POLITECNICO DI TORINO

Master Degree Course in Electronic Engineering

Master Degree Thesis

Study and Development of a Radiation-Hardened Implementation of the RISC-V Processor on Reconfigurable Devices

Supervisors

Prof. Luca Sterpone

Post Doc. Sarah Azimi

Ph.D. Corrado De Sio

Candidate

Eleonora Vacca

Academic Year 2020/2021

Summary

Human progress and the curiosity that accompanies it, the will to know the unknown and to find its secrets have fueled the research in the aerospace field. Nowadays, aerospace companies are looking for the best hardware and software solutions that satisfy on one hand their hunger for performance and efficiency and on the other hand solutions require to be verifiable, economical, safe. In this context, RISCV ISA has emerged as a good candidate capable of meeting these prerequisites, with the addition of being an open source alternative to commercial ISAs. Although RISCV is part of a rather young new reality, it has quickly become a major player in the research world and beyond, both because it is an open source with an efficient design but also because it is involved in the continuous development of resources such as operating system support, debuggers, emulators, useful to facilitate its use and adoption. However, another prerogative that a design must possess to be used in the field of aerospace is reliability. A reliable system is widely viewed to be a system that tolerates faults; in the aerospace applications faults may occur as consequence of radiations which manifest themselves as SEU and SEMU. Having at hand a SoC design of a RISCV implemention for FPGA, this thesis work starts with initial investigation of the RISCV architecture and how it has been placed into FPGA resources. The interest in the placement arises from the idea that the placement algorithm used by the FPGA tool does not take into account the field of application of the design and so uses a default directive for the place and route algorithm, leading to potentially critical situations. Once it has been ascertained that the FPGA design tool returns an implementation that is not

robust, the next step consisted in the extraction of a salient module for the correct functioning of the processor, as is the ALU, from its RTL description with the aim of make it robust. During the ALU hardening process, consisting in applying the TMR technique, it has been evaluated how the place and route affects the robustness of the design. In particular, three designs which present differences in terms of placement have been analyzed: the original placement, which is the one proposed by the tool, versus two custom placements. The developed placements aimed to emphasize in one case the isolation among replicas while the other one implements the worst case scenario where replicas share a large amount of resources. For each placement a fault injection campaign was carried out with the purpose of simulating the effect of SEU in the design. In order to test the behavior of the TMR ALU, a test program has been developed with the purpose of stimulate as much as possible all the available functionality of the module. The data collected revealed that the original placement is subject more to failures than the isolated placement and less with respect to the worst case. Therefore, the isolated design has been taken as reference for the further investigation and optimization. From further injection campaign, performed on sensitive area of the TMR-isolated placement design, it has been possible to observe how the voter which is an essential module for the TMR technique is also the major source of failures, due to the convergence of the nets of the replicas in the same sites. In-depth analyzes showed that although the replicas are positioned far from each other, the convergence of the nets to the voter pushes the tool's routing algorithm to assign routes such that the nets of a replica are articulated in the area where another replica is arranged. So, isolation at the cell level is not enough but it must also be applied at the routing level. Having this in mind, a basic router has been implemented: it's able to provide a feasible path, that reflect custom constraints, that in this context consist in keep the routing within a certain area (i.e where the parent replica is defined).

General Structure

Chapter 1 an introduction about the RISCV architecture and the reason why it is one of the most widely used architectures.

Chapter 2 is a look inside the FPGA world, from the general structure to the detail of the specific board used in this thesis work.

Chapter 3 is a summary on the radiation effects theory and the most common solution adopted to mitigate its effects. .

Chapter 4 describes the approach aimed to analyze the design , the development of a Python environment used to observe and control the design, the hardening process of the RISCV ALU and the realization of its different placement arrangements. Finally, the implementation of an FPGA router to apply isolation at the net level

In chapter 5 the different placements are discussed together with the fault injection campaigns conducted on each of them, accompanied by the analysis of the obtained results.

Chapter 6 is devoted to conclusions and prospects for future research work.

Contents

List of Figures 8

Abbreviations 11

1 Introduction 13

1.1 RISCV: From The Origin to PULPissimo RI5CY 13

1.1.1 RI5CY 15

1.2 RISCV Employement for Space Application 18

2 Technology Background 21

2.1 Nexys Video Board 22

2.2 PYNQ-Z2 Development Board 24

2.3 Xilinx FPGA Design Flow 25

2.3.1 RTL-to-Bitstream Design Flow 26

2.3.2 Software Development 28

3 Radiation effects on FPGA Devices 31

3.1 Radiation effects 31

3.1.1 SEU: single event upset 32

3.1.2 SEU on FPGA 34

3.2 Fault Tolerant Design Technique 35

4 Analysis and Hardening of the RI5CY design on FPGA 39

4.1 Development Of The FPGA Equivalent Enviroment To Observe And Control Any Design . . . 40
4.1.1 Design Information Extraction . . . 42
4.1.2 FPGA Equivalent Enviroment . . . 43
4.2 RI5CY ALU Hardening . . . 46
4.2.1 RI5CY ALU . . . 46
4.2.2 TMR on RI5CY . . . 47
4.3 Test Enviroment Design on Vivado Design Tool . . . 48
4.3.1 Hardware Context Development on Vivado Design Suite . . . 49
4.4 Different Placement Arrangements . . . 50
4.5 TMR Voter . . . 55
4.6 A Basic Router To Apply Isolation at Net Level . . . 56
4.6.1 Routing Problem on FPGA . . . 57
4.6.2 General Routing Background . . . 58
4.6.3 A Look Inside the Xilinx Artix-7 Switch Matrix . . . 59
4.6.4 The Router . . . 62

5 Experimental Analysis 69
5.1 Bitstream File Organization . . . 69
5.2 Software Test Enviroment . . . 70
5.3 Fault Injection . . . 72
5.4 Experimental Results . . . 73
5.4.1 Comparison Between the Three Placements Arrangements . . . 73
5.4.2 Voter Switch Matrices Fault Injection . . . 73
5.4.3 Fault injection applied to the design of the TMR technique extended to the voter . . . 76

6 Conclusions 79
6.1 Future Work . . . 80

List of Figures

1.1 RI5CY 4-stage pipelined microprocessor 15

2.1 Diligent Nexys Video Board Overview 23

2.2 PYNQ-Z2 Board Overview 24

3.1 SRAM Cell : 6 transistors structure 33

4.1 A SLICE internal overview 41

4.2 Tcl report property command output example on LUT 42

4.3 *PULPissimo* design plot on the FPGA equivalent enviroment with architectural depth equal to 5 45

4.4 Triple Modular Redundancy applied to RI5CY ALU 48

4.5 Hardware enviroment to test TMR ALU 50

4.6 TMR ALU post place and route view without constraints. 52

4.7 TMR ALU post place and route view with isolation constraints . . 53

4.8 TMR ALU post place and route view with worst case scenario . . . 55

4.9 Configurable Logic Block view and its associated interconnections . 58

4.10 Switch Matrix view : node's pips example 60

4.11 Switch Matrix view with all its pips highlighted 61

4.12 Fixed Route example: the blue and the yellow net with and without constraint 67

4.13 Before and after net isolation applied to the the yellow and green nets 68

5.1 Bitstream organization structure: frame recursion scheme 70

5.2 Routing resources close to the voter module 74

5.3 On the right the bitmap of the design, on the left the device view of Vivado . . . 75
5.4 Voter bitmap . . . 75

Abbreviations

ALU	Arithmetic Logic Unit.
ASIC	Application Specifc Integrated Circuit.
BRAM	Block Ram Memories.
CLB	Configurable Logic Block.
COTS	Commercial Off The Shelf.
FPGA	Field Programmable Gate Array.
HDL	Hardware Description Language.
IC	Integrated Circuit.
IP	Intellectual Property.
IPC	Instructions Per Cycle.
ISA	Instruction Set Architecture.
LUT	Look-Up Tables.
PL	Programmable Logic.
PS	Processing System.

RTL	Register Transfer Level.
SEU	Single Event Upset.
SIMD	Single Instruction Multiple Data.
SoC	System-On-Chip.
TMR	Triple Modular Redundancy.

Chapter 1

Introduction

At the base of technological progress is sharing. Without sharing and collaboration, the world as we know it today and which we continue to build day by day, would not have been possible. The standardization of both software and hardware, their advancement and the open source development and provision of these, has had an impact on technical progress beyond all expectations, on a global scale.

RISCV ISA promotes a contributing reality that marked the beginning of a new era of the open computing, where open source has become synonymous of innovation through collaboration. This thesis work was possible thanks to these cooperation concepts, since the research starts from an open source distribution of the RTL description of a hardware design compliant with the RISCV instruction set architecture.

1.1 RISCV: From The Origin to PULPissimo RI5CY

The RISCV project began in 2010 at the University of California, Berkeley [1]. RISCV is characterized by a modular and intuitive design, which makes it easy to use and understand, so it is heavily used for academic as well as more practical purposes. Usually, its structure composed of basic modules can be easily customized, if needed, by adding optional extensions. Its design aims to provide a good trade off between low power and performance. The base instruction set has been developed for both 64 and 32 bit parallelism referred to as RV64I and RV32I, respectively. It's a load and store architecture therefore, no operations can be performed directly on operands saved in memory but instructions address only registers; RISCV register file can be either on 32 or 16 integer registers (according to the variant), and, when the floating-point extension is implemented, separate 32 floating-point registers. Except for the first integer register, that is a zero register which means that any write operation in it has no effect and that any read operation returns a zero, all the other registers are general purpose registers. Each instruction of the ISA is encoded on 32-bit and each type of instruction (R-type, I-type, Branch, Store, Jump) has a specific instruction format.

The RISCV organization maintains a list of RISCV CPU and SoC implementation, one of this is the open-source RI5CY processor [2] as part of the Parallel Ultra-Low Power (PULP) project born from a cooperation between ETH Zurich and the University of Bologna.

Starting point of this work is the PULPIssimo single-core microcontroller based on RI5CY processor.

- **PULPissimo**

  PULPissimo is the microcontroller architecture of the more recent PULP chips, it is a single-core system where the core can be either the RI5CY processor or the Ibex, known as Zero-RISCY core. The core adopted in this study is the RI5CY one. As a complete system on chip, its architecture includes:

  - RI5CY 4-stage pipelined processor
  - uDMA
  - L2 multi-bank RAM
  - Support for Hardware Processing Engines (HWPEs)
  - Interrupt controller
  - Peripherals: I2C, SPI, JTAG, SDIO, UART

1.1.1 RI5CY

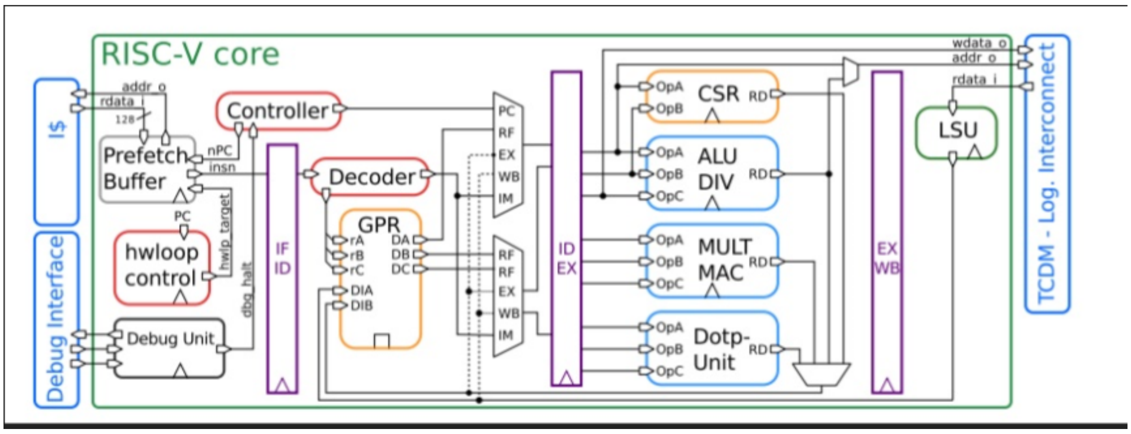


Figure 1.1. RI5CY 4-stage pipelined microprocessor

RI5CY processor is a 32-bit pipelined processor that supports the base instruction set RV32I, compressed instructions (RV32C) and multiplication instruction set extension (RV32M). It can be configured to have single-precision floating-point instruction set extension (RV32F). The number of pipeline stage adopted has been a crucial and strongly motivated choice of designers: as part of Parallel Ultra-Low Power (PULP) project the core aim to provide both high IPC but at low cost in terms of power; in order obtain high performance, typically, high number of pipeline stage is chosen but leads to some pros as the working frequency can be increased as well as the throughput but also have some cons as the data and control hazards increases causing stalls and deteriorating the IPC. To correctly handle this kind of situation, the core grows also in complexity which means having branch predictor, prefetch buffers, and speculation. Complexity may solve performance problems but increase the power consumption, which is not in line with the low power policy of the project. Therefore, the core is on 4 pipeline stage (IF, ID, EX, M) to ensure the balance between IPC and power consumption.

- *Instruction Fetch Unit*

As said before, the core supports both the base instruction set, where instructions are encoded on 32 bit, and the compressed one with instructions on 16 bits. Even if the decoder is able to easily recognize a 16-bit compressed instruction and decompress it, some issues arise when the number of compressed instructions is odd and it is followed by a 32 bit instruction; this situation causes a misalignment that requires an additional fetch cycle for which the processor will have to be stalled. To reduce the cost of this situations, designer added a prefetch buffer so that a whole cache line is read at one time, instead of a single instruction, plus an additional register used to store the last instruction. This register is needed when a 32-bit instruction is split across two cache lines, again causing a misalignment: the idea is that in case of unaligned 32-b instruction, this register will store the least 16-bits of the current instruction, that can be combined with the higher 16 bits coming from the prefetcher and forwarded to the ID stage. This solution allows unaligned access to instruction cache avoiding stalls creation except in case of branches, jumps or hardware loops that require a new cache line to be fetched to get the new instruction.

- *Hardware Control Loop Engine*

Loops are execution and memory expensive parts of the program code and for this reason optimization techniques are sought. Typically, in most of the common ISA the execution of a loop requires the fetching of instruction devoted to the increment or decrement of a counter, instructions to check whether the end of the loop has been reached and according to the comparison result either loop again, which means fetching instructions from the initial address of the loop or go ahead with the program code. All these instructions constitutes an overhead both in terms of memory (cause they will be stored in the cache) and performance, since each of them take time to execute, time which is not spent doing useful work. To reduce this overhead, techniques as unrolling or branch prediction can be adopted but the first will cause increment in program code size and may leads to performance loss due to cache miss, the second one requires additional hardware support and may fails: When a branch miss-prediction occurs, typically the pipeline must flush and fetch from the correct address and thus, a number of cycles are wasted equal to the number of pipe stages there are from the fetch stage to the execute stage. With a view to low energy consumption, the use of these techniques is replaced with ad hoc hardware that has the purpose of reducing the overhead loop to zero. In the RI5CY core loops are handled by Hardware loop-controller which is in charge of providing the start loop address to the fetch as long as the loop counter has not reached 0. Using this ad hoc hardware resource eliminates instructions to test the loop counter and perform branches, thereby reducing the number of instructions fetched. Control Loop Engine has been realized by means of a controller and a set of registers to store loop information (loop counter, the start address, and the end address).

