2 Fault list conversion

In order to perform fault injection, as depicted in paragraph 4.5, an analysis on the endpoint affected by the detected fault has been performed, leading to the result listed in table 5.6.
This is necessary to distinguish which faults have to be injected and which instead can be considered already functionally detected. In fact, it can be assumed that if a fault is directly affecting a PO there is no need to include it in the sequential simulation, since PO are considered observation points and fault effects have not to be propagated further.
The same goes for faults affecting very sensitive signals as clock gating or enable signals: it can be assumed that in presence of a delay fault serious synchronization issues would be caused that would most likely cause the whole circuit to fail preventing the test itself to be executed, given its nature.

Primitive	**Program 1**	**Program 2**	**Program 3**	**Program 4**	**Random**
FF (D)	440	409	209	623	188
Latch (D)	6,044	6,363	6,339	6,444	6,145
Prim. Out.	298	167	193	453	206
FF (CK)	1	1	1	1	1
Latch (EN)	33	33	33	33	33

Table 5.6: Number of faults per primitive class

Looking at table 5.6 we can recognize that the vast majority of the detected faults are affecting latches: this is due to the fact that the register file in particular and also other registers have been mapped to this kind of logic element during the synthesis process.
In order to functionally detect these fault, after the injection their effect should traverse pipeline registers and reach the address or data port of the memory, requiring multiple clock cycles of execution.

It is possible to see how well program 4 has performed with respect to the other considered. A total of 453 faults detected have their endpoint represented by primary outputs, against the 298 listed in program 1 and the 206 in program random. This signifies that program 4 has targeted better than others the data and instruction interfaces.
This can be extended on a general level, since also the number of detected faults affecting flip-flops and latches input is higher.

5.4.3 Sequential level fault simulation

As explained in the previous chapter, sequential simulations were launched on a time-based schedule with the purpose of speeding up the process. For each time

interval, a new fault list had to be converted and simulated. Even if the simulated faults were the same from one sequential simulation to the next one, what changed was the number of injections performed, since combinational simulation may have produced new patterns for each of the fault in the assigned time window.
In table 5.7 it is reported a cumulative result of all the sequential fault simulation performed. These data represent the results that can be obtained with SBST techniques using these test programs.

Parameter	Program 1	Program 2	Program 3	Program 4	Random
Injected	6,484	6,772	6,629	7,067	6,333
Det. by Sim.	4,797	5,420	4,781	6,660	5,011
Det. by Impl.	332	201	227	487	206
Relative FC	75.25%	80.61%	73.04%	94.59%	76.23%
Func. FC	14.45%	15.84%	14.11%	20.14%	14.12%

Table 5.7: Functional fault simulation results

Fault injections have revealed that, out of the whole list of faults to inject, in case of program 1 a total of 4,797 faults have been successfully propagated to one or more POs. These should be then added to those faults that have been detected by implication, described in table 5.6, achieving a relative fault coverage of 75.25%.Then this percentage was related to the full set of faults analyzed in the first place, finding that it is equivalent to the 14.45% of the global fault list.
The same conclusions have been drawn also for the other test programs, as reported in the table.

Concerning program 2 instead, the relative fault coverage statistic indicates that the program was able propagate the misbehaviour to the primary outputs in more than the 80% of the cases. Also, the global fault coverage is near to 16%, which is a slightly better result with respect to other programs.
This result is good also because of the short duration of the test.

A total of 4,781 faults have been successfully observed on POs for program 3, achieving a relative fault coverage of 73.04%. This leads to set the global fault coverage at 14.11% of the global fault list. It can be noticed that this program is the one that has produced the least satisfying results, both considering the relative and the global fault coverage.
Even so, it is to be remembered that this is the shortest program among the considered ones: if the clock period imposed during synthesis is considered, the program would last only $86.3\mu s$, approximately half of the second shortest program.

Going on in analyzing what has been obtained with program 4, it must be emphasised that almost all of the path delay fault detected in the core have been marked as functionally detectable, with a relative fault coverage of 94.59%.

Considering that this program had already achieved the best overall coverage at combinational level, it can be said that this is the best performing in the set, achieving 20.14% of fault coverage, which is an increment of 4.3% with respect to program 2 that was the better so far. It has to be said, by the way, that the program duration is by far the longest in the set, approximately 10 times longer that program 3.

