POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Elettronica

Tesi Magistrale

Protection and characterization of an open source soft core against radiation effects

Relatore
Marco Parvis

Correlatore:
Stefania Bufalino

Filippo Minnella

Aprile 2018

Contents

List of Tables 4

List of Figures 5

1 Introduction 7
- 1.1 Basic concepts on SRAM-based FPGA . . . 8
  - 1.1.1 Kintex Ultrascale . . . 8
- 1.2 Radiation effects on electronics . . . 10
  - 1.2.1 Cumulative Effects . . . 10
  - 1.2.2 Single Event Effects . . . 11
  - 1.2.3 SEE in SRAM-based FPGA . . . 14
- 1.3 Inner Tracking System (ITS) . . . 16

2 Core selection and implementation 17
- 2.1 Core specifications . . . 17
  - 2.1.1 Survey on available architectures . . . 17
  - 2.1.2 VexRiscv . . . 20
- 2.2 Implementation flow . . . 21
  - 2.2.1 Description . . . 21
- 2.3 Core verification and implementation results . . . 24

3 Fault Tolerance 26
- 3.1 Fault Tolerance . . . 26
  - 3.1.1 Hardware redundancy . . . 27
  - 3.1.2 Information redundancy . . . 29
  - 3.1.3 Time redundancy . . . 30
  - 3.1.4 Design choices . . . 30
  - 3.1.5 Design automation . . . 32
- 3.2 Memory design . . . 34
- 3.3 Implementation flow . . . 37
  - 3.3.1 Description . . . 37
  - 3.3.2 Implementation results . . . 37

4 Core characterization 40
- 4.1 Metrics . . . 40
  - 4.1.1 Test environments . . . 41

4.2 Testing circuitry . . . 45
4.3 Implementation Flow . . . 47
4.3.1 Description . . . 47
4.4 Results . . . 48
4.4.1 Results evaluation . . . 49
4.4.2 Comparison with other projects . . . 49

5 Conclusions and future work 52

A 53
A.1 Kintex Ultrascale FPGA Architecture . . . 53
A.1.1 Ultrascale resources . . . 54
A.1.2 Development Kit . . . 60
A.2 ITS . . . 62
A.2.1 Readout Unit . . . 62
A.2.2 GBT-SCA . . . 63
A.2.3 SPI interface . . . 65

B 66
B.1 Hardware Licenses . . . 66
B.2 RISC-V . . . 68

C 69
C.1 AES algorithm . . . 69

D Design Automation 71
D.1 Memory substitution . . . 71
D.2 Memory mapping . . . 73

E VHDL files 77
E.1 Core files . . . 77
E.2 Memory files . . . 86

Bibliography 94

List of Tables

1.1 Ultrascale Cross-Sections 15
2.1 Resources used for Murax core with related percentages compared to XCKU040 resources. 24
3.1 ECC equations, all the bits marked with 'x' are associated to the column parity bit. 36
3.2 Resources used by the different core versions with percentages referred to XCKU040 FPGA. 37
4.1 Murax core characterization with fault injection, the table is showing percentages related to faulty and stopped processors added to MTTF evaluation. 48
4.2 Murax core characterization with particles beam, differently from fault injection the percentages of cores that survived each run is shown. 49
4.3 Resources used for MicroBlaze processor and related percentages related to XCKU040 FPGA. 50
4.4 Microblaze characterization with fault injection 50
4.5 Microblaze characterization with particles beam 50
A.1 Resources related to Ultrascale family [12] 53
A.2 KU040 and KU060 resources [12] 54
A.3 LEDs pins mapping [15] 60
A.4 LEDs pins mapping [15] 61
B.1 Summary of the licenses considered 67

List of Figures

1.1 Internal structure of a CLB cell [11]. 9
1.2 Internal structure of a switch point, the gate of each switch is driven by the content of a configurable SRAM memory cell 9
1.3 The first three images show the effect of the pairs generation inside the oxide, the last one the defects generation at the interface between oxide and silicum 11
1.4 Effects of a particle when hits a MOSFET, the image shows the electron-hole pairs generated. 12
1.5 Parasitic BJTs in a CMOS structure. The first is a pnp based on the p-mos source and on substrate while the second one is a npn with collector corresponding to CMOS well, base to substrate and emitter to the n-mos one. 13
1.6 Typical Structure of a Vertical Power MOSFET 14
2.1 Top entity block diagram, the PLL receives in input the single-ended clock generated by IBUFDS and produces the 40MHz one supplied to the core. . . 21
2.2 Core implementation flow diagram. It is both including netlist and software compile processes. 25
3.1 Block scheme of a system implementing Triple Modular Redundancy. Three hardware replica are shown together with output voter. 28
3.2 Block scheme of a system implementing Duplicate With Comparison. Two hardware replica are shown together with mismatch comparator. 28
3.3 Block scheme of a system implementing Error Correcting Code. Memory inputs and outputs are elaborated by the encoder and the decoder, the latter produces final mismatch lines. 29
3.4 Internal structure of the built-in ECC for Ultrascale BRAMs, the input and output data have a 64 bits parallelism. The scheme implemented by Xilinx is similar to the one in figure 3.3 32
3.5 Distributed TMR with sequential loop voting. The main difference with the scheme shown in figure 3.1 is the double step voting logic that includes even the sequential loops. 33
3.6 Memory system schematic, the output data is depending on values read from DATA and ECC BRAMs. 35
3.7 Core protection implementation flow diagram. Both hardware and software compile processes are taking into account the new memory system. 39
4.1 System architecture for tests with fault injection, the external devices used are talking to the FPGA using SPI, for the PC, and JTAG, for the JCM. . . 43

4.2 System architecture for tests with particles beam, the main difference with respect to fault injection architecture is the usage of SCA and CRU to communicate with central PC. 43
4.3 Test circuitry block diagram, the FSM is communicating with BRAMs and handling both output registers, to SPI, and data for processor testing. 45
4.4 Test circuitry implementation flow diagram 51
A.1 Block representation of BRAM used in SDP mode [13] 56
A.2 Block representation of BRAM used in TDP mode [13] 57
A.3 PLLs block diagrams [14] 57
A.4 Block representation of an IBUFDS [10] 58
A.5 Block representation of an IBUF [10] 59
A.6 Block representation of an IBUF [10] 59
A.7 Conditioning circuitry schematic [15] 60
A.8 JTAG Chain schematic [15] 61
A.9 GBT-SCA block diagram [16]. 64
D.1 Protected memory insertion flow diagram 76

Chapter 1

Introduction

The effects of external radiations on electronic systems are becoming more and more evident with the scaling of the technologies used to produce integrated circuits; in order to reduce these effects, particular techniques are applied during the design and the production of the electronic devices. These problems are crucial for the following fields:

- Automotive;
- Space;
- High Energy Physics;

Because of the low cost and the high versatility of FPGAs, these devices are replacing custom solutions and radiation hardened microcontrollers in electronic systems working in harsh radiation environments; especially for what concerns the high energy physics experiments, the high bandwidth guaranteed by commercial SRAM-based FPGAs is particularly useful for the Readout electronics.

Despite these advantages, a microcontroller able to easily perform common automation jobs, like communicating with other systems, could be necessary in order to avoid the growing of firmware complexity for FPGAs-based systems; for this reason, Soft Cores are programmed on FPGAs.

The usage of soft microcontrollers in harsh radiation environments requires the application of specific techniques with the object of reducing the possibility of a misbehavior during the operational time of the device.

The purpose of this study is, starting from a core already designed and tested at functional level, to explore the techniques that could be applied in order to harden the core using both commercial tools and custom approaches. The whole research has been performed in the framework of the ITS (Inner Tracking System) Detector update for the ALICE experiment; the design has been developed and tested on a Kintex Ultrascale XCKU040 FPGA from Xilinx.

1.1 Basic concepts on SRAM-based FPGA

A Field Programmable Gate Array (FPGA) is a programmable device with the peculiarity of guaranteeing a huge quantity of available logic with respect to the previous technologies; there are different types of technologies used to produce an FPGA:

- Antifuse technology: one-time programmability, the antifuse is naturally acting as open circuit and when programmed it turns into a short circuit;
- Flash tecnology: same tecnology as Nand Flash, it is reprogrammable but for a limited amount of time. The programmed firmware is mantained even with power cycle;
- Static RAM (SRAM) technology: same technology as Static Ram, reprogrammable an infinite amount of time but the programmed firmware is volatile with respect to power cycles;

A huge percentage of modern FPGAs are based on SRAM technology because it's cheaper and guarantees higher performances; in order to remove the problem of the firmware volatility some on-board solution can be applied, like automatically programming the FPGA at power-up using a memory chip to store the bitstream.

The general architecture of an SRAM-based FPGA is composed by:

- IOBs: Input/Output Blocks of the system, at least composed by the pins needed to reprogram the FPGA;
- CLBs: Configurable Logic Blocks, used to implement the logic functions. Their architecture may differ depending on the FPGA used;
- Routing resources: used to connect CLBs to CLBs and CLBs to IOBs;

Because the testing platform for the final design will be a Kintex Ultrascale, the internal structure of this FPGA family will be discussed.

1.1.1 Kintex Ultrascale

The Ultrascale family is similar to the other Xilinx FPGAs in the general blocks organization; the CLBs are disposed in arrays, surrounded by IOBs and everything is interconnected through routing switches, usually made with pass transistor technology, that form the General Routing Matrix (GRM); in each intersection between a vertical and an horizontal channel a routing resource is inserted. There are other types of logic elements composing common Xilinx FPGAs arhictectures like:

- Block RAM (BRAM): particularly useful when huge amounts of memory are required;
- Delay-Locked Loops (DLL): used for delay compensation in clock distribution;
- Phase-Locked Loops (PLL): used to obtain different clock frequencies and reduce skew problems in clock distribution;

- Digital Signal Processor (DSP): used to implement complex arithmetic functions with high performances;

The particularity of the Ultrascale family with respect to other FPGAs from Xilinx is the structure of the CLB block; the number of inputs is dependent on the number of outputs defined, if there is only one output then the inputs are 6 while if there are 2 outputs the maximum number of inputs is 5. The internal resources of a single CLB are shown in the image below:

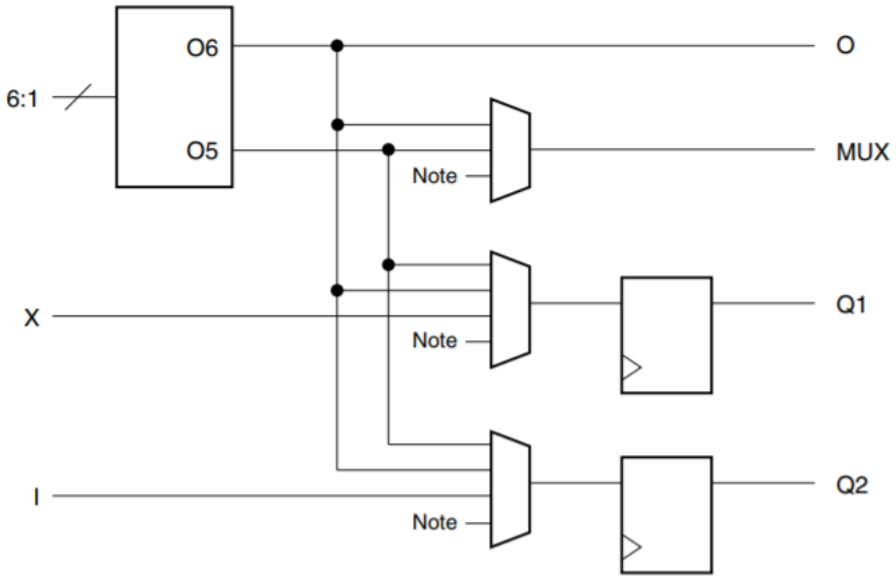


Figure 1.1. Internal structure of a CLB cell [11].

Another important circuit is the one that composes a routing resource, even called switch point; usually, a programmable routing element is implemented using 6 pass transistor in order to allow all the output directions. The image below shows the typical structure of a unit:

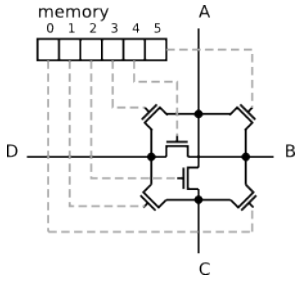


Figure 1.2. Internal structure of a switch point, the gate of each switch is driven by the content of a configurable SRAM memory cell

1.2 Radiation effects on electronics

The effects of the external radiations on electronic circuits are chategorized in the following way:

- Cumulative Effects:
  - Total Ionizing Dose;
  - Cumulative Displacement;
- Single Event Effects (SEE):
  - Soft Errors:
    - Single Event Upset (SEU):
      - Single Bit Upset (SBU);
      - Multiple Bit Upset (MBU);
      - Single Error Functional Interrupt (SEFI);
    - Single Event Transient (SET);
  - Hard Errors:
    - Single Event BurnOut (SEBO);
    - Single Event Gate Rupture (SEGR);
    - Single Event Latchup (SEL);

1.2.1 Cumulative Effects

TID The first type of cumulative effects are taking place for the whole lifetime of the device and they will produce a misbehavior only if the total ionizing radiaton received (TID) is higher than the maximum tolerated from the device. The TID is the measure of the ionizing energy deposited by the radiation that passes through the device, the official measurement unit is Gray (Gy) even if in many applications is used the rad; the equivalence between the two units is: 1Gy = 100rad.

The effects of TID are the following:

- Accumulation of electron-hole pairs in the oxide part of the MOS structure because of the ionizing energy deposited by the radiation; the recombination rate of these pairs is low in the oxide structure and even lower if the MOS is polarized, for these reasons the particles start to drift inside the electric field until the electrons will leave the oxide while the holes will be trapped creating defects in the structure. The accumulation of the holes will polarize, negatively or positively depending on the type of transistor, the MOS structure modifying its behavior;
- Accumulation of defects in the interface between the semiconductor and the oxide. This effect will modify the mobility of the electrons inside the conductive channel of the MOS structure and modify the threshold voltage of the transistor;

The main difference between the two effects is that the first one generates very fast and can be reduced heating up the material to 400°C while the second one generates slowly but cannot be reduced with high temperatures.

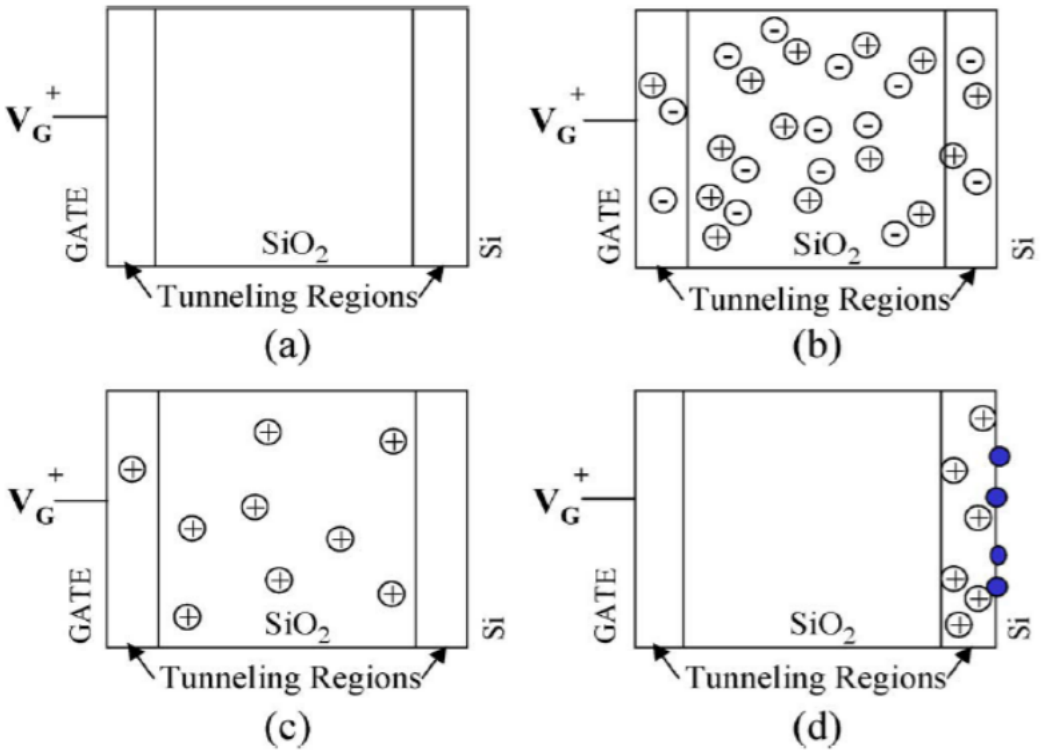


Figure 1.3. The first three images show the effect of the pairs generation inside the oxide, the last one the defects generation at the interface between oxide and silicum

Cumulative Displacement Another type of cumulative effects is called displacement, it is based on the quantity of energy transferred by the impact of particles to the semiconductor lattice; if the energy tranferred is higher than the displacement energy than the atom will be removed from its original position in the lattice and this will change the electrical parameters of the electronic device, worsening the performances.

1.2.2 Single Event Effects

SEEs are taking place because of a single particle hitting the device and generating an error in the circuit, not because of the cumulation of the radiation effects. The rising of a SEE error is directly dependent on the quantity of energy transferred by the particle that is hitting the silicum, this quantity is called Linear Energy Transfer (LET) and is expressed as $\mathrm{MeV\,cm^2\,g^{-1}}$; the maximum quantity of energy that is not creating a bit flip is named LET_{th}.

These effects can be divided in two categories:

- Soft Errors: bit flip in the value stored by a memory cell, this cell could be both related to logic or configuration memory. Some particular techniques can be implemented in order to detect and mask this errors and in any case, after a power cycle, the system returns to work correctly;
- Hard Errors: permanent damage to the device, the device cannot work properly anymore;

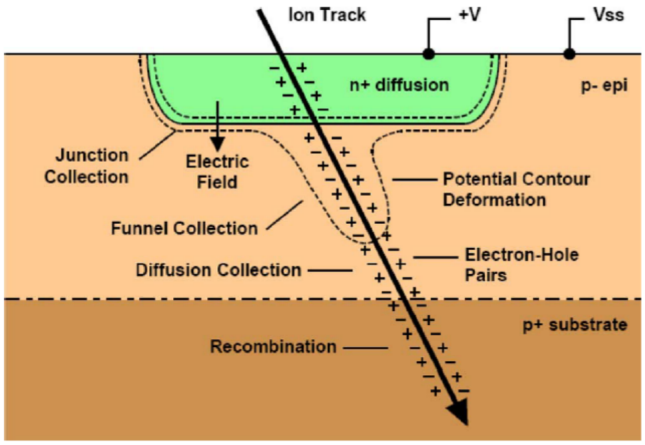


Figure 1.4. Effects of a particle when hits a MOSFET, the image shows the electron-hole pairs generated.

Soft Errors

SET The SET is a current or voltage spike produced by a particle hitting the material, this kind of event causes an error only if the transient change of logic value in the circuit is sampled by a memory element. Because the arising of a SET is strongly dependent on the timing of the event, it is hard to estimate the probability of an error.

SEU The SEU is produced by a particle hitting the depletion region of a p-n junction, this event will generate electron-hole pairs that means a current spike flowing in the structure. An SRAM cell is composed by two inverters, both composed by two transistors; a particle hitting one of the MOS transistor provokes a state change that will force the same effect on the opposite inverter and this will produce a flip of the stored content value. The probabilty of having a SEU in a device is called Cross-Section, it is expressed as $\sigma = \frac{N_{events}}{\Phi}$, where Φ is the fluence, and is measured in cm^2. The value used for the fluence depends on the type of particles that hit the device:

- If the particles are protons than the fluence is independent from the angle of incidence;

- If the particles are heavy ions than the fluence is dependent from the angle of incidence and the new fluence used to evaluate the Cross-Section is: $\Phi_s = \Phi cos(\theta)$;

SEUs are classified depending on the effects that they have on the memory content of the affected device:

- Single Bit Upset: the particle flips the content of only one memory cell;
- Multiple Bit Upset: the particle flips the content of more than one memory cell. The reasons could be:
  - A particle that hits more than one memory cell in sequence;
  - A particle that hits a region common to many memory cells;
  - The secondary particles generated from the first hit generate a second event on another memory cell;
- Single Event Functional Interrupt: the particle creates an event on the circuitry related to power-on, reset or Joint Test Action Group (JTAG) interface;

Hard Errors

SEL The SEL is produced by a particle hitting a CMOS structure and activating the positive feedback structure formed by the parasitic BJTs shown in the image below.

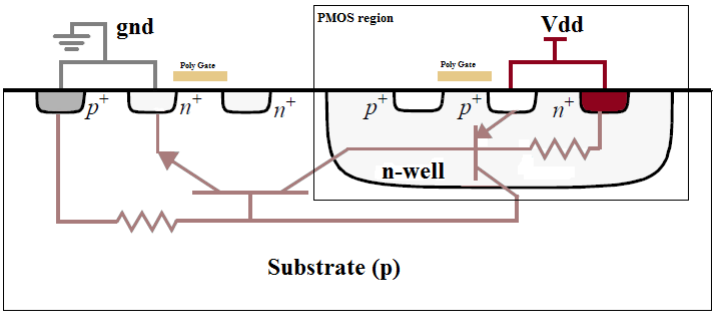


Figure 1.5. Parasitic BJTs in a CMOS structure. The first is a pnp based on the p-mos source and on substrate while the second one is a npn with collector corresponding to CMOS well, base to substrate and emitter to the n-mos one.

An increase of the collector current of the pnp transistor rise up the base current for the npn transistor that is directly connected to the base of the pnp one, this mechanism is a positive feedback that will increase the current. The effect of a SEL is dependent on the resistance of the parasitic structure, it can be:

- Fatal: current density exceeds the maximum safe value;
- Temporal: higher heat imply higher current consumption, in this case a power cycle can restore the circuit;

SEBO The SEBO is an effect related to the presence of Bipolar Junction Transistor (BJT) in the device analyzed; usually this problem can be related to power MOSFETs because of the parasitic BJT present right under the source. A particle, hitting the p-type substrate right under the source, will generate a base current turning on the device and will lead to overheat and destruction of the structure.

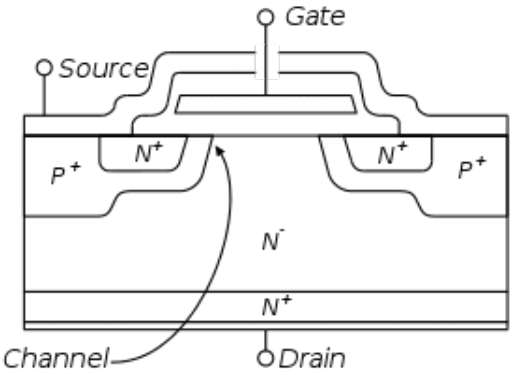


Figure 1.6. Typical Structure of a Vertical Power MOSFET

SEGR The Single Event Gate Rupture (SEGR) is caused by an accumulation of charges in the semiconductor beneath the oxide of a MOSFET; this process will enhance the electric field through the silicum dioxide, the leakage current will raise and the overheating process, caused by the high current density, could damage the insulator and the device in a permanent way.

1.2.3 SEE in SRAM-based FPGA

The SRAM-Based FPGAs CRAM and BRAMs are composed by a huge quantity of memory cells that can be affected by external radiations, for this reason this type of devices is particularly susceptible to SEEs. Considering a particle hitting an SRAM cell, an upset could affect:

- Routing Logic: one of the SRAM cells driving a pass transistor;
- CLBs LUT: one of the SRAM cells related to the internal functional logic resulting in a behavioral change;
- User memory: one of the SRAM cells used for data storage;
- IOBs: an upset could change the driver behavior related to an I/O pin;

Even if most of an SRAM-based FPGA resources are susceptible to SEEs, a huge part of the configuration memory is not used, for this reason a lot of events will not compromise the correct behavior of the design. The configuration memory bits significants for a design

are called essential bits and are, usually, a low percentage of the whole CRAM; this means that the probability of having a SEU must be weighted in order to estimate the effects on the design used.

Using extensive characterization, the CRAM and the BRAMs cross-sections have been evaluated for 20 nm Ultrascale technology:

	CRAM	BRAM
$\sigma(\frac{cm^2}{bit})$	2.55E-15	4.43E-15

Table 1.1. Ultrascale Cross-Sections

1.3 Inner Tracking System (ITS)

The ITS is a complex electronic system part of LHC ALICE experiment that aims to study heavy-ions collisions, at a centre-of-mass energy of ~ 5.5TeV per nucleon [8]. It has the goal of studying the behavior of dark matter at high densities and temperature. Inner Tracking System is devoted to the data taking part. During the next years an update of the system will be performed to reach the following structure composed by 3 regions of sensors:

- Inner barrel: 3 internal layers;
- Middle barrel: 2 internal layers;
- Outer barrel: 2 internal layers;

Each part is characterized by a specific mechanical arrangement of the sensors around the beam axis. This sensing region communicates with the external world using dedicated busses. The Readout Electronics (RE) works as interface between staves and readout, control and trigger system of ALICE experiment. From the experiment point of view, it aims to collects the data from the sensors, organizes them in order to be easily readable and sends them through e-links (fiber optical communication protocol) to the Common Readout Unit (CRU). A more detailed explanation of the Readout Electronics is performed in appendix A.2.1.

RE will be positioned in the cavern close to particles accelerator and sensing electronics, this means that is needed to verify its behavior in an harsh radiation environment. Depending on the type of components and instruments considered, the main effects caused are different. The Readout Electronics will use Kintex Ultrascale XCKU060 FPGA as main component of the system. For SRAM-based FPGAs the main problems are caused by SEUs and not by TID [9], this means that some techniques must be applied in order to reduce the impact of events during the operational life of the design.

Chapter 2

Core selection and implementation

The main topic of this chapter is to select an Instruction Set Architecture (ISA) and then a core compliant with the specifications received. At first the specifications considered during the selection process are discussed, then some open-cores will briefly described and compared in order to select the targeted device. To reduce the time needed to design, test and validate the core, the research will focus on an already implemented netlist based on the chosen ISA. In order to ease the usage of this design on the targeted FPGA, a toolchain, that performs all the operations needed to obtain a working bitstream, is built.

2.1 Core specifications

The object is to obtain a core with the following characteristics:

- Low resource utilization: in order to reduce the quantity of essential bits related to the core implementation, is fundamental to use a small core;
- Automation purposes: because the processor should be used only to run simple application, complex units like Floating Point Unit (FPU) or Coprocessors are not needed. However, in order to target future applications, it would be good to have a core easily extendable in his functions;
- C compiler available: in order to easily program the core, an already available C compiler is fundamental;
- Free from patents: not only the core but even the ISA must be completely open;

2.1.1 Survey on available architectures

A lot of different open cores already implemented are available for the usage, the analysis will focus only on some of them. Targeting very poor cores in terms of resources leads to low parallelisms, like 8 or 16 bits; the problem of this architectures is that many of them are owned by companies or that no C compiler is available.

Atmel AVR An example of 8-bit microcontroller is the one developed by Atmel in '96; the AVR family is based on a Reduced Instruction Set Computer (RISC) architecture. These microcontrollers were particularly popular because one of the firsts implementing flash-based memory instead of the previous technologies used for non-volatile applications. A C compiler is present for these devices. Some RTL netlists have been designed and published as open-cores.

MIPS MIPS processors are based on a superscalar, RISC architecture and the parallelism of the busses could be 32 or 64 bits; these types of devices were particularly popular during the 90's where a lot of general computing systems used processors based on this architecture, lately the usage reduced to some particular applications. Around the end of the 90's, these devices were dominant in the embedded market. Since the release of the first version, many upgrades have been published reaching the actual implementation called MIPS32/64. For this latest release a superset architecture has been designed, it is called microMIPS32/64 and adds the support to 16-bits instructions. There are a lot of Application Specific Extensions that are used to introduce functionalities related to particular applications, to this category are related, for example, MIPS implementation for multi-threading, microcontrollers and DSPs. The license for MIPS architectures is now owned by Imagination Techlogies, that acquired MIPS technologies.

OpenSPARC The OpenSPARC project is based on a family of microprocessors RISC-based with a parallelism of 64-bits, these processors are particularly used in applications where multi-threading is particularly important, such as servers. The first core belonging to this family is based on the UltraSPARC T1, a device commercialized by Sun Microsystems in 2005; after some months the company decided to publish, with an open-source license, the design files related to the core creating the OpenSPARC T1. In the 2007 the UltraSPARC T2 has been developed and again the design files were published, the related core is called OpenSPARC T2.

MSP430 The MSP430 is a family of microcontrollers with a parallelism of 16-bits, the architecture is not considered as fully RISC because, in some cases, the result of the computations are directly stored in the memory. This type of processors is usually used in the field of embedded systems thanks to their low power consumption. Different versions of the same architecture are implemented for different applications. The MSP430 is completely open-source, from the ISA to the CADs needed for the development boards, for this reason a lot of different open-cores, based on this architecture, have been designed; one of the most important is the openMSP430, it is completely compatible with the specifications and the development tools produced by Texas Instruments. A fault tolerant version of this core has already been implemented by Thales Alenia for space applications.

OpenRISC 1000 The OpenRISC 1000 is an architecture that characterizes a family of processors RISC-based and with a parallelism of 32 bits; there are different cores implemented targeting different applications, from network services to embedded systems. This is an open-source project with the object of creating a completely free platform for hardware

developing. Among the different open-cores implemented there is a fault tolerant System on a Chip (SoC) designed by Microtec.

RISC-V The RISC-V is an ISA designed by the Berkeley Foundation and it is based on the first RISC architectures for processors. This Instruction Set Architecture has been developed to have a completely free, customizable and simple framework for both academical and commercial purposes. The target of this project is to have a common platform that can be adapted to different fields, like embedded systems or servers, without changes in the main architecture but expanding the functionalities of the system starting from the basic ones. The core of this ISA is only composed by standard computational, memory and control-flow instructions; for more complex operations, standard or custom extensions could be used to enlarge the instruction set. There are a lot of different RISC-V open-cores that implement fault tolerant designs in order to protect the processor from radiations. A full description of this ISA is written in appendix B.2

Core selection To select one core some considerations must be done on the architectures previously analyzed:

- AVR: a way to reduce the number of resources used to implement the core is to use architectures with small widths of the busses. 8-bits AVR cores from Atmel are especially used in embedded applications and meet this characteristic. The main problem in targeting this family of devices is related to the licenses that could restrict their usage;

- MIPS: the same considerations can be done for MIPS architectures. Even if these processors have a minimum parallelism of 32 bits, the huge flexibility of the architecture helps in finding already designed solutions for a lot of different application fields. However, MIPS is a licensed product, this means not considerable as open-source, and, for this reason, has been discarded from the possible choices;

- OpenSPARC: this architecture is different from the previous two because is open-source. The main problem is related to the fact that the target application related to these devices requires much more resources than the ones needed for automation purposes and this means that is not a good choice;

- openMSP430: this is the first architecture described that is both open-source and targets an embedded application; it has the smallest parallelism between the open-core considered, is compatible with the toolchain developed by Texas Instruments to compile the software and has a huge set of available CADs for the development kits. In addition, Thales Alenia developed a radiation hardened version of the processor. However, because of the low architecture flexibility, is discarded from the possible choices;

- OpenRISC: as the previous one, this family of processors is open-source and has some particular implementations targeting embedded applications. The lower parallelism available is 32-bits and the GNU toolchain has been ported on this architecture in

order to allow the C and C++ code developing. As for the other open-cores, a lot of different fault tolerant versions have been designed;

- RISC-V: Among the options considered, this ISA is the most recent, it has a parallelism of 32-bits and has different cores designed to target applications that need simple resources. The C compiler is an adapted version of GCC. Similarly to other open solutions, is easy to find versions of the core implementing high reliability techniques in order to improve the resistance against radiations.

All the last three architectures can be considered as good choices for the project, however the simple structure and modularity of the RISC-V, that allows to easily extend the standard ISA, is the reason why this one will be the targeted family of devices.

2.1.2 VexRiscv

Among the huge amount of available cores that refers to the RISC-V ISA, the selected one is the VexRiscv with the Murax SoC. The related netlist is automatically generated through an high-level language called SpinalHDL were additional resources, instructions and features can be added to the core. Some of the specifications related to the VexRiscv processor are:

- Instruction set: RV32IM that includes standard integer instructions;
- 5 stages pipeline;
- Architecture optimized for FGPA;
- AXI4 and Avalon ready;
- Optional extension for hardware MUL/DIV instructions;
- Optional instructions and data caches;
- Optional MMU;

While the features implemented by the Murax SoC are:

- JTAG debugger;
- On-Chip RAM with 8 kB dimension;
- Interrupt support;
- APB bus for peripherals;
- 32 GPIO pins;
- One 16-bits prescaler, two 16-bits timers;
- One UART peripheral;

Because is particularly useful to the testing process, that will be discussed in chapter 4, the quantity of RAM has been reduced to 4kB. The license used for this core is MIT, it allows to use the core without restrictions similarly to the BSD license.

2.2 Implementation flow

The flow chart in image 2.2 represents the steps used to implement the final bitstream for the FPGA starting from the HDL files.

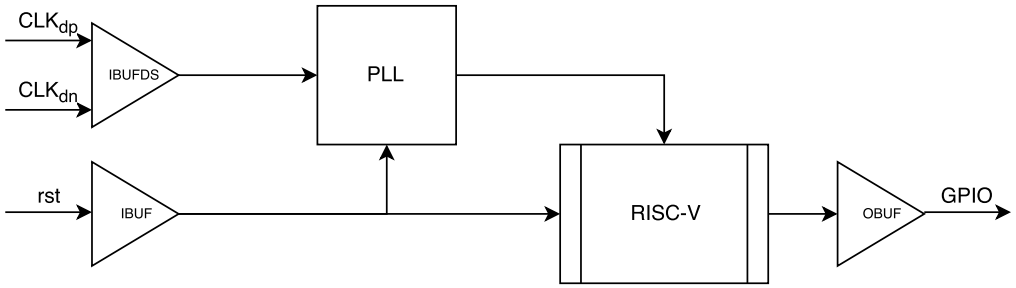


Figure 2.1. Top entity block diagram, the PLL receives in input the single-ended clock generated by IBUFDS and produces the 40MHz one supplied to the core.

2.2.1 Description

Being the test architecture a Xilinx FPGA, the tool normally used to synthesize and implement the design in a programmable bitstream is Vivado. Even if this program fully supports the XCKU040, Synplify from Synopsys has been used for the synthesis of the netlist, this is because is able to insert automatically inside the design the fault tolerance as explained in chapter 3. The file containing a full description of the core is *core.v*, the synthesis tool takes in input this document and produces a *.edif* file. This representation maps the whole circuit on the available resources of the FPGA. In order to reduce the quantity of hardware used, a directive is given to the compiler to map on BRAMs the hugest quantity of storage elements. Another advantageous aspect is that, having the RAM of the core mapped on this type of resources, is easier to initialize the instruction memory directly from the implementation flow and using tools specifically provided by Vivado. The other two units mapped on BRAMs are the streaming FIFOs of the serial interface and the register file of the core. A further directive imposed to the compiler removes the insertion of the I/O buffers, this allows to export the whole synthesized netlist and to import and use it directly in other designs. Synplify exports both the edif and the constrains generated post-synthesis in a *.xdc* file.

After that the core synthesized netlist is generated, a new project needs to be opened on Vivado; this second flow synthesizes the top entity where the PLL and the I/O buffers, needed to have the design working on the FPGA are declared. Because this project is developed in the framework of the ITS upgrade and will be tested even on the Readout Unit used to collect data from the experiment, the chosen operating frequency is 40MHz, the same used on the board. Finally, the core entity is synthesized as black box. To insert the core synthesized netlist inside the design compiled by Vivado, the black box is updated with the edif file generated by Synplify. Once that the project is completed with all his sub-components, the Place&Route process starts. The implementation phase place all the

cells to the related resources, on the targeted FPGA, then generates the routing logics needed to connect the different blocks inside the device. In order to program the FPGA and load the compiled code, the following files are needed:

- Memory Map Information (MMI) file: this file is an XML that preserves the informations on the BRAMs used for the RAM of the processor. It is composed by the following fields:
  - AddressSpace: defines a contiguous address space. The characterizing parameters are: name, begin, end. The name is the value used to distinguish each single AddressSpace while begin and end define the address space range. More AddressSpace can refer to the same address space but the name must be unique for each one of them;
  - BusBlock: an AddressSpace is composed by a finite number of BusBlocks. Each block refers to a part of the address space and the whole range is filled using the declaration order of the BusBlocks;
  - BitLane: each BusBlock is defined by the BitLanes. This parameter gives detailed informations on the BRAM mapped and on how the processor access the memory. The following values must be set for a single BitLane:
    * MemType: refers to the type of BRAM targeted, for Ultrascale technology can be RAMB36 and RAMB18;
    * Placement: locates the specific resource on the FPGA;
    * DataWidth: defines which portion of the data is related to the BRAM and is characterized by MSB and LSB variables;
    * AddressRange: set the address space portion for the BitLane;
    * Parity: set if the parity bits of the memory are used;
- Bitstream (BIT) file: contains the configuration bits for the CRAM of the FPGA. Through the bitstream every programmable resource of the device can be configured, not only the SRAM cells related to the LUTs, the routing logic or the I/O but even the initial values stored in the memory used;
- Memory (MEM) file: composed by the hexadecimal conversion of the binary generated by the C compiler. Each i_{th} line of the file is related to the word in the corresponding position of the BRAM;

Starting from the bitstream generated by Vivado, the configuration bits, related to the content of the BRAMs indexed by the MMI file, must be updated with the values of the MEM file. Usually, in order to program a microcontroller, the binary file generated by the software compiler is flashed inside the ROM memory of the device. For a soft-core the same mechanism could be used, a JTAG interface is provided and can be externally accessed using some of the FPGA I/O pins. However, the *Updatemem* tool from Xilinx allows to write program code inside the bitstream, this means that the access to the programming interface is not anymore needed to have the core correctly working and the related additional hardware can be removed. Another benefit is that, in the case of standalone systems without

an easy access of the internal resources, reprogramming the FPGA means automatically have all the resources of the processor correctly initialized; this is a very important feature for systems targeting high reliabilty. The C compiler related to the RISC-V ISA is GCC while the hexadecimal conversion is done using a custom C code.

2.3 Core verification and implementation results

The following table lists the resources occupied by the core and by the entity used to place the design on the FPGA:

Resource	Used (%)
LUTLUT	915 (0.4%)
FF	1008 (0.2%)
BRAM	3 (0.5%)
IO	14 (4.5%)
BUFG	1 (0.2%)
PLL	1 (5.0%)

Table 2.1. Resources used for Murax core with related percentages compared to XCKU040 resources.

These data state that the percentage of resources used is low and this helps to improve the cross-section of the core. The maximum frequency of the design is 213MHz, so, because the target frequency is 40MHz, the performances are compliant with the specifications. The 3 BRAMs are used for: processor RAM, serial interface FIFO and processor register file. The BUFGs and the PLL works for clock conditioning and distribution. Only one BRAM is used to implement the whole RAM, this is because one resource is able to store 4kB of data that is equal to the whole primary memory of the design.

To verify the correct behavior of the core, because is an already fully tested design in terms of RTL description errors, only the correct execution of a test code is checked. This software is a counter that writes the value in output using the I/O lines. The platform used for the verification process is provided by Vivado and called Debug Core, this powerfull tool gives the possibility to scope the internal signals of each design linking the desired nets to the Debug Core after the synthesis process and without changes in the RTL netlist. An added programming file with *.ltx* extension is provided and must be downloaded during the configuration process together with the bitstream. Using the scope interface provided by Xilinx, the waveforms are sampled and automatically sent to the PC thanks to the integrated serial interface. The sampling process can be triggered using the manual interface provided or the automatic functions available. The signals monitored to check the behavior of the system are the following:

- Program Counter: stores the address of the instructions read from the IRAM;
- Read data: the output data from the RAM;
- Output lines: the GPIO lines of the microcontroller;

The debug core is useful during the verification process but is a resources wasting method, for this reason it must be removed when the validation is finished.

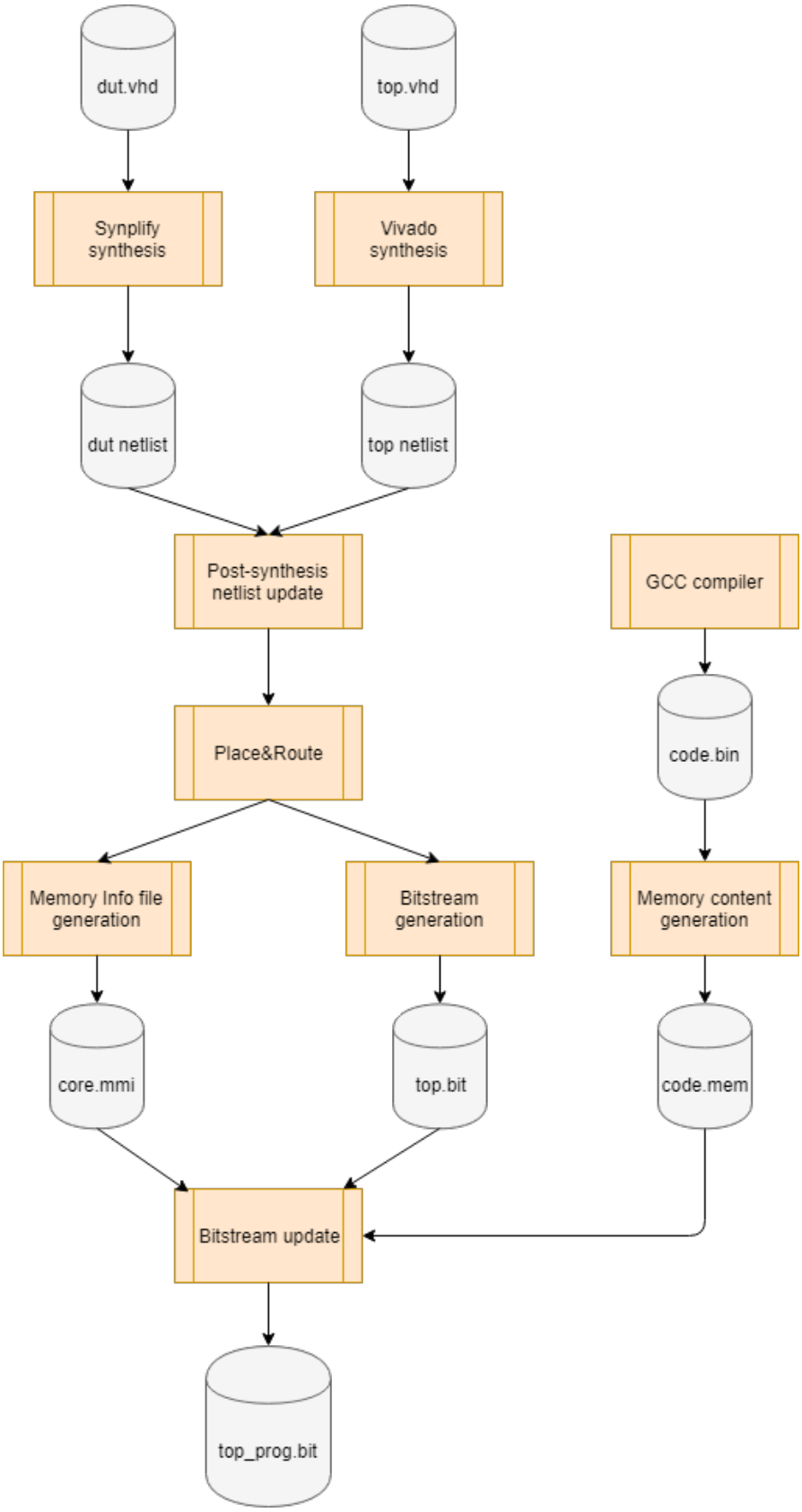


Figure 2.2. Core implementation flow diagram. It is both including netlist and software compile processes.

Chapter 3

Fault Tolerance

Once that the development flow needed to implement the processor on the target FPGA, in this case the XCKU040 from Xilinx, is designed and tested, the fault tolerance must be inserted in the core. The idea is to work on the synthesized netlist without affecting the RTL description, this is why the object will be to modify the design flow and to use some commercial tools, together with custom solutions, to ease this process. In this chapter at first there is a description of the most common techniques for hardware high reliability, then the modified design flow is presented and finally the results, in terms of validation of the new processors and of resources occupancy, are shown.

3.1 Fault Tolerance

In order to correctly describe the techniques used to harden the behavior of a design against external radiations, some concepts must be defined:

- Dependable system: able to mitigate a failure that has rate of appearance and effects too dangerous and not compliant with the specifications. This characteristic is divided in:
  - Reliability: ability to provide correct services after the failure;
  - Safety: ability to avoid catastrophic consequences on the users or on the environment;
- Reliable system: is the concept that better defines the fault tolerance, consists in the idea that each fault could manifest as system failure, for this reason it should be masked and, if possible, removed in order to avoid system misbehaviors;
- Safe system: similar to a reliable system because it acts detecting and, whenever possible, removing the fault. The main difference consists in the idea that some faults could produce an unexpected but harmless behavior;

An example could be useful to better understand the differences between the three concepts. Considering a general system that produces the outputs depending on some inputs and a general fault in the internal logic:

- A system that is not dependable would produce the wrong outputs;
- A safe system would produce the wrong outputs communicating the misbehavior to the user;
- A reliable system would produce the correct outputs;

In order to transform a non dependable system in a reliable one, some techniques can be applied to the technology used to implement the hardware or, during the design flow, modifying the netlist produced.

Having an hardened technology, able to withstand the external interference and to avoid the faults raising, guarantees to the designers that no SEU can arise on the hardware, this means that the design flow is independent from the application targeted. The main problems of this technique are related to:

- High costs needed to manufacture a radiation hardened component;
- Usually the technologies used are not advanced;

The techniques applied at design time are cheaper by order of magnitudes with respect to the previous ones and can use commercial powerful technologies for the realization of the final product. They are based on the concept of redundancy and are divided in subgroups depending on the type of resources that produces the overhead.

3.1.1 Hardware redundancy

The basic idea is to use more hardware to reach fault tolerance. There are three related subtechniques:

- Passive redundancy: the simpler scheme consists in having three different modules doing the same job, this is called Triple Modular Redundancy (TMR); the output of the system is obtained majority voting partial ones. If a module fails because of an upset in his internal logic than the error is masked thanks to the correct output provided by the other two blocks. When there is faults accumulation, because no technique is applied to correct them, then there could be multiple events arising in the internal logic of the modules and, finally, the system could output wrong values. Figure 3.1 shows a block diagram representing TMR;
- Active redundancy: another simple scheme built upon two replicas of the same module. One core is providing the outputs of the whole system while the second one is used as spare copy in order to check that no errors arise in the structure. To communicate this event, a comparator, that takes in input the outputs of the two module and that returns as result the presence of a misbehavior, is used. This technique implements error detection without error masking and consumes less area. This architecture is called Duplicate With Comparison (DWC). A more complex usage of active redundancy is called standby sparing and is divided in two different techniques:

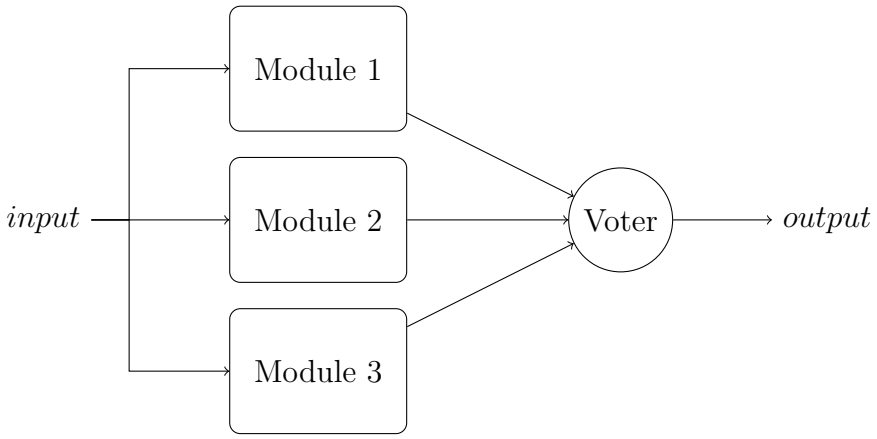


Figure 3.1. Block scheme of a system implementing Triple Modular Redundancy. Three hardware replica are shown together with output voter.

- Active standby: N DWC blocks work in parallel, taking the same inputs and providing the same outputs, and the output of one between them is used as return value of the system. Once that the selected module fails, another working module takes his place as "main" block. This method produces better results with respect to TMR in terms of error masking, because is able to mask N-1 permanent faults; the disadvantage is that introduces 2*N modules, against 3, not considering the overhead of the comparators and of the switch.
- Cold standby: N DWC blocks compose the system, one is working and the others are off. Once that an error is detected at an active module output, one of the switched-off blocks is turned on, the context is restored and it's selected as main component. It is better with respect to active standby for what concerns the power consumption but introduces overhead in the time reponse because of the restoring process;

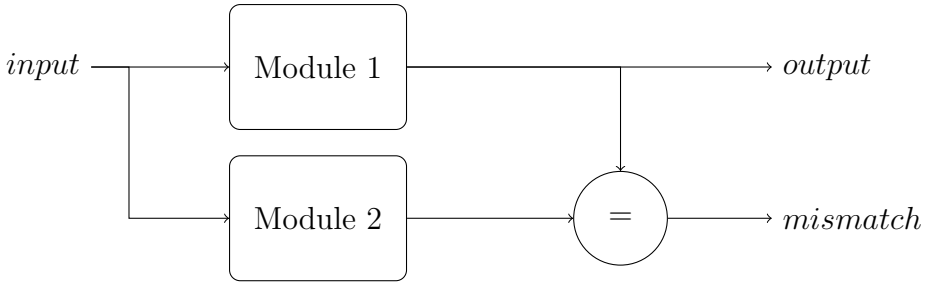


Figure 3.2. Block scheme of a system implementing Duplicate With Comparison. Two hardware replica are shown together with mismatch comparator.

- Hybrid redundancy: based on the two previous architectures mix, is composed by:
  - N active modules implementing DWC;
  - M spare modules implementing DWC;

- 1 output switch that selects three of the active modules;
- One output voter that takes in input the lines selected by the switch;
- A block containing the smart logic useful to substitute, when it fails, an active module with a spare one;

This solution is better than active and passive redundancy, for what concerns the resistance to SEUs, but introduces an high overhead in terms of resources used and power consumption.

3.1.2 Information redundancy

Information redundancy consists in adding informations to a stored data in order to detect and correct an upset that could affect the value, the redundant part will be a function of the original one. Usually the main system is composed by one block implementing the encoding part, the data is stored together with the redundant value in the memory and anytime that is read a decoder returns the correct value.

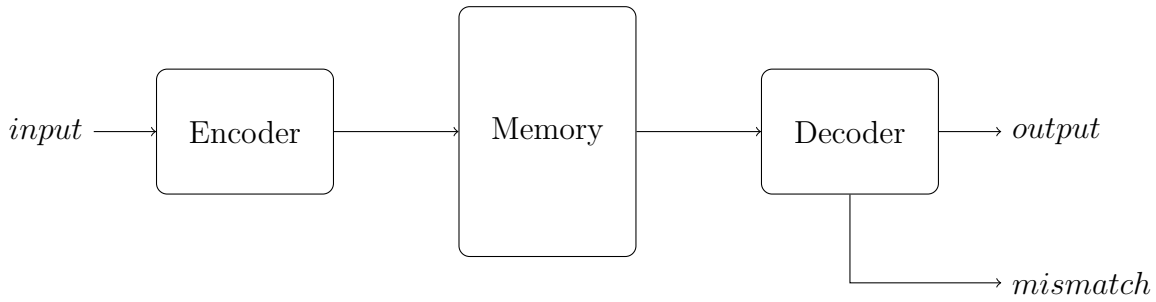


Figure 3.3. Block scheme of a system implementing Error Correcting Code. Memory inputs and outputs are elaborated by the encoder and the decoder, the latter produces final mismatch lines.

There are many techniques that differ in the algorithm used and the quantity of bits needed for the data redundancy. Two simple and known algorithms are:

- Parity: 1 redundant bit is added. The parity bit has positive value when the data, depending on the implementation, has an even or odd number of ones. This algorithm is only able to detect single errors in the frame;

- Hamming: the number of redundant bits is growing logarithmically with the width. Given an error in the stored value, once that it is read and the redundant bits are recomputed they will be different from the original ones. Comparing the two redundancies, the error can be located in the frame. This algorithm is a SEC-DED, this means that is able to correct a single error in the data and to detect them in the case of a double event;

3.1.3 Time redundancy

It is based on the idea of introducing overhead reiterating the computation of the results. A simple algorithm introducing time redundancy is the following:

- Taking input values;
- Evaluating the outputs and storing them as O1;
- Evaluating again the outputs and storing them as O2;
- Comparing them;
- Raising an error signal if a mismatch is detected;

This technique can be easily implemented in software, for this reason is considered as the best, in terms of power consumption, among the presented ones.

3.1.4 Design choices

Depending on the intensity of the flux hitting the device during the service life, specific techniques must be applied in order to harden the design against external radiations. Analyzing carefully these data allows to avoid the overdesign that would waste resources on the FPGA. At first the SEU arising rate in the device must be estimated, the following data are needed:

- Device cross-section: for Xilinx Ultrascale FPGAs the CRAM and BRAMs cross-sections are expressed in section 1.2.3;
- Radiation Flux: the flux generated by the LHC and hitting the device;
- Operating time interval: total time in which particles are hitting the device;
- Number of bits: number of configuration bits for the FPGA;

The formula used to evaluate the number of events given the previous conditions is

$$N_{events} = \sigma \cdot \int_0^{\Delta t} \Phi dt \cdot n_{bits} \tag{3.1}$$

integrating flux over time the fluence (Φ_t) is obtained:

$$N_{events} = \sigma \cdot \Phi_t \cdot n_{bits} \tag{3.2}$$

considering a constant flux over the time interval:

$$\Phi_t = \Phi \cdot \Delta t \tag{3.3}$$

Using the following data and the previously mentioned formula the SEU rate can be evaluated for the targeted device:

- Flux equal to $1\text{kHz}\,\text{cm}^{-2}$;
- Operating time interval of 24hour;
- Number of bits equal to $1.28 \cdot 10^8$bits (considering the XCKU040);

$$\Delta t = \frac{1}{\sigma \cdot \Phi \cdot n_{bits}} = 51.06\text{min} \tag{3.4}$$

This means that is hard to accumulate errors inside one FPGA. Having normally just one error affecting the device, TMR technique should be enough to protect the core. If one error compromises the behavior of a module, the other two continue to work properly. Obviously, there are sensitive points that would break the design if a SEU happens in the related CRAM but their cross-sections are a lower percentage of the total one. Even if the TMR should protect sufficiently the processor memory, in order to fully protect the instructions stored, Hamming code is used for SEC-DED implementation. The target is only the IRAM because the other modules using storage elements, like the pipeline or the internal register file, are using much less resources and so have a lower σ. In any case, the content of these storage elements is not fixed as for the code memory but, most of the times, is rewritten during the execution of the program and this means that the real interval of time in which the errors could accumulate inside the device is lower than the one of the general design. To implement the ECC two methods could be used, through the internal circuitry of the BRAMs or designing the memory system by scratch.

Internal ECC Xilinx is providing all the BRAMs with an input encoder and an output decoder implementing a SEC-DED algorithm to protect data. This solution is for sure the one that uses less resources to preserve values but has the following problems:

- The width of a single word must be 64-bits;
- It is not allowed to use the byte write mode of the BRAMs;
- It's not possible to initialize the parity bits using the Updatemem tool from Xilinx;

A complex circuitry could be needed to solve the first two problems, this is because is necessary to perform the write process handling at runtime while, the initialization process of the memory, requires a custom circuitry that writes the instructions inside the BRAMs when the system is restarted. A block diagram representing the built-in ECC is shown in picture 3.4

Custom ECC The Hamming code is implemented using custom circuitry and parity bits are stored in a dedicated memory. A lot of open designs are implementing ECC algorithms and can be easily cloned, this means that only the modules needed to handle the byte wide write process and the update of the redundant bits must be designed. Using this approach both the instructions and Hamming bits can be initialized using the Updatemem tool. This solution is more flexible than the previous one and independent from the FPGA used.

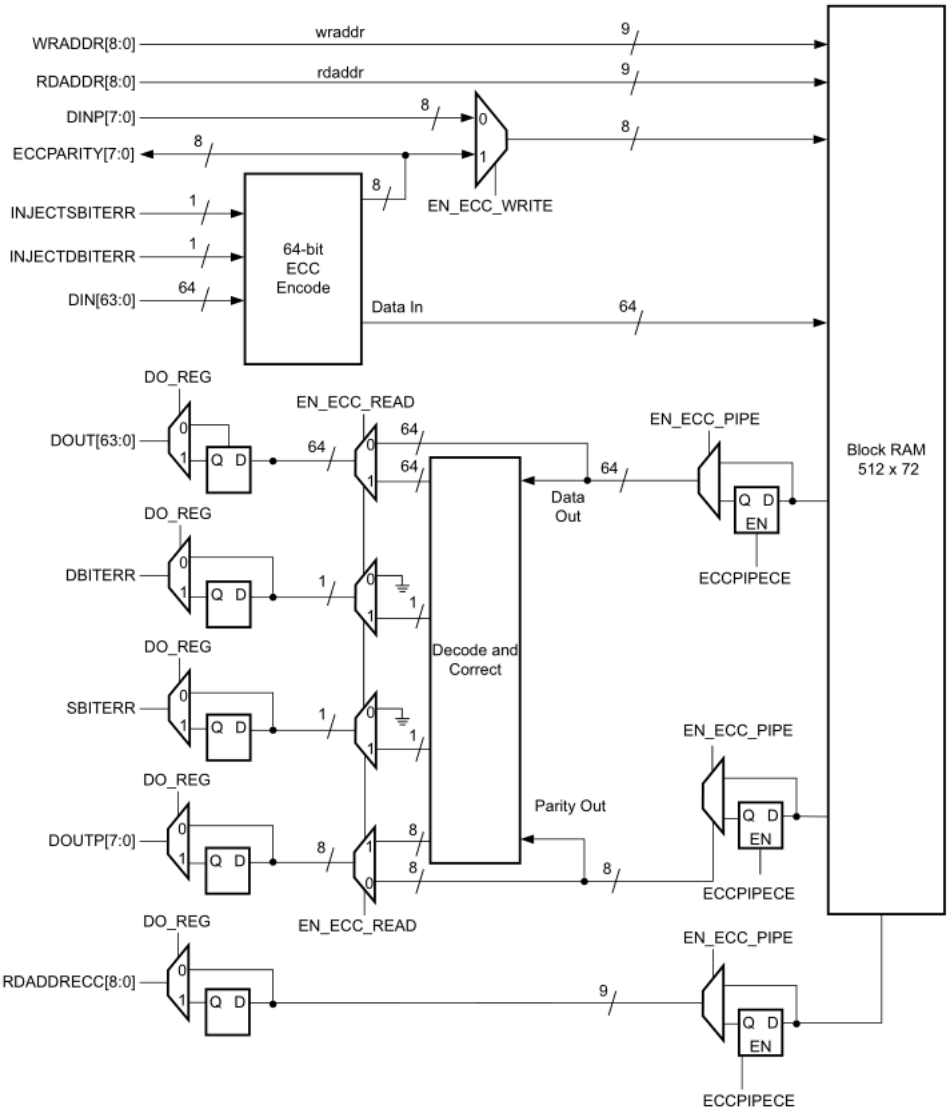


Figure 3.4. Internal structure of the built-in ECC for Ultrascale BRAMs, the input and output data have a 64 bits parallelism. The scheme implemented by Xilinx is similar to the one in figure 3.3

3.1.5 Design automation

Synplify is a powerful tool able to automatically implement fault tolerance techniques in the design, the limits of this program are related to the technology targeted. The following solutions are available for Ultrascale family:

- Hardware redundancy:
  - Distributed TMR: triplicates the modules, insert a voter in each sequential loop and at the outputs. Allows optional voting for each internal register;
  - Block TMR: triplicates the modules and insert a voter at the outputs;

  - Local TMR: triplication applied on sequential elements with the output voter;
  - DWC;

- Information redundancy: the ECC can be automatically inserted with some limitations:
  - Byte wide write enable is not supported;
  - The set/reset signal of the BRAMs must be aynchronous;
  - The clock enable for the output register must be removed;
  - Block RAMs explicitly instantiated with built-in parity bits enabled;

The hardware redundancy will be inserted using Synplify while the information redundancy will depend on a custom design and then will be added replacing the normal memory system with the netlist of the protected one. The technique used to protect the core is distributed TMR, this is because it automatically implements the logic needed to restore the state of the circuit after an event hits the sequential blocks.

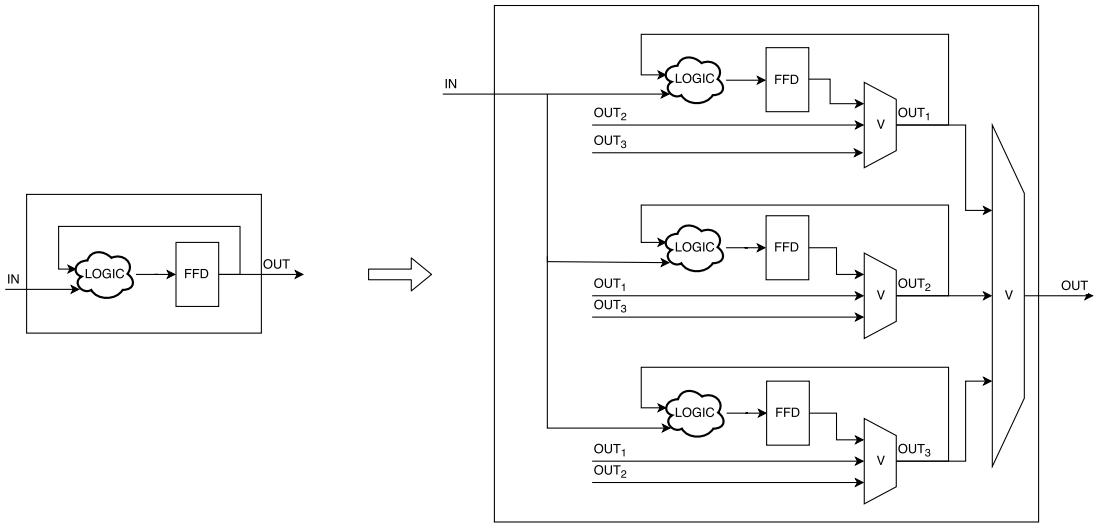


Figure 3.5. Distributed TMR with sequential loop voting. The main difference with the scheme shown in figure 3.1 is the double step voting logic that includes even the sequential loops.

3.2 Memory design

The purpose of this memory system is to implement information redundancy with custom circuitry, the design differs from the simple scheme presented in figure 3.3 because some logic is added in order to support byte wide write process. Analyzing the post-synthesis netlist, some characteristics can be derived for the processor RAM:

- Data are accessed through only one port that is in read first mode, this means that if a read and a write are enabled than the output value will be the content of the cell before the change;
- Synthesizing the processor with a RAM of 4 kB, only one BRAM is used and the port allows to acces frames of 32 bits;
- Byte wide write enable is supported and each byte of the input line will be a replica of the targeted value. The single byte in a frame is addressed using the write enables of the BRAM;
- The Read Enable is tied to '1';

The new memory system needs these characteristics in order to replace the old one and must implement the ECC trasparently to the read and write processes. A block diagram of the design is shown in image 3.6.

Both memories used to store the data and the parity bits are implemented using BRAMs but they have different behavior, the first one is a Simple Dual Port memory, where port A is used and the other one is disabled, the second one is a True Dual Port memory, where both the accesses to the component are used independently. The usage of a more complex structure for the ECC memory allows a real time evaluation of the parity bits, here is how the two ports control circuitries behave:

- Port A: the first output port is directly connected to the input of the decoder, this means that is basically used only for read operations except when the same location is addressed for both read and write processes. In this particular case the new value is written through Port A and will be forwarded to the output, for this reason is configured in WRITE_FIRST mode;
- Port B: the second output port is left floating, this means that is used only in write mode. This access to the memory is not used if no data must be stored and if the parity bits are needed the same clock cycle, in this case the other port is used because is considered as a violation trying to obtain a value from a port writing through the other one;

Here is a description of read and write operations:

- Write:
  - Cycle 1: the input value is directly written inside the data memory and is stored in a register, WEA values are preserved inside flip-flops;

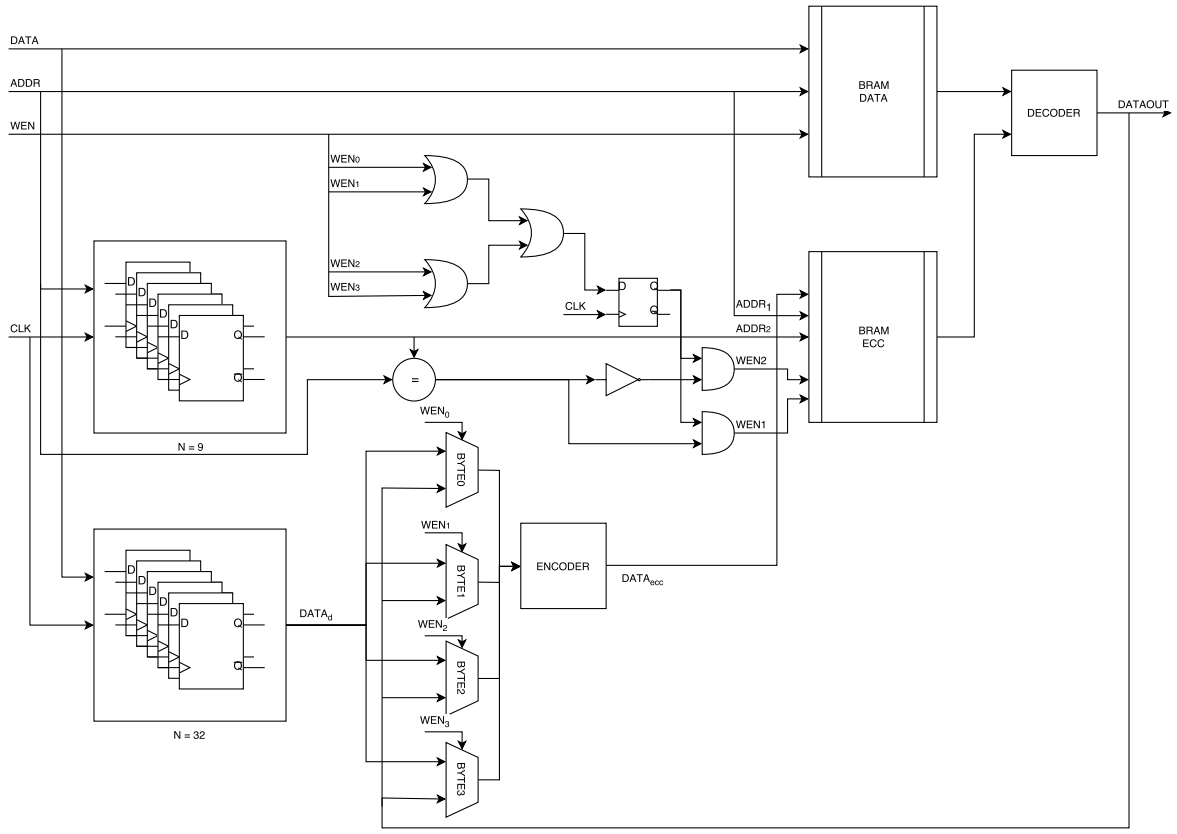


Figure 3.6. Memory system schematic, the output data is depending on values read from DATA and ECC BRAMs.

– Cycle 2: the output of the data memory is looped back, an array of multiplexers merges the old data and the one stored in the register to recreate the written value, the result is given in input to the encoder. Finally the parity bits are evaluated and stored inside the ECC memory, the write port is enabled using a signal generated from the bit by bit or of stored WEA lines;

- Read: the value read is the output of the decoder that takes in input the frames obtained from data and ECC memories;

Because the first memory is READ_FIRST, by specifications, when a write operation is performed the new value can be read only after one cycle; using this mechanism to store the parity bits allows to have them available as soon as the related frame can be read.

The selected Error Correction Code is a SEC-DED that uses 7 redundant bits, it is implemented through the usage of the Encoder and the Decoder shown in the previous schematic.

Encoder Given d as the input data and p as the output parity bits, these tables are representing the logic functions implemented by the encoder. Each result bit is the xor of all input bits signed with an "x".

	p0	p1	p2	p3	p4	p5	p6		p0	p1	p2	p3	p4	p5	p6
d0	x	x	x					d19	x	x			x	x	
d1	x	x		x				d20	x		x		x	x	
d2	x		x	x				d21	x	x	x		x	x	
d3	x	x	x	x				d22	x			x	x	x	
d4	x	x			x			d23	x	x		x	x	x	
d5	x		x		x			d24	x		x	x	x	x	
d6	x	x	x		x			d25	x	x	x	x	x	x	
d7	x			x	x			d26	x	x					x
d8	x	x		x	x			d27	x		x				x
d9	x		x	x	x			d28	x	x	x				x
d10	x	x	x	x	x			d29	x			x			x
d11	x	x				x		d30	x	x		x			x
d12	x		x			x		d31	x		x	x			x
d13	x	x	x			x		p1	x						
d14	x			x		x		p2	x						
d15	x	x		x		x		p3	x						
d16	x		x	x		x		p4	x						
d17	x	x	x	x		x		p5	x						
d18	x				x	x		p6	x						

Table 3.1. ECC equations, all the bits marked with 'x' are associated to the column parity bit.

Decoder The decoder works in the following way:

- The input data is encoded again;
- The syndrome is computed:
  - Bit 1 - Bit 6: computed as the differences between the new and the stored redundant values
  - Bit 0: negative if the parities of the redundant values are equal;
- If bit 0 is equal to 1 then this means that just one error affected the data while, if it is 0 and the others are not 0, then there is a double fault;

3.3 Implementation flow

To insert the new memory system inside the core structure, a modified version of the previous flow, shown in image D.1, is used.

3.3.1 Description

The new developed framework allows to choose which of the four possible configurations, depending on the fault tolerance techniques selected, is synthesized. TMR addition is not changing the implementation flow because the whole netlist elaboration is automatically performed by Synplify, the ECC, instead, introduces some additional steps:

- A second synthesis is performed with Synplify in order to obtain the netlist of the protected memory system, the generated circuit will substitute the BRAMs used in the raw processor through some scripts that automatically recognize the RAM and perform the cut and paste process;
- In order to correctly update the new RAM, not only the compiled code but even the related parity bits for each words must be at first computed and then added in the initialization process. To generate the memory content for ECC BRAMs the binary program is taken, parsed and encoded using the same logic functions implemented by the hardware components;

3.3.2 Implementation results

The next table shows the results in terms of resources usage for the different core configurations:

Resource	None	ECC	DTMR	DTMR-ECC	Available
LUTLUT	915 (0.4%)	942 (0.4%)	7965 (3.3%)	8129 (3.4%)	242400
FF	1008 (0.2%)	1022 (0.2%)	3374 (0.7%)	3440 (0.7%)	484800
BRAM	3 (0.5%)	4 (0.7%)	9 (1.5%)	12 (2.0%)	600
IO	14 (4.5%)	14 (4.5%)	14 (4.5%)	14 (4.5%)	312
BUFG	1 (0.2%)	1 (0.2%)	1 (0.2%)	1 (0.2%)	480
PLL	1 (5.0%)	1 (5.0%)	1 (5.0%)	1 (5.0%)	20

Table 3.2. Resources used by the different core versions with percentages referred to XCKU040 FPGA.

No significant overhead is introduced by the ECC in the design, a low amount of LUTLUT and Flip-Flops is used with the addition of just one BRAM, as expected. The Distributed TMR is generating a huge overhead, this is because, with respect to other TMRs, this technique votes all the internal registers and the sequential loops, in order to restore the processor state. Flip-flops are three times the original ones while LUTLUTs are more than eigth times the ones of the normal processor, this parameter expresses the huge amount of resources needed by the voting logic. As expected, even the BRAMs are three times the

original ones considering both the DTMR and the DTMR-ECC versions. The overhead introduced by the protected memories is low even in the DTMR core version.

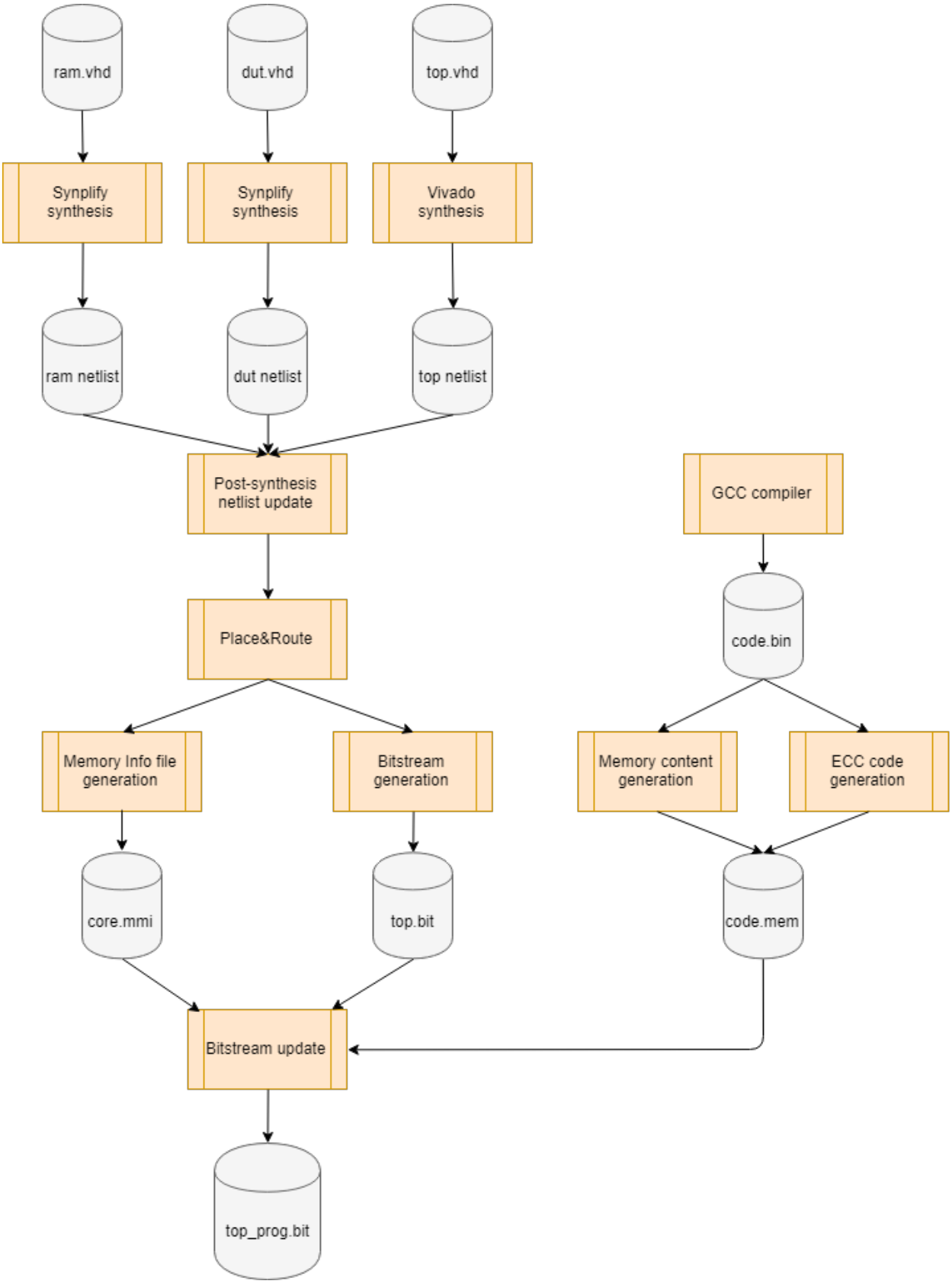


Figure 3.7. Core protection implementation flow diagram. Both hardware and software compile processes are taking into account the new memory system.

Chapter 4

Core characterization

Obtained the final flow able to synthesize fault tolerant cores, with different techniques, the next step is to describe how these designs can be characterized in terms of ability to resist against external radiations. At first the metrics, that means how the different processors statuses are classified and which algorithm is executed, are defined. The second topic covers the design of the testing circuitry and the related implementation flow, then the different environments used for cores characterization are described. Finally the results are reported and discussed comparing the data obtained for RISC-V cores with the one collected for the radiation hardened version of Microblaze processor, designed by Xilinx.

4.1 Metrics

The reliability of a system is usually modeled by the following exponential equation:

$$R(t) = e^{-\lambda t} : \tag{4.1}$$

where λ is the failure rate and t is the time instant considered. To evaluate the Mean Time To Failure (MTTF) the expression used is:

$$MTTF = \frac{1}{\lambda} \tag{4.2}$$

For each processor version a reliability function is estimated, to reach this object the MTTF will be obtained through experimental measures using different testing tehcniques. The Mean Time To Failure is the expected quantity of time between two different system misbehaviors and its measurements will depend on the test environment where data are collected. Before data taking process discussion, the possible processor statuses in radiation environments are described. They will be classified in the following way:

- Run: the processor is correctly running and no error is produced in output;
- Failure: the processor continues to execute the algorithm but the output values are not correct;
- Critical failure: the processor stops or is in failure state for a not negligible amount of time;

The experimental MTTF is measured as the time between two critical failures. In order to have a reliable estimation of this value, an extensive testing process, that allows to take a lot of data, is needed.

4.1.1 Test environments

As stated before the data taking process and the related statistics depends on the test environment. There are basically two different methods used to characterize the design:

- Fault Injection;
- Test with particles beam;

In both cases the system will recreate the same operating conditions of the Readout Unit designed for ITS upgrade. These are the most important characteristics:

- The fault rate for the design is of 1 each 51.06min with an external flux equal to $1 \cdot \mathrm{kHz\,cm^{-2}}$;
- Scrubbing, that means rewriting the FPGA CRAM during the system operational life to avoid faults accumulation, is performed each 3 s
- A complete FPGA reprogramming process is done each 30 min;

There are two different types of scrubbing:

- Blind scrubbing: the whole CRAM is continously rewritten;
- Partial reconfiguration: if a SEU happens in a particular section of the CRAM, it is corrected addressing the single memory location;

For Xilinx FPGAs the smallest amount of CRAM that can be addressed externally is a frame. Depending on the family considered, the width, in terms of 32-bits words containted, is different. For Ultrascale technology is 123 words.

Fault Injection It means to manually flip a bit in the configuration memory, this method is used to emulate SEUs in FPGAs CRAM (Maybe even BRAMs). Ultrascale family allows to access directly this memory through different interfaces, a detailed explanation is present in appendix A.1. An external device called JTAG Configuration Manager (JCM), produced by the BYU group, allows to use the JTAG interface for fault injection purposes. Starting by the bitstream related to the design that will be tested, the fault injection process works in the following way:

1. A frame, a word and a bit are selected randomly;
2. The bit value is flipped inside the golden bitstream;
3. The corrupted configuration is programmed inside the CRAM of the device through the JTAG port;

4. Restarts from point 1;

The stated operational conditions are implying the presence of a scrubbing process each 3 s. Because considering the previously detailed operating conditions there is one fault each 51.06min, is safe to assume that at least one scrubbing process happens between a SEU and the following one. To emulate this behavior during the fault injection process, blind scrubbing is performed after the bit flip. To increase the quantity of data taken for each processor typology and avoid FPGA systematic effects, for each fault injection session N processors will run in parallel on the device. The testing process will work in the following way:

1. The FPGA is programmed with the targeted bitstream;
2. The JCM injects a fault;
3. The JCM scrubs the CRAM;
4. The processors statuses are monitored:
   - If all of them are not working anymore or if the testing circuitry implemented has problems the test stops, data on the number of faults survived by each DUT are stored and the process restarts from point 1.
   - If at least one of the core tested is still correctly behaving, the process continues from point 2.

An iteration of the previous loop that starts from point 1 is called run. Each fault survived is corresponding, statistically, to an interval of time equal to 51.06 min. For each core replica the total time survived is:

$$\Delta t_i = N_i \cdot \Delta t_f \tag{4.3}$$

where N is the number of faults survived and Δt_f is the time between two faults. The MTTF can be experimentally evlauated in the following way:

$$MTTF = \frac{\sum_{i=1}^{M} \Delta t_i}{M} = \Delta t_f \cdot \frac{\sum_{i=1}^{M} N_i}{M} \tag{4.4}$$

where M is the number of replica tested.

Once that MTTF is computed both the reliability function and cores cross-sections will be evaluated. The testing system structure is shown in picture 4.1.

Test with particles beam This methodology is using a real flux of neutrons to test design strength. In order to perform this test the design must be integrated in the RUv1 firmware, the main differences are:

- The FPGA is already full so only one replica for each core type can be inserted inside the design, this reduce the quantity of taken data;

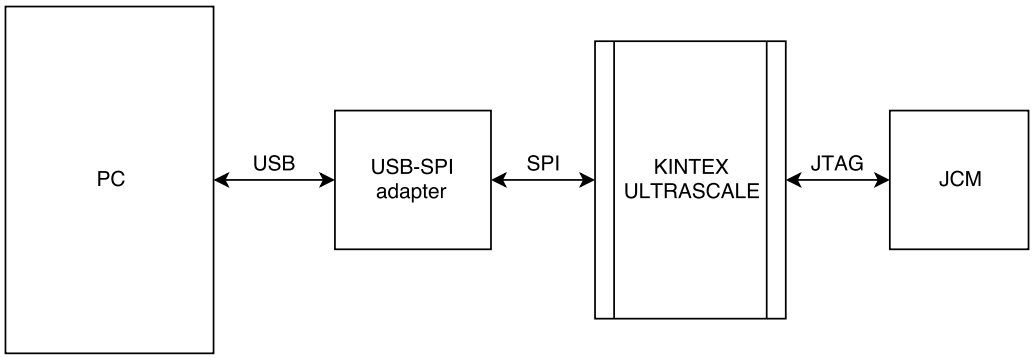


Figure 4.1. System architecture for tests with fault injection, the external devices used are talking to the FPGA using SPI, for the PC, and JTAG, for the JCM.

- The SPI interface will communicate with SCA SPI controller, so needs new low level drivers.
- Data will not be composed only by run counters and error counters but even by time instants, this means that MTTF will depend on real values;
- The quantity of SEUs affecting the FPGA is not known but will be statistically evaluated;

During test execution the flux will not be constant during each session, for this reason the measured fluence is used to obtain the quantity of events (equation 3.2). Starting from N_{events} estimation the previous formula can be used to evaluate the MTTF under beam conditions proper to the ALICE experiment. The whole setup is compliant with the same specifications stated for fault injection process and is represented by the image below:

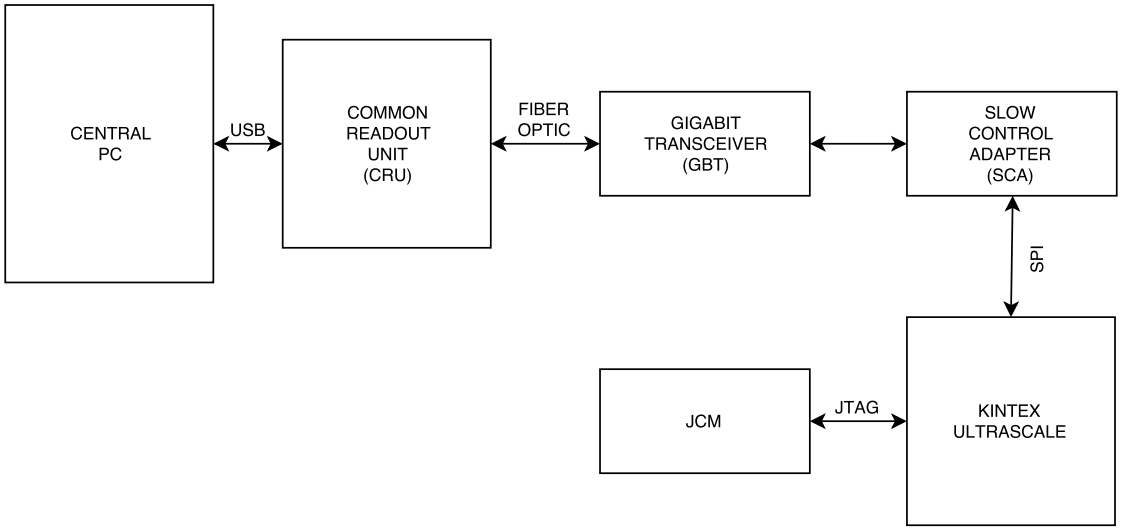


Figure 4.2. System architecture for tests with particles beam, the main difference with respect to fault injection architecture is the usage of SCA and CRU to communicate with central PC.

Algorithm Because the monitoring process is performed only through outputs reading, an algorithm that is strongly stressing the internal resources of the core is needed, this is because an error in the internal logic would be easily propagated and observed if all the processor units are frequently used. An algorithm that respects all these features is the Advanced Encryption Standard. The version implemented is encrypting a 128 bits word starting from a 128 bits key that remains constant for each data word, output frame width is equal to the previous ones. A more detailed explanation of the algorithm is performed in Appendix C.1.

4.2 Testing circuitry

In order to correctly implement the chosen algorithm for tests, a specific architecture is necessary. The following picture shows a block diagram of the testing circuitry:

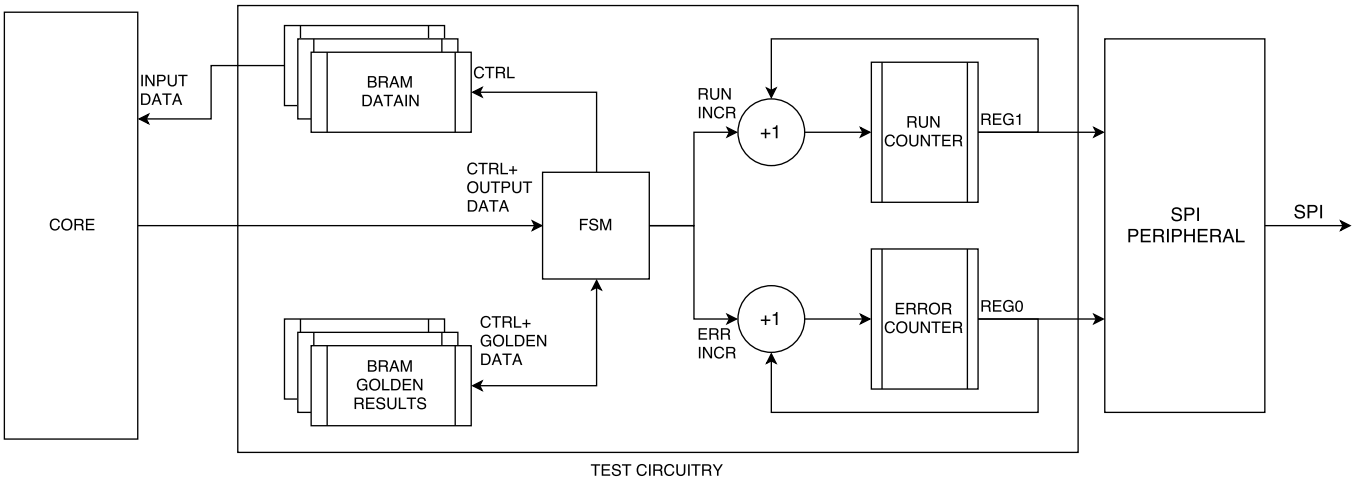


Figure 4.3. Test circuitry block diagram, the FSM is communicating with BRAMs and handling both output registers, to SPI, and data for processor testing.

Looking at the schematic, is easy to divide the design in four different areas:

- Device Under Test (DUT): a processor;
- Storage Elements (SE): components containing the text that will be encrypted and the encoded version;
- Finite State Machine (FSM): handling data exchange with the processor and state counters;
- Communication: circuitry implementing the communication interface needed to retrieve data;

In order to fully protect the testing circuitry against external radiations, the design is triplicated using the same technique implemented for RISC-V cores, that is DTMR.

Storage Elements The different data are contained in two different BRAMs, one for source data and the other containing the golden results of the encryption process. These resources are directly driven by the FSM that is addressing the different memory locations. Both the components are only in read mode, no data is written inside during the testing process. DTMR is used to protect the whole design, this means that even BRAMs are triplicated and voted in output. In addition to the previous technique, the two storage elements are replaced with ECC protected memories equal to the ones used to protect the cores IRAM. Hardening the testing circuitry this much is fundamental because it should not fail frequently during tests, but introduces the disadvantage of having a huge overhead in terms of BRAMs used; this will limit the processors tests parallelization.

Finite State Machine It is the smart part of the testing circuitry able to provide data to the processor, to retrieve the encrypted one and to increase the error and run counters read. The algorithm implemented by the FSM and the processor is the following:

1. The circuit is idle, waiting that the processor activates output enable lines;
2. The core writes those outputs, the FSM reads one 32 bits word from the source BRAM and returns it to the processor. This step is repeated 4 times;
3. The circuit is idle, waiting for the encryption process end;
4. The core writes those outputs, the FSM reads one 32 bits word from the destination BRAM and compares it to the one returned by the processor. This step is repeated 4 times;
5. The result of the comparison process is taken, and accordingly both the run counter and the error counter are incremented. The former increments each time an encryption ends while the latter each time there is a mismatch between the data provided by the processor and the golden value;
6. Restarts from point 1;

The FSM is protected by DTMR technique.

Communication The communication protocol used to retrieve data from the FPGA is the Serial Peripheral Interface (SPI), that is one of the most common standard for wired communication. There aren't particular specifications related to the minimum bitrate required, this protocol has been selected because allows an easy integration in the structure of the Readout Unit. As described in A.2.1, the Slow Control Adapter (SCA) is a radiation hardened Integrated Circuit (IC) used to implement high speed communication between the external readout electronics and the components on the Readout Unit. SPI is one of the available controllers. This controller is used for test sessions in environments with radiations. When fault injection is used, together with the development kit, some GPIOs are programmed as SPI pins and used together with a USB to SPI dongle. The interface inserted in the FPGA firmware is directly taken from an open design and is adapted in order to use it properly in the test architecture, this means that is reading 2 counters for each one of the processors present in the design. Appendix **??** is providing a detailed description of this unit. The algorithm implemented, by the FSM handling the SPI interface, allows to use a full duplex communication where the master, that is the external device, sends a register address that refers to a counter and, in the meanwhile, the slave, that is the design contained in the FPGA, returns the value referred to the previously addressed data. This module is protected through DTMR.

4.3 Implementation Flow

The flow used to obtain a running test architecture on the FPGA is similar to the one obtained for fault tolerant cores and is shown in image 4.4

4.3.1 Description

The main flow characteristics are:

- DUT netlist is changing depending on the type of test:
  - Fault injection: as discussed before, a huge parallelization of data taking process is introduced using multiple instances of the same core in the design. The DUT entity is composed by a SPI interface and twenty submodules each one including one testing circuitry;
  - Particles beam: because of the previously mentioned problems with the available resources on the FPGA, DUT entity is composed by one test module for each core;

  Core netlists depend on the test implemented;
- The testing circuitry storage elements use the same protected memory designed for processors. The content must be initialized to store the source data and the encrypted ones. Differently from core implementations, this BRAMs content is static. To remove the time overhead, introduced during bitstream update, memories are synthesized directly with initial words already configured, this means that edif files already contain all the informations required.
- Because a reliable process is needed, the testing circuitry is protected with Distributed TMR similarly to processors.

This complex implementation flow is optimized in terms of performances, only FSMs, the SPI interface and some glue logic is really synthesized. All the BRAMs and the processors are configured as black box, this means that the huge percentage of the compilation load is removed because they are all pre-synthesized netlists.

4.4 Results

Extensive Fault Injection usage allows to create a good statistic. The total number of runs (N_r) performed for each design is 1000, and one single run contained 20 processors (N_p). Considering the equation 4.4:

$$MTTF = \Delta t_f \cdot \frac{\sum_{i=1}^{N_r \cdot N_p} N_i}{N_r \cdot N_p} \tag{4.5}$$

To estimate the relability curve λ is needed:

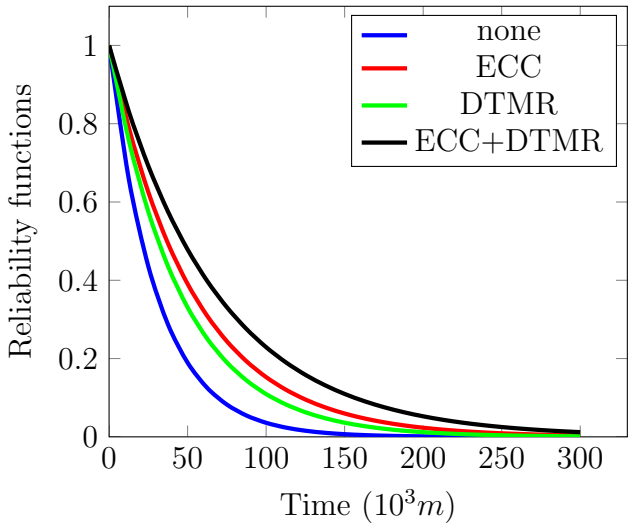
$$\lambda = \frac{1}{MTTF} \tag{4.6}$$

And the reliability function depending on this parameter is expressed by formula 4.1. The table below shows the MTTF obtained for the different versions of the core using fault injection:

	None	ECC	DTMR	ECC-DTMR
$\%_{faulty}$	35 %	37 %	39 %	47 %
$\%_{stopped}$	65 %	63 %	61 %	53 %
MTTF (10^3min)	30.06	53.19	45.18	67.85

Table 4.1. Murax core characterization with fault injection, the table is showing percentages related to faulty and stopped processors added to MTTF evaluation.

The graph below shows the reliability curves:



Tests on a real environment have just one processor present in FPGA firmware, this means that the total cores replica are the same as the number of runs. Having a low amount of data from particles beam tests, MTTF won't be estimated but the other parameters are shown in the following table:

	None	ECC	DTMR	ECC-DTMR
$\%_{survived}$	54.54 %	69.31 %	93.06 %	99.01 %
$\%_{faulty}$	28.28 %	18.81 %	5.94 %	0 %
$\%_{stopped}$	17.17 %	11.88 %	0 %	0.99 %

Table 4.2. Murax core characterization with particles beam, differently from fault injection the percentages of cores that survived each run is shown.

4.4.1 Results evaluation

The quantity of data taken with fault injection is a lot higher than the ones obtained through testing with particles beam, so statistics obtained with the former method should be more reliable than the other ones. Looking at MTTFs, the results are compliant with expectations, all the protected versions are resisting better to SEUs than the basic one. However, the low quantity of resources used by the not radiation hardened version allows an MTTF that is higher than expected and states that it could be used in environments with this flux intensity. Talking about the protected versions, the DTMR is behaving worst than the ECC. Using a technique such as Distributed TMR could be really weigthful in terms of logic overhead, instead the ECC technique is adding a small quantity of blocks to the synthesized netlist. All these considerations explain why the DTMR design is behaving worse than the ECC one. However the situation is reversed looking at the statistics obtained through tests with particles beam, the triplicated version is lasting much more than the one implementing only information redundancy. These differences between data obtained could possibly be caused by:

- Sistematic effects: this means that having just one replica synthesized that uses always the same resources on the FPGA could lead to some constant weaknesses in each core. To correctly test each processor it is needed to distribute it through the whole FPGA, as done for fault injection tests.

- BRAMs errors: theoretically, the probability of having an error in a Block RAM is higher than the one related to CRAMs, almost two times. The JCM is injecting faults through the whole FPGA with the same probability, this means that the quantity of errors happening inside core memories is different with respect to the real one.

4.4.2 Comparison with other projects

The behavior of the obtained netlist is compared with the MicroBlaze designed by Xilinx in terms of reliability. Results are weighted with respect to quantity of resources occupied. Table 4.3 is showing some core implementation data. The blocks used by the protected solution are higher compared to the ECC-DTMR version of murax core, however BRAMs are lower considering that the total RAM has a double storage capability. Table 4.4 shows the related MTTFs obtained with fault injection for Microblaze, while the table 4.5 shows the results obtained with particle beams tests.

Resource	None (%)	TMR
LUTLUT	2665(1.1%)	9933(4.10%)
FF	3086(0.64%)	11652(2.40%)
BRAM	2(0.33%)	6(1.00%)
DSP	3(0.16%)	9(0.47%)

Table 4.3. Resources used for MicroBlaze processor and related percentages related to XCKU040 FPGA.

	None	TMR (M)
$\%_{faulty}$	45.82 %	0.02 %
$\%_{stopped}$	54.18 %	99.98 %
MTTF (10^3min)	29.75	11.11

Table 4.4. Microblaze characterization with fault injection

	None (M)	TMR (M)
$\%_{survived}$	52.53 %	45.55 %
$\%_{faulty}$	34.34 %	5.94 %
$\%_{stopped}$	13.13 %	48.51 %

Table 4.5. Microblaze characterization with particles beam

Generally the Microblaze is able to fastly execute the algorithm, however this difference is vanished by the greater endurance of Murax protected versions. From MTTF results is easy to say that the custom core is lasting more than the solution proposed by Xilinx, however most of the times the latter is stopping without producing faulty results; this means that it is a safer solution.

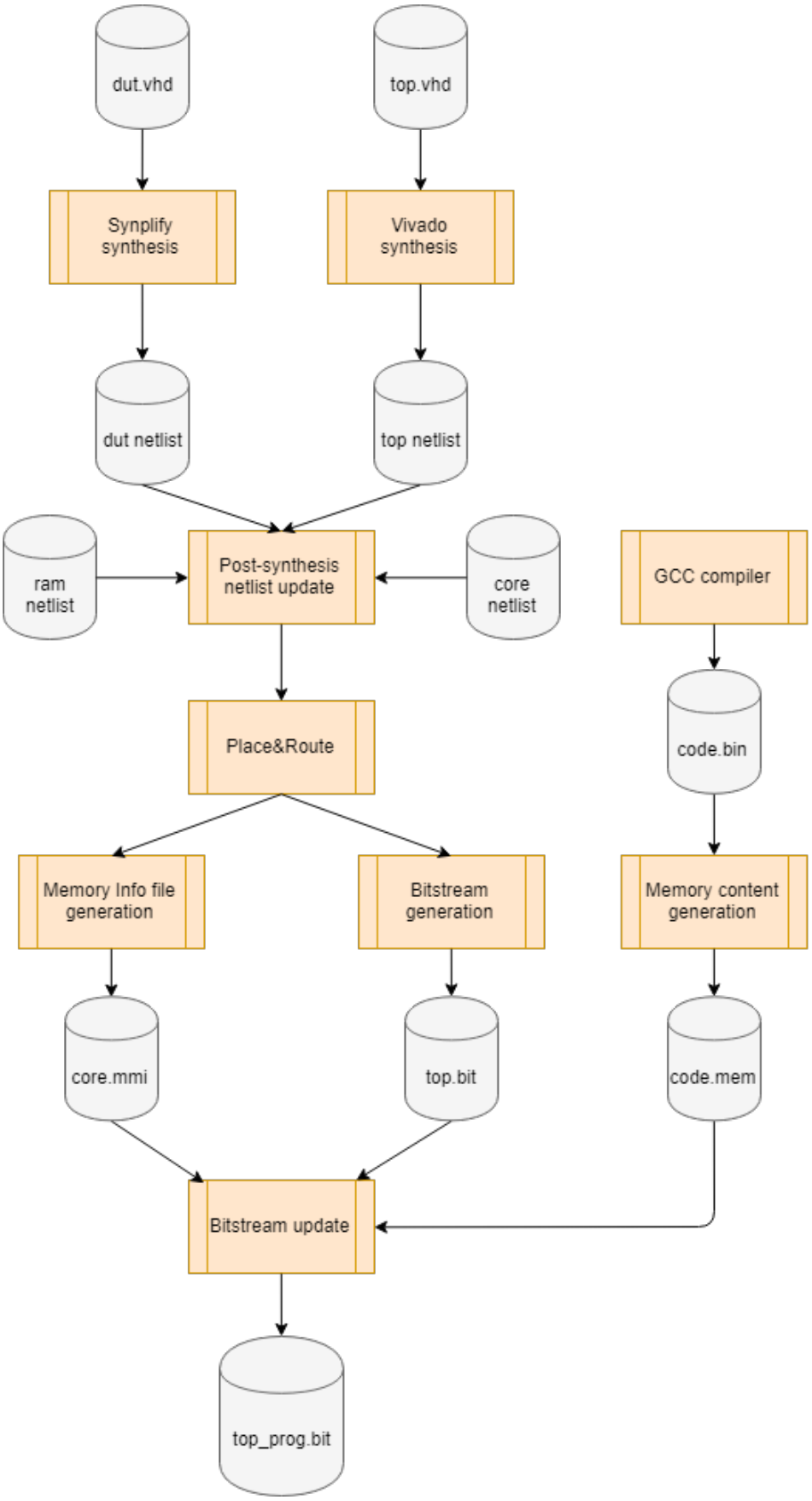


Figure 4.4. Test circuitry implementation flow diagram

Chapter 5

Conclusions and future work

This work started with a survey on the radiation effects in electronic devices and on the available architectures for open-cores. After the core selection process ended the design was adapted to fit the targeted FPGA using Synplify for synthesis process and Vivado for place&route and bitstream generation. Last steps were dedicated to:

- Design protection against radiations using Synplify and custom solutions to overcome some tool limits;
- Test the design and characterize it;

The object of introducing an automized method to protect a processor against radiations is reached, the characterization of different core types shows that design protection increases the MTTF with respect to the basic version. In order to improve both the radiation hardness for this core and to increase the flexibility of this methodology, the future work will be organized in:

- Research of other techniques that could be handled by the implementation flow and automatically applied to the core structure;
- Application of the same implementation flow to another core, increasing the number of designs targeted the flexibility will be improved;
- Application of the same implementation flow to other FPGA families, obviously using the proper tool used for place&route and bitstream generation;

A first improvement could be applied to the memory system. For applications that need a longer MTTF, a way to avoid faults accumulation in IRAM is fundamental. BRAM scrubbing is good to correct SEUs that happened in memory content. This method is not necessary for the targeted environment, this is the reason that brought to not implement this mechanism. Another improvement could try to increase core safety, this means to stop core jobs when some not correctable SEU affects his behavior. Even if some more steps are necessary to have a fully reliable and safe processor, the results obtained by this work are enough to slightly improve the behavior of murax core under radiations effects.

Appendix A

This appendix is focused on the hardware resources targeted for cores implementation and tests.

A.1 Kintex Ultrascale FPGA Architecture

Xilinx Ultrascale architecture is targeting different application fields and focuses on lowering power consumption, using advanced technologies, together with system high performances. There are different families of Ultrascale devices. Kintex Ultrascale comprises a set of FPGAs designed to reach the optimal ratio between price and performance. The following table resumes some general characteristics of this familiy in terms of resources available and related operating speed:

Resources	Value
Logic cells (K)	318-1451
Block Memory (Mb)	12.7-75.9
DSP (Slices)	768-5520
DSP Performances (GMAC/s)	8180
Transceivers	12-64
Max. Transceivers Speed (Gb/s)	16.3
Max. Serial Bandwidth (full duplex) (Gb/s)	2086
Memory Interface Performance (Mb/s)	2400
I/O Pins	312-832

Table A.1. Resources related to Ultrascale family [12]

During the thesis work two particular FPGAs, belonging to this family, were used to test the design:

- XCKU040: contained in the development kit AES-KU040-DB-G designed by AVNET;
- XCKU060: used in the Readout Unit designed for ITS upgrade;

The following table states the differences in resources between the two devices:

Resources	KU040	KU060
Logic cells	530250	725550
CLB Flip-Flops	484800	663360
CLB LUTs	242400	331680
Maximum Distributed RAM (Mb)	7.0	9.1
Block RAM Blocks	600	1080
Block Memory (Mb)	21.1	38.0
CMTs (1 MMCM, 2 PLLs)	10	12
I/O DLLs	40	48
Maximum HP I/Os	416	520
Maximum HR I/Os	104	104
DSP (Slices)	1920	2760
System Monitor	1	1
PCIe Gen3 x8	3	3
GTH 16.3Gb/s Transceivers	20	32

Table A.2. KU040 and KU060 resources [12]

The FPGA used in the devkit has a lower amount of resources with respect to the other one but is considerable as a good device to test the design because the technology used is the same.

A.1.1 Ultrascale resources

To fully understand the functionalities of Ultrascale devices an overview on the different resources characteristics is necessary, the remaining part of this section will focus only on the ones used during development process.

Configurable Logic Block (CLB) This is the main type of resources used to implement general-purpose combinational and sequential circuits. Each CLB is composed by one slice, each slice contains:

- 8 programmable LUTs;
- 16 Flip Flops;

These components are grouped in an array of sub-circuits each one organized as shown in figure 1.1. Depending on the way the device is programmed, CLBs can behave in different ways:

- 6 input LUTs;
- Dual 5 input LUTs;
- Distributed Memory and Shift Register Logic. Slices are divided in two types:
  - SLICEL;

  - SLICEM: can be organized as memory elements and shift registers. Combining the resources of a SLICEM together, tha maximum quantity of memory obtained is 512 bits while the shift register has a width of 256 bits;

- High-speed carry logic for arithmetic functions, used only for small ones because for complex computations DSPs are more optimized;
- Wide multiplexers for efficient utilization. Each 6-input LUT is able to implement a 4 to 1 multiplexer, all the LUTs in a slice can be combined in order to form a 32 to 1 mutliplexer;
- Dedicated storage elements that can be configured as flip-flops or latches with control signals;

Block RAM This primitive is very useful when a huge amount of memory is needed in the design, the usage of these resources allows to reduce the quantity of CLBs dedicated to store data. These are the main characteristics related to this component:

- Two different access ports, they are configured in order to handle concurrent read and write requests:
  - WRITE_FIRST: data written is directly forwarded to the output lines;
  - READ_FIRST: data read is the stored one;
  - NO_CHANGE: data remains equal;
- Able to store up to 36kb;
- Can be organized in the following ways:
  - 32kw of 1 bit;
  - 16kw of 2 bits;
  - 8kw of 4 bits;
  - 4kw of 9 bits;
  - 2kw of 18 bits;
  - 1kw of 36 bits;
  - 512 words of 72 bits;
- Can be divided in two different sub-blocks, each one can store up to 18 kb;
- Two different ways to behave:
  - Simple Dual Port (SDP): both ports are used together for read and write processes in parallel, maximum data width is doubled to 72 bits;
  - True Dual Port (TDP): each port is independent and can be used for read and write processes. Conflicts in memory accesses to the same address are the main problem. They are handled in the following way:

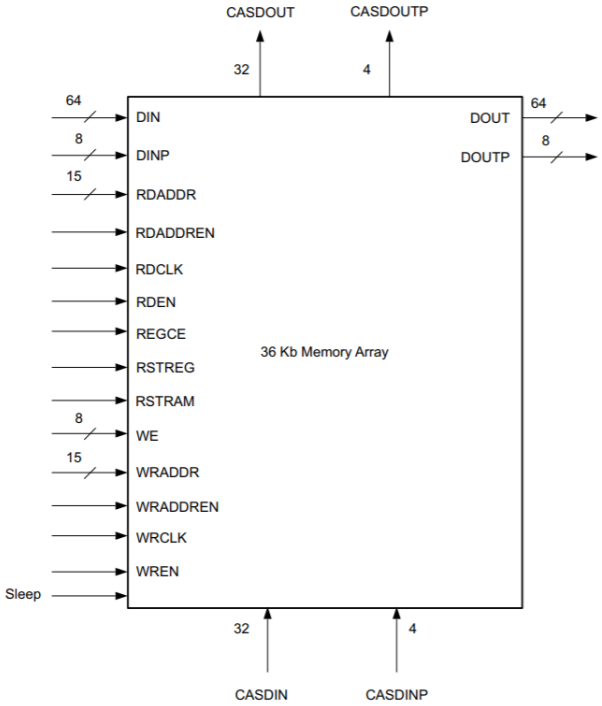


Figure A.1. Block representation of BRAM used in SDP mode [13]

  * Two reads complete successfully;
  * Two writes produce not deterministic data in the memory cell;
  * One write concurrent to one read process produces a deterministic result only if write port is in READ_FIRST mode;

- Read and write processes are performed in one clock cycle;
- Block RAMs can be cascaded to obtain larger memory blocks, using built-in output registers the read process can be pipelined in order to increase performances;
- Synchronous set/reset for memory content and the output register;
- Built-in ECC for 64-bits data. If BRAMs are using this feature, they must be configured in SDP mode and byte wide write process is not allowed;
- Block RAM primitives for Ultrascale technologies are RAMB36E2 and RAMB18E2;

PLL These components are used as:

- Frequency synthesizer;
- Phase shifter;
- Duty cycle programming;

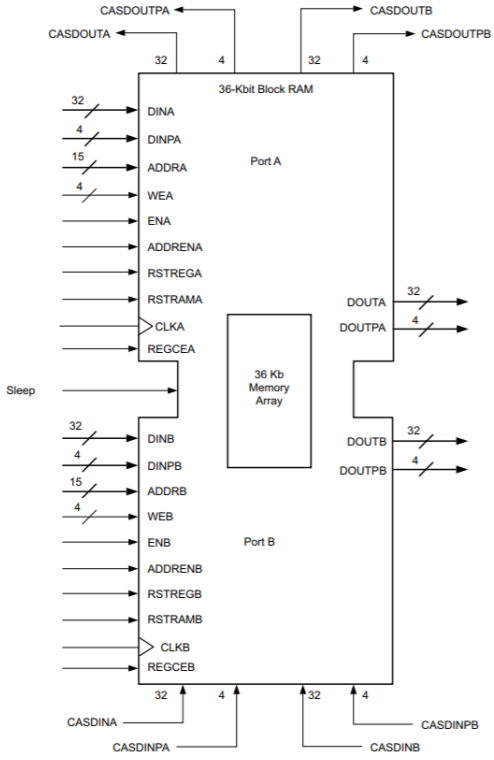


Figure A.2. Block representation of BRAM used in TDP mode [13]

- Clock Deskew;

For Ultrascale technologies PLLs have two different primitives, PLLE3_BASE and PLLE3_ADV. The latter implements even the Dynamic Reconfiguration Protocol to change PLL settings runtime. A complete description of all the available configuration attributes is present in reference [14].

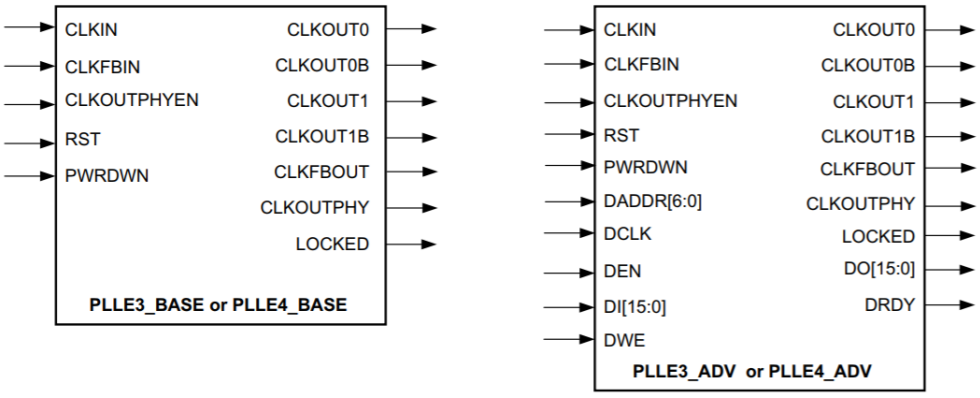


Figure A.3. PLLs block diagrams [14]

The primitive used to obtain the targeted clock frequency is PLLE3_BASE. Starting from

the input CLKIN, the outputs CLKOUT0 and CLKOUT1 are obtained through the following equation:

$$f_{CLKOUT0} = f_{CLKIN} \cdot \frac{CLKFBOUT\_MULT}{CLKOUT0\_DIVIDE \cdot DIVCLK\_DIVIDE} \tag{A.1}$$

$$f_{CLKOUT1} = f_{CLKIN} \cdot \frac{CLKFBOUT\_MULT}{CLKOUT1\_DIVIDE \cdot DIVCLK\_DIVIDE} \tag{A.2}$$

The parameters used in these equations are defined as primitive attributes and has the following characteristics:

- CLKFBOUT_MULT: all output clocks are multiplied by this integer value. For ultrascale technology and considering the targeted primitive the range of possible values is from 1 to 19;
- DIVCLK_DIVIDE: all output clocks are divided by this integer value. For ultrascale technology and considering the targeted primitive the range of possible values is from 1 to 15;
- CLKOUT(0/1)_DIVIDE: only the related clock output is divided by this integer value. For ultrascale technology and considering the targeted primitive the range of possible values is from 1 to 128;

IBUFDS Both in the development kit and in the Readout Unit designed for ITS upgrade the clock is provided through two differential lines, in order to obtain a single-ended signal IBUFDS primitive is used. The image below shows a block diagram representation of this component.

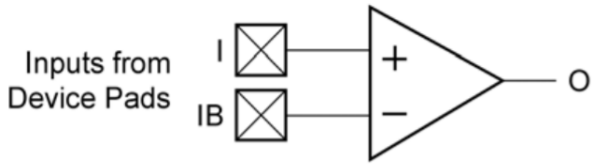


Figure A.4. Block representation of an IBUFDS [10]

The output clock is equal to I if the inputs are different and is remaining the same if they are equal.

IBUF Driver needed for all input signals to the FPGA, receives in input a single-ended signal and provides it to the design programmed.
Some attributes can be defined for this primitive. The only one used to place cores and testing designs on the FPGA is IOSTANDARD, it defines the standard for logic levels.

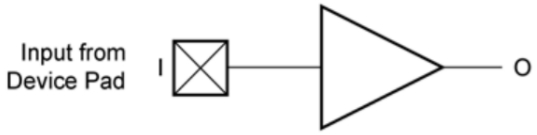


Figure A.5. Block representation of an IBUF [10]

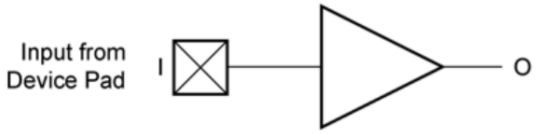


Figure A.6. Block representation of an IBUF [10]

OBUF Driver needed for all outputs of the FPGA, receives in inputs a single-ended signal internal to the design and provides it to the external environment.
Some attributes can be defined for this primitive. The only one used to place cores and testing designs on the FPGA is IOSTANDARD, it defines the standard for logic levels.

A.1.2 Development Kit

The development kit designed by AVNET contains a Xilinx XCKU040-1FBVA676 FPGA and provides different high-speed communication interfaces to the FPGA together with some specific features needed to use all the functions provided by the device. Because many of the available board characteristics were not used, only some resources will be described. For a more detailed documentation there is the manual provided by AVNET [15].

System clock The system clock provided to the FPGA has an associated frequency of 250 MHz. Silicon Labs Si510 oscillator is used to generate the periodic signal, some conditioning circuitry is used to provide the Ultrascale with a cleaner squarewave.

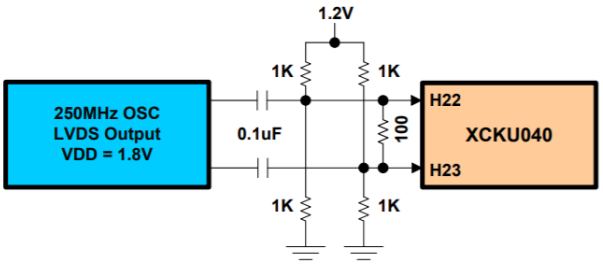


Figure A.7. Conditioning circuitry schematic [15]

H22 and H23 are the two pins used for positive and negative signals of the clock differential pair.

JTAG Programmer AVNET provides basically two methods to program the FPGA, the first uses the standard header provided by Xilinx while the other uses USB connection through Digilent JTAG-SMT2 module. To check if the configuration process ends there is a blue led called DONE LED (FPGA pin D16). In order to take out from the JTAG chain the FMC HPC module a jumper is used.

Pushbuttons There are 4 user available pushbuttons

Pushbutton	Pin
3	K20
2	K21
1	L18
0	K18

Table A.3. LEDs pins mapping [15]

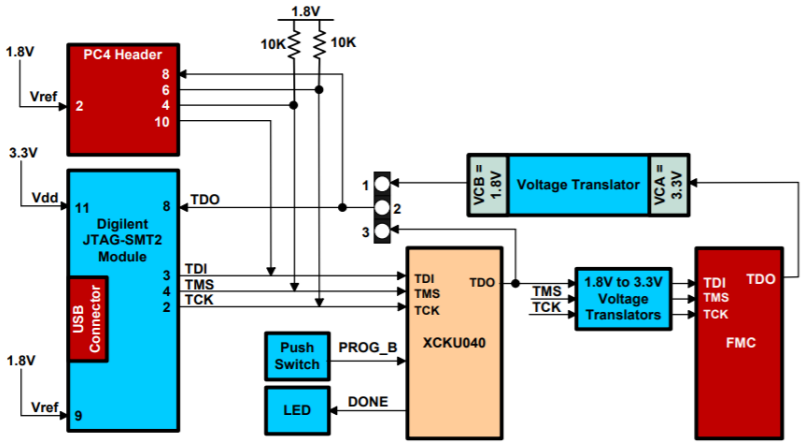


Figure A.8. JTAG Chain schematic [15]

LED There are 8 user available LEDs connected to the FPGA, they are on when the input lines are driven high. This table shows the pins connected to LEDs:

LED	Pin
7	H17
6	H18
5	E16
4	E17
3	E18
2	H16
1	G16
0	D16

Table A.4. LEDs pins mapping [15]

A.2 ITS

The Inner Tracking System is used to collect data for the ALICE experiment. It is composed by a sensing structure divided in Inner, Middle and Outer Barrels.

Inner Barrel The three layers of the Inner Barrel are composed by a different number of identical staves each one based on a structure of nine chips. The communication between the Readout Electronics and the sensors is performed by dedicated bus lines, control and data, one for each chip and all going to the Readout Unit. There are two operating modes for each sensor:

- Acquisition mode: the chips are taking data and sending them through the dedicated line. Control line will carry only the trigger and some slow control commands;
- Setup and Contol mode: the chips are not taking data, control lines are carrying informations on their status and parameters for setup.

The maximum communication speed for data link is 1.2Gbps.

Middle and Outer Barrel The Middle and Outer Barrel has a structure similar to the Inner one, they are both composed by two layers, each one composed by some staves structured in an array of modules. Each module is divided in two rows of 7 elements. The interface between sensors and Readout Unit is equal to the ones implemented for Inner Barrel chips and the maximum speed for data link is 400Mbps.

A.2.1 Readout Unit

It is the board that collects, elaborates and sends to the CRU all the data coming from staves. The main functions provided by this system are:

- Monitor power supplies, to deal with eventual latch-ups;
- Handles the trigger, clock and control signals needed to communicate with sensors/-modules;
- Monitor sensors status;
- Formats data before sending them to CRU;

The components used during the test with real particles flux are:

- SRAM FPGA: as stated before, the FPGA used is from KU060 family;
- GBT-SCA: is the Giga-Bit Transceiver Slow Control Adapter, it is composed by some controllers implementing different communication protocols;

To remotely control all the devices on the board and to exchange data, fiber optic links are used together with the Giga-Bit transceiver system reaching a very high bandwidth useful to transfer the huge quantity of data coming from LHC experiments.

A.2.2 GBT-SCA

Before detailing the Slow Control Adapter structure, a basic description of the GigaBit Transceiver system is needed. It was developed to communicate through the optical link the three following informations required by High Energy Physics experiments:

- Readout Data;
- Trigger and Clock informations;
- Detector control and monitoring informations;

In order to control the front-end board devices, that means the components mounted on the boards close to radiation fluxes, the GBT-SCA was designed. It contains some controllers useful to communicate using some of the most widespread protocols. The particularity of this chip is that it is radiation hardened by design implementing dedicated techniques useful for protection. It is using a commercial 130nm CMOS technology and designed to work in parallel with the GBT transceivers. Starting from the electrical signals got from transceivers, it receives some commands and redirects them to the targeted devices. The available interfaces are:

- 1 SPI master;
- 16 I2C master;
- 1 JTAG master;
- 32 GPIO;
- 31 analog inputs multiplexed to an internal 12-bits ADC;
- 4 8-bits DACs;

Image A.9 shows a block diagram of the internal SCA structure. In order to drive each single interface some commands are addressed using the fiber optic bus, it is based on two different protocol layers:

- E-link protocol: low level protocol implemented by the GBT system;
- SCA commands protocol: communication controllers are externally seen as independent destinations of messages and composed by a set of registers that can be written or read;

After that one complete message is sent to the SCA chip, it returns an answer that is composed by different informations depending on the type of command communicated. Each command frame is divided in the following fields:

- ID: message identification number, returned equal by the answer packet sent by the SCA;

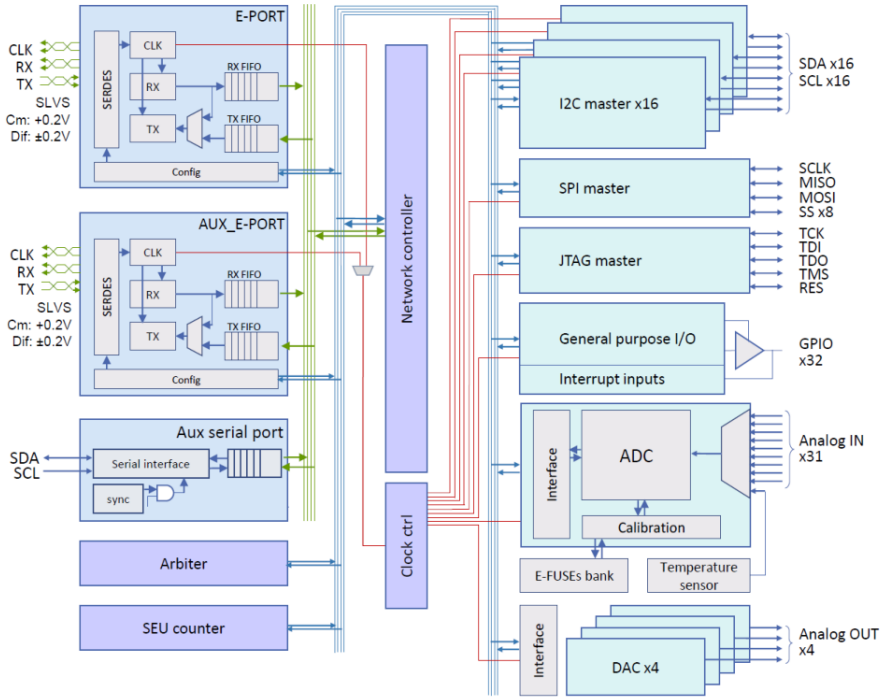


Figure A.9. GBT-SCA block diagram [16].

- Channel: decides which set of interface registers is addressed. These are the available ones:
  - CTRL;
  - SPI;
  - GPIO;
  - I2C0-16;
  - JTAG;
  - ADC;
  - DAC;
- Length: specifies the number of bits associated to the data field;
- Command/Error: if the packet goes towards the SCA it contains the info on the specific register addressed and on the operation required, else this field carries info on errors that happened during controller operations;
- Data: depending on the operation performed it could contain read/written values;

The interface used during the test with particles beam is the SCA SPI master.

A.2.3 SPI interface

The GBT-SCA includes a full-duplex synchronous SPI master interface with the following characteristics:

- 8 slave select lines;
- Maximum single transaction length of 128 bits;
- Transfer rate range from 156kHz to 20MHz;
- Supports all different settings for bus operating modes;

These are the steps used to initialize and perform a communication over the bus:

- Control register is set with the following characteristics:
  - Maximum 128 bits length;
  - Serial clock is low in idle state;
  - Both MISO and MOSI lines works on the rising edge;
  - Bits are transmitted from MSB to LSB;
  - Slave Select line is automatically controlled each time a transaction is performed;
- Frequency divider register is set to reach 1MHz;
- Data register is set;
- SPI transaction is triggered;
- Readback message is obtained with information on the previous communication process;

A detailed commands description is present in [16].

Appendix B

This appendix is focused on the theoretical concepts related to core selection process and to the finally targeted architecture.

B.1 Hardware Licenses

There are many types of open-source licenses and the main difference is related to the possibility of redistribution of a project derived from a licensed one. Before listing some of the most important ones, these concepts must be defined:

- Copyright: is the property that protects an author's work from being copied, distributed or used;
- Permission: right to use a design/product covered by copyright;
- Disclaimer:

The following criteria are used to characterize the licenses:

- Creator attribution of derived works;
- Derived works must remain open source;
- Derived works can have a different license type;
- Derived work can be sold for profit;
- Patent restrictions;

These are some of the most common open licenses:

- MIT license;
- Simplified BSD license;
- Modified BSD license;
- Creative Commons Attribution 3.0 license;
- Creative Commons Attribution Sharealike 3.0 license;

- TAPR Open Hardware License;
- GPL/LGPL;

This table synthesizes the informations on considered licenses types:

License	MIT	Simplified BSD	Modified BSD
Creator Attribution	Optional	Optional	Not permitted
Open Source	No	No	No
Derived works must remain open source	No	No	No
Derived works can have a different license type	Yes	Yes	Yes
Derived works can be sold for profit	Yes	Yes	Yes
Patent restrictions	No	No	No

License	CCA 3.0	CCA SA 3.0	TAPR OHL	GPL/LGPL
Creator Attribution	Required	Required	Optional	Not permitted
Open Source	No	Yes	Yes	Yes
Derived works must remain open source	No	Yes	Yes	Yes
Derived works can have a different license type	Yes	Yes	No	No
Derived works can be sold for profit	Yes	Yes	Yes	Yes
Patent restrictions	No	No	No	Yes

Table B.1. Summary of the licenses considered

B.2 RISC-V

The RISC-V Instruction Set Architecture is developed by the CS division internal to the EECS department related to Berkeley and aims to provide a supported ISA for accademical and commercial purposes. It is a RISC-based solution with a strong modularity that allows to extend the offered functionalities. Despite the targeted applications, this ISA is now a standard for industrial products based on open-source platforms. The following RISC-V description is based on the specifications detailed in the reference manual [17]

As already stated, it is a modular ISA that allows to extend the basic functionalities implemented. The common part among all the possible versions is integer based and very similar to the one of the first RISC architectures without the branch delay slot[1] and with the additional support to variable instructions length. Each different version is named in the following way:

1. RV followed by the architecture parallelism;
2. I that is the basic instruction set integer part;
3. All the other additional functions indexed by a letter;

These are the standard extensions detailed in the RISC-V specifications:

- M: includes the integer multiplication and division;
- A: adds instructions that perform atomic operations on memories;
- F: standard single-precision floating point instructions, registers, load instruction and store instrucion;
- D: similar to F with the difference that expands the already present floating point registers;
- G: general purpose extension that includes all the previous ones;

The architecture targeted by this project is RV32IM.

[1]The Branch Delay Slot defines the positions related to instructions successive to a jump that are, independently from the result, always executed by the processor.

Appendix C

C.1 AES algorithm

Advanced Encryption Standard is an algorithm used to protect data in systems that need high security. The original name is Rijndael and was developed by Joaen Daemen and Vincent Rijmen with the object of a standard way to preserve frames from external attacks better than Data Encryption Standard (DES). These are the main characteristics:

- Variable widths available for key and data: 128, 192 and 256 bits;
- All data are processed in parallel;
- The number of rounds is different depending on the key size, for 128 bits the algorithm core is repeated ten times;
- The word that must be encrypted is organized in a matrix were each byte is addressed by a row and column number. Generally it is organized by columns;
- Each round, except the tenth, is composed by the following steps:
  - Encryption:
    - Substitute bytes: this process is simply based on a 16x16 lookup table, called s-box, were each position is composed by a byte, this means that the whole matrix is able to contain all the 256 combination with an 8 bits value. The substitution process is done dividing the byte in nibbles [1] were the lowest nibble is addressing rows and the other one columns of the s-box. Usually the lookup table is formed with the object of having a low correlation between inputs and outputs;
    - Shift rows: considering an input data of 128 bits composed by 16 bytes, the obtained matrix dimension is 4x4. In this case the algorithm is:
      - The second row is shifted left one time;
      - The third row is shifted left two times;
      - The fourth row is shifted left three times;

[1] A nibble is composed by four bits

    * Mix Columns: this process is based on matrices multiplication, the result is just the product between data matrix and trasformation one;
    * Add Round Key: the encrypted data is transformed in an array of 44 words each one composed by 128 bits. This vector is evaluated in the following way:
        · The first word is a key replica;
        · Each successive word is the result of the *xor* operation between the four back one and a *temp* value formed in two different ways:
            1. If the word index module 4 does not return 0, the *temp* value is equal to the previous word;
            2. Else, before the last *xor* operation, the following functions are applied:
                (a) One byte left shift;
                (b) S-box substitution;
                (c) The *xor* between (a) and (b) produces *temp*;

– Decryption:

    * Inverse Shift Rows: is doing opposite operations with respect to Shift Rows;
    * Inverse Substitutes bytes: is composed by a lookup table designed to be the inverse matrix of the s-box;
    * Inverse Add Round Key: inverse operations with respect to Add Round Key step;
    * Inverse Mix Columns: multiplication of encrypted data with the inversed matrix used in step Mix Columns;

Appendix D

Design Automation

The part related to design automation is important for the flow that finally produces the fault tolerant core. There are two specific processes that required specific scripting to reach the desired results:

- Memory substitution;
- Memory mapping;

D.1 Memory substitution

This function takes in input the following netlists:

- Core;
- Protected memory system;

The output is composed by the post-synthesis netlist including the protected memory system. To automize this operation some considerations are needed on the possible processor starting architecture, both the radiation hardened and the not protected designs are valid starting points for this procedure. The main difference between the two structures is related to the output mismatch lines [1]:

- Single core: there is just one RAM instance, this means that the mismatch lines are directly forwarded to processor outputs. As analyzed before the decoder is able to provide informations on the presence of a single bit error and of a double bit error, to handle them two signals are added to core interface;
- TMR core: there are three RAM instances but, to mantain the same interface described for the not triplicated version, the single bit error and the double bit error lines are high if an error is detected in one of the internal mismatch signals;

[1] These lines were not used in the testing architecture because it was shared by both the Murax core and the MicroBlaze from Xilinx

The substitution script is written using TCL language, executed by Vivado and performed through the following steps:

- The different core instances paths are stored in a list:
  - Single core: path to the top wrapper;
  - TMR core: path to the three instances, Synplify is instantiating each replica using the name TMRn where n is the related number;
- All the BRAMs used are listed using *all_rams* command;
- Processors RAM are selectd using regular expressions, specifically each BRAM path is compared with each core instance and *ram_symbol* [2] string.
- If one path matches the conditions:
  - The following nets names are stored:
    - Clock;
    - Data In;
    - Data Out;
    - Address;
    - Write Enables;
    - Read Enable;
  - All the previous signals are disconnected from the BRAM that is removed from the design;
  - A black-box cell is instantiated and updated with the protected memory system netlist;
  - All stored signals are connected to the new RAM interface;
- The mismatch lines are directly connected to the core interface or provided in input to a tree of or gates;

The flow diagram in figure D.1 is representing the substitution process.

[2]It is a string common to single processor, TMR processor and the name used for protected memory top entity.

D.2 Memory mapping

This process is useful to isolate and store all the informations needed by *Updatemem* tool in memory content update procedure. The first part is common to the memory substitution process, all the BRAMs internal to processor RAMs are selected and the related paths are stored in a list. There are four different memory systems depending on the core version:

- Not protected: only one data BRAM, bytes at lower addresses are indexing least significant positions;
- ECC: two BRAMs, one for data and the other for redundant bits, the former is exactly as the not protected one while the latter is following the same mapping order with a lower input data widths;
- TMR: three BRAMs, all of them for data. The particularity with respect to the first case is that Synplify is swapping bytes positions for single 32-bits frames inside the memory;
- TMR-ECC: exactly as for the single ECC version, the memory system is similar to the counterpart without information redundancy. Because bytes endianness is reversed, the ECC BRAM content is evaluated with changed data;

For versions implementing hardware redundancy the content bytes swapping is performed during code compiling. As stated in chapter 3, the result of this script is a *.mmi* file carrying all the needed informations to update targeted BRAMs. This is the algorithm used:

- BRAMs are stored in a list through *all_rams* command;
- Processors related blocks are selected in the following way:
  - If ECC is not applied, the generic *ram_symbol* expression is used for regular expression research;
  - If ECC is present, a distinction is done between data and redundant memory through *data_mem* and *ecc_mem* patterns.
- The list of BRAMs obtained is parsed and one *.mmi* file is written for each block;

Similarly to memory substitution, this procedure is executed through a TCL script by Vivado and it is listed in next page.

Listing D.1. Memory mapping script

```
# Obtain RAM lists

set gram       [split [all_rams] " "]

# Selects only RAMB36 type and saving the sites in an array

set bram       [list ]
set bramsites [list ]

set c     0
set nram 0

foreach ram $gram {
  set temp [get_property PRIMITIVE_TYPE [get_cells $ram]]
  if {$temp == "BLOCKRAM.BRAM.RAMB36E2" && [regexp "${soc_inst}ram_symbol" $ram] } {
    incr nram
  }
}

if {!$nram} {
  puts "ERROR: no BRAM has been found core ${soc_inst} instantiation"
}

while {$c!=$nram} {
  foreach ram $gram {

    set temp [get_property PRIMITIVE_TYPE [get_cells $ram]]
    if {$temp == "BLOCKRAM.BRAM.RAMB36E2" && [regexp "${soc_inst}ram_symbol" $ram] } {
      set length [string length $ram]
      set i      [string index $ram [expr $length - 3]]
      if {$i==$c} {
        set site [lindex [split [get_sites -of_objects [get_cells $ram]] "_"] 1]
        puts "INFO: Mapping $ram with site $site"
        set bram      [lappend bram      $ram]
        set bramsites [lappend bramsites $site]
        incr c
      }
    }
    # puts $c
  }
}

set MSB 31
puts "INFO: MSB of the data is $MSB"

set address_space     65535
set ram_address_space [expr $address_space / [ expr [expr $MSB + 1] * $nram]]

set fp [open "../netlist/murax_tmr${tmr}_ecc${ecc}_core${core}.mmi" "w"]

puts $fp "<?xml version=\"1.0\" encoding=\"UTF-8\"?>"
puts $fp "<!-- The time is: [clock format [clock seconds] -format %H:%M:%S] -->"
puts $fp "<MemInfo Version=\"1\" Minor=\"1\">"
puts $fp "  <Processor Endianness=\"Little\" InstPath=\"soc_inst\">"
puts $fp "    <AddressSpace Name=\"soc_inst_mem\" Begin=\"0\" End=\"[expr $address_space -1]\">"
puts $fp "      <BusBlock>"

set n 0

foreach bram_1bit $bram {
  set length [string length $bram_1bit]
  puts $fp "        <BitLane MemType=\"RAMB36\" Placement=\"[lindex $bramsites $n]\">"
  puts $fp "          <DataWidth MSB=\"$MSB\" LSB=\"0\"/>"
  puts $fp "          <AddressRange Begin=\"[expr $ram_address_space * $n]\"
  End=\"[expr [expr $ram_address_space * [expr $n + 1]] -1]\"/>"
  puts $fp "          <Parity ON=\"true\" NumBits=\"0\"/>"
  puts $fp "        </BitLane>"
```

```
  incr n
}
puts $fp "      </BusBlock>"
puts $fp "    </AddressSpace>"
puts $fp "  </Processor>"
puts $fp "  <Config>"
puts $fp "    <Option Name=\"Part\" Val=\"xcku040-fbva676-1-c\"/>"
puts $fp "  </Config>"
puts $fp "</MemInfo>"
close $fp
```

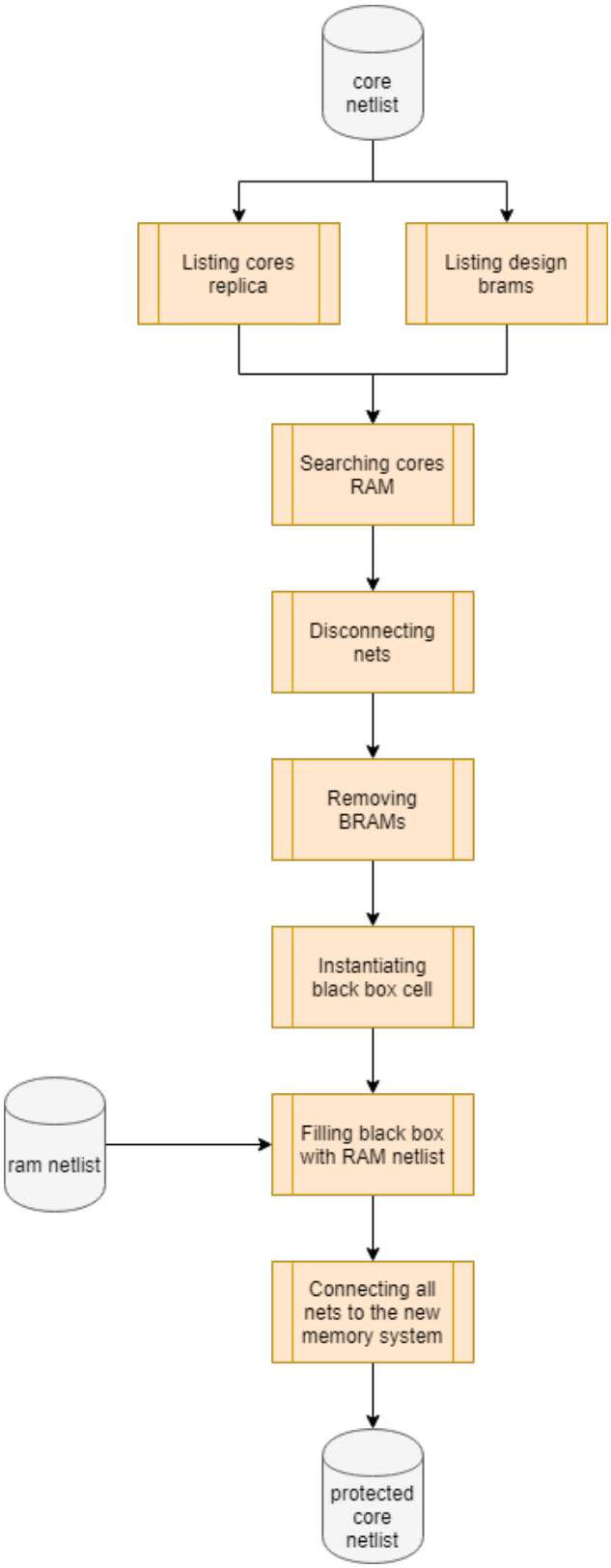


Figure D.1. Protected memory insertion flow diagram

Appendix E

VHDL files

E.1 Core files

Listing E.1. Top entity

```
library synplify;
use synplify.attributes.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
-- use work.syn_attr.all;
-- use work.hrel_attr.all;
use work.arith.all;
-- use work.all;


library unisim;
use unisim.vcomponents.all;

entity top is
  generic(
    RAMDIM: integer
  );
  port(
    io_asyncReset:         in    std_logic;
    io_mainClk:            in    std_logic;
    io_gpioA_read:         in    std_logic_vector(31 downto 0);
    io_gpioA_write:        out   std_logic_vector(31 downto 0);
    io_gpioA_writeEnable: out   std_logic_vector(31 downto 0);
    obram_err_sbit:        out   std_logic;
    obram_err_dbit:        out   std_logic;
    io_uart_txd:           out   std_logic;
    io_uart_rxd:           in    std_logic
  );
end entity;

architecture soc_noecc of top is

  component murax_wrapper is
    port(
      io_asyncReset:         in  std_logic;
      io_mainClk:            in  std_logic;
      io_jtag_tms:           in  std_logic;
      io_jtag_tdi:           in  std_logic;
      io_jtag_tdo:           out std_logic;
      io_jtag_tck:           in  std_logic;
      io_gpioA_read:         in  std_logic_vector(31 downto 0);
      io_gpioA_write:        out std_logic_vector(31 downto 0);
```

```
      io_gpioA_writeEnable: out std_logic_vector(31 downto 0);
      io_uart_txd:          out std_logic;
      io_uart_rxd:          in  std_logic
    );
  end component;

  component jtag_controller is
    port(
      io_asyncReset: in  std_logic;
      io_jtag_tms:   out std_logic:='1';
      io_jtag_tdi:   out std_logic;
      io_jtag_tdo:   in  std_logic;
      io_jtag_tck:   out std_logic;
      io_mainClk:    in  std_logic;
      jtag_mainClk:  in  std_logic
    );
  end component;

  signal io_jtag_tms:            std_logic;
  signal io_jtag_tdi:            std_logic;
  signal io_jtag_tdo:            std_logic;
  signal io_jtag_tck:            std_logic;
  signal jtag_mainClk:           std_logic;

begin

  jtag_inst : jtag_controller
  port map (
    io_asyncReset => io_asyncReset,
    io_mainClk    => io_mainClk,
    io_jtag_tms   => io_jtag_tms,
    io_jtag_tdi   => io_jtag_tdi,
    io_jtag_tdo   => io_jtag_tdo,
    io_jtag_tck   => io_jtag_tck,
    jtag_mainClk  => jtag_mainClk
  );

  soc_inst : murax_wrapper
  port map (
    io_asyncReset        => io_asyncReset,
    io_mainClk           => io_mainClk,
    io_jtag_tms          => io_jtag_tms,
    io_jtag_tdi          => io_jtag_tdi,
    io_jtag_tdo          => io_jtag_tdo,
    io_jtag_tck          => io_jtag_tck,
    io_gpioA_read        => io_gpioA_read,
    io_gpioA_write       => io_gpioA_write,
    io_gpioA_writeEnable => io_gpioA_writeEnable,
    io_uart_txd          => io_uart_txd,
    io_uart_rxd          => io_uart_rxd
  );

end soc_noecc;

architecture soc_ecc of top is

  component murax_wrapper is
    port(
      io_asyncReset:        in  std_logic;
      io_mainClk:           in  std_logic;
      io_jtag_tms:          in  std_logic;
      io_jtag_tdi:          in  std_logic;
      io_jtag_tdo:          out std_logic;
      io_jtag_tck:          in  std_logic;
      io_gpioA_read:        in  std_logic_vector(31 downto 0);
      io_gpioA_write:       out std_logic_vector(31 downto 0);
      io_gpioA_writeEnable: out std_logic_vector(31 downto 0);
      io_uart_txd:          out std_logic;
```

```
      io_uart_rxd:          in  std_logic
    );
  end component;

  component jtag_controller is
    port(
      io_asyncReset: in  std_logic;
      io_jtag_tms:   out std_logic:='1';
      io_jtag_tdi:   out std_logic;
      io_jtag_tdo:   in  std_logic;
      io_jtag_tck:   out std_logic;
      io_mainClk:    in  std_logic;
      jtag_mainClk:  in  std_logic
    );
  end component;

  component emip is
    generic(
      RAMDIM: integer
    );
    port(
      clk:                  in std_logic;
      rsti:                 in std_logic;
      obram_err_sbit:       out std_logic;
      obram_err_dbit:       out std_logic
    );
  end component;

  signal io_jtag_tms:           std_logic;
  signal io_jtag_tdi:           std_logic;
  signal io_jtag_tdo:           std_logic;
  signal io_jtag_tck:           std_logic;
  signal jtag_mainClk:          std_logic;

  attribute syn_keep of obram_err_sbit: signal is true;
  attribute syn_keep of obram_err_dbit: signal is true;

begin


  jtag_inst : jtag_controller
  port map (
    io_asyncReset => io_asyncReset,
    io_mainClk    => io_mainClk,
    io_jtag_tms   => io_jtag_tms,
    io_jtag_tdi   => io_jtag_tdi,
    io_jtag_tdo   => io_jtag_tdo,
    io_jtag_tck   => io_jtag_tck,
    jtag_mainClk  => jtag_mainClk
  );

  soc_inst : murax_wrapper
  port map (
    io_asyncReset        => io_asyncReset,
    io_mainClk           => io_mainClk,
    io_jtag_tms          => io_jtag_tms,
    io_jtag_tdi          => io_jtag_tdi,
    io_jtag_tdo          => io_jtag_tdo,
    io_jtag_tck          => io_jtag_tck,
    io_gpioA_read        => io_gpioA_read,
    io_gpioA_write       => io_gpioA_write,
    io_gpioA_writeEnable => io_gpioA_writeEnable,
    io_uart_txd          => io_uart_txd,
    io_uart_rxd          => io_uart_rxd
  );

  emip_inst : emip
  generic map(
```

```
    RAMDIM => RAMDIM
  )
  port map (
    clk                   => io_mainClk,
    rsti                  => io_asyncReset,
    obram_err_sbit        => obram_err_sbit,
    obram_err_dbit        => obram_err_dbit
  );

end soc_ecc;

configuration noTMR_ECC of top is
 for soc_ecc
   for soc_inst:murax_wrapper
     use entity work.murax_wrapper(wrapper);
   end for;
   for emip_inst:emip
     use entity work.emip(notmr);
   end for;
 end for;
end configuration noTMR_ECC;

configuration noTMR_noECC of top is
 for soc_noecc
   for soc_inst:murax_wrapper
     use entity work.murax_wrapper(wrapper);
   end for;
 end for;
end configuration noTMR_noECC;

configuration TMR_ECC of top is
 for soc_ecc
   for soc_inst:murax_wrapper
     use entity work.murax_wrapper(wrapper_distributed);
   end for;
   for emip_inst:emip
     use entity work.emip(tmr);
   end for;
 end for;
end configuration TMR_ECC;

configuration TMR_noECC of top is
 for soc_noecc
   for soc_inst:murax_wrapper
     use entity work.murax_wrapper(wrapper_distributed);
   end for;
 end for;
end configuration TMR_noECC;
```

Listing E.2. Error monitoring IP

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.syn_attr.all;
use work.hrel_attr.all;

-- Here must be defined the ports for the error checking.
-- In order to detect the correct working of the device,
-- the following information will be monitored:
-- - ECC codes of the BRAMs

entity emip is
  generic(
    RAMDIM: integer
  );
  port(
    clk:                  in std_logic;
    rsti:                 in std_logic;
    obram_err_sbit:       out std_logic;
    obram_err_dbit:       out std_logic
  );
end entity;

architecture notmr of emip is

  attribute syn_noprune  of notmr: architecture is true;
  attribute syn_preserve of notmr: architecture is true;

  signal obram_err_sbit_sysram: std_logic;
  signal obram_err_dbit_sysram: std_logic;

  attribute syn_keep of obram_err_sbit_sysram: signal is true;
  attribute syn_keep of obram_err_dbit_sysram: signal is true;

  component emip_bram is
    generic(
      N: integer
    );
    port(
      clk:            in std_logic;
      rsti:           in std_logic;
      ibram_err_sbit: in std_logic_vector(N-1 downto 0);
      ibram_err_dbit: in std_logic_vector(N-1 downto 0);
      obram_err_sbit: out std_logic;
      obram_err_dbit: out std_logic
    );
  end component;

begin

  obram_err_sbit <= obram_err_sbit_sysram;
  obram_err_dbit <= obram_err_dbit_sysram;

  SYSRAM: emip_bram
    generic map(
      N => RAMDIM
    )
    port map(
      clk            => clk,
      rsti           => rsti,
      obram_err_sbit => obram_err_sbit_sysram,
      obram_err_dbit => obram_err_dbit_sysram,
      ibram_err_sbit => open,
      ibram_err_dbit => open
    );

end notmr;
```

```
architecture tmr of emip is

  attribute syn_noprune  of tmr: architecture is true;
  attribute syn_preserve of tmr: architecture is true;

  signal obram_err_sbit_sysram0: std_logic;
  signal obram_err_dbit_sysram0: std_logic;

  signal obram_err_sbit_sysram1: std_logic;
  signal obram_err_dbit_sysram1: std_logic;

  signal obram_err_sbit_sysram2: std_logic;
  signal obram_err_dbit_sysram2: std_logic;

  attribute syn_keep of obram_err_sbit_sysram0: signal is true;
  attribute syn_keep of obram_err_dbit_sysram0: signal is true;

  attribute syn_keep of obram_err_sbit_sysram1: signal is true;
  attribute syn_keep of obram_err_dbit_sysram1: signal is true;

  attribute syn_keep of obram_err_sbit_sysram2: signal is true;
  attribute syn_keep of obram_err_dbit_sysram2: signal is true;

  attribute syn_radhardlevel of tmr: architecture is "distributed_tmr";

  component emip_bram is
    generic(
      N: integer
    );
    port(
      clk:            in std_logic;
      rsti:           in std_logic;
      ibram_err_sbit: in std_logic_vector(N-1 downto 0);
      ibram_err_dbit: in std_logic_vector(N-1 downto 0);
      obram_err_sbit: out std_logic;
      obram_err_dbit: out std_logic
    );
  end component;

begin

  obram_err_sbit <= obram_err_sbit_sysram0 or obram_err_sbit_sysram1 or obram_err_sbit_sysram2;
  obram_err_dbit <= obram_err_dbit_sysram0 or obram_err_dbit_sysram1 or obram_err_dbit_sysram2;

  SYSRAM0: emip_bram
    generic map(
      N => RAMDIM
    )
    port map(
      clk            => clk,
      rsti           => rsti,
      obram_err_sbit => obram_err_sbit_sysram0,
      obram_err_dbit => obram_err_dbit_sysram0,
      ibram_err_sbit => open,
      ibram_err_dbit => open
    );

  SYSRAM1: emip_bram
    generic map(
      N => RAMDIM
    )
    port map(
      clk            => clk,
      rsti           => rsti,
      obram_err_sbit => obram_err_sbit_sysram1,
      obram_err_dbit => obram_err_dbit_sysram1,
      ibram_err_sbit => open,
```

```
      ibram_err_dbit => open
    );

  SYSRAM2: emip_bram
    generic map(
      N => RAMDIM
    )
    port map(
      clk             => clk,
      rsti            => rsti,
      obram_err_sbit => obram_err_sbit_sysram2,
      obram_err_dbit => obram_err_dbit_sysram2,
      ibram_err_sbit => open,
      ibram_err_dbit => open
    );

end tmr;
```

Listing E.3. Error monitoring IP BRAM

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.syn_attr.all;

-- This is a generic component for the evaluation of ECC flags for BRAM

entity emip_bram is
  generic(
    N: integer
  );
  port(
    clk:            in std_logic;
    rsti:           in std_logic;
    ibram_err_sbit: in std_logic_vector(N-1 downto 0);
    ibram_err_dbit: in std_logic_vector(N-1 downto 0);
    obram_err_sbit: out std_logic;
    obram_err_dbit: out std_logic
  );
end entity;

architecture rtl of emip_bram is

  attribute syn_noprune  of rtl: architecture is true;
  attribute syn_preserve of rtl: architecture is true;

  signal obram_err_sbit_tmp: std_logic_vector(N-1 downto 0);
  signal obram_err_dbit_tmp: std_logic_vector(N-1 downto 0);
  signal ibram_err_sbit_tmp: std_logic_vector(N-1 downto 0);
  signal ibram_err_dbit_tmp: std_logic_vector(N-1 downto 0);

  -- attribute syn_keep of obram_err_sbit_tmp: signal is true;
  -- attribute syn_keep of obram_err_dbit_tmp: signal is true;

begin

  REG_i: process(clk, rsti, ibram_err_sbit, ibram_err_dbit)
  begin
    if rsti = '1' then
      ibram_err_sbit_tmp <= (others => '0');
      ibram_err_dbit_tmp <= (others => '0');
    elsif clk'event and clk = '1' then
      ibram_err_sbit_tmp <= ibram_err_sbit;
      ibram_err_dbit_tmp <= ibram_err_dbit;
    end if;
  end process;

  obram_err_sbit_tmp(0) <= ibram_err_sbit_tmp(0);

  SBIT: for i in 1 to N-1 generate
    obram_err_sbit_tmp(i) <= obram_err_sbit_tmp(i-1) or ibram_err_sbit_tmp(i);
  end generate;

  obram_err_dbit_tmp(0) <= ibram_err_dbit_tmp(0);

  DBIT: for i in 1 to N-1 generate
    obram_err_dbit_tmp(i) <= obram_err_dbit_tmp(i-1) or ibram_err_dbit_tmp(i);
  end generate;

  REG_o: process(clk, rsti, obram_err_sbit_tmp(N-1), obram_err_dbit_tmp(N-1))
  begin
    if rsti = '1' then
      obram_err_sbit <= '0';
      obram_err_dbit <= '0';
    elsif clk'event and clk = '1' then
      obram_err_sbit <= obram_err_sbit_tmp(N-1);
      obram_err_dbit <= obram_err_dbit_tmp(N-1);
```

```
    end if;
  end process;

end rtl;
```

E.2 Memory files

Listing E.4. Memory top entity

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.arith.all;
use work.hrel_attr.all;
use work.syn_attr.all;
use work.comp_pkg.all;

entity ram is
  generic(
    width:      integer:=1024;
    word_width: integer:=32;
    init:       string;
    init_ecc:   string
  );
  port(
    clk:      in  std_logic;
    rst:      in  std_logic;
    din:      in  std_logic_vector(31 downto 0);
    addr:     in  std_logic_vector(log2(width)-1  downto 0);
    ren:      in  std_logic;
    wen:      in  std_logic_vector(3  downto 0);
    dout:     out std_logic_vector(31 downto 0);
    sbiterr:  out std_logic;
    dbiterr:  out std_logic
  );
end entity;

architecture ecc_secded of ram is

  component ram_ctrl is
    generic(
      width: integer:=1024
    );
    port(
      clk:         in    std_logic;
      rst:         in    std_logic;
      din:         in    std_logic_vector(31            downto 0);
      addr:        in    std_logic_vector(log2(width)-1 downto 0);
      ren:         in    std_logic;
      wen:         in    std_logic_vector(3             downto 0);
      dout_noenc:  in    std_logic_vector(31            downto 0);
      dout_ecc:    in    std_logic_vector(7             downto 0);
      dout:        out   std_logic_vector(31            downto 0);
      din_ecc:     out   std_logic_vector(7             downto 0);
      wen1_ecc:    inout std_logic;
      wen2_ecc:    out   std_logic;
      addr_ecc:    out   std_logic_vector(log2(width)-1 downto 0);
      sbiterr:     out   std_logic;
      dbiterr:     out   std_logic
    );
  end component;

  attribute black_box of ram_ctrl: component is "true";

  signal dout_noenc: std_logic_vector(31            downto 0);
  signal dout_ecc:   std_logic_vector(7             downto 0);
  signal din_ecc:    std_logic_vector(7             downto 0);
  signal wen1_ecc:   std_logic_vector(0             downto 0);
  signal wen2_ecc:   std_logic_vector(0             downto 0);
  signal addr_ecc:   std_logic_vector(log2(width)-1 downto 0);
  signal addr_shecc: std_logic_vector(log2(width)+1 downto 0);
  signal addr_sh:    std_logic_vector(log2(width)+1 downto 0);
```

```
begin

  ram_ctrl_inst: ram_ctrl
  generic map(
    width => 1024
  )
  port map(
    clk         => clk,
    rst         => rst,
    din         => din,
    addr        => addr,
    ren         => ren,
    wen         => wen,
    dout_noenc  => dout_noenc,
    dout_ecc    => dout_ecc,
    dout        => dout,
    din_ecc     => din_ecc,
    wen1_ecc    => wen1_ecc(0),
    wen2_ecc    => wen2_ecc(0),
    addr_ecc    => addr_ecc,
    sbiterr     => sbiterr,
    dbiterr     => dbiterr
  );

  data_mem_inst: bram_sdp
  generic map(
    DATA => 32,
    ADDR => log2(width),
    init => init
  )
  port map(
    a_clk  => clk,
    a_wr   => wen,
    a_en   => ren,
    a_addr => addr,
    a_din  => din,
    a_dout => dout_noenc
  );

  addr_shecc <= "00" & addr_ecc;
  addr_sh    <= "00" & addr;
  ecc_mem_inst: bram_tdp
  generic map(
    DATA => 8,
    ADDR => log2(width)+2,
    init => init_ecc
  )
  port map(
    -- Port A
    a_clk  => clk,
    a_wr   => wen1_ecc,
    a_en   => ren,
    a_addr => addr_sh,
    a_din  => din_ecc,
    a_dout => dout_ecc,
    b_clk  => clk,
    b_wr   => wen2_ecc,
    b_en   => '1',
    b_addr => addr_shecc,
    b_din  => din_ecc,
    b_dout => open
  );

end ecc_secded;
```

Listing E.5. Ram control circuitry

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.arith.all;
use work.hrel_attr.all;
use work.comp_pkg.all;

entity ram_ctrl is
  generic(
    width: integer:=1024
  );
  port(
    clk:         in    std_logic;
    rst:         in    std_logic;
    din:         in    std_logic_vector(31            downto 0);
    addr:        in    std_logic_vector(log2(width)-1 downto 0);
    ren:         in    std_logic;
    wen:         in    std_logic_vector(3             downto 0);
    dout_noenc:  in    std_logic_vector(31            downto 0);
    dout_ecc:    in    std_logic_vector(7             downto 0);
    dout:        out   std_logic_vector(31            downto 0);
    din_ecc:     out   std_logic_vector(7             downto 0);
    wen1_ecc:    inout std_logic;
    wen2_ecc:    out   std_logic;
    addr_ecc:    out   std_logic_vector(log2(width)-1 downto 0);
    sbiterr:     out   std_logic;
    dbiterr:     out   std_logic
  );
end entity;

architecture read_first of ram_ctrl is

  signal addr_ecc_i: std_logic_vector(log2(width)-1 downto 0);
  signal dout_i:     std_logic_vector(31 downto 0);
  signal dout_r:     std_logic_vector(31 downto 0);
  signal din_r:      std_logic_vector(31 downto 0);
  signal din_ecc_i:  std_logic_vector(7  downto 0);
  signal dout_ecc_i: std_logic_vector(6  downto 0);
  signal err:        std_logic_vector(1  downto 0);
  signal wen_r:      std_logic_vector(3  downto 0);
  signal wen_i:      std_logic;
  signal wend_i:     std_logic;
  signal addr_eq_w:  std_logic;
  signal outmux:     std_logic;
  signal addr_xor:   std_logic_vector(log2(width)-1 downto 0);

begin

  addr_registern_inst: registern
  generic map(
    n => log2(width)
  )
  port map(
    clk => clk,
    rst => rst,
    en  => '1',
    d   => addr,
    q   => addr_ecc_i
  );

  addr_xor <= (others => wen1_ecc);
  addr_ecc <= addr_ecc_i xor addr_xor;

  addr_eqcomparator_inst1: eqcomparator
  generic map(
    n => log2(width)
  )
```

```
port map(
  a => addr,
  b => addr_ecc_i,
  e => addr_eq_w
);

wen_i <= (wen(0) or wen(1) or wen(2) or wen(3)) and ren;

wen_ffd_inst: ffd
port map(
  clk => clk,
  rst => rst,
  en  => '1',
  d   => wen_i,
  q   => wend_i
);

wen_registern_inst: registern
generic map(
  n   => 4
)
port map(
  clk => clk,
  rst => rst,
  en  => '1',
  d   => wen,
  q   => wen_r
);

din_registern_inst: registern
generic map(
  n   => 32
)
port map(
  clk => clk,
  rst => rst,
  en  => '1',
  d   => din,
  q   => din_r
);

dout_r_generate_inst: for i in 0 to 3 generate
  dout_r_mux21gen_inst: mux21gen
  generic map(
    n => 8
  )
  port map(
    a => dout_noenc((8*(i+1))-1 downto 8*i),
    b => din_r((8*(i+1))-1 downto 8*i),
    s => wen_r(i),
    c => dout_r((8*(i+1))-1 downto 8*i)
  );
end generate;

wen1_ecc <= wend_i and (addr_eq_w and ren);

wen2_ecc <= wend_i and (not(addr_eq_w) or not(ren));

outmux  <= '0';

din_ecc_encoder_inst: encoder
port map(
  din  => dout_r,
  dout => din_ecc(6 downto 0)
);
din_ecc(7) <= '0';

dout_decoder_inst: decoder
```

```
  port map(
    din  => dout_noenc,
    p    => dout_ecc(6 downto 0),
    err  => err,
    dout => dout_i
  );

  dout_mux21gen_inst: mux21gen
  generic map(
    n => 32
  )
  port map(
    a => dout_i,
    b => dout_noenc,
    s => outmux,
    c => dout
  );

  sbiterr <= err(0) and not(outmux);
  dbiterr <= err(1) and not(outmux);

end read_first;
```

Listing E.6. ECC encoder

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.arith.all;
use work.hrel_attr.all;

entity encoder is
  port(
    din:  in  std_logic_vector(31 downto 0);
    dout: out std_logic_vector(63 downto 0)
  );
end entity;

architecture structural of encoder is

  attribute syn_radhardlevel of structural: architecture is "none";

  signal by: byte_word;
  signal c:  hor_parity;
  signal p:  std_logic_vector(7 downto 0);

begin

  HPA: for i in 0 to 3 generate
    by(i)   <= din((8*(i+1))-1 downto 8*i);
    c(i)(0) <= by(i)(0) xor by(i)(1) xor by(i)(3) xor by(i)(4) xor by(i)(6);
    c(i)(1) <= by(i)(0) xor by(i)(2) xor by(i)(3) xor by(i)(5) xor by(i)(6);
    c(i)(2) <= by(i)(1) xor by(i)(2) xor by(i)(3) xor by(i)(7);
    c(i)(3) <= by(i)(4) xor by(i)(5) xor by(i)(6) xor by(i)(7);
    c(i)(4) <= by(i)(0) xor by(i)(1) xor by(i)(2) xor by(i)(3) xor \
  by(i)(4) xor by(i)(5) xor by(i)(6) xor by(i)(7);
  end generate;

  VPA: for i in 0 to 7 generate
    p(i) <= by(0)(i) xor by(1)(i) xor by(2)(i) xor by(3)(i);
  end generate;

  dout(31 downto 0)  <= din;
  dout(39 downto 32) <= p;
  dout(44 downto 40) <= c(0);
  dout(49 downto 45) <= c(1);
  dout(54 downto 50) <= c(2);
  dout(59 downto 55) <= c(3);
  dout(63 downto 60) <= "0000";

end structural;
```

Listing E.7. ECC decoder

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.arith.all;
use work.hrel_attr.all;

entity decoder is
  port(
    din:  in  std_logic_vector(63 downto 0);
    dout: out std_logic_vector(31 downto 0)
  );
end entity;

architecture structural of decoder is

  component encoder is
    port(
      din:  in  std_logic_vector(31 downto 0);
      dout: out std_logic_vector(63 downto 0)
    );
  end component;

  attribute syn_radhardlevel of structural: architecture is "none";

  signal by:       byte_word;
  signal by_ser:   byte_word;
  signal mask_ser: byte_word;
  signal sp_med:   byte_word; -- Syndorme parity bits xored with med
  signal ci:       hor_parity;
  signal cr:       hor_parity;
  signal cser:     hor_parity;
  signal sc:       hor_parity; -- Syndrome check bits
  signal scser:    hor_parity; -- Syndrome check bits
  signal med:      std_logic_vector(3  downto 0); -- Multiple error detection bits
  signal pi:       std_logic_vector(7  downto 0);
  signal pser:     std_logic_vector(7  downto 0);
  signal sp:       std_logic_vector(7  downto 0); -- Syndorme parity bits
  signal din_enc:  std_logic_vector(63 downto 0);
  signal din_ser:  std_logic_vector(31 downto 0);
  signal tmp_ser:  std_logic_vector(63 downto 0);

begin

  enc_inst0: encoder
  port map(
    din  => din(31 downto 0),
    dout => din_enc
  );

  enc_inst1: encoder
  port map(
    din  => din_ser(31 downto 0),
    dout => tmp_ser
  );

  pi    <= din(39 downto 32);
  ci(0) <= din(44 downto 40);
  ci(1) <= din(49 downto 45);
  ci(2) <= din(54 downto 50);
  ci(3) <= din(59 downto 55);

  cr(0) <= din_enc(44 downto 40);
  cr(1) <= din_enc(49 downto 45);
  cr(2) <= din_enc(54 downto 50);
  cr(3) <= din_enc(59 downto 55);

  pser    <= tmp_ser(39 downto 32);
```

```
  cser(0) <= tmp_ser(44 downto 40);
  cser(1) <= tmp_ser(49 downto 45);
  cser(2) <= tmp_ser(54 downto 50);
  cser(3) <= tmp_ser(59 downto 55);

  -- Generate syndrome for check bits and single error correction in the rows

  CORR: for i in 0 to 3 generate
    sc(i)   <= cr(i) xor ci(i);
    by(i)   <= din((8*(i+1))-1 downto 8*i);

    MASK_PROC: process(sc)
      variable mask_ser_var: byte_word;
    begin
      mask_ser_var(i) := (others => '0');
      mask_ser_var(i)(3) := sc(i)(0) and sc(i)(1) and sc(i)(2) and sc(i)(4);
      if (mask_ser_var(i)(3)='0') then
        mask_ser_var(i)(6) := sc(i)(0) and sc(i)(1) and sc(i)(3) and sc(i)(4);
        if (mask_ser_var(i)(6)='0') then
          mask_ser_var(i)(0) := sc(i)(0) and sc(i)(1) and sc(i)(4);
          if (mask_ser_var(i)(0)='0') then
            mask_ser_var(i)(1) := sc(i)(0) and sc(i)(2) and sc(i)(4);
            if (mask_ser_var(i)(1)='0') then
              mask_ser_var(i)(2) := sc(i)(1) and sc(i)(2) and sc(i)(4);
              if (mask_ser_var(i)(2)='0') then
                mask_ser_var(i)(5) := sc(i)(1) and sc(i)(3) and sc(i)(4);
                if (mask_ser_var(i)(5)='0') then
                  mask_ser_var(i)(4) := sc(i)(0) and sc(i)(3) and sc(i)(4);
                  if (mask_ser_var(i)(4)='0') then
                    mask_ser_var(i)(7) := sc(i)(2) and sc(i)(3) and sc(i)(4);
                  end if;
                end if;
              end if;
            end if;
          end if;
        end if;
      end if;
      mask_ser(i) <= mask_ser_var(i);
    end process;

    --by_ser(i) <= by(i);
    by_ser(i) <= by(i) xor mask_ser(i);
    din_ser((8*(i+1))-1 downto 8*i) <= by_ser(i);

    -- Generate multipler error detection bits

    scser(i) <= cser(i) xor ci(i);
    med(i)   <= scser(i)(0) or scser(i)(1) or scser(i)(2) or scser(i)(3) or scser(i)(4);

    SPMED: for j in 0 to 7 generate
      sp_med(i)(j) <= med(i) and sp(j);
    end generate;

    dout((8*(i+1))-1 downto 8*i) <= by_ser(i) xor sp_med(i);
  end generate;

  -- Generate syndrome for parity bits and multiple error correction between rows

  sp <= pi xor pser;

end structural;
```

Bibliography

[1] Indira Gandhi Centre, *Understanding radiation effects in SRAM-based field programmable gate arrays for implementing instrumentation and control systems of nuclear power plants*, T.S. Nidhin, Anindya Bhattacharyya, R.P. Behera, T. Jayanthi, K. Velusamy, 2017.

[2] IEEE, *The Impact of Radiation-Induced Failure Mechanisms in Electronic Components on System Reliability*, Donald C. Mayer, Rokutaro Koