UNIVERSITÀ DI PISA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
Laurea Magistrale in Ingegneria Informatica

Exploiting Partial Dynamic Reconfiguration for On-Line On-Demand Detection of Permanent Faults in SRAM-based FPGAs

Master's Thesis

Supervisors:
Prof. Cinzia BERNARDESCHI
Prof. Andrea DOMENICI

Candidate
Domenico SORRENTI

Academic Year 2012/2013

Acknowledgements

I would like to thank those who have helped me during the development of this thesis, making it clear that I take the blame for each mistake, which could be found in this work.
First of all, thanks to my supervisors Prof. Cinzia Bernardeschi and Prof. Andrea Domenici for their support and valuable advice.
I thank the Cognitronics and Sensor Systems Group for allowing me to develop my thesis and join the group, in particular Prof. Mario Porrmann, Dario Cozzi, Jens Hagemeyer and Sebastian Korf for their continuing help and useful suggestions.
I would like to thank Luca Cassano for his constant assistance before, during and after my visiting period in Germany.
I also thank Lydia Ntokou, Anna Isakova and Luca Santangelo for keeping me company during my period in Germany.
Finally, I would like to thank my family, Laura and her family for their support. This work is dedicated to them.

Contents

List of Figures

List of Tables

Abstract

FPGAs become ever more popular thanks to their features, such as reconfigurability and short time to market. When FPGAs operate in harsh environments, like in space, soft faults can occur (SEU) due to radiation, as well as permanent faults (TID, Aging). Testing of logic resources has already been widely considered in literature, on the other hand this work aims at detecting permanent faults which can affect the interconnection infrastructure. In modern FPGAs the routing resources represent up to 80% of the whole chip area. Few works consider permanent faults and none of them deals with on-line testing of permanent faults with independent test circuits. In this work a first approach is presented, where different circuits have been developed and tested on a DB-V4 daughterboard of the RAPTOR prototyping system.

Introduction

FPGAs are increasingly being used in safety-related and critical applications, for this reason testing of these devices is gaining importance. Unfortunately, existing testing techniques developed for other devices, such as ASICs, are not adequate, therefore new techniques have been and are still being developed.
Around twenty years ago, permanent faults were the main problem, then with the transistors industry progress, that is smaller transistors, they have lost importance with respect to transient ones. Nowadays, the new devices employ small transistors, but in a really high number, and hence permanent faults are gaining importance again. Testing the interconnection infrastructure, which in modern devices can reach up to 80% of the chip area, is gaining importance although many works are focused only upon logical resources. Sometimes it is the application or the operating environment that requires fault detection, for example, the Adaptive Computing Systems (ACSes) rely on reconfigurable hardware for adapting the system to changes of the external environment, exploiting hardware sharing, to increase the functional density and reduce power consumption, and reusing modules for different applications. Where a direct human intervention is not possible, such as in space, fault detection and fault tolerance mechanisms are essential. Various fault tolerance techniques exist, but these are not always suitable, for example, redundancy and voting algorithms are expensive in terms of space, weight and power for these systems, which are typically subject to strict constraints.
The goal of this work is to conceive, develop and verify a mechanism which permits permanent faults in routing resources of FPGAs to be detected.
The thesis is organized as follows:

1. **Background** (Chapter 1), which provides a brief description about FPGAs, architecture and configuration, the more common testing approaches, the Xilinx Description Language and the Dynamically Reconfigurable Processing Module;

2. **Related Work** (Chapter 2), which reports some works associated to FPGA testing;

3. **Testing Circuits** (Chapter 3), in which the developed mechanism is shown in detail;
4. **Testing Circuits Verification** (Chapter 4), in which the developed mechanism is verified and related results are shown;
5. **Conclusions and Future Work** (Chapter 5), in which final remarks and possible improvements are reported.

Chapter 1

Background

1.1 Field Programmable Gate Array

Field Programmable Gate Arrays (FPGAs) are integrated circuits, which feature the capability to be reconfigured via software more times; as consequence of this flexibility, these devices present some advantages with respect to other technologies, such as ASICs (Application Specific Integrated Circuits).
ASICs require high investments during the design phase, in particular about non-recurring expenses (NRE), whereas FPGA functionalities are not fixed by the vendor, which can fabricate them in large scale lowering the price. Then end users can adapt an FPGA to their own need by means of a configuration. This leads to a sizeable decrease of the TTM (Time To Market), which is quite important from an economic point of view. But probably the most important advantage with respect to ASICs is the possibility to correct errors without further costs and any time. Thanks to this feature FPGAs were initially largely used for prototyping and exactly this represents the reason of the starting diffusion of these devices.
There are also some disadvantages, such as a higher cost for unit in high volume designs and a bigger size with respect to ASICs, which are realized custom. In particular, FPGAs can take up an area 10 times bigger and be at least 3 times slower than ASICs, the latter aspect is the consequence of replacing metal tracks (ASICs) with programmable blocks containing active elements (FPGAs), which introduce propagation delay. A more complete treatise can be found in [1].
Another possible comparison is with general purpose processors, which feature a high flexibility like FPGAs, but they have typically a higher power consumption and a lower elaboration time. For further information see [2].
With time passing FPGAs cost is decreasing, whereas their performance is increasing, therefore nowadays FPGAs can be used for digital signal processing, hardware acceleration, cryptography and so on; because of their

flexibility they are suitable for several fields of application.

1.1.1 FPGA Architecture

In this section a brief description about FPGA architecture is given. FPGA architecture is vendor dependent, but the most common consists in a matrix of configurable logic blocks (CLBs) connected through a routing infrastructure.

An FPGA is made of three essential blocks (Figure 1.1):

- CLB, in which the logic functions are implemented;
- Input-Output Block (IOB), which represents connection with outside;
- Switch matrix (INT), by which is possible to drive signals inside an FPGA. Routing is obtained by means of Programmable Interconnection Points (PIPs), which are CMOS transistors that can be activated and deactivated in order to get a custom path within a switch matrix. Switch matrices are linked together by fixed wires.

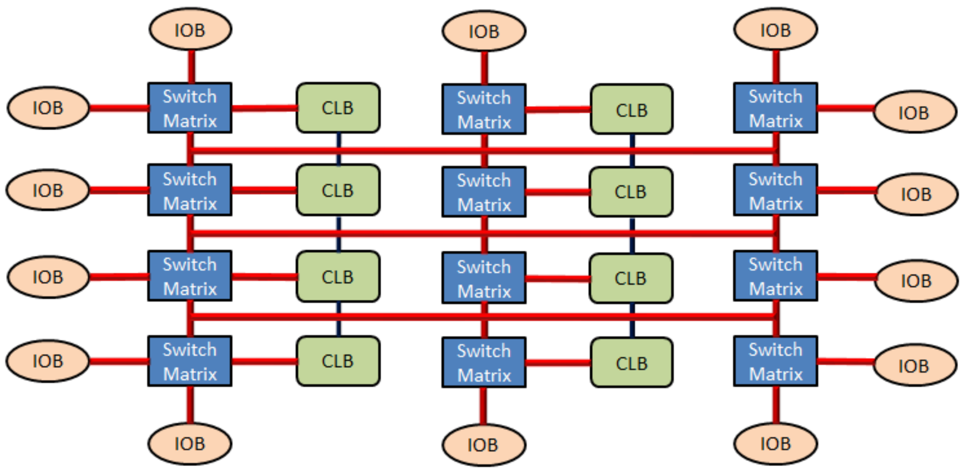


Figure 1.1: FPGA common architecture

CLB The CLB structure depends on vendors and FPGA families, for example a Xilinx Virtex-4 CLB consists of 4 slices (Figure 1.2), whereas in a Virtex-5 CLB there are 2 slices.

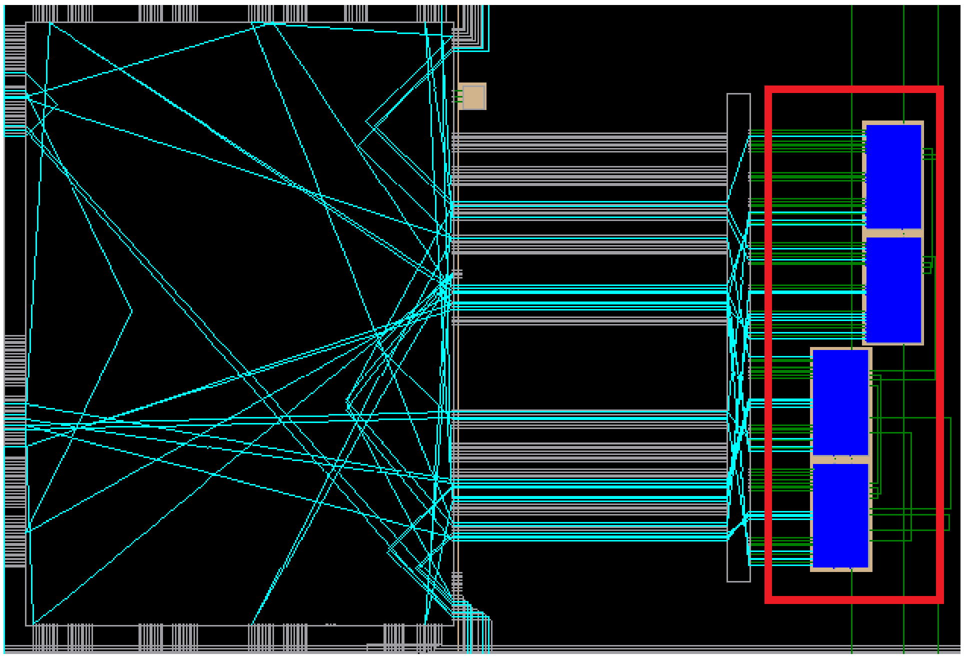


Figure 1.2: Virtex-4 CLB

Focusing on Virtex-4, a slice is split in part F and G. Each part consists, from a simplified viewpoint, in a 4-input Look Up Table (LUT) and a Flip-Flop (FF). There are two types of slice:

- Slice L (Figure 1.3);
- Slice M.

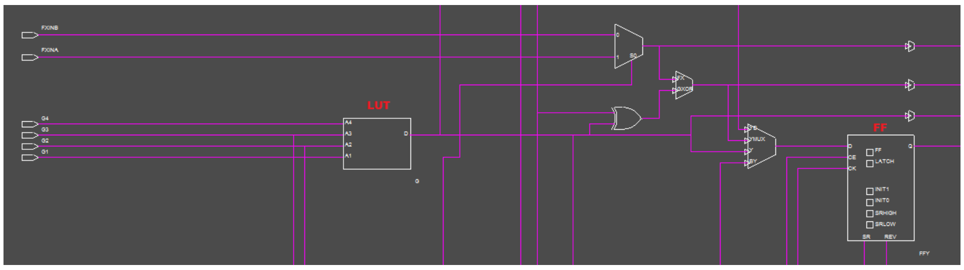


Figure 1.3: Virtex-4 Slice L part G

In a slice M, LUTs can operate as a ROM, shift register or distributed RAM.

PIP Regarding an FPGA architecture 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 switched element, whose state is determined by the value contained in a configuration cell (see Figure 1.4).

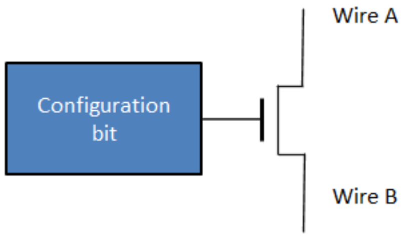


Figure 1.4: A configuration bit determines the state of a PIP

Several types of PIPs are used:

- **Cross Point PIP** (Figure 1.5), which connects wire segments placed in different plans, that is a horizontal segment with a vertical one or vice versa;
- **Break Point PIP** (Figure 1.6), which connects wire segments placed in a same plan, that is two horizontal or vertical segments;
- **Multiplexer PIP** (Figure 1.7), which in turn is split in:
  - **Decoded multiplexer**, which is made of 2^k Cross Point PIPs connected to a common output wire segment. It is driven by k configuration bits;
  - **Non-Decoded multiplexer**, which differs from a Decoded one because it consists of a single configuration bit for each CMOS transistor, in other words k wire segments are controlled by k configuration bits.
- **Compound PIP** (Figure 1.8), which is made of four Cross Point and two Break Point PIPs.

For further information see [3].

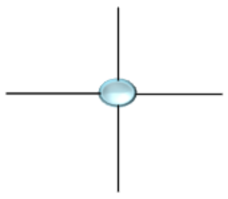


Figure 1.5: Cross Point PIP

Figure 1.6: Break Point PIP

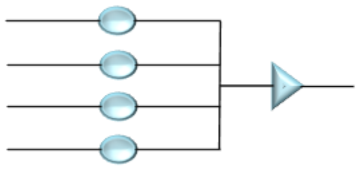


Figure 1.7: Multiplexer PIP

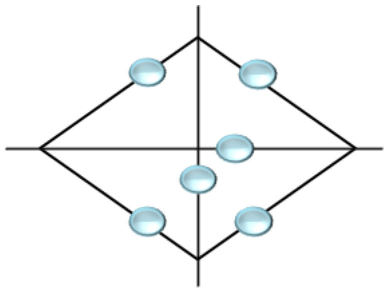


Figure 1.8: Compound PIP

1.1.2 Programming an FPGA

FPGAs are programmable devices, for this reason their behaviour has to be defined by the end user, which can do that by means of either a hardware description language (HDL) or a schematic design. The former should be preferred when the design is rather big and the most used HDLs are Verilog and VHDL.
By the time in which the behaviour has been defined, using a design automation tool, typically supplied free of charge by vendors, a technology-mapped netlist is generated. 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 an actual FPGA architecture. On the result of the previous process, several verification methodologies, such as timing analysis and simulation, can be performed. Once that the result is verified, a binary file, called bitstream, is generated and loaded into the device in order to configure or reconfigure it.
In Xilinx language *design implementation* is the process of translating, mapping, placing, routing and generating a bitstream file for a given design. All these tools are included and integrated in the Xilinx® ISE® Design Suite.
Figure 1.9 depicts the whole flow:

- the HDL file represents the input;
- **Synthesize**, which generates a supported netlist type (EDIF or NGC) for the Xilinx implementation tools;
- **Implement design**, which is made of:
  - **Translate**, which converts input design netlists and creates a single NGD netlist;
  - **Map**, which maps the design into CLBs and IOBs;
  - **Place and Route**, which places the design and calculates routes for the nets.
- **Generate Programming File**, which generates the bitstream which is loaded into the device.

For further information about the Xilinx design automation tool see [4].

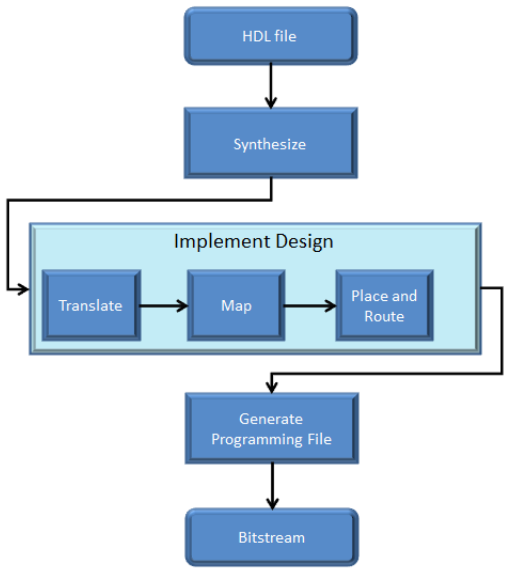


Figure 1.9: Xilinx design implementation

1.1.3 Configuring an FPGA

Once that a bitstream is ready, it has to be loaded into a device. This section presents how a bitstream can be loaded and in which way an FPGA can be reconfigured, as well as a short description about the configuration memory and the bitstream format.

Partial reconfiguration

FPGAs feature a high flexibility due to the possibility to be reprogrammed via software; with the **Partial Reconfiguration** (PR) they have reached an even higher flexibility level, inasmuch as the PR permits to reconfigure a specified part of the device, without affecting the others.
An example of how PR works is depicted in Figure 1.10, 1.11 and 1.12.
Figure 1.10 shows a scenario where there are three bitstreams in the configuration memory and two bitstream queues, which involve different parts of the device.
In Figure 1.11 the bitstream 1 has been replaced without affecting the part of the device where other bitstreams are currently operating.
Finally, in Figure 1.12 the bitstream A has been replaced with the bitstream B, without affecting other device parts.

PR groups Two groups of PR exist:

- **dynamic** PR, which permits to reconfigure some parts of the device, without disrupting applications which are currently operating on the

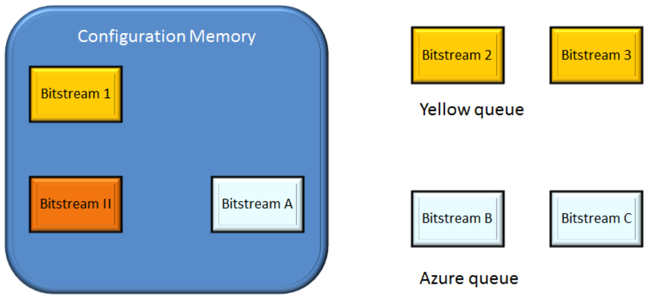


Figure 1.10: PR starting scenario

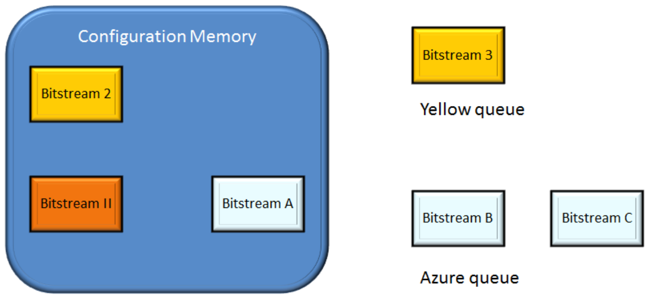


Figure 1.11: PR bitstream 1 replacing

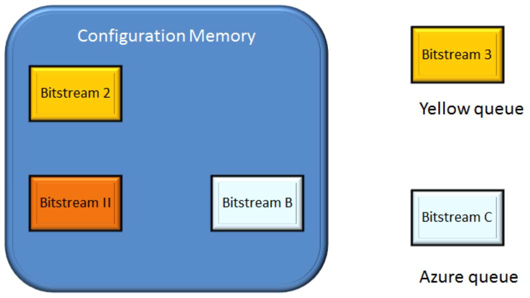


Figure 1.12: PR bitstream A replacing

rest of the device;

- **static** PR, in which, during a reconfiguration, the rest of the device is stopped until the reconfiguration has been accomplished.

PR styles There are two different styles of PR:

- **module-based**, where a modular design conceived for reconfiguring large logic blocks is used (for further information see [5]);
- **difference-based**, which is foreseen for dealing with small changes in a design.

Two different ways exist for obtaining these small changes:

- a *front-end approach*, where it is needed to re-synthesize and re-implement the design in order to get a NCD (Native Circuit Description) file and thence a partial bitstream, for instance, using Xilinx® ISE® Design Suite;
- a *back-end approach*, where changes are directly made with FPGA Editor, a Xilinx tool used for displaying and configuring FPGAs.

Why use the PR?

There are several benefits using a partial reconfiguration in place of a full one, such as:

Application A requirement of various applications is the capability to adapt their own behaviour on the basis of different events which can occur, in particular in the space or aerospace field, but not only. These changes could be, for instance, the adoption of a different protocol or algorithm, basing on collected or observed data. These applications were plainly not supported by previous architectures, for the reason that a full configuration is not suitable.

Performance The PR permits to reconfigure a specified part of the device, without affecting the others, that means that the system continues to operate without service interruption.

Resource sharing The PR allows more applications to run at the same time on a single device and hence this leads to a better utilization of hardware resources, and therefrom a reducing power consumption and decreasing cost.

Configuration time Although it could seem an almost worthless benefit, with the possibility of loading only a partial bitstream, therefore reprogramming only a part of the device, in contrast to a full bitstream, which reconfigures the whole device, the configuration time has undergone a sizeable lowering. Table 1.1 reports the length of some full bitstreams, related to the Virtex devices, and it notes that a full reconfiguration can take a considerable amount of time to be performed.

Family	**Device**	**Size(MiB)**
Virtex-4	XC4VFX100	3.94
Virtex-5	LX330T	9.86
Virtex-6	XC6VLX760	22.03
Virtex-7	7V2000T	53.33

Table 1.1: Bitstream lengths

How perform a partial reconfiguration?

There are four interfaces which can be used for loading a bitstream:

1. **Serial** interface;
2. **SelectMAP** interface;
3. **ICAP** interface;
4. **JTAG** interface.

Some information about these interfaces are here reported, for full details see [6].

Serial interface With the use of this interface an FPGA is configured by loading one configuration bit for each CLK cycle.

SelectMap interface This interface provides a bidirectional 8, 16 and 32bit bus and can be used both for configuration and readback, that is the process of reading back data from an FPGA, initially conceived to verify whether the design was loaded properly.

ICAP interface The Internal Configuration Access Port (ICAP) permits the access to the configuration data in a way similar to the SelectMap interface. It can be used to perform both PR and readback operation, but not a full configuration.

JTAG interface The use of this interface was standardized by IEEE in 1990 (IEEE 1149.1 standard) and the Virtex-families are fully compliant with it.