- *Load–Store Unit*

  The original RISCV supports only one addressing mode where effective address is computed by adding an offset coming from an immediate to the base address stored in a register. RI5CY includes the ability to store the offset in both a register and an immediate. In addition, post-increment addressing mode is available, which is useful when execution involves regular memory access patterns, such as occurs in matrix multiplication operations. Unaligned data memory accesses, common in vector operations, are also provided,

- *EX-Stage*

  - ALU

    In addition to the common logic and arithmetic operations on 32 bits, the ALU of the RI5CY can be configured to work in packed-SIMD mode [3] which basically consist into work with 8- and 16-bit data: for example in case of vector mode 8, each ALU operand on 32-bit will be considered as made by four independent 8-bit data that will be processed in parallel through a vectorized datapath segmented into four parts. Moreover, original RISC-V ISA has been expanded, in this RI5CY implementation, to support bit manipulation instructions such as *bclr* or *bset* able to clear/set a set of bits, or even *cnt* (count number of bits that are set to 1) and many others. The ALU hardware resources such as comparators, shifters, adders are also employed to implement long integer division algorithm. The divider does not provide great performance, as its latency can reach up to 32 cycles depending on the input operand, but it allows for significant savings in the area thanks to the sharing of resources with the ALU.

– Multiplier

  The multiplier is a complex modular structure consisting of a 32bit operand multiplier, two dot-product multipliers inserted with the purpose of reducing the effort of producing a result with a larger dynamic than the operands that are either on 8- or 16 bits, and a fractional multiplier. It produces the result of multiplication between two vectors and accumulate it in a 32-bit value in one cycle. Since a vector can contain either two 16-bit elements or four 8-bits elements, to correctly perform signed and unsigned multiplications, the 8-/16-bit inputs are sign extended; For the purpose of gain in performance, multiplications exploit carry-save format.

The RI5CY designers, running a series of benchmarks on the processor, observed that the core with the extended instruction set, supported by the additional hardware, performs about 37% better on general purpose applications and when used in SIMD mode, performance increases even more.

1.2 RISCV Employment for Space Application

Hardwares and Softwares that are employed for aerospace applications need to be *secure and trusted.* The power behind open source IP cores, as the RISCV, lies in the total transparency that characterizes open sources, which facilitates the process of creating safe and reliable designs. Since the details of the IP are available, as its RTL description, it is easier to perform inspections and possibly apply changes or improvements avoiding the need to design everything from scratch. As opposed to commercial ISAs and protected IPs which are characterized by restrictions that limit their inspection. Typically, when a processor core has to be used in aerospace applications [4] it has to be taken into account how the space environment will induce functional faults and how to mitigate them. Possible approaches are:

- a Fault-Tolerant IP Core designed from scratch
- a Fault-Tolerant IP processor obtained modifying a COTS IP core
- a COTS IP core without any modifications

Commercial off-the-shelf (COTS) products are designed to minimize the effort associated with their installation in the system and the subsequent interaction with other components that make up the system. Therefore, the only improvement that can be made to increase fault tolerance is at the system level. The solution of COTS IP, as it is, is widely adopted cause it is low cost and ensure performance. However, they're not designed to be suitable for space and sometimes their lack of functionalities can't be compensated at system level: For example, if the IP core or ISA are not designed to be transparent to the user, they may be modeled as a black box whose functionality is known but not how it has been implemented. In the event of a failure, the user does not know what is going on inside the module and therefore is unable to tune system-level response adequately, reducing security and availability. On the other hand, using a modified COTS IP, enhanced for fault tolerance, is an expensive solution since to be modified, first IP licenses or access to source code must be purchased. In this scenario, the open source RISCV makes its way since, as there is no need to pay to change its RTL description or use its ISA, designers can make any changes dedicated to fault tolerance freely. Another reason for adopting the RISCV ISA for aerospace applications is that aerospace platforms often have service lives measured in decades and the RISCV base ISA is *frozen* so that designer can be sure that the platform they will create around the RISCV will be of a long-term stability. For example, a software created for a RISCV hardware that exists now will continue to run on future RISCV chips. That means code can be validated once and then reused on future generations of RISCV processors.

Chapter 2

Technology Background

PULPissimo project is a Microcontroller provided in SystemVerilog RTL description. Moreover, the designers distribute working FPGA implementations of the microcontroller, based on several Xilinx FPGA boards. Implemented design means that the bitstream to be loaded in the configuration memory of the board has been already generated and the design is delivered as post *place&route*. FPGA, which stands for Field Programmable Gate Arrays, is a semiconductor device that consists of a bi-dimensional array structure of configurable logic blocks (CLBs) that are connected by means of programmable interconnects (Switch Boxes). In Xilinx FPGAs, each CLB consists of two SLICEs and has associated a Switch Matrix. Each SLICE contains 4 Look-Up Tables (LUT), some logic gates, 8 flip-flops and a carry logic block. In addition to the CLBs, some special purpose blocks are present in the programmable logic matrix, such as dual-port RAM module known as block RAM (BRAM) and digital signal processor (DSP). These blocks are connected to the rest of the logic by means of ad hoc interconnection matrices which in turn connect with the CLB switch matrices.

The main advantage of the FPGA over the IC is its ability to be dynamically reconfigured. This process is very similar to loading a program into a processor, except that in memory we are not loading machine instructions, but bits that configure hardware so that it behaves in a certain design-defined way, and can involve some or all of the available resources in the FPGA. FPGA logic can be used to implement any kind of circuit or algorithm; obviously the performances that can be obtained using programmable logic are lower than those of the same design implemented on ASIC but thanks to technological progress, today there are FPGAs able to run up to 500MHz. In general FPGA are well suited for prototyping as in the case of RISCV implementation since the design flow is simplistic and there's no development cost. Among the various Xilix FPGA boards supported by *PULPissimo*, the implementation on the Digilent Nexys Video board has been chosen as starting point for this thesis work. Afterwards, more in-depth analysis and design improvements were tested using the PYNQ-Z2 board.

2.1 Nexys Video Board

The Nexys Video [5] is a development platform based on the latest Artix-7™ FPGA from Xilinx [6]. The board is equipped with an efficient FPGA which, accompanied by external memories, high-speed digital video ports, and 24-bit audio codec, make it particularly suitable for video and audio processing applications. The platform includes several built-in peripherals such as Ethernet, USB-UART, a vast assortment of switches, buttons, and LEDs and connector for USB-HID devices, and also OLED Display. Going into the details of the features that the Nexys video presents, the remarkable ones are the internal clock speeds that exceeds 450 MHz which is of interest considering it is an FPGA device, on-chip analog-to-digital converter and 740 DSP slices. Regarding programmable logic, this is divided into 33,650 logic slices. About memory, there are close to 13 Mbits of fast block RAM spread on the programmable logic space and two external memories: a 512MiByte volatile DDR3 memory and a 32MiByte non-volatile Serial Flash device. The DDR3 uses a 16-bit wide memory component soldered on the board with industry-standard interface. The Serial Flash is on a dedicated quad-mode (x4) SPI bus. Also, a MicroSD card connector is available on board. Each FPGA slice contains four LUTs and eight flip-flops; only some slices can use their LUTs as distributed RAM or SRLs. Each

DSP slice contains a pre-adder, a 25x18 multiplier, an adder, and an accumulator. The Block RAMs are fundamentally 36 Kb in size but each block can also be used as two independent 18 Kb blocks.

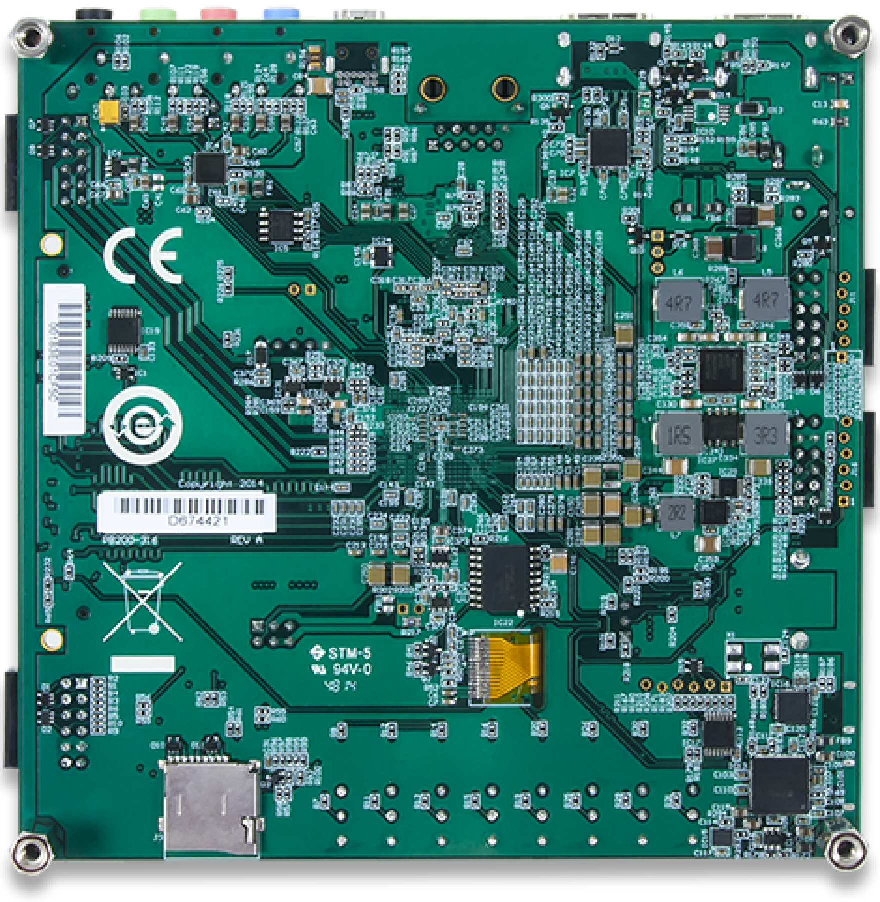


Figure 2.1. Diligent Nexys Video Board Overview

2.2 PYNQ-Z2 Development Board

PYNQ™-Z2 [7] is a board designed for the Xilinx University Program to support Python Productivity for Zynq as it is based on the Xilinx Zynq SoC and for embedded systems development. The board is equipped with a Xilinx Zynq-7020 XC7Z020-1CLG400C SoC, which integrates two ARM Cortex-A9 cores running at 650 MHz, 512 MB of DDR3 RAM and 16 MB of QSPI flash storage. In addition to the ARM cores, the board is provided with programmable logic (PL) organized in:

- 13,300 logic slices, each with four 6-input LUTs and 8 flipflops
- 630 KB block RAM
- 220 DSP slices
- On-chip Xilinx analog-to-digital converter (XADC)

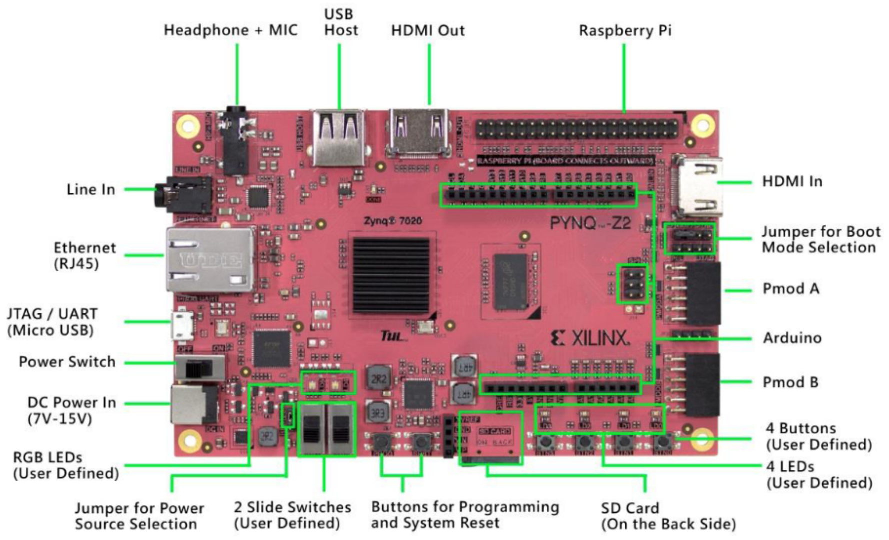


Figure 2.2. PYNQ-Z2 Board Overview

This logic is programmable from JTAG, Quad-SPI flash, and MicroSD card. There are also HDMI in and HDMI out, Gigabit Ethernet and two USB ports, one of which is a micro USB from which the board can also be powered if no external power regulator is used. The USB_UART allows PC applications to communicate with the board using standard COM port commands (or the tty interface in Linux and MacOS). The Zynq PS UART 0 controller is used to connect to the UART device. About the clock resources, a 50 MHz oscillator is present on the board which is used to provide the clock input to the Zynq processing system (PS) which in turn is used to generate the clocks for each of the PS subsystem. The 50 MHz input allows the processor to operate at a maximum frequency of 650 MHz. Moreover, the PYNQ-Z2 has an external 125 MHz reference clock connected to pin H16 of the PL. The external reference clock allows the PL to be used independently of the PS. The PS incorporates an AXI memory port interface, a DDR controller, the associated PHY and a dedicated I/O bank. Another relevant features are the additional connectors that make the board compatible with Arduino and Raspberry Pi. Like all development boards, there is no shortage of leds, buttons, switches and numerous FPGA I/O pins.

2.3 Xilinx FPGA Design Flow

Together with the boards, Xilinx provides *Vivado Design Suite* a software suite produced for synthesis and analysis of HDL designs. The tool offers the ability to follow either the traditional register transfer layer (RTL) to FPGA bitstream design flow or use system-level integration flows that rely on design by intellectual property (IP) and C-based design [8]. At each stage of the design flow, analysis and verification are enabled, allowing you to: perform a logical simulation of the design, estimate power consumption, define constraints, I/O and clock planning, design rule checks (DRC), modify implementation results. The design flow can be performed either by means of a graphical feedback using the graphical user interface (GUI) known as the Vivado Integrated Design Environment (IDE), which allows to view the evolution of design starting from the IP declaration up to its implementation on a physical resource, or through Tcl commands that can be organized in a script or launched interactively with the advancement of the design. When Tcl commands / scripts are used, they can be passed to the tool either through the Tcl console present in the IDE or using the Vivado Design Tcl shell suite. It's possible to use Tcl scripts to run the entire design flow, including design analysis, or to run only parts of the flow. Vivado Design Flow complies with the industry standards that require:

- Tcl
- AXI4, IP-XACT
- Synopsys design constraints (SDC)
- Verilog, VHDL, VHDL-2008, SystemVerilog
- SystemC, C, C++

2.3.1 RTL-to-Bitstream Design Flow

The starting point of the design flow consists in specifying the RTL sources files; supported format are Verilog, VHDL, SystemVerilog and XDC (design constraints). Each project can support multiple source file types. An IP can be logic circuit, embedded processors, digital signal processing (DSP) modules or C-based DSP algorithm designs. When an IP has to be integrated in a project as standalone module or within the context of the system-level design, then Vivado IP Integrator environment is called which facilitates the packaging of the IP according to the IP-XACT protocol and then made available through the Vivado IP catalog. Xilinx IP utilizes the AXI4 interconnect standard to enable faster system-level integration [9]. In particular is possible to configure and connect IPs using a block design style interface and easily connect entire interfaces by drawing connections similar to a schematic or using connection automation features provided with a set of DRCs (to ensure proper IP configuration and connectivity). These IP block designs are then validated, packaged, and treated as a single design source. After the block design has been generated, design flow may proceed with the logic simulation aimed to test the behavioral correctness of the design. Of course, in order to test any design, testbench need to be written and set as top module in the simulator. After parsing and compiling the design, if no error occurs, a waveform window opens showing the top-level HDL objects. Before going ahead with the Synthesis of the design, is possible to assign logical and physical constraints to each design resource with flexible granularity, from the whole IP to the individual LUTs in which it will be mapped during the Synthesis process. About physical constraints, these define pin placement and absolute or relative placement of cells such as block RAMs, LUTs, Flip-Flops, and device configuration settings. Synthesis is the process of transforming an RTL-specified design into a gate-level representation. Input of the process is the HDL description of the design and it returns the netlist as output. By default, the Vivado Design Suite uses an out-of-context (OOC) or bottom-up design flow to synthesize IP cores from the Xilinx IP Catalog and block designs from the Vivado IP integrator. Vivado let user set synthesis constraint about strategy to be adopted by the tool, as for example if the synthesis must follow an area optimization directive or timing optimization strategy or some of the other available options about power consumption optimization. After the design has been synthesized, by running the command *open synthesized design* either from the GUI or adopting the equivalent Tcl version, is possible to inspect the Synthesis output. From this perspective the design logic and hierarchy can be observed, as well as the resource utilization (like how many LUTs have been used) and is possible to perform timing estimation or run design rule checks (DRCs) which in general is a good practice before going on with the design flow.

