POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Generation and evaluation of software programs for delay testing of microprocessors

Supervisors

Dr. Riccardo CANTORO

Prof. Matteo SONZA REORDA

Candidate

Riccardo MASANTE

October 2020

Abstract

Nowadays, the electronic devices have invaded many different fields becoming very important for most of the objects that are used every days. The quality of these electronic devices is directly connected to their dependability. Moreover, the semiconductor industries are constantly looking for new technologies that allow big improvements in terms of performance. In the last few years, the channel length of a transistor is set to few tens of nanometers; this is the principal cause of the increasing complexity in manufacturing processes, but it allows very high working frequency and dense designs, leading also more frequent physical defects and devices die in less time. For these reasons electronic devices have to be tested deeply and researchers have studied several faults models to represent the behaviour of faulty circuit, such as the path delay fault model. Moreover, different strategies for the test exist, each one with advantage and disadvantage. The Software Based Self Test (SBST) is the methodology adopted in this thesis, it is based on forcing the processor to execute parts of code saved in memory checking the produced result to prove the correct behaviour of the device. This testing method is reliable, can be applied in circuits that are not physically accessible, and can detect faults that might show up for a short period of time respect to the device life. It can be applied without external tester, and can be performed keeping the device under test in its operational environment. This is the main strength of this technique, as tests can be applied periodically and during the whole life span. Furthermore, SBST does not cause overtesting phenomenon and can be performed at-speed, that are two concepts that are suitable for path delay model.

In this context the most challenging problem is to find automatic or systematic methods to generate software programs able to detect path delay faults for sequential circuit, because random programs result inefficient. Commercial tools can be used for this purpose, such as the Automatic Test Patterns Generation (ATPG) that, having available the hardware description of the device and its fault list, is able to classify faults and to compute test patterns that detect them. Commercial ATPG tool are too slow for sequential circuits, but can be used to generate patterns for the combinational logic, this implies that these patterns have to be translated in software instructions, which is the most challenging part of the work.

The method described above is obtained by trial and error process built on the open-source SoC Pulpino, developed by ETH Zurich and Università di Bologna and configured to use the *RI5CY* core. The method efficiency and the fault coverage achieved by this system are compared with programs written using other strategies. The fault coverage is computed using an innovative sequential fault simulator written for academic research.

Acknowledgements

Voglio dedicare questo spazio dell'elaborato per ricordare in italiano tutte le persone che mi hanno aiutato e mi son state vicine durante la scrittura di questa tesi.

Ringrazio Riccardo Cantoro e Matteo Sonza Reorda, i miei relatori, i quali mi hanno dato grandi stimoli per studiare e affrontare problemi attuali. Inoltre, Riccardo mi ha accompagnato da vicino in questo lavoro dandomi diversi spunti per migliorare strategie e risolvere problemi incontrati durante il percorso.

Ringrazio Sandro e Dario con cui ho collaborato per questo lavoro, essi mi hanno dato una grande mano, passandomi le loro grandi conoscenze.

Ringrazio Matteo, mio compagno della magistrale. Questi ultimi due anni sembrano passati in un attimo insieme, mi mancheranno i momenti meno seri, come le nostre battaglie lanciandoci le bacche in cortile del Politecnico. Mi mancheranno anche i momenti più seri in cui si arrivava a litigare per decidere come organizzare i nostri progetti.

Ringrazio la mia famiglia che è stata messa a dura prova con le mie stressanti lamentele, dovendomi sopportare tanti giorni in casa a causa delle restrizioni per sconfiggere la pandemia.

Ringrazio la mia ragazza Barbara, la quale mi ha accompagnato nella scrittura di questo elaborato incoraggiandomi e spingendomi sempre a fare qualcosa in più di quello che volevo.

Ringrazio i miei amici del gruppo "Delusione" e "Polli Marini" che mi hanno regalato tanti momenti divertenti, i quali sono serviti per distrarmi e per mantenere il morale sempre alto.

Grazie a questa sfida ho potuto imparare tantissime cose, che spero di poter spendere in ambito lavorativo in modo da poter dare un seguito a questo elaborato.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The importance to test electronic devices

Over the last few years the request of electronics devices is growing constantly. For example, in the automotive industry, nowadays the 40% of the cost of a car is due to its electronic components and will continue to grow in the upcoming years, as it is shown in Figure 1.1.

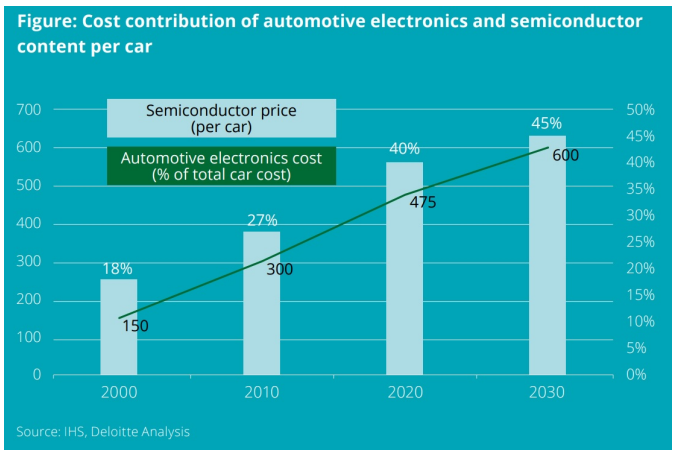

Figure 1.1: Cost of Electronics component in a car

As of 2017, the average vehicle has over 50 actuators, like transmission control unit (TCU), adaptive cruise control (ACC) and many others, typically controlled by power MOSFETs, micro controllers or other power semiconductor devices. It is important to test these electronic systems periodically to avoid malfunctions and to guarantee their dependability.

Furthermore, the semiconductor industries are constantly looking for new technologies that allow big improvements in terms of performance carrying a very fast evolution with respect to the past. One important progress concerns the manufacturing phase of the silicon dies, that are produced towards more complex and sophisticated processes. Nowadays, the channel length of a transistor is set to few nanometers. This type of the shrinkage is the principal cause of the increasing complexity in manufacturing processes, but it allows very high working frequency and dense designs, leading also more frequent physical defects and devices die in less time. Indeed, the new generation ICs are more prone to process variations, manufacturing defects, ageing effects, parasitic effects, electromagnetic interference and overheating-related issues. The older generation devices were less sensitive to degrading. All this reasons are modifying the shape of a typical failure rate curve that is represented in figure 1.2, in recent years many electronic product life cycle exhibit that curve. Moreover with new technology this become more narrow and deep, becoming more and more similar to a "U" shaped.

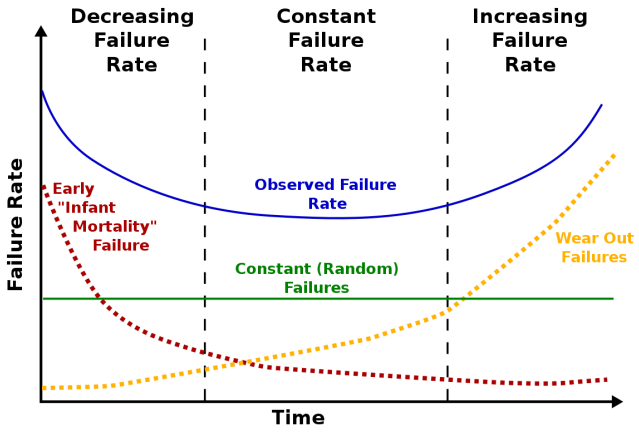

Figure 1.2: Typical failure rate curve, extracted from [4]

Those motivations highlight that testing technique must evolve constantly following the semiconductor industry improvements. The work of this thesis is going to be justified by means of some little hints of theory.

1.2 Theory in a nutshell

1.2.1 Fault models

In literature many models of faults exist, one of the most cited is the stuck-at-0/1 model. This model was created to represent the short and open circuit effect in

a devices. Nowadays this model is outdated because the increases in density of VLSI devices are shifting the focus on other problems: power dissipation, line resistivity and cross talk. They increase the premature failure or cause temporary malfunction of the IC. The stuck-at model does not represent correctly this type of problems, that are depicted better using delay faults model. In literature there are many types of fault belonging to delay fault model. In this thesis the focus is on transition delay faults and mainly on path delay faults. Just to introduce them:

1. **Transition delay fault**: it models a delay defect affecting a single gate, that results slower or faster than expected. The number of faults grows up linearly with respect to number of gates, so the fault list can be filled by all the faults extractable from a whole CPU. To produce efficient test is relatively easy, for this purpose there exists many adapted strategies from the stuck-at model (that was studied deeply in the past). There are some studies in literature that show that there are many cases in which these faults do not individually cause failure in the device.

2. **Path delay fault**: it is the evolution of the transition delay fault. It models a transition that crosses an entire paths, arriving to the end point early or late respect to what is expected. In this model the number of faults grows up exponentially with respect to number of gates, so it is impossible to test all the faults extractable from a device. Moreover, to test these faults is very difficult because there not exists efficient strategies adaptable from other models, but it is important because the path delay model represents in the best way what really happens in a electronic circuit since it is able to catch both lumped and distributed delay defects [8].

1.2.2 Test type

At-speed tests are needed in order to detect faults belonging to delay model. The *Software Based Self Test*, which is the methodology adopted in this thesis is based on forcing the processor to execute parts of code saved in memory checking the produced result to prove the correct behaviour of the device. This testing method is reliable, can be applied in circuits that are not physically accessible, and can detect faults that might show up for a short period of time respect to the device life. It can be applied without external tester, and can be performed keeping the device under test in its operational environment. This is the main strength of this technique, as tests can be applied periodically and during the whole life span. Furthermore, SBST does not cause overtesting phenomenon.

1.3 Thesis goal and environment used

State-of-the-art works highlighted that random program are completely inefficient to detect long path delay faults, so we worked on a systematic method to create assembly programs able to test long path delay faults in a sequential device using a SBST technique. The open-source SoC *PULPino* was used to evaluate the method. *PULPino* was developed by ETH Zurich and Università di Bologna and was configured to use the *RI5CY* core [2]. It is a 4-stage in-order 32b RISC-V processor core. The ISA of RI5CY is extended to support multiple additional instructions including hardware loops, post-increment load and store instructions and additional ALU instructions that are not part of the standard RISC-V ISA [6]. It can support different instruction set, for this study only RV32I, RV32C, RV32M are activated that allow to use integer,compressed and multiplication instructions. The core is presented in figure 1.3. This device was chosen because is

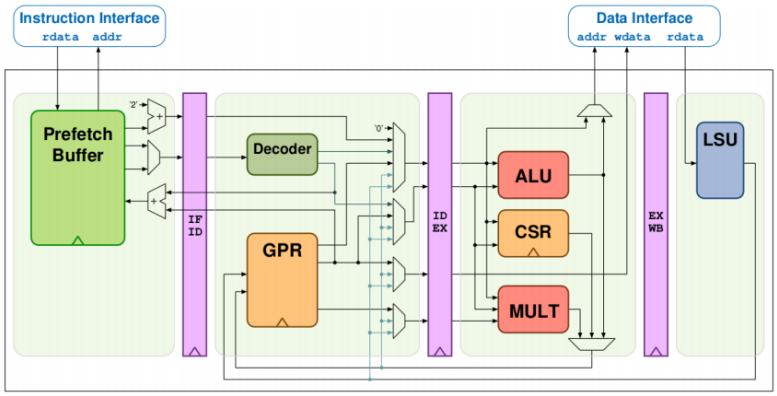

Figure 1.3: PULPino RI5CY core diagram, extracted from [2]

a real commercial microcontroller device, that is used in other academic research centers worldwide and its architecture is used by several companies. The proposed method is evaluated only on the synthesized core without considering all the parts concerning the peripherals.
The efficiency of the method is also evaluated using a sequential simulator, and the fault coverage obtained is compared with other results that are obtained in past.
In this thesis we want to explain some theoretical concept to prepare the reader, at that point the simulator to evaluate the method is showed in deep, and then the method is presented followed by the results obtained.

Chapter 2

Theory of Delay test

The concept of Delay test was studied following the birth of sequential circuit. These types of devices use a clock to synchronize all other signals internal to the circuit. To make sure that the circuit works correctly, the transitions applied to the input of the circuit must arrive to the circuit's output within a specific interval of time defined by the flip flop. This interval is computed subtracting the clock period to the *hold time* and *setup time*. The hold time is the minimum amount of time in which the output of a flip flop must be stable after the clock edge. The *setup time* is the maximum amount of time within the input data must be stable such that the data is stored correctly in the flip flop.

Sequential circuits are designed to respect these constraints, but in special cases it is possible that the delay of a single gate is different from the delay that would be expected. This problem can cause that the total delay of a single path violates the *setup* or the *hold time*, resulting that the device doesn't work correctly for a short period of time or for its entire life. When this happens the circuit is affected by a *delay fault*.

Some models have been developed to try and represent what happens in a real faulty circuit with the purpose of testing and detecting this type of faults. The objective of this chapter is to provide some theoretical concepts useful to appreciate the problems of the faulty models analysed and how faults can be tested.

2.1 Delay faults model

To clarify some concepts, it is assumed that a delay can start from any input to the output for each gate in a sequential circuit. In addition - as it is defined in the majority of libraries - the delay can be of two types: a *fall* or a *rise* delay. There are several models to represent delay faults in literature as transition fault model, gate delay fault model, path delay fault model, segment delay fault model and line

delay fault model [1]. The focus of this thesis will be on *transition delay* faults and *path delay* faults.