Configuration Memory Architecture Overview

The configuration memory in Virtex-4 FPGA is frame-based, that is it is arranged in frames. With respect to the previous Virtex families, these frames have a fixed length: 41 words (for further information [7], [8]).

Bitstream structure overview A full bitstream may be thought as made of five parts:

- an informative part, with information such as design name, generating time and so on;
- a dummy word $FFFFFFFF_{16}$ and a synchronize word $AA995566_{16}$, for alignment issues;
- a sequence of packets, each one consists of a header and a payload;
- configuration words;
- another sequence of packets.

An example of full bitstream is exhibited in Figure 1.13.

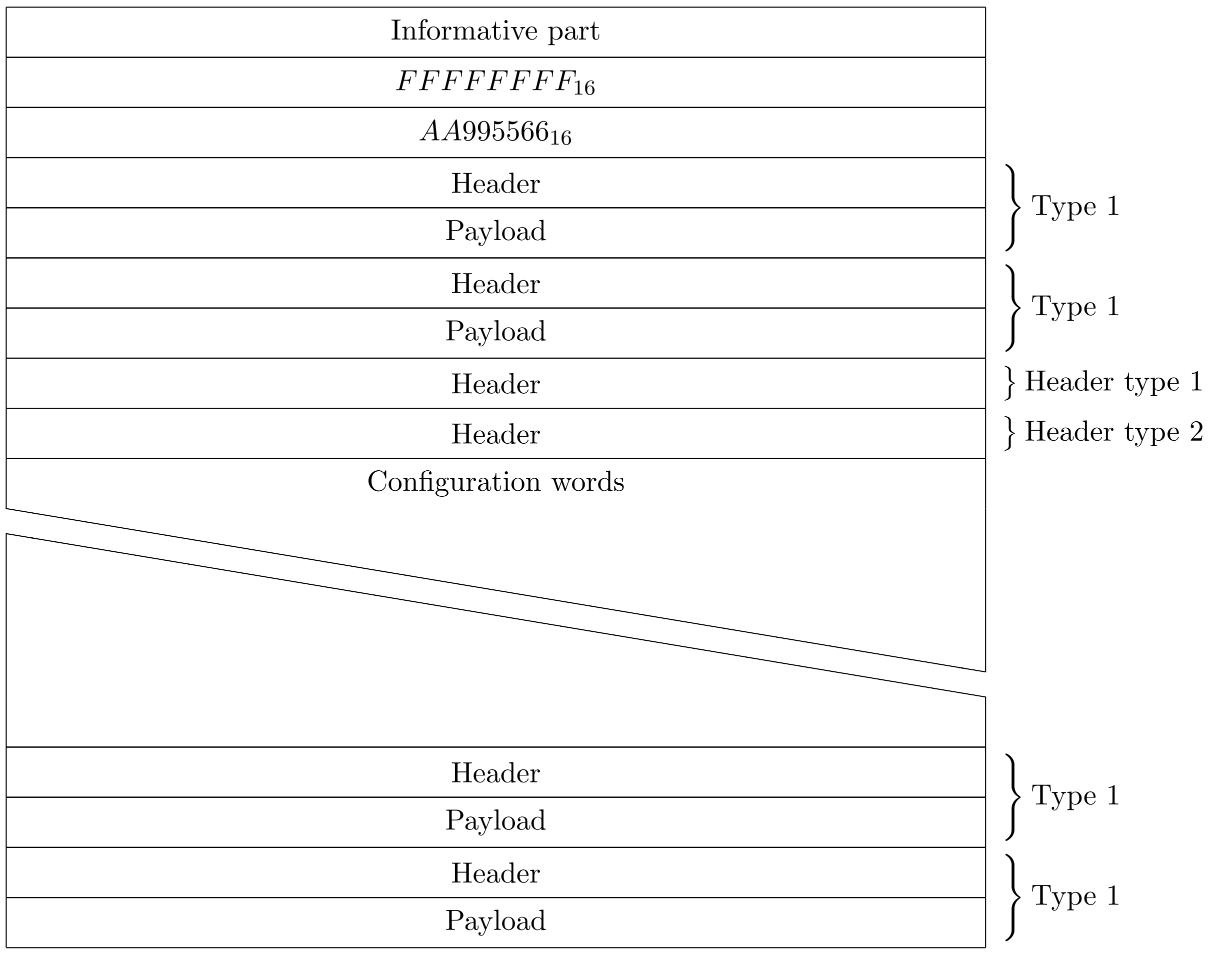


Figure 1.13: Full bitstream example

Packets The packet header specifies what operation has to be performed, the address of the register in which to write and the word count, that is the number of data words which follow the header and that have to be written in a specified register. There are two packet types: *type 1* and *type 2*. The former is far more common than the second one, indeed the latter has been created only to bypass the case in which the number of words, which are specified in the word count field in the packet $1'$s header, are too much with respect to the field size. In this particular case a packet 2 is used, which has not an address field in his header, but only the word count field, whereas the address has to be specified by a previous packet 1 with count word field equals to 0. In other words, a packet 2 is always preceded by a packet 1.

Configuration registers Some fundamental registers, which are involved during the configuration process, are hereinafter reported.

- **Cyclic Redundancy Check** (CRC). Inside the bitstream there is a precalculated 16 bit checksum, if no disabled, which is written in the CRC register. The configuration logic calculates the checksum during the configuration process and compares it with what is written in the CRC register. This option allows checking whether the transferring data succeeds or not, in this last case an error is signalled;

- **IPCODE**. In this particular register a word has to be written, which is unique for each device. The configuration logic compares the IPCODE′s content with a hard-wired unique word and in case of mismatch an error is signalled;

- **Frame Address Register** (FAR). This register takes a fundamental role in loading the configuration frames, it represents the place where the address of the first frame is written. For a full bitstream this is always 0_{16} and the configuration logic is in charge of incrementing it for every frame which is received.

Frame Address Format

The frame address, whose format is shown in Figure 1.14, is a 32 bit word and consists of several parts:

- 9 bits are not used;
- **Top/Bottom** (1 bit)
- **Block Type** (3 bits)
- **Row Address** (5 bits)
- **Column Address** (8 bits)
- **Minor Address** (6 bits)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Not Used									T	Type			Row					Column								Minor					

Figure 1.14: Frame address format

The various parts which make up a frame address are described below.

Top/Bottom Virtex-4 devices have been designed as separated in two halves: *top* and *down* half. When a full bitstream is loaded, the FAR address is written with 0_{16}, which represents the upper-center and left-most column of the device. In Figure 1.15 the top half is selected with the top/bottom bit.

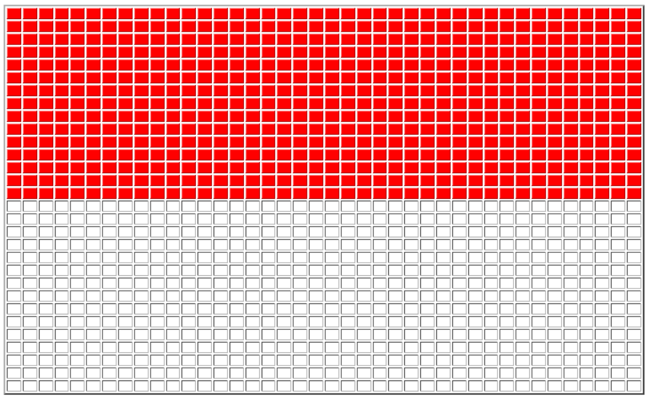


Figure 1.15: FAR Top/Bottom

Block Type This part of the address specifies what kind of resources will be configured by that frame. It can assume the following value:

- 000_2: CLB, IO, DSP, CLK, MGT;
- 001_2: block RAM interconnect;
- 010_2: block RAM content;
- 011_2: it is reserved;
- 100_2: it is reserved;
- 101_2: it is reserved;
- 110_2: block RAM data integrity.

Row Address It is used to select a row of frames and has been introduced with the Virtex-4 family. Its number increases with the distance from the center of device, both in top and bottom half. In Figure 1.16 a row in the top half is selected with the row address.

Column Address This field selects one column of frames in the row which is specified by the Row Address field, obviously in the half of device which has been specified by the Top/Bottom field. It starts with 0_2 and is incremented moving toward right in the device. In Figure 1.17 one column in the row is selected with the column address.

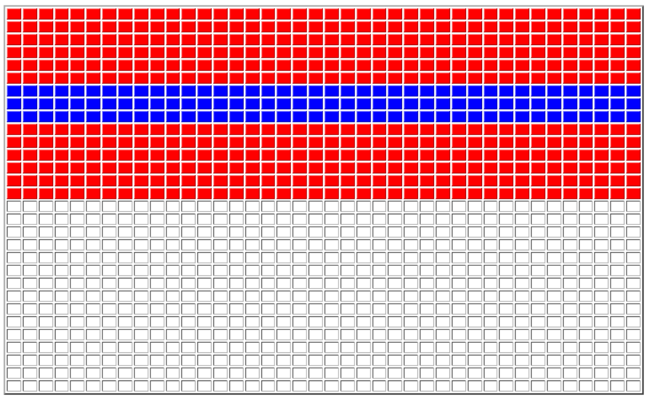


Figure 1.16: FAR Row Address

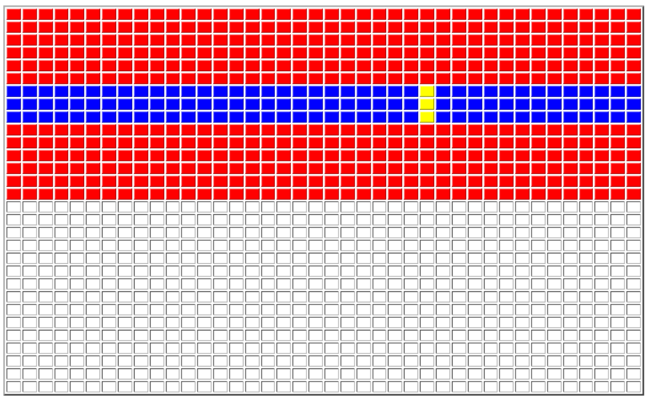


Figure 1.17: FAR Column Address

Minor Address It selects one frame inside a column of frames, in fact a frame represents the smallest addressable part in a device and a way for indicating directly a word within a frame does not exist. Finally, in Figure 1.18 one frame in the column is selected with the minor address.

1.1.4 FPGA testing

FPGAs are suitable for many fields of application. In particular, the growing use of FPGAs in safety-critical systems requires that the system does not fail because of faults and that it features a gradual degradation of performance due to aging. Moreover nowadays the testing has acquired a lead role in the life cycle of these devices, due, again, to their increasing use in critical applications.
This section briefly introduces possible causes of faults, which can affect these devices, a first fault classification based on duration of their effect, and a description about faults which can occur in the PIPs of the interconnection

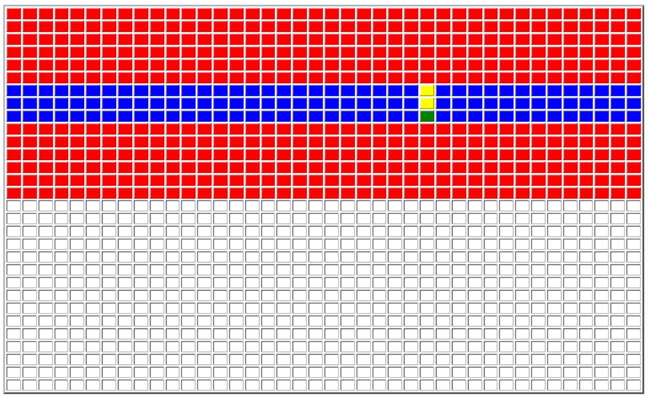


Figure 1.18: FAR Minor Address

infrastructure.

Fault causes

FPGAs are electronic devices and hence prone to an unavoidable aging of their own components, with a consequent degradation in terms of performance, which can lead to a failure. Figure 1.19 depicts the "bathtub" curve which is typically used for indicating the failure rate evolution with respect to time.

In addition, FPGAs can operate in harsh environments, for example in space, where radiations can affect the device causing even a failure. Figure 1.20 depicts how the radiation effect can propagate itself throughout the system and can lead to a failure.

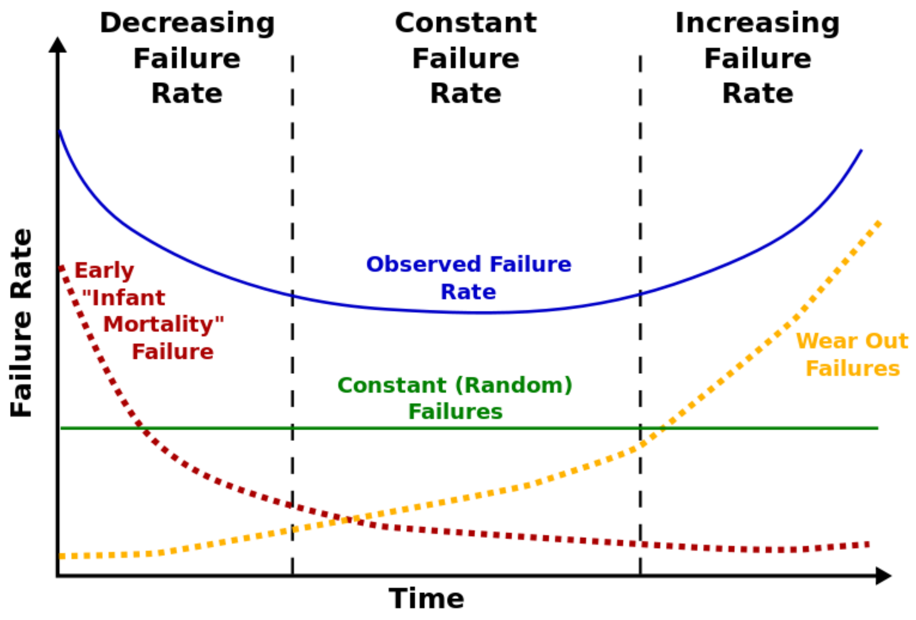


Figure 1.19: Failure rate: bathtub curve

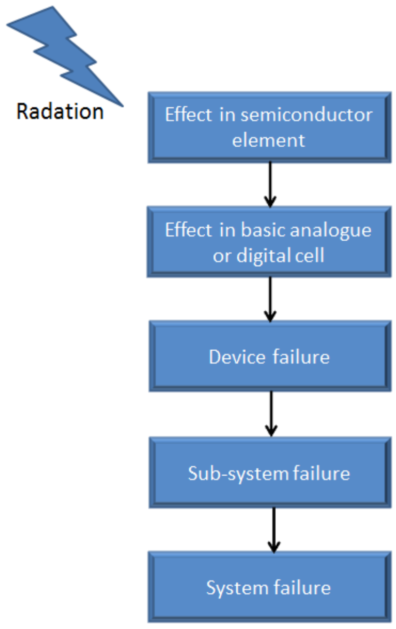

Radiation
Effect in semiconductor element
Effect in basic analogue or digital cell
Device failure
Sub-system failure
System failure

Figure 1.20: Effect propagation chain

Fault classification

An FPGA can be affected by:

- **Single Event Upset** (SEU), that is the effect of a particle on a memory element, such as a LUT or a register, which causes a change of the stored value;
- **Total Ionizing Dose** (TID), that is an accumulation of charge in the silicon injected by radiations; when the charge reaches a sizeable value, a transistor is not no longer able to switch between state on and off.

As SEU belongs to the *soft* error type [9], on the other hand, TID belongs to the *permanent* one. For a more complete classification see [3].
Some works on SEU and TID are here reported:

- **SEU**
  - In [10] a prover of SEUs unexcitability based on a model checker tool is shown;
  - In [11] an automatic test pattern generation tool for detection of SEUs is presented; it is based on the use of Genetic Algorithms (GAs);
  - In [8] Xilinx proposes how dealing with SEU using a combination of PR and readback.
- **TID**
  - In [12] a complete testing and diagnosing flow is proposed; it consists of several available tools.

PIPs behaviour in presence of fault

As previously said, a fault can affect the configuration memory, in particular a configuration bit which controls a PIP. Consequences can be different:

- **Open**;
- **Conflict**;
- **Input Antenna**;
- **Output Antenna**;
- **Bridge**;

Figure 1.21 depicts a switch matrix, whose PIPs are not affected by faults.

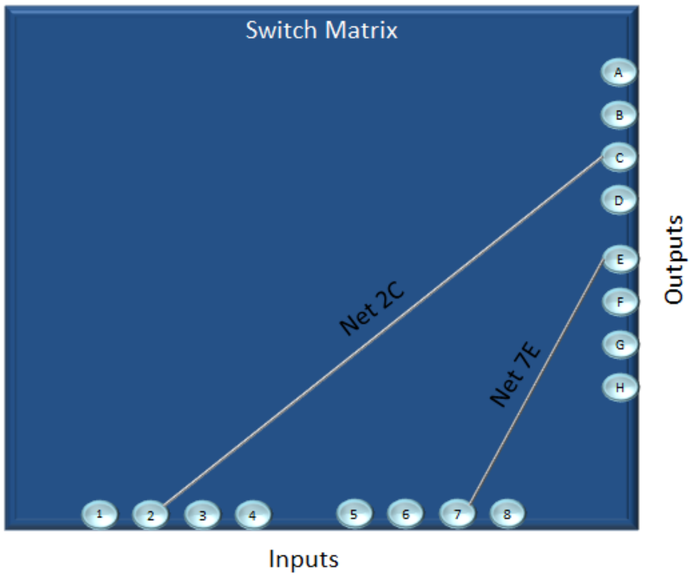


Figure 1.21: Switch matrix with no faulty PIPs

Open There are two possible cases of open:

- Figure 1.22, where the net *7E* is deleted, because the PIP is no longer programmed;
- Figure 1.23, where the net *7E* is deleted and a new net, for example, *5E*, between an unused input node 5 and the previously used ouput node E, is created.

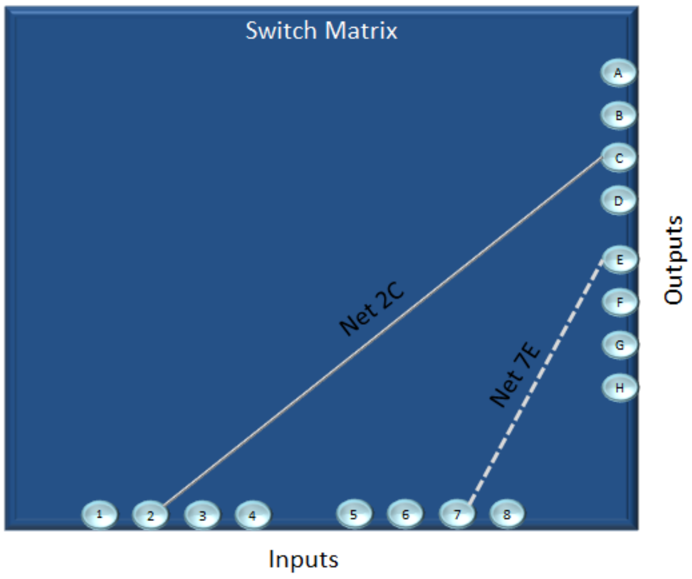


Figure 1.22: Open PIP case 1

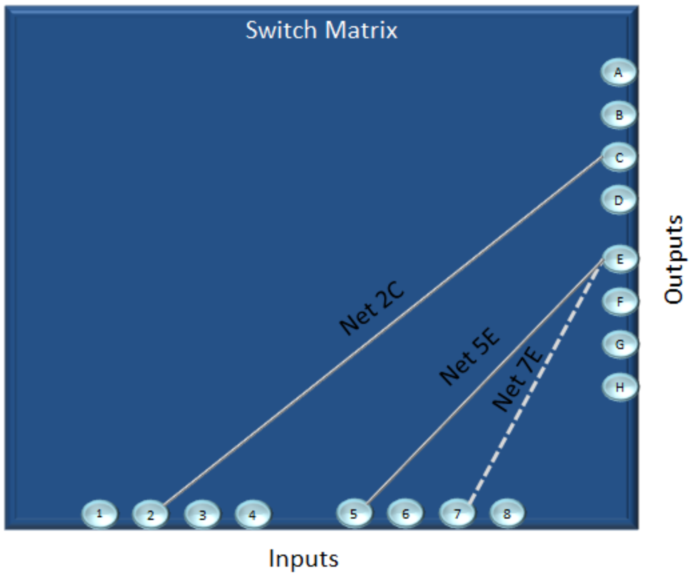


Figure 1.23: Open PIP case 2

Conflict Figure 1.24, where a new PIP is added between two previously used input and output nodes (net *7C* is created).

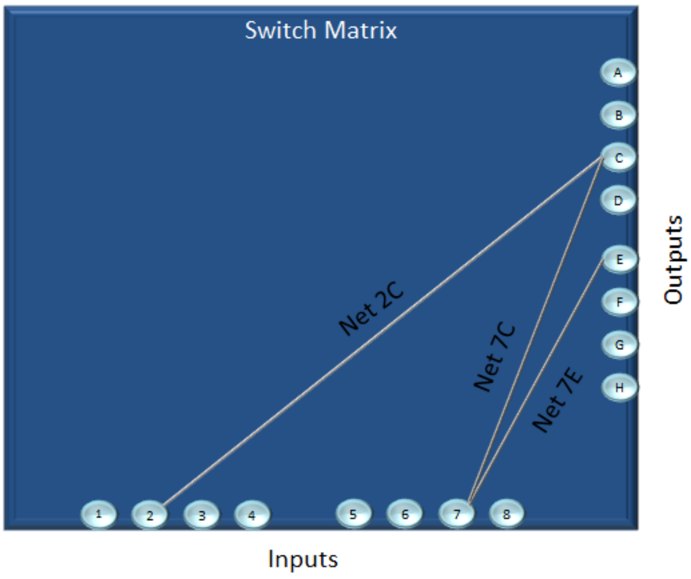


Figure 1.24: Conflict PIP

Antenna Two types of antenna are possible:

- *Input*: Figure 1.25, where a new PIP is programmed creating a new net, for example, *4C* between an unused input node 4 and a used output node C;
- *Output*: Figure 1.26, where a new PIP is programmed creating a new net, for example, *2D* between a used input node 2 and an unused output node D .