Running instead program random some interesting results have been obtained and some observations need to be done.
Remembering the considerations made before when discussing the combinational level results it is possible to see that, despite the large gap in terms of stuck-at and transition delay fault coverages, the results are comparable to what has been achieved by running other test programs.
Moreover, it can be seen that the relative fault coverage and the absolute number of detected faults are higher than program 1 and program 2, despite the lower number of injections performed.
These considerations pull again in the direction of concluding that no correlation among the considered fault models coverages can be exposed. In fact, it was expected to find out a much lower functional path delay fault coverage from program random, but it was not the case. This also exposes the possibility that the structure of the program has been more adapt to assist the propagation through pipeline stages of the faults. By the way, more in depth analysis are required to confirm this hypothesis, which are out of the scope of this work.

5.4.4 Test programs effectiveness

After that, an analysis on the effectiveness and general performance of the test programs has been performed. This is useful to understand how well critical faults in the design have been covered and if there is the possibility to test critical faults resorting to SBST techniques.

The detected faults were observed more in depth, evaluating their criticality and reporting some features of the path that had originated those. These information are reported in table 5.8.

For completeness, here are reported the modules traversed by the paths included in the previous table.
Path 4323 and 4321 are located in the module named *ex_stage_i_alu_i_int_div_div_i_add_100*, which is part of the divisor circuit contained inside the alu. Path 10873 goes through two different modules, named *load_store_unit_i_add_463_aco* and *load_store_unit_i_mult_add_463_aco*, which appear to be part of the addressing logic of the memory stage.
Path 7158 is again found in *ex_stage_i_alu_i_int_div_div_i_add_100*, while

Program 1				
Sim. type	**Path**	**Gates**	**Slack**	**Fault type**
Combinational	4323	95	1.776067	str
Sequential	4323	95	1.776067	str
Program 2				
Sim. type	**Path**	**Gates**	**Slack**	**Fault type**
Combinational	4321	95	1.770068	str
Sequential	4321	95	1.770068	str
Program 3				
Sim. type	**Path**	**Gates**	**Slack**	**Fault type**
Combinational	10873	23	4.199069	str
Sequential	10873	23	4.199069	str
Program 4				
Sim. type	**Path**	**Gates**	**Slack**	**Fault type**
Combinational	4321	95	1.770068	str
Sequential	4321	95	1.770068	str
Program random				
Sim. type	**Path**	**Gates**	**Slack**	**Fault type**
Combinational	7158	38	3.542068	str
Sequential	11330	20	4.305068	stf

Table 5.8: Most critical detected faults per program

path 11330 is found in the two same modules of path 10873 *load_store_unit_i_add_463_aco* and *load_store_unit_i_mult_add_463_aco.*

It can be seen that from the point of view of criticality of the detection the programs that have performed better are surely program 1, 2 and 4, since they were able to catch and propagate to the PO faults originated by two of the most critical paths of the design, composed by a total of 95 gates and with a relatively small slack. Program 3 instead did not perform well even on this aspect, since it was only able to provide detection only for very short paths.
An interesting example of fault masking can be observed by looking at what reported for program random. In fact, fault *str* on path 7158 was observed on the PPO, but the program failed to propagate that specific fault to the PO, as the most critical fault observed after the sequential simulation is different.
More considerations about critical paths and program effectiveness can be found in the next paragraph.

5.5 Result analysis and considerations

The pie chart in figure 5.1 shows the cumulative fault coverage obtained at combinational level by the full set of test programs used. In green it is reported the set of faults excluded from the fault simulation due to untestability, while in blue it is shown the ideal coverage achieved. In orange instead is shown the amount of not detected faults.

What can be evinced from this chart is that most probably it is not possible to adapt stuck-at oriented test programs to achieve path delay fault detection, both for what has been shown previously about the correlation between the two fault models and more in general for the very low coverage achieved even when combining multiple programs.
More about the reasons that has led almost 70% of the faults to result undetected can be said only with a deeper analysis of the test programs and the fault list, which is out of the scope of this work. What can be said is that the faults have been classified as undetectable because they were not controlled, according to the fault simulator analysis. Therefore, it is ideally possible to catch those faults if more reliable and tailored techniques for test generation are adopted.