2.1.1 Transition Delay Faults

In this model each fault influences a single gate, testing these faults means watching if a transition is delayed or anticipated in a wire respect to what is expected. For each input of a gate there can be two type of faults:

- **Slow-to-rise (str)**: this fault happens when the transition applied is due to a rising edge.
- **Slow-to-fall (stf)**: on the other hand, this fault happens when the triggering transition is a falling edge.

This is a point of strengths for this model as it means that the number of faults will grow up linearly respect to the number of gates. Three main assumption can be established thanks to this model:

- A fault-free system is composed only by a gate with nominal delay.
- A different delay from the nominal one, whether it is positive or negative, is a fault.
- A delay fault that spreads through a path and reaches an output will surely provoke a failure, whether it is crossing a short or a long path.

This implies that to detect a transition delay fault it is necessary to apply a pattern to the input. A **pattern** is a set of values given to the inputs in a sequence. In this model a pattern is composed by two **vectors**, each vector is a set of values given to the inputs at the same time. The first vector is used to initialise all the gates under test, the second vector activates and propagates the transition to one of the outputs. The second one is the same used to test *stuck-at fault*. For example, when a *stf* transition is created, the first vector may set the gate under test to '1', the second vector must be able to detect a stuck-at-1 on that gate and set it to '0'. If the transition is able to reach an output than the fault is considered to be detectable by that pattern.
As explained, the stuck-at fault method is easily adaptable to this model to test this type of fault. This is a big advantage, because testing these faults becomes relatively easy.

The main problem is that this model does not uncover well defects related to real timing. So patterns that can detect these types of faults are not very efficient in testing real faulty devices. For example, this model does not provide any information on the size of the fault, which is an important factor in this context.

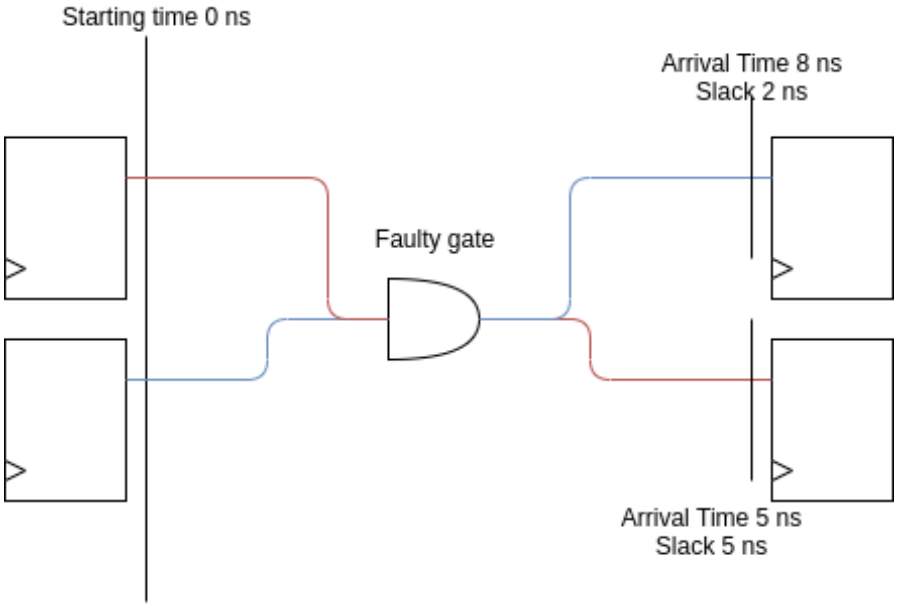

Figure 2.1: Two differents path crossing the same faulty gate

In the example shown in figure 2.1, there are two test patterns that can detect the same faults. The first one is propagated by a path that has a nominal slack of 2 ns (coloured in blue in figure 2.1) and the second one of 5 ns (coloured in red). The **Slack** is defined as the maximum extra delay admitted to violate the setup time of the memory element to which it is connected. This information is very important because while the first pattern is able to detect a minimum extra delay of 2 ns on that gate, the second one is only able to detect 5 ns. This is a big deficiency in this model, because there are three possible situations that can happen in a real circuit that depend on the extra delay defect present on the gate:

- **Value of real delay greater than 5 ns**: the fault generates probably a failure in the system, but the two patterns are able to detect it.

- **Value of real delay between 2 and 5 ns**: also in this case the fault violates the setup time of the system because it is certainly greater than the slack of the **critical path** (path with the smallest value of slack), that has a slack that can be minor or equal to 2 ns. However, only the first pattern is able to detect this faults with a at-speed test.

- **Value of real delay smaller than 2 ns**: no patterns will detect the fault and if the critical path through this gate has a slack that is smaller than 2 ns the fault will probably provoke a failure.

This simple example demonstrates that this model is not very efficient, and introduces the concept of path delay model. In this model there is an implicit information that substitutes the size of the fault, solving this issue.

2.1.2 Path delay faults

The path delay faults can be considered as an evolution of the transition model, since they can represent both individual and distributed faults.

For this model, a circuit is marked as faulty when the sum of the delays of all the gates in a path is different from the nominal one. To detect a fault of this type a pattern must be composed by 2 vectors as for the transition delay faults. The pattern must be able to launch the transition on the first gate of the path, which is defined as the **start point**, than the transition must cross all the gates and arrive to the output, defined as **end point**. A path is not only defined as a set of gates, but also as a set of all possible transitions that can be generated from a set of gates. This concept is introduced with the assumption that each type of transitions (*str* or *stf*) can have a different value of delay, and therefore all possible combinations must be evaluated.

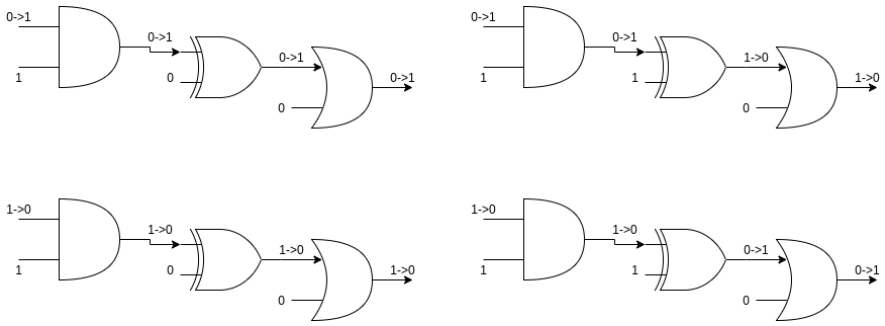

Figure 2.2: Four different delay path in the same topological path

It is visible from the example in figure 2.2 that four different delay paths - where each of them could have a different total nominal delay - are generated from the same topological path.

Another example to explain a critical point of this model is taken from the c6288 study in [12]. The c6288 is a 16x16 multiplier characterized by a symmetric structure. As shown in figure 2.3, all the squares represented are full adders. On the top of the figure it is visible that each full adder has 4 topological paths. So for each path two types of faults can be generated, making it eight in total. On the bottom of the figure, the 15 different longest paths are highlighted. They are all potentially equal one another but they probably require different patterns to be

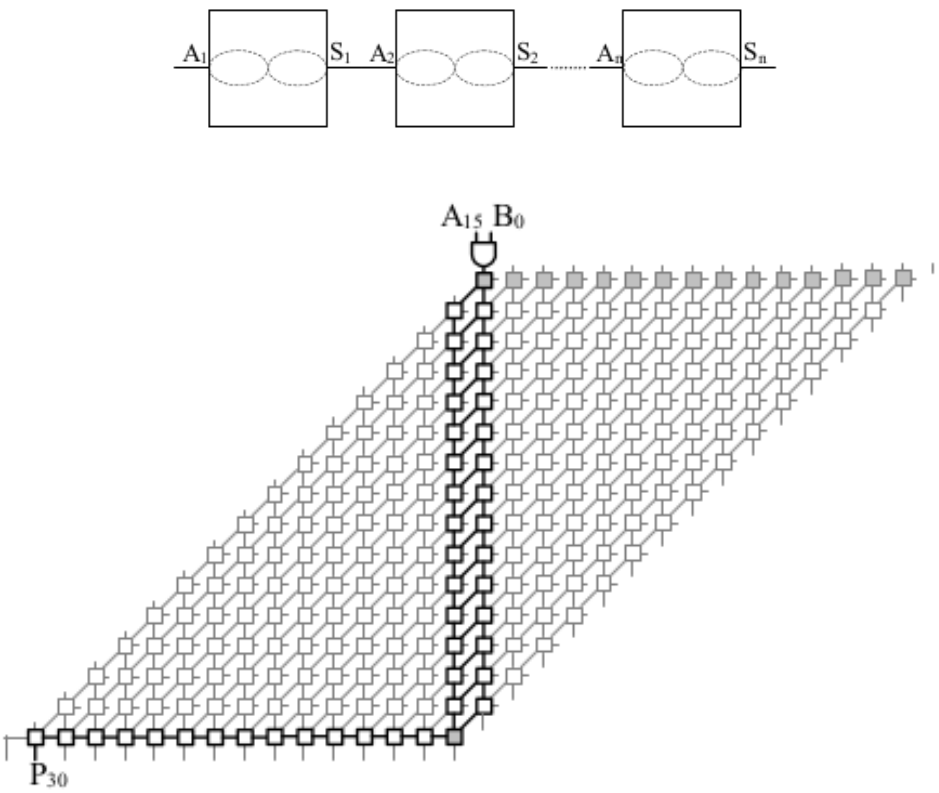

Figure 2.3: Simplification of each full adder on top and c6288 longest path on bottom

sensitized. Each path is composed by 30 full adders, so there are $30x15x8 = 3600$ different faults in this case.

This demonstrates that the topological paths need to be filtered before being tested, because they grow exponentially respect to the number of gates of the circuit. Usually it is important to test the path with a small slack because - in case of extra delays - this would probably violate the setup time. In some cases it is also important to detect the shortest paths as, in case of a deficit delay, they can violate the hold time.

The table 2.1 below summarizes the difference of the two models explained in this section. Path delay model better reflects the faults that can happen in real system. While it is very important to test them, it is also very difficult. Classifying the models according to their testability could help to reduce number of faults on the fault list.

Delay fault model	number of faults w.r.t. number of gates	faults that can be tested	size of detectable faults	test generation
transition	linear	lumped at gate	large	modified stuck-at ATPG
path	exponential	distribuited along paths	small to large	hard

Table 2.1: Comparison of different fault model[1]

2.2 Classifying path delay fault

Path delay model introduces another problem, each fault can be detected with tests of different quality, and not all paths can be tested with a high quality. For this reason, it is necessary to classify the path by its sensitization criteria: robust, non-robust, functional unsensitizable and functional sensitizable.

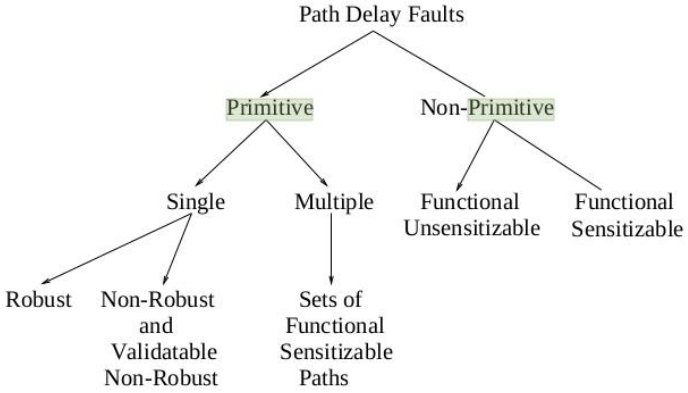

Figure 2.4: Path faults classification, scheme taken from [1]

By the classification presented in figure 2.4 it is possible to understand that there are some faults that do not need to be tested. Primitive faults represent faults that have to be tested in order to guarantee the temporal correctness of the circuit. Non-primitive faults can never independently affect the performance of the circuit, so they must not be taken into consideration, as testing them is useless. Following the tree there is the distinction between single and multiple faults. The single faults are testable individually, this group of faults is the one that is analysed in this thesis, because the ATPG of Tetramax provides patterns to detect them using high quality tests. Multiple faults can create a failure only if there is another specific fault that puts the circuit in some condition making the device transparent to the target fault.

Before progressing in this section some definition must be clarified: [1]

- **Controlling value (cv)**: this value is given to an input to control the output,

the controlling value can determine the output value regardless the value of other inputs.

- **Non controlling value (ncv)**: it is the complement of the cv. In And-gate '0' is a controlling value, because if an input has that value the output is always '0'. Then '1' is a ncv.
- **On-input**: It is an input that is directly on the path.
- **Off-input**: It is an input that can control a gate on the path, but is not an on-input.
- **Sensitizable path**: A path P is sensitizable if there exists a pattern that can propagate a transition on the path.
- **Static sensitizable path**: A path P is defined static sensitizable if there exists a pattern such that all off-inputs in P settle at respective non controlling values under the second vector of the pattern.
- **False path**: A path that can never propagate a transition to one output. A false path cannot be sensitized.
- **True path**: A path that is not a false path.

The following subsections will detail the different types of fault.