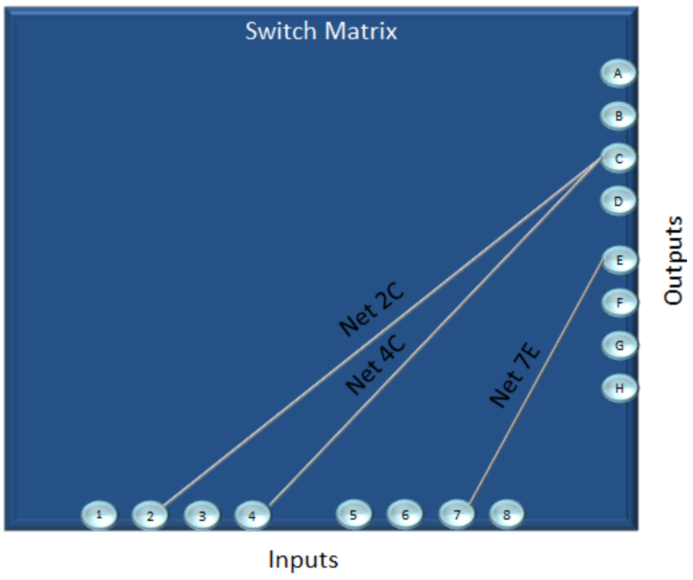


Figure 1.25: Input Antenna PIP

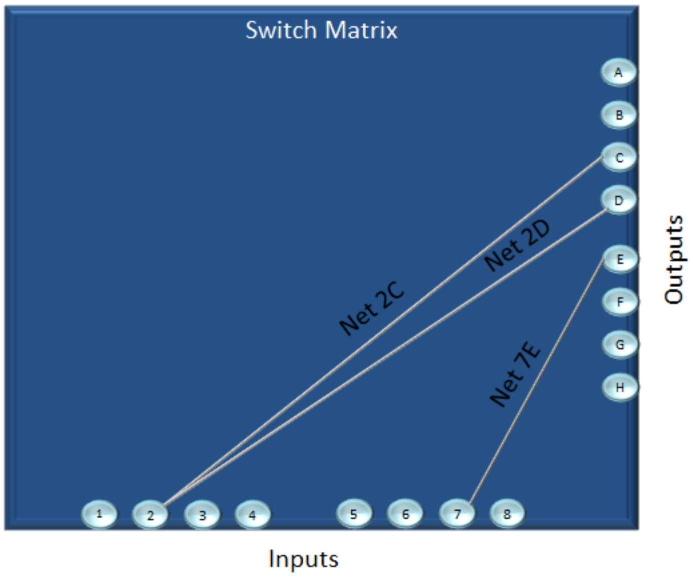


Figure 1.26: Output Antenna PIP

Bridge Figure 1.27, where the net *7E* is deleted and a new net, for instance, *2E*, between a used input node and the output node of the net *7E*, is created.

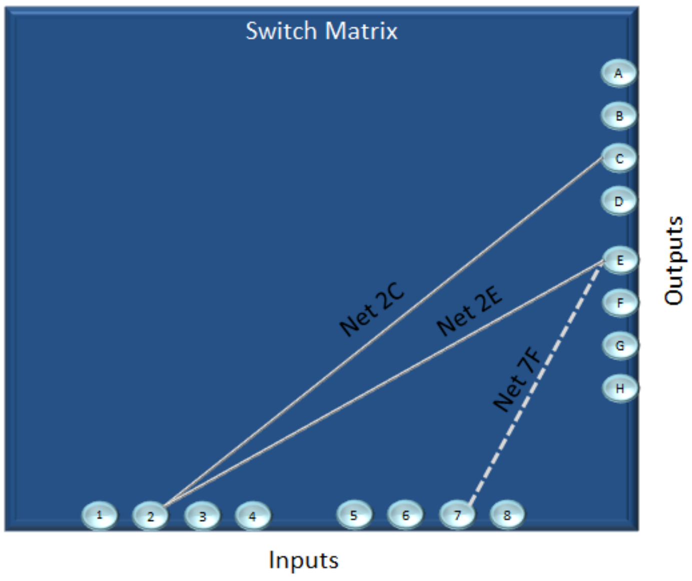


Figure 1.27: Bridge PIP

For further information and more details see [3].

1.1.5 Testing approaches: External and Built-in-Self-Test

There are various testing methodologies and they can be classified in two big families:

- **Application-independent testing**, in which the goal is to detect every structural defect because of the manufacturing process in the whole FPGA;
- **Application-dependent testing**, in which the aim is to detect defects only in resources actually used by the design.

In order to perform whatever test, the following items are needed:

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

Amongst various methodologies the most common are the external and the **Built-in-Self-Test** (BIST) approach.
The former approach was the first utilized and consists in considering the circuit under test like a black box, providing input stimuli from outside the device through IOBs or JTAG interface, for example. Then responses are collected by means of IOBs or a readback and they are externally assessed.
On the other hand, the BIST approach, which is currently the most utilized, allows a device to test itself without acting from outside. This approach is faster, simpler, inexpensive and does not require external test equipment, whereas its disadvantages are additional design requirements and area overhead. The advent of the PR has helped to diffuse the BIST approach, because it allows a device to test its own parts without disrupting applications which are currently running.
The idea on which the BIST approach is based is to configure some CLBs as **Test Patter Generator** (TPG), which is in charge of generating input stimuli, and some others as **Output Response Analyzer** (ORA), which is in charge of assessing responses of the CUT and giving a related result, that is whether the CUT is faulty or not. Figure 1.28 depicts a simple BIST scheme, where the CUT is represented by a **wire under test** (WUT). With the term wire a set of PIPs and physical wire segments is meant.

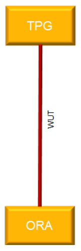


Figure 1.28: BIST approach scheme

In [13] the first BIST approach for testing the interconnection infrastructure in an FPGA is reported.

BIST approach evolution From its first definition the BIST approach is passed through various improvement steps:

- **Comparison-Based**;
- **Parity-Based**;
- **Cross-Coupled Parity Based**.

Comparison-Based This was the initial stage, in which the name derives from the fact that the ORA compares responses received on the WUTs. It represents the most simple BIST approach, where the TPG generates identical signals on the WUTs, which are received by the ORA. A problem with this approach is that equivalent faults (that is faults with the same effects) are not detected. In addition, if, for example, the TPG is faulty and provides same signals on the WUTs, the ORA is not able to detect the fault. Figure 1.29 depicts the Comparison-Based approach with two WUTs.

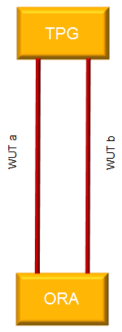


Figure 1.29: Comparison-Based BIST approach

Parity-Based This version was introduced for the first time in [14]. It consists in realizing the TPG with a n-bit counter, therefore the input stimuli are 2^n, and using a parity bit, which is transmitted on fault-free routing resources. This version is affected by the fact that if the parity remains correct even in presence of a fault (for example if the TPG is frozen to a fixed input), the fault is not detected by the ORA. With respect to the Comparison-Based approach, now the ORA is able to detect equivalent faults on the WUTs, and sometimes if the TPG is faulty, this can be detected through a readback of the state of the counter. Figure 1.30 depicts the Parity-Based approach with two WUTs and the parity bit.

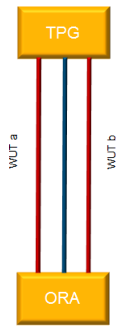


Figure 1.30: Parity-Based BIST approach

Cross-Coupled Parity Based This version is the most recent, it aims to outdo the too rigid constraint of transmitting the parity on fault-free routing resources. It is realized through the use of two n-bit counters, which compose the TPG, whose parity bits are exchanged and sent to two different ORAs (see Figure 1.31); in this way if a TPG is faulty the ORA is able to detect the fault, without needing to resort to a readback of the state of the counters. It was proposed in [15] for Virtex-4 devices.

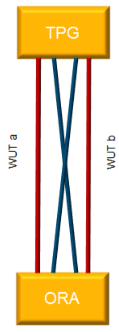


Figure 1.31: Cross-Couple Parity BIST approach

1.2 Xilinx Description Language

The Xilinx Description Language (XDL) is a human-readable format compatible with the more known and used Netlist Circuit Description (NCD). Both these languages have been created by Xilinx for describing and representing FPGA designs.
With the XDL it is possible to obtain a full description about:

1. the available resources in an FPGA. This report can occupy from a few MiBs (smaller or older devices) up to several GiBs (newer devices);

2. FPGA netlists, such as complete systems, hard macros or modules.

Notwithstanding the same language is used for both cases, descriptions differ in both syntax and structure [16].

What is a hard macro? A hard macro is a pre-placed and pre-routed design block. If a hard macro is moved along an FPGA, all its elements maintain the same position within it.

1.2.1 DHHarMa

DHHarMa[17] is a software flow which exploits the XDL, in particular it can be used for the automatic generation of homogeneous HMs and it offers the possibility to constrain both placing and routing phases.

1.3 Dynamically Reconfigurable Processing Module (DRPM)

The project in [18], under ESA contract, is aimed at designing, developing and validating a DRPM demonstrator. The system updates and adapts its functionalities through a dynamically exchange of processing modules at run-time. Because of the harsh environment in which the system operates, that is in space, particular attention was paid about mitigation of radiations and recovery mechanisms in case of failure. The DRPM is particular indicated for space mission in which there is the need to adapt data processing algorithms basing on the collected data.
The DRPM consists in three fundamental components:

- **RAPTOR-X64** baseboard;
- **DB-V4** daughterboard;
- **DB-SPACE** daughterboard.

1.3.1 RAPTOR-X64

It is a rapid prototyping system based on a modular approach: the base system provides the communication infrastructure and some management facilities, which can be used by various extension modules. Figure 1.32 depicts the RAPTOR-X64 baseboard.

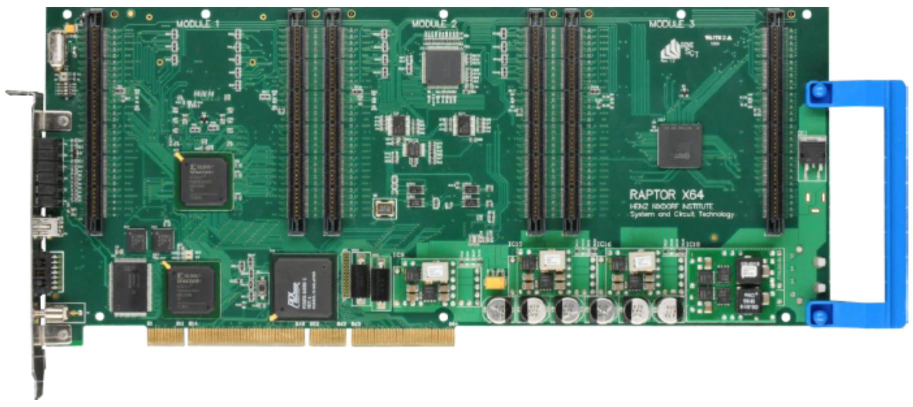


Figure 1.32: The RAPTOR-X64 baseboard

1.3.2 DB-V4

It is a RAPTOR-X64 extension module and hosts:

- Xilinx Virtex-4 FX100 FPGA;

- 4 GiB DDR2 RAM.

Figure 1.33 depicts the DB-V4 daughterboard.

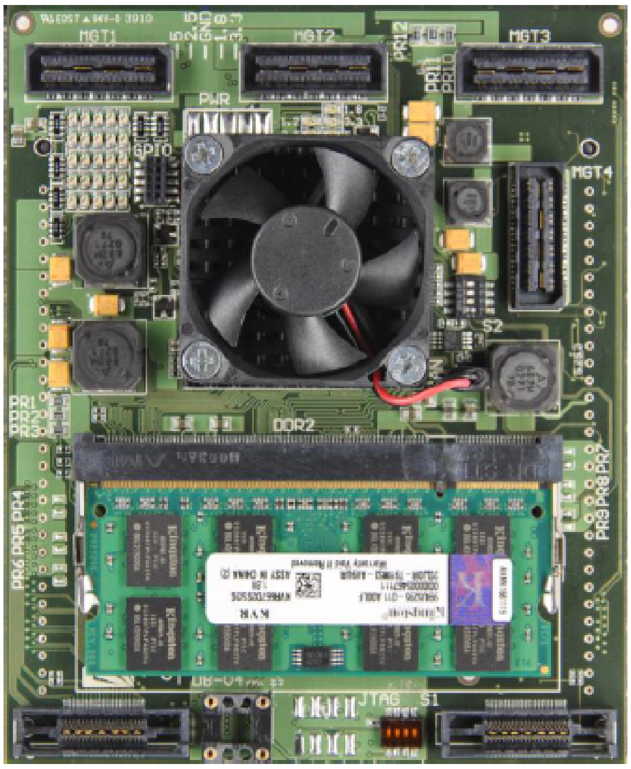


Figure 1.33: The DB-V4 daughterboard

PR-FPGA

It is the main payload processing module of the system. The FPGA consists in three essential parts:

- **Static processing** components part, which includes a memory controller, which is connected to a DDR2 RAM, and several communication bridges;
- **Self-hosting reconfiguration** (SHR) components part, which includes a MicroBlaze processor and is in charge of carrying out the PR through the ICAP interface;
- **Dynamic processing** components part, which can be dynamically reconfigured in order to adapt the system to changed environment situations or to perform on-demand operations.

Figure 1.34 depicts the PR-FPGA module. The Host PC is the backplane for the RAPTOR-X64 baseboard.

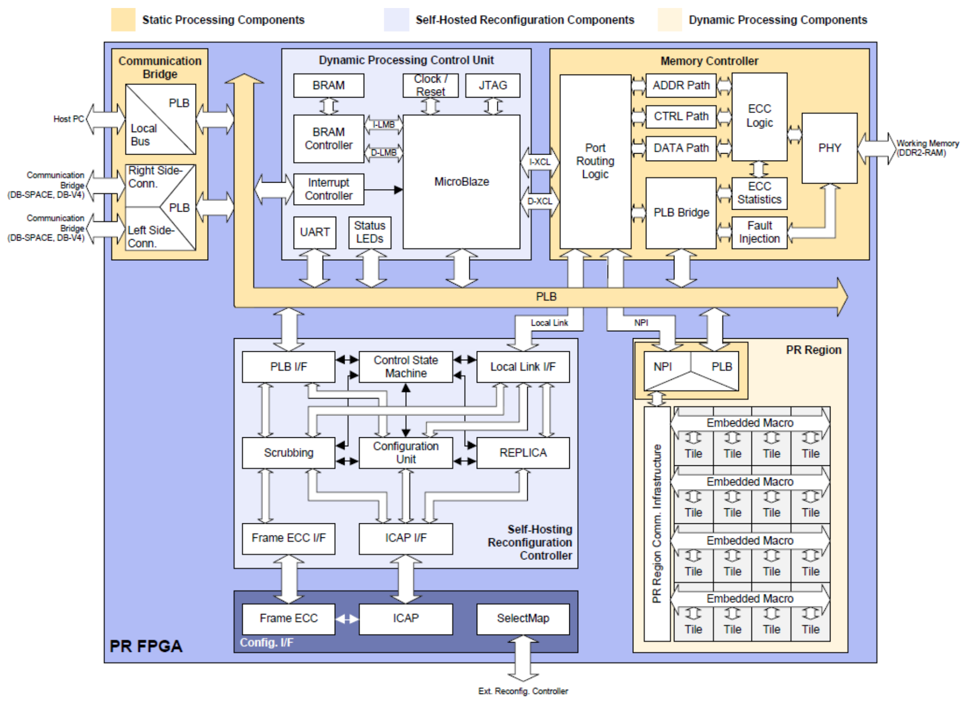


Figure 1.34: The PR-FPGA module

1.3.3 DB-SPACE

This daughterboard is conceived for space missions, in fact it hosts a *Leon2-FT* core, which is the SEU fault tolerant (FT) version of the Leon2 processor, in particular FFs are protected with *triple modular redundancy* (TMR), while internal and external memories with *Error Detection And Correction* (EDAC) mechanisms. Figure 1.35 exhibits the DB-SPACE daughterboard.

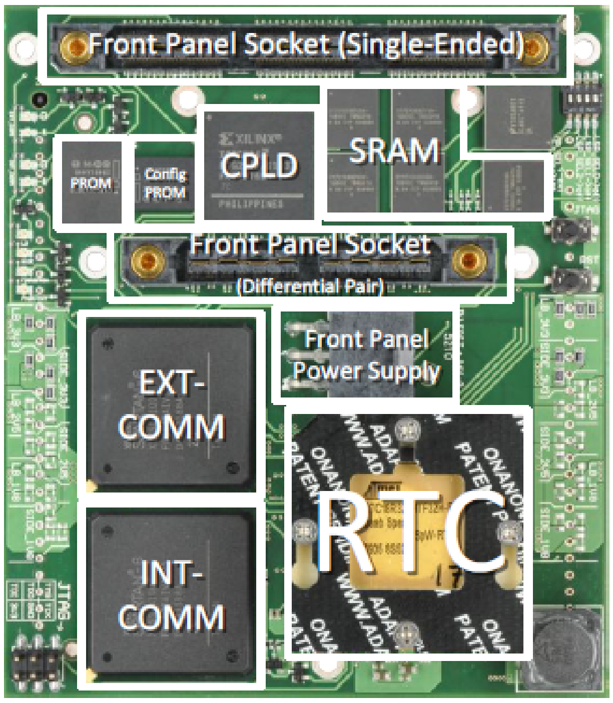


Figure 1.35: The DB-SPACE daughterboard

1.4 Thesis Statement

Critical operations are processed on the DB-SPACE daughterboard, where the various components are fault tolerant, but there are cases in which some important operations are performed elsewhere, in fact the PR-FPGA is typically used on-demand by the Leon2-FT processor, which delegates some operations to it.
For example, the Leon2-FT processor has to take some decisions about moving and, for some reasons, it may need to calculate a Fast Fourier Transform (FFT), therefore it delegates this operation to the PR-FPGA. The MicroBlaze processor loads a bitstream from the DDR2-RAM, through the memory controller, into the SHR controller for performing the PR. The loaded bitstream is relative to a hard macro which is able to perform the FFT. Supposing the chosen empty space, in which the hard macro has been placed, is in somehow faulty and hence the hard macro could not work or worse could give a wrong result.
The aim of this thesis is to provide some mechanisms, in particular hard macros (HMs), which allow the MicroBlaze processor to detect faults in the chosen space. The processor could decide whether launching testing periodically, during idle time, or just before to execute a new PR, that is on-demand.
Figure 1.36 depicts a scheme in which fault detection is not carried out, the scenario is that the Leon2-FT processor delegates an operation A to the MicroBlaze processor, which performs all the due steps to obtain the operation A results. On the other hand, when fault detection mechanisms are available, the previous scenario can be changed as depicted in Figure 1.37, where before loading the hard macro related to the operation A, the MicroBlaze processor may decide of verifying the chosen space with one or a set of testing HMs.

