UNIVERSITÀ DI PISA

Scuola di Ingegneria
Laurea magistrale in Ingegneria Informatica

Master's thesis

Design and implementation of a routing algorithm to maximize test coverage of permanent faults in FPGAs

Supervisors

Prof. Cinzia BERNARDESCHI

Prof. Andrea DOMENICI

Candidate

Filippo MASCOLO

Academic Year 2014/2015

Acknowledgements

Firstly, I would like to express my sincere gratitude to my supervisors Prof. Cinzia Bernardeschi and Prof. Andrea Domenici for giving me their support and the opportunity to live such a magnificent experience in Germany for writing this thesis.

A very special thanks goes to my tutors Dario Cozzi and Sebastian Korf for helping me continuously in my research, for their patience and motivation.

I must acknowledge Luca Cassano for his assistance during my research study and thesis writing.

Last but not the least, I would like to thank my family because they were always supporting me and encouraging me with best wishes.

Contents

List of Figures

List of Tables

Abstract

Partially reconfigurable systems are more and more employed in many application fields, including aerospace. SRAM-based FPGAs represent an extremely interesting hardware platform for this kind of systems, because they offer flexibility as well as processing power. On the other hand, radiations in the atmosphere make the problem of permanent faults in these devices relevant. The goal of this thesis is the design and implementation of a routing algorithm to maximize test coverage of permanent faults in routing resources of SRAM-based FPGAs, using testing circuits placed on the FPGA. Routing resources represent up to 80% of the whole chip area in modern FPGAs, and the proposed algorithm can cover all physical wires of an arbitrary selected large region of the FPGA.

This work is part of a project aimed at developing a software flow for testing and diagnosing faults due to radiations, during a space mission. Once faults have been detected and diagnosed, patching the discovered faulty resources is possible.

Introduction

Electronic devices are used in several application fields, from the entertainment market to military equipment, from mobile phones to satellites. In particular, SRAM-based FPGAs represent a very interesting hardware platform for this range of systems, because they offer flexibility as well as processing power. A particular kind of applications is the one called mission-critical, where failures may result in significant economical losses, as in the case of satellites, which cannot be repaired or returned for maintenance if some parts stop working. In this case, FPGAs are responsible for handling the major tasks of a satellite mission, for example route computation, control of experiments and communications, and their capability to tolerate faults is a key requirement.

When FPGAs operate in a space environment, both temporary and permanent faults can occur due to radiation. Temporary faults are Single Event Upset (SEUs), i.e., modifications of the content of memory elements in the device, and Single Event Transients (SETs), i.e., undesired transient electrical impulses. Permanent faults induced by radiations on electronic devices are caused by the Total Ionizing Dose (TID), i.e., the accumulation of charge trapped in the oxide layer of transistors in CMOS circuits. The TID causes a degradation of performance and may ultimately cause the complete destruction of parts of the system.

This thesis is motivated by the need to develop testing techniques addressing the requirements of space-based FPGA applications, and in particular, those exploiting FPGA reconfigurability. Reconfigurable FPGAs offer designers the possibility of partitioning the computing resources of a chip into a number of regions, each independently and dynamically reconfigurable. A given region, for example, could be used to control a satellite's movement in the initial phases of a mission, and to control its payload after the satellite has reached a stable orbit. It is therefore important to test a particular region of the FPGA before it is reconfigured for a new task. Several testing techniques have been developed, and in this thesis the on-line and application-independent approach is considered. The on-line testing technique, with the availability of multiple reconfigurable regions, allows tests to be made at run time without influencing the rest of the FPGA. The application-independent approach makes tests more general without considering the specific application that will be used.

This work presents an on-line on-demand approach to test faults in the routing resources of FPGAs. The proposed approach relies on a set of testing circuits, composed of a Test Pattern Generator (TPG) and an Output Response Analyzer (ORA), to test the physical wires and Programmable Interconnection Points (PIPs) between the TPG and the ORA. Moreover, the approach exploits an ad-hoc designed place-and-route algorithm, named U-TURN, to maximize the coverage of permanent faults for these circuits. The

approach uses partial dynamic reconfiguration to place the testing circuits at run time on the free areas of the FPGA to test them before the functional modules are placed, when reconfigurations are required. Experimental results have shown that it is possible to generate, place and route the testing circuits needed to detect the 100% of the physical wires and up to the 86% of the PIPs in a reasonable time.

This work is part of the OLTRE project (On-Line Testing of permanent Radiation Effects) that aims at supporting on-line on-demand testing, diagnosing and fault masking for dynamically reconfigurable systems on SRAM-based FPGAs. This project is made in cooperation between University of Pisa, Politecnico di Torino, University of Bielefeld and it is funded by ESA (European Space Agency).

Chapter 2 describes the starting point of this thesis showing testing circuits previously developed for our purpose. Chapter 3 provides an overview of the architectural structure of FPGAs, then it gives some information about their programming and finally a study of the faults that can occur. Chapter 4 provides the background concepts for the development of the routing algorithm, giving details on its design. Then, Chapter 5 presents the implementation of the algorithm and the complete testing flow. The results achieved with this algorithm, running on different families of FPGAs, are in Chapter 6. As a conclusion, Chapter 7 will analyse the results

of the presented approach, considering the possible future work related to this thesis.

2 Related work

Two kinds of testing methods can be performed to check if an FPGA is fault free: *application-independent* and *application-dependent*.

Application-independent methods, such as [1] [2] [3] [4], are meant to detect every structural defect causes by the manufacturing process of the whole FPGA. On the other hand, application-dependent methods, such as [5] [6] [7], focus detecting errors only in resources actually used by the design. With both approaches, tests can be *off-line* or *on-line*. Off-line tests are usually made by the manufacturer. Instead On-line test are made at run time on unused areas of the FPGA while the remaining parts continue their normal operations.

Permanent faults causes by TID have not yet been extensively addressed by testing techniques. In the last years, the shrinking of the feature size in the CMOS technology made SEUs the predominant radiation effect in electronic devices. Therefore, researches focused much more on the detection of SEU effects than on TID.

Among various methodologies the most common are the *external* and *Built-In-Self-Test* (BIST) approaches. The former approach consists in considering the CUT (Circuit Under Test) as a black box, providing input stimuli from outside the device. Then responses are collected to check whether a fault occurred. On the other hand, the BIST approach allows the

device to test its own resources without acting from outside. With either approach, the following items are needed:

1. a mechanism to provide a set of input stimuli;
2. the circuit under test (CUT);
3. a mechanism to analyse responses in order to discriminate whether the CUT is fault free or not.

Usually testing circuits are composed of a *Test Pattern Generator* (TPG) that provides input stimuli and of an *Output Response Analyzer* (ORA) that observes the output of the resources under test and determines whether they are faulty or not. These techniques may be divided into two sub-categories: *Comparison-based* [4], [8] and *Parity-based* [9] [10].

In the Comparison-based approach, the ORA knows the expected output associated with the input stimuli generated by the TPG and by comparing it with the actual output of the resources under test, it is able to determine whether a fault occurred. With this approach it is not possible to detect faults in the TPG and those faults that do not interfere with the actual output of the resources under test. With the Parity-based techniques these limitations have been overcome. The TPG calculates the parity bit on its output and the ORA calculates the parity bit on the received signals. The ORA is able to detect whether a fault occurred by comparing these two parity bits. The ORA does not need to know the expected output, because it relies on the parity bit produced by the TPG.

Parity-based testing approaches may be additionally classified in two categories: single parity [9], and cross-coupled parity [10]. In the single parity-based technique, the TPG is a n-bit counter and produces n+1 output bits where the last one is the parity bit calculated on the other n bits. The ORA, as soon as it receives the n+1 bits, calculates the parity bit on the first n bits and compares it with the received parity bit. In this technique some faults in the TPG cannot be detected and it is necessary that the parity bit is sent on a fault free wire. In cross-coupled parity-based techniques the TPG is composed of two independent n-bit counters, let us call them TPG_i and TPG_j; each TPGs produces n output bits plus one parity bit. Similarly, the ORA is duplicated: ORA_i receives the n input bits from TPG_i and the parity bit from TPG_j; conversely, ORA_j receives the n input bits from TPG_j and the parity from TPG_i. In this way all the faults occurring into the TPGs may also be detected.

3 Background

3.1 Field Programmable Gate Array

Field Programmable Gate Arrays (FPGAs) are pre-fabricated, electrically programmable, silicon devices, composed of programmable logic blocks, a programmable routing structure and programmable Input/Output pads. Since the birth of the integrated circuit technology in 1960s, many attempts have been done to achieve programmable devices in order to give to hardware architects the possibility of exploiting hardware performance and software flexibility at the same time.

The first modern FPGA was introduced by Xilinx in 1984 under the name of XC2064; since then the FPGA technology has dramatically grown in terms of scale of integration, performance and competitiveness against other technologies. The ability of being programmed, and most of the times reprogrammed, provides many advantages over other hardware technologies.

Application Specific Integrated Circuits (ASICs) offer better performance in terms of computational time, area requirements and power consumption at the cost of much longer time to market and economic effort: a full-custom ASIC design needs many months of engineering work and hundreds of millions of dollars to be completed, since state-of-the-art tools for synthesis, placement & routing, extraction, simulation, timing and power

analysis, great engineering effort and very expensive foundry masks are needed. An FPGA design needs between only a few dollars and a few thousand dollars for units purchase, much less engineering effort and much shorter time to be designed and configured, and often an FPGA device can be reconfigured if a mistake was made during the design cycle. Figure 3.1 shows how FPGAs per unit cost is invariable while ASICs per unit cost decreases by increasing the number of produced units.

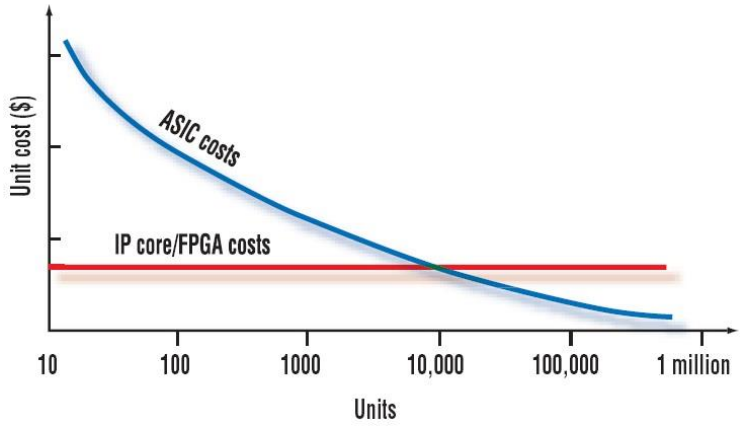


Figure 3.1: Unit Cost (in dollars) / unit diagram

Given this, only large scale productions can afford a full custom ASIC design, while small and medium scale productions prefer saving money and time by the use of FPGAs devices. For this kind of productions FPGAs represent nowadays the best tradeoff between performance on one hand and cost and time to market on the other hand.

Nowadays FPGAs have become the dominant programmable logic technology no longer being used merely as glue logic or as prototyping

devices, and starting being used to implement sub-systems or complete systems (System-on-Chip).

3.1.1 FPGA architecture

An FPGA is a prefabricated array of configurable logic blocks, interconnected by a programmable routing architecture and surrounded by programmable input/output blocks.

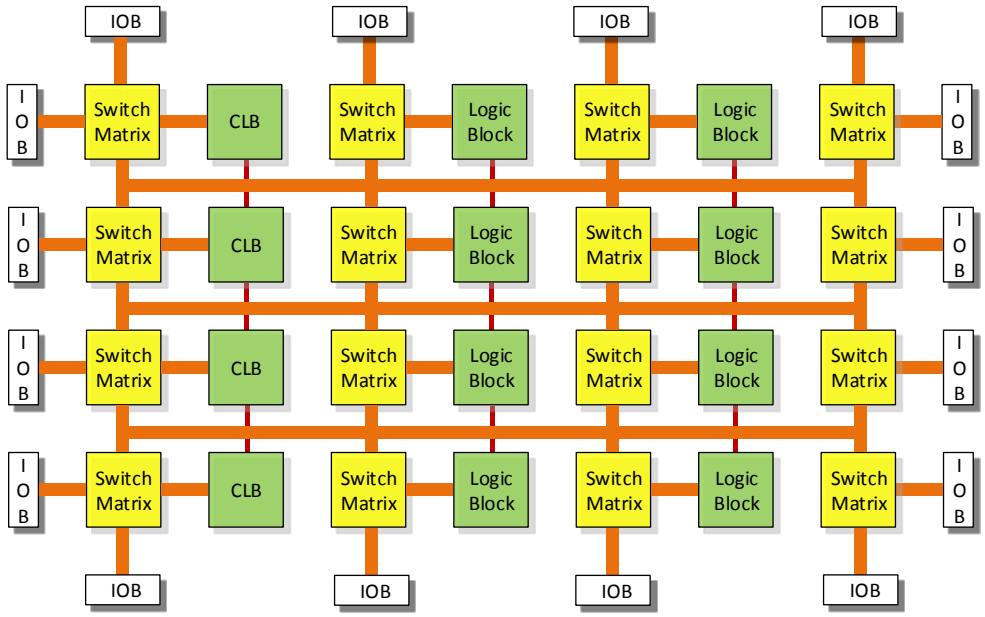


Figure 3.2: FPGA typical architecture

Figure 3.2 shows the basic architecture of an FPGA chip, that is composed of three types of basic blocks:

- **CLB**: Configurable Logic Blocks are the logic resources, which may be simple combinatorial logic (*Soft Logic Blocks*) or memories, multiplexers, ALUs and other kinds of prefabricated circuitry (*Hard*

Logic Blocks). The structure of a CLB is hierarchically divided into Slices;

- **IOB**: Input/Output blocks provide external connections for the FPGA. Since these blocks are placed on the border of the FPGA they are used to get the signals into and out of the FPGA;
- **Switch Matrix**: all the FPGA blocks are connected to each other with a programmable routing architecture. Thanks to the switch matrix is possible to route a signal inside the FPGA. The routing is made by activating the *Programmable Interconnection Points* (PIPs) which are placed inside the switch matrix. All switch matrices are connected by a complex structure of fixed connections that are presented in detail below.

Figure 3.3 was created using FPGA Editor [11], a graphical application provided by Xilinx for displaying and configuring FPGAs. This figure is useful to explain some terminology that is extensively used in the rest of this thesis. For simplification purpose in this figure only two switch matrices and several connections are represented.