2.2.1 Robust testable delay faults

The robust testable path criteria is detected in an independent way respect to the delays occurring on signals other than the target path. To understand the definition of this criteria, the definition of **robust off-input** must be introduced. Calling f an on-input, the relative off-input g can be define as robust when these two conditions are respected:

- If on f there is a cv->ncv transition, then g must have a cv->ncv transition or a stable ncv.
- If on f there is a ncv->cv transition, then g must have a ncv value

As is presented in figure 2.5 the off-path in the example is a robust one, because the two conditions explained above have been respected. A path is defined robustly testable if there exists a pattern that guarantees that all the off-input of the path are robust. This pattern ensures the test is performed with a high quality because it is not influenced by the presence of other faults and the test can detect the targeted faults.

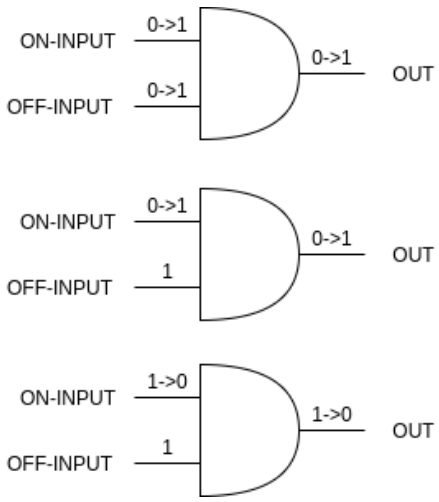

Figure 2.5: Verifying robust off-path condition

2.2.2 Non robust testable delay faults

Non-robust testable path can be tested by patterns, but the quality of this test is not very high, because any fault tested in this manner can be masked by another delay fault on the off-input. The condition of non robust off-input is the following: if there is a cv->ncv transition on the on-input there must also be a transition ncv->cv on the off-input. If a path is composed by a gate that has at least one non robust off-input, while all others are robustly testable, that path is still considered to be a non robust testable path. An example of non-robust testable path is presented in 2.6.

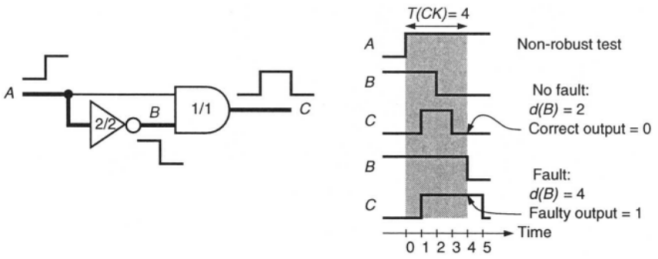

Figure 2.6: Example of a non robust testable path [7]

Another fault happening at the same time can cause masking of the target fault. It is explained by the example in figure 2.7. The transition on the 'd' wire comes before to the 'c' one, so they are not synchronized. This means that there is an interval of time that they are both high, this situation allows the propagation of the transition from input 'c' to the output 'e'. The path to be tested is the one

that passes a-c-e wire. If a fault affects the d wire provoking an extra delay in that wire, the output will remain fixed to low value. For this reason, this type of fault needs to be validated. For this scope it is necessary to generate a pattern that is able to test the transition delay on the d wire and the target path delay, if this pattern exists, then the fault can be validated.

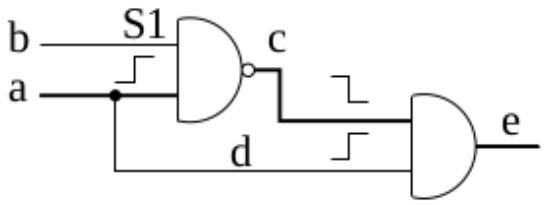

Figure 2.7: How a non-robust testable path can be masked [1]

2.2.3 Functional sensitizable path

These types of faults require other faults to be detected, these paths are not static sensitizable, but functional sensitizable. This means that they require others signal to be delayed for them to be detected. Usually "functional sensitizable off-input" is defined as that situation in which a ncv->cv is generated on both on and off inputs. If a path is composed by only robust, non robust and at least one functional sensitizable off-input, then it is a functional sensitizable path.

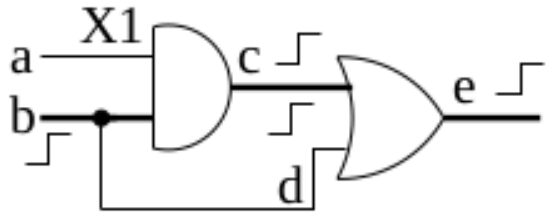

Figure 2.8: Example of functional sensitizable path [1]

As shown in figure 2.8 if in a free-fault circuit the transition on 'd' comes before the transition on 'c', then it masks the propagation from 'c' to 'e', due to the cv transition on 'd'. If for some reason the transition in 'd' is delayed it is possible that it makes the gate transparent to propagate the transition from 'c' to 'e', allowing to detect the fault on the targeted path.

This is a low quality type of test for fault detection because it depends on other extra delays that are random. With the method explained in this thesis they are not taken in considerations because the ATPG used cannot generate patterns for this type of testing. In the commercial world there are few algorithms that are able to do this work [1], because they need to generate patterns that are able to test the fault in the off-input and at the same time detect the fault on the on-input (that is valid only if the first fault occurs).

2.2.4 Untestable path and useless path

There is a high number of topological paths that are untestable. If does there is no pattern that can test a fault, it will be marked as untestable. One of the common set of paths in this category are the convergent ones, because the convergence limits the number of input usable to sensityze the path, decreasing the probability that a test pattern exists. The family of faults crossing a combinational module that results as false paths are named structural untestable paths. They can be eliminated from the fault list, because in real devices they cannot occur.

In sequential circuits, especially in circuits with a pipeline, there are many faults that are useless to be tested. To understand the next concept some definitions must be introduced. Usually a sequential circuit can be divided in combinational part and sequential part, as it is presented in figure 2.9. The combinational netlist

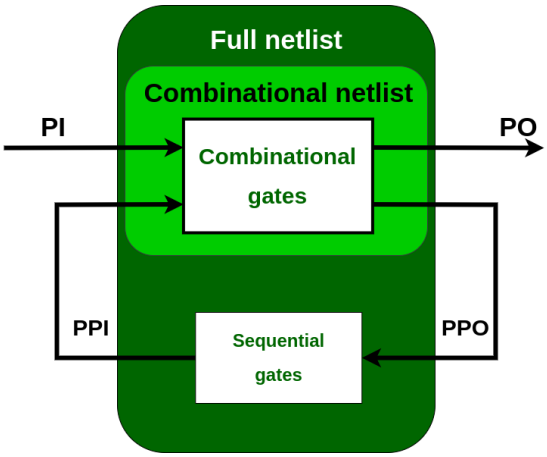

Figure 2.9: Huffman model

is composed of only combinational gate, it has a lot of inputs and outputs that can be subdivided in two big groups:

- **Primary inputs (PI)**: They are the input of the total sequential circuit.

- **Pseudo primary inputs (PPI)**: They are the input each single combinational part of the circuit, they are connected to the output of a sequential gate.
- **Primary output (PO)**: They are the output of the total sequential circuit.
- **Pseudo primary output (PPO)**: They are the output of the combinational part and they are connected to the input of sequential gates.

Obviously, the path crosses only the combinational part of the total circuit. In a circuit with a pipeline the user can control only the primary input, and so he can control the PPI by propagating some value on the circuit. It is important to understand that not all the configuration of PPI can be set by changing the PI values. These faults are untestable for SBST for example, but in every case they should be removed, because they will never produce a failure in the system, since they are not stressable during the functional life of the circuit.

This part underlines the importance to classify the paths and the patterns as if they were bound together. The scope of the classification must be to create test patterns with high quality and avoid testing useless paths. The classification of the path for the scope of this thesis is left to a commercial tool as will be explained later. The next section will explain how the circuits are physically tested.

2.3 Test application

There are lots of methods to test these types of faults in the real world. This section will begin by introducing a process based on scan chains that is valid for testing real circuits. However, as they present some issues, based on the considerations that will be explained, the functional test (another possible type of test application) is often the preferable to the scan test.

2.3.1 Scan chain

The scan chains are born to transform the sequential circuit in a combinational one. This is due to the fact that the automatic test pattern generation tools (ATPG) have some problems working with sequential netlists. In sequential mode the ATPG spends a lot of CPU time obtaining an unsatisfactory fault coverage. To solve this problem scan chain can introduce two operative modes: normal mode and testing mode. During the normal mode the SoC works normally, during test mode all its internal memory elements are connected as a chain making them accessible from the external. In this way, the circuit can be used as a combinational one, because all PPIs are accessible from the external, making all memory elements as a point of control and observation.

Delay testing can be performed by this process, but there are some limitations. In general, testing a delay fault requires both applying a pattern composed by a pair of vectors and sampling the outputs at the circuit operating frequency. To do so, three different solutions can be adopted: launch on capture, launch on shift and enhanced scan.

Launch on capture

This technique requires 3 phases:

- Load a vector V1 into the scan chain
- After a clock cycle a V2 is generated by the combinational netlist and it is stored in the chain. In this moment the transition is formed.
- At the end of the combinational gates a third vector V3 is created (that is the captured one), it is stored on the chain and unloaded from it.

This solution does not allow to have full control of the second vector: the first one can be chosen by the test designer, the second one is generated by the combinational part of the circuit. This reduces the efficiency of this method.

Launch on shift

This technique is a little bit different from the previous one presented; it requires three phases:

- Load a vector V1 into the scan chain
- The combinational result due to V1 is thrown away, the V1 vector is shifted in the chain generating the V2.
- At the end of the combinational gates a third vector V3 is created (that is the captured one), it is stored on the chain and unloaded from it.

Also in this case it is difficult to perform tests that find high coverage, because it is not possible to manage both the two vectors directly.

Enhanced scan

Each memory gate is substituted by a latch and a flip flop, in this way in each memory element can be stored two bits. This allows to save the two chosen vectors directly by the test designer. Also in this case it is managed in three phases:

- Load two vector V1 and V2 into the scan chain, and V1 is connected to the input of combinational netlist.

- V2 is shifted in the PPI of the circuit generating the transition. The output obtained by V1 is thrown away.
- At the end of the combinational gates a third vector V3 is created (that is the captured one), it is stored on the chain and unloaded from it.

The first two methods are not highly efficient because they do not give the possibility to completely control the two input vectors but only one, on the other hand, enhanced scan allows to do that. There are some issues bound in this process, the first one is that to do this type of test the circuit must be prepared by substituting flip flops with the chain FF (with the necessary connections). The second one is that this process can potentially generate transitions that are not normally generated by the operational work, this problem can cause overtesting. A circuit is overtested when it is marked as faulty, but it is able to work well. Moreover, uploading and downloading scan chain consumes a lot of time, making this technique very slow. For these reasons, functional technique is explored.

2.3.2 Functional Test

In this thesis the Functional testing is the chosen test method. This technique recognises if the device is working well by applying functional inputs to the DUT and observing the output. This type of test is the opposite of what is introduced by Design for Testability. The circuit that are tested functionally are designed only to do what they have to do, there are potentially no hardware internal module used to test the circuit. So, the idea is to test the circuit controlling the input and watching if the outputs are correct. It is also different from the structural test, because a circuit can be potentially tested without knowing its structure.

Nowadays it is commonly used to test many devices as ICs, boards and systems. The main cause of its utilization is that it can cover some faults that are not covered by other types of test. Moreover it can be utilized for end of manufacturing tests, incoming inspection and in-field tests. The last one is the most important one because it is the strong point of this solution. Potentially there are solutions that allow the circuit to run autotest during its life to verify other faults that affect the circuit for a limited period of time. However, the execution of this type of test is difficult, because there are several constraints on input and output signal.

There is a special functional test that is the *Software-based self-test* abbreviated as SBST that is able to test processors and SoC. It is used for end-manufacturing and for in-field test. A test of this type requires a program, which is compiled and saved in the instruction memory, before being run. The instructions sequence of this program generates some patterns in the processor that induces transition crossing the gates. While the program is in execution, the outputs are observed to understand if there are some faults.

This kind of test has a lot of merits:

- It can be performed at-speed.
- Reaches good fault coverage catching a high percentage of defects.
- Manufacturers use it more often as final test.
- Often it can be used as an in-field test.
- It has a relatively low execution cost.
- The DUT has not performance drop due to the test.

However, it requires suitable stimuli and ad hoc mechanisms that allows to access to the DUT during test.

This type of test can only detect the faults that can really affect a device during its work. Since the processor usually works with its assembly instruction, it can detect only real faults that can occur on the operational phase of the processor. There is no possibility that a non functional fault will be detected, because illegal patterns cannot be set on the PPI. This also eliminates the possibility of overtesting the circuit. For example, in scan chain there is the possibility to load in the PPI some patterns that are not functional, they can detect some faults that might mark the device as faulty, despite it maybe be working properly. Since those non functional faults cannot be generated on the normal use of the circuit, they never cause failure.

Chapter 3

Functional Simulation flow

This chapter explains how the assembly program can be compiled and simulated in a sequential circuit in order to compute the total fault coverage. This innovative functional simulation flow was developed in previous works [5], because there is not a unique commercial tool that is able to simulate test for sequential circuit using the path delay fault model and SBST technique. This flow is organized in different steps that are summarized in the flowchart in figure 3.1, each of them is handled in this chapter explaining how and why they should be performed. Moreover, each phase needs a different commercial tool that has to be used in the correct way to obtain good result.