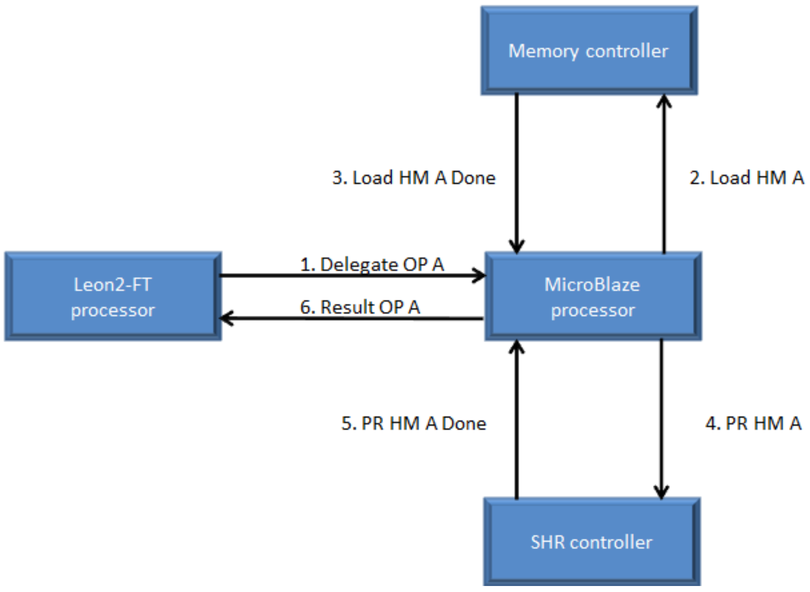


Figure 1.36: Example of interaction between Leon2-FT and MicroBlaze processors without fault detection

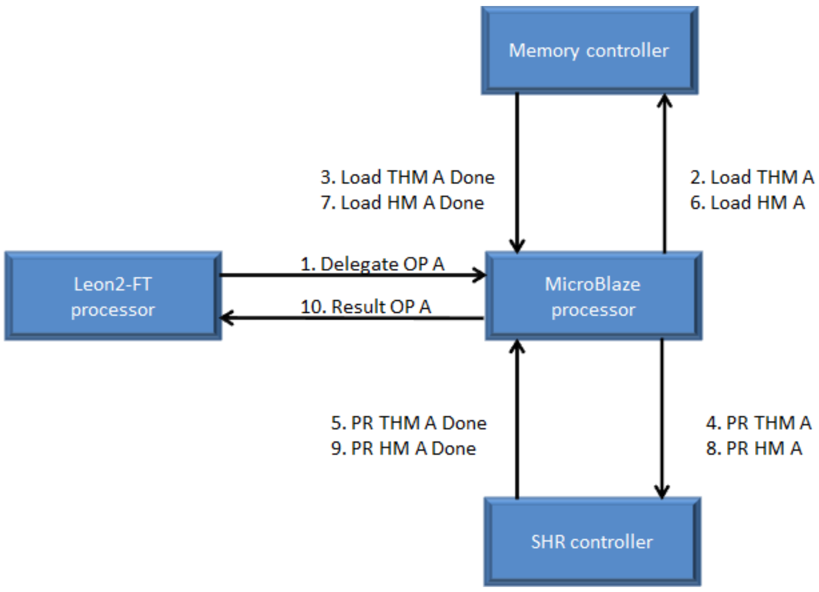


Figure 1.37: Example of interaction between Leon2-FT and MicroBlaze processors with fault detection

Chapter 2

Related Work

2.1 Introduction

FPGA testing occupies an important part in the literature pertaining to FPGAs, above all as a result of their increase of popularity and use in safety-related and space applications.
At the beginning most of the works were addressed to the Off-Line testing, that is where possible running applications are stopped and the device is then tested. This service interruption is not feasible in continuous operating contexts. Recently, in particular with the introduction of the PR, the focus was moved on performing On-Line testing, allowing running applications to continue their elaborations.
In the following sections some works related to the FPGA testing are presented. Finally, the approach object of this thesis is briefly introduced.

2.2 Logical resources testing

Works in this section are focused on logical resources testing:

1. [19], where only faults in memory elements are addressed, supposing that IOBs and interconnections have already been tested;

2. [20], where testing integration in a runtime reconfigurable system is emphasized. Logical resources are divided in containers and the logic, within the container under test, is tested through a comparison-based BIST approach, utilizing TPG and ORA placed in different containers;

3. [21], where the use of testing HMs and the constraint of using only some columns for each testing session are introduced, in particular five columns are considered:

   - one contains two TPGs;

- two contain ORAs;
- two contain Blocks Under Test (BUTs).

The approach is a comparison-based BIST, where each ORA compares two responses, which come from two different BUTs, driven by different TPGs. Finally, columns are shifted on the next testing session.

4. [22], where a comparison-based BIST is used and columns are shifted in each testing session; in particular this work is focused on Xilinx Virtex-4 devices and uses the PR for keeping the testing time low, because it mainly depends on the reconfiguration time.

2.3 Routing resources testing

Works in this section are aimed at detecting faults in routing resources.

1. [23], where an external approach is used, rather than a BIST. It exploits the JTAG interface for providing input stimuli and retrieving results;
2. [24], where a comparison-based BIST approach is used, in particular test vectors are stored in LUTs and then subsequently applied to CUTs;
3. [25], where a comparison-based BIST approach is used and it exhibits the feature of moving the part under test in every testing session for both logical and routing resources. Input stimuli for routing resources are generated as in [23] with a *Walking-0* or *Walking-1* approach;
4. [26], where the cross-coupled parity BIST and registers for storing test results are used. This work represents the starting point for the approach proposed in this thesis.

2.4 The proposed approach

All the aforementioned works feature interesting properties, but no one combines:

- **Hard macros utilization**, these pre-built blocks can be stored in the DDR2 RAM and loaded on-demand by the MicroBlaze;
- **Cross-coupled parity approach**, which outdoes limitations of the comparison-based and single parity approaches;
- **Interconnection infrastructure testing**, which, as previously said, by now occupies up to 80% of whole chip area;

- **Focus on permanent faults**, which nowadays, with really bigger devices, are gaining importance again;
- **PR and readback**, which represent the key for an On-Line testing approach as well as for keeping the testing time low;
- **On-Line testing**, which permits the use of these mechanisms even in situations where a service interruption is not feasible, such as in ACSes operating in space.

The proposed approach aims at combining all these elements, with a reasonable tradeoff between simplicity and effectiveness.
In order to realize easy placeable testing hard macros, the use of the following elements must be avoided:

- global signals, as clock and reset, therefore they should be internally generated;
- particular resources, as the CAPTURE unit, which is used when the content of registers is read through a readback, for example where results are stored in registers as in [26].

Moreover, the quantity of the used logical resources has to be kept small in respect to the total available and the approach has to be optimized for Xilinx Virtex-4 devices, the same with which the DB-V4 daughterboard is equipped.

Chapter 3

Testing Circuits

3.1 Introduction

Three different testing circuits have been developed, each has different characteristics and features a diverse granularity level in terms of number of WUT. More precisely they are:

- A version with 6WUTs, which thanks to a feedback mechanism is able to detect transient faults, as well permanents;
- A version with 8WUTs, which is a variation of the previous version, it has no longer the feedback mechanism, but is able to test eight wires at the same time;
- A version with 1WUT, which represents the higher reachable granularity level, it employs a completely new approach because of the constraint of having only a wire between TPG and ORA.

There are actually two variants for the first two circuits, one without and one with a clock cycle counter. This counter could be used to check if there is performance degradation, for instance due to aging.
As previously said, in order to ease the placement of the hard macros inside an FPGA, the planning phase of these circuits has to eliminate, and where it is not possible, to reduce:

- global signals, e.g. Clock and Reset;
- using specific resources, e.g. the CAPTURE unit;
- used logical resources.

Consequently, it has also been necessary to develop other logic, as a ring oscillator for generating a clock signal and a circuit for generating a reset signal.

The results of the ORA assessment task are stored in distributed RAM and in due time these will be read by means of a readback operation.
Before presenting the developed testing circuits in details, some useful information about TPG and ORA are given in the following section.

3.1.1 Test Pattern Generator

The function of this component is to generate some specific signals, which are propagated on the wires under test, thereby testing their status. The testing signals derive from [26], representing the starting point for 6WUTs and 8WUTs circuits. There are overall four configurations of these signals, which are shown in Table 3.1.

Num	**I**	**II**	**III**	**IV**	**V**	**VI**
1	0	0	0	1	1	1
2	0	1	1	1	0	0
3	1	0	1	0	1	0
4	1	1	0	0	0	1

Table 3.1: TPG signals configurations

In reality, the TPG is made of two internal TPGs (up and down counting TPG) and for that reason in the Table 3.1 the columns have been highlighted with different colours, in particular, each colour represents a different internal TPG.

3.1.2 Output Response Analyzer

Functions of this component are essentially two:

1. Assessing if the received signals, by means of the WUTs, are what it is expected;

2. Basing on the assessment at the point 1, providing the test results, that is whether a fault has been detected or not. Hereinafter it will be adopted the following convention: a *1* will indicate that at least one fault has been noticed, whereas a *0* that no faults have been detected.

Likewise at the TPG, the ORA is made of two internal ORAs.
The four expected configurations are shown below in Table 3.2, where each colour represents the group of signals which is assessed by the same internal ORA.
It is useful to note that columns **III** and **IV** do not belong to the same groups any longer, as in Table 3.1, but their groups have been exchanged, thence the name of this approach: *cross-coupled* parity. The assessment process is very simple, that is:

Num	I	II	III	IV	V	VI
1	0	0	0	1	1	1
2	0	1	1	1	0	0
3	1	0	1	0	1	0
4	1	1	0	0	0	1

Table 3.2: ORA expected configurations

- In the event that one of these configurations has been received, both the internal ORAs will give '0' as output;
- In the event that one configuration, which is not in Table 3.2, has been received, then both internal ORAs will give '1' as output;
- In the case that one configuration, which is partially in Table 3.2, has been received, one internal ORA will give '0' as output and the other '1', this latter is the ORA which is related to the missing part of the configuration in Table 3.2. If, for instance, the received configuration is "000101", by observing the first row of the Table 3.2, it is possible noticing that part of the configuration is present for the blue group, but not for the green one, therefore for the aforementioned reasons, the *blue* internal ORA will give '0' and the *green* one '1' as output.

3.2 6WUTs and 8WUTs testing circuits

Figure 3.1 shows the high-level abstraction scheme of the 6WUTs testing circuit, while the complete scheme with internal components is reported in Figure 3.2. Tables 3.1 and 3.2 above represent the 6WUTs TPG and the 6WUTs ORA signal configurations.
Outwardly the differences between 6WUTs and 8WUTs are fundamentally about the number of links between TPG and ORA. Tables 3.1 and 3.2 change to Tables 3.3 and 3.2, respectively.

Num	I	II	III	IV	V	VI	VII	VIII
1	0	0	0	1	1	1	1	0
2	0	1	1	0	1	0	0	1
3	1	0	1	1	0	1	0	0
4	1	1	0	0	0	0	1	1

Table 3.3: 8WTUs TPG signals configurations

Num	**I**	**II**	**III**	**IV**	**V**	**VI**	**VII**	**VIII**
1	0	0	0	1	1	1	1	0
2	0	1	1	0	1	0	0	1
3	1	0	1	1	0	1	0	0
4	1	1	0	0	0	0	1	1

Table 3.4: 8WTUs ORA expected configurations

The operating principles of both TPG and ORA are equal to those of the 6WUTs version.

3.2.1 6WUTs and 8WUTs planning

In both 8WUTs and 6WUTs versions there are two FSMs (**F**inite **S**tate **M**achines), which behave as coordinator amongst units and generate some important signals, needed for:

- starting the write of results in distributed RAM;
- requesting the next input to the TPG.

Moreover, in order to improve the following diagnostic phase, it has been chosen to use some signals, which are generated by the TPG, to address the distributed RAM, in particular columns **I**, **II**, **V** e **VI** of the Table 3.3.

By observing the first group of the Table 3.1, that is columns **I**, **II** and **III**, it is easy to note that the first two columns represent a mere 2-bit counter, which starts from 00_b up to 11_b, whereas the latter column is the parity, in particular the even parity, inasmuch as it is be '0' when the number of '1' in columns **I** and **II** is even. For the aforementioned reasons:

- signals relating to the columns **I** and **II** are called **Cu1** and **Cu0**, respectively. Indeed, **Cu** derives from **C**ounter-**U**p.
- the signal relating to the column **III** is called **PEven**, hence the internal TPG is called **Even Parity TPG**.

On the other hand, by observing the second group in Table 3.1, it could be noted that in this case the columns **IV** and **V** also represent a 2-bit counter, but now it starts from 11_b and reaches 00_b, that is it is a **C**ounter-**D**own, therefore these signals are called **Cd1** and **Cd2**, respectively. Regarding to the column **VI**, it also represents a parity, which is now odd, in other words it will be '0' when the number of '1' in the columns **IV** e **V** is odd, hence the name **POdd** for the signal and **Odd Parity TPG** for the internal TPG. Concerning the internal ORAs, their names derive from the parity type, that is:

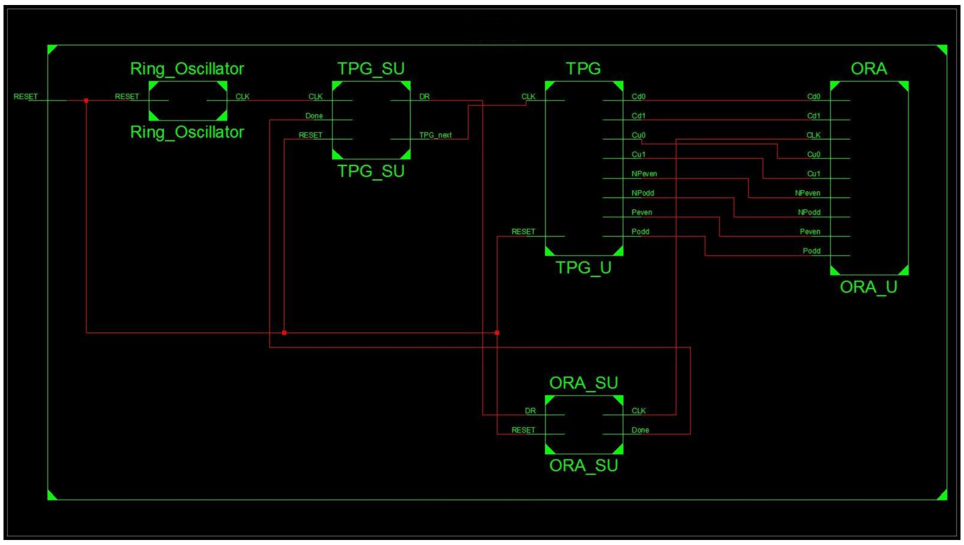


Figure 3.1: 6WUTs RTL complete scheme

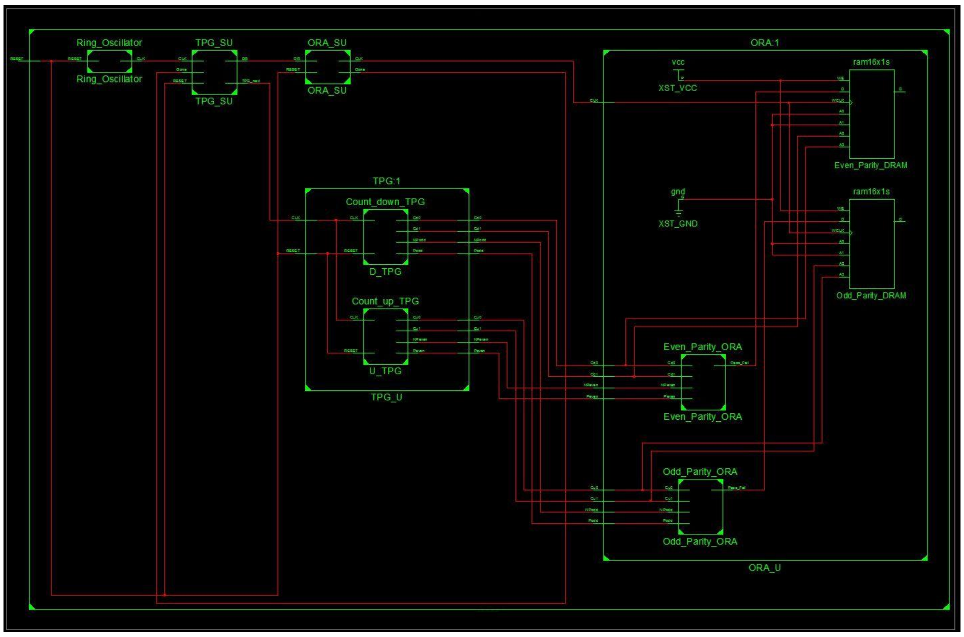


Figure 3.2: 6WUTs RTL complete scheme with internal component

- the internal ORA which uses PEven in its assessment is called **Even Parity ORA**, in particular it uses the configurations of the second group in Table 3.2;
- the internal ORA which uses POdd in its assessment is called **Odd Parity ORA**, in particular it uses the configurations of the first group

in Table 3.2.

The 8WUTs version differs from the 6WUTs version for the presence of two further wires, related to the columns **IV** and **VIII** of Table 3.3 and 3.4, which are called **NPEven** and **NPOdd**, respectively. The names of the internal ORAs, TPGs and the other six wires are unchanged.

3.2.2 6WUTs implementation

The implementation of this version is based on [26].

Components implementation

As it is possible to note in Figure 3.3 and 3.4 the internal TPGs are two mere counters, in which the parity is represented by the next state of CD1 or CU1, depending on which type of internal TPG is considered.

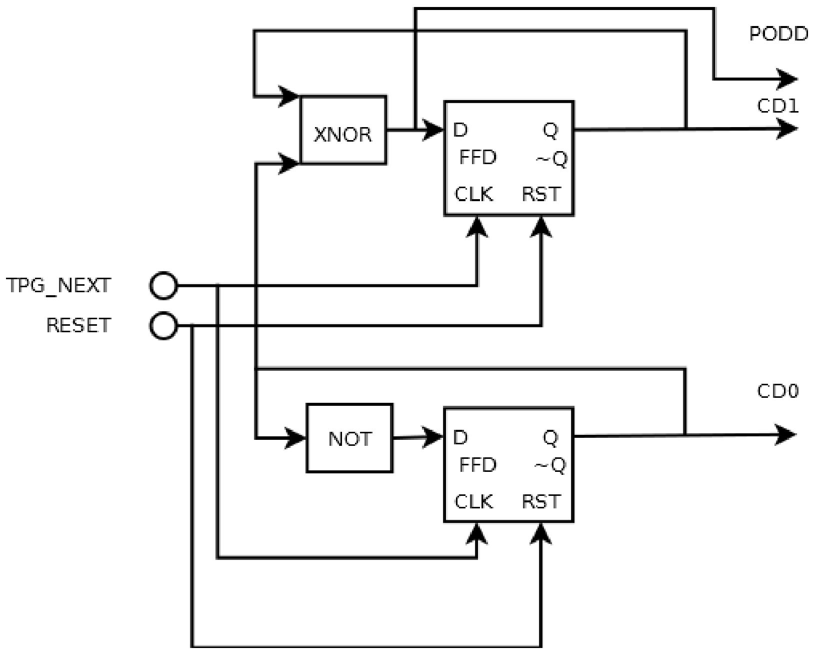


Figure 3.3: 6WUTs ODD TPG

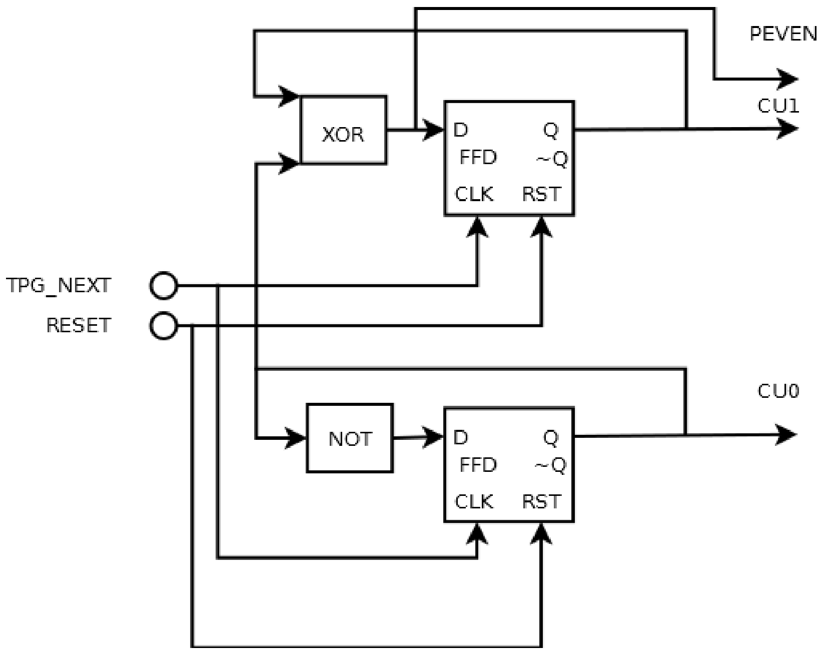


Figure 3.4: 6WUTs EVEN TPG

In internal ORAs the assessment is performed by means of:

- XNOR gate in the ODD Parity ORA (Figure 3.5);
- XOR gate in the EVEN Parity ORA (Figure 3.6).

It is possible to note that there is a feedback wire, whose task is to keep track of possible faults detected in previous configurations, transitory faults included.