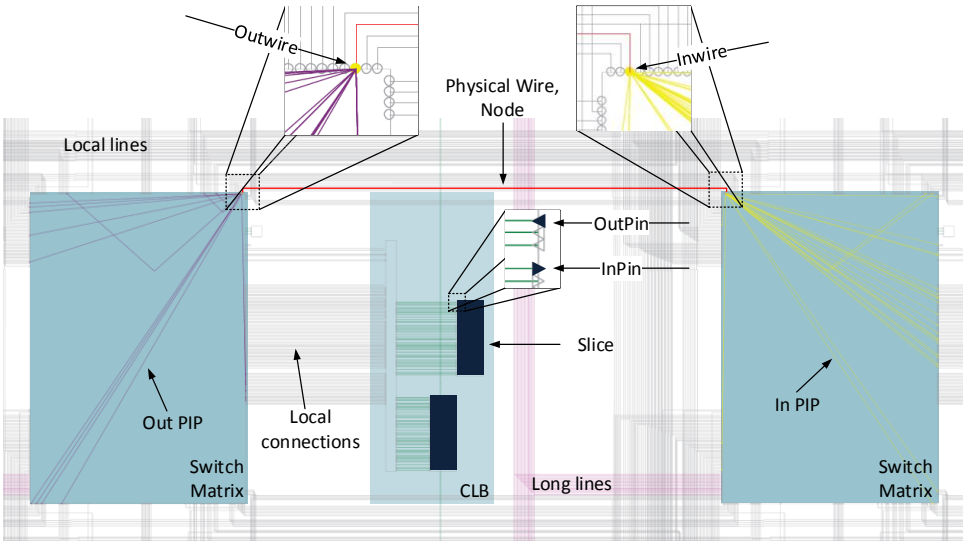


Figure 3.3: FPGA Editor screenshot of a Xilinx FPGA

- **Slice**: includes the configurable resources for implementing boolean functions, flip-flops and carry-propagation logic;
- **Pin**: connection point of one Slice and one physical wire. If the direction of the signal goes to the Slice, the Pin is called *InPin*. Otherwise, the Pin is called *OutPin*;
- **Physical wire**: hardwire interconnections of switch matrices;
- **Wire**: connection point of physical wire and switch matrix. If the direction of the signal goes to the switch matrix, the wire is called *InWire*. Otherwise, it is called *OutWire*;
- **Programmable Interconnection Point (PIP)**: the programmable routing infrastructure consists of a set of wire segments, which can be interconnected by means of programmable elements: PIPs. A PIP is a switching element, whose state is determined by the value

contained in a configuration cell of a transistor as shown in Figure 3.4.

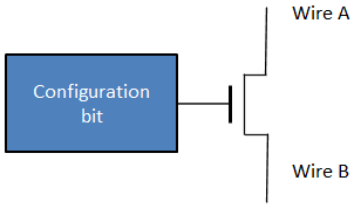
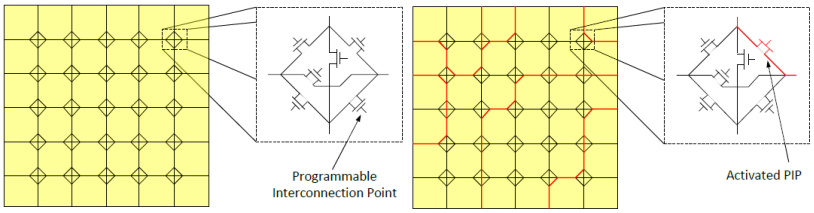


Figure 3.4: A configuration bit determines the state of a PIP

Figure 3.5a shows the structure of a switch matrix where every connection between the lines is determined by 6 transistors. Activating them it is possible to drive a signal through the switch matrix. Figure 3.5b shows a possible configuration of the programmable routing architecture.



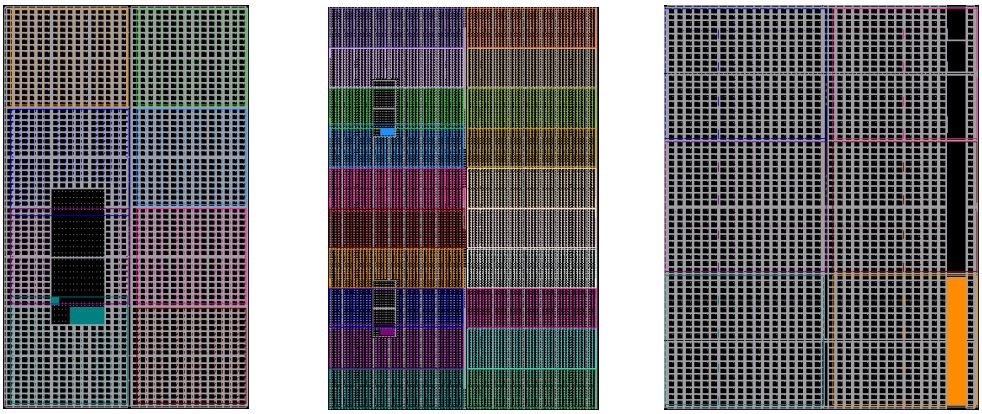
(a) switch matrix overview **(b) example of activated PIPs**

Figure 3.5: Programmable Interconnection Points inside a switch matrix

Depending on the direction of a PIP, it is classified as *InPIP* or *OutPIP*. If the signal goes through the PIP and out from the switch matrix, than the PIP is called OutPIP (in FPGA Editor is highlighted in purple). Otherwise, the PIP is called InPIP (highlighted in yellow);

- **Global connections**: wires that connect all switch matrices among them. *Local lines* and *Long lines*, shown in the figure above, belong to that category and their names come from the length of the connection;
- **Local connections**: wires that connect all Switch Matrices with the related CLB.

Modern FPGAs are split into multiple clock regions whose number and shape vary among different families of FPGA. Each clock region contains a certain numbers of components like switch matrices and CLB and an associated clocking. Clock regions are fundamental parts of FPGAs that allow for zero skew clock distribution inside the region itself. Figure 3.6a, 3.6b, 3.6c highlight clock regions of different FPGAs.



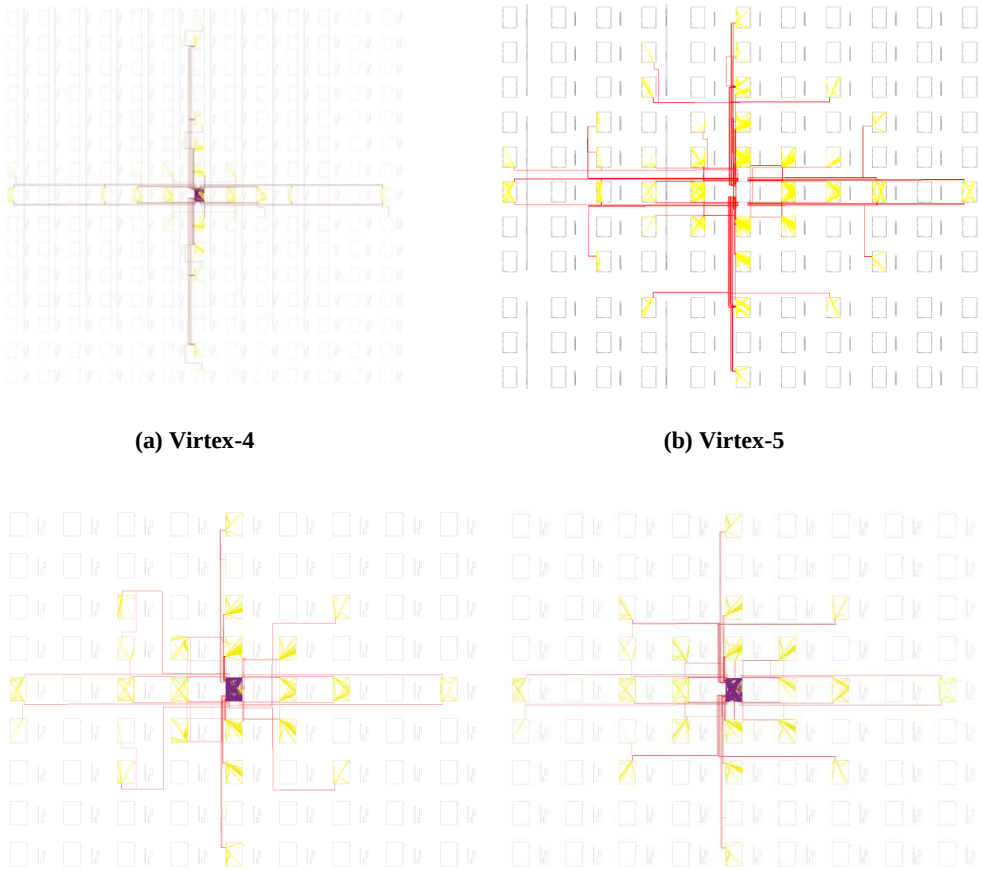
(a) Virtex-4 FX12 **(b) Virtex-4 FX100** **(c) Virtex-5 LX20**

Figure 3.6: Clock regions of different FPGA families

3.1.2 Global connections analysis

The programmable routing architecture provides an array of routing switches between each internal components. Each programmable element is located inside a switch matrix, allowing multiple connections to the general routing matrix.

The structure of the routing architecture (composed by global connections) is considerably different among Xilinx FPGA families. The number of inWires for every physical wire and the distance of the switch matrices connected characterize different families of FPGA. In Figure 3.7a, 3.7b, 3.7c, 3.7d the lines connected to outWires of single switch matrix are depicted for each Xilinx FPGA family, therefore the PIPs of the inWires are selected. According to classification of the lines given in section 3.1.1, the outPIP (colored in purple) are only within the starting switch matrix, therefore they are related only to the outWires, because no outWires are selected in the other switch matrix. Hence, all wires connected to the considered lines, are of the inWire type; so the PIPs connected to them are only inPIPs.



(a) Virtex-4 **(b) Virtex-5**

(c) Virtex-6 **(d) Spartan-6**

Figure 3.7: Connection structure in Virtex-4-5-6 and Spartan-6

3.1.3 Designing an FPGA-based system

Every FPGA relies on an underlying programming technology that is used to specify the functionality that each block implements, to configure interconnections between blocks and to interconnect I/O pads with blocks.

FPGA programming consists in defining the hardware structure of the device by producing a programming code, called *bitstream*. After being downloaded in the device, the bitstream enables or disables gates in the

logic blocks to implement a certain function and enables or disables connections between wires to connect or disconnect two logic blocks or a logic block and an I/O pad.

The end user can define the behavior of an FPGA using a hardware description language (HDL) or a schematic design. Then it is possible to use a design automation tool, typically provided by vendors, to generate a technology-mapped *netlist*. A netlist is a textual description of a circuit diagram, which provides a map of how its elements are interconnected. Then a process called *Place-and-Route* can be performed in order to adapt a netlist to the actual FPGA architecture. In the Xilinx terminology *design implementation* is a process composed by translation, mapping, routing and generating a bitstream file for a given design. All these tools are integrated in the Xilinx ISE Design Suite, a tool provided by Xilinx. The following picture shows the entire flow.

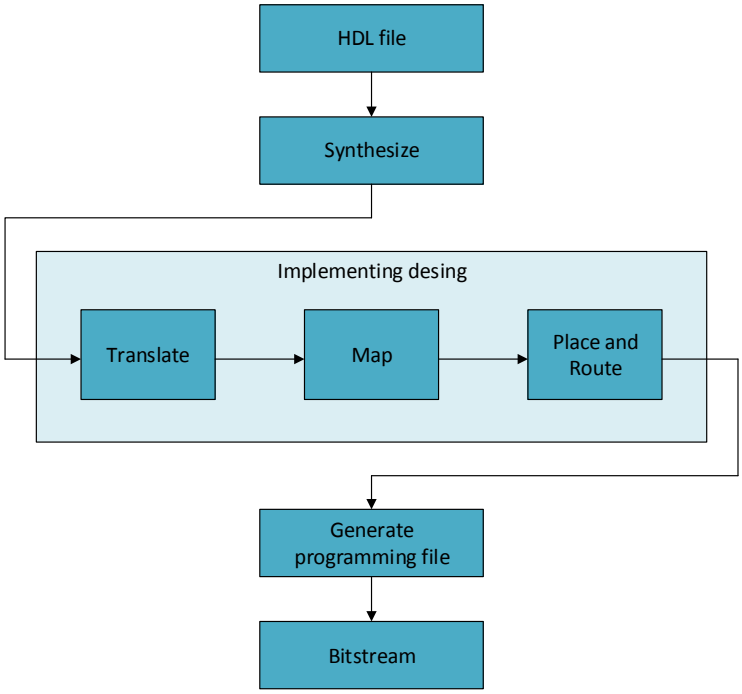


Figure 3.8: Xilinx design implementation

- The **HDL** file represents the input;
- **Synthetize** generates a supported netlist type for the Xilinx implementation tools;
- **Translate** converts the input design netlist (EDIF or NGC) in a NGD netlist;
- **Map** maps the design into CLB and IOBs;
- **Generate programming file** generates the bitstream which is loaded into the device.

For more information about the Xilinx design automation tool see [12].

3.1.4 Dynamic Partial Reconfiguration

Since 2000 FPGAs were designed with a high flexibility feature: the *Dynamic Partial Reconfiguration* (DPR) [13] [14]. It gives the designer the ability to reconfigure a certain part of the FPGA during run-time without influencing other parts. Thanks to this technique the FPGA can be reconfigured on the fly, without switching off or resetting the whole system. Moreover it is possible to reconfigure only a specific part of the FPGA, while the rest of system remains unchanged.

A reconfigurable system typically includes an area for static system components (*base region*) and one or more partially reconfigurable regions (*PR regions*) for dynamic system components. The dynamic system components are represented by the partial reconfiguration modules (PR modules). The placement of a PR module is done by configuring a predefined area in a PR.

Hence a partial reconfigurable system design require a partitioning of the FPGA in order to reconfigure only specific areas. In particular two different kinds of region are created: static and dynamic region. The static region contains components which are not reconfigured in the system, therefore the configuration of base region is made once in the initialization of the system and cannot be changed at run-time. The reconfigurable region is used for run-time reconfiguration, hence it is possible to place and remove PR module based on the system needs.

3.2 Faults in FPGAs

This work is a part of a major project [15] [16] that aims to study the fault tolerance of FPGAs when they are used in space flight missions. In that environment FPGAs are exposed to radiation that could cause malfunctions. This chapter gives a general idea of the radiation faults effects that may occur in space environment, focusing on permanent ones.

A *fault* is defined as a malfunction of an internal component of the system. If activated by the operation of the system, it can be propagated to the outputs of this component, becoming an error. Finally, if the error is propagated and produces a malfunction of the system outputs a failure occurred.

Faults can be introduced in the system both by the user and the surrounding environment. The user can cause faults providing wrong inputs to the system, bringing it to an incoherent state. On the other side, the environment could cause several kind of faults, depending on the nature of the solicitation it provides to the system. In space and avionic applications the most critical environmental factor that could lead to failures is radiation. Radiation may cause both short- and long-term damages in electronic systems. Short-term damages are the well-studied *Single Event Upset* (SEUs) and *Single Event Transient* (SETs) [17]. Long-term damages are caused by the *Total Ionizing Dose* (TID), i.e. the accumulation of charge trapped in the oxide layer of transistors in CMOS circuits [18]. TID first

causes degradation of the performance of the system, and ultimately it may cause failures.

The following sections summarize the radiation effects explained in [19].

3.2.1 Single Event Effects

Single Event Effects (SEEs) are models of the effect of the funneling induced by a single particle in a certain location within the device. Depending on the hit characteristics and time, the electric fields and energy of the incident particle, the funneling can produce different functional behaviors. SEEs can be temporary faults, Singe Event Transients (SETs), that affect the device for a certain period of time, at most until a power cycle is performed, and these are called soft errors; otherwise, if the produced fault is permanent, damaging the device itself, it is called hard error.

A Single Event Upset (SEU) is the effect of a particle that changes the value of a memory element, as a latch or a cell within a memory array. When a SET is generated by an ionizing particle within a memory element, it could force the feedback loop to change its value thus modifying the actual value stored in the memory element.

SEUs are not usually permanent faults, because at the first writing operation of the affected memory element the wrong value will be overwritten. However, there are some cases, in which the memory element

could not be written again, thus changing the SEU effect to a permanent fault, until a reset or power cycle is performed.