This tool gives the possibility to run each steps separately, so the user can execute only one step or a group of steps as he wants. This is a very good point, because it allows to create script to manage different situation, for example a single step can be executed more times using loop, this strategy can be applied for path extraction as it will be explained in next chapters. Moreover, the complete flow is used to evaluate the assembly program generated by the method (that is the most important part of this thesis).

3.1 Synthesis

As it is possible to see in the flowchart the first step to do is the synthesis. Having the rtl-level description, the devices under test can be synthesise, that means produce a structural description from a behavioural one.

The synthesis process can be done in some steps: infer the logic and state elements, perform the technology-independent optimization, map the elements to the target technology library and finally perform the technology dependent optimization. This process is done by synthesis tools that are able to produce a gate-level netlist of the DUT. The tool chosen is design_vision that is produced by

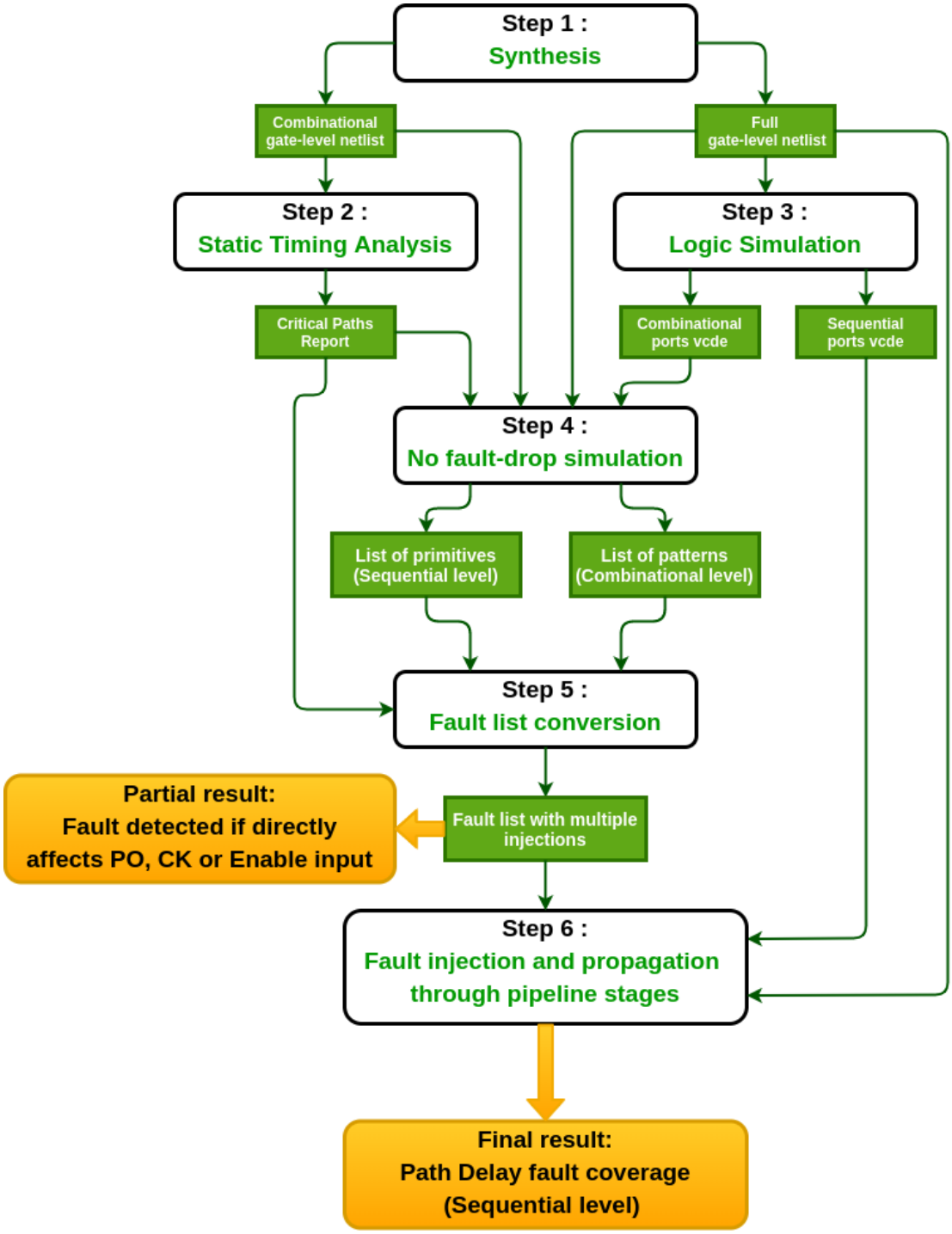

Figure 3.1: Flowchart of the tool from [5]

Synopsis, it supports *VHDL*, *Verilog* and *SystemVerilog* as description language. Moreover it can be used with the graphical interface or using the command line making scripting easier. The technology library used is the Nandgate that is used a lot in academic research.

The goal of this flow's step is to generate two different netlist, the .sdf and .sdc file. The first netlist is composed by all combinational cells of the device under test, instead the other one is the top level netlist, it contains all the components of the devices. So on the next steps there is the possibility to work with the combinational level or with the sequential one as it will be explained. The tool is created so that the user can manage a lot of parameters, the most important are the names of the rtl, the output netlist, and the timing and area contraints.

The combinational netlist is very useful for the extracting path, because it is easy to extract path from PPI and PPO without memory elements. However, in some cases this step should not be performed because there are some companies that usually provide the netlist already synthesised to the test designers. Obviously in that case this step has to be skip, and the tool give the possibility to do that.

3.2 Static timing analysis

This section relates the topological path extraction using a STA, the inputs of this step are the combinational netlist, the .sdc and .sdf files, whereas the output is the target path list. For each path two faults are generated, since two possible transition can be generated on the start point: str or stf. The Static timing Analysis is a simulation method that is used to compute the expected value of delay in each node of the netlist. It doesn't execute a logic simulation, but simply sums the delay of each gate defined in the library in order to understand if the circuit can operate at a certain frequency. In this case it is used to classify all the target paths depending on their slack values.

To execute the STA, Prime Time is chosen, the complete tool provides all the script needed to extract the path. Some parameters are manageable by the user: he can choose the combinational netlist, the maximum number of path to extract, the maximum number of path to extract for each end point. In the first version there was the possibility to set a maximum slack of the path extraction, then another parameter is added to impose also the minimum slack. These are two important variables because they allow to the users to extract pattern in different interval of slack. Another feature added in newer version allows to exclude some endpoints from the extraction. So the user can list the PPOs that are not observable in any way, and the tool doesn't extract any paths that end with those output, because they will generate undetectable faults.

Using this step there is not the possibility to study the logic sensitization of the path, that would have been comfortable for the faults filtering. So the path list has to be filtered at a later time, because there are lots of topological path that are extractable using STA but are structurally unsensitizable. This can be resolved by generating a loop that can extract some paths and reject the useless ones, such that a clean path list is generated. This step is usually used to obtain one of the two lists explained below:

- List of sensitizable testable paths that are in a specific interval of low value of slack. This type of paths in a faulty circuit violate the setup time with an high probability.
- List of sensitizable testable paths that are in a specific interval of high value of slack. This type of paths in a faulty circuit violate the hold time with an high probability.

3.3 Logic simulation

To execute this step it is necessary to have the synthesised netlist, testbench files and the test program. The latter has to be compiled then the tool reads the netlist and the testbench and executes the logic simulation returning two .vcde files.

The logic simulation is performed by Questasim that is a HDL simulation environment that supports different languages as Verilog, SystemVerilog and VHDL. It has the task to take the compiled program and simulate the logical propagation of signal imposed by the program. This tool can be launched by GUI or without it, this allows to the user to watch the internal waves of the circuit that is very useful to understand how assembly instructions are propagated through the memory elements of the core. When the DUT is pipelined this task is very important, because the user must be able to set specific values in internal register to test path faults. He can directly change only the PIs of the system using the assembly program, so using the Questasim GUI he can understand how the wave are propagated through the PPIs and he can have a good control of the entire core. At first glance it is difficult especially if the netlist read is the synthesised one, because the name of the PPI are not self-explanatory, an example is presented in figure 3.2

For this thesis the testbench is already provided together with the rtl description of the core. By the testbench it is possible to manage some features of the logical simulation as the clock period and the condition in which the simulation is over. In that moment two .vcde files are created, the acronyms means *Extended Value Change Dump*, indeed they are text files in which the users can dump specified signals. The simulator will save in that file every changing value of those signals marking also in which instant of time their values are changed. The two .vcde saved

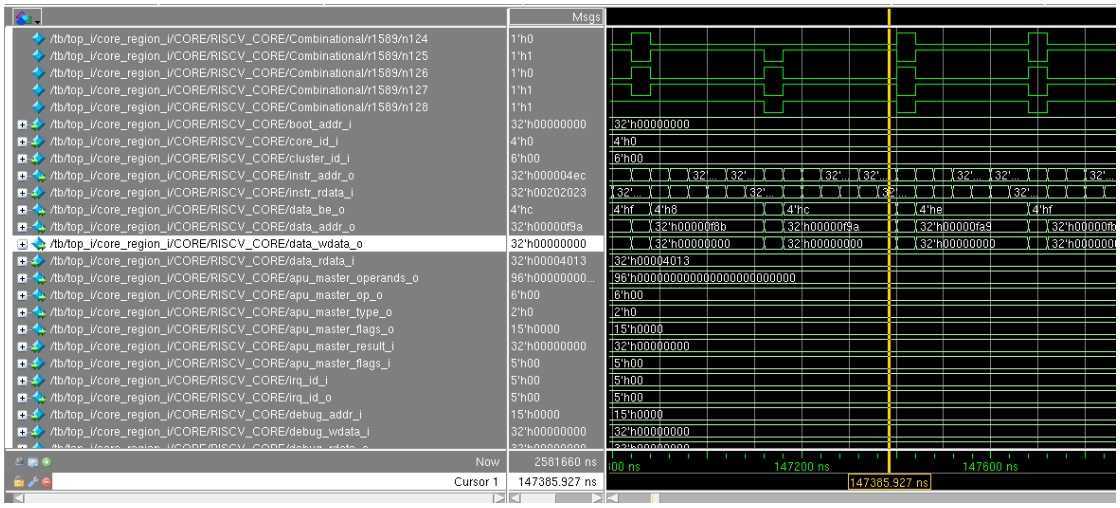

Figure 3.2: Example of logic simulation, on the left is possible to see the signals in a hierarchical way, not all signals are self-explanatory

are the dump of the fault-free devices, in the first one the changing of all registers content are saved, so all the PIs, PPIs, POs and PPOs transitions, differently in the second one only the changing values of the PIs and POs of the system are stored. The first .vcde file will be used in the combinational fault simulation because from that the combinational pattern can be extracted. The second one will be used at final step in the fault injection.

As it is explained before in some cases the synthesised netlist and the fault lists are already provided by the device manufacturer, so the fault simulation can start from this step.

3.4 Combinational level fault simulation

This step often is time consuming because computationally is the most complex. The tool explained in detail in [5] is able to complete the no-drop combinational fault simulation using also the fault simulator created by Synopsys, named Tetramax. The no-drop simulation is based on the principe that if a program pattern is able to detect a fault, that fault is not deleted from the fault list, because a fault detected at combinational level, couldn't be detected at sequential one, as it will be explained soon.

The input of this step are:

- **Combinational gate-level netlist**: This is used by Tetramax to create an internal representation of the fault-free circuit that is used to fault simulate

the device.

- **Full gate-level netlist**: In this step is used only to extract primitives and PO of the top level design, these file will be used in step 5.
- **Path list file**: In this file there are listed all the path that has to be simulated, it can arrive from the device manufacturer or from the step 2 of this tool. Each path of this list has a name that is maintained unaltered, but for each path two faults are generated, one for each possible transition (str or stf) on the start point. Tetramax is able to create a fault list itemizing all the name of path and associating it to the respective transition type, this list is saved in a appropriate file.
- **Combinational ports vcde**: From this file Tetramax can extract the patterns provided by the Logic simulation. Moreover it can understand which are all the correct transitions: for each fault a representation replica is created and the transitions on the PPO are compared with the one saved in the .vcde, if there are some difference the fault can be marked as detected at combinational level.

So, Tetramax takes the four inputs file, creates a replica of the representation of each possible faulty devices and control if there is one pattern that, considering the fault, provokes a PPO changing respect to the data saved by the fault-free device. This implies that Tetramax is able to test only single path delay faults, so it is able to detect robustly o non-robustly testable path fault, in case of non-robustly it is not able to validate them. Each fault is classified so:

- **Detected** : The fault has generated a difference in one PPO of the circuit. So there is a pattern that was able to sensitize the fault, provoking a delayed transition on a observation point.
- **Not detected** : The correct transition was sensitized correctly but this has not caused a difference from the fault-free circuit behaviour.
- **Not controlled** : The fault can be observed by some PPOs, but the program simulated was not able to sensitize the fault. So it is not detected, but potentially it could be detected by other patterns.
- **Undetectable** : For structural reason the fault can't be detected, there are no way to sensitize the path or propagate the faults to observation points

Obviously if a fault is combinationally undetectable it is also sequentially undetectable. So this phase is important also to filter the faults that are useless to test sequentially.