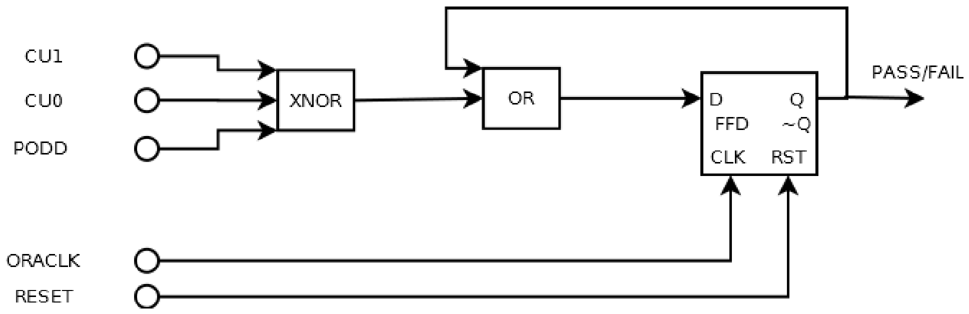


Figure 3.5: ODD parity ORA

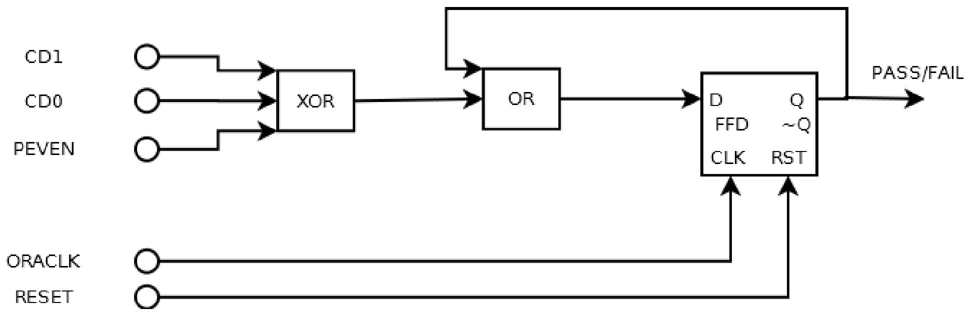


Figure 3.6: EVEN parity ORA

Complete scheme

In Figure 3.7 a 6WUTs complete scheme is shown. Its operating principle is as follows:

1. The RESET signal permits to initialize the system:
    (a) FSMs in initial state;
    (b) The first input configuration is generated upon the wires under test as consequence of the initialize process.
2. When RESET signal is low, the FSM TPG_SU becomes sensitive to the clock signal, which is generated by a ring oscillator;
3. The TPG_SU signals to the ORA the presence of valid data upon the wires under test;
4. The ORA_SU FSM launches the writing procedure of results in distributed RAM through ORACLK and signals the availability to receive the next input configuration by means of DONE;
5. The TPG_SU requests the next input configuration to the TPG by means of TPG_NEXT and all is repeated from point 3.

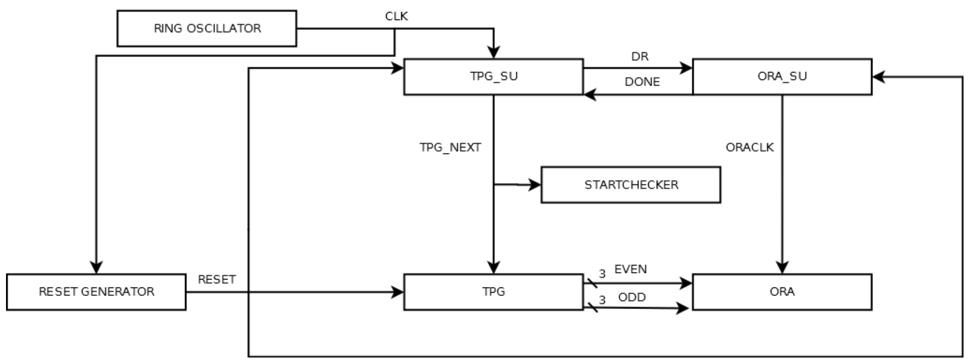


Figure 3.7: 6WUTs complete scheme

3.2.3 8WUTs implementation

As previously mentioned the only external difference with the 6WUTs is the presence of two further wires between TPG and ORA, that is **NPOdd** and **NPEven**.

Components implementation

Figure 3.8 and 3.9 depict the two internal TPGs.

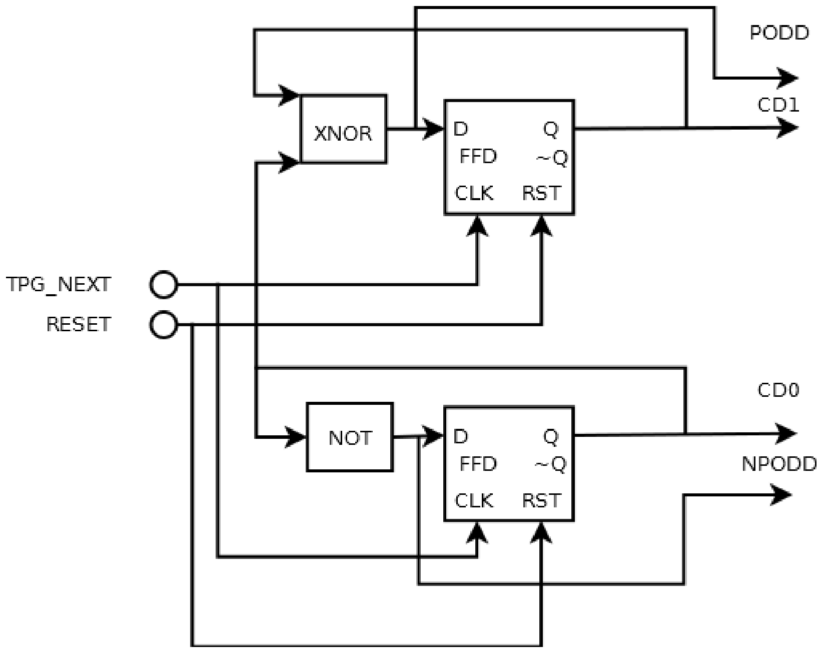


Figure 3.8: 8WUTs ODD TPG

It forthwith stands out the likeness with the other version that has been previously presented, whereas the structure of ORA (Figure 3.10 and 3.11)

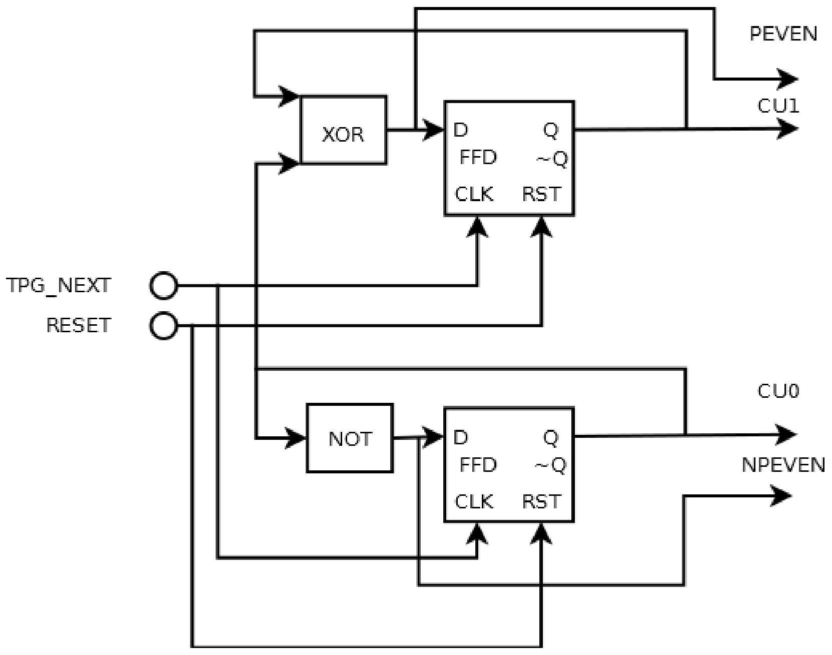


Figure 3.9: 8WUTs EVEN TPG

is to some extent different. The feedback has been removed and two further wires can be put under test using the two released inputs, these additional wires are **NPOdd** and **NPEven**.
As regards to the assessment logic in the internal ORAs, it has been obtained through a synthesis process, in which the Karnaugh′s maps have been employed. It is reported in appendix A.

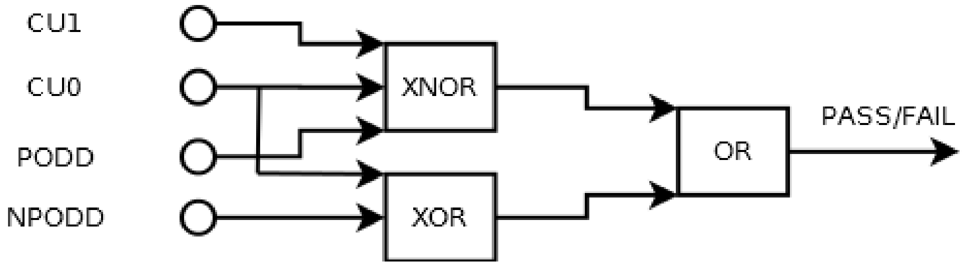


Figure 3.10: 8WUTs ODD parity ORA

Complete scheme

In Figure 3.12 a 8WUTs complete scheme is shown. Its operating principle is equal to that of the 6WUTs version.

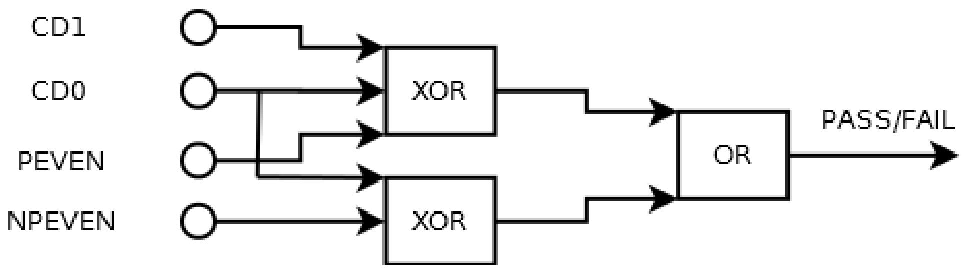


Figure 3.11: 8WUTs EVEN parity ORA

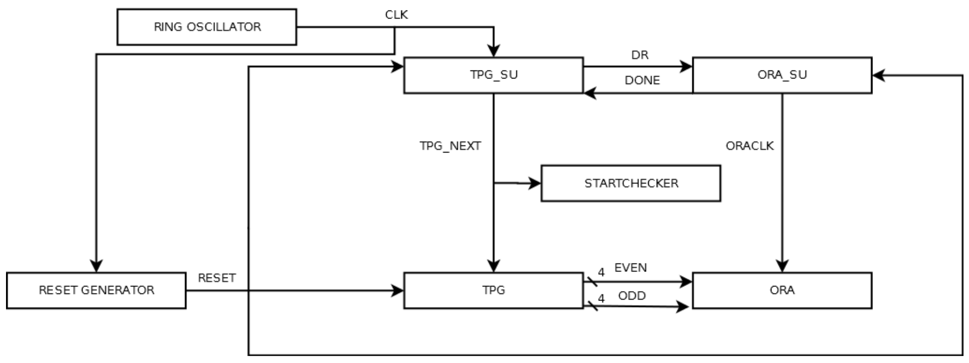


Figure 3.12: 8WUTs complete scheme

3.2.4 Auxiliary logic

In this section all the supporting logic which is used by both 6WUTs and 8WUTs is presented:

- Ring oscillator;
- TPG_SU and ORA_SU;
- Reset generator circuit;
- Startchecker.

Ring Oscillator

It is briefly shown here a ring oscillator, which generates a clock signal used by other units. The adopted structure is exhibited in Figure 3.13.

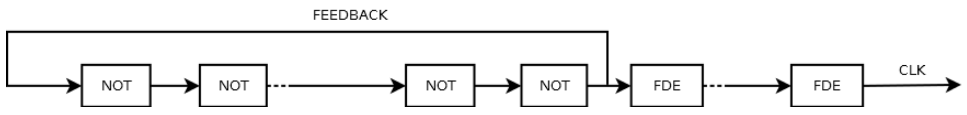


Figure 3.13: Ring oscillator

The **FDE** (Figure 3.14) or **F**requency **D**ivider **E**lement divides the frequency of the input signal by 2.

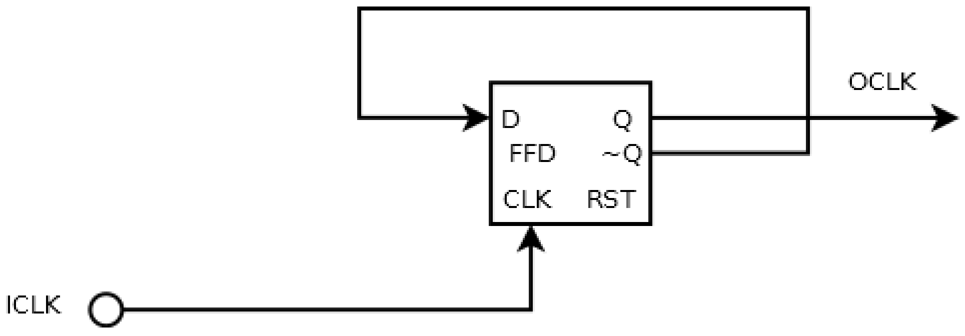


Figure 3.14: Frequency divider element

It is useful to note that the circuit has been developed using the VHDL *generate* construct, therefore there is the possibility of choosing the number of NOT gates or FDEs for influencing the frequency of the generated signal.

TPG_SU and ORA_SU

It follows a presentation of the two FSMs: TPG_SU and ORA_SU.
As previously mentioned, they are essentially coordinators amongst units and also generate some important signals. In particular the ORA_SU has been conceived to be used also with "smarter" device (devices that signal when results have been written) with some little changes, like introducing a further signal to indicate when the write was effectively done.
Figure 3.15 shows the state transition diagram of TPG_SU, where the two output signals are, respectively:

- TPG_NEXT, for requesting the next input configuration to the TPG;
- DR, that is Data Ready, for signalling that the configuration, upon the wires under test, is valid.

Conversely, in Figure 3.16 the state transition diagram of the ORA_SU is reported, where the output signals are respectively:

- ORA_CLK, to starting the write in distributed RAM;
- DONE, for signalling the availability to receive the next input configuration. In particular this signal becomes significant in presence of smarter devices or when some particular timing constraints are present, such as *t-hold*; in the case of distributed RAM this constraint is not present.

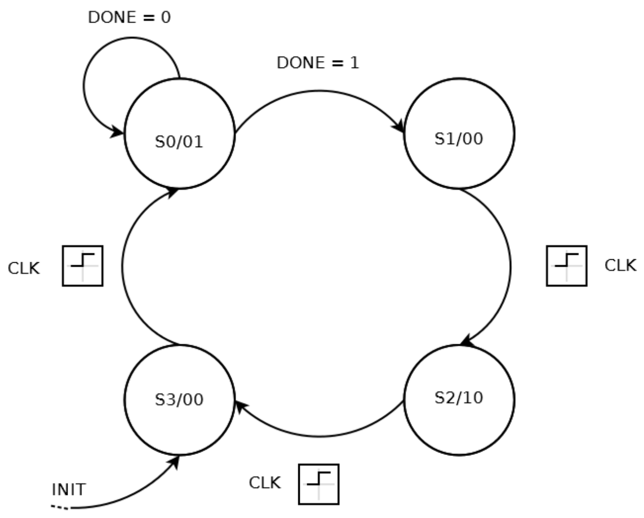


Figure 3.15: TPG_SU state transition diagram

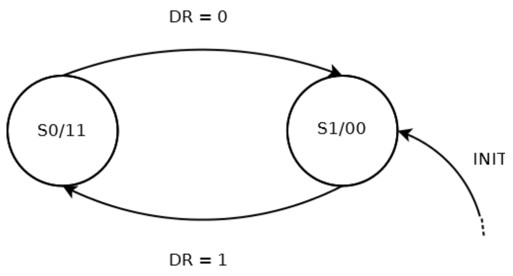


Figure 3.16: ORA_SU state transition diagram

In Figure 3.17 an example of simulation of the FSMs with Modelsim [27] is exhibited. The simulation is not behavioural, but post place and route, in other words delays of both logic and wires propagation are considered.

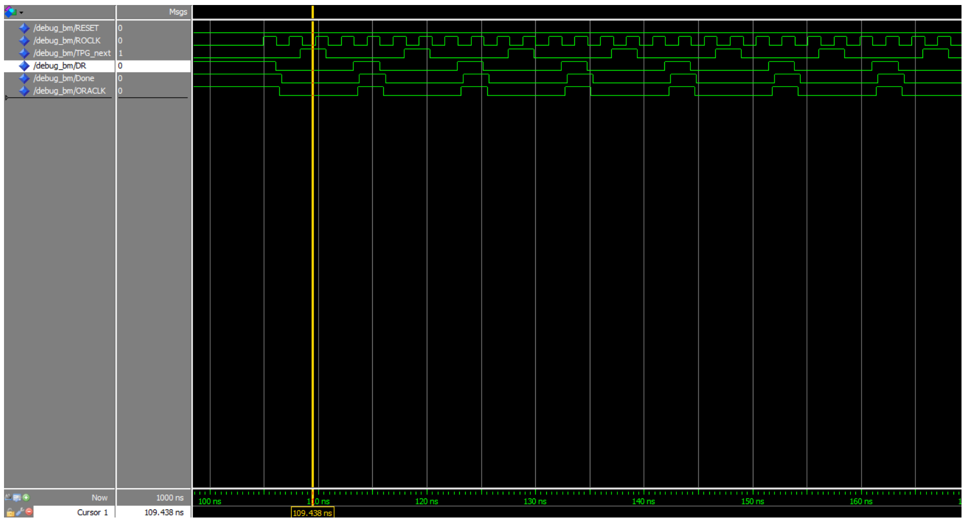


Figure 3.17: FSMs ModelSim simulation

Reset generator circuit

It is now briefly presented the circuit which generates the RESET signal (Figure 3.18).
It fundamentally consists of a shift register, 32 bits in this case, and its operating principle is the following:

- In the initial state its state is 1111111111111111_b;
- On every clock cycle:
  - LSB (Least Significant Bit) is propagated upon the output wire;
  - The state is shifted toward right of one position;
  - It is added 0 as MSB (Most Significant Bit).
- It is obtained a reset signal, which is high for 16 clock cycles.

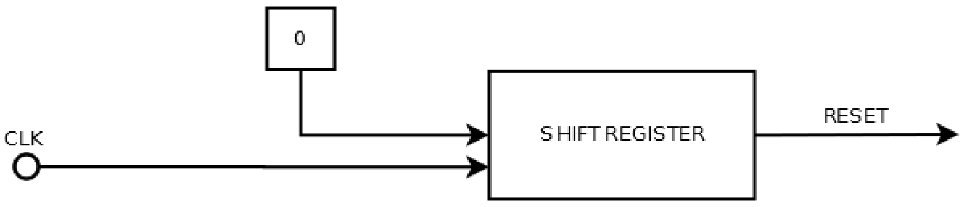


Figure 3.18: Reset generator circuit

Startchecker

The purpose of this component is to allow checking whether the test has been started or not from external.
As consequence, now there are three results and their possible combinations are reported in Table 3.5.

Startchecker DRAM	**ODD parity ORA DRAM**	**EVEN parity ORA DRAM**	**Description**
1	-	-	Test has not been started
0	0	0	Test has been started and no fault has been detected
0	1	0	Test has been started and it has been detected at least a fault in the Odd Parity ORA
0	0	1	Test has been started and it has been detected at least a fault in the Even Parity ORA
0	1	1	Test has been started and it has been detected at least a fault in both Even Parity ORA and Odd parity ORA

Table 3.5: Result configurations

A possible implementation is exhibited in Figure 3.19 and its operating principle is the following:

- DRAM is initialized to ‘1‘, that is the test has not been started;
- When TPG_NEXT requests a new input configuration to the TPG, the AND gate′s inputs are ‘1‘ and this enables the writing of ‘0‘ in distributed RAM, in other words the test has been started;
- When the writing has been accomplished, the distributed RAM′s output is ‘0‘ and the AND gate becomes insensitive to further changes of TPG_NEXT.

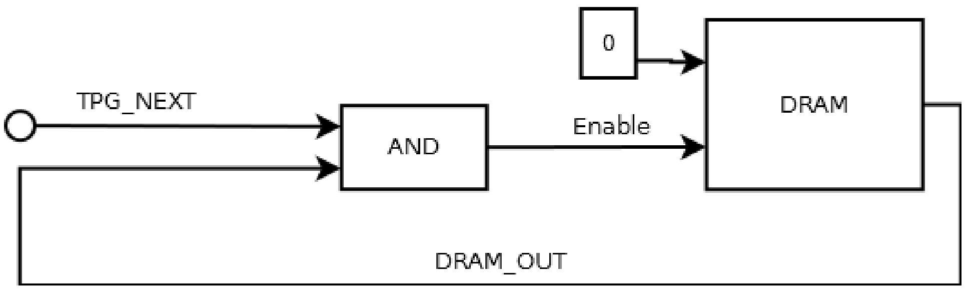


Figure 3.19: Startchecker implementation

Distributed RAM addressing (Only 8WUTs)

It is useful to mention that some TPG outputs, in particular columns **I**, **II**, **V** and **VI** of Table 3.3, are used for addressing the distributed RAM, this could permit to obtain some useful information for the diagnostic phase. Moreover, considering that distributed RAMs are initialized to $FFFF_{16}$, in the very rare case of a stuck-at fault of all the WUTs at a configuration which is present in Table 3.4, the fault is anyway detected, because the configuration in the distributed RAM is different from the expected one (four '0').

3.3 1WUT planning

In this case the design of the testing hard macro has been lead by the requirement that there must be only a WUT between TPG and ORA. It is naturally the wire subjected to the test.

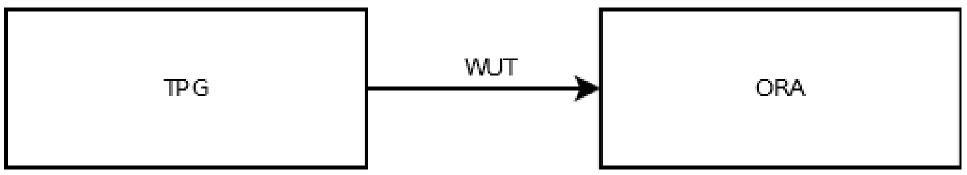


Figure 3.20: 1WUT generale scheme

A general scheme is exhibited in Figure 3.20. This approach is to a certain extent new. The first issue is how to realize the TPG and the ORA. A first scheme is shown in Figure 3.21, where the sequence "0110" has been selected. Such sequence has been chosen due to the fact that it includes both high-to-low and low-to-high transitions. In addition it is very easy to realize in hardware, using a 2-bit counter and a XOR gate.
One challenge in the implementation of the new ORA has been how to make it able to discriminate amongst consecutive inputs. In particular problems

can occur in two cases:

- in the same sequence, there are two consecutive ‘1‘;
- between consecutive sequences (i-th and i+1-th), there are two consecutive ‘0‘: the last ‘0‘ of the i-th sequence and the first ‘0‘ of the i+1-th one.