However, it can be assumed that compacting and properly arranging the instructions currently listed in all the test programs, the fault coverage achieved by applying functional tests can be increased already at combinational level. Ideally, a total of 8082 faults could be observed at this stage, which approximately corresponds to an increment of 7% with respect to the maximum achieved coverage by a single test program.
It can be said that it is possible, in that case, to achieve a total fault coverage of 22.78% at combinational level.

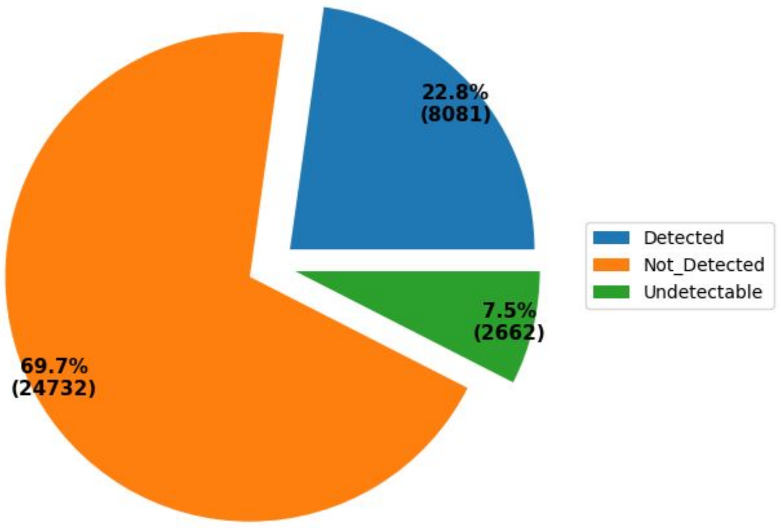

Figure 5.1: Cumulative combinational fault coverage

In figure 5.2 and figure 5.3 are displayed important information about the performance that the considered test programs achieved, from which some ideas

about testability and test program generation can be evinced.
In particular, figure 5.2 shows the number of faults that can be detected at combinational level by one or more test programs. As it can be seen, the vast majority of the detected faults (72.4%) can be detected by all the test programs. The 7.0% of faults have been detected by 4 programs out of 5 and similar percentages are evaluated if only 3 or 2 programs are considered. Only a very small percentage, exactly the 7.5% of the total number of faults, are detected by just one out of the set of test programs used.

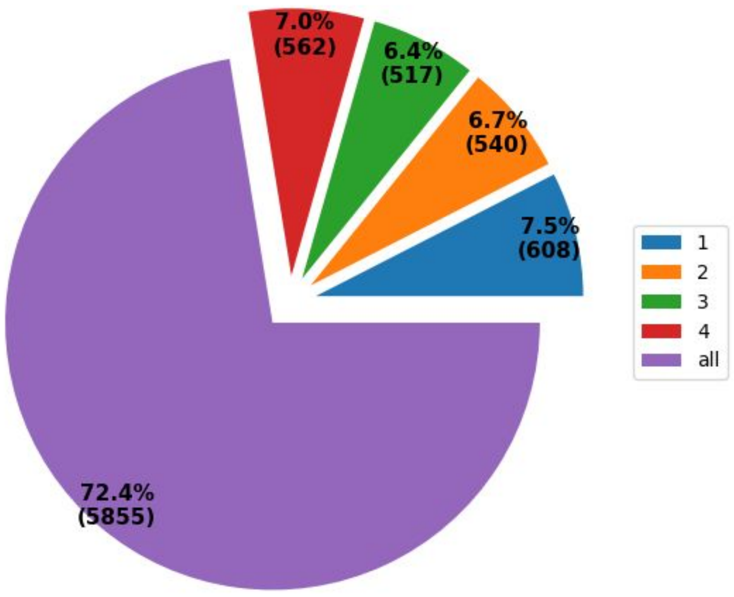

Figure 5.2: Number of faults detected at combinational level shared by programs

From these information it can be evinced that many faults can be intended as *"easy-to-test"*. In fact, since the programs are very different from each other, it is possible to assume that there exist multiple couples of instructions that are able to catch those faults.
As expected, among those faults that have been caught by only one of the test programs, it is possible to find faults originated by some of the slowest paths in the circuit.
This is most probably due to the higher number of gates featured in those paths, which makes harder to produce the appropriate values and conditions for the transition to be propagated. Hence, it is possible to assume that those faults should be targeted specifically when devising a test program to ensure coverage.