The complete tool provided by [5] is able to execute the no-fault-drop simulation by execute repeatedly the combinational simulation. This tool is very useful especially because allows to manage lots of parameters, the most used are:

- Name of clock pin.
- Strobe offset: used to speed up simulation deleting the pattern relative to the preset of the processor
- Max process: allows to divide the execution of the simulation in different processes, that in parallel can run different simulations part.
- Max detection: max number of detection allowed for each faults, after that the fault is dropped

The result of this step is a file that lists all faults detected, and for each faults there is itemized all patterns that detects the relative fault. Each fault detected in this phase generates a potential failure in the memory element connected to the end point of the target fault. If the end point is a Primary output of the device the fault is also detected at the sequential level, if it is a PPI then the fault has to be propagated to a PO. This situation is evaluated in the next step.

3.5 Sequential level fault simulation

Generally the tool used nowadays are prepared to simulate sequential circuit using scan chain. They usually have two operative mode: the *basic scan* that simulates a circuit prepared with a scan chain and the *fast sequential* that once the pattern is loaded it works in operative mode for a fixed and limited number of clock cycles. These two mode don't guarantee that the simulation results are reliable for this scope. Moreover, Tetramax supports also a full sequential mode, but claims that the end point of each path is connected to a capture point, i.e. a Primary output or a scan flip flop. Indeed this mode was created for other model of faults, in transition delay mode for example after having defined the PIs and POs the simulator can be run in full sequential mode and it is able to verify the propagation of the delay till the primary output.

Given the considerations just made, the last step is performed by ZO1X that is a sequential fault simulator for automotive circuit, but it doesn't implement the path delay testing. Anyway, in this type of simulation ZO1X is used only to propagate faults that are detected at combinational level to a primary output. If a fault is detected by the previous step means that the test program is able to propagate a transition till the end point of a path, this implies that in a faulty device the transition will arrive in late in the memory element, so the memory

element will store a wrong value. The idea of this tool is to model this situation by implementing a bit flip on the endpoint of the target fault in the correct moment and then to simulate the test program in order to understand if this failure is able to cross sequentially all the pipeline and arrive to a PO.

This type of injection has done one time for each combinational fault detection. This motivation helps to understand why the sequential simulation are slower than the combinational one. Indeed it is necessary to speed up the process, so every time that a fault is sequentially detected is dropped from the fault list. Another important parameter provided by the tool is **zoix_interval**: it is a number that represent the interval of time between two launch of ZO1X, it is expressed in minute. It is very important because using the correct value it can be used to speed up the total simulation. Explaining with an example if that value is set to 60 minutes means that the execution of no-drop simulation is interrupted every hours to do the ZO1X injection. Doing the ZO1X injection too often slows the total execution due to the Tetramax-ZO1X switching. However every ZO1X injection allows probably to mark some faults as sequentially detected and so to reduce the fault list speeding up the combinational simulation. The best compromise is dependent on the application and so has to be found.

This step concludes this chapter because once completed the fault injections the simulation is over. The complete tool will save the sequential result on dedicated files, in which the sequential fault coverage can be read, and all faults are classified as sequentially detected or not detected.

Chapter 4

Methodology to generate assembly code

The first step of the proposed methodology consists in the fault list definition. Since paths may involve a variable number of gates they must be classified depending on their slack value. If a path has a little value of slack, a small defect should be enough to cause failure in the device, so small defects in few gates are enough to violate the setup time of the system in this type of paths. This group of fault will be named Long Paths because usually they are composed by a big number of gates. Initially only these ones are taken in consideration.

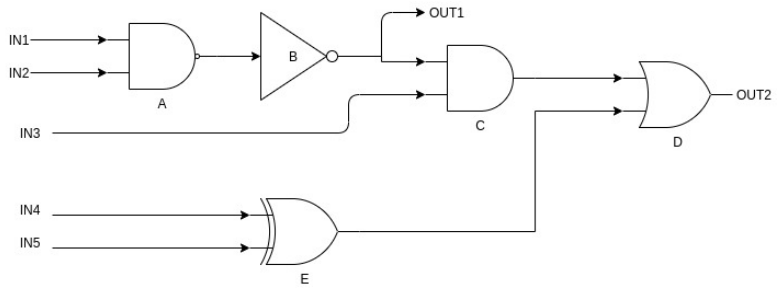

Figure 4.1: Simple circuit

Assuming that the five gates represented in figure 4.1 are all characterized by a delay value of one generic unit of time, and assuming that the five inputs are PPIs of a sequential circuit and the two outputs are PPOs. This example shows the concept presented above. Assuming that the clock period is equals to five units of time, in this circuit there are some paths composed by two gates and others are composed by four gates. In this simplification these paths have respectively a slack of three and one units of time. Intuitively, defects of small entity can cause

the faults on the long path. In the example an extra delay of one unit of time is not enough to cause the fault on the short paths, but it can damage the long ones. Moreover, the variations of the circuit can cause a bigger delay values on a long path respect to a shorter one, so long paths can provoke failures with an high probability.
The first step of this analysis is to understand how a significant number of path can be extracted from a synthesised netlist. Then, the systematic method - to generate programs able to detect the faults - is proposed.

4.1 Extracting Long Paths

The long path extraction needs a Static Time Analizer and an ATPG tool. The Static Time Analizer must be able to extract all the paths concerning a specific interval of slack. The idea is to set a slack interval that includes the topological longest paths, extract and filter them rejecting all paths that can not be tested. This reject operation is performed by the commercial ATPG tool, but it can be performed by any tools able to mark faults as testable or untestable. The process is repeated shifting the interval until a slack limit defined by the user is reached. Therefore, a possible iterative algorithm that can be used for this scope is presented.

```
1  Slack_min = 0 ns
2  Slack_interval = 0.01 ns
3  testable_path_list = empty_list
4  while (target slack is reached)
5  {
6      (path_list, Slack_interval) = path_extractor(Slack_interval,
7          Slack_min, max_num_of_paths, n_endpoints)
8      testable_path_list_tmp = classify_by_ATPG(path_list)
9      testable_path_list.append(testable_path_list_tmp)
10     Slack_min = Slack_min + Slack_interval
11 }
```

The algorithm's result is saved in a file that will contain the path list that can be marked as structural testable in the combinational logic. The two main operator - the path extractor and fault classifier - are explained in next subsections.

4.1.1 Path extractor

The commercial tool PrimeTime is used to execute the path extraction. This tool has one parameter that represents the maximum number of path extractable, this can not exceed 2'000'000, and another parameter used to set the maximum number of extraction for each endpoints. These parameters could cause some

problems, because if they are not calibrated correctly, the process will saturate and the extracted fault list will not include all the long path faults. To solve this problem a recursive function is written:

```
path_list path_extractor(Slack_interval,Slack_min,max_num_of_path,
    n_endpoint)
{
    max_path_per_endpoint =  max_num_of_path / n_endpoint
    path_list_tmp = extract_path(Slack_min, Slack_min +
        Slack_interval, max_path_per_endpoint,max_num_of_path)
    n = count_max_path_per_end_point(path_list_tmp)
    if ( n == max_path_per_endpoint)
        path_extractor(Slack_interval/2,Slack_min,num_max_of_path,
            n_endpoint)
    else
        return path_list_tmp
}
```

The algorithm extracts a group of paths, if in this group there exists a set of paths having the same endpoint with a cardinality that is equal to the maximum number of paths per endpoint, the interval of slack will be halved and the function restarted in a recursive way. The "extract_path" function is performed by the STA.

4.1.2 Classifying Faults

In addition to the path extraction, the objective of this section is to filter only the testable paths from the combinational logic. The other faults are not considered because they have not logical effect on the circuit behaviour and cannot cause failures, moreover using this classification is possible to speed up a lot the test phases. The path list generated by the extractor is analyzed using the ATPG tool of Tetramax for this purpose.

The ATPG tool classifies faults using four different types, as it possible to read in [10]:

1. "**DT - Detected**: 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."

2. "**ND - Not Detected**: An ND fault indicates that test generation has not yet been able to create a pattern that controls or observes the fault."

3. "**AU - ATPG Untestable**: Include faults which can neither be hard detected under the current ATPG conditions nor proved redundant."

4. "**UD - Undetectable**: The "undetectable" fault classes include faults which cannot be detected (either hard or possible) under any conditions."

Only the faults that are classified as DT or ND are taken in consideration using this analysis. Just to compare with the theory presented in previous chapters, the obtained faults include all the path delay faults that are primitive and single in figure 2.4. It has to be specified that these faults can be tested in combinational module, but this does not imply that they are functionally testable, because the ATPG can use all the PPIs to find test patterns. This means that a sequence of assembly instructions could not exist, since they could not be functionally tested in the sequential logic. The strategy studied to execute the translation - from patterns for the combinational logic to functional patterns - will be presented in the next section.

4.2 Building strategy to create testing program

The solutions proposed to create test program can be partially automated, but not completely. The long paths are composed by many gates (some paths are composed by up to 200 gates), indeed - to propagate a transition throw that path - it has to control lots of input pins using a single functional pattern. In order to set many pins at the same time, the sequence of assembly instructions to be used is very specific for each type of fault. This problem generates some others:

- The pure random program results inefficient. Each test pattern for path delay is composed of two vectors, that are translated in two assembly instructions. Each instruction usually is composed by opcodes and operands that can be "immediate" values or saved into register. In order to test long path faults almost all bits of the compiled instruction are important and specific to test the fault. If one single bit is set in the wrong way, the fault could not be detected. The random program should guess all the bits of the two test instructions and also the operands to load on the registers to detect the fault. The efficiency of random programs grows up if it is possible to suggest the correct sequence of instructions to the assembly generator, so that only the operand are chosen in a random way.

- The assembly program written for other faults model are highly ineffective. This is not intuitive, since a common thought could be that an assembly program that is written to test transition delay faults is also able to detect a good part of path delay faults. However, this is partially true, these types of programs are able to detect a good part of short/medium path delay faults, but they are inefficient for long ones. This probably happens because all the transitions can be propagated through lots different paths, and it is more

probable that each transitions is detectable by the shortest one. In general this type of programs can be included in the random type, so their inefficiency is connected to the consideration explained above.

- The ATPG tool can not merge many faults generating few patterns that detect lots of faults. The ATPG tool has to use a big number of pins to cover a single fault, so there is a limited amount of free pins that allow to detect other faults using the same pattern. This last point is a strength for the strategy explained below.

All these considerations were not formally proved, but they were verified heuristically on the device under test

The method that is created to solve these problems is summarized in the scheme in figure 4.2. It is composed by a loop that usually is solved in two or three cycles obtaining good results. It is subdivided in steps, the first one simply requires another path classification.

4.2.1 Path classification by module

This phase takes as input the entire path list extracted by the STA and subdivides it in different path lists. In each path list there are only paths that cross the same module. This operation simplifies the interpretation of the ATPG patterns. Usually the ATPG try to test multiple faults using the same pattern. This is useful if the merged faults cross the same module, otherwise it is difficult to translate a single pattern using one single couple of assembly instructions.

The classification can be done simply analyzing the file in which is saved the path list. In this file it is possible to see the hierarchical structure of each gates, using this information each path can be connected to the module crossed. For each module - involved in this context - a custom path list is created.

4.2.2 Generate assembly program

This subsection explains how are performed the steps 2a, 2b and 3 of the flow presented in figure 4.2. As it will be explained, this process has to be repeated more times to obtain an effective assembly program.