Following step in the Design Flow is the Implementation, which refers to the process that transforms the gate netlist, produced as output of the synthesis, and constraints into a placed and routed design. The implementation process performed by Vivado Design Tool walks through the following sub-processes:

- *Design Optimization*: the logical design is optimized in order to fit the target Xilinx device
- *Power Optimization*: optional step that can be enabled to reduce the power

demands of the board.

- *Placement*: each logic block required by the circuit is mapped into a physical resource available in the FPGA. The choice of a particular resource with respect to a completely equivalent one may be driven by the need to minimize the required wiring (wire-length driven placement) or to balance the wiring density across the FPGA (routability-driven placement) or to maximize circuit speed (timing-driven placement).
- *Post-Place Power Opt Design*: optional additional optimization can be performed to reduce power after placement.
- *Post-Place Phys Opt Design*: once the placement has been completed, timing estimation is conducted and used to optimize both logic and placement, including replication of high fanout drivers.
- *Route Design*: a router assigns to the circuit networks the interconnection resources available on the FPGA.
- *Post-Route Phys Opt Design (optional)*: final refinement on logic, placement, and routing can be enabled; this will take into account actual routed delays.

Once the design has been successfully implemented, reflecting the custom constraints and target specifications, Design Flow proceeds with last step of writing the bitstream and programming the device with the current design configuration stored in the bitstream. By default, the *write_bitstream* Tcl command generates a binary bitstream (.bit) file, however the bitstream format and content can be customized if necessary.

2.3.2 Software Development

In case the design realized should use the ARM cores available in the Xilinx boards, then Xilinx provides a software development environment, called *Vitis*, to facilitate this point. Xilinx Vitis is able to understand the software requirements of the custom hardware design that has been defined in the Vivado Design Suite. Based on this design, the tool automatically sets a number of different parameters such as memory maps, peripheral register settings, tool and library paths, JTAG and flash memory settings, debugger connections. Basic requirement under this automation process is the hardware extraction from Vivavo Design Tool, which is performed by running the Tcl *export_hardware* command which will produce a file .xsa containing the HW configuration of the core used in the design. On the Vitis end, by simply creating a new application project, importing the hardware platform (.xsa file) and writing some C code, its possible to quickly develop and run software application on the customized hardware design. Software application is typically provided as .elf file and should be used together with the bitstream file to program the board.

Chapter 3

Radiation effects on FPGA Devices

3.1 Radiation effects

Space is a narrow environment not only to man's life but also to the electronics he produces; the main problem that a electronic designer has to face is to guarantee the reliability of the product when it has to be employed for space applications. Reliability is a defined as the ability of a system to perform its intended function for a given period of time in a stated environment without failure [10]. In case of space application, the mission length is about years and often without possibility of human assistance, so that reliability becomes a crucial point defining the quality of a device, since most of the time on-field adjustment are not possible. In space, the reliability of any electronic component is jeopardized by the high presence of radiation. These radiations are not all of the same nature but are classified according to their source: the radiations produced by the Sun, the radiations called *cosmic rays* located outside the solar system and finally those that are trapped by the Earth's magnetic field (mainly protons and high energy electrons). Cosmic rays are typically made by heavy ions and protons that presents high energy profile which can deposit enough energy within a sensitive node that an integrated circuit

(IC) can be upset. These single event upsets (SEUs) are orders of magnitude greater than soft error rates on the ground, but totally analogous. A soft error is defined as that event for which a data or signal assumes an incorrect value that will not damage the system hardware; as the only damage is a wrong value on the data that is being processed therefore, a reboot or simply processing again same data can solve the error. When SEU takes place in space, if the particle has enough energy, then its passage through a critical device region can even lead to permanent failure of an IC due to single-particle-induced latchup [11], burn out, or dielectric/gate rupture [12] and in this case it is referred to as hard error. Solar radiations are a mix of protons, ions, neutrons, gamma rays and represent the main cause of damage to electronic components because they occur in a massive way during solar events where not only their energy is higher, but they are also present in greater quantities, to the point that protons are deposited on sensitive electronics, in the days or weeks following a solar event, with an amount equal to that of years of ordinary operation in space.

3.1.1 SEU: single event upset

All those phenomena that affect electronic circuits due to the interaction of energetic particles with the Si substrate are grouped under the acronym SEU. The peculiarity of these phenomena is that they cause an increase in charge in the transistors, which can lead to changes in internal voltages that in turn can alter the state of the stored or transmitted data. From the electrical point of view, what a SEU can entail is: temporary loss of data in the memory elements, corruption of the logic waveforms and/or latch-up in the transistors, i.e. the phenomenon whereby a low impedance path is created between power supply and ground. The errors associated with SEU usually do not cause system destruction and they can be recovered by rewriting the data as they simply turn into a bit-flip. To illustrate what the occurrence of an SEU in a memory element implies, the behavior of a static CMOS cell (the typical 6-transistor SRAM structure) can be analyzed. When the cell is not involved in a reading or writing process, it is disconnected from the bit lines and retains the state into which it was last written, that can be either 0 or 1. The transistors are organized in two branches, where at each branch a pair of MOS (p and n) implement an inverter. For each branch, one transistor will be turned off. When a particle hits the memory element, causing an increase in charge that turns on one of the transistors off, then, in the side of the transistor that is turned on there will be an increase in output voltage, which if higher than the toggling threshold of the technology, can trigger a regenerative action by toggling the other inverter into a change of state: radiation manifests as bit-flip.

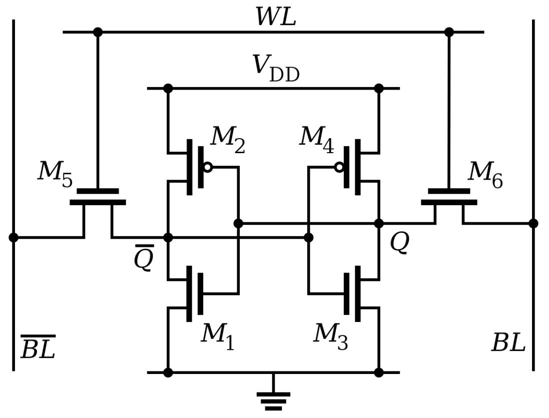


Figure 3.1. SRAM Cell : 6 transistors structure

Dynamic memories are also more sensitive to this type of phenomena because their storage mechanism is based on the retention of charge in small capacitors that, by their intrinsic nature, lose the stored information over time and therefore rely on the use of refresh mechanisms to ensure that the information is not lost. When a particle hits the dynamic memory element, if strongly ionized, it may be able to totally discharge the capacitor so that at the next refresh operation it appears to be fully discharged and thus the memory error will be preserved.

3.1.2 SEU on FPGA

When the radiation hits application-specific integrated circuits (ASICs), the resulting errors are bit-flips that propagate through the circuit and can be intercepted by memory elements, generating at most wrong computations. In the FPGA scenario, the configuration memory is certainly the most vulnerable element to SEU since, when a bit-flip occurs in the configuration memory this results in an actual modification of the logic circuit implemented[13] [**23**] [**21**]. Here the key difference compared to ASICs, as in programmable logic the errors are structural and require reconfiguration to be resolved. As reported in the Design Flow section, any circuit being implemented on FPGA is made by CLBs which are configured to implement a certain logic function and that are interconnected by means of programmable interconnections. All this *programmability* implies that somewhere there are bits stored related to each CLB status (content of the LUT, operating mode) and to each routing resources. These bits compose the configuration memory which store the binary information to implement a specific design on the target FPGA resources. The consequences of a SEU on FPGA can be either the alteration of the logic function implemented by the LUTs or their behavior since they can also be used as distributed ram, or it can influence the routing resources and therefore modify the net paths [14]. The faults models that derive from this scenario are:

- *Antenna fault*: The path, made by several routing resources, is also connected to a floating segment, so delivering unknown value to the output end.
- *Open fault*: The path is interrupted at a certain point, associated to the configuration bit affected by SEU, creating an unconnected signal.
- *Conflict fault*: Two signals are connected together creating a short circuit leading to undefined output value.
- *Bridge fault*: It affects multiplexer selection signal, that, when SEU occurs on selection bits, wrong input is delivered at the output.

3.2 Fault Tolerant Design Technique

At the basis of the fault-tolerant design [15] concept, there is the *resignation* to the fact that faults are events that happen and that the probability that a fault occurs increases over time. Having in mind this assumption, fault tolerance design aims to reduce the impact of faults during the operative life of product to guarantee systems which deliver an accurate service, even in the presence of active faults. Faults can be either masked or detected: when masked system will perform correctly even if in presence of an error but error is not detected and removed, while in case of fault detection the system realize that a fault has occurred and therefore can prevent faults from accumulating. Fault-tolerant circuits can be obtained by means of two approaches: either using hardened technology or by hardening the design.

- *Hardened Technology* It consist into implement electronics with ad hoc technology modified either at process level (transistors are shielded during the fabrication process) or at structural level (redundant method is applied to logic cell/gate, i.e using more transistors than required, so that they are structured in such away that the same charge particle cannot strike with different transistors of the same cell). This method increases the device reliability as well as its cost, so that taking into account also the fact that performance level achievable by hardened technologies is not so high, summarily the method is not so convenient.

- *Hardened by Design* This second approach consist into applying hardware redundancy, i.e replicate modules. Redundancy may be applied at different level such as system level, where for example three separate microprocessor boards may independently compute an answer to a calculation and compare their answers and any system that produces a minority result will recalculate. Logic may be added such that if repeated errors occur from the same system, that board is shut down. Redundant elements may be used at the circuit level, for example triplicating a critical section of the circuit as the ALU of a microprocessor or at data level, i.e using additional parity bits to check for and possibly correct corrupted data.

- *Hardware Redundancy* Modules are replicated N times, where N depends on the reliability level to be achieved. Modules can be exactly equal or just equivalent from functional point of view. Hardware redundancy can be either active or passive. When passive approach is adopted, after the module replication a simplified majority voter is adopted to dermine the output result. The modules will be fed with same input X, each producing an output value Yi that in turns are given as input to the voter. This approach ensure a correct output in case a fault affect one block at a time; on the contrary, if two faults occur, each on a different module, at the same instant, leading to the same incorrect output, this will be recognized by the voter as the correct output even if it is not! Other problems arise when faults affect the gate driving input X, or the voter itself. In these cases, the output will not be correct regardless of whether the replicas are working properly. In case of active redundancy, the module is replicated and the voter implements a comparator, whose output signal drives a latch. When the comparison determines that the outputs are different ,and that therefore a fault has occurred, the latch does not sample the output value and typically the system is stopped, a test routine is run, the fault detected and eventually removed, avoiding fault accumulation.
- *Information Redundancy* This technique is used to detect and correct faults that occur in stored data or in communication channels. It may be used in parallel with the HW redundancy. It commonly consists of adding some check bits (parity or hamming code) to the stored or sent data. Then, at the receiver end, same function used to generate the check bit is applied to the received data and check bits produced and received are compared: in case of a mismatch, the data is typically corrected.
- *Time Redundancy* Redundancy is applied at software level: an operation, that can be a critical application, a single instruction or a procedure,

is performed twice, with same input values but at different time. The outputs of the two executions are compared and in case of mismatch, the operation can be executed a third time. With this method there's no HW overhead but you pay in performance since the execution time is doubled.

Chapter 4

Analysis and Hardening of the RI5CY design on FPGA

As anticipated in the previous sections, the object of interest of this thesis work is the implementation of the RI5CY processor on FPGA for aerospace applications. Its use in Space environment requires that the design must be subjected to testing procedures to verify its correct functioning when faults occur. Being implemented on a programmable logic platform, this also implies differences on what may be any criticality due to the radiation effect with respect to an ASIC implementation[16]. In particular, it is useful to remember that for designs implemented on FPGA, the most vulnerable part is the configuration memory which contains the binary information that configure the FPGA resources in such a way as to implement the target design. Conventionally, there are two approaches that can be adopted to characterize and verify the fault tolerance capability of a design on FPGA: irradiate the device (expensive choice) or simulate the effect of radiation causing bit-flips in the configuration memory[**27**][18]. The fault-injection method, inducing a bit-flip, is the most commonly adopted solution. However, despite the advantage of being a low cost solution, it presents some controversies regarding the choice of the bits in which to inject the fault. The ideal choice would be to inject on all the bits that make up the configuration memory but this would cause an enormous time effort;

an alternative could be to inject in a statistical way but also in this case there is the risk of wasting time and resources to inject in bits that make up the configuration memory but which are not used to implement the design. So, the main problem, when using the fault-injection method, is to identify regions of the bitstream into which it is reasonable to inject, taking into consideration the fact that the difficulty is greater the less the documentation provided by companies on the content and organization of the bitstream, i.e. how the bits are associated with each physical resource. In this scenario, displaying the design on the device view can be of great use, so as to understand the device structure and consequently where the DUT components are mapped. Regarding this, the first approach to *PULPIssimo* was to identify any critical areas created by the place and route algorithm adopted by Vivado Design Tool. The idea is that the areas of greatest risk are those in which different modules, which are unrelated and functionally independent, are mapped together, i.e share the same CLB or even the same SLICE. In this context, it is likely that a fault in one of the two modules will also cause a misbehavior in the other module which is not directly affected. However, there is no much information that can be extracted directly from the tool on the post-implementation design, that is the tool is unable to provide information on the occurrence of these overlaps and/or on the resources and modules involved. This has led to the need to study the FPGA structure and to replicate its environment in order to have a more complete view of how the design was arranged on the resources, and resulting in greater control of the placement, should changes be deemed necessary.

4.1 Development Of The FPGA Equivalent Enviroment To Observe And Control Any Design

The Artix-7™ FPGA contained in the Nexys Video board is organized in 10 clock regions, which are arranged in a 5x2 matrix structure, i.e each row consists of two adjacent clock regions. Each clock region is organized into CLBs and each CLB is made up of two SLICEs. In the nomenclature adopted by Xilinx, each SLICE is identified by a pair of x and y coordinates; an accurate analysis revealed continuity in the assignment of the values of x and y to the SLICEs among the different clock regions: even when black spot are met in the device structure, where no SLICE are physically present, the tool pretends that it is filled with *transparent* SLICEs so that the assignment, for the SLICEs following the spot, will continue normally as the spot was not empty. Therefore, for simplicity, we can assume the entire structure as a matrix of 250 rows and 164 columns where the matrix cell correspond to a SLICE identified by x and y coordinates that grow with number of rows and columns starting from the bottom of the structure(SLICE_X0Y0 will correspond to the first cell starting from the bottom left, in clock region X0Y0). Then, each SLICE is made up of four 6-input LUTs, carry logic and 8 flip-flops. The choice of the SLICE as unit of measurement is consistent with the information provided by the tool related the position assigned to the design modules, after the place and route. The knowledge of the location of each module is extremely useful in identifying whether there's a possible critical situation or not.