3.2.2 Total Ionizing Dose

Differently form the Single Event Effects, Total Ionizing Dose (TID) is the effect of the accumulation of the charge injected by radiation. TID models the effects of a charge accumulation and displacement damages that, together, lead to different malfunctions. First, a global worsening of the device performance is registered; transistors slow down and the power consumption increases. In memory circuits, ionizing dose affects the sensitivity of the logic states of memory cells asymmetrically, causing an imbalance. In Flash memories, it has been proven that TID leads to a change in the threshold of the floating gate transistors so that they lose their re-programmability functionality. A second effect of TID is the change in the SEE sensitiveness. One consequence of this is that SEUs can cause the so-called "*stuck bits*", that are memory cells whose value is modified by a SEU but because of the ionizing dose, their correct value cannot be restored.

In general, TID effects can be annealed by means of heating the device, in order to provide enough energy to the crystalline lattice so that atomic locations can be restored and trapped charges can be released. To summarize, TID provokes three kinds of effects: performance degradation, power consumption increase and programmability loss.

3.2.3 Fault effects on design

In modern FPGAs, the routing resources represent up to 90% of the whole chip area. When a permanent fault occurs on the device, the routing resources can be affected in different ways. Four categories of possible faults are presented: Stuck-at-0, Stuck-at-1, Stuck-off, Stuck-on that are explained in details in the chapter 4.1.

In order to classify and localize the effects on the routing resources, caused by a permanent fault, it is necessary to recognize the modification introduced in the application. Most part of the configuration memory bits is related to the switch matrices, which control the routing resources. Each net of a circuit is realized by connections of logic modules through Programmable Interconnection Points. A SEU or permanent fault in the configuration bit, which controls a PIP, can alter or interrupt the propagation of one or more signals. The schematic representation of the effect scenario can be described considering the original interconnection condition, illustrated in Figure 3.9, that provides the implementation of two different routing nets *net 2C* and *net 7E* using respectively the two PIPs *I2→OC* and *I7→OE*. Considering the configuration illustrated in Figure 3.9 it is possible to identify all the possible effects induced by a modification of a PIP resource.

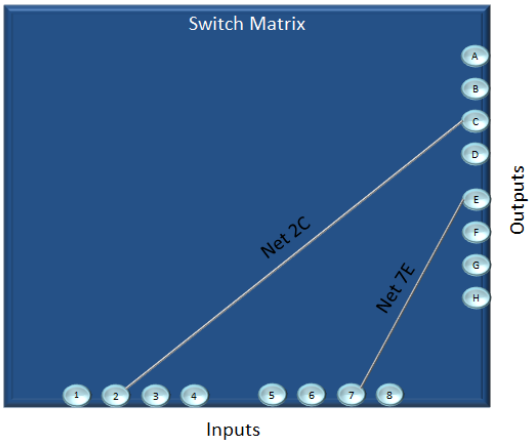
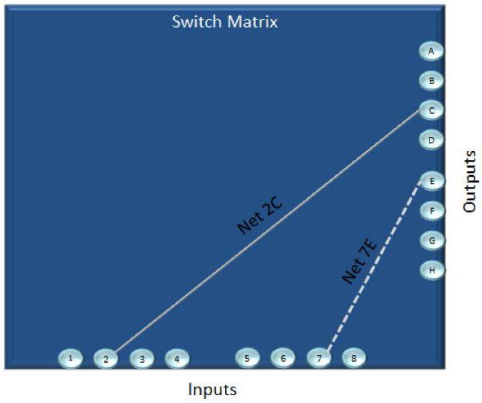
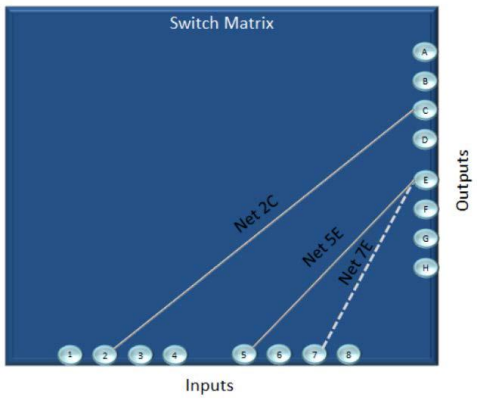


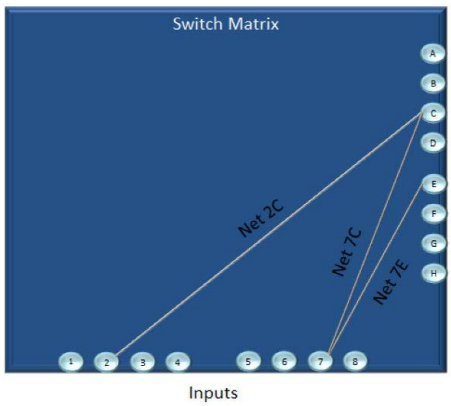
Figure 3.9: Routing condition without errors



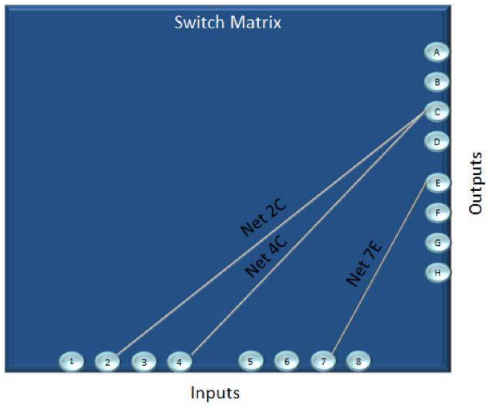
(a) the Open effect, first case



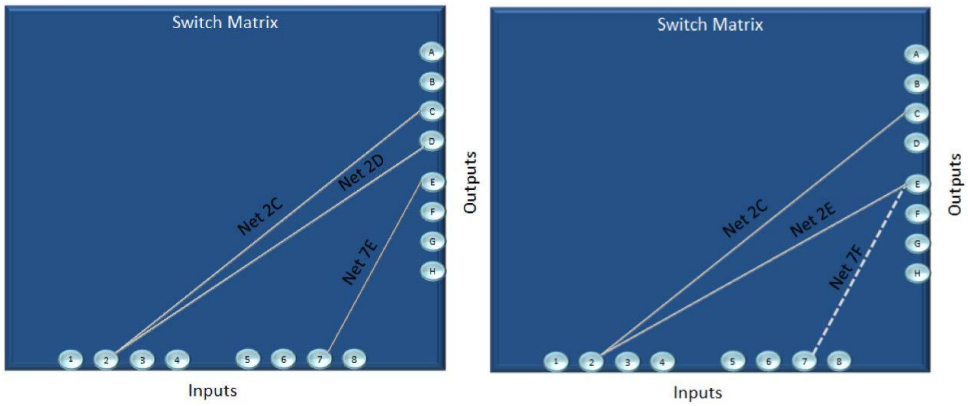
(b) the Open effect, second case



(c) the Conflict effect



(d) the Input Antenna effect



(e) the Output Antenna effect **(f) the Bridge effect**

Figure 3.10: Permanent fault effect cases

- **Open**: The PIP corresponding to the *net 7E* it is not programmed any more. Therefore, *I7* and *OE* are not connected. There are two cases classifiable as Open. The first case is illustrated in Figure 3.10a where the *net 7E* is deleted. The second case is illustrated in Figure 3.10b, the *net 7E* is deleted and a new net, for example *net 5E*, connects an unused input node *5* to the previously used output node *E*, is created. In the second case, the signal *net 7E* has an undefined logic value;
- **Conflict**: A new PIP, corresponding to the *net 7C*, is added between an input node *7* and an output node *C*, both previously used, as illustrated in Figure 3.10c. The new PIP creates a conflict on the output node *C*. The propagated signal is not identifiable by means of only a topological analysis;

- **Input Antenna:** A new PIP, corresponding to the *net 4C* is added between an unused input node *4* and a used output node *C*, as illustrated in Figure 3.10d. The new PIP can influence the behavior of the output node depending on the output logic value assumed by the nodes of the CLB;
- **Output Antenna**: A new PIP, corresponding to the *net 2D*, is added between a used input node *2* and an unused output node *D* (as illustrated in Figure 3.10e). The new PIP does not influence the behavior of the implemented circuits;
- **Bridge:** The PIP corresponding to the *net 7E* is disabled while a new PIP, corresponding to the *net 2E*, is instantiated between a used input node and the output nodes of the previously used *net 7E* as illustrated in Figure 3.10f. The behavior of the implemented circuit is modified.

If a fault modifies the routing of the FPGA, without affecting the behavior of the system, this effect can be categorized in:

- **Tolerant**: The configuration of the programming PIP is not affected by modifications. The modification of the bits in the configuration memory does not create any modification of the topological instances of the nets;
- **Unrouted**: The modification of the PIP cannot be classified in anyone of the considered classes.

For further details on permanent faults that can affect an FPGA see [19].

Table 3.1 shows how a specific permanent fault can turn into one or more effects presented in this paragraph (see chapter 4.1 for the definitions of fault types). This thesis is considering these effects in order to detect a possible permanent fault within a routing resource.

Type of Permanent Fault	Permanent Fault Effect
Stuck-at-1 (wire)	Open
Stuck-at-0 (wire)	Open
Stuck-off (PIP)	Open
Stuck-on (PIP)	Conflict, Antenna, Bridge

Table 3.1: Relation between the Permanent Fault and its effect

3.3 Testing circuit for on line testing

Figure 3.11 shows a high level view of testing circuits previously developed [20] to make tests with the BIST approach.

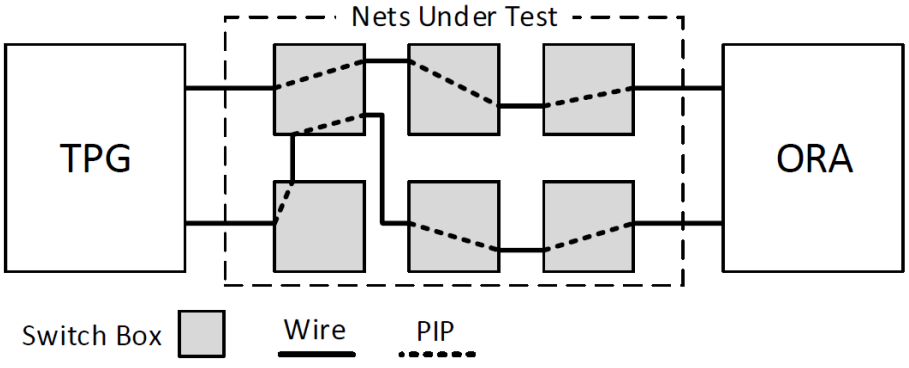


Figure 3.11: High level view of testing circuit

The most important components are the TPG and the ORA:

- *Test Pattern Generator* (TPG) is used for a generation of particular input stimuli for the resources that have to be tested;
- *Output Response Analyser* (ORA) is the component that reads the outputs from the resources under test and establishes if a fault occurred.

All the connections between these components are called *Nets Under Test* (NUT) and represent the routing resources that will be tested. A NUT must be routed starting from an OutPin of a CLB and ending in an InPin of another CLB crossing a certain number of physical wires and PIPs. It is worth noting that the dimension of the TPG and the ORA is very small and for this reason it tests can be made also in small areas. This circuit is already been validate and it is able to detect the 100% of the faults in the resources under test.

The structure of the designed testing circuit is depicted in Figure 3.12.

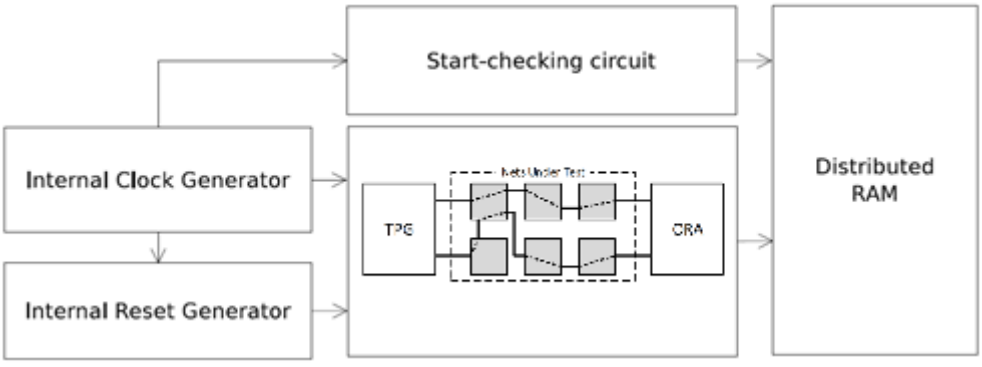
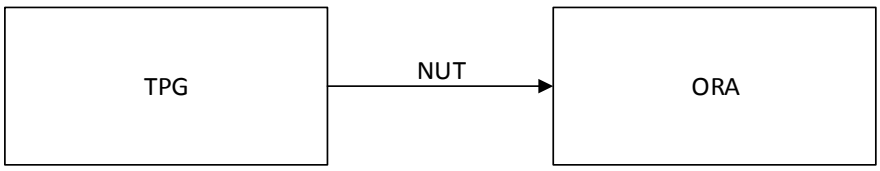


Figure 3.12: The structure of a testing circuit

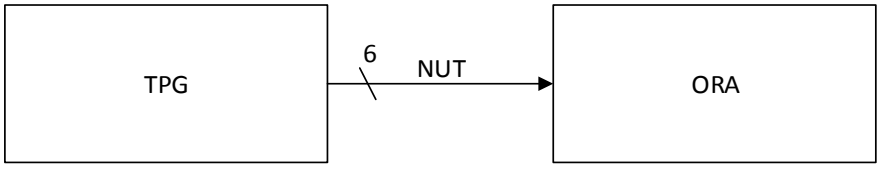
Clock Generator and *Reset Generator* are inside the circuit, therefore the clock and reset signals are generated by dedicated modules in order to make the testing structure entirely independent of the region of the FPGA on which it is placed. Indeed, no external clock and reset signals are used, so it is possible to change the area under test by only replacing the testing circuit, without any changes in the logic. Results are stored in the configuration memory of the FPGA and it can be used the read-back techniques [21] to get them. The start-checking circuit is able to verify whether the testing circuit has been correctly configured and the test correctly started. In fact some faults may prevent the test to start at all.

For further details on internal logic and behavior of this circuit see [20].

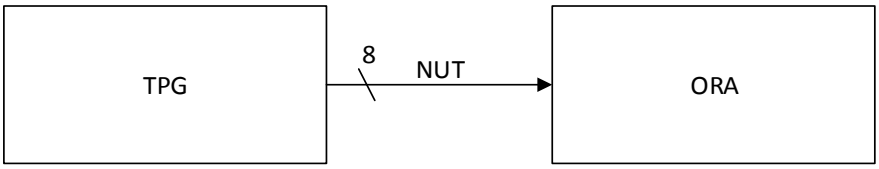
There are three different versions of the testing circuit to make on-line tests, which differ substantially on the number of nets that can be tested at the same time. Figure 3.13a, 3.13b, 3.13c depict a high level view of them.



(a) NUT1



(b) NUT6



(c) NUT8

Figure 3.13: Different testing circuits

Since these circuits can test multiple NUTs we can make both a fine- and a coarse-grained test. This circuit can detect the stuck-at-0/1, stuck-on and the stuck-off permanent faults.