A similar analysis has been conducted also for the functionally detected faults. In this case the amount of detections shared by multiple programs is distributed in a more heterogeneous fashion among the test programs. We can see that the 42.3% of the faults detected at combinational level has been successfully propagated and observed on the primary outputs by all test programs; another 12.7% have been detected by 4 test programs, while 10.4% and 19.4% have been detected respectively only by 3 and 2 programs. Lastly, it is possible to see that 920 out of 8,082 faults have been functionally detected by only one test program and a very small percentage, namely the 3.8% has not been functionally detected.

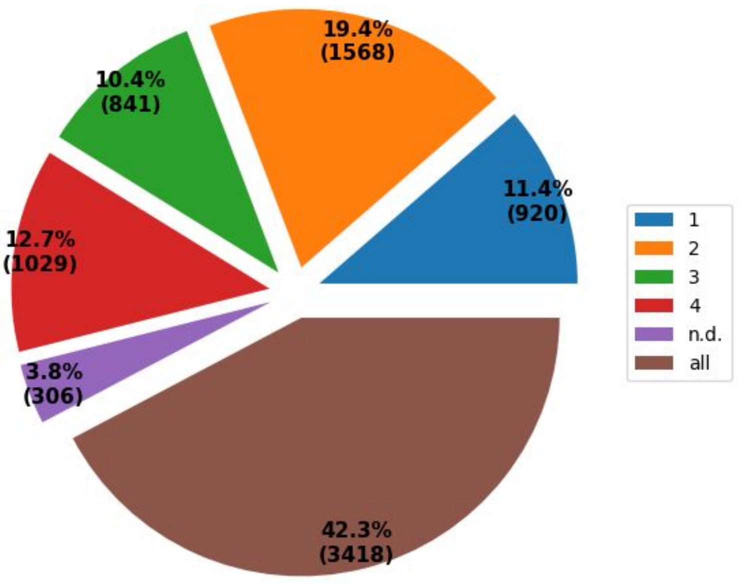

Figure 5.3: Number of faults detected at sequential level shared by programs

Again, it is possible to evince that the functional detection of most of the faults can be achieved fairly easily, but still, ensuring the propagation for a relevant slice of faults is an hard task.
Among those faults that were functionally detected by a single test program and those functionally undetected, there are several faults originated from slow paths. This problem can be caused by the fact that the test programs were not written with the goal of detecting path delay faults, hence some necessaries code features for propagation may lack (e.g. store instructions), or may be not sufficient. In fact, among the undetected faults also paths that have an average slack are listed, so the cause of not detection may not be just related to structural or functional properties of the device but also to the code structure.

In order to give a statistical overview on faults testability, in figure 5.4 is shown the cumulative number of faults detected, focusing on program 4.
More specifically, values on the x axis represent the number of injections performed,

while on the y axis it is reported the cumulative number of faults detected with less than "x" injections. This specific graph has been obtained without considering equivalent faults for simplicity. Data reported represent a worst-case scenario, since it is not said that all injection are strictly required for the detection.
The number of required injections has been evaluated by comparing the detection time to the scheduled injection times and finding how many of those were scheduled before the detection time.