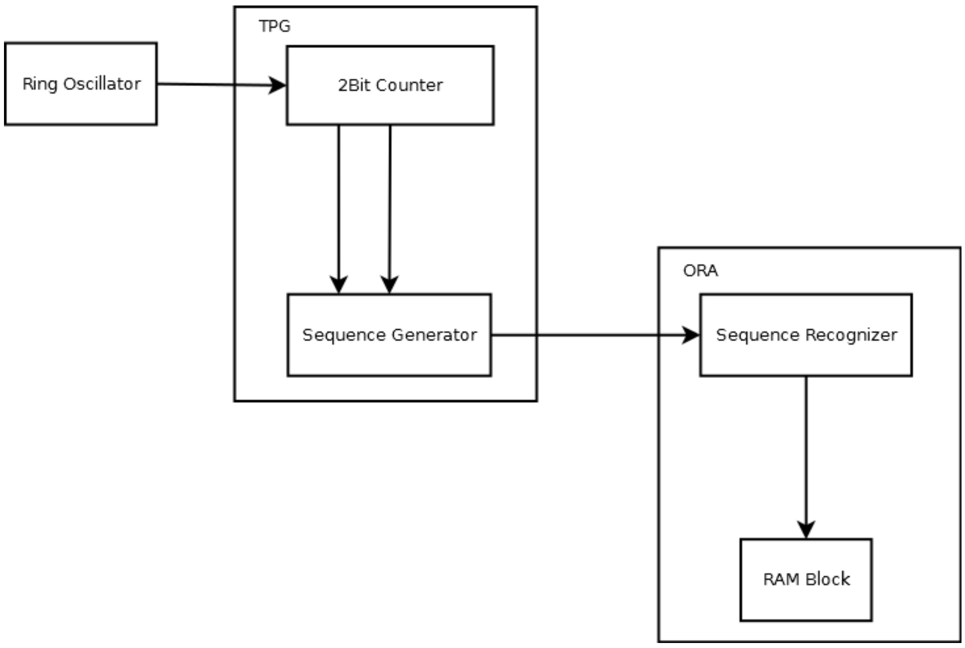


Figure 3.21: 1WUT first scheme

A possible solution would have been to encode signals with the Manchester code [28]. The aforesaid problem would have not been present, for the reason that in this coding there is a transition for each sent bit.
Figure 3.22 shows how the scheme would have appeared using the Manchester encoding. A Manchester encoder is quite easy to realize and it could be expressed through the following formula:

$$\textit{Encoded Sequence}=\textit{CLK xor Sequence}$$

To be more accurate, the latter formula is valid only if it is adopted one of the following standards:

- **IEEE 802.4**;
- **IEEE 802.3**.

On the other hand, by using the G.E. Thomas convention, which was published in 1949, the formula is not valid any more.

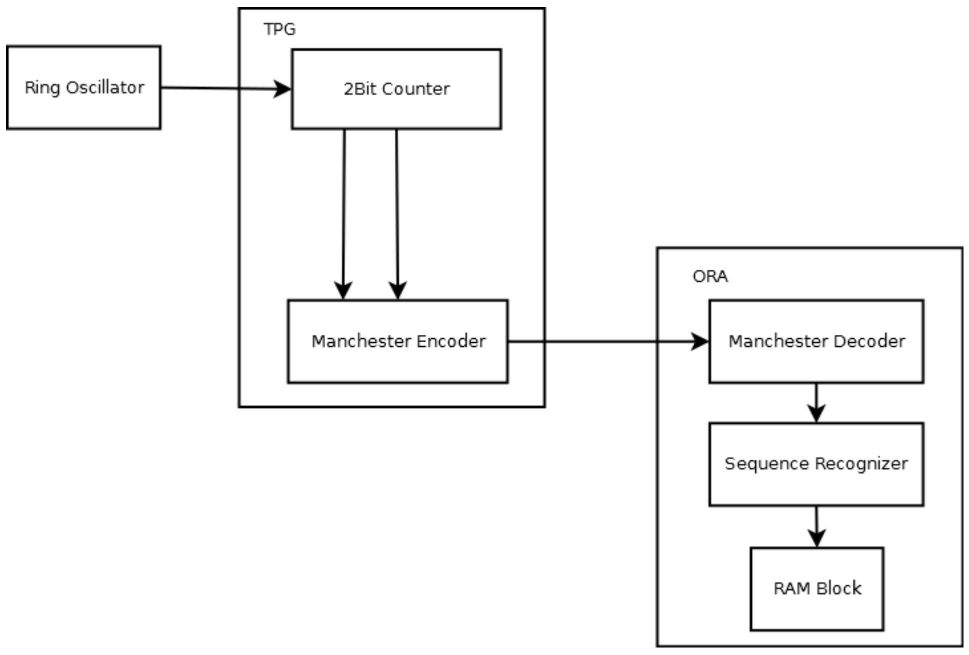


Figure 3.22: 1WUT with Manchester encoding

In particular, in both IEEE standards, a low-to-high transition represents a 1 and a high-to-low transition represents a 0.
Regarding the decoder, which is far more complex than the encoder, there are two possibilities:

- Using a particular counting algorithm [29], which exploits a clock signal with a frequency 8-16 times higher than the clock signal used by the encoder, in order to sample transitions for each sent bit. In the event of using clock signals generated by accurate and controllable oscillators, this choice would probably be the one which uses less logic;
- Using a **PLL** (**P**hase **L**ock **L**oop), in order to extract the clock signal from the received signal and afterwards, by exploiting it, to obtain the decoded sequence.

Unfortunately, in both choices a clock synchronisation is required, which could be obtained by means of using a preamble and a shift register.

Some more problems occur in the implementation of the decoder using the Manchester encoding.
In the first choice, two oscillators are required, which should be quite accurate and controllable. Ring oscillators are not well suited in this case, because the clock signal frequency will be given by propagation delays, which will change with respect to the chosen routing, within FPGAs context. Conversely, in the second choice, the amount of required logic is absolutely ex-

cessive. Furthermore, in both choices it is needed to consider logic for clock synchronization as well. Since the goal of the test circuit is the testing of a single wire, this might burst the limits of the testing circuit.
For the aforementioned reasons, a new approach needs to be developed.

3.4 1WUT “CLK Approach“ planning

The idea behind this approach is quite simple: with the requirement that upon the wire both types of transition must be present, that is high-to-low and low-to-high, a *clock signal* could be used.
In this way there are no sequences any more (with the same meaning of the previous approach) and the sequence recognizer is no longer needed.
A new issue arises: how the ORA can detect the accomplishment of a new test.
The ultimately idea is to send a finite number of clock cycles, $2^n - 1$. A high level scheme is exhibited in Figure 3.23.

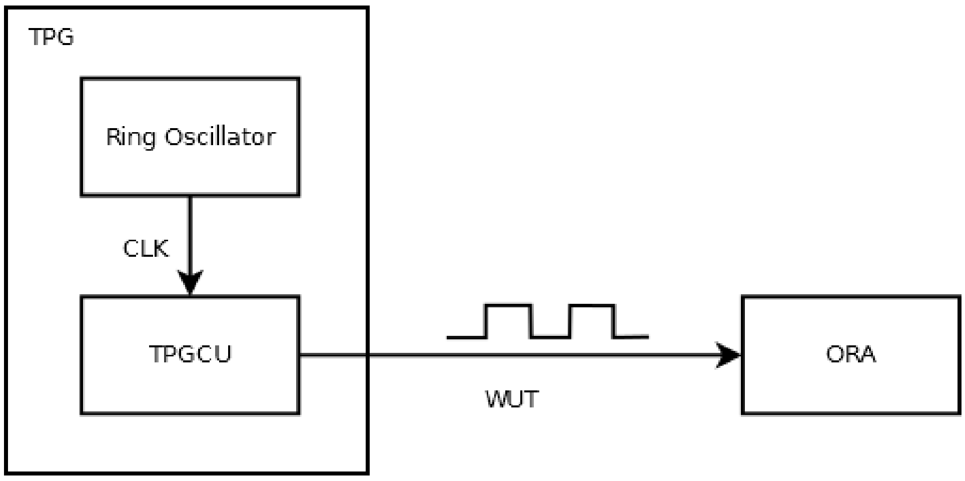


Figure 3.23: 1WUT high level “CLK Approach“ scheme

3.4.1 Implementation

In this section the circuit implementation will be described in terms of its function and structure:

- TPG;
- ORA.

TPG

This component is made of two main parts:

- the usual ring oscillator which generates a clock signal;
- **TPGCU** (**TPG** **C**ontrol **U**nit) FSM which disables clock propagation upon the WUT when $2^n - 1$ cycles are reached, where n is the size in terms of bits of an internal counter.

Figure 3.24 provides a scheme of this component.

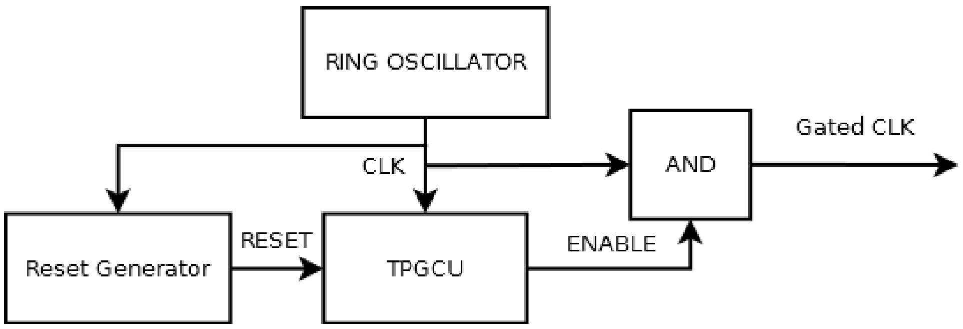


Figure 3.24: 1WUT “CLK Approach“ TPG

In Figure 3.25 the state transition diagram of **TPGCU** is reported, where the only output is the ENABLE signal.

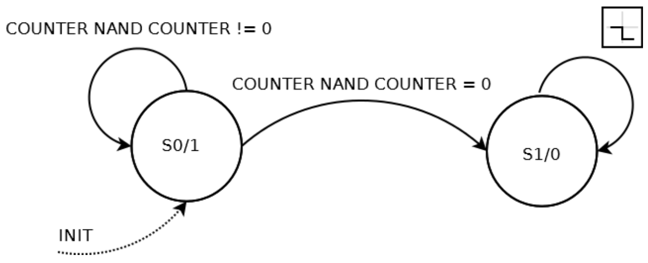


Figure 3.25: 1WUT TPGCU state transition diagram

It is worth noting that transitions occur on high-to-low edge of the clock signal, the reason is to avoid that the last cycle′s high part will be shorter than the others.

Startchecker

This component has been foreseen also for 1WUT version and in this case there are only two results, whose possible configurations are reported in Table 3.6. The scheme of this component is shown in Figure 3.26 and also in this case its operating principle is rather simple, which is as follows:

- The distributed RAM is initialized to ‘1‘, that is the test has not been started;

Startchecker DRAM	**ORA DRAM**	**Description**
1	-	Test has not been started
0	0	Test has been started and not fault has been detected
0	1	Test has been started and it has been detected at least a fault

Table 3.6: 1WUT startchecker results

- By the time in which ENABLE signal changes from ‘1‘ to ‘0‘, the NOT gate′s output follows this transition allowing a writing of ‘0‘ in the distributed RAM, that is the test has been started.

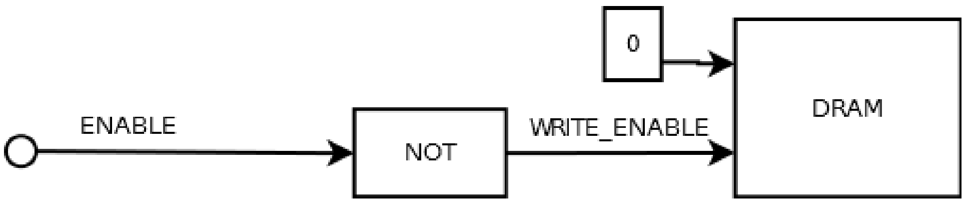


Figure 3.26: 1WUT startchecker scheme

ORA

The ORA represents the central part of the circuit, as well as the more complex.
A possible representation is shown in Figure 3.27.

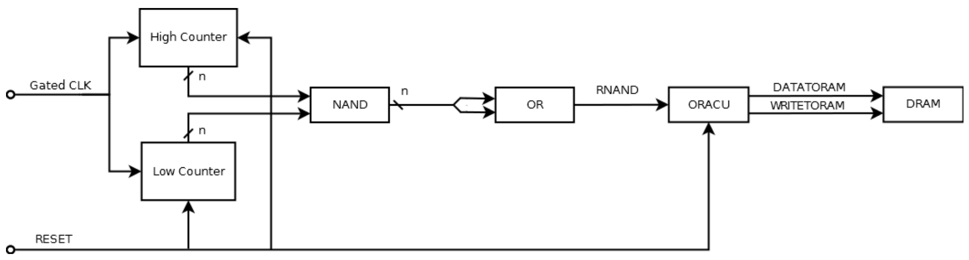


Figure 3.27: 1WUT “CLK Approach“ ORA

The High and Low counters keep track of how many high and low clock edges, respectively, are sent upon the WUT. Their size in bit is n.
In the absence of faults both counters will reach a configuration with n bits equal to ‘1‘, therefore the NAND gate′s output will be a configuration with n bits equal to ‘0‘ and finally the OR gate′s output will be ‘0‘.

By that time the **ORACU** FSM will begin the writing of ‘0‘ in distributed RAM, indicating that no faults have been detected.
In the case of fault, there are two possible scenarios, depending on the fault type:

- stuck-at ‘0‘ or ‘1‘ or even in the event that due to one or more shorts the ORA will receive less clock cycles than $2^n - 1$. In this case the OR gate′s output stays to ‘1‘ and the ORACU will not enable the writing to the distributed RAM, where the initialization value is ‘1‘, that is a fault has been detected;
- one or more shorts generate one or more additionally clock cycles, the counters will resume from a configuration with n bits equal to ‘0‘. In this particular case the ORACU will enable the writing of ‘0‘ in distributed RAM, since it is not able to predict the future, but as soon as the OR gate′s output becomes ‘1‘ again, in other words after the 2^n-th cycle, the FSM will command a write of ‘1‘ in the distributed RAM and it will become insensitive to further changes of the OR gate′s output.

ORACU

The ORACU block contains the FSM, and besides other related logic (see Figure 3.28).

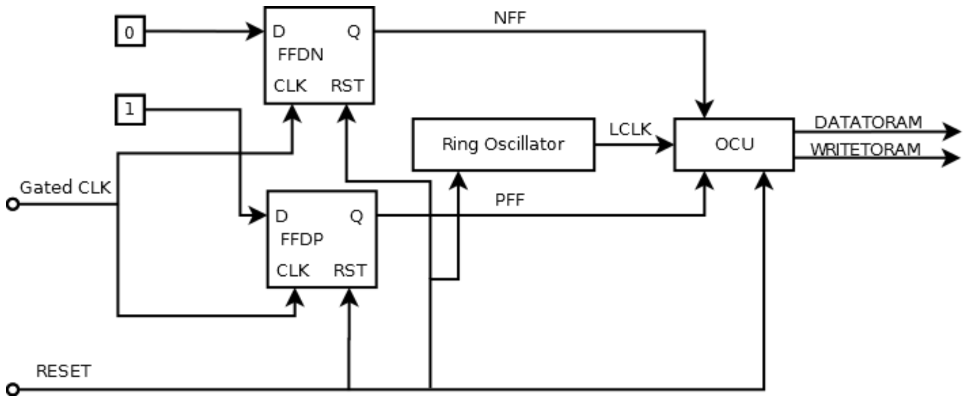


Figure 3.28: 1WUT “CLK Approach“ ORACU

FFDN and **FFDP** are FF D-negative-edge-triggered and D-positive-edge-triggered, they are initialized to ‘0‘ and ‘1‘, respectively. These components have two aims:

- to uncouple clocks, making them independent from each other, therefore there are no constraints about the relative frequency speed between TPG and ORA clock signals, thus it is no longer needed that the ORA′s clock signal is faster than the TPG′s one;

- to sample and hold signals for a reasonable amount of time, in order to permit to the FSM to determine whether the OR gate′s output has been equal to ‘0‘ previously and afterwards is returned to ‘1‘ or not. Indeed, the amount of time, in which that output is ‘0‘, could be relatively short and it could happen that the FSM is not able to notice this change.

Figure 3.29 depicts the state transition diagram of **OCU** (**ORA C**ontrol **U**nit) FSM, where the outputs are DATATORAM and WRITETORAM, respectively.

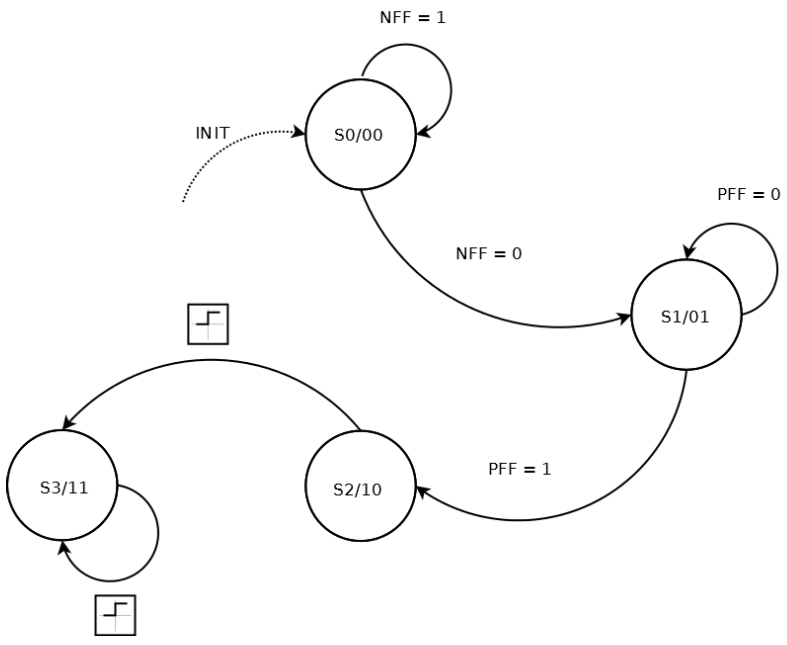


Figure 3.29: 1WUT OCU state transitions diagram

Chapter 4

Testing Circuits Verification

4.1 Introduction

In order to assess how the developed testing circuits behave in presence of faults a verification phase has been foreseen, which does not include only faults along the wires under test, but in any PIP of the circuit.
Goal of this section is to show how the verification is performed, in particular how faults are injected in the device in order to validate the testing circuits, and what are the results.

4.2 Faulty Bitstreams Creation

Faults are injected through loading bitstreams which piggyback faults, hereinafter called faulty bitstreams. The idea, on which this process is based on, is to start with an XDL file, one for each testing circuit, which is the description of the hard macro where information about the used PIPs are included. At this point removing PIP by PIP it is generated a new XDL file for each existing PIP. Each generated file has a missing PIP, which represents the piggybacked fault. At the end, from each generated XDL file, a faulty bitstream is obtained.
Figure 4.1 depicts an HM portion where there are all the PIPs, while Figure 4.2 depicts the same portion in which a PIP has been removed.