Build a scheme of instructions using logic simulator

The objective of this step (2a) is to select which instructions are the most efficient to stress the target module. The logic simulator is used at this scope, the waves relatives to the module under test has to be added to the Graphical User Interface

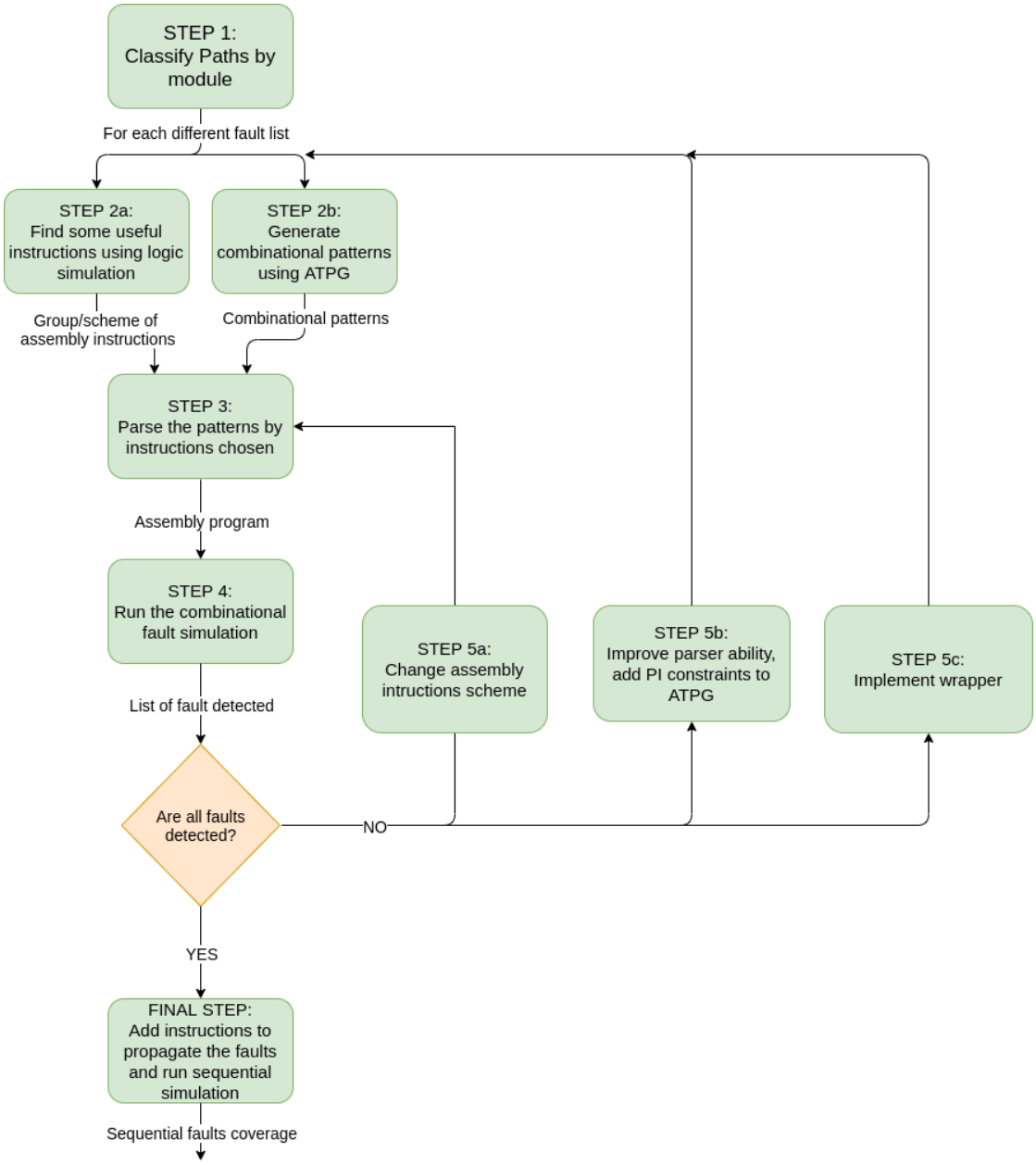

Figure 4.2: The systematic method that allows to generate efficient assembly programs

of the simulator, and they are analyzed to understand which instructions can control the input pins of the module. These instructions has to be selected and saved.

A typical template, which can be used to translate each combinational pattern, is the following:

- **LD instructions**: The goal is to set the value in registers that have to be used in the test instructions. If the analyzed fault belongs to the decode stage, the ATPG will indicate which values has to be set in which register, instead if it belongs to other parts of the pipeline it is necessary to save in registers the operands of the target unit. For example, if the fault cross the execution unit, the pattern will indicate which transition has to be applied to the ALU or MULT operands.

- **First test instruction to set the transition**: it is the real first instruction that provokes the transition, each pin is set to the value requested by the first vector of the pattern. It is important to use the registers where the operands are saved in the correct order.

  This instruction can be extracted from the ISA. For example, if the fault cross the decode stage, the correct one will be indicate directly by the ATPG, in other cases it has to be derived from other signals, for example the alu opcode.

- **Second test instruction to launch and capture the transition**: Using this instruction the transition is generated and the result is captured in combinational level. This instruction has to be able to write the correct values on all PPIs relatives to the second vector of the pattern.

  All the consideration of the previous point are valid. If the translation is done in the correct way, the fault has to be detected by the combinational simulation after this instruction.

- **Store instruction to propagate the result**: the goal is to propagate the fault in the memory. A strategy is to save in memory the endpoint of the path, in order to save a wrong value in case of failure. In this way, the test can observe the fault. Another strategy is to execute the store only if the device is working correctly.

The output of this step is the correct scheme of instructions and also a group of instructions that probably are more suitable to preset, launch and capture the correct transition.

Generate combinational patterns

This is the step 2b of the flow and can be execute in parallel respect to the previous step presented. In the first cycle of the flow only the combinational netlist and the fault list are given to the ATPG, the latter elaborates them and returns a set of

patterns to detect all faults. Since all faults are potentially detectable using the filtering operation explained above, the *ATPG effort* must be set to a value that is able to find all the necessary patterns. The *ATPG effort* is a parameter of the commercial tools, it is used to set how many attempts are allowed to generate a test patterns. If the ATPG can not find the pattern using the attempts granted, the fault will be marked as *aborted*. So, if the *ATPG effort* is set to high value the ATPG can detect many faults, but it requires long time.

The output of this step usually is a .STIL file where there are listed all PPIs involved and the relative patterns. Since patterns generated are generated for the combinational logic, indeed these patterns have to be translated in functional ones using an automatic parser.

Parse patterns in assembly instruction

The scope of this section is to translate the combinational patterns generated by the ATPG into functional patterns. Using the scheme of instruction previous selected, it is possible to write a *parser* that do the translation in an automatic way. The parser should be able to read the stil file and generate an assembly program, the resulting assembly program must apply the functional patterns to the device.

This step is the critical one in the flow, because it is really difficult to generate a parser that is able to summarize each transitions needed on all pins into assembly instructions. In many cases this is not possible. For example the ATPG may ask some values on a special refister of the processor, that are not always controllable by means of assembly instruction, or the ATPG could ask to launch a transition using an illegal instruction and to capture it by a normal instruction. This is not possible because the processor will generate an exception between the two test instructions, which does not allow to execute the second instruction that generates the transition requested.

To translate the patterns using the instruction scheme presented above have often produced good results. It is used as a starting point, because it helps to obtain a coverage that is greater than zero by the first simulation, this is necessary to evolve the process. Indeed, if the DT faults are analyzed it is possible to understand which input vectors are translated correctly by the parser. To improve the coverage and the parser efficiency it is possible to follow one of the three ways presented in figure 4.2 and thorough in the next subsection.

4.2.3 Enhance fault coverage

This subsection explains the steps 5a, 5b and 5c of the flow presented in figure 4.2. The step 4 is not thorough because it is the combinational fault simulation that is already presented in 3.4 and it will not be repeated.

Three feedback branches of the flow are proposed to increase the coverage. The easiest one is performed by changing the the operands or the test instructions (step 5a). It can be performed also in a pseudorandom way, which is very easy to be applied and returns good results in a short time.

Improve parser ability

Creating a good parser is a complex task because the synthesizer assigns alphanumeric acronyms to the circuit nets; this creates some problems in the identification of which assembly instructions are more suitable then others. Moreover, if a PPI is useless for the purpose of launching the transition, the ATPG tool marked it with the "N" letter. This situation causes some issues in the automatic generation of assembly code because if that letters are not translate correctly is possible to generate illegal instruction and to manage the illegal instruction can create some problem to capture and to propagate the transition correctly. In general, the parser optimization is performed trying to expand his ability in order to translate as many PPIs as possible.

This optimization alone is not able to obtain good result, because if the ATPG is unconstrained, it will generate patterns that cannot be translated in assembly instructions.

Adding ATPG Constraints

The ATPG work can be optimized adding constraints, that means to bind the values of some PPI to a static value. In this way the ATPG will try to find pattern changing the values of the PPIs unconstrained.

This strategy is also used to produce patterns that can be now translated by the parser already generated. Observing the waves in the logic simulation it is possible to see which PPIs can be constrained. The candidates are the pins that are not controllable by the assembly instructions in a certain template. It is necessary to pay attention because in this phase some faults can be marked as ATPG untestable, this implies that the pin constrained are essential to test these faults. If this problem occurs, the added constraints have to be eliminated, then it is possible to try to follow one of the other two branches of the feedback flow. This strategy can be used together to the previous one.

Modifying the netlist

This is a powerful solution, because it allows to bind PPI values one each others using specific logic gates or module. Moreover if correctly used it can increase the efficiency of the parser. In order to maintain the high speed of the process, the gates or the modules added must be combinational.

This solution is explained using an example that is applied in the patterns generation of the test that are reported in the next chapter.

The Nandgate library is used for the core synthesis. The flip-flops in this library have two output signals, Q and QN, thus one is negated with respect to the other. So each gate shows two pins that results as PPIs of the combinational module, like is showed in figure 4.3.

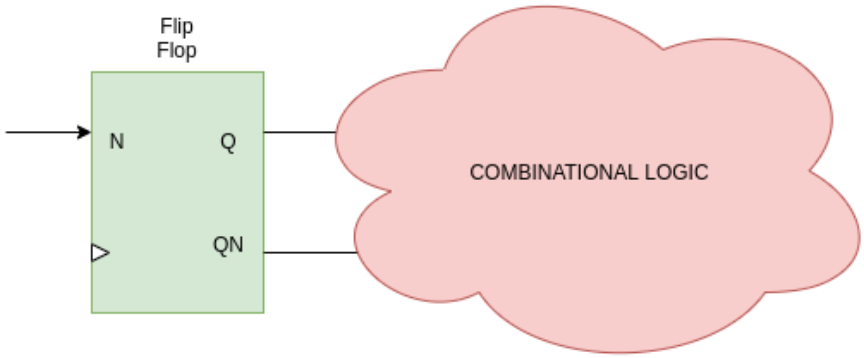

Figure 4.3: Normal Flip Flop

This situation creates some problems because the ATPG tool reads only the combinational netlist, so it does not know that the two PPIs must have opposite values, it could decide to find solution applying the same values to the two pins. To solve this problem, the Flip Flop are modified as is presented in figure 4.4. A not gate is included in the combinational logic and excluded from the register. This special netlist is used only for the ATPG and parsing operation, instead the original netlist is analyzed to perform the simulation.

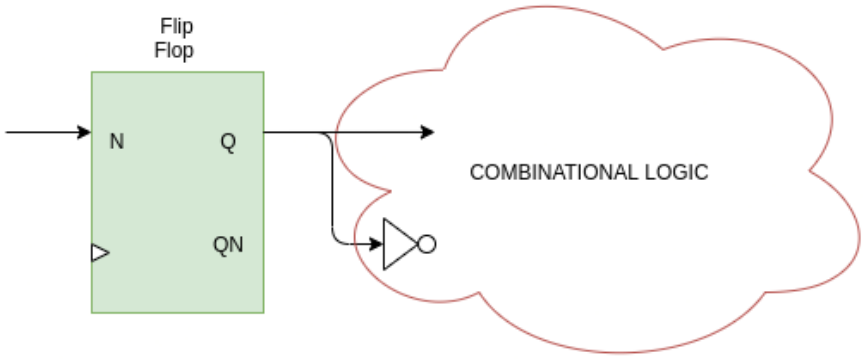

Figure 4.4: Modified Flip Flop

In this way the ATPG can only act on the Q wire of the register. While the other pin will be set to the complementary value. Moreover, the parser will have less bit to manage, which could simplify his work.

Some hints are provided to generalize this solution: if a decode module is written before each stage of pipeline, the generation of functional patterns will be much easier since the ATPG will return directly the instructions, so this could speed up a lot the automatic process and make the parser much more trivial. However the human effort will just be moved to the decode modules which would be written by hand. This would mean that a deep knowing of the processor at the RTL level is requested. For this two reasons this solution is often rejected. In general, good results can be reached also using hybrid solution, i.e. trying to increase the fault coverage using two of the strategies presented.

4.2.4 Sequential fault simulation

Every time that a flow cycle terminates the combinational simulation is performed to evaluate the efficiency of the program written. If the combinational fault coverage is sufficiently high, the new challenge becomes to find an instruction that allows to propagate the faults to the memory or to a PO of the system. Usually this is solved simply writing a store instruction after each pattern that is able to discriminate a fault-free device from a faulty one. Then the complete sequential simulation is done using the tool explained in previous chapter. This is the final step of the method because it is performed to evaluate the assembly programs written. In general, it gives very good results that are presented in next chapter.

Chapter 5

Experimental results

In this section presents the results obtained according to the method explained in the previous chapter. Moreover, other assembly programs are evaluated to understand if there are connections between different models of fault, although the result obtained are corrupt. All steps of the method are presented in this chapter to show a practical example.

5.1 Testing long Path delay faults

5.1.1 Extracting path using path extractor

The netlist of the riscv core included in the Pulpino Soc has been synthesised using a 5.0 ns clock constraint. An in-depth search has been performed scanning the whole slack range, splitting it into sections of variable width, as it is explained in chapter 4. For each of these sections, a number of paths has been extracted and analysed using ATPG techniques with reasonable efforts. After the analysis, according to the classification performed by the ATPG engine, some faults have been retained (those that have been detected and the aborted ones) while some others have been discarded (those marked as structurally undetectable and ATPG untestable).

The first testable path according to the analysis showed a slack of 1.3 ns. In order to focus only on a relevant path (timing-wise) it has been decided to take into account just a portion of the slack range evaluated as:

$$Range\_min = 1.3ns$$

$$Range\_max = Range\_min + (clock\_period - Range\_min) * 0.3$$

This has lead to set a *Range_max = 2.5ns*.

Other paths ranging in the slack set [2.51 ns - 5.0 ns] have been assessed as not of interest.