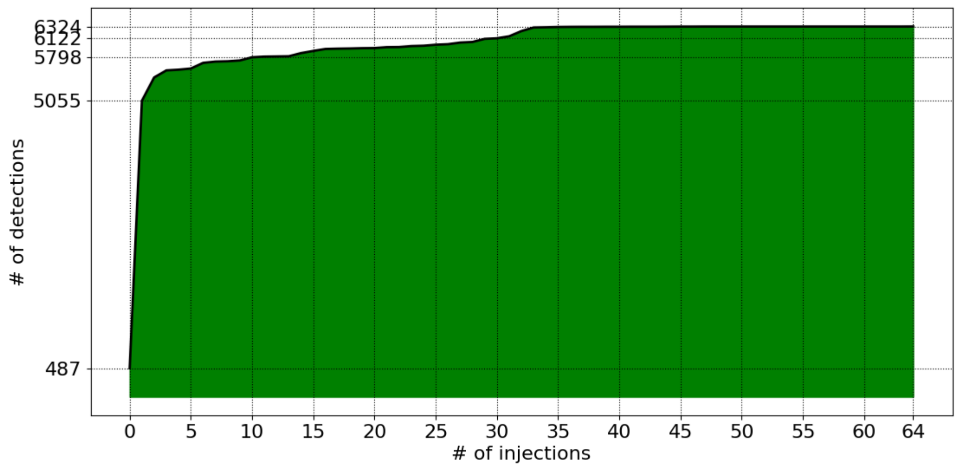

Figure 5.4: Number of detected vs. number of injections

It is again clear that the most part of the faults is very easy to detect: a total of 5,055 faults require 1 or less injections to be detected. The number of detection keeps increasing with the number of injection, but with a much slower rate: 10 or less injection lead to the detection of 5,798 faults, while performing 30 or less injections has led to achieve 6,122 functional detections.
Going on, very few faults require a number of injections higher than 40, in fact this region of the curve is almost flat. In the end, the fault that has revealed harder to detect has required a total of 64 injections before being observable at one of the PO.

Surely, further investigation should be done to identify proper methods for test programs generation, but these data may be of interest in narrowing the variables to explore in that regard.
In table 5.9 are reported some characteristics of the most critical faults that have been functionally detected. For each of those, from left to right are reported : a number in the first column that serves as identifier; the number of gates included in that path; the relative slack with respect to the imposed clock period (5.0 ns); the

program number that has produced the detection at combinational level; the type of the fault detected at combinational level; the program number that has produced the detection at sequential level; the type of the fault detected at sequential level. For simplicity, only a subset of faults is reported here.

Path N°	**Gates**	**Slack (ns)**	**Pr. (Comb)**	**Type (Comb.)**	**Pr. (Func.)**	**Type (Func.)**
4321	95	1.770068	2, 4	str	2, 4	str
4323	95	1.776067	1, 2, 4	str	1, 2, 4	str
4325	95	1.785068	1, 2, 4	str	1, 2, 4	str
4326	95	1.791067	1, 2, 4	str	1, 2, 4	str
4328	94	1.797068	2, 4	str	2, 4	str
4331	94	1.797068	2, 4	str	2, 4	str
4342	92	1.840068	2, 4	str	2, 4	str
4343	92	1.846068	1, 2, 4	str	1, 2, 4	str
4348	92	1.855068	1, 2, 4	str	1, 2, 4	str
4356	92	1.861068	1, 2, 4	str	1, 2, 4	str
4358	91	1.867068	2, 4	str	2, 4	str
4360	91	1.867068	2, 4	str	2, 4	str
4372	89	1.937068	2, 4	str	2, 4	str
4373	89	1.943068	1, 2, 4	str	1, 2, 4	str
4375	89	1.952068	1, 2, 4	str	1, 2, 4	str
4385	89	1.958068	1, 2, 4	str	1, 2, 4	str
4387	88	1.964068	2, 4	str	2, 4	str
4390	88	1.964068	2, 4	str	4	str
4402	86	2.025069	2, 4	str	2	str
4403	86	2.031068	1, 2, 4	str	1, 2	str

Table 5.9: Most critical functionally detected faults information

From table 5.9 it can be evinced that the most suitable test programs for path delay functional fault testing are program 4, program 2 and program 1, since those were able to detect the most critical paths. In particular, program 4 manages to catch almost all of those even from a functional perspective.
For this reason, it is proposed to analyze it more in depth to understand what characteristics of that program has allowed to achieve such results.

Instead, in table 5.10 are reported some characteristics of those paths that have been detected at combinational level but have not been observed on the primary outputs.

Again, it is possible to notice that program 4 has performed better than the others even in terms of criticality of detected faults, covering those faults at least at combinational level.