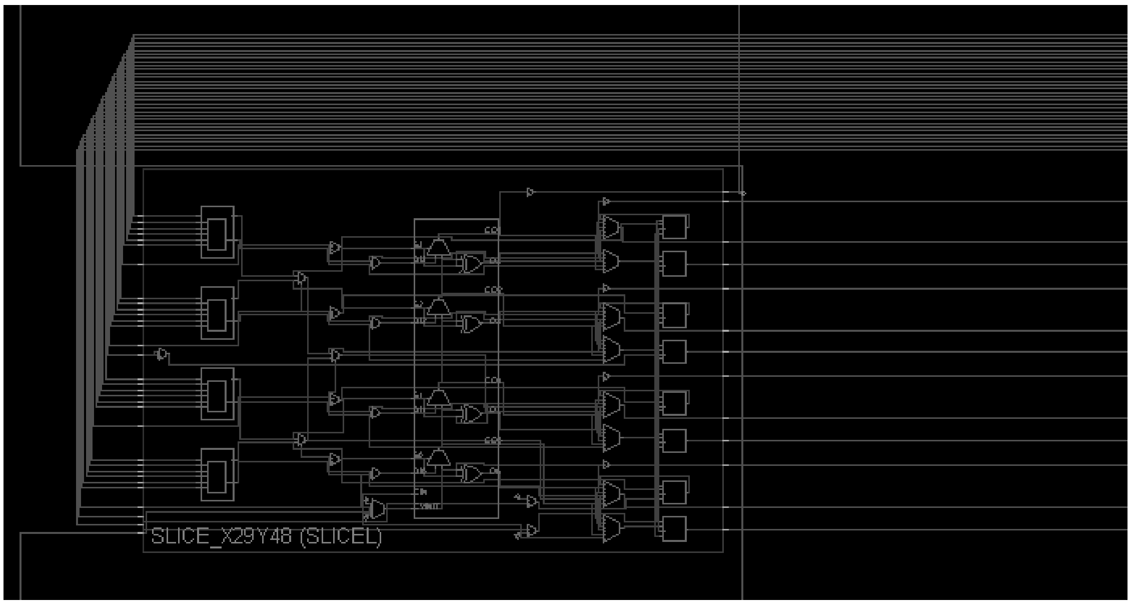
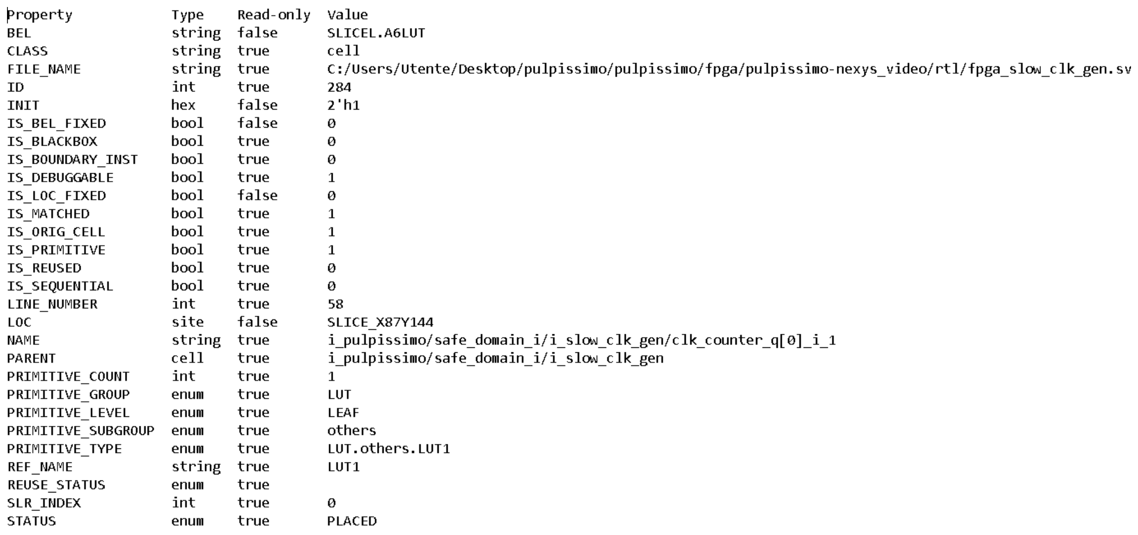


Figure 4.1. A SLICE internal overview

4.1.1 Design Information Extraction

In order to identify overlap occurrences, as first step is necessary to extract the physical location assigned by the tool to each component that constitutes the design. This information can be obtained if you have a post place and route design available (otherwise you have to proceed by executing all the steps of the design flow), thus by opening this design and running a Tcl script on it where the commands supported by Vivado are used. The idea is to adopt a general approach, which does not require a priori knowledge of how current design is made, of its hierarchy or the modules that constitute it, but these are information that we want to extract in an automatic way by processing the data that Vivado provides. To do this, it is possible to do an extraction of properties, related to what Vivado Design defines as *primitive type*. Primitive type is a resource which is directly recognized by implementation software and typically corresponds to a logic resource in a target FPGA. Vivado Design distinguishes five primitive types which are LUT, BMEM, FLOP-LATCH, DMEM, CARRY. So running the Tcl *report_property* command, filtering by primitive type, we get a list of properties for each resource used by the current design.



```
Property            Type    Read-only  Value
BEL                 string  false      SLICEL.A6LUT
CLASS               string  true       cell
FILE_NAME           string  true       C:/Users/Utente/Desktop/pulpissimo/pulpissimo/fpga/pulpissimo-nexys_video/rtl/fpga_slow_clk_gen.sv
ID                  int     true       284
INIT                hex     false      2'h1
IS_BEL_FIXED        bool    false      0
IS_BLACKBOX         bool    true       0
IS_BOUNDARY_INST    bool    true       0
IS_DEBUGGABLE       bool    true       1
IS_LOC_FIXED        bool    false      0
IS_MATCHED          bool    true       1
IS_ORIG_CELL        bool    true       1
IS_PRIMITIVE        bool    true       1
IS_REUSED           bool    true       0
IS_SEQUENTIAL       bool    true       0
LINE_NUMBER         int     true       58
LOC                 site    false      SLICE_X87Y144
NAME                string  true       i_pulpissimo/safe_domain_i/i_slow_clk_gen/clk_counter_q[0]_i_1
PARENT              cell    true       i_pulpissimo/safe_domain_i/i_slow_clk_gen
PRIMITIVE_COUNT     int     true       1
PRIMITIVE_GROUP     enum    true       LUT
PRIMITIVE_LEVEL     enum    true       LEAF
PRIMITIVE_SUBGROUP  enum    true       others
PRIMITIVE_TYPE      enum    true       LUT.others.LUT1
REF_NAME            string  true       LUT1
REUSE_STATUS        enum    true
SLR_INDEX           int     true       0
STATUS              enum    true       PLACED
```

Figure 4.2. Tcl report property command output example on LUT

As shown in the example reported in the figure 4.2, among the property the ones of interest are:

- *LOC*: It reports the physical resource coordinates as SLICE_XiYj.
- *BEL*: It refers to the physical LUT used in the SLICE and if it is used as 5-input or 6-input LUT.
- *NAME*: It's the name of the logical cell mapped on that LUT
- *PARENT*: Name of the module to which this leaf cell belongs.

It is useful to note that both the name of the leaf cell and the module to which it belongs, which Vivado refers to as parent, are provided as a hierarchical path. After pulling out only the properties of interest and organizing them in two files (for each primitive type), one of which contains the cell name followed by its position (SLICE and BEL) and the other only the parent names, following step is making a fictional FPGA environment.

4.1.2 FPGA Equivalent Enviroment

Vivado Design Tool offers limited controllability of the design, which therefore must be achieved through the back door. The idea is to create an environment that gives both the ability to view where the various modules are located and that indicates where are the overlaps of different modules and in addition allows to change the placement. This environment was reproduced in Python, creating a two-dimensional array of objects, with the same size as the target FPGA structure. This object reproduces the structure of the SLICE, in particular it has attributes that refer to the LUTs, an attribute that indicates whether this SLICE is used by the current design, one that indicates if all its resources are occupied and finally one that informs about which modules are mapped into it. Initially the environment is presented as empty, so that all the attributes of each SLICE are initialized to zero, thus it requires to be filled with information extracted from the design. This is done using the two files arranged during the information extraction phase described in the previous paragraph. Since the parent modules are provided as a hierarchical path, this gives the possibility to choose the hierarchical level at which to stop the discrimination among the various modules, i.e given for example the two parents *pulpissimo/safe_domain/slow_clk_gen* and *pulpissimo/safe_domain/slow_clk_gen/slow_clk_manager*, if 4 is indicated as hierarchical level up to which discriminate, then the leaf cells belonging to first parent path will be marked with different color with respect to the one of the second, even if they belong to same macro block. Summarizing, to map any design on this environment it is necessary to provide the two files containing the information on the design (the one in which all the cells are listed with name and position, and the one with all the parent modules) and the hierarchical level of interest. In addition, it will be necessary to indicate the output file in which the occurrence of the overlaps will be reported: for each overlap found, the modules involved and the slice are reported. The figure shows the *PULPissimo* mapping on the environment: for each parent module, the tool automatically assigns a color and a numerical value, which are used both to create the plot and to discriminate if two or more LUTs belonging to the same SLICE are leaf cells of different modules. If so, the SLICE in question is marked in red in the plot, the values assigned to the LUTs are read and translated into the correspondent parent module and these, together with the SLICE's coordinates are written to the output file.

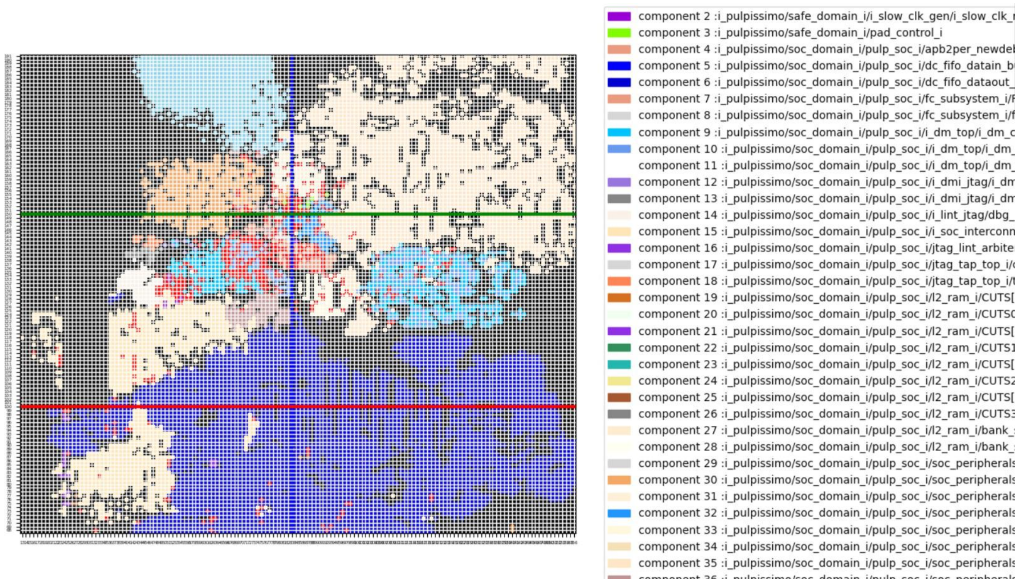


Figure 4.3. *PULPissimo* design plot on the FPGA equivalent enviroment with architectural depth equal to 5

As feared, Xilinx's Vivado tool maps different modules in the same SLICE, making these sites vulnerable to faults. For example, with a hierarchical depth of 5, the number of SLICEs with at least one overlap is 475 and this number increases as the number of hierarchical levels increases, since the tool discriminates more the modules and their leaf cells. The following table 4.1 shows the result obtained by mapping the same design with different architectural levels.

Hierarchical Level	**Overlaps**
5	475
7	945
9	1234

Table 4.1. Overlap count on different hierarchical level

The choice of the hierarchical level in which the design is to be analyzed is crucial since the highest value is not always the optimal one: trying to discern the modules present in the design as much as possible, could lead to the fact that leaf cells belonging to the same module but declared in different sub-blocks, trigger an overlap because they are mapped to the same slice, which is reasonable since they implement complementary functions. So it would not make any sense to think of moving these cells, on the contrary doing so would have negative effects on performance.

4.2 RI5CY ALU Hardening

Having identified some unpleasant situations due to the placement algorithm adopted by Vivado Design, the next step is to evaluate how much the placement actually affects the reliability of the design. Wanting to kill two birds with one stone, the placement analysis was performed on a RI5CY module that was extracted from the processor context and to which the Triple Modular Redundancy (TMR) technique for fault tolerance was applied, with the aim of reinserting it back into the RI5CY hierarchy once it had been hardened.

4.2.1 RI5CY ALU

The Arithmetic Logic Unit present in RI5CY, has been designed to be performant, so it is able to execute instructions that are not typical of the basic instruction set of RISCV. Described in SystemVerilog, like the rest of the project, it is defined as a behavioral model, i.e. every function of the design is described as a set of concurrent algorithms. Within the ALU of the RI5CY, there is a partitioned adder, typically used in datapaths for multimedia signal processing. These adders have the characteristic of being run-time reconfigurable to operate on simple integers with 8, 16 or 32 bits of precision. In particular, being partitioned, execution occurs in parallel, so the circuit executes one 32-bit, two 16-bit, or four 8-bit additions depending on partitioning signals. In addition to the modules typically supported by ALUs to perform comparisons, shifting and bitwise operations, this ALU also supports bit manipulation operations, bit count operations and bit reverse operations. Most of the operations available for integers also support vectorized data. When vectorial instructions are executed, operations are performed in a SIMD-like manner on multiple sub-word elements simultaneously. This is done by segmenting the data path into smaller parts when 8- or 16-bit operations are to be executed. Vectorial instructions come in two versions: - 8-Bit, to perform four operations on the 4 bytes within a 32-bit word at the same time - 16-Bit, to perform two operations on the 2 half-words within a 32-bit word at the same time. Understanding how the ALU works, what are the information that it must receive as inputs, their format and meaning, the way in which these are processed to provide the final output, is essential both to create a test routine that is able to stimulate it adequately during the campaign of fault injection, both to replicate the behavior and create a gold reference with which to compare the results from time to time after the injection.

4.2.2 TMR on RI5CY

Once identified, extracted and analyzed in its behavior, the ALU of RI5CY has been hardened by applying the TMR technique: keeping the original interface of the ALU, so as not to have difficulties of reinsertion within the processor, a new entity has been created, within which three independent replicas of the ALU have been instantiated, all fed with the same inputs but generating independent outputs, and a majority voter that receives these replicas' outputs as input and uses them to generate the final output to be given to the next stage. The voter is implemented by a simple combinatorial function with three inputs and one output, which adopts bitwise operation to determine the correct output. Such a function is $V = XY + YZ + XZ$, where X, Y, Z are the outputs from the three replications and where product implies logical conjunction, and sum implies logical disjunction (AND and OR operations, respectively). In simple words, the voter evaluates if two of the three inputs are equal and if so, the common value will be the value provided as final output. So, the idea is that even if one of the outputs is wrong cause one of the ALU is affected by a fault, this error does not propagate since it is covered by the actions of the voter who is able to produce the right output. However, for a matter of statistics, it has been added an output signal to the voter called *corrupted* that is set to 1 every time there is at least one of the three ALU that produces an output different from the others. Thanks to this signal, during the fault injection, even if the final result matches with the one produced by the golden reference, it is possible to realize that at least one of the 3 ALUs has failed.