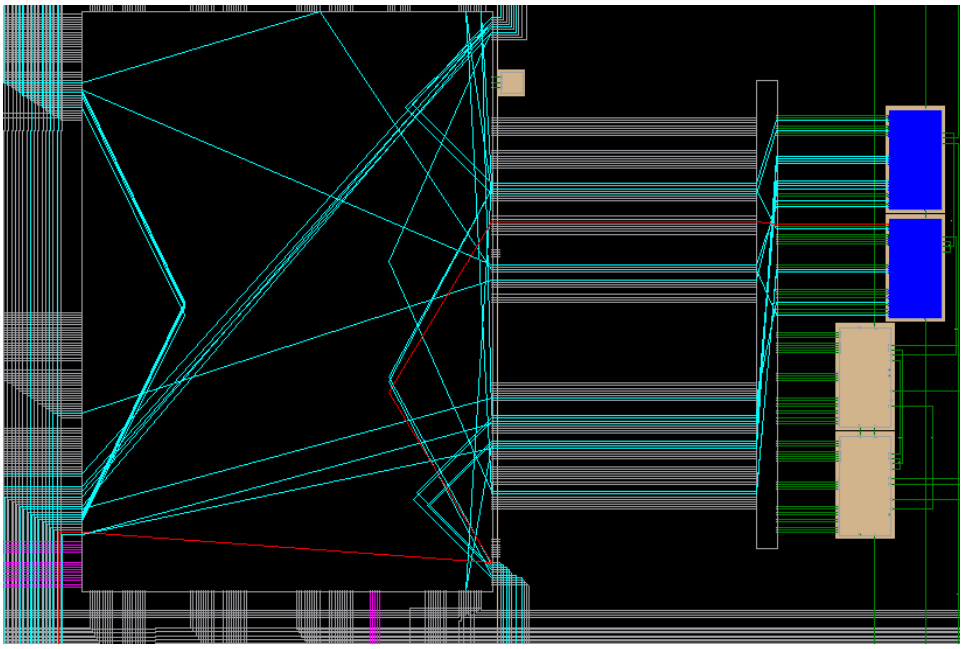


Figure 4.1: No faulty bitstream

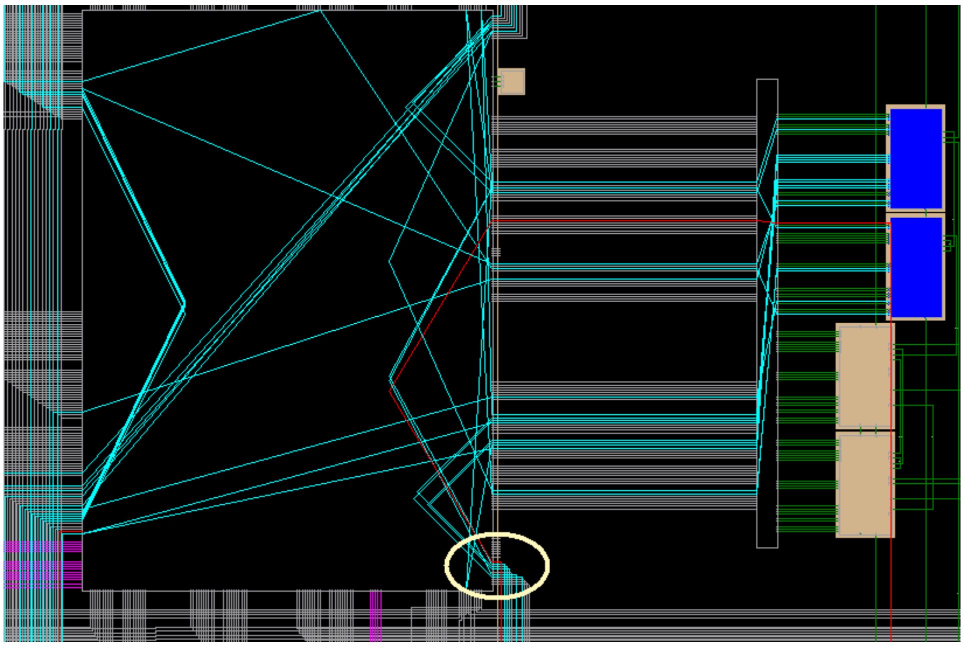


Figure 4.2: Faulty bitstream with a missing PIP

The flow which creates faulty bitstreams starting from an XDL file is automatic and it is shown in Figure 4.3. In particular:

- The **starting XDL** is obtained by means of DHHarMa, that allows to set constrains on both placing and routing of the testing circuits; although it is possible using other tools provided by the vendor;
- **Create faulty XDL files** consists in a C program, which parses the starting XDL file and creates a new faulty XDL file for each PIP;
- **Create faulty HMs** consists in a bash script, which using the *XDL* command, provided by Xilinx, translates faulty XDL files in HMs;
- **Create faulty bitstreams** consists in a script for Xilinx FPGA Editor, which generates the faulty bitstreams by means of the command *bitgen*.

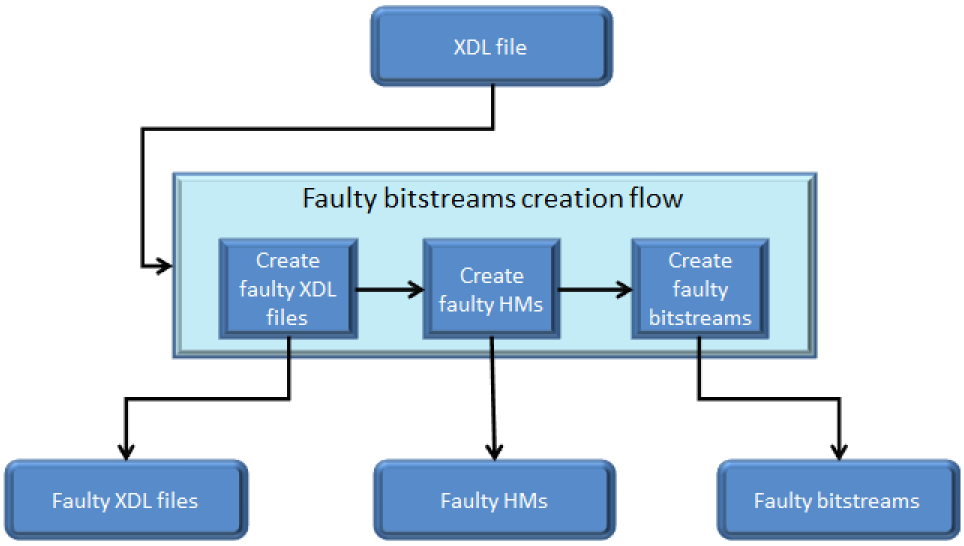


Figure 4.3: Faulty bitstreams creation flow

4.3 Faulty Bitstreams Injection

In order to:

- move faulty bitstreams from the host PC into the DDR2 RAM, which is on the DB-V4 daughterboard;
- perform the PR, for each stored faulty bitstream, through the SHR controller in the PR-FPGA module;

a synchronization protocol, which automatizes the injection phase, has been developed. The PR-FPGA module is implemented using Xilinx EDK Design tools, that is an integrated environment of developing that includes

a GNU compiler and a debugger for C/C++ software, whereby it is possible to set the MicroBlaze processor behaviour.
The synchronization protocol between Host PC and MicroBlaze is based on messages exchange. The injection takes place in two distinct steps:

1. **Loading**: as Figure 4.4 depicts, a faulty bitstream is loaded into the memory, in particular through the communication bridge between local bus and Processor Local Bus (PLB), then the memory controller is in charge of completing the loading;

2. **Reconfiguration**: as Figure 4.5 depicts, a faulty bitstream is taken from the DDR2 RAM and then, through a local link, it is moved to the SHR controller, which performs the PR by means of the ICAP interface.

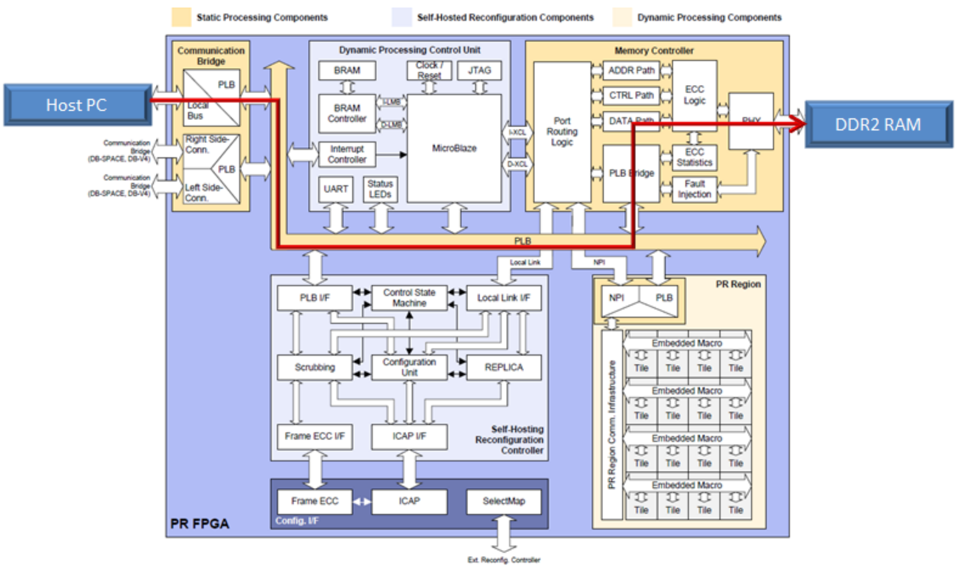


Figure 4.4: Faulty bitstream loading phase

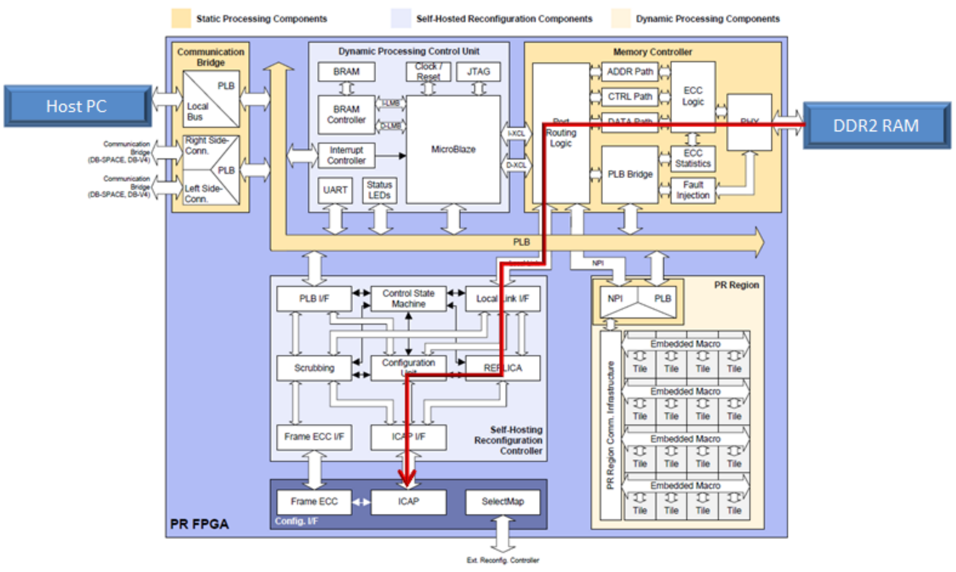

Static Processing Components
Self-Hosted Reconfiguration Components
Dynamic Processing Components
Host PC
DDR2 RAM
Communication Bridge
Dynamic Processing Control Unit
Memory Controller
MicroBlaze
PLB
PR Region
Self-Hosting Reconfiguration Controller
ICAP
PR FPGA
Config. I/F
Ext. Reconfig. Controller

Figure 4.5: Faulty bitstream PR phase

4.4 Verification Results

The testing results are stored in LUTs, operating as distributed RAM and thence in slices of type M. They are collected exploiting the operation of readback and a synchronization protocol between Host PC and MicroBlaze. Figure 4.6 depicts the whole PR-FPGA module and highlights a placed testing hard macro.

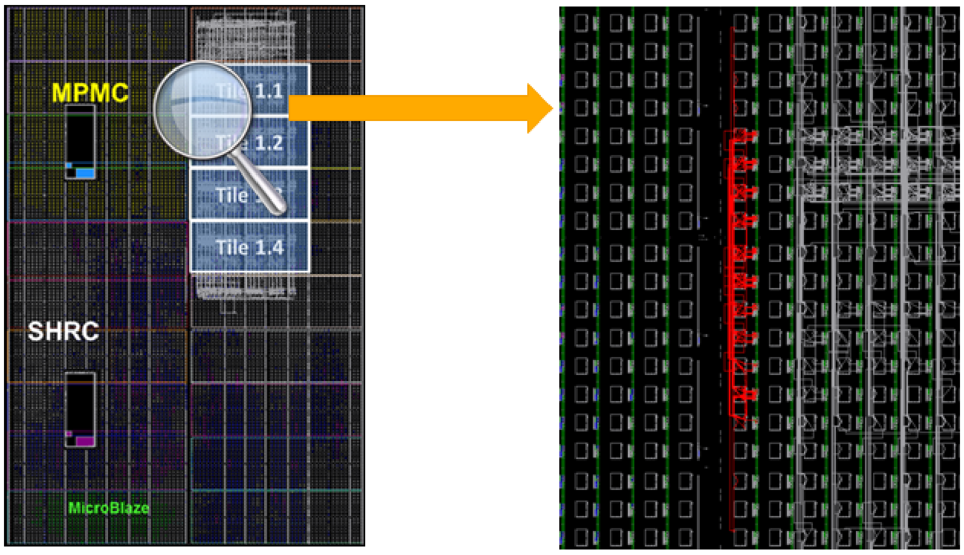


Figure 4.6: PR-FPGA module and a placed testing HM

Results include information about:

- **fault detection**, that is whether a fault is detected or not in that particular test;
- **testing start**, that is whether that particular test is able to start or not. This is useful because some faults can impede the starting of a test, in fact it is enough to think about a fault in the ring oscillator, which generates the clock signal used in the circuit.

With two possible results four combinations can be obtained:

- **Not started and fault detected**, when a fault is detected, but the circuit is not able to start; it is useful to remember that the default value in the distributed RAM is set to fault detected, therefore if a fault impedes the test starting and affects the assessment logic, that value does not change;
- **Not started and no fault detected**, when a fault is not detected and the circuit is not able to start;
- **Started and fault detected**, when a fault is detected and the circuit is able to start;
- **Started and no fault detected**, when the circuit is able to start, but no fault is detected. It represents the bad case, because the fault is undetectable with that testing HM.

The testing results for each circuit are shown below. In particular, diagrams are structured as follows:

- the total number of loaded faulty bitstreams is shown in grey;
- the number of *good cases*, that is all that cases in which is possible to obtain in somehow some information about the fault, is shown in green;
- the number of *bad cases* is shown in red;
- the number of each single component which makes up the *good cases* is shown in blue.

Results are obtained running the testing HMs on a Xilinx Virtex-4 FX100 FPGA.

4.4.1 6WUTs results

Figure 4.7 depicts the results related to the 6WUTs version.

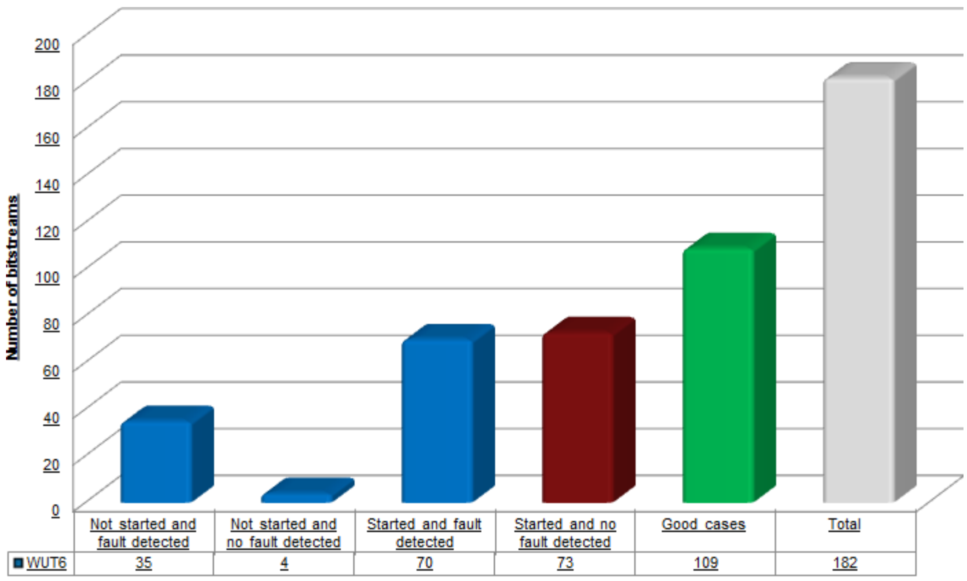


Figure 4.7: 6WUTs verification results

4.4.2 8WUTs results

Figure 4.8 depicts the results associated to the 8WUTs version.

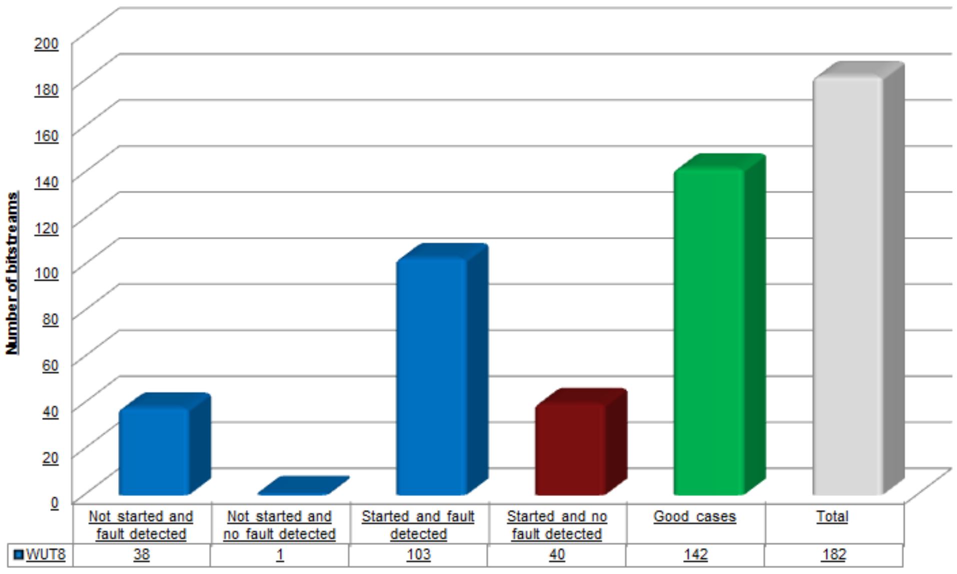
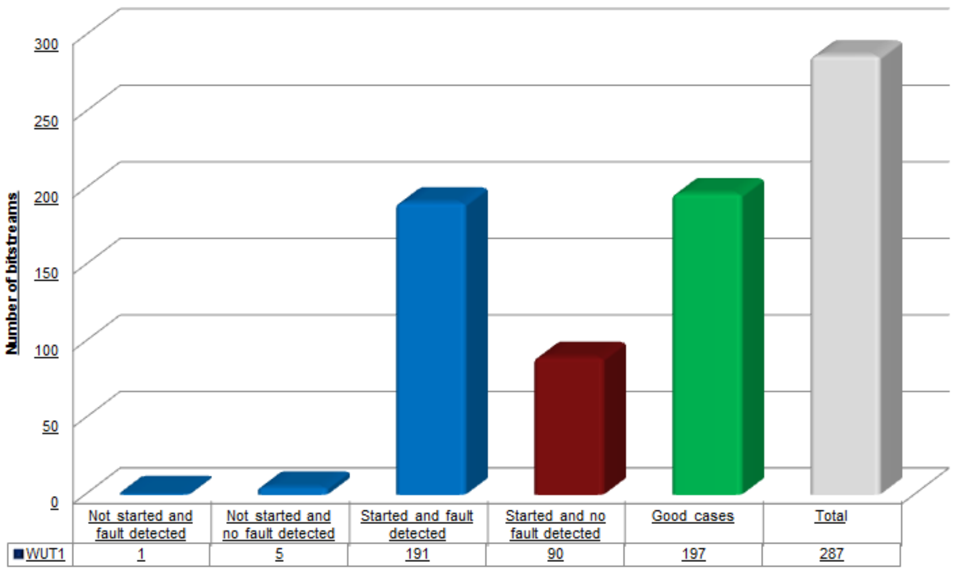


Figure 4.8: 8WUTs verification results

4.4.3 1WUT results

Figure 4.9 depicts the results connected to the 1WUT version.

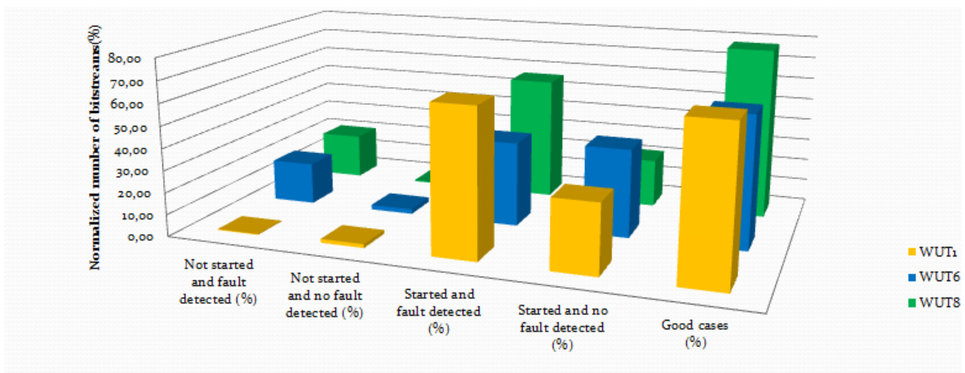


	Not started and fault detected	Not started and no fault detected	Started and fault detected	Started and no fault detected	Good cases	Total
■ WUT1	1	5	191	90	197	287

Figure 4.9: 1WUT verification results

4.4.4 Summary results

The summarized results are shown in Figure 4.10, in particular they are reported in percentage with respect to their own total number of loaded faulty bitstreams.



	Not started and fault detected (%)	Not started and no fault detected (%)	Started and fault detected (%)	Started and no fault detected (%)	Good cases (%)
WUT1	0,35	1,74	66,55	31,36	68,64
WUT6	19,23	2,20	38,46	40,11	59,89
WUT8	20,88	0,55	56,59	21,98	78,02

Figure 4.10: Verification results

4.4.5 Results analysis

In order to find out why:

- **not started and no fault detected** cases are possible;
- there are many *bad cases*;
- 8WUTs HM features a rather higher *fault coverage* than the structurally alike 6WUTs version;

a deeply analysis of results has been performed and its outcomes are hereinafter reported.