4 Routing algorithm design

On-line testing of reconfigurable FPGAs exploits dynamic partial reconfiguration to place testing circuits on the unused regions of an SRAM-based FPGA device, called target regions, before placing on it the functional modules of a reconfigurable system. With this approach it is possible to check if the target region of the FPGA is free of faults at run-time. The testing approach used in this work is application-independent and cross-coupled parity-based.

The testing circuit is composed by the TPG and the ORA (see section 3.3). A stimulus is sent over the NUT through the OutPin of a CLB that hosts the TPG and received through the InPin of another CLB that hosts the ORA (see section 3.1.1). All the wires and all PIPs connecting these components represent the NUT. Therefore the first step involves deciding where to place the TPG and the ORA on the target region, while the second requires choosing the exact wires to connect TPG and ORA. This corresponds to the so-called *Place-and-Route* process (for more information see [22]).

The TPG, the ORA and the supporting circuity are available in an XDL file and pre-placed. The algorithm developed in this thesis (*U-TURN*) chooses a set of NUTs that maximizes test coverage of physical wires. The algorithm is written in the C++ programming language and takes as input the following files:

- the XDL file representing the testing circuit;
- an image file representing a specific FPGA;
- a file specifying the partitioning of the FPGA in different regions;
- the name of the region we want to test.

Then U-TURN starts the routing of the NUTs to cover all physical wires in the region under test. During the execution two files are printed out: the *physical wires testability report* and the *PIPs testability report*. When U-TURN terminates several testing circuits are created using the computed NUTs and a summary of tested resources is printed out. The following figure shows the inputs and the outputs of the algorithm.

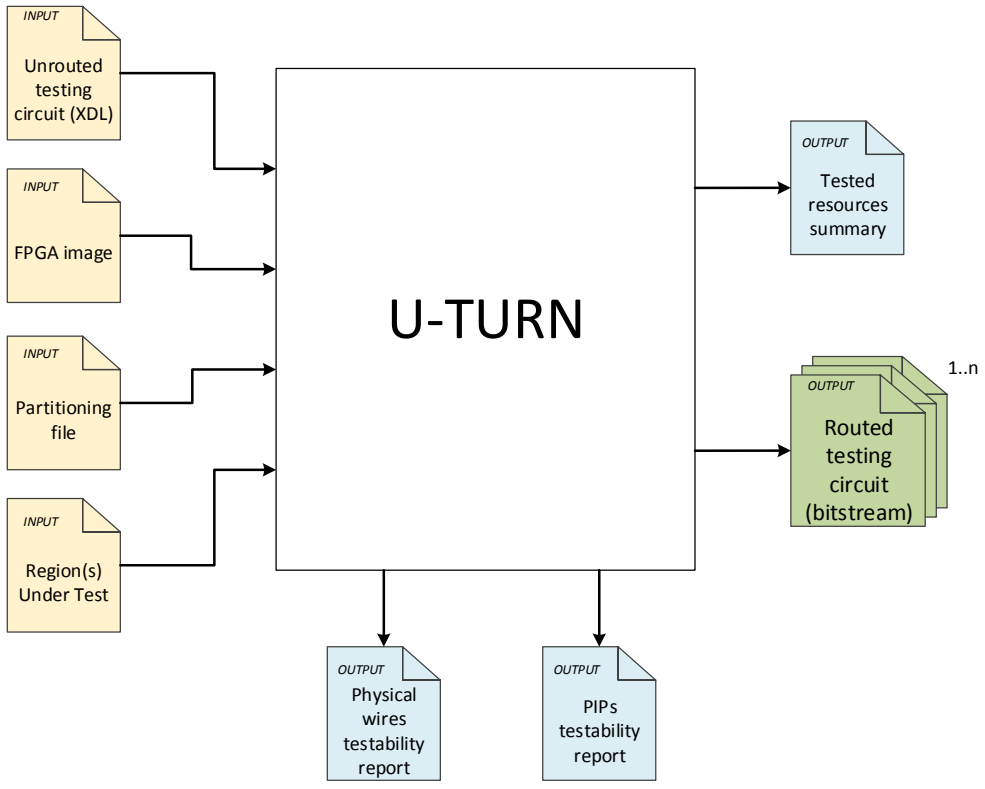


Figure 4.1: U-TURN inputs / outputs

There are also other components in the TPG and in the ORA that are connected to each other for the proper functioning of the testing circuit itself [20]. It is worth noting that these resources do not belong to the NUT so they are not tested by the testing circuit. These resources must be free of faults for a correct behavior of the testing circuit, therefore the TPG and the ORA should be placed as close as possible to reduce the probability of faults presence. A test of them should be performed in a second step, changing the position of the entire testing circuit and using them in the related NUT.

It should be noted that the longer is the NUT, the larger is the amount of resources tested at a time. But it was found that it is not possible to have an arbitrary length of the NUT. A limitation on the numbers of the PIPs that could be used in a NUT is given by the FPGA architecture itself; in fact a signal could change its logic value if it is driven over a long path. A limit of 100 PIPs has been set to ensure that testing circuits can still evaluate the presence of faults.

A representation of the FPGA is useful for developing an algorithm capable of routing the NUT. This structure has to provide a good description of the FPGA resources, but at the same time it must be suitable for our purpose. Figure 4.2 shows several switch matrices and some connections between them. It should be noted that there is a chance of loops in the FPGA architecture (e.g. pw2, pip4, pw9, pip9, pw10, pip10, pip11, pw5, pip3) and this characteristic should be visible also in the representation.

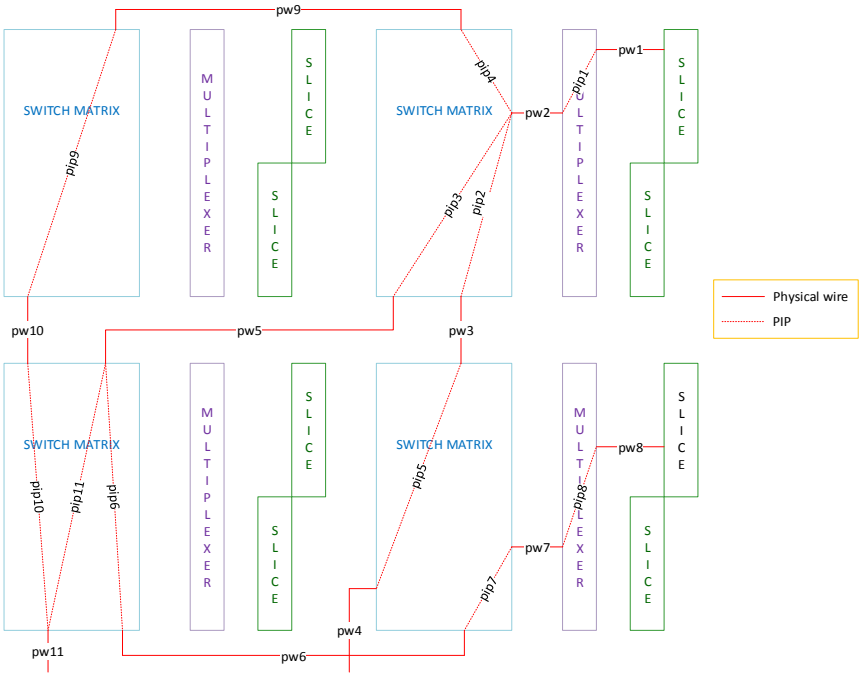


Figure 4.2: Switch matrix connections for graph creation

For these reasons we adopt from graph theory the concept of directed cyclic graph to describe routing resources of a FPGA region. The following figure depicts this representation where Nodes denote physical wires and Edges represent PIPs.

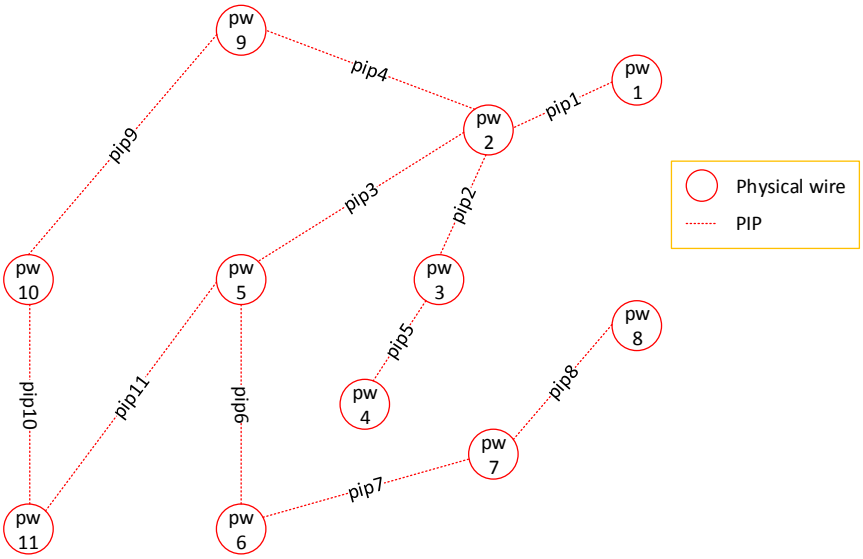


Figure 4.3: Graph representation of the FPGA

Obviously the graph dimension depends on the region size; as an example the figure below shows the representation of an entire clock region of a Virtex-4 FX12.

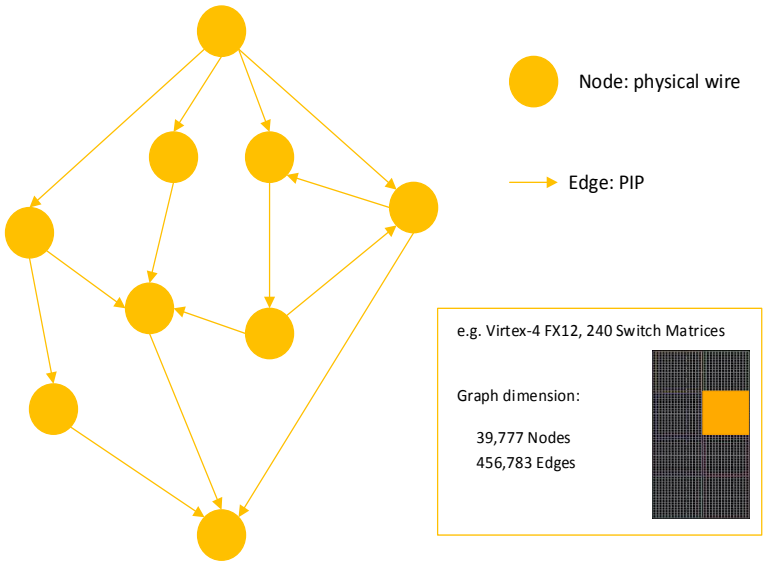


Figure 4.4: Graph representing a clock region of a VIrte-4 FX12

Several kind of permanent faults can affect a FPGA (see section 3.2) so the first step consists in the understanding which faults are detectable with our testing circuits and how to model them to run tests.

4.1 Permanent fault model

This section describes the permanent faults that we are considering, their effects and how we model them to perform a real test. A net represents the connection between two or more components and it is composed of physical wires and PIPs.

4.1.1 Physical wire stuck-at-0/1

Figure 4.5 is an FPGA Editor screenshot where the red line represents a physical wire that connects several switch matrices.

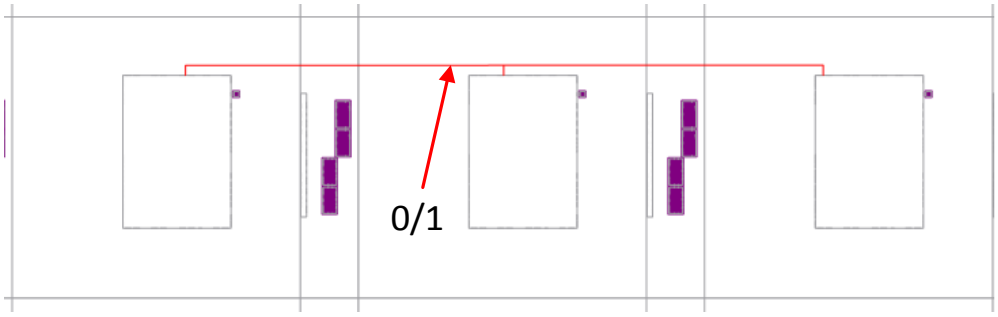


Figure 4.5: Stuck-at-0/1 on a physical wire

A Stuck-at-0 fault on a physical wire forces the logic value of the net to ‘0’. It is possible to check if a physical wire is stuck-at-0-free by using it in a NUT while the TPG is sending a ‘1’ over it. If the ORA receives a ‘0’, it

means that a fault occurred. Therefore if test is passed we are able to verify that all physical wires that belong to the NUT are stuck-at-0-free. A similar approach can be applied in the case of Stuck-at-1.

To verify if all physical wires are stuck-at-0/1-free using a graph representation it is necessary to resolve the *nodes-covering problem*: we need to discover several paths such that all nodes are crossed at least once. If we are using the NUT1 testing circuit the graph has only 1 OutPin and 1 InPin and the NUT has to be routed among these nodes. The following figure highlights 3 different paths to cover all nodes of the graph.

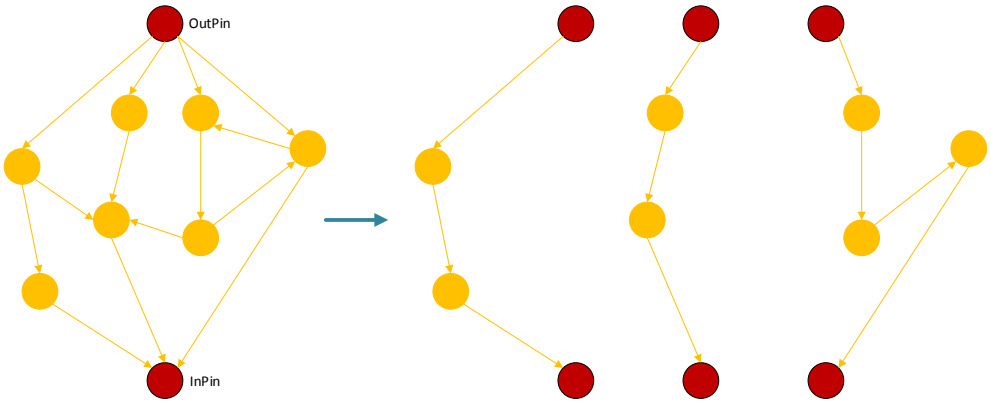


Figure 4.6: Testing stuck-at-0/1 in graph representation

4.1.2 PIP stuck-off

Figure 4.7 is an FPGA Editor screenshot where the red dotted line represents a PIP affected by a stuck-off fault.

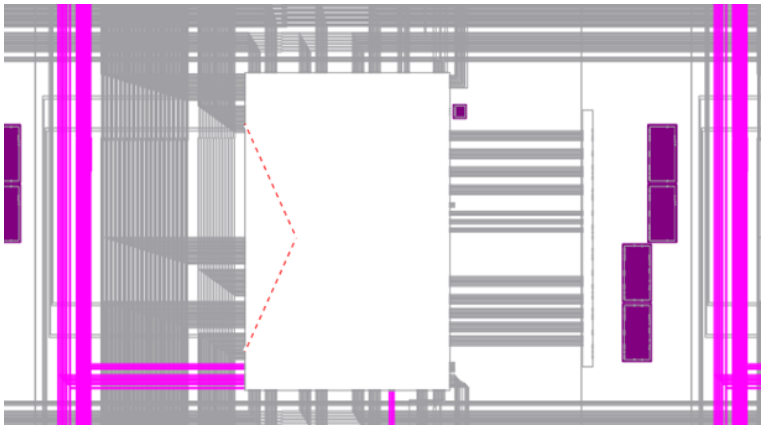


Figure 4.7: Stuck-off for a PIP

A PIP affected by this kind of fault is always deactivated and the two physical wires are unconnected. Therefore if a path uses a stuck-off PIP its logic value will be unknown. It is possible to check if a PIP is stuck-off free by using it in a NUT while the TPG is sending a stimulus over it. If the ORA receives a different logic value a fault occurred. Therefore if the test is passed we are able to verify that all PIPs that belong to the NUT are stuck-off-free.