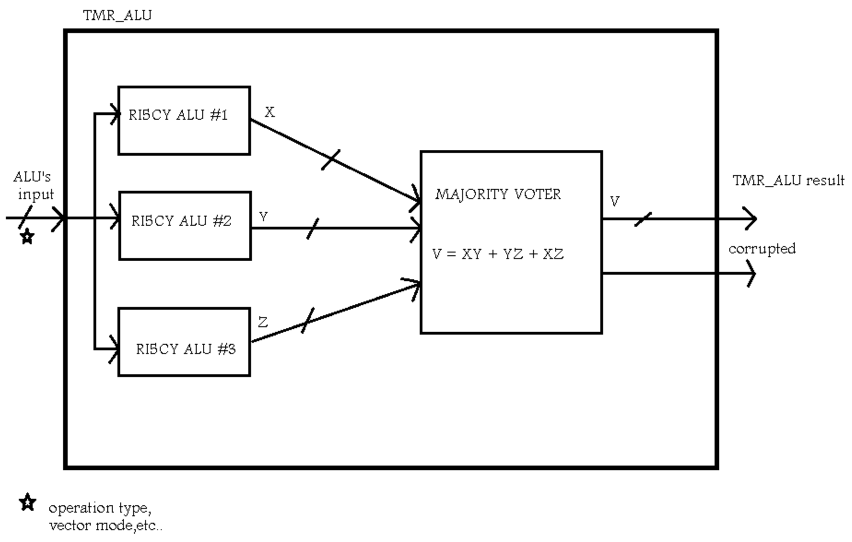


Figure 4.4. Triple Modular Redundancy applied to RI5CY ALU

4.3 Test Environment Design on Vivado Design Tool

So far, hardening of the RI5CY ALU has taken place outside of the Vivado Design hardware development environment. Once the structure has been modified, simulated by means of a testbench specifically created to verify the correctness of the connections and the functionality of the voter, it is now necessary to implement it on FPGA. Since the implementation of the TMR ALU on FPGA is aimed at performing fault injection campaigns to verify how reliable the design is, this module that we will henceforth call as device under test (DUT) must be placed in a hardware context that allows us to test it and to collect the data produced. Typically, what is needed is to have on the same board, both a processor core that stimulates the DUT and monitors its behavior, and the FPGA on which the DUT is implemented.

4.3.1 Hardware Context Development on Vivado Design Suite

Just as a reminder, Vivado Design is the official tool distributed by Xilinx to create any design and implement it on one of its FPGAs. Whenever a new project is created, the first step is to choose the board on which it should be implemented. In this way, the details about the FPGA that will be used, the available peripherals and cores and all the IPs available for that target board are available to the designer. In this phase of the thesis work, the board used is the PYNQ-Z2 which has both a processor and an FGPA. The easiest and most effective way to realize the context hardware in which to insert the DUT is to exploit the IP Integrator proposed by Xilinx to realize a block design by simply selecting IPs from a catalog and connecting them together as needed. In order for the hardened ALU to be inserted into the block design, it must be "converted" to a customized IP that will be inserted into the IP catalog and then selected and added to the block diagram. The customized IP, typically, is placed within an AXI4 interface. The Advanced eXtensible Interface (AXI) is designed for AMBA-based FPGAs as a protocol for inter-block communication IP. Interconnect IP is intended for memory-mapped transfers only. Vivado Design provides a tool called Create and Package IP Wizard, which can be selected from the tools menu. Once open, it offers a choice between packaging the current design into an IP or creating a new AXI4 peripheral. Since the hardware context requires a processor core to act as the master while the DUT is the slave, the optimal solution for time expenditure and complexity is to adopt the AXI4 peripheral method that comes as a base IP into which you must enter your HDL code. When the AXI4 alternative is selected to create the new IP, the number of registers it must have must be indicated. The registers of the AXI interface will then be used to pass the operands, the operation code and all the inputs necessary for the ALU to operate, and the ALU will provide the computational output which will in turn be written to one of the registers of the AXI. Being a memory mapped peripheral, the core processor has easy access to the AXI registers to both send inputs and read the output produced by the ALU. Therefore, the data exchange between DUT and processor core is trivially limited to a read and write of registers each of which has a well-known memory address. So, once packed into a new IP, the TMR ALU is inserted into the project's IP catalog. In this IP catalog you can find the processing system present on the board, and that will be the master in the testing routine. Below is the schematic of the block design.

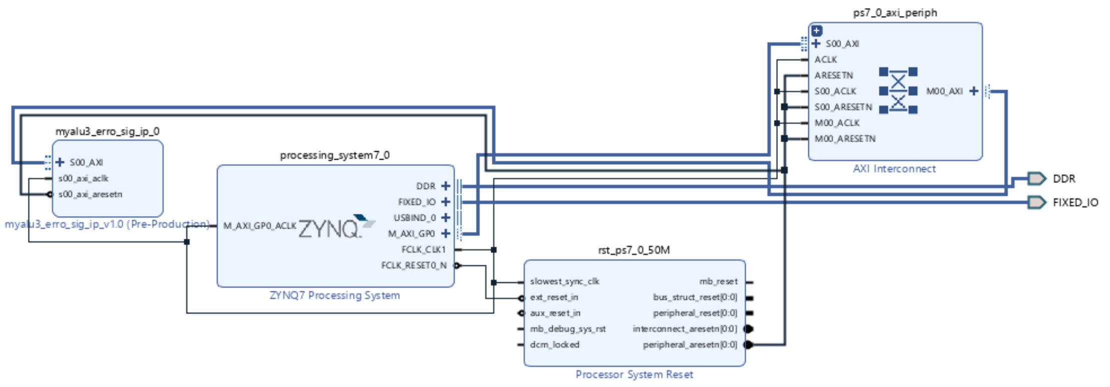


Figure 4.5. Hardware enviroment to test TMR ALU

4.4 Different Placement Arrangements

As anticipated in the previous sections, the main purpose of realizing an equivalent FPGA environment is not only to have more observability on how each logical cell is mapped on a physical resource, but also to be able to control its placement, i.e. change it according to the needs of the design. So, starting from the previously generated block design, we continue with the design flow expected for FPGAs and then after the block diagram elaboration phase, where the RTL source files (which are text) are read and bits of code representing real hardware structures are recognized, the flow continues by translating them into generic technology cells such as ports, comparators, muxes, registers, adders, to build a generic design netlist. This step is necessary because the next steps (high-level and low-level optimizations) are timing-driven, and thus need constraints; But the constraints cannot be applied to RTL description, they need to be applied to a netlist. At this time, being the first run of the design flow, it is not known a priori the result of the placement algorithm adopted by the tool, so it does not make sense to impose any constraint but rather it is necessary to create an untouched design to be used as a yardstick against the modified versions that will be made. The elaboration phase is followed by the synthesis and finally by the implementation, performed leaving the basic settings proposed by Vivado. Upon completion of the implementation phase, the design is provided as a post place and route and is observable in the Device View proposed by Vivado GUI. Since we are looking for situations in which independent modules share the same physical resource, it is useful to spend a first look at the placement from the Device View, highlighting with different colors the three replicas of the TMR structure. As you can see from figure 4.6, the placement algorithm goes against the idea of the TMR technique in which you need to have independent modules, distinct and separate, for it to work properly.

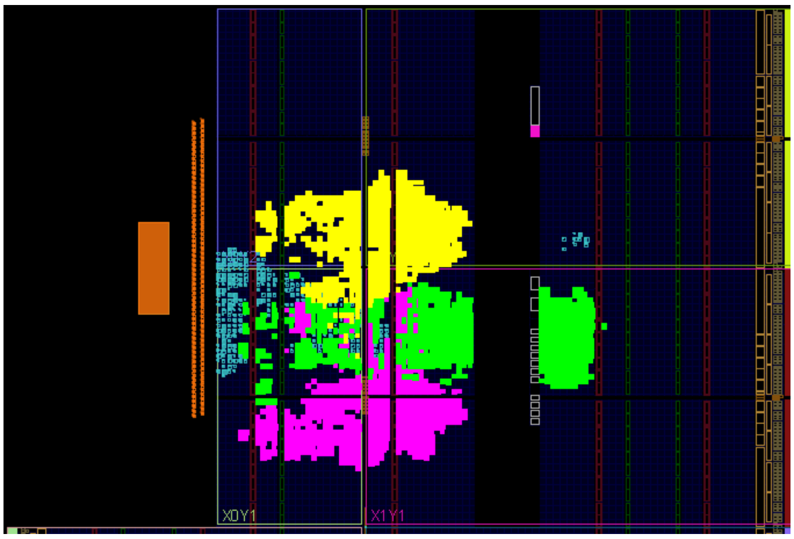


Figure 4.6. TMR ALU post place and route view without constraints.

The three ALUs are highlighted in three colors, while the non-illuminated blocks are those relating to the logic used to implement the IP and the connections between this and the processor core, which is represented by the orange-colored region on the left. Probably using performance as an optimization criterion, the reason why there is a concentration of overlaps in the central part of the placement is to be found in the need for proximity between the inputs and/or outputs of the three ALUs with the registers of the AXI interface and with the modules that define the majority voter. So, starting with this untouched design, the next step is to generate a design in which isolation is highlighted and another one in which the worst case is implemented. In order to build the two scenarios, the first step is to extract from the design all the information needed to remap it into the Python replica environment used previously. This information, again, is reduced to generating reports on the primitive types used by the design and classifying them according to the ALU they belong to, information that resides in the parent field of each cell that is extracted. The classification is done automatically by the tool, as simply distinguishes all the parent modules available, that in this case are just the three ALUs, the voter, the AXI interface and the Reset Unit (automatically generated during the block design phase). What facilitates the placement reconfiguration process is that the parent module is contained in the name Vivado assigns to the cell, so you can imagine the project as a list of cells each with its own unique name, which you are able to selectively pick according to the module you want to move, simply by using a string matching method. To this end, two functions have been created in Python which generate the Tcl commands corresponding to the two different placements.

The first function is the one that allows us to generate the design with isolation: this, having received the list of parent modules of the design, starts by counting the number of LUT cells associated with each ALU and taking into account the structure of the SLICE, which is able to map up to 4 LUTs, defines the edges of a placement perimeter. In this way, an ordered design is obtained in aligned rectangles, as shown in figure 4.7. Wanting then to extend the isolation also for the voter and the AXI interface, the same principle has been adopted by creating rectangular structures placed near the interface with the core processor.

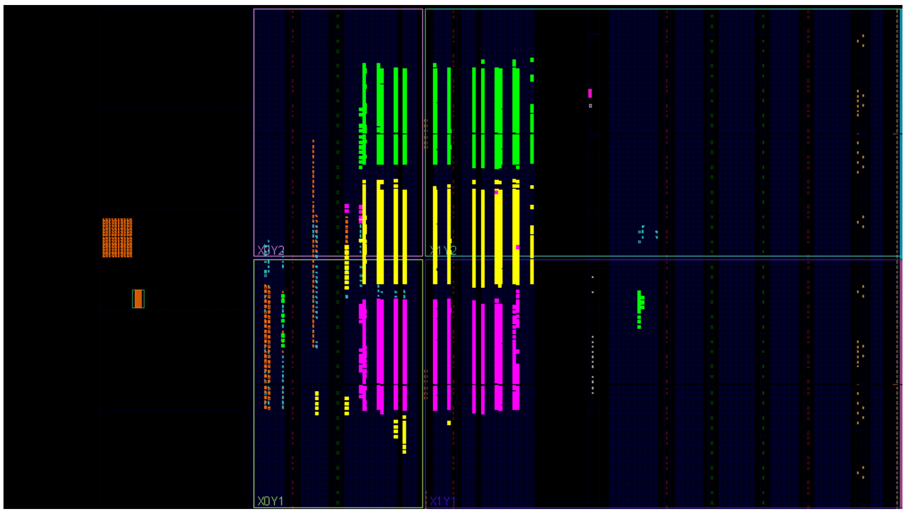


Figure 4.7. TMR ALU post place and route view with isolation constraints

The placement is imposed by means of the place_cell property which requires the name of the cell followed by the exact location defined by the SLICE coordinates and the BEL site. Thus, the output of the function is a Tcl file where for each cell in the project, the place_cell property is set which in turn the Vivado tool will convert into constraints. In order to keep the changes, the constraints must be applied on the synthesized project then before any placement procedure. So, once the constraints have been saved, the project must run the synthesis and implementation phase again. The worst case scenario, in which the three ALUs are totally mixed in terms of placement, has been achieved by means of a function that reads all the parent modules of the project and maps them one at a time by choosing the leaf cells and placing each of them in a different SLICE, trying not to assign the same SLICE twice to the same parent module. By doing so, the placement trend maximizes the case where in a SLICE the four LUTs are all linked to different parents, as it extends to all SLICEs. Obviously this second design has high congestion and very high critical path delay, so in order for this design to actually work on the board, it is necessary to reduce the clock frequency that the core processor delivers to the DUT, moving from the original value of 50Mhz to 25Mhz.

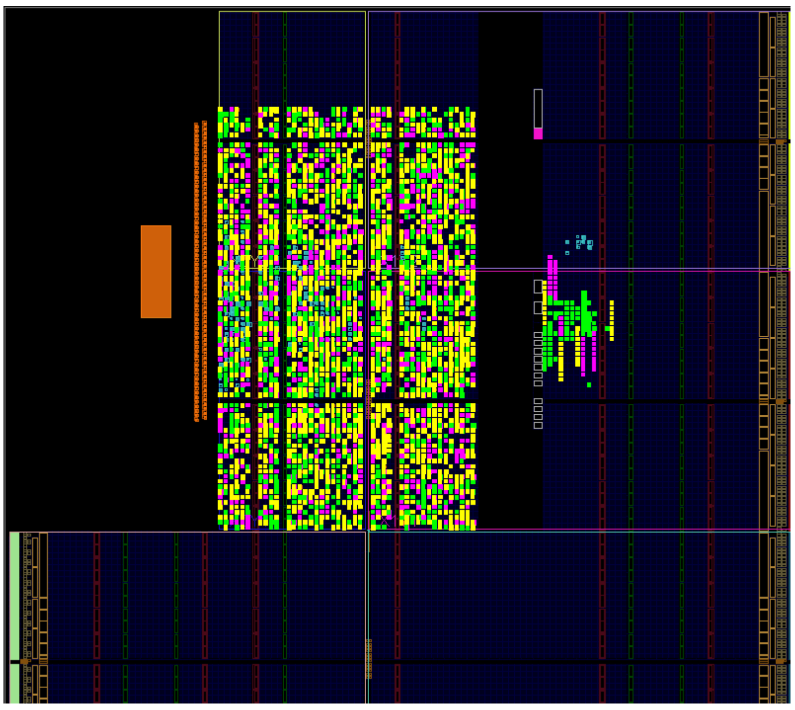


Figure 4.8. TMR ALU post place and route view with worst case scenario

4.5 TMR Voter

Until now, the majority voter has been considered infallible, which is surreal given that, being a pure combinational circuit without any kind of protection, it is just as prone to failure as any other logic. Indeed, if in the case of an error in one of the replicas, this can be masked by the voter and thus blocked, in the case of an error in the voter this will propagate to the next stage. For this reason a complete analysis of the TMR technique requires that also the voter itself is replicated and that the three independent outputs produced are delivered to the next stage, which in the current situation is defined by the processor core. Therefore, following the same steps previously adopted, a new IP has been created in which both ALU and Voter are tripled and placed in the AXI interface, which this time needs three output registers. The placement of this design has not been changed, as the TMR system arranged in this way is expected to be less susceptible to failure.

4.6 A Basic Router To Apply Isolation at Net Level

Isolated placement revealed that while we are able to easily enforce the exact location of each cell with LUT granularity, the routing mechanism adopted by Vivado Design assigns paths to networks that pass through regions that conflict with the isolation principle, thus making the TMR technique quite useless. In a nutshell, what we expected was that the routing resources allocated for each ALU replica would be distributed only in the region where the replica is defined, what we observed is that due to the high convergence of the outgoing networks towards the voter, most of the networks belonging to a replica, in order to reach the voter module, pass through the placement of one of the other replicas. Some might wonder where the problem lies. The fact is that in FPGAs, the path is defined by programmable interconnections, often called switch matrices, which are configured by setting specific bits in the configuration memory. If, for example, the same switch matrix is used by two nets belonging to different modules, a bit flip in the bitstream region associated with that interconnect resource will cause an error on both nets. In the case of TMR, if the two nets in question are the output signals to be delivered to the voter, it is very likely that the final result will be wrong.