Not started and no fault detected case This case can happen when a fault affects a wire which brings the starting information and does not affect, directly or indirectly, wires between TPG and ORA. Figure 4.11 depicts a possible fault which leads to this case.

Bad cases One reason is rather obvious: if a fault occurs in resources which are placed after the ORA, the latter cannot detect it. Figure 4.12 depicts a particular case, where a fault occurs on the wire which brings the assessment result. If a stuck-at 0 fault occurs, the logic value is fixed to 0 and only it can be written in the memory, that is this fault is theoretically able to mask other faults as well as it is undetectable. The second reason derives from how the *gnd* (logic value 0) and *vcc* (logic value 1) signals are typically generated within a hard macro. They are obtained by means of a

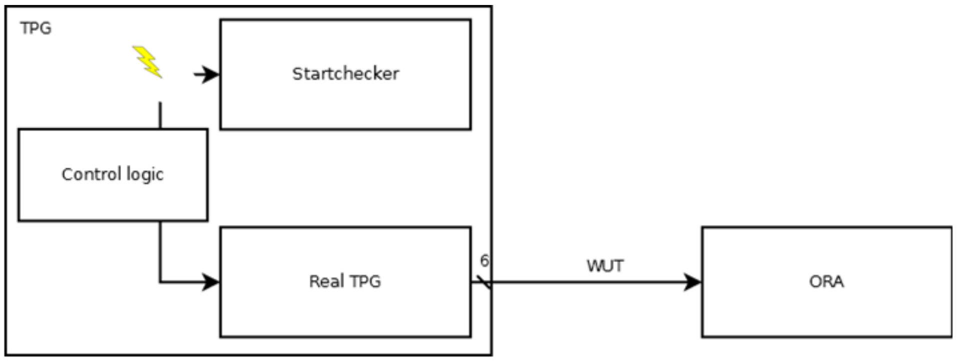


Figure 4.11: Not started and no fault detected scenario

LUT which gives the desired logic value as fixed output. As consequence the wire, which distributes that logic value, is rather long and with high fanout; now supposing a stuck-at 0 for *gnd* or a stuck-at 1 for *vcc*, the circuit is not able to detect the fault, because there are no detectable changes. For example the *gnd* wire has 15 PIPs in the 6WUTs version, therefore in the event of a stuck-at 0 verification they represent 15 bad cases.

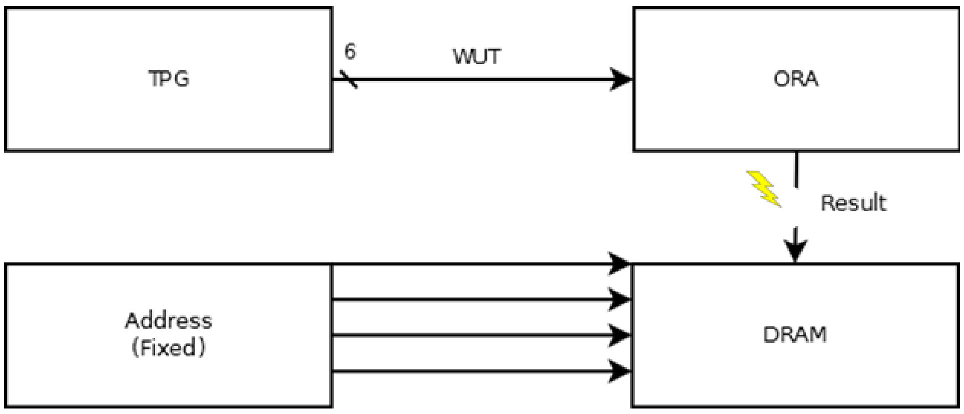


Figure 4.12: Bad cases: masking faults case

8WUTs higher fault coverage The justification lies on the choice of using two signals to address the distributed RAM. Indeed, both in 1WUT and 6WUTs, the results are always written in the same position, that is the first address. Supposing a stuck-at 0 fault in a whatever PIP along one of the four wires, which are used for addressing, this fault is not detectable in the 6WUTs and 1WUT because the address does not change, whereas in the 8WUTs this type of faults are detectable on the addressing wires. Figure 4.13 depicts this case with the 6WUTs.

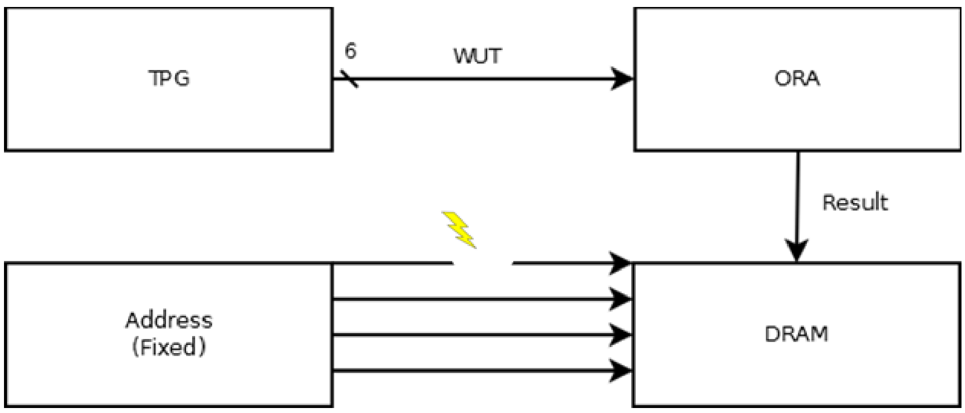
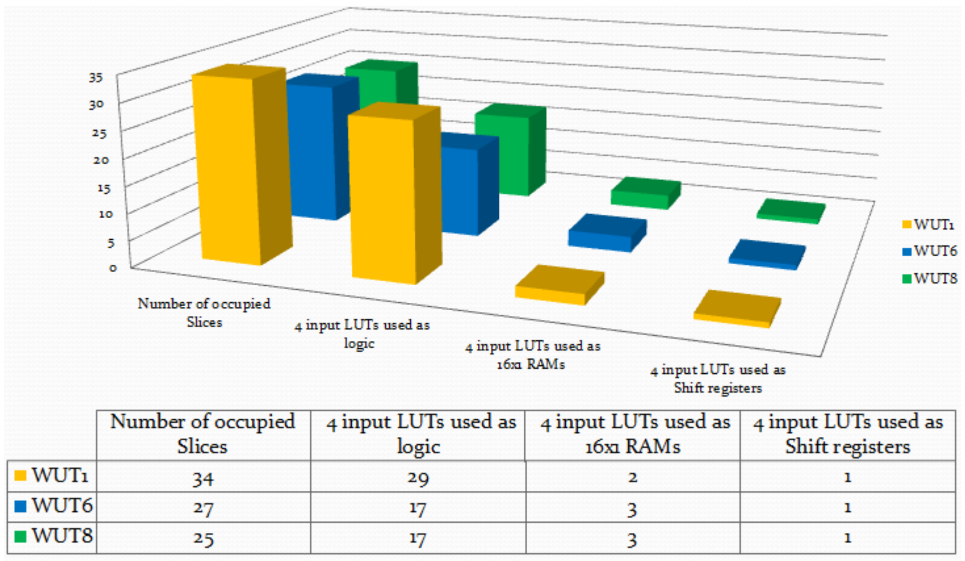


Figure 4.13: Undetectable fault on addressing wires

4.4.6 Device utilization

In this section an idea about testing HMs size is given. Figure 4.14 reports size in terms of occupied slices, LUTs used as logic, distributed RAM and shift register for each circuit. Just to give an idea, the Xilinx Virtex-4 FX100 has 42176 slices and thence 84352 LUTs.



	Number of occupied Slices	4 input LUTs used as logic	4 input LUTs used as 16x1 RAMs	4 input LUTs used as Shift registers
🟨 WUT1	34	29	2	1
🟦 WUT6	27	17	3	1
🟩 WUT8	25	17	3	1

Figure 4.14: HMs sizes summary

Chapter 5

Conclusions and Future Work

In this thesis three circuits for detecting faults, which affect routing resources of FPGAs, have been conceived, developed and verified. Partial reconfiguration has been successfully exploited in order to make possible On-Line testing and it has been shown that the fault coverage reached with a single testing HM, on a Xilinx Virtex-4 FX100 FPGA, varies between 60% and 78%. Even though the fault coverage can seem low it is useful to remember that:

- a tradeoff between effectiveness and simplicity has been taken in account in the design of testing HMs;
- all faults along the wires under test are detected;
- testing HMs use only CLBs and their interconnection resources and they are independent of external signals, such as reset and clock, therefore they can be replicated in different portions of the device and can operate at the same time.

Moreover, hard macros are optimized for Xilinx Virtex-4 devices and an automatic verification flow for stuck-at faults has been developed. Particular attention has been paid to make the verification flow able to obtain results independently where HMs are positioned in the bitstream, which is obtained with a readback, because that position depends on where the distributed RAM, which contains results, is placed within a frame. This feature allows creating different hard macros, starting from the same HDL description of a testing circuit, changing the placing and routing, for example by means of constraints. In this way it is possible to obtain a set of testing HMs, which are sequentially usable in every frame in order to test all the routing resources in that frame and improve fault coverage. It is also possible to realize custom HMs, that is HMs which are able to detect faults in routing

resources which are actually used by a given design. It is enough to constrain the placing and routing phases, for example with the aid of DHHarMa.
As further work, other interesting aspect will be considered, such as verifying how testing circuits behave in the presence of faults (multiple fault analysis), as well as improving circuits in terms of fault coverage and utilized logic, and providing better support for new device families.

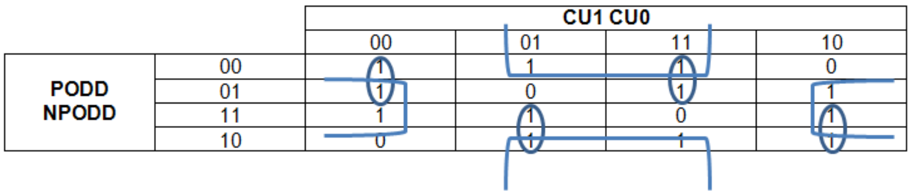
Appendix A

.1 ORA 8WUTs synthesis

.1.1 ODD parity ORA

Cu1	**Cu0**	**PODD**	**NPODD**	**Pass/Fail**
0	0	1	0	0
0	1	0	1	0
1	0	0	0	0
1	1	1	1	0
Whatever else				0

Table 1: ODD parity ORA truth table



		CU1 CU0			
		00	01	11	10
PODD NPODD	00	1	1	1	0
	01	1	0	1	1
	11	1	1	0	1
	10	0	1	1	1

Figure 1: ODD parity ORA Karnaugh‘s map

Synthesis

$$\begin{aligned} Pass/Fail = {} & Cu0 \cdot \overline{NPodd} + \overline{Cu0} \cdot NPodd \\ & + \overline{Cu0} \cdot \overline{Cu1} \cdot \overline{Podd} + Cu1 \cdot \overline{Cu0} \cdot Podd \\ & + \overline{Cu1} \cdot Cu0 \cdot Podd + Cu1 \cdot Cu0 \cdot \overline{Podd} \end{aligned} \tag{1}$$

$$\begin{aligned} Pass/Fail = {} & Cu0 \oplus NPodd + \overline{Cu0} \\ & \cdot (\overline{Cu1} \cdot \overline{Podd} + Cu1 \cdot Podd) \\ & + Cu0 \cdot (\overline{Cu1} \cdot Podd + Cu1 \cdot \overline{Podd}) \end{aligned} \tag{2}$$

$$Pass/Fail = Cu0 \;\oplus\; NPodd \;+\; \overline{Cu0} \\ \cdot\; (\overline{Cu1 \;\oplus\; Podd}) \;+\; Cu0 \;\cdot\; (Cu1 \;\oplus\; Podd) \tag{3}$$

$$Pass/Fail = Cu0 \;\oplus\; NPodd \;+\; \overline{Cu0 \;\oplus\; Cu1 \;\oplus\; Podd} \tag{4}$$

Proof

Cu0	**NPodd**	$Cu0 \;\oplus\; NPodd$
0	0	0
0	1	1
1	0	1
1	1	0

Table 2: ODD parity ORA proof table 1

Cu1	**Cu0**	**Podd**	$\overline{Cu1 \oplus Cu0 \oplus Podd}$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Table 3: ODD parity ORA proof table 2

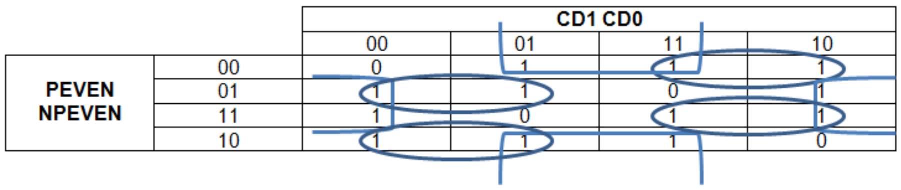
Cu1	**Cu0**	**Podd**	**NPodd**	$Cu0 \oplus NPodd + \overline{Cu1 \oplus Cu0 \oplus Podd}$
0	0	0	0	1
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

Table 4: ODD parity ORA proof table 3

.1.2 EVEN parity ORA

Cu1	**Cu0**	**PODD**	**NPODD**	**Pass/Fail**
1	1	0	1	0
1	0	1	0	0
0	1	1	1	0
0	0	0	0	0
Whatever else				0

Table 5: EVEN parity ORA truth table



		CD1 CD0			
		00	01	11	10
PEVEN NPEVEN	00	0	1	1	1
	01	1	1	0	1
	11	1	0	1	1
	10	1	1	1	0

Figure 2: EVEN parity ORA Karnaugh‘s map

Synthesis

$$\begin{aligned} Pass/Fail = Cd0\ \cdot\ \overline{NPeven}\ +\ \overline{Cd0}\ \cdot\ NPeven \\ \cdot\ Cd0\ \cdot\ \overline{Cd1}\ \cdot\ Peven\ +\ Cd1\ \cdot\ Cd0\ \cdot\ Peven \\ +\ \overline{Cd1}\ \cdot\ Cd0\ \cdot\ \overline{Peven}\ +\ Cd1\ \cdot\ \overline{Cd0}\ \cdot\ \overline{Peven} \end{aligned} \tag{5}$$

$$\begin{aligned} Pass/Fail = Cd0\ \oplus\ NPeven\ +\ \overline{Cd0} \\ \cdot\ (Cd1\ \cdot\ \overline{Peven}\ +\ \overline{Cd1}\ \cdot\ Peven) \\ +\ Cd0\ \cdot\ (\overline{Cd1}\ \cdot\ \overline{Peven}\ +\ Cd1\ \cdot\ Peven) \end{aligned} \tag{6}$$

$$\begin{aligned} Pass/Fail = Cd0\ \oplus\ NPeven\ +\ \overline{Cd0} \\ \cdot\ (Cd1\ \oplus\ Peven)\ +\ Cd0\ \cdot\ (\overline{Cd1\ \oplus\ Peven}) \end{aligned} \tag{7}$$

$$Pass/Fail = Cd0\ \oplus\ NPeven\ +\ Cd0\ \oplus\ Cd1\ \oplus\ Peven \tag{8}$$

Proof

Cd0	**NPeven**	$Cd0\ \oplus\ NPeven$
0	0	0
0	1	1
1	0	1
1	1	0

Table 6: EVEN parity ORA proof table 1

Cd1	**Cd0**	**Peven**	$Cd1 \oplus Cd0 \oplus Peven$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Table 7: EVEN parity ORA proof table 2

Cd1	**Cd0**	**Peven**	**NPeven**	$Cd0 \oplus NPeven + Cd1 \oplus Cd0 \oplus Peven$
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

Table 8: EVEN parity ORA proof table 3

Bibliography

[1] I. Kuon and J. Rose, “Measuring the gap between fpgas and asics,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 26, pp. 203–215, Feb 2007.

[2] K. Underwood, “Fpgas vs. cpus: Trends in peak floating-point performance,” in *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, FPGA ’04, (New York, NY, USA), pp. 171–180, ACM, 2004.

[3] M. V. Niccolò Battezzati, Luca Sterpone, *Re-configurable Field Programmable Gate Arrays for Mission-Critical Applications.* Springer New York, July 2014.

[4] Xilinx, *ISE In-Depth Tutorial*, April 2012. UG695.

[5] Xilinx, *Partial Reconfiguration Flow Presentation Manual.* Xilinx University Program.

[6] Xilinx, *Partial Reconfiguration User Guide*, January 2012. UG702.

[7] Xilinx, *Virtex Series Configuration Architecture User Guide*, September 2000. XAPP151.

[8] Xilinx, *Correcting Single-Event Upsets in Virtex-4 FPGA Configuration Memory*, October 2005. XAPP1088.

[9] R. Baumann, “Radiation-induced soft errors in advanced semiconductor technologies,” *Device and Materials Reliability, IEEE Transactions on*, vol. 5, pp. 305–316, Sept 2005.

[10] C. Bernardeschi, L. Cassano, and A. Domenici, “Seu-x: A seu unexcitability prover for sram-fpgas,” in *On-Line Testing Symposium (IOLTS), 2012 IEEE 18th International*, pp. 25–30, June 2012.

[11] M. C. C. Bernardeschi, L. Cassano and A. Domenici, “Gabes: a genetic algorithm based environment for seu testing in sram-fpgas,” *Journal of Systems Architecture*, vol. 59, pp. 1243–1254, November 2013.

[12] L. Cassano, D. Cozzi, S. Korf, J. Hagemeyer, M. Porrmann, and L. Sterpone, "On-line testing of permanent radiation effects in reconfigurable systems," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pp. 717–720, March 2013.

[13] C. Stroud, S. Wijesuriya, C. Hamilton, and M. Abramovici, "Built-in self-test of fpga interconnect," in *Test Conference, 1998. Proceedings., International*, pp. 404–411, Oct 1998.

[14] B. C. X. Sun, J. Xu and P. Trouborst, "Novel technique for bist of fpga interconnects," in *Proc. IEEE International Test Conf*, pp. 795–803, 2000.

[15] B. Dixon and C. Stroud, "Analysis and evaluation of routing bist approaches for fpgas," in *Proc. IEEE North Atlantic Test Workshop*, pp. 85–91, 2007.

[16] C. Beckhoff, D. Koch, and J. Torresen, "The xilinx design language (xdl): Tutorial and use cases," in *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2011 6th International Workshop on*, pp. 1–8, June 2011.

[17] S. Korf, D. Cozzi, M. Koester, J. Hagemeyer, M. Porrmann, U. Ruckert, and M. Santambrogio, "Automatic hdl-based generation of homogeneous hard macros for fpgas," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pp. 125–132, May 2011.

[18] D. Cozzi, J. Hagemeyer, S. Korf, D. Jungewelter, and M. Porrmann, *DRPM User Manual Guide*, February 2014.

[19] W.-K. Huang, F. Meyer, N. Park, and F. Lombardi, "Testing memory modules in sram-based configurable fpgas," in *Memory Technology, Design and Testing, 1997. Proceedings., International Workshop on*, pp. 79–86, Aug 1997.

[20] L. Bauer, C. Braun, M. Imhof, M. Kochte, H. Zhang, H. Wunderlich, and J. Henkel, "Otera: Online test strategies for reliable reconfigurable architectures; invited paper for the ahs-2012 special session ;dependability by reconfigurable hardware;," in *Adaptive Hardware and Systems (AHS), 2012 NASA/ESA Conference on*, pp. 38–45, June 2012.

[21] Z. Zhang, Z. Wen, L. Chen, F. Zhang, and T. Zhou, "A novel bist approach for testing logic resources using hard macro," in *Neural Networks and Signal Processing, 2008 International Conference on*, pp. 379–381, June 2008.

[22] S. Dhingra, C. Stroud, and D. Milton, “Bist for logic and memory resources in virtex-4 fpgas,” in *IEEE North Atlantic Test Workshop - NATW*, pp. 19–27, 2006.

[23] A. Hassan, J. Rajski, and V. Agarwal, “Testing and diagnosis of interconnects using boundary scan architecture,” in *Test Conference, 1988. Proceedings. New Frontiers in Testing, International*, pp. 126–137, Sep 1988.

[24] J. S and V. K. Agrawal, “Article: Detection and diagnosis of faults in the routing resources of a sram based fpgas,” *International Journal of Computer Applications*, vol. 53, pp. 18–22, September 2012. Published by Foundation of Computer Science, New York, USA.

[25] M. Abramovici, C. Strond, C. Hamilton, S. Wijesuriya, and V. Verma, “Using roving stars for on-line testing and diagnosis of fpgas in fault-tolerant applications,” in *Test Conference, 1999. Proceedings. International*, pp. 973–982, 1999.

[26] J. Yao, B. Dixon, C. Stroud, and V. Nelson, “System-level built-in self-test of global routing resources in virtex-4 fpgas,” in *System Theory, 2009. SSST 2009. 41st Southeastern Symposium on*, pp. 29–32, March 2009.

[27] M. Technology, *User Manual*, August 2002. Version 5.6 d.

[28] Atmel, *Manchester Coding Basics*, September 2009. Application Note.

[29] Xilinx, *Manchester Encoder-Decoder for Xilinx CPLDs*, October 2002. XAPP339.