To verify if all PIPs are stuck-off free using a graph representation it is necessary to resolve the *edges-covering problem*: we need to add some NUTs to the previous such that all edges are crossed at least once. If we are using the NUT1 testing circuit the following figure shows paths to be added to cover all edges where larger arrows represent PIPs not tested yet.

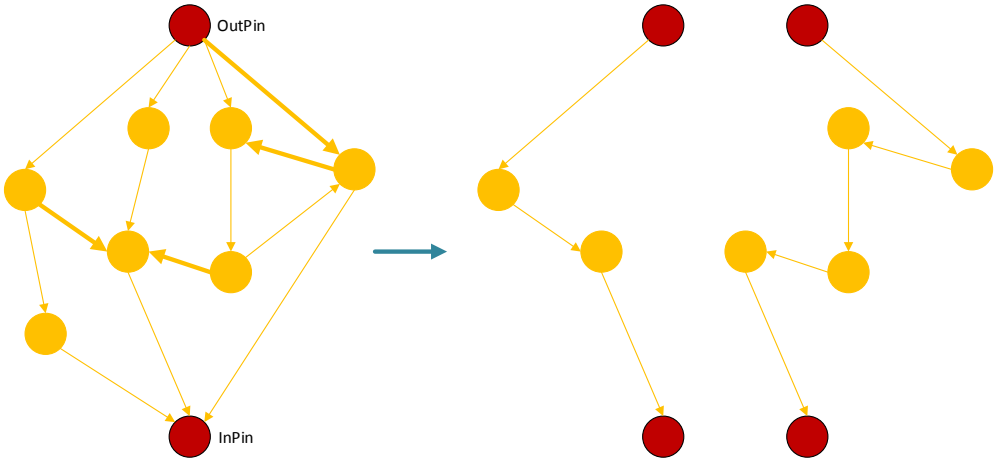


Figure 4.8: Testing stuck-off in graph representation

4.1.3 PIP stuck-on

Figure 4.9 is a FPGA Editor screenshot where a red line represents a PIP affected by a stuck-on fault.

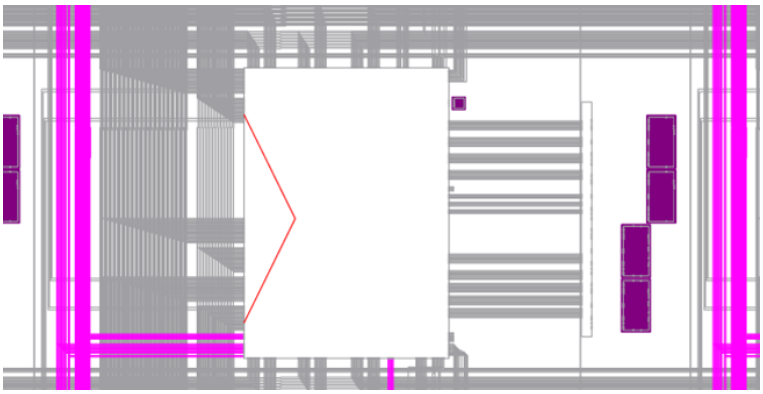


Figure 4.9: Stuck-on for a PIP

A PIP affected by this kind of fault is always activated and creates a connection between two physical wires. This PIP can create an antenna or short two nets of the design. The result of the short can be either a wired-

AND short or a wired-OR short. It is possible to check if a PIP is stuck-on free using two NUTs that can be shorted by it. If both TPGs send stimuli over their own NUT and the related ORA receives a different logic value, then a fault occurred. Therefore if the test is passed, all PIPs that can create a short between the tested NUTs are stuck-on-free.

To test this kind of fault we need two NUTs because we would test PIPs that can connect more nets. For example Figure 4.10 shows the NUT6 testing circuit that has 6 OutPins and 6 InPins and can test 6 nets at the same time. It is worth noting that these nets must be independent thus no resources can be shared between them.

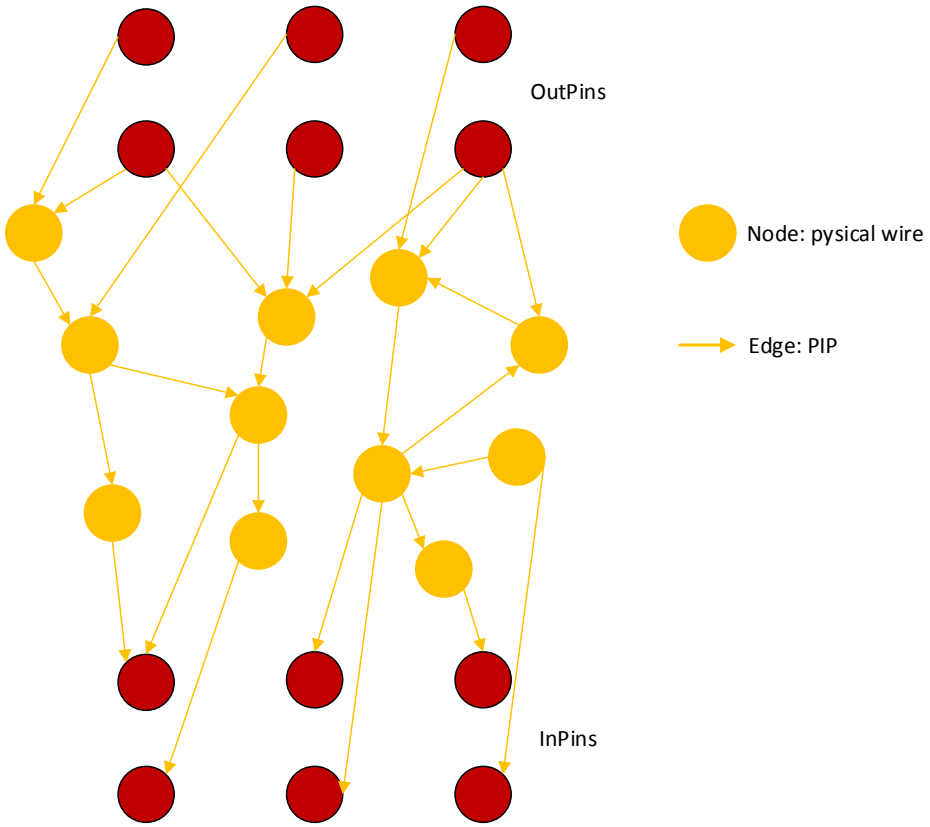


Figure 4.10: Graph representation of NUT6 testing circuit

As an example Figure 4.11 shows highlighted in green the PIPs that can create a short fault between 2 different routed NUTs. If this test is passed we are able to assert that these PIPs are stuck-on free.

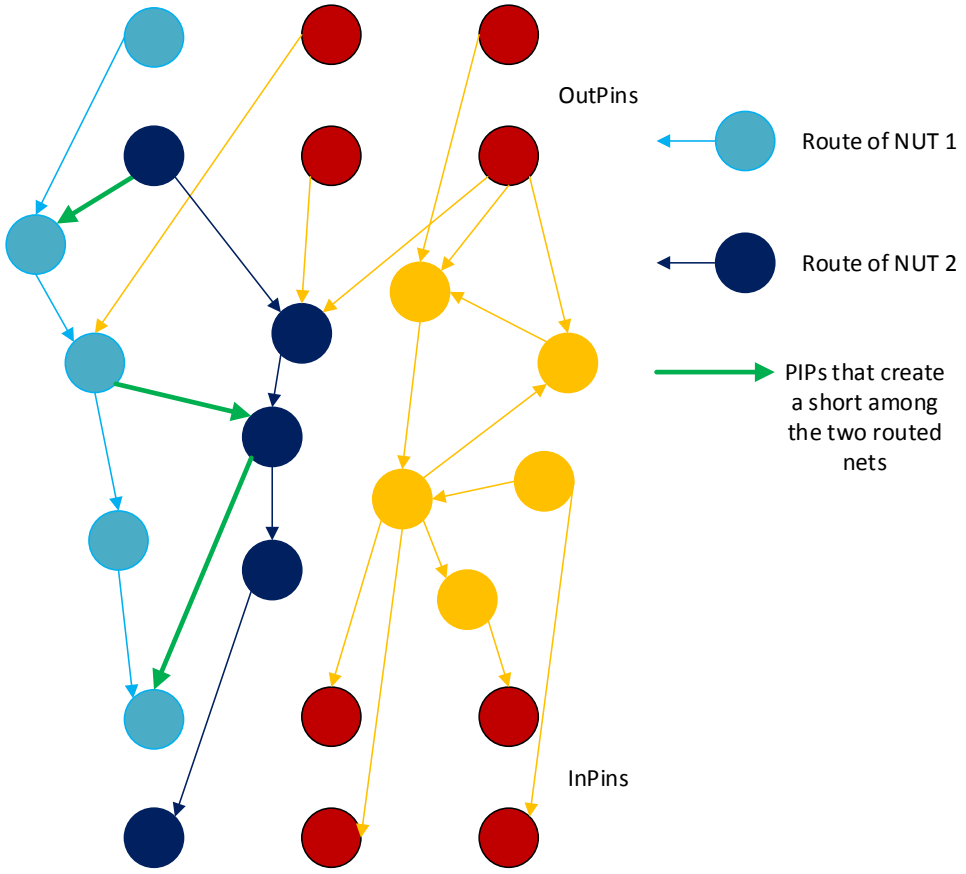


Figure 4.11: Testing stuck-on in graph representation

4.2 Routing resources analysis

As previously discussed, the aim of this work is to find a flow capable of testing on demand routing resources of a specific region of the FPGA. To check if a resource is free of fault, it is necessary to send a signal on it and verify if the received signal is the same as the one sent.

Since we are using the partial reconfiguration technique, a partitioning of the FPGA in different regions is needed. Thanks to this technique it is possible to dynamically modify logic blocks of a region by downloading partial bit files, while the remaining logic in other regions continues to operate without interruption. Due to the partitioning, the routing resources that are on the edges of a region can connect components belonging to different regions. If a stimulus is sent over these resources, a conflict can occur in the component belonging to the other region. We have to be careful when using these resources, and in most cases their employment should be avoided.

We can assume that a **Resources Categorization** phase is needed to understand what is really testable. That phase takes as input the FPGA partitioning and the list of partitions to be tested as shown in Figure 4.12. During this phase, resources are marked depending on their testability.

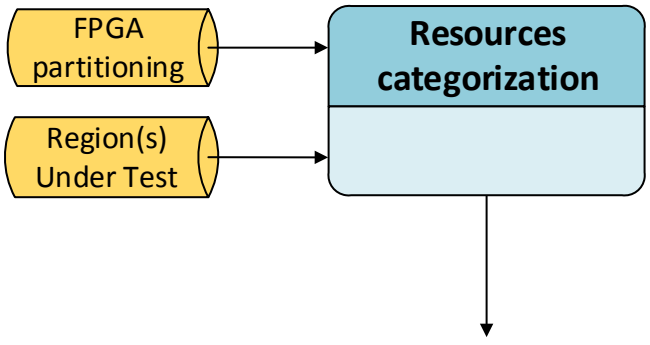


Figure 4.12: Resources categorization flow

Resource marking is made by following a procedure explained in the following. Figure 4.13 shows a simplified version of an FPGA partitioned in two different areas: one static and one reconfigurable that has to be tested.

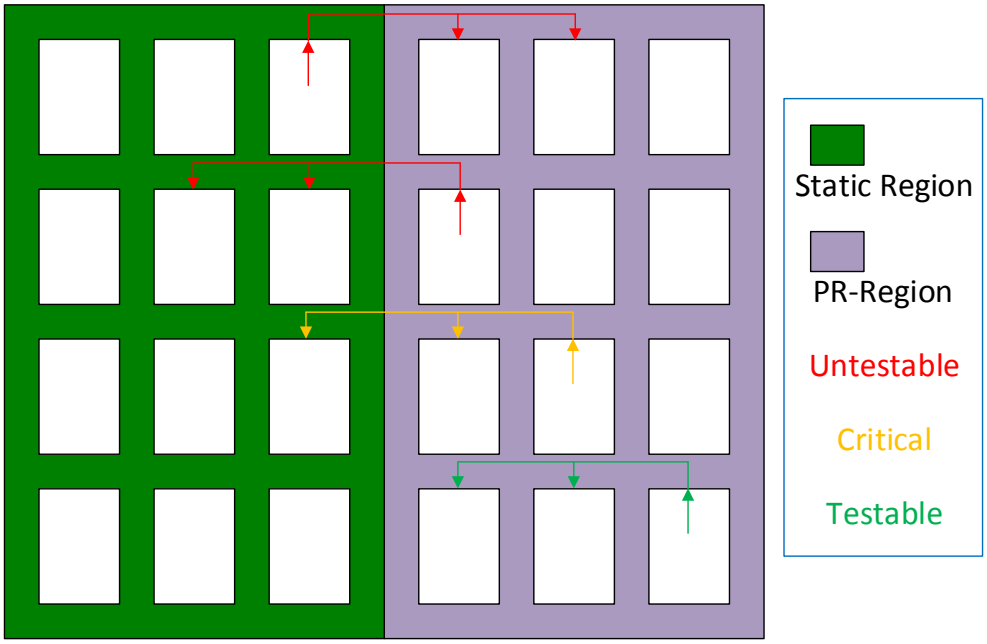


Figure 4.13: Analysis of testability

Physical wires are initialized according to the InWire and the OutWire position. A physical wire is marked as:

- **Untestable**
    - if the OutWire stays in another region because it would be impossible drive a signal on the physical wire;

- if all InWires stay in another region because it would be impossible read-back the sent value;

- **Critical**

  if the OutWire and the InWires are in the region under test but there is also an InWire in another region; if this resource is used an error can be injected in the other region;

- **Testable**

  if and only if the OutWire and all InWires are in the region to be tested.

It is important to notice that this categorization depends on the FPGA partitioning, but there is also another kind of testability according to the capability of the testing circuit. In fact our testing circuit is not able to test all resources (for example the DSP and BRAM resources). Therefore it is appropriate to introduce a new kind of category: the **unsupported**.

Once all physical wires are marked, an additional phase is performed. In that phase all physical wires inherit the testability of connected resources applying the following priority rules:

1. Unsupported;
2. Untestable;
3. Critical;
4. Testable.

But it is worth noting that only testable physical wires can become unsupported.

When the algorithm is running, all this information is stored in the data structures of the program. It is possible to generate script files readable by FPGA Editor to visualize resources in different color according to their testability. Figure 4.14 is an FPGA Editor screenshot that represent a switch matrix of the Virtex-4 FX12 where physical wires and PIPs are highlighted.