Vivado Design provides a tool that allows to manually re-route a net selecting each time the next node on the route from a list of available nodes that updates every time a new node is picked and added to the path. The process is facilitated by the use of the graphic interface that has the advantage of a visual feedback on each move and resource used, so there is a concrete idea of the progress of the routing and the direction taken from time to time. Obviously, in the case of customizing a small number of routes, the manual routing supported by Vivado is quite efficient and for each routing, by the time you start to understand the function of each node and the displacement it determines, without trial and error in route decisions, the time it takes is about 5 minutes. But when the number of routes increases, the time effort also increases and it is no longer feasible to use the manual routing tool. For this reason, there is a need for an external router that is able to automate this process and execute it on a list of nets, following imposed constraints.

4.6.1 Routing Problem on FPGA

Most of the FPGA area is dedicated to programmable interconnects and typically the delays associated with these are much greater than those of the implemented circuit logic, making routing one of the main performance-dependent problems in FPGAs. The fact that the routing resources are fixed increases the difficulty since the paths have limited flexibility and all design constraints must be satisfied in any case. In addition, the routing problem is architecture-dependent, as the router must have detailed knowledge of all the interconnection information of the target FPGA. In Xilinx FPGAs, the programmable CLB blocks are organized in a two-dimensional array structure, and between the rows and columns of this array, there are horizontal and vertical routing channels that allow them exchange data. The interconnection resources are divided into two types: connection boxes and switch boxes [17]. The task of the connection box is to allow access to the CLB by connecting its input and output pins to the channel wires. It is characterized by its flexibility, defined by the number of wires to which each pin of the logic block can connect, and its topology, i.e. the pattern of switches making the connection. Switch boxes, also called switch matrices, allow switching between vertical and horizontal wires. Its flexibility is defined by the number of wiring segments that each wiring segment entering the matrix can connect to. When the switch matrix is able to connect wiring segments of the same domains called planar and obviously has a reduced flexibility: that is, supposing we have two adjacent matrices A and B, and that the access nodes are indicated with a number, being planar means that the only lawful connection for node 0_A is the one with node 0_B, while connection 0_A - 1_B is not feasible. The wiring segments are also classified according to the connection length they provide:

- *single-length lines*: are those connections that span through one CLB only.
- *double-length lines*: the wiring segment spans two CLBs leading to lower delays.
- *long lines*: the wiring segment typically spans four CLBs, that makes it the appropriate choice when long connection is needed with low-skew.

In Xilinx FPGAs, each CLB has its own associated switch box which is identified by the same x, y coordinates as the CLB to which it belongs, as shown in figure 4.9 where the black box on the right is the switch matrix.

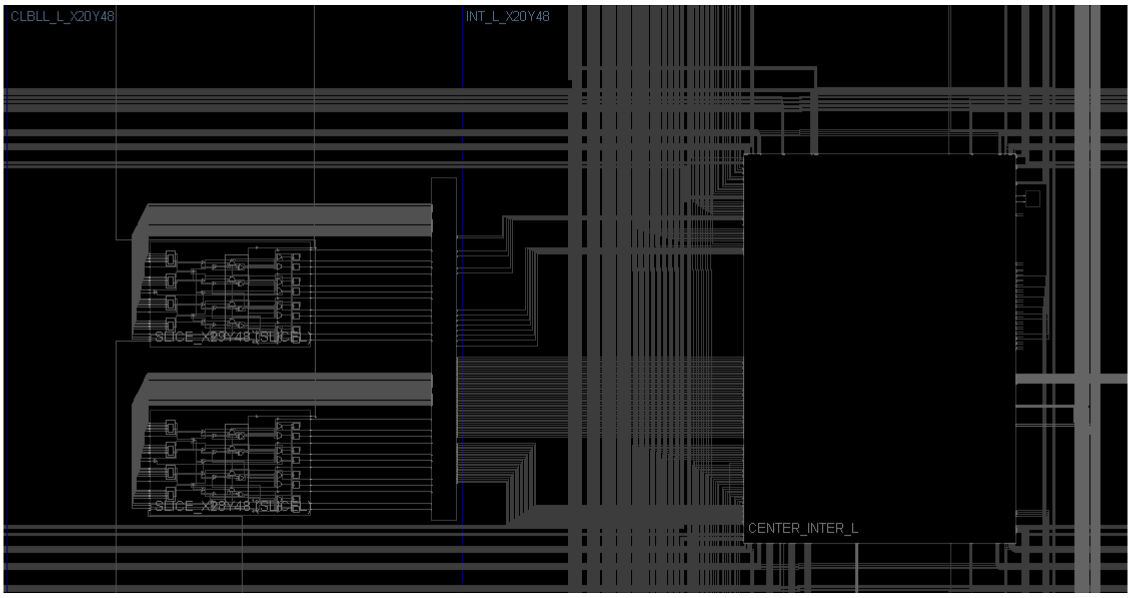


Figure 4.9. Configurable Logic Block view and its associated interconnections

4.6.2 General Routing Background

The routing procedure [19] consists of two consecutive steps: global routing and detailed routing, which are typically performed in that order. The task of global routing is to perform a raw routing whose purpose is to identify for each connection the path with the minimum distance. Among the minimum distance paths, further sorting is performed by selecting those that pass through the least congested routing channels. So let's say that its main objective is to find the optimal route to guarantee better performance, but also and above all to avoid congestion. Once all connections have been roughly routed, the solution is optimized and passed to the detailed router. Given a two-point connection, the detail router determines the specific wiring segments among those present in the routing channel assigned by the global router. Typically, this is done by constructing a graph to represent the connection using the various routing resources available in that channel, such as wires, connection boxes, switch matrices, and logical blocks. Each route has a cost function whose value takes into account the usage of each wire segment and the distance of the interconnection points. The optimal solution is the shortest path.

4.6.3 A Look Inside the Xilinx Artix-7 Switch Matrix

The difficulty in making a router lies in knowing in detail how the interconnection resources are made and the difficulty increases the more these resources are complex and articulated to achieve greater flexibility. The main problem is that there is no detailed documentation and the information must be extracted manually from the available tools and interpreted. So a lot of work, time and energy has been spent in understanding how the Artix-7 Switch Matrix works in terms of connections between wiring segments. The switch matrix can be viewed as a black box where a series of nodes are placed along its perimeter. Each node has associated some PIPs. PIP stands for programmable interconnect point and corresponds to a switch that connects two wires, controlled by SRAM configuration cell. In a few words, given a node, its pips correspond to a set of reachable nodes belonging both to the same matrix and to others, therefore compatible to be connected. An example is shown in the figure below: selecting a node, all its pips are displayed as highlighted segment where the direction is indicated by the arrow.

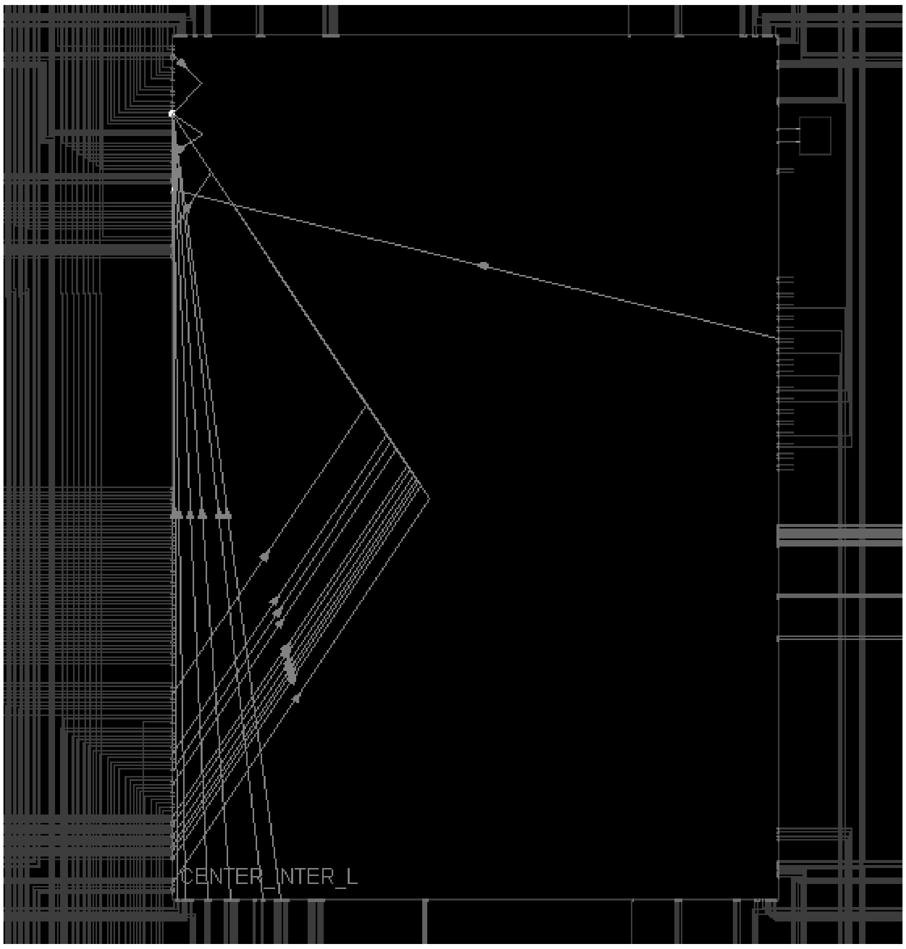


Figure 4.10. Switch Matrix view : node's pips example

In the FPGA Artix-7 for each S-matrix there are 3737 pips, hence the difficulty in implementing a router from scratch involving all nodes belonging to the matrix. Just to realize the complexity, figure 4.11 shows an S-matrix where all its pips are highlighted. Also, there is no immediate way to tell what kind of displacement determines the choice of a certain node over another, other than experience.

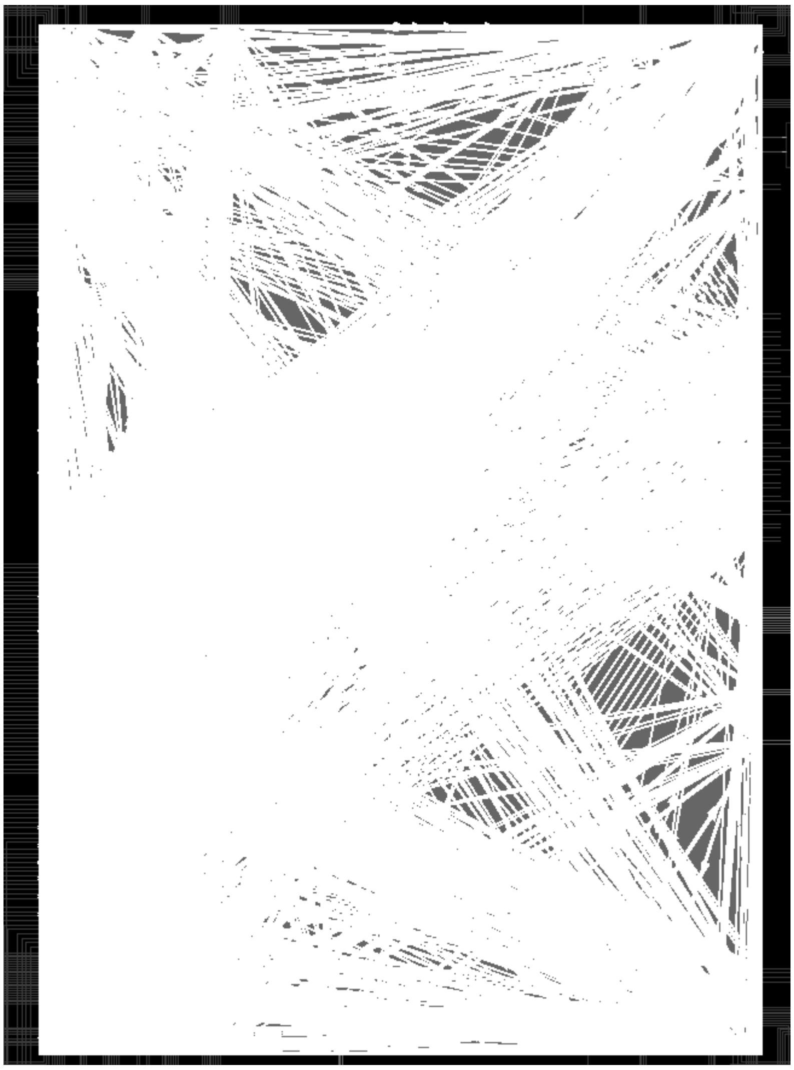


Figure 4.11. Switch Matrix view with all its pips highlighted

In aid to this, is certainly the manual routing tool of Vivado, which supported by the GUI, allows you to see for each selected node, those who are the reachable nodes and the movement that each choice determines (information extractable from the coordinates associated with the Switch Matrix to which the chosen node connects). The Xilinx matrix supports nodes that join single, double and long length lines. Due to the high time effort, currently 167 nodes behavior has been discriminated in detail, a sufficient number to realize a basic router.