Still, with the current information available it is not possible to say if these faults could have been detected functionally if the program was composed differently, but it is relevant that path lengths and slacks are comparable to the ones in table

Path N°	Gates	Slack (ns)	Pr. (Comb)	Type (Comb.)
4341	92	1.840068	4	str
4349	92	1.855068	4	str
4371	89	1.937068	4	str
4376	89	1.952068	4	str
4388	88	1.964068	4	str
4411	85	2.052068	2, 4	str
4471	80	2.210069	4	str
4474	80	2.225069	4	str
4481	79	2.237069	4	str
4512	77	2.307069	4	str
4515	77	2.322069	4	str
4520	76	2.334069	4	str
4621	68	2.580069	4	str
4625	68	2.595069	4	str
4638	67	2.607069	4	str
4661	65	2.677069	4	str
4666	65	2.692069	4	str
4679	64	2.704069	4	str
4772	56	2.950069	4	str
4785	56	2.965069	4	str

Table 5.10: Most critical functionally undetected faults information

5.9, element that suggest that detection on PO may be possible. Of course path topology may be extremely different, so further investigation are required to have a certain answer to the question.

The number indicated in the column *Path N°* in tables 5.9 and 5.10 refers to the classification performed during the static timing analysis. Hence, if a path is marked with the number *x*, it means that *x-1* slowest (more critical) paths exist, according to the STA. To properly evaluate the functional fault coverage of the programs, it should be fundamental to find a method to evaluate if paths are functionally testable or not, but this is a very complex task not considered in this thesis, being its ultimate objective to provide the means of evaluating the functional coverage of a test program.
For sure, what can be said is that a total of 1,331 undetectable redundant paths have been found within the fault list during the fault simulation using Tetramax, and 790 of those paths were slower than the most critical functionally detected path. For these paths is not possible to produce tests because of structural properties that block the propagation of a transition along them.
From this point of view then, the effectiveness of the programs can be considered

higher than what may seem, since the undetectable redundant paths correspond to false paths and thus they do not affect the timing of the circuit and should not be tested.

Nevertheless, the tests used fail to catch several of the most critical paths not only from a functional perspective, but already at combinational level, so effort should be primarily put in producing new tests targeting those faults that have not been observable even at that stage, provided that those faults can be identified as functionally detectable.

To sum up what has been discussed in this chapter, table 5.11 collects the fault coverage percentages for each program on the three fault models mentioned in this thesis.

Parameter	**Program 1**	**Program 2**	**Program 3**	**Program 4**	**Random**
Clock cycles	64,527	36,500	17,308	181,370	32,455
SAF FC%	86.77	81.79	81.37	82.97	59.44
TDF FC%	41.90	44.21	63.16	61.90	24.41
PDF FC%	14.45	15.84	14.11	20.14	14.12

Table 5.11: Fault coverage comparison

An information that can be extracted from this table is that it is hard to notice a correlation among the three fault models, since the differences of what has been achieved for stuck-at and transition fault models does not reflect a significant variation with respect to the path delay fault model.
As already highlighted at page 46, program 3 and program random show a very large gap in stuck-at fault coverage, but yet the path delay functional fault coverage is almost the same. This happens again for the same two test programs but this time for the transition delay, where the coverage gap reaches almost 40% but leads to the same result relatively to path delay.
At the same time, program 4 achieves a lower coverage on transition delay with respect to program 3 but manages to achieve the best path delay coverage. This result may be dictated by the fact that program 4 is ten times longer than program 3 in terms of clock cycles, hence more couples of instructions can be extracted from it, leading to a higher chance of detection.

Chapter 6

Conclusion and future improvements

The work carried out in this thesis had the primary goal of devising a flow aimed at path delay functional fault test simulation and provide an example of its implementation, based on the integration of several commercial software tools, that could automate the process of functional fault coverage evaluation on a target device.

In conjunction to this, two secondary objectives were pursued: expose a possible correlation between stuck-at fault coverage and path delay fault coverage achieved by the same test programs; provide some insights about the functional testability of path delay faults.

6.1 Functional test flow

Concerning the main objective, it can be said that it was proposed a very general and reproducible approach that can be tailored and reconfigured basing on the availability of the tools and the requirements of the user.
Generality is ensured by the possibility of using different software produced by different vendors, providing that the correct implementation adjustments are taken in consideration.