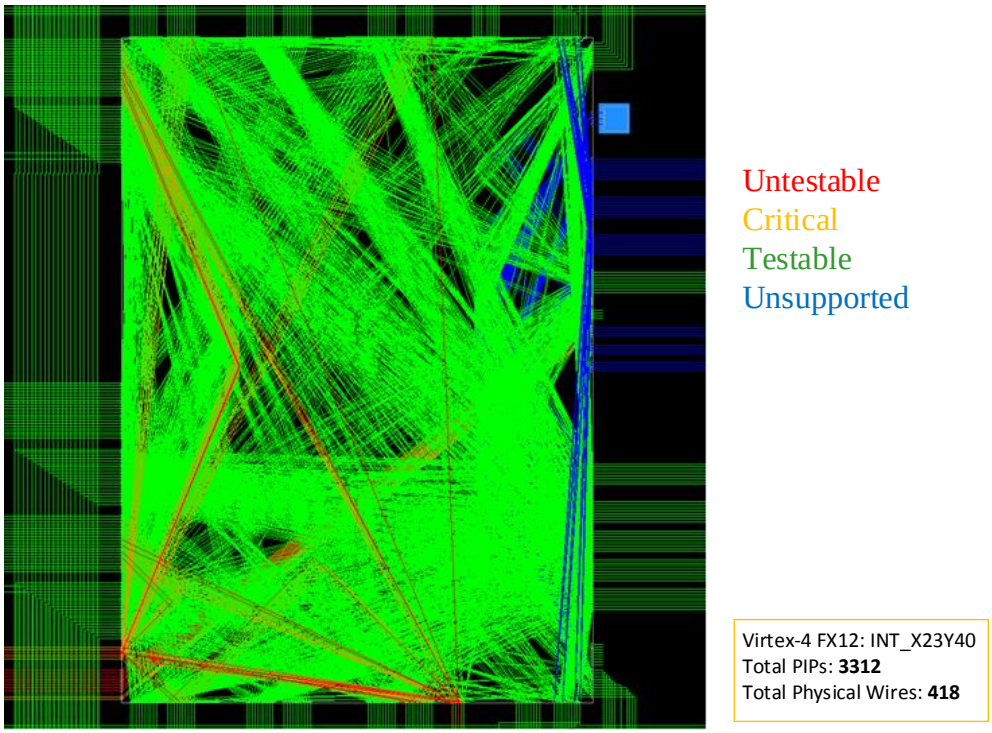


Figure 4.14: Untestable,Critical,Testable,Unsupported resources of a switch matrix on Virtex-4

Two different reports are printed out at the end of this phase:

- Physical wires testability report;
- PIPs testability report.

A subset of these reports is shown in Figure 4.15 and Figure 4.16. The first one lists physical wires testability of all tiles in the region and also the conflicted areas. By "*Tile*" we mean a general component of an FPGA, like a switch matrix or CBL, and in square brackets their own coordinates inside the FPGA are shown. The first column lists all physical wires belonging to that tile, the second describes their testability and the third defines the conflicted regions. If the conflicted region has a name it is shown, otherwise the coordinate of the inWire or the Outwire that causes the non-testability is printed. The second report gives more details adding PIPs information.

Tile: Switch Matrix @ [49,26]

Wire: BEST_LOGIC_OUTS0	Testable	
Wire: BYP_INT_B5	Testable	
Wire: BYP_INT_B7	Testable	
Wire: E2BEG6	Testable	
…		
Wire: LH0	Critical	[26,5],[26,11]
Wire: LV0	Critical	Base1
…		
Wire: LH12	Untestable	[26,11]
Wire: LV12	Untestable	Base0,Base1
…		

Figure 4.15: Physical wires report

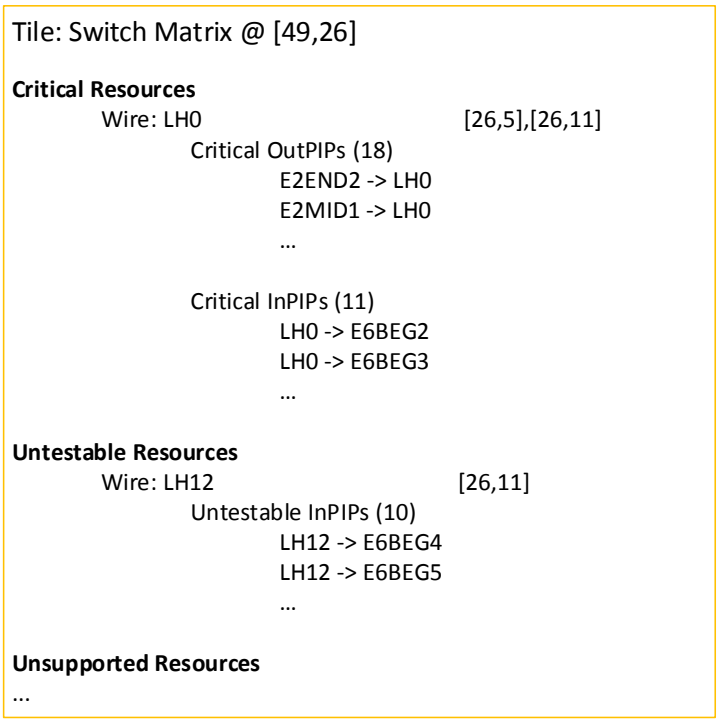


Figure 4.16: PIPs report

Since the reports are very long and hard to read two more heat-maps are created to better understand how many testable resources are in the region under test. The numbers in Figure 4.17 represent the percentage of testable PIPs in a clock region of a Virtex-4 FX12, that is used as region under test. This represents what is reachable according to the partitioning of the FPGA and because of it, this heat-map is **Circuit Independent**. It should be noted that larger values are in the middle of the region, while numbers decrease as you go away from the centre. This happens because the tiles near edges have more physical wires that reach different regions. There are 87% of testable physical wires and 78% of testable PIPs in a clock region of a Virtex-4 FX12, without considering the testing circuit.

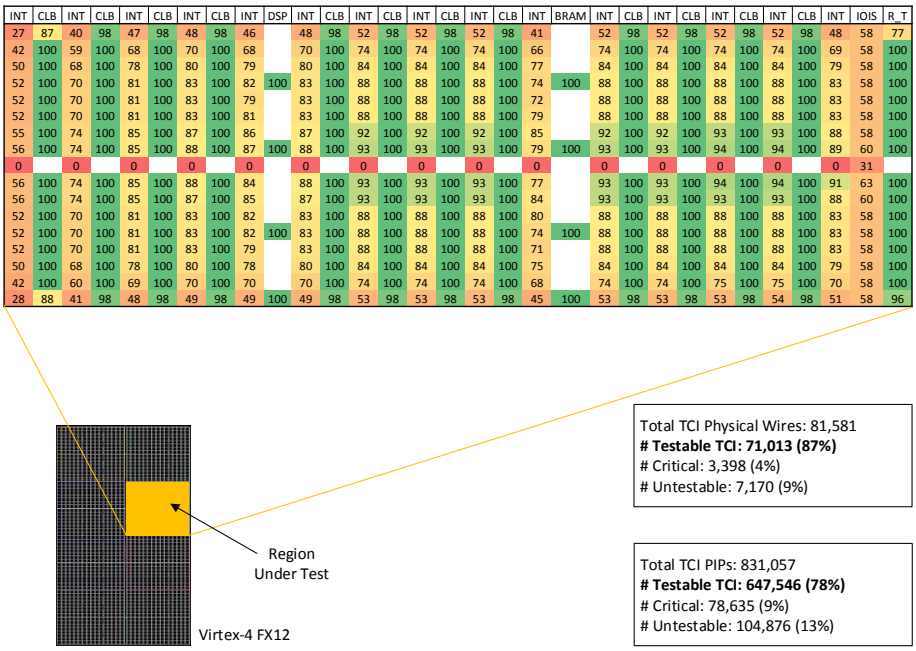


Figure 4.17: Test Circuit Independent heat-map

Numbers differ when the testing circuit is considered because it could not test all resources such as DSP, BRAM, IOIS. Figure 4.18 shows a **Circuit Dependent** heat-map where some testable resources become unsupported. In particular 44% of testable physical wires and 29% of testable PIPs are now unsupported.

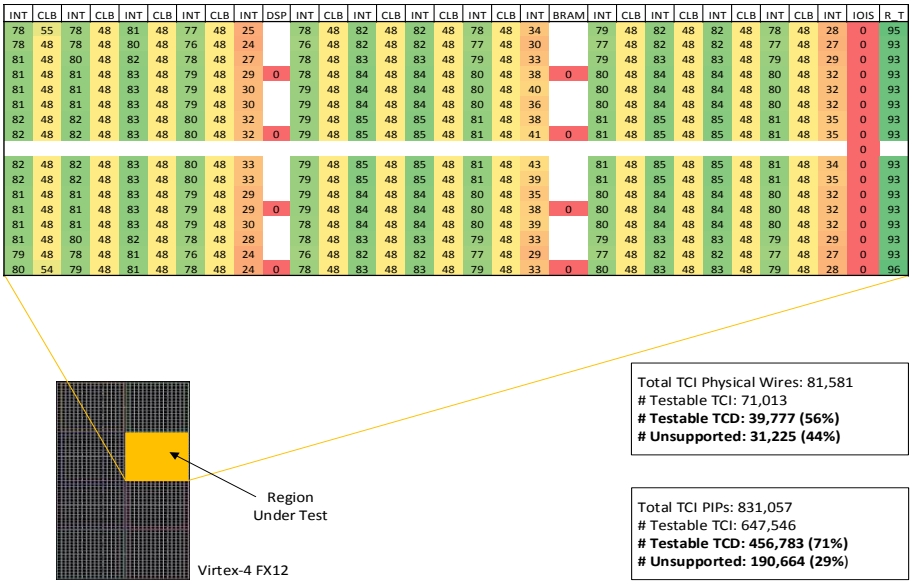


Figure 4.18: Test Circuit Dependent heat-map

5 Proposed routing algorithm

Once the routing resources categorization is made the algorithm capable to route the NUT (or NUTs) over testable resources of the FPGA region can be applied.

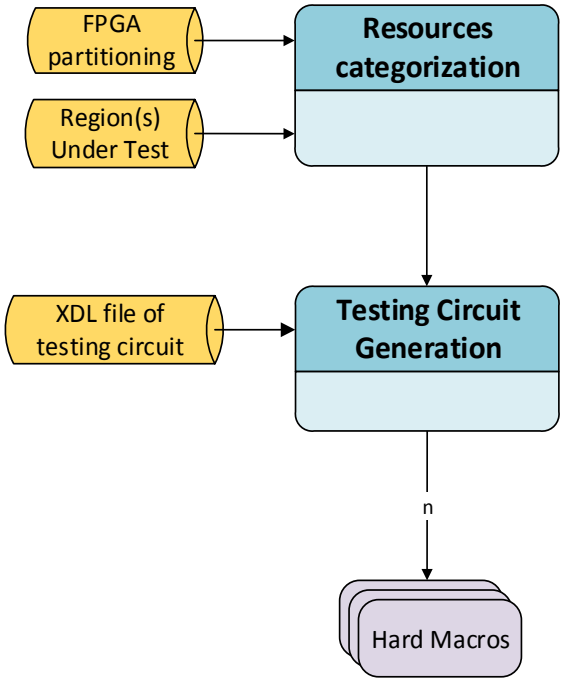


Figure 5.1: Complete project flow

The complete flow of this work is illustrated in Figure 5.1 where the **Testing Circuit Generation** phase operates on the data structure filled by the previous phase and takes as input an XDL file [23]. That file represents the testing circuit (NUT1 or NUT6 or NUT8) where the positions of the TPG and the ORA are already defined and it misses only the routing of the NUT (or NUTs). A NUT is complete in any of the following situations:

- when there are no more physical wires that can be added to the NUT;

or

- the NUT reaches 100 routing resources (due to the limitation on 100 PIPs that could be used).

Since there are much larger routing resources in a region under test, when the routing algorithm computation is finished, several testing circuits are created and they will be used to check if that region is free of faults.

In the early stages of development I studied the FPGA architectures of different families (see section 3.1.2) to find out if they have something in common that can be exploited. It can be observed that connection types are quite different among families but switch matrices have the same connections between each other in the same FPGA. Since there are a large numbers of switch matrices in a region (e.g. a Virtex-4 FX12 contains 240 switch matrices in a clock region, see Figure 4.4), it is useful to apply the *divide et impera* paradigm to reduce the complexity of the problem; the idea is to target one of them at a time maximizing its number of tested resources.

Since routing resources mainly consist of PIPs (e.g. a switch matrix in a Virtex-4 FX12 contains 3312 PIPs and 418 physical wires, that allow connections to other components as shown in Figure 4.14) the first approach was to maximize the coverage of PIPs. Later we understood that a better

approach to be followed, particularly regarding the computation time of the algorithm, was to maximize the use of physical wires instead of PIPs.

Keeping in mind the limitation on PIP numbers that can be used in a NUT and considering the high number of resources, it is easy to understand that we need more than one NUT to test all resources of a single switch matrix. As a result the number of testing circuits needed to test all resources of a switch matrix change according to the testing circuit type (NUT1, NUT6, or NUT8).

5.1 U-Turn implementation

The main goal of the algorithm is to maximize the number of physical wires used for a Net Under Test. The number of wires in a NUT is incremented by leaving and returning to one *Switch Matrix Under Test* (SMUT); for this reason the algorithm is named **U-Turn**. It runs recursively on each switch matrix inside the *Region Under Test* (RUT) to maximize the use of physical wires connected to the SMUT that have not been tested yet. For a better understanding of the routing algorithm's behavior Figure 5.2 and Figure 5.3 show a simplified vision of an FPGA, showing only the switch matrices. The complete test flow performs the following steps:

1. Define region under test (RUT);
2. Select switch matrix that we want to test (SMUT);
3. Place TPG and ORA;
4. Route the NUT from TPG to SMUT;

5. Add physical wires to the NUT by leaving and returning in the SMUT (done by U-Turn algorithm);
6. Route the NUT from SMUT to ORA