4.6.4 The Router

The extracted nodes were classified and grouped according to the type of displacement they determine. Given the coordinates (x,y) of the starting Switch Matrix, the feasible displacements are:

- (x-1, y), (x-2, y), (x-4, y)
- (x-1, y-1), (x-2, y-4)
- (x, y-1), (x, y-2), (x, y-6)
- (x+1, y-1), (x+2, y-4)
- (x-1, y+1), (x-2, y+4)
- (x, y+1), (x, y+2), (x, y+6)
- (x+1, y+1), (x+2, y+4)

Nodes that cause the same type of displacement, for example all those that involve a shift to the east, regardless of offset amount, were grouped together to create macro groups that are: east, west, north, south. The remaining nodes, those that determine a diagonal shift, are used as a secondary resource, in case no node in the macro group in question is available. The realized router is based on a kind of database of text files: each node, identified by a unique name, has associated a file of the same name, where are listed all its pips (compatible nodes for a connection). Each node is also characterized by a function that reflects its properties: it receives the current coordinates of the Switch Matrix as input and returns the new target coordinates, which depend on the offset associated with the node. The node classification mentioned earlier was accomplished by means of four functions, where for each of them, a dictionary was instantiated; the dictionary's keys are the node names and the values are the node's function. Thus, for example, in the function associated with the west direction, all nodes that result in a reduction of the x coordinate populate the dictionary. In addition, displacements were further classified based on whether they occur horizontally or vertically. Since each CLB is associated with a switch matrix, and since CLBs are organized in a two-dimensional array, the router works in a matrix environment, where each cell, defined by a pair of coordinates (x,y), is an object that implements the basic version (given the reduced number of associated nodes) of the Xilinx FPGA switch matrix. As the router is to be used to customize the routing with respect to the one provided by Vivado, constraints can be set that refer to the extremes of a perimeter within which the routing is to be contained. So it receives input values of x_{max}, x_{min}, y_{max}, y_{min} which must be updated whenever the nets to be routed belong to different modules. On the basis of the isolated placement made and the relative position of the replicas of the ALU with respect to the voter, the routing rule adopted is that as long as the value of x is between x_{min} and x_{max}, corresponding to the extremes in which the parent replica is defined in the placement, the assigned path must be horizontal as long as the resources available make it possible. Then, past x_{max} value, the path must proceed vertically when feasible. It is important to emphasize "when possible" in the path constriction, because obviously if in the current switch matrix there are no nodes available that allow horizontal (or vertical) movement, then the path must necessarily continue vertically (or horizontally) for a segment, and then resume the initial trend. It is precisely when horizontal movement is not feasible that the y_{min} and y_{max} coordinates come into play, which, in vertical movement, determine whether the net should be routed up or down. The principle adopted is that in the case the current value of y is at least 6 units smaller than y_{max} (which we remember is the maximum coordinate in which the parent module is defined) then it is possible to continue upwards, otherwise downwards. Similarly, if the current value of y is larger than y_{min} of at least 6 units, then it is possible to continue downwards, otherwise upwards. The choice of value six as threshold is determined by the fact that it also corresponds to the maximum offset caused by "vertical" nodes. Considering that in Vivado the nets that have a fixed routing have higher priority with respect to the un-fixed ones, and that as soon as the constrained route is assigned, the entire design is routed again to avoid conflicts, it is not necessary to extract information about the routing resources used by the design and therefore at router startup the matrix environment can be left empty, with all its resources available. The start point and the end point to be reached are identified by (x,y) coordinates and a node name, which are information easily extracted from the original routing assigned to the net,. In particular, the start coordinates and name are those of the output node used to exit the CLB that drives the net, while the destination coordinates and node name refers to the CLB that will receive the net as input. In order to better explain how the router works, these coordinates will be referred to as (start_x, start_y) and (end_x, end_y), respectively. Also, the constraint that will be used in the path definition must be assigned (x_{max}, x_{min}, y_{max}, y_{min}). In the first part of the path, defined by x_{min} and x_{max}, the routing is chosen to move horizontally. This choice comes from the fact that in the specific case, it was decided to re-route the output signals of the ALUs that converge to the voter, and therefore wanting to apply the isolation principle, it is necessary that the path remains within the rectangular perimeter of the ALU to which it belongs. Thus, for simplicity's sake it is better to try to impose a path that extends horizontally in the length of the ALU. So, the first decision to be made is whether to go east or west. This choice is determined by the difference between end_x and start_x: if it is less than zero, then the choice of nodes falls on the group of nodes associated with the westward movement, otherwise on the eastward ones. Assuming that a hypothetical case requires to find a node that allows westward movement, then, given the current node (the one exiting CLB, if we are at the first iteration), the algorithm proceeds looking for the file associated to the current node, that contains all its pips, and seeking among the compatible nodes those that induce a westward movement. This search returns a list of nodes, in which only one node is selected randomly. Once the candidate node is taken, the algorithm continues with the following steps:

1. If the node is available, the following updates are made and then continue with step 2. Otherwise, go to step 3.

- last_node = current_node

- last_x = start_x
- last_y = start_y
- current_node = chosen_node
- (start_x , start_y) = node_name(start_x, start_y)
- The switch matrix at the start_x and start_y coordinate should be updated with the resource used.

2. Verify to be in the x_{min} - x_{max} constraint, calculate the new distance between start_x and end_x in order to determine if the displacement should be east or west. Depending on this, search among the nodes compatible with the current node those that determine the displacement in that direction, take a random one and go to step 1. If there are no compatible nodes go to point 4. If outside the constraint, the current node is passed as the start node for the vertical path.

3.If the node is not available, return to the list of nodes previously extracted and take another one, check availability and return to point 1 if yes, otherwise stay in point 3 until one is available. If, despite having tried all the nodes, there is not one available, go to step 4.

4.If the compatibility list is empty or all compatible nodes are not available then go to step 5.

5. Calculate the distance between y_{max} and start_y. If greater than six, then among the nodes compatible with the current node, look for the ones that allow upward movement, otherwise downward. From the list of nodes drawn, choose one randomly and return to step 1, otherwise stay at this point until an available node is found in the list. If the list runs out or is empty, go to step 6.

6. The vertical movement in the selected direction has not been successful, repeat the same steps of point 5 but with the nodes of opposite direction. If also in this case a node cannot be found then go to step 7, otherwise go to step 1.

7. The current node is a dead point that does not allow continuing towards any direction, it should be eliminated and excluded from the next choices by adding it

to a black list of tried but not useful nodes. The following updates are performed:

- picked_node.append(current_node)
- The switch matrix at coordinate start_x and start_y should be updated and the current node removed from the used resources.
- current_node = last_node
- start_x = last_x
- start_y = last_y

Next, take another node from the list of nodes compatible with the current_node, excluding those in the picked_node list, and return to step 1.

The same principle is adopted when the routing occurs vertically, i.e. when start_x is outside the constraint x_{min} - x_{max}. In this case, the north or south direction is determined by the current distance between start_y and end_y, which is checked at each iteration. If there are no nodes available to continue in this direction, then, for a stretch, the choice falls between the nodes that allow horizontal movement, and also in this case the choice between east and west is determined by the distance between end_x and start_x. Once found the node for horizontal movement, continue vertically until distance from end_y, in absolute value, is less than or equal to 2. At that point it is possible to:

- A. continue horizontally trying to reach end_x
- B. if point A is not successful, the routing can be left incomplete because the Vivado routing process will complete it starting from the last imposed constraint.

The route is returned as a list of nodes in the order in which they are to be assigned. The node list, along with the name and *set_property fixed_route* define a Tcl command, which passed to Vivado, sets the net routing as a constraint.

So far, the router has been tested successfully on 64 routes per execution. The 64 nets correspond to the output values produced by two replicas (each on 32 bits) that, in the original place and route, were arranged in the resources of the third replica, the central one. Figure 4.12 shows a zoom-in on two nets whose routing is forced: the original routing is defined by a continuous line, the constrained one by a dashed line.

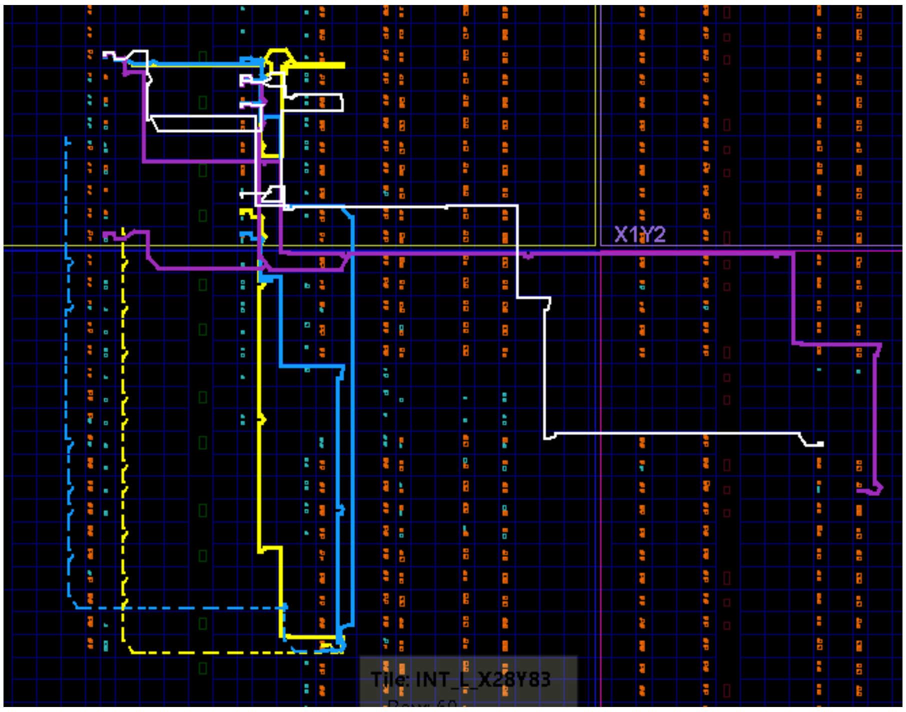


Figure 4.12. Fixed Route example: the blue and the yellow net with and without constraint

The next two images show the before and after routing constraint applied to the nets highlighted in green and yellow. Not always the constraint assigned by the tool is accepted by Vivado (about 5 nets out of 64 have not assumed the desired configuration). The problem should be sought in the compatibility between nodes, which should be further investigated.

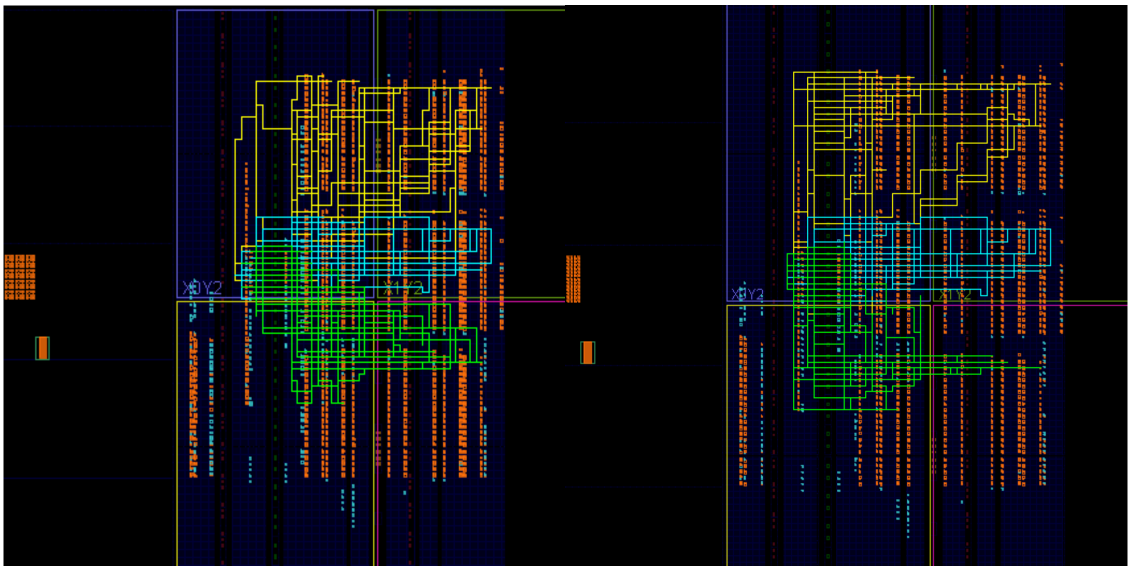


Figure 4.13. Before and after net isolation applied to the the yellow and green nets

Chapter 5

Experimental Analysis

Once all designs have been successfully implemented, the design flow carried to the bitstream generation, all that remains is to test the design reliability. The latter is questioned when the implemented design must be placed in an extreme environment, subject to radiation. The effect of such radiation, in the case of programmable logic is often, if not always, simulated by inducing a bit-flip in the configuration memory of the FPGA.

5.1 Bitstream File Organization

The bitstream is a binary file organized into frames, each containing a certain number of bits. The number of frames and bits per frame depend on the programmable logic with which the bitstream is coupled. In the case of the FPGA included in the PYNQ-Z2 board, used to implement all the designs that were discussed in the previous chapter, the bitstream consists of 10k frames each composed of 3231 bits. It is as important as it is interesting to understand how the internal structure of the bitstream is related to the FPGA resources, i.e., to understand in detail which bit configures what. From empirical research, an estimation of the relationship between frames and resources was made that led to the realization of a recurrence pattern in the organization of the bitstream. In particular, assuming that the basic unit in the FPGA is the CLB and that each CLB has its own matrix of interconnections, then in the bitstream it is possible to identify the sequence of frames that reflects this structure. Specifically: 26 frames are used to configure the switch matrix, followed by 16 frames used for LUTs and FFs of the CLB. After these 16 frames, another 26 frames of the next switch matrix of the adjacent CLB will follow, and so on. Each frame, dedicates 64 of the 3231 bits to configure a single resource, i.e. in the 26 frames of interconnections, every 64 bits refer to a different matrix; in the 16 frames of the CLB, every 64 bits we have bits referring to a different CLB. A graphical explanation is shown in the figure below.

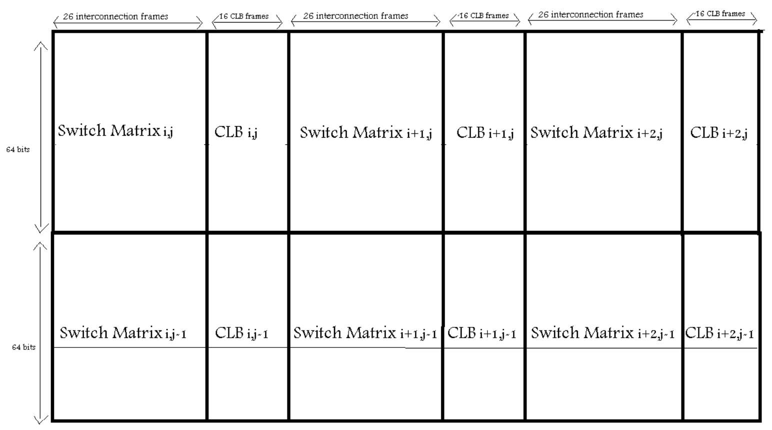


Figure 5.1. Bitstream organization structure: frame recursion scheme

The detailed knowledge of the bitstream structure, the correlation between bits and resources is extremely useful and necessary when it is intended to inject faults selectively, on specific resources of the design.

5.2 Software Test Enviroment

In the previous chapter it was explained in detail how the hardware test environment was made, and that remembering, required the presence of a processor that was able to send stimuli to DUT and read the responses. The DUT AXI interface is easily accessible from the core, as it is memory mapped which means that its registers are processed as main memory address by the processing unit. Recalling also that the DUT is an ALU TMR, a test routine has been realized and executed by the processor core, which has two purposes: on one hand it generates and writes in the AXI registers of the ALU the operands, the type of operation and the execution mode, on the other hand, it performs in parallel the same operations, so as to act as a golden reference. In detail, the test routine consists of having the ALU perform the supported operations, as detailed in the RI5CY manual, in the three supported vector modes. So not only the normal behavior of the ALU was tested but also its execution feature in SIMD. The applied stimuli consist of 30.000 operations, of which 10.000 executed in vector mode 32 (i.e. on data with size equal to the parallelism of the ALU), 10.000 executed in vector mode 16 (parallel execution on two pairs of 16 bits data) and the remaining 10.000 in vector mode 8 (on 4 pairs of 8-bit data). The operands are generated from time to time when a new operation, randomly chosen from the list of the supported ones, is to be performed and they assume both positive and negative values covering the whole range of representable values. These must be passed, together with the OP code and the ALU setting bits, to the AXI registers and at the same time to the golden reference implemented on the processor core. When the result is ready, the routine takes care of reading the contents of the output register of the DUT and compare it with the one generated by itself. Two counters progressively keep track of the number of matches and mismatches per execution. At the end, the execution output is just the number of mismatches encountered over 30,000 operations. A mismatch refers to a failure of the TMR system since, if the final result is incongruent, either the fault injected is such as to cause error on 2 replicas, distorting the voting system, or the faults afflict the voter. However, as reported in the previous chapter, a signal has been added to the traditional voting system that is set to 1 when there is a replica that produces a different result. At the level of the AXI interface, this signal drives one of its registers which will be set to 1 when one of the ALUs fails. This signal can be indicative of how effective the TMR technique is and therefore how many incorrect computations would have occurred without it. Since both stimulus generation and golden reference are implemented on the same board, an host environment should : - program the board by loading the test routine on the processor and the bitstream file on the FPGA configuration memory. - read from the serial port the result produced by the processor at the end of each execution. For this purpose, a Python serial interface has been realized to know the result of the test routine, and again by means of a python script the process of loading the bitstream and elf file of the test routine on the platform was automated.