The proposed implementation adds some interesting and useful features to the basic flow in order to optimize it and produce the fault coverage evaluation with the minimum cost in terms of simulation time.
Since the no fault-drop combinational fault simulation is the most time-consuming phase of the proposed implementation, efforts have been put in optimizing it from different perspectives.

The total duration depends on the test program used, since the more detection are performed the more simulations will be scheduled. In case of the experimental tests performed, the elapsed simulation time was in any case less than 48 hours, during which thousands of fault simulations had to be run using five concurrent processes. This by the way can be improved by simply increasing the number of employed resources. Also, being the fault detection record and other important variables (paths list, active fault list, simulation schedule etc.) managed with a shared memory section, the necessary means of synchronization among the processes could add overhead to the simulation time by keeping processes in a wait state, especially when the number of child instances is large.
There is room for improvement in this, for example splitting simulations also by faults and not only by patterns, assigning a portion of faults to each child process ensuring independence among those and letting each one manage its own variables, providing that results are merged at the end of computation.
Another option could be to manage simulations with pipes, avoiding to open/close the fault simulator and repeat section of the simulation scripts like initialization or paths addition, allowing to save time.
Moreover, an optimal time interval with which to call functional fault simulations should be found, in order to balance the time spent in both of the phases: if this interval is too narrow, the combinational fault simulation will have a smaller time window to find new patterns and the probability to have fault injection with no fault detection increases; instead, if the interval is too wide the number of patterns found during combinational fault simulations may be higher than what is necessary to achieve functional detection, increasing the time of both combinational (the faults will be dropped later, leading to overhead and unnecessary complexity) and sequential (more than the optimal number of injections will be performed, increasing simulation time) fault simulations.

In order to ease future development, the proposed implementation of the flow offers reconfigurability thanks to the possibility of including new scripts and new variables when necessary by just modifying the provided configuration files according to the user needs.
Repeatability of the experiments is ensured by the choice of using widespread and proven software tools.

For what concerns the obtained results, it was shown that approximately from 73% to 94.6% of faults detected at combinational level, hence detectable with scan techniques, are propagated to the primary outputs of the processor core proving the effectiveness of SBST techniques.
However, the test programs used for the flow validation have proven to be not optimal to cover all the faults included in the fault list, and also the fault list itself has been taken raw, without performing any kind of analysis on possible untestable

faults. So, in order to have a more representative result it is proposed to improve both aspects in future.

6.2 Side goals

Analyzing instead what has been observed in direction of the side objectives, no evidence of a correlation between stuck-at and path delay fault coverage has been found. Highlighting a correlation between the two could have helped in terms of test program generation and minimization, allowing to borrow techniques and even code snippets used for that fault model.
However, test programs that have achieved very different fault coverages on the stuck-at fault model have revealed a similar fault coverage on path delays. For the same reasons, no relation can be evinced also with respect to the transition fault coverage.

About the functional testability, by running fault simulations at combinational level it has been shown that a large set of the detected faults is shared among all the test programs.
Given that the programs have been composed using different techniques and feature different sets of instructions, it appears that this set of faults is "easy-to-test", since it is possible to find multiple patterns in multiple programs that allow their detection. Unfortunately, faults contained in this set are usually originated by non-critical paths, composed by few tens of gates and characterized by a large slack.
A small but significative percentage (7.5% in the presented case study) of faults is on the contrary detected by only one test program, meaning that their detection is harder to achieve. As it might have been reasonable to expect, faults in this set are usually (but not only) originated from longer paths, characterized by a number of gates ranging from 80 to 100 resulting in a shorter slack.
Even if producing tests for these faults can be tricky, results have shown that it is possible to detect them even with a functional approach. It is then suggested to specifically target only this set of faults during test generation, in order to reduce the complexity of code generation and to produce more effective tests (since these are the paths that may fail earlier than others).
By extension, it is expected that a good portion of less critical faults will be also covered because of their high testability, and if not, there is a chance to detect them by random code generation, as shown in the results.