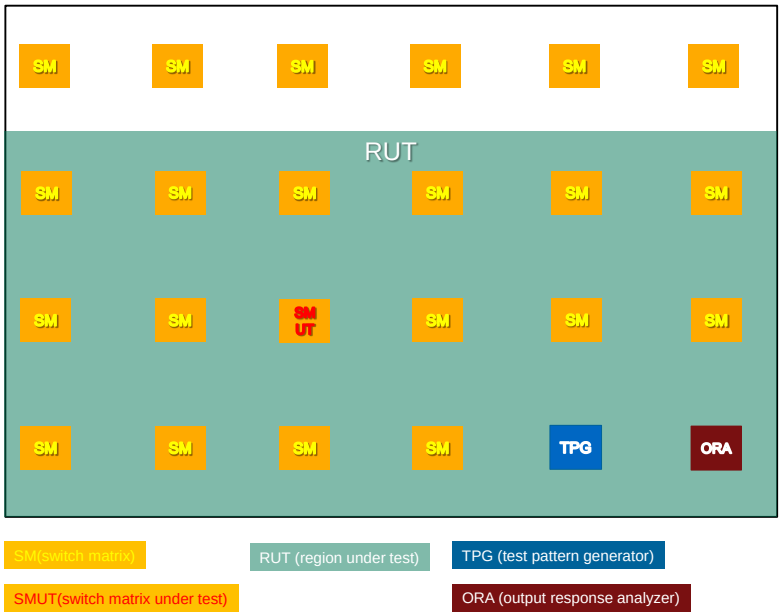


Figure 5.2: Simplify vision of the FPGA for U-Turn

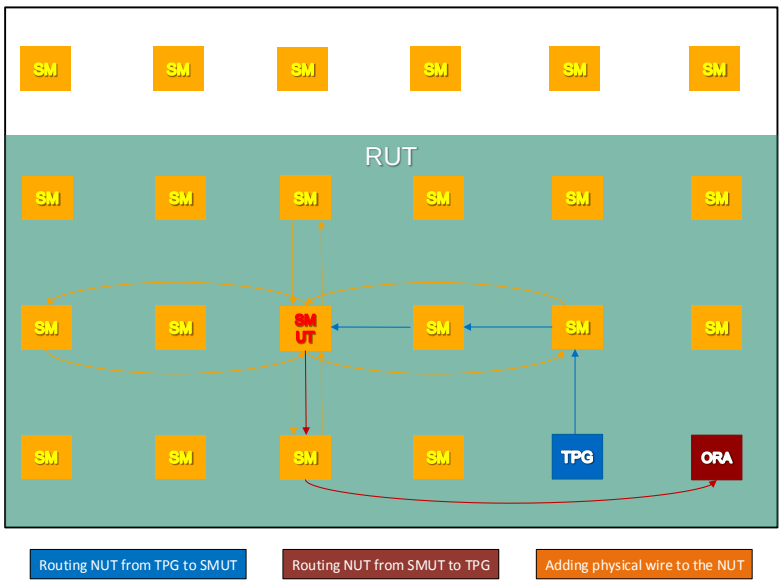


Figure 5.3: High level view of U-Turn

A high level pseudo code is listed below:

```
pick a SM in a RUT (SMUT);
    if the SMUT still contains non-visited physical wires:
        1   create a new empty NUT;
        2   add connection between the TPG and the SMUT to the
            NUT;
        3   create the graph;
        4   add to the current NUT a physical wire if and only if
            a) there is another physical wire at one of the
               destination SMs that allows to return to the SMUT;
            or
            b) an internal bounce[1] in the SMUT is possible
               (integrates stuck-at errors of the bouncing wires);
        5   if there are physical wires still connectable to the same
            NUT, go to 4;
        6   add connection between the SMUT and the ORA to the
            NUT
        7   store fully routed test design
repeat this process until all SMs have been analised;
```

5.1.1 NUT creation

Once the SMUT has been picked the algorithm searches for an untested physical wire among the physical wires connected to the SMUT. As soon as one is selected, called **Starting Physical Wire**, a new NUT is created with an associated counter. A NUT is represented as a list of nodes and edges while the counter denotes the number of untested physical wires in the NUT that are not tested before. At the beginning the NUT is empty

[1] Normaly a PIP is classified as an inPIP or outPIP but some of them are both. Therefore it is possible to drive a signal entering in a SM, through an inPIP, to a in/outPIP. Starting from it we can choose to drive the signal out of the SM (using it as an outPIP) or bouncing (using it as an inPIP) inside the SM and drive the signal to another PIP.

and the counter value is ‘0’, because the Starting Physical Wire will be added after connecting the TPG to the SMUT.

5.1.2 Connecting the TPG to the SMUT

At this point the NUT is initialized by adding physical wires and PIPs to connect the TPG to the Starting Physical Wire of the SMUT. Recalling the limitation on the PIPs and since we would maximize tested resources of the SMUT this path should be as short as possible. To reach this goal we use the *Iterative Deepening Depth-First Search* algorithm (IDDFS) [24], based on the *depth first search* strategy. With this strategy, a depth-limited search is run repeatedly, increasing the limit value of the depth at every step. Therefore the shortest path connecting the TPG to the SMUT is found.

5.1.3 Graph creation

It is particularly important to optimize the creation of the graph to minimize the occupied memory. Since the algorithm works on the nodes and edges also the computation time depends on the graph’s dimension. A first approach was to represent the entire region under test as a graph, but that solution was discarded because it was too onerous. As an example, considering an entire clock region as region under test of a Virtex-4 FX12, the machine would need more than 20 gigabytes of RAM. It is worth noting that this FPGA was selected because it is one of the smallest.

Due to the limitation of the number of PIPs that can be used in a NUT it is useful to create a graph with dimension limited to the area formed by resources that are directly reachable from the SMUT. To better understand this approach, Figure 5.4 highlights the connections of a SMUT in a Virtex-4 and the limits of the graph dimension.

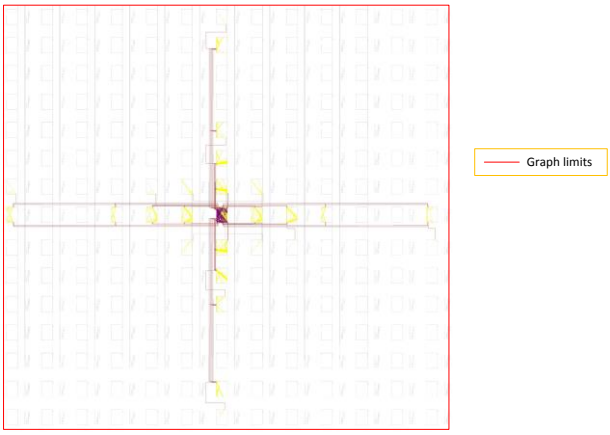


Figure 5.4: Graph limits for the Virtex-4

In this way a graph is created once for each Starting Physical Wire and only a few hundred megabytes of memory are occupied. As a result only physical wires inside the red square can be added to the NUT. It is important that the area has this extension otherwise is not possible to leave and return immediately to the SMUT, but it is necessary to use some physical wires that are not directly connected to the SMUT.

5.1.4 Populating the NUT

At this stage we have a graph whose root is the node that represents the Starting Physical Wire and a NUT connecting the TPG to the root. Then

a sort of breadth first search algorithm is executed on the graph to optimize the solution for the NUT in order to use as many physical wires as possible. During the execution of U-Turn 2 kinds of solution are created:

- **best solution** that is empty at the beginning;
- **temporary solution** that contains the actual NUT, therefore the connection from TPG to the root.

The temporary solution is modified adding all neighbour nodes, one at a time, of the root giving priority to those representing physical wires not tested. To achieve more randomness, a neighbour is taken casually and then the algorithm runs recursively on that neighbour.

Every time a node representing an exit point from the SMUT is selected, the temporary solution is compared with the best one: if the counter related to the best solution is smaller than the temporary one, then the best solution is overwritten with the temporary one. In this way the best solution will maximize the number of physical wires not tested in the current NUT. When the temporary solution reaches the threshold of PIPs, it is discarded and the algorithm can speed up returning to the previous level of neighbourhood. It is worth noting that loops should be avoided and, in order to achieve this, each node of the graph can be visited at most once.

5.1.5 Connecting the SMUT to the ORA

To complete the NUT we have to provide the connection to the ORA. Once again it is possible by using the IDDFS algorithm (see section 5.1.2).

5.1.6 Storing the full design

Finally as soon as all NUTs are computed, the algorithm creates the full testing circuits ready to be downloaded into a device to make the test. Figure 5.5 is an FPGA Editor screenshot that represent a full design of a NUT1 testing circuit targeting a switch matrix. Light blue lines represent connections between the TPG and the ORA not belonging to the Net Under Test. The red line is the NUT and as it can be seen it uses many resources belonging to one switch matrix under test. Figure 5.6 shows a zoom on that SMUT.

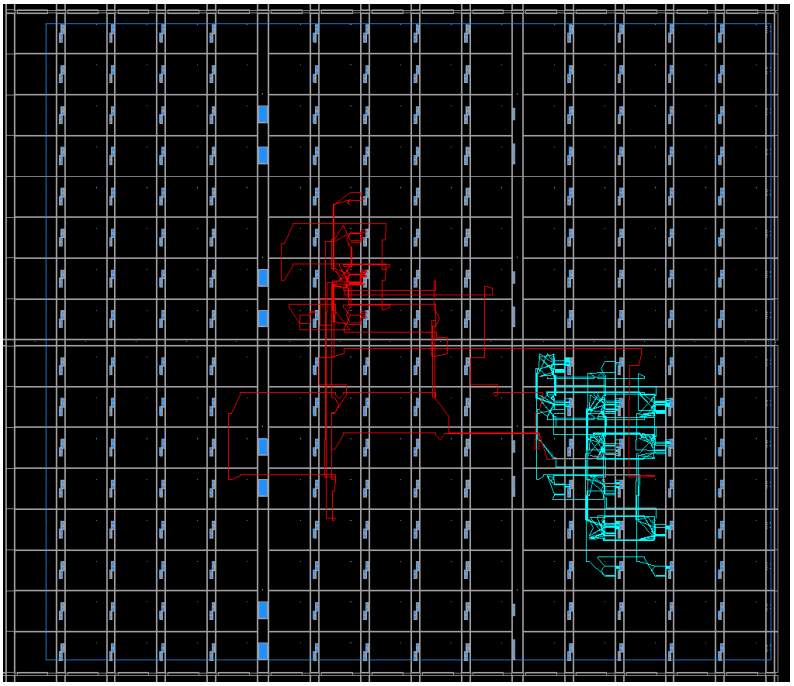


Figure 5.5: FPGA Editor screenshot of NUT1 full design

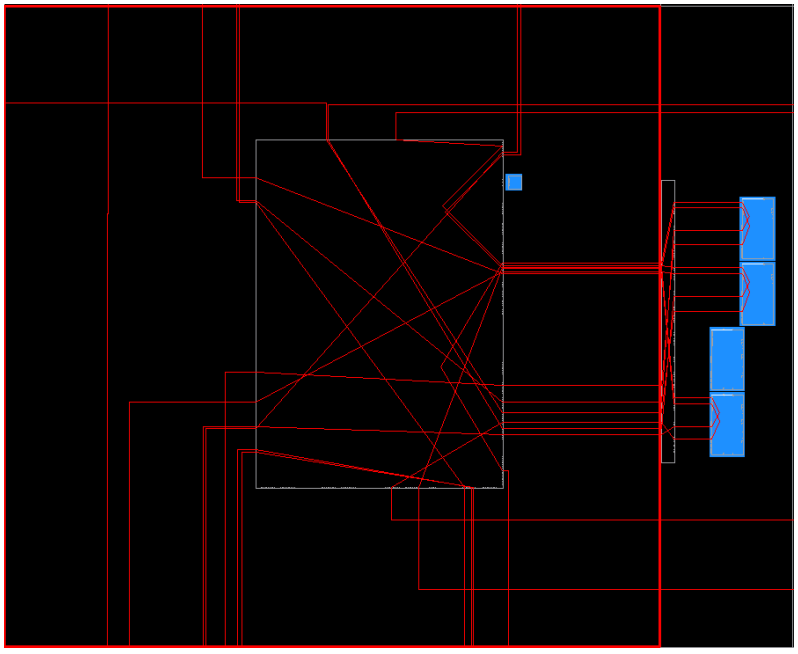


Figure 5.6: FPGA Editor screenshot of zoom on SMUT of NUT1

5.1.7 NUT6 and NUT8 special cases

It should be highlighted that all these steps have to be followed for each NUT, therefore if we are using the NUT6 or NUT8 testing circuit we need to perform them 6 or 8 times. However, attention must be paid to the independence of the NUTs of a single testing circuit otherwise the entire testing circuit does not work properly. Independence means that no routing resources have to be shared among the NUTs. When nodes are used in a NUT, they are marked as “already used” and cannot be added to NUTs belonging to the same testing circuit.

It can happen that after having found some NUTs is not possible to find another independent NUT for the current testing circuit. When it happens this physical wire is added to a special list and it will be tested by another testing circuit. If this occurs for the remaining physical wires of a SMUT, the testing circuit misses one or more NUTs. To avoid interfering with the correct behavior of the testing circuits, missing NUTs are added in a redundant way. Therefore with these redundant NUTs we are not testing anything new but the testing circuit is still working properly.

5.1.8 U-Turn parameters

The algorithm was implemented with the capability to change its behavior according to several parameters that are listed below:

- **Regions_to_test**

  It contains the names of target region we want to test;

- **HM_type**

  It is an integer value that represents the testing circuit type (NUT1, NUT6 or NUT8);

  Possible values: 1, 2, 3;

- **Testing_HM_XLD**

  It contains the path to the XDL file of the testing circuit we intend to use;

- **Compute_TCI_analysis**

  It is a boolean value that enables the analysis of testability without considering the testing circuit;

  Possible values: true, false;

- **Route_ORA_and_TPG**

  It is a boolean value that enables the routing from the TPG to the SMUT and from the SMUT to the ORA;

  Possible values: true, false;

- **PIP_limit_for_each_NUT**

  It is an integer value that represents the maximum number of PIPs that could be used for each NUT. It is set to 100;

  Possible values: any positive number;

- **PIP_limit_before_come_back**

It is an integer value that represents the maximum number of PIPs that could be used outside the SMUT before coming back to it. It is set to 7;

Possible values: any positive number;

- **Test_also_LOCAL**

  It is a boolean value that enables testing of LOCAL resources (see section 3.1.1). It is set to true;

  Possible values: true, false;

- **Target_level**

  It is an integer value that represents the test will be permormed:

  1 test of stuck-at-0/1 for the physical wires

  2 test of stuck-off for the PIPs

  3 test of stuck-on for the PIPs;

  Possible values: 1, 2, 3.

5.2 Why U-Turn

This section describes the reasons why we implemented this kind of algorithm. At the beginning two other architecture independent algorithm have been designed to maximize test coverage. Both of them use the graph representation of the FPGA as well as U-Turn.

The first one is a modified version of the breadth first search algorithm. Modifications are needed because the original BFS visits nodes only once and does not work with cyclic graphs. The modified version has been designed to visit nodes more times because it could allow to find longer paths exploiting cycles. The general behavior of that algorithm is to find several paths from a source to a destination node of the graph, then the longest one is chosen as the NUT. To do that we need to keep track, inside each node, of previous nodes because different paths can share some resources. Therefore the idea is to use the concept of colored graphs to label each complete path. Problems are in space and time complexity. Regarding the space each node has to store 3 additional items for each path the node belongs to (previous nodes, previous edges, color of the path) that affects the memory used. Concerning time it should be noted that if '*N'* is the number of nodes in the graph, the algorithm could visited all of them *'N'* times. Considering the dimension of the graph space and time complexity grow too much. With this algorithm we are sure to find the best solution for a NUT but its complexity is too high.

The second algorithm is a Heuristic one that improves space and time complexity of the previous one but does not ensure finding the best solution for a NUT. In this approach nodes have been used in previous NUTs should not appear in the next ones. To achieve this a weight, initialized to '0', is assigned to each node. The algorithm gives a higher priority to nodes with lower weight. The weight updating of generic node '*n*' is made following this formula:

$$Weight(n) = \alpha * UsedWeight(n) + (1 - \alpha) * DirectionWeight(n + 1, sink)$$

Where:

- **UsedWeight(n)** is the time that the node '*n*' has been used in previous NUTs;
- **DirectionWeight(n+1,sink)** is the orientation offset between a neighbour node '*n+1*' and the '*sink'* (destination) using Manhattan distance (details on Manhattan Distance are in [25]);
- **α = 0.7** because *usedWeight()* term should add more weight since used nodes should be avoided using again.

With this approach the only additional information is the weight on each node. Therefore this algorithm is faster and uses less memory than the previous one, but it was discarded because it could not find the best solution.

Finally we developed U-Turn because it is a good compromise between space and time complexity. Each node can be visited at most once and the only additional information is related to the “already tested” attribute.