Within this range, a total of 10018 faults belonging to both detected and aborted classes have been found. They are saved in a fault list, that is ordered by crescent slack and for each path two faults are generated. For each fault the number of the gates of the path and the number of the pins needed to propagate the correct transition on the path is counted. A piece of the result data is reported below.

```
NAME OF PATH          N_GATES     N_PIN
**default**_426       99          106
**default**_426       99          84
**default**_427       98          68
**default**_427       98          98
**default**_428       98          68
**default**_428       98          98
**default**_429       98          68
**default**_429       98          98
**default**_430       99          196
**default**_430       99          74
**default**_431       99          196
**default**_431       99          74
**default**_432       98          143
**default**_432       98          73
**default**_433       98          68
**default**_433       98          98
**default**_434       98          68
```

The means of all values are reported in table 5.1.

Number of faults	Number of gates per paths	Number of pins per paths
10018	88.59	104.95

Table 5.1: Considering Long path faults, the mean values of the number of gates per path, and the number of pins needed to be set to detect the faults are reported

The path lists extracted are classified depending on the module crossed, as it was explained in previous chapter.

5.1.2 Classifying paths by module

The faults were originated in different parts of the circuit. The main targeted modules are:

1. **ex_stage_i_alu_i_int_div_div_i_add_100**
   (riscv_ex_stage.sv) : 90 faults.
   Adder module belonging to the divisor instantiated in the ALU of the core.

2. **load_store_unit_i_add_463_aco**
   (riscv_load_store_unit.sv) : 158 faults
   Adder module that evaluates the data memory address in the case of a store instruction.

3. **load_store_unit_i_mult_add_463_aco**
   load_store_unit_i_add_463_aco
   (riscv_load_store_unit.sv) : 922 faults
   The additional multiplier is used to compute the operands to be used in the address calculation in some special cases like pre-post increment addressing methods.

4. **r1589**
   (riscv_id_stage.sv) : 8848 faults
   Adder module used to compute the instruction memory address used to fetch the instruction that needs to be executed next.

As it is possible to see, the majority of faults belongs to the r1589 module, that it is also the most complex to be tested.

5.1.3 Generating ASM codes

First group: the divisor

The division and remainder instructions take between 2 and 32 cycles. The number of cycles depend on the values of the operands [9]. This can be a problem because the combinational ATPG process cannot be used in a simple way; for this reason, the scheme of instructions used is the one explained in section 4.2.2. After some load instructions used to set pseudo-random operands of the division, a first instruction is written to pre-set the transition. Observing the wave on the logic simulator, *srl* and *li* are chosen as they have complementary ALU opcode respect to *remu*, that is the instruction that completes the transition and captures the faults. *Remu* is chosen because partial reminders of the division also affects the total reminder of the division, so if a faults occurs in the first cycle of the division execution, probably it is propagated through the other cycles. A final store instruction is used to propagate faults sequentially.

As shown in table 5.2 the first group does not reach 100% fault coverage due to limitations introduced by the functional behaviour of the netlist. As previously explained, these faults originated from the ALU opcode bits and cross the divisor

until reaching the register used to store the partial result during computation. In order to generate the required transition, it is necessary to produce a couple of instructions with the following conditions: the second one must be a *remu* instruction while the first can be selected among a certain number of instructions. To test those paths, it is required to sensitize an internal path of the divisor, but no instruction exists that can access and stimulate the divisor other than *div[u]* and *rem[u]*, which in turn cannot generate a transition on the alu opcode.

This group follows the first *feedback strategy* explained in the previous chapter, in fact the operands are chosen in a random way. Considering that the faults were few, the instruction were written by hand; in other cases a cleverer strategy to choose operands must be developed.

Second group: Adder in Load Store Unit

All the faults belonging to the second group are paths that pass through an adder used to either compute the address of a store or load instructions. This group of faults is easy to test, because all the paths start and end in the Load Store Unit module, that is a combinational module and it can be easily isolated and managed by the ATPG tool. Moreover, the faults did not propagate because all the end points of this group are a Primary Output of the Riscv core, and all start points are directly an operand of the adder. So, the ATPG process is used to understand which operands have to be used to detect the faults, then two instructions are used to launch and capture the transitions: *add* and *p.sw*. A parser written in python has been developed to translate automatically the STIL file generated by the ATPG to the assembly program.
Using this process all faults are detected.

Third group: Multiplier in Load Store Unit

The third group is very similar to the second one, the paths pass through a multiplier that is used as a MUX, then pass through the same adder of the second group. The starting point is an operand bit of the multiplier and the end point is a bit of the memory address. The two instructions chosen to launch and capture the transition are the *p.sw* with post increment and a normal *p.sw* instruction. Using this couple of instructions it is possible to test all *stf* faults, while the *str* are all classified as ATPG untestable, and for this reason are not considered in this analysis. Watching the wave in the simulation it is visible that *p.sw* with post increment is performed in two clock cycles if the sum of the operands is not a multiple of four. This constraint could be a problem because it is necessary to pre-set the value of the transition in one clock cycle only, to control the process easily. The idea for solving this issue is to modify the load store unit module so

that only operands where the sum is a multiple of four can be used. This way *p.sw* with post increment takes one clock cycle only, and the same strategy used before can be performed.

```
1 assign alu_operand_b_ex0=alu_operand_a_ex[0];
2 assign alu_operand_b_ex1=alu_operand_a_ex[0]^alu_operand_a_ex[1];
```

These two lines are added to the verilog file of the load store module: by binding the least significant bits of operand *b* to the least significant bits of operand *a* it is possible to obtain a sum that is always a multiple of four. Analysing this module using the ATPG, all faults are detectable, and so a custom parser was developed in python to translate the STIL file to the assembly program.
All testable faults are detected.

Fourth group strategy: r1589 module

This type of faults are very difficult to be detected, because they cross the adder used to update the program counter. This adder, named *r1589*, as to sum a 32-bit operand with a 12-bit immediate. Initially, it is isolated and an ATPG process is launched on this module. This operation highlights that to cover all the faults is necessary to access to a forbidden part of memory. So to test all these types of fault the user has to handle an invalid memory access exception. Using the synthesised netlist it is not possible to manage this type of exception, indeed an illegal instruction can be managed. So, an illegal instruction is injected from the extern of the core when the Iram Address points to a forbidden part of memory. In this way the exception triggers an handler function that simply returns to the normal flow of execution.

An ATPG process is executed on the combinational part of the processor core, the reading of the stil file is very difficult due to the high number of pins involved in the patterns generated. A parser that simplifies the process was written, it simply associates the name of the pin with the value that the pin must have. Some constraints are imposed to the ATPG in order to write a code that follows the same scheme for each pattern, as was explained in chapter 4. This type of solution has to manage lots of operands that are saved in 4/5 registers using load instructions. The two test instructions are chosen by the ATPG, the first is a *lw* and the second a *jalr* that allow to execute a jump to absolute address, that is one of the operand previously cited. Then, a store instruction is written to propagate the fault sequentially, this instruction is not executed if the circuit is faulty, such that the tester can notice the fault.

The complexity of these faults required that all the branches of the method are exploited. An example of a group of instructions written by the automatic parser

is presented below. All these instructions can reproduce only one combinational pattern.

```
1     asm("li x10,0x0");
2     asm("li x14,0x0");
3     asm("li x15,0x80000002");
4     asm("li x5,0xe1aa9355");
5     asm("li x6,0xe1aa9355");
6     asm("add x7,x5,x6");
7     asm(".word 0xc0003");
8     asm(".word 0xffe78267");
9     asm("sw x15,32(sp)");
```

From eight to twelve instructions are needed for each patterns, furthermore the ATPG process write 3600 different patterns to test all the faults concerning this module, so the assembly code - written using this method - is composed by several lines. Using this strategy, 8858 faults are detected and the remaining 30 faults are ATPG untestable. So the 100% of fault coverage was reached in the simulation on combinational module and thanks to the store instruction they are also functionally detected.

5.1.4 Results and considerations

The approach in general was the same explained in the chapter 4, but it can not be automatically run. It is human time consuming, because lots of time is used to study the waves on the logic simulator, but the efficiency is high. This method generates a set of ASM codes whose characteristics are reported in Table 5.2.

Program Name	Fault group	# instructions	Detected	Undetected	Total number of faults
div.S	1	161	50%	50%	90
load_store_unit.S	2	943	100%	0%	158
load_store_unit_mul.S	3	4783	100%	0%	922
main.c + buono_load_tot.c	4	39718	100%	0%	8848

Table 5.2: Test programs and their characteristics

The table shows that the coverage is good for each module. The total coverage obtained is 99,5 %, and the faults undetected are probably functionally untestable as was explained in previous section, but there are some problems that need to be explained. Each section needs a high number of lines of codes requiring a big memory to store the test programs. This problem is intrinsic in the fault model, i.e. the high number of input pins used for test only a fault does not allow to generate a single pattern that can detect more than six faults at the same time, in some

cases one pattern can also test one fault. So, the ATPG returns a big number of patterns that has to be translated into ASM instructions.

However, the first three groups can be improved, since the two test instructions does not change and the registers where the operands and the result are saved are also fixed. Thanks to these considerations, it is possible to generate a new efficient pattern just changing the operand saved in the registers, creating a program that execute these steps:

1. launch and capture the transition by the two test instructions explained before.
2. propagate the faults through the sequential logic.
3. do some operations on operands to change them in a smart way.
4. repeat from the first point.

The code written using this strategy probably could be very short, but the faults coverage will be reduced and the execution time will increase. A compromise has to be reached between fault coverage, code size and execution time.
The fourth group is hard to optimize. The two tests instructions are various, indeed the patterns returned by the ATPG set the operands, the opcode and also the registers where are stored the operands. So, it is difficult to implement a loop program as in previous cases. As it is possible to see in the table 5.2 the fourth group takes up almost 90% of the lines of the total code, so the optimization explained before are useless in this study.
After these considerations, the fault coverage of these assembly codes is compared with the results obtained simulating other programs, this study allows to understand if there are connection between programs written due to test other model of fault, furthermore it demonstrates that the method is efficient to test long path faults.

5.1.5 Comparison with other model of faults

Six different programs are considered to compare results obtained with different faults model:

1. **Program 1** takes 64502 clock cycles and can be classified as a medium-sized test program. Each functionality of the core is tested using series of different macros. Watching the code it is possible to see that the alu, load and store unit, multiplier are all tested using different macros that are also used to test branches, compressed instructions and register file. The structure of each macros is similar, each one is composed by a sequence of normal instruction followed by some store instructions. There is one specific macro a little bit different from the others, it is used to test illegal instructions and the relative handler.

2. **Program 2** is a shorter than previous one, it takes 36500 clock cycles.
   It offers a test for principal units and it is characterized by two arithmetic instructions followed by a store of the result. The operands are principally written in a random way, therefore there are some operands that are built in a specific ways, for example some operand are written to generates checkerboard patterns. The random parts is built by a big loop that changes only the operands and maintains the same instruction sequence. Moreover also register file, csr and hardware loops are tested respectively by march algorithms and custom algorithms.

3. **Program 3** is composed by 42970 clock cycles. For each target modules a different file is written, the alu, multiplier, crs, branch and register file are all included in this program. In general to test alu and multiplier some operands are loaded on registers and a group of operations are executed, followed by some store instruction. This sequence is repeated in small loop, but in general there is a small variety of instructions: they are part of a small group of the entire ISA. The other target modules are tested in a very similar way respect to what viewed in previous program.

4. **Program 4** is the longest one, indeed it lasts for 181,370 clock cycles.
   The instructions sequence is very similar to the program 2, indeed it is composed by a couple of operations followed by a store instruction that save the result in the memory. Differently of other programs at the beginning of each procedures the register file is reset with some xor operations. This program stress the core using also vector operations and hardware interrupts.

5. **Program 5** is the shortest one, it takes only 17,269 clock cycles to be executed.
   It is similar to all the programs already discussed, but it is composed using a fixed instructions structure: are used two loads to save some operands in registers that are elaborated by an instruction that is chosen to stress a specific functionalities and finally a store instruction.

6. **Program 6** is written in order to obtain an high coverage in transition delay faults. The alu is tested parsing the patterns provided thanks to a decoder that forces the ATPG to find only functional patterns, that are translated by a parser. For testing the other part of the execution unit was written a script that can generate two random instructions preceded by load instructions to load the operands and store to store the results. The script also control if the group of instructions is helpful in improve coverage otherwise they are rejected. Also the register file is tested generating strategic transitions in decoders and encoders connected to it. Internally of the code there are some branches and hardware loops to test also the fetch and decode stage of the pipeline

Each program is simulated using three fault models explained in the previous chapters: stuck-at, transition delay and delay path. The faults of the whole CPU are included in the fault list to simulate the coverage of the first two models, indeed the fault list of the path model includes all the long paths explained in previous sections. In table 5.3 are reported the fault coverage obtained using the simulation of the 6 programs presented before. Also the result of the totality of the top five programs is computed because they all are written for the stuck-at model. To complete the comparison the delay path program presented above is simulated using all the fault lists.