5.3 Fault Injection

Fault injection is a widely used technique to simulate and test the effect of radiation. In the world of programmable logic, injection is done by forcing a flip in one or more bits of the device configuration memory. Specifically, in this research work, injection was done statically, that is, by injecting an error into the bitstream before it is loaded into memory. The bitstream was corrupted by means of PyXEL [20], which is a Python library created specifically for this type of operation. PyXEL allows you to choose in which region of the bitstream to inject the fault both in terms of frames and bits. This is a useful tool to avoid wasting time and resources by injecting into regions of the bitstream that are not used to implement the project under test. Each injection campaign consists of generating 10k corrupted bistreams, each with a single bit-flip located at a random point in a specific region of the bitstream. The region was chosen such that the three placements being compared had resources defined there. For each of the 10k faulty bitstreams, the test routine was executed.

5.4 Experimental Results

5.4.1 Comparison Between the Three Placements Arrangements

The following table collect the data obtained from the testing campaign carried out on the three placements.

	Original Placement	Isolated Placement	Mixed Placement
TMR Failures	$37/10^4$	$24/10^4$	$56/10^4$
Wrong Computations	$435972/3*10^8$	$304804/3*10^8$	$953792/3*10^8$
Bitstreams Causing Errrors	$408/10^4$	$291/10^4$	$822/10^4$

Table 5.1. Fault injection results as a function of different placements of the same design

5.4.2 Voter Switch Matrices Fault Injection

Results from the placement injection campaigns show that in the design where the replicas are arranged through a spacing policy, the TMR technique is more valid and makes the design fault tolerant in a more efficient way. However, the system does not reach 0 failures even if the isolation is implemented. This is probably due to the voter module, which as mentioned in the previous chapter, is a convoy point for the signals coming out of the replicas, which are the cornerstone of the voting process. The image below shows how the switch matrices in the proximity of the voter are, perhaps, the most critical point of the whole design, because through each of them pass all the three replicas results to be voted. So, a fault in one of the matrices, brings with it a high probability of failure of the TMR. Wanting to evaluate the actual sensitivity of these matrices, a targeted fault injection campaign was performed.

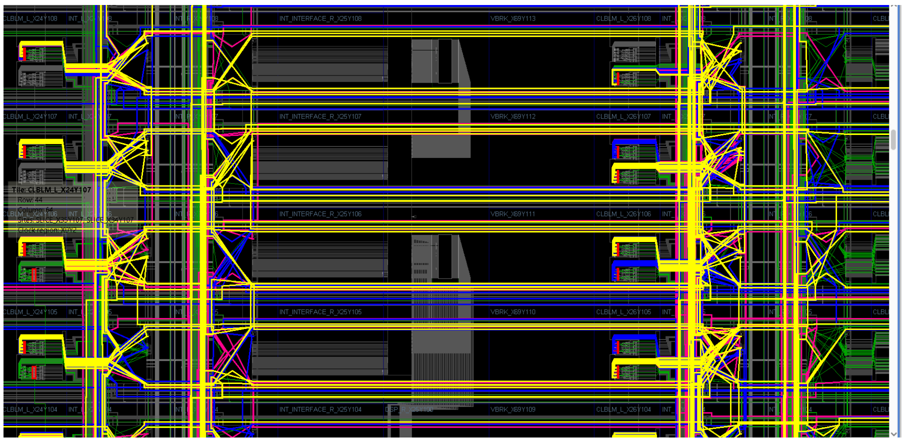


Figure 5.2. Routing resources close to the voter module

It is in this context that knowledge of the bitstream organization comes in handy, since it is intended to inject only into the interconnection resources of the voter. To facilitate the identification of the region of the bitstream used to implement the voter, it has been used a tool provided by PyXEL that given the bitstream generate a Polar Uplink Tool Bitmap (PMB) file, which as the name suggests is a bitmap, i.e. an image where there is a visual correspondence between bit and position of the resource that it configures in the device.Comparison of the two representations of the same design leads to the assumption that the voter (the green module) can be included between bitstream frames 788 and 949 as shown in the following figures.

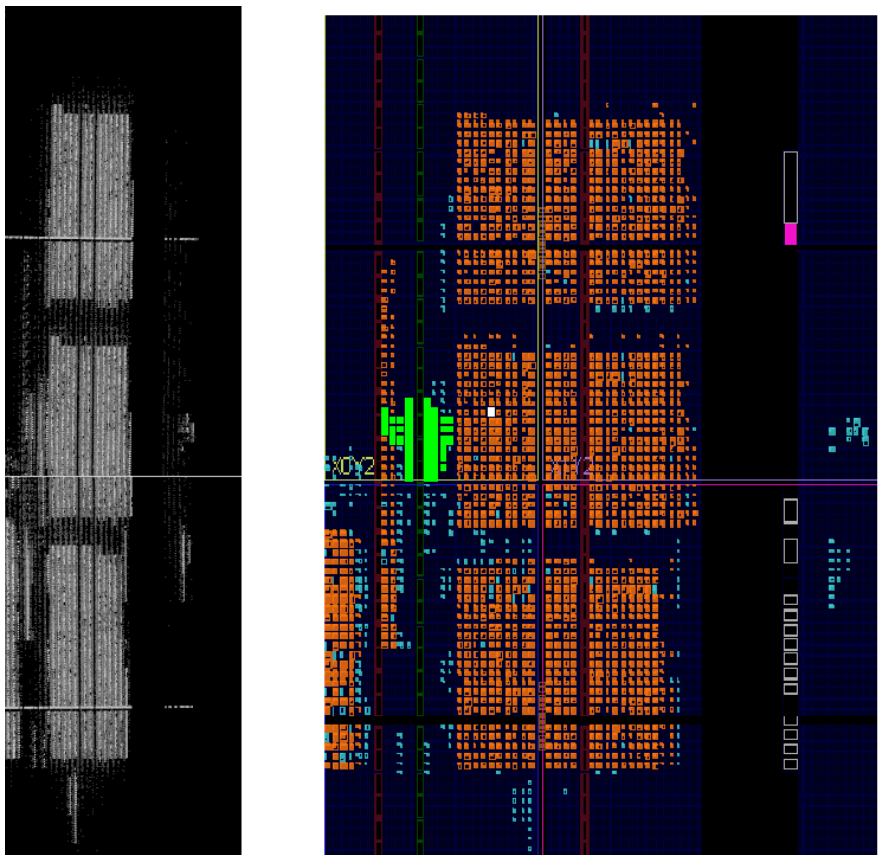


Figure 5.3. On the right the bitmap of the design, on the left the device view of Vivado

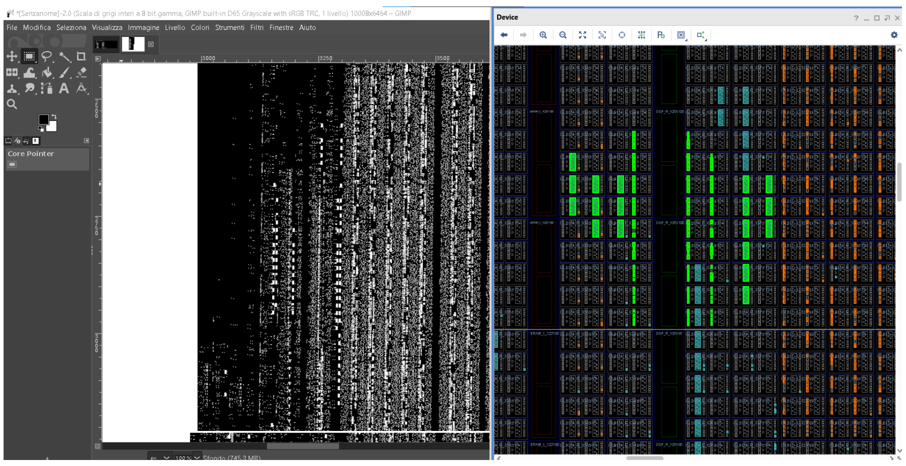


Figure 5.4. Voter bitmap

The results show that single bit-flips located only in the voter interconnect arrays, which are primarily responsible for routing the output signals of the ALUs to the voter, result in greater failure of the TMR technique than single bit-flips located in the logic cells of the design. In this case, the number of erroneous computations, having injected only in the interconnection resources, indicates how a bit-flip in them causes a change in the logical state of the signal as well as its path, thus confirming the idea that the voter and the resources associated to it are a very sensitive part of the project.

	Voter S-matrix Injection Result
TMR Failures	$165/10^4$
Wrong Computations	$3460699/3*10^8$
Bitstreams Causing Errrors	$515/10^4$

Table 5.2. Result of the fault injection campaigns performed on the Voter interconnection resources

5.4.3 Fault injection applied to the design of the TMR technique extended to the voter

To complete the TMR evaluation, the design in which the TMR technique is applied to both the ALU and the Voter was tested. In this condition, the only reasonable faults are those due to the AXI peripheral that is used to communicate with the uProcessor on the board; thus, it is likely that the AXI interface, due to a bit-flip, provides either different input values to the design on FPGA than those processed by the golden reference, or that the fault acts on the output register used to store the result produced by the DUT and that is read by the processor.

	Injection Result
TMR Failures	$16/10^4$
Wrong Computations detected by Voter0	$199902/3*10^8$
Wrong Computations detected by Voter1	$199744/3*10^8$
Wrong Computations detected by Voter1	$199615/3*10^8$

Table 5.3. Result of the fault injection campaigns performed on the extended TMR design

Chapter 6

Conclusions

In the presented research work, an alternative environment has been implemented to modify the place and route of projects implemented on FPGAs. The need for such an external support was born after the analysis of a RISCV design on FPGA when it was observed that the implementation generated by the tool distributed by the manufacturers was not sufficiently robust. In particular, the realized place-and-route environment allows to assign the location and the resource to the logical cells of the design with a fine detail equal to the primitive type. Having a total control of the placement, at the level of a single LUT, allows preventing what have been defined as potentially critical cases, i.e. those circumstances in which the same physical resource (SLICE) is assigned to more than one module. The criticality is mainly associated with the use of designs in adverse environments, characterized by the high presence of radiation, which, causing an SEU in one of the modules, can affect also the module not directly hit, but close to it, thus decreasing the reliability of the entire design. Assuming therefore use of the RISCV processor for aerospace applications, two optimizations were performed: at the architectural level, a fault tolerance technique was applied to make the design more hardy with respect to radiation. This technique, known as Triple Modular Redundancy (TMR) involved the arithmetic logic unit of the processor. Subsequently, at the place&route level, the developed tool was used to implement several ALU TMR placement solutions to evaluate how much the placement, on a fault tolerant design, affects its reliability. Fault injection campaigns performed on the various placement arrangements, aimed at simulating the effect of radiation on the design, reported that the TMR technique at the architectural level should be accompanied by module isolation. Moreover, the isolation must be extended also to the routing resources used by the modules and for this reason a basic router capable of assigning constrained paths to the signals has been realized. The router was used to modify the routing of the output signals of the ALU replicas, preventing them from using resources that were defined outside the perimeter associated with the replica they belonged to. The platform tries to be as generic as possible, and therefore, to be able to modify designs in an automated way without knowing the purpose or the hierarchy of the implemented design, but only by setting constraints from the outside, working mainly on the primitive types used in the original implementation and relying primarily on knowledge of the structure and organization of physical resources on FPGAs.

6.1 Future Work

The routing problem is one of the biggest and most fascinating challenges, especially when it comes to programmable logic since most of the FPGA area is occupied by interconnections having reduced flexibility. As the router is dependent on the architecture of the logic, in-depth knowledge must be gained before being able to create a complete and efficient one, and this is often limited by the paucity of details that are provided by the manufacturers. So far, the implemented router is capable of connecting a small number of wiring segments, i.e. it does not exploit all the nodes in the interconnection arrays, nor all their pips. Therefore,the idea is to continue to study the structure of the interconnections, to know in detail the function of each node and segment in order to realize a complete router that allows to explore the design space also from the point of view of interconnections.

Bibliography

[1] Andrew Waterman et al. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0.* http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html. May 2014.

[2] Pasquale Davide Schiavone Andreas Traber Micheal Gautschi. *The RI5CY: User Manual Revision 4.0.* https://www.pulp-platform.org/docs/ri5cy_user_manual.pdf. Apr. 2019.

[3] David Kaeli et al. «Chapter 2 - Device architectures». In: *Heterogeneous Computing with OpenCL 2.0.* Ed. by David Kaeli et al. Boston: Morgan Kaufmann, 2015, pp. 15–39. ISBN: 978-0-12-801414-1. DOI: https://doi.org/10.1016/B978-0-12-801414-1.00002-8. URL: https://www.sciencedirect.com/science/article/pii/B9780128014141000028.

[4] Stefano Di Mascio et al. «The Case for RISC-V in Space». In: *Applications in Electronics Pervading Industry, Environment and Society.* Ed. by Sergio Saponara and Alessandro De Gloria. Cham: Springer International Publishing, 2019, pp. 319–325.

[5] *Nexys Video™ FPGA Board Reference Manual.* https://digilent.com/reference/_media/reference/programmable-logic/nexys-video/nexys-video_rm.pdf. Sept. 2020.

[6] Xilinx. *7 Series FPGAs Data Sheet: Overview.* https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf. Sept. 2020.

[7] *PYNQ-Z2 Reference Manual v1.0*. https://www.mouser.com/datasheet/2/744/pynqz2_user_manual_v1_0-1525725.pdf. May 2018.

[8] *Vivado Design Suite User Guide: Design Flows Over*. https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx2020_2/ug892-vivado-design-flows-overview.pdf. Feb. 2021.

[9] *AXI Documentation Webpage*. https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Introduction/About-the-AXI-protocol?lang=en.

[10] Milton Ohring. *Reliability and Failure of Electronic Materials and Devices, 2nd Edition*. eng. Academic Press, 2011. ISBN: 0-12-088574-3.

[11] G Bruguier and J.-M Palau. «Single particle-induced latchup». eng. In: *IEEE transactions on nuclear science* 43.2 (1996), pp. 522–532. ISSN: 0018-9499.

[12] G.K Lum et al. «New experimental findings for single-event gate rupture in MOS capacitors and linear devices». eng. In: *IEEE transactions on nuclear science* 51.6 (2004), pp. 3263–3269. ISSN: 0018-9499.

[13] L. Bozzoli B. Du C. De Sio S. Azimi and L. Sterpone. *Radiation-induced Single Event Transient effects during the reconfiguration process of SRAM-based FPGAs*. Microelectronics Reliability, vol. 100–101, 2019, doi: 10.1016/j.microrel.2019.06.034.

[14] Luis Alberto Aranda et al. «Analysis of the Critical Bits of a RISC-V Processor Implemented in an SRAM-Based FPGA for Space Applications». In: *Electronics* 9.1 (2020). ISSN: 2079-9292. URL: https://www.mdpi.com/2079-9292/9/1/175.

[15] Elena Dubrova. *Fault-Tolerant Design*. eng. 2013th ed. Fault-Tolerant Design. New York, NY: Springer New York, 2013. ISBN: 9781461421122.

[16] C De Sio L Bozzoli B Du S Azimi and L Sterpone. *Radiation-induced SET on Flash-based FPGAs: analysis and filtering methods*. ARCS 2017; 30th International Conference on Architecture of Computing Systems.

[17] Daniel Francisco Gomez Prado. «Tutorial on FPGA Routing». In: *Electrónica - UNMSM* (June 2006), pp. 23–33. URL: https://revistasinvestigacion.unmsm.edu.pe/index.php/electron/article/view/4510.

[18] Sarah Azimi et Al. Boyang Du Luca Sterpone. *Ultrahigh energy heavy ion test beam on Xilinx Kintex-7 SRAM-based FPGA*. IEEE Transactions on Nuclear Science 66 (7), 1813-1819.

[19] Stephen Dean Brown. *Routing Algorithms and Architectures for Field-Programmable Gate Arrays*. 1992.

[20] Ludovica Bozzoli et al. «PyXEL: An Integrated Environment for the Analysis of Fault Effects in SRAM-Based FPGA Routing». In: IEEE, 2018.