6 Experimental results

In the following, figures of the U-Turn algorithm, that aims to use routing resources as much as possible, are reported. We run it on the following FPGA families:

- Artix-7
- Spartan-6
- Virtex-4;
- Virtex-5;
- Virtex-6.

Table 6.1 shows the number of switch matrices, physical wires and PIPs in the region under test, which coincides with an entire clock region of each tested FPGA.

Device	#Switch Matrices	#Physical Wires	#PIPs
Artix-7 XC7A100	1,600	169,037	1,941,969
Spartan-6 LX9	105	41,784	462,225
Virtex-4 FX 12	240	81,581	831,057
Virtex-4 FX100	672	222,240	2,323,415
Virtex-5 LX20T	360	133,405	1,495,969
Virtex-6 CX130T	1,480	498,621	5,621,864

Table 6.1: Number of Switch Matrices, Physical Wires, and PIPs for each tested device

The figures in Table 6.2 correspond to the Testable, Critical, Untestable and Unsupported physical wires in the region under test of each tested FPGA. The percentage is calculated on the total number of physical wires.

Device	#Testable Physical Wires	%	#Critical Physical Wires	%	#Untestable Physical Wires	%	#Unsupported Physical Wires	%
Artix-7 XC7A100	102,149	61%	7,183	4%	22,255	13%	37,450	22%
Spartan-6 LX9	26,026	62%	164	1%	6,379	15%	9,215	22%
Virtex-4 FX12	39,777	49%	3,409	4%	7,170	9%	31,225	38%
Virtex-4 FX100	114,778	52%	7,747	3%	16,811	8%	82,904	37%
Virtex-5 LX20T	80,064	60%	2,912	2%	17,299	13%	33,130	25%
Virtex-6 CX130T	331,684	66%	2,508	1%	28,269	6%	136,160	27%

Table 6.2: Physical Wires testability for each tested device

The figures in Table 6.3 correspond to the Testable, Critical, Untestable and Unsupported PIPs in the region under test of each tested FPGA. The percentage is calculated on the total number of PIPs.

Device	#Testable PIPs	%	#Critical PIPs	%	#Untestable PIPs	%	#Unsupported PIPs	%
Artix-7 XC7A100	1,254,235	65%	54,757	3%	350,357	18%	282,620	14%
Spartan-6 LX9	275,573	60%	2,879	1%	103,832	22%	79,941	17%
Virtex-4 FX12	456,783	55%	78,734	9%	104,876	13%	190,664	23%
Virtex-4 FX100	1,361,555	59%	191,544	8%	258,242	11%	512,074	22%
Virtex-5 LX20T	946,897	63%	87,017	6%	185,975	12%	276,080	19%
Virtex-6 CX130T	4,130,930	74%	70,946	1%	396,545	7%	1,023,443	18%

Table 6.3: PIPs testability for each tested device

The following tables show results achieved with the U-Turn algorithm when it runs on each tested FPGA. All of them have a similar structure where columns have the following meaning:

- **Test Structure** represents the type of testing circuit used;
- **Stuck-at-0/1 tested** represents the number of testable physical wire on which the fault stuck-at-0/1 is tested;
- **Stuck-off tested** represents the number of testable PIPs on which the fault stuck-off is tested;
- **Time** (in minutes) represents the time taken by the algorithm to compute all Nets Under Test;
- **#Testing circuits**: represents the number of testing circuits needed to test the entire region under test.

- **Artix-7 XC7A100T**

Test Structure	Stuck-at-0/1 tested	%	Stuck-off tested	%	Stuck-on tested	%	Time	#Testing circuits
NUT 1	102,149	100%	316300	25%	0	0%	22 min	8,042
NUT 6	102,149	100%	331,409	26%	490,145	39%	25 min	2,365
NUT 8	102,149	100%	329,897	26%	560,572	45%	27 min	2,039

Table 6.4: Artix-7 XC7A100T algorithm results

- **Spartan-6 LX9**

Test Structure	Stuck-at-0/1 tested	%	Stuck-off tested	%	Stuck-on tested	%	Time	#Testing circuits
NUT 1	26,026	100%	78,632	29%	0	0%	2 min	1,617
NUT 6	26,026	100%	78,467	29%	114,324	42%	3 min	604
NUT 8	26,026	100%	76,271	28%	125,505	46%	1 min	531

Table 6.5: Spartan-6 LX9 algorithm results

- **Virtex-4 FX12**

Test Structure	Stuck-at-0/1 tested	%	Stuck-off tested	%	Stuck-on tested	%	Time	#Testing circuits
NUT 1	39,777	100%	120,299	26%	0	0%	6 min	3,316
NUT 6	39,777	100%	118,597	26%	159,272	35%	6 min	1,073
NUT 8	39,777	100%	114,800	25%	169,781	37%	7 min	929

Table 6.6: Virtex-4 FX12 algorithm results

- **Virtex-4 FX100**

Test Structure	Stuck-at-0/1 tested	%	Stuck-off tested	%	Stuck-on tested	%	Time	#Testing circuits
NUT 1	114,778	100%	335,749	26%	0	0%	34 min	9,252
NUT 6	114,778	100%	362,198	27%	478,208	35%	54 min	2,977
NUT 8	114,778	100%	352,286	26%	509,572	38%	52 min	2,578

Table 6.7: Virtex-4 FX100 algorithm results

- **Virtex-5 LX20T**

Test Structure	Stuck-at-0/1 tested	%	Stuck-off tested	%	Stuck-on tested	%	Time	#Testing circuits
NUT 1	80,064	100%	249,391	26%	0	0%	10 min	6,727
NUT 6	80,064	100%	241,919	26%	373,990	40%	11 min	1,813
NUT 8	80,064	100%	235,170	25%	392,972	42%	13 min	1,569

Table 6.8: Virtex-5 LX20 algorithm results

- **Virtex-6 CX130T**

Test Structure	Stuck-at-0/1 tested	%	Stuck-off tested	%	Stuck-on tested	%	Time	#Testing circuits
NUT 1	331,684	100%	1,087,549	26%	0	0%	1h 12m	20,271
NUT 6	331,684	100%	1,091,914	26%	1,637,105	40%	1h 23m	6,769
NUT 8	331,684	100%	1,071,290	26%	1,850,052	45%	1h 23m	5,872

Table 6.9: Virtex-6 CX130T algorithm results

As we can see the 100% of coverage of physical wires is reached in any family of FPGA, proving the validity of the approach. Even if we are not targeting the PIPs, with this algorithm we test, on average, the stuck-off for 26% and the stuck-on for 40% of them.

The stuck-off test figures result from the fact that there exist more than one PIP to reach a physical wire. As an example Figure 6.1 shows all PIPs (colored in yellow) that can be used to reach a physical wire (colored in red).

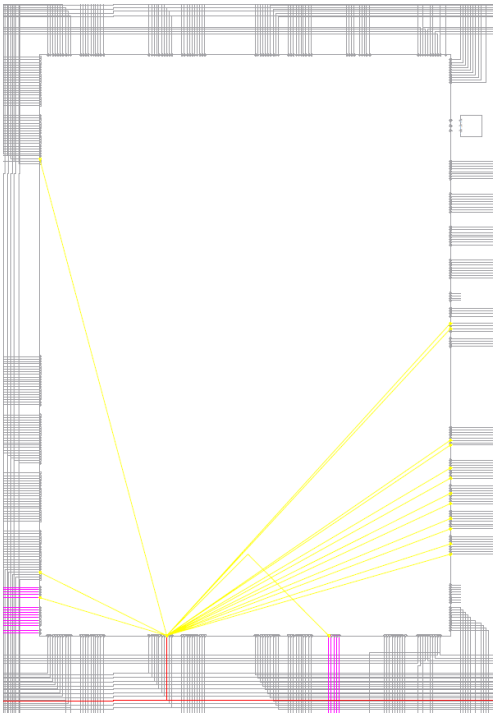


Figure 6.1: FPGA Editor screenshot for PIPs connected to a physical wire

Taking into account this example we should use 16 times the same physical wire, in different NUT, changing every time the PIP used to test the stuck-off for all these PIPs.

Regarding the stuck-on we reach better results than the stuck-off because of the algorithm behavior. Recalling that test of stuck-on is made using NUTs that can be shorted by a PIP, the closer are the NUTs, the larger is the probability that they can be shorted. Reducing the value of PIP_limit_before_come_back parameter (presented in section 5.1.8), we force the NUTs to come back to the SMUT as soon as possible. As a result the probability to short two NUTs is incremented because they will use resources that belong to the same SMUT, therefore are very close to each other.

We run the algorithm changing its parameters in order to test more PIPs and we found out a combination of them to increase these numbers (see Table 6.10), but not enough to reach the 100%. It could be a future work.

- **Virtex-4 FX100 (enhanced)**

Test Structure	Stuck-at-0/1 tested	%	Stuck-off tested	%	Stuck-on tested	%	Time	#Testing circuits
NUT 1	114,778	100%	1,130,421	83%	0	0%	2h 40min	113,753
NUT 6	114,778	100%	819,929	60%	1,161,915	86%	2h 10min	11,124
NUT 8	114,778	100%	704,264	52%	1,087,306	80%	1h 53min	7,048

Table 6.10: Virtex-4 FX100 algorithm results (enhanced version)

As in section 5.1 the complete flow includes also the connection of the TPG and the ORA. This connection adds several routing resources that do not belong to the NUT and therefore they are not tested directly. A test is successfully passed if and only if the following conditions occur:

1. testing circuit is working properly;

2. no faults affect the NUT (or NUTs).

As a result if a test is passed we are able to verify that the resources belonging to the NUT and also those connecting the TPG and the ORA are free of faults. This is a positive side-effect due to the architecture of the testing circuit itself.

On the other hand if the test is not passed we cannot assure that the fault is in the NUT. A good approach should be to use resources already tested, and fault free, as connection between the TPG and the ORA.

7 Conclusions and future work

In this thesis a routing algorithm to maximize fault coverage of permanent faults in routing resources of FPGAs is presented. An optimization of its behavior has been assessed during the implementation. The time used by U-TURN to complete the routing of all NUTs is about one hour with a memory occupation of about 4 gigabytes. Many tests were made in order to find the best combination of parameters. Moreover, U-Turn is designed to operate with any FPGA and it provides several parameters that allow the user to change its behavior without making any change to the code. The developed algorithm achieved the objective of 100% of coverage of physical wires.

Currently the placement of the TPG and the ORA is made manually and their location never change during the execution. A future work should provide a smart placement of these components in order to obtain several improvements:

- minimizing the distance between these components and the NUT making tests using fewer resources unrelated to the SMUT, thus increasing the number of physical wires directly connected to the SMUT with respect to the total number of physical wires in the NUT;
- placing more than one testing circuit in the same region under test thus providing a substantial speed-up of the entire test;

- ensuring that the TPG and the ORA use already tested resources to make tests more reliable.

Moreover, since the algorithm works on a representation of the FPGA, it is possible to make changes to the algorithm in order to target the PIPs instead of the physical wires by maximizing the use of edges instead of nodes.

Finally another future work could be the design of new testing circuits that allow to test those resources that are currently unsupported.

8 Bibliography

[1] W. Huang, F. Meyer, N. Park e F. Lombardi, «Testing Memory Modules in SRAM-based Configurable FPGAs,» in *Proceedings of the Interational Workshop on Memory Technology, Desing and Testing*, August 1997.

[2] M. Renovell, J. Portal, J. Figuras e Y. Zorian, «Minimizing the Number of Test Configurations for Different FPGA Families,» in *Proceedings of the Eighth Asian Test Symposium*, 1999.

[3] J. Smith, T. Xia e C. Stroud, «An Automated BIST Architecture for Testing and Diagnosing FPGA Interconnect Faults,» in *Journal of Electronic Testing*, 2006.

[4] M. Abramovici, C. Strond, C. Hamilton, S. Wijesuriya e V. Verma, «Using roving stars for on-line testing and diagnosis of fpgas in fault-tolerant applications,» in *Proceedings of the International Test Conference*, 1999.

[5] M. Rozkovec, J. Jenicek e O. Novak, «Application Dependent FPGA Testing Method,» in *Proceedings of the 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, September 2010.

[6] M. Tahoori, «Application-Dependent Testing of FPGAs,» in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2006.

[7] C. Bernardeschi, L. Cassano, M. G. C. A. Cimino e A. Domenici, «Gabes: A genetic algorithm based environment for SEU testing in sram-fpgas,» in *Journal of Systems Architecture*, 2013.

[8] J. S e V. K. Agrawal, «Detection and diagnosis of faults in the routing resources of a sram based fpgas,» *International Journal of Computer Applications*, September 2012.

[9] X. Sun, P. Trouborst, J. Xu e B. Chan, «Novel technique for built-in-self-test of fpga interconnects,» in *Proceedings of the IEEE International Test Conference*, 2000.

[10] J. Yao, B. Dixon, C. Stroud e V. Nelson, «System-level built-in self-test of global routing resources in virtex-4 fpgas,» in *Proceedings of the 41st Southeastern Symposium on System Theory*, 2009.

[11] Xilinx, «FPGA Editor Guide,» September 2015. [Online]. Available: http://www.xilinx.com/support/sw_manuals/2_1i/download/fpedit.pdf .

[12] Xilinx, «ISE In-Depth Tutorial,» April 2012. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/ise_tutorial_ug695.pdf.

[13] Xilinx, Partial Reconfiguration Flow presentation Manual, Xilinx

University Program, 2015.

[14] Xilinx, «Partial Reconfiguration User Guide,» January 2012. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/ug702.pdf.

[15] L. Cassano, D. Cozzi, S. Korf, J. Hagemeyer, M. Porrman e L. Sterpone, «On-Line Testing of Permanent Radiation Effects in Reconfigurable Systems,» in *Design, Automation & Test in Europe Conference & Exibition (DATE)*, March 2013.

[16] D. Sorrenti, D. Cozzi, S. Korf, L. Cassano, J. Hagemeyer, M. Porrman e C. Bernardeschi, «Exploiting Dynamic Partial Reconfiguration for On-Line On-Demand Testing of Permanent Faults in Reconfigurable Systems,» in *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, October 2014.

[17] R. Baumann, «Radiation-induced soft errors in advanced semiconductor technologies,» in *IEEE Transaction on Device and Material Reliability*, September 2005.

[18] J. Wang, «Radiation effects in FPGAs,» in *Proceeding of the 9th Workshop on Electronics for LHC Experiments*, October 2003.

[19] N. Battezzati, L. sterpone e M. Violante, Re-configurable Field Programmable Gate Arrays for Mission-Critical Applications, July

2010.

[20] D. Sorrenti, «Exploiting Partial Dynamic Reconfiguration for On-Line On-Demand Detection of Permanent Faults in SRAM-based FPGAs,» in *master thesis University of Pisa*, 2013.

[21] Xilinx, «Configuration and Readback of Virtex FPGAs Using JTAG Boundary-Scan,» February 2007. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp139.pdf.

[22] Xilinx, «Command Line Tools User Guide,» December 2009. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/devref.pdf.

[23] C. Beckhoff, D. Koch e J. Torresen, «The Xilinx Design Language (XDL): Tutorial and Use Cases,» in *Reconfigurable Communication-centric System-on-Chip (ReCoSoC)*, June 2011.

[24] D. Cozzi, «Homogeneous communication router for Xilinx FPGAs,» in *master thesis Politecnico di Torino*, 2010.

[25] P. E. Black, «Mahnattan distance,» [Online]. Available: http://xlinux.nist.gov/dads/HTML/manhattanDistance.html.