The downside of functional test is that, as it has been shown, some of the faults have to cause misbehaviour several times at combinational level before being functionally observable.
This may add complexity in test program generation since multiple test patterns

for each fault should be included. It is proposed to analyze more in depth this aspect since the estimation performed within this thesis was a worst-case scenario, hence it is possible that the same faults could be functionally detected also when performing less injections.

Bibliography

[1] 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*. Accessed 03/03/2020. URL: `https://riscv.org/wp-content/uploads/2016/01/Wed1315-PULP-riscv3_noanim.pdf`.

[2] Wikipedia contributors. *Bathtub curve*. Accessed 03/03/2020. URL: `https://en.wikipedia.org/wiki/Bathtub_curve`.

[3] Virendra Singh ; Michiko Inoue ; Kewal K. Saluja ; Hideo Fujiwara. «Instruction-Based Self-Testing of Delay Faults in Pipelined Processors». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems (Volume: 14 , Issue: 11)*. https://ieeexplore.ieee.org/document/4019469. IEEE, 2006. DOI: `10.1109/TVLSI.2006.886412`.

[4] *GitHub - pulp-platform/pulpino : An open-source microcontroller system based on RISC-V*. Accessed 03/03/2020. URL: `https://github.com/pulp-platform/pulpino`.

[5] Shi-Yu Huang. *Slides from the course 'EE-6250 VLSI Testing'*. Accessed 03/03/2020. URL: `https://www.ee.nthu.edu.tw/~syhuang/testing/ch6.delay_test.pdf`.

[6] V. D. Agrawall M. L. Bushnell. *Essential of electronic testing for digital memory and mixed signal VLSI circuits*. Kluwer Academic Publishers, 2000.

[7] M. Sharma ; J.H. Patel. «Enhanced delay defect coverage with path-segments». In: *Proceedings International Test Conference 2000 (IEEE Cat. No.00CH37159)*. https://ieeexplore.ieee.org/document/894228. IEEE, 2000. DOI: `10.1109/TEST.2000.894228`.

[8] Jim Plusquellic. *Slides from the course 'CMPE 646: VLSI Design Verification and Test'*. Accessed 03/03/2020. URL: `http://ece-research.unm.edu/jimp/vlsi%5C_test/slides/html/delay%5C_faults1.html`.

[9] K. Christou ; M.K. Michael ; P. Bernardi ; M. Grosso ; E. Sanchez ; M. Sonza Reorda. «A Novel SBST Generation Technique for Path-Delay Faults in Microprocessors Exploiting Gate- and RT-Level Descriptions». In: *26th IEEE VLSI Test Symposium (vts 2008)*. https://ieeexplore.ieee.org/document/4511756. IEEE, 2008. DOI: `10.1109/VTS.2008.37`.

[10] P. Bernardi ; M. Grosso ; E. Sanchez ; M. Sonza Reorda. «A Deterministic Methodology for Identifying Functionally Untestable Path-Delay Faults in Microprocessor Cores». In: *2008 Ninth International Workshop on Microprocessor Test and Verification*. https://ieeexplore.ieee.org/document/5070942. IEEE, 2008. DOI: `10.1109/MTV.2008.9`.

[11] Adit D. Singh. «Scan based two-pattern tests: should they target opens instead of TDFs?» In: *2015 16th Latin-American Test Symposium (LATS)*. https://ieeexplore.ieee.org/document/7102526. IEEE, 2015. DOI: `10.1109/LATW.2015.7102526`.

[12] Synopsys. *Tetramax ATPG and Tetramax II ADV ATPG User Guide, Version O-2018.06-SP4*. 2018, pp. 411, 412.

[13] Synopsys. *Z01X Functional Safety Assurance*. Accessed 03/03/2020. URL: `https://www.synopsys.com/verification/simulation/z01x-functional-safety.html`.

[14] *The Official YAML Web Site*. Accessed 03/03/2020. URL: `https://yaml.org/`.

[15] W. Qiu ; D.M.H. Walker. «Testing the path delay faults of ISCAS85 circuit c6288». In: *Proceedings. 4th International Workshop on Microprocessor Test and Verification - Common Challenges and Solutions*. https://ieeexplore.ieee.org/document/1250258. IEEE, 2003. DOI: `10.1109/MTV.2003.1250258`.