Program	**# clock cycles**	**SAF coverage**	**TDF coverage**	**PDF comb. coverage**	**PDF seq. coverage**
SAF program 1	64,502	86.73%	65.01%	0.33%	0.32%
SAF program 2	36,394	81.79%	44.64%	0.27%	0.27%
SAF program 3	42,970	80.43%	61.44%	0.27%	0.22%
SAF program 4	118,098	82.97%	63.83%	0.4%	0.32%
SAF program 5	17,269	81.37%	63.52%	0.09%	0.08%
SAF total	279,233	90.91%	83.98%	0.52%	0.4%
TDF program 6	23,451	80.5%	74.25%	0.27%	0.23%
PDF program	45,605	38.5%	14%	99.5%	99.5%

Table 5.3: Fault coverage of different programs using different fault list and models: stuck-at, transition delay and long path delay faults

Data are collected in a XY graph to understand if there is a relationship between the different fault coverage obtained, as is presented in figure 5.1, but the number of test is not sufficient to draw conclusion. In general seems that there isn't relationship between data, they seem not strictly correlated.

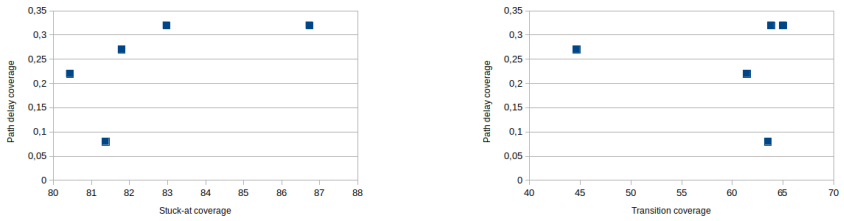

Figure 5.1: XY graph to search relationship between coverage obtained using different fault model

As it is possible to see the coverage obtained is very low considering the path delay model. A data that amazes is the one related to the sixth program. It was

expected the best one in path delay coverage respect to the top five programs. The motivation that hypothetically justifies the bad result is that the program 6 is written for the whole cpu, indeed the long paths cross only some modules. For the same reason the program written for the delay path obtains bad result in transition delay model. To study a better correlation and to confirm this hypothesis the load store adder module is isolated, such that the workflow is repeated to confirm the hypothesis only in a small part of the processor. The results of this experiment are presented in table 5.4.

Program	**Transition delay coverage**	**Path Delay sequential coverage**
SAF program 1	100%	1.01%
SAF program 2	99.75%	0.4%
SAF program 3	98.61%	0.4%
SAF program 4	100%	1.01%
SAF program 5	100%	0.9%
SAF total	100%	1.2%
TDF program 6	100%	0%
PDF program	94.2%	100%

Table 5.4: Fault simulation on Load store adder module

In general, this study confirms that if a certain program is good for transition delay testing, it should not test long path delay faults. On the contrary, a program designed for test long path delay faults through a certain module, it is able also to test transition delay faults. To conclude this section, since there is not a clear correlation between the different model of faults, it is useless to try to obtain high coverage adapting the technique used for *SAF* and *TDF* models also for long path delay faults. Probably this approach can be used for short path as explained in the next section.

5.2 Difference in short Path delay faults

Using the same approach explained in previous chapter, a number of short paths is extracted. A fault list of 10274 path delay is generated, this number is chosen because very similar to the long path fault list, in order to make the two study comparable. An interesting data is the interval of slack obtained in the two cases:

- **Long path interval of extraction:** [1.3 ns - 2.5 ns]
- **Short path interval of extraction:** [4.8 ns - 4.97 ns]

It is interesting to see that a very small part of faults is rejected extracting topological short path, because the majority are marked as testable by the ATPG process. Two faults are extracted For each path generating a fault list that is ordered by decrescent slack. Moreover, the ATPG is used to count how many pins are needed to detect each fault, a small part of the list is reported below to show an example.

```
NAME OF PATH          N_GATES     N_PIN
**default**_217       2           10
**default**_218       2           20
**default**_218       2           19
**default**_219       2           20
**default**_219       2           19
**default**_220       2           19
**default**_220       2           20
**default**_221       2           6
**default**_221       2           5
**default**_222       2           6
**default**_222       2           5
**default**_223       2           7
**default**_223       2           7
**default**_224       2           8
**default**_224       2           7
**default**_225       2           20
**default**_225       2           10
**default**_226       2           20
```

The table 5.5 summarizes all data of short paths to compare the result respect to long ones. Obviously there are few gates for each fault in short paths respect to the long ones, moreover there are few pins that need to be set for the fault detection in short path respect to long one.

Path Type	**Faults number**	**Gates number per paths**	**Pins number per paths**
Long	10018	88.59	104.95
Short	10274	2.99	6.74

Table 5.5: Considering short and long path faults, the mean values of the gates number per path and the pins number needed to be set to detect the faults are reported

The path classification for this type of faults is not performed, because the paths are so short to cross a module.

Moreover all the programs - written for the other model of faults - are tested obtaining the fault coverage presented in 5.6

Program	**# clock cycles**	**SAF coverage**	**TDF coverage**	**PDF comb. coverage**	**PDF seq. coverage**
SAF program 1	64,502	86.73%	65.01%	71.98%	50.1%
SAF program 2	36,394	81.79%	44.64%	73.76%	58%
SAF program 3	42,970	80.43%	61.44%	74.78%	55.7%
SAF program 4	118,098	82.97%	63.83%	76.51%	67.6%
SAF program 5	17,269	81.37%	63.52%	73.16%	51.25%
SAF total	279,233	90.91%	83.98%	82.75%	74.5%
TDF program 6	23,451	80.5%	74.25%	73.6%	56.7%

Table 5.6: Fault coverage of different programs using different fault lists and models: stuck-at, transition delay and short path delay faults

As it is possible to see, the fault coverage are higher than the ones computed for the long paths, each program is able to detect a reasonable part of the fault list. So, the programs - written in order to detect fault for stuck-at and transition delay model - reach also good coverage testing short path delay faults. As in previous case, the data are not sufficient to understand if there is correlation between different fault model. These results are represented in figure 5.2.

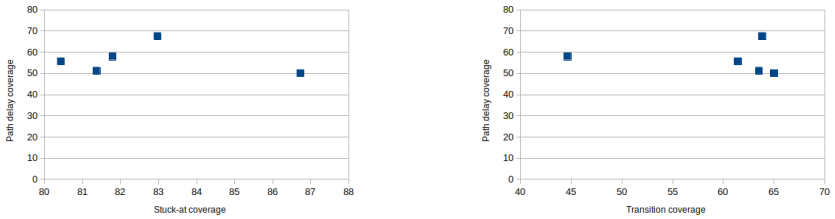

Figure 5.2: XY graph to search relationship between coverage obtained using different fault model

The short path faults extracted present some difficult to generate systematic test. These paths are so short that they does not pass in any module and they are scattered in all the processor, so they can not be classified as long paths. Moreover, each fault can be tested controlling a little number of PPIs. This means that the ATPG process can merge lots of faults under the same pattern, complicating the parsing process. For example, the ATPG tool can merge lots of faults that are in decode stage with others in execution stages generating one single pattern. This situation is difficult to be handled, because it implies to create a parser that can manage different patterns at the same time. Probably this can be solved launching

an ATPG process for each endpoint, parse the patterns generating a sequence of assembly instructions and then try to merge the assembly instruction.

An other difference between long and short paths is the simulation time. The combinational simulation of long path faults takes long time, instead the sequential simulation is relatively fast using the solution proposed in chapter 3. To explain better this concept, a no drop simulation is launched, so, all the functional patterns - that can detected a certain fault - are saved using tetramax, then for each temporal interval between patterns a bit flip is injected on the end point of the path under test to verify if the propagation to the PO happened correctly. If the fault is propagated to a PO by the program then the fault is marked as sequential detected and dropped for the next simulations and injections. Considering long path, each combinational simulation takes lots of time, but the number of simulations and injections are few, so the sequential simulation results relatively fast. In short path simulation the number of patterns that can detect each fault explodes. This situation implies that the tool has to manage a big number of fast simulations and injection. Moreover, there are some faults that can not be propagated to a PO, so they generate thousands of simulations and injections without ever being dropped. It is a big problem because the time of total sequential simulation increases a lot. This is solved by a new parameter that control the max number of combinational detection of each faults. When the tool has detected that faults for the maximum number of times it is automatically dropped. Changing that parameter, the simulation can be speed up, but the coverage obtained using this strategy are an approximation of the complete simulation. A practical example is presented in table 5.7.

Program 1 - fault drop limit	**Simulation time**	**Sequential fault coverage**
No limit	160 hours	50.44%
50	25 hours	50.13%

Table 5.7: Table useful to understand how the simulation time changes respect to the fault drop limit. It reports also the approximation in fault coverage.

This example shows that imposing the maximum number of a combinational detection per single fault to 50, the simulation time decreases by six times, instead the sequential fault coverage remains the same. In conclusion, this parameter is useless to test Long Path delay faults due to their low combinational detection, instead it is very useful in short path delay testing because speed up remarkably the total simulation time.

Chapter 6

Conclusion and future improvements

The goal of this thesis is to create a systematic method for the generation of assembly programs used for the in field test of the path delay model. The focus has shifted only on the long path delay because, as analysed in the literature review, it is clear that the number of faults grows up exponentially with the increase of the number of gates, making it impossible to analyse all the extractable faults. Long path faults are the most important as they often violate the setup time on the flip flop of the processor pipeline creating failures in the entire system. The results obtained are satisfactory, since the method created is able to obtain a coverage very close to 100%, whereas the programs written for others model of faults are inefficient.

It is important to highlight the experiment done on the load store unit because it helps appreciate that a program that obtains good coverage testing faults on the path delay model, will also obtain good results in the transition delay model. If it is possible to detect a long path in a module using an assembly program, then that program seems to be valid also for the transition and stuckat model. Probably this observation cannot be extended to every type of circuit, but there is a possibility that it is valid for all devices with a schematic structure, as adder and multiplier. This study is not part of the scope of this thesis; however, future testing should be made to confirm this possibility. In this case it can be seen that the contrary is true: a program able to reach high coverage using stuck-at and transition model is not able to detect long path delay.

This approach can be defined as systematic because it deeply exploits the ATPG to generate the assembly program. The translation from combinational to functional patterns is the main obstacle of this project but it is also its warhorse. Three methods are proposed to increase the fault coverage, in such a way that it

becomes more and more efficient. Writing a good parser requires a large amount of time because the test designer must have a good knowledge of the functionalities and modules of the device. This information is not always available, however, there are good user guides for *RI5CY* core that facilitate this work. The method is written to be generic and so in theory should be suitable for any sequential circuit. However, the application of this method has been only tested on this processor. Further research should also consider applying this method to other cores to extend its validity.

The systematic method adopts a precise flow but many tactical plans have been produced by hand, looking at the modelsim simulations. The ideal result would be to change it from a systematic to an automated method. The initial idea was to allow an evolutionary program the possibility to generate instructions, therefore launching the fault simulations, checking the results and consequently evolving and increasing its coverage. This approach was immediately discarded because each fault simulation required several minutes, this meant that the algorithm would not converge in an appropriate interval. Moreover there was the problem that a randomly generated program is not very efficient and for this reason the evolutionary program would struggle to work. The patterns generation processes on combinatory modules by means of ATPG are faster. But, by changing perspective, it might be possible to create an evolutionary program able to generate assembly instructions and constraints on the pseudo primary inputs for the ATPG. This should base its evolution on the comparison between the patterns obtained by the logical simulations of the instructions and the patterns coming from the ATPG. When it finds some similarities it can push to make the patterns converge by modifying both constraints and assembly instructions.

Another critical point in this method comes from the assembly programs it generates. These have too many instructions to allow an in-field test, because they fill all the space in the Instruction Memory. Unfortunately, it is close to impossible to think of a solution to reduce the lines of code because the problem is partially intrinsic in the fault model. In fact, long paths require very specific patterns for them to be detected, this implicates that it is hard to find patterns able to detect multiple faults at time. Sometimes only the instruction operands must change along the program, instead the two test instructions are fixed, this situation allows to write cyclic program that are probably able to obtain coverage very similar to that achieved, saving up some lines of code. In other cases the specificity of patterns - concerning the operands, the registers where to save the operands or even the opcode of the instructions - nullifies the alternative to run cycles. In case a software solution cannot be found, it is possible to create a tester with a bigger memory.

It was tried to extend this method by also applying it to the study of the

short path delay, but this requires the implementation of some modifications as previously shown. This is not strictly necessary because - unlike for long paths - the programs written for other faults model are suitable also for the short paths. In fact, satisfactory results have been achieved for short paths by simply simulating programs written either for the transition or stuck-at model (which is equal to use random programs). As already stated, it is possible to make a classification for endpoints, this extends the capability of the method used in this project even for the short path. Moreover, it may be possible to assign the task of writing the assembly code to an evolutionary program, since the combinational simulation for short path is faster.

Beyond of all the problems that were generated from this work with their relative solutions, the goal is considered to be achieved, since a method able to detect faults with a good efficiency in a systematic way has been found.

Bibliography

[1] Kwang-Ting Cheng Angela Krstic. *Delay fault testing for VLSI circuits*. Kluwer Academic Publishers.

[2] 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.

[3] Wikipedia contributors. *Automotive electronics*. Accessed 02/10/2020. URL: https://en.wikipedia.org/wiki/Automotive_electronics.

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

[5] Dario Foti. «New techniques for path delay faults functional test». MA thesis. Politecnico di Torino, 2020.

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

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

[8] 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.

[9] *RI5CY: User Manual*. Accessed 03/03/2020. URL: https://www.pulp-platform.org/docs/ri5cy_user_manual.pdf.

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

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

[12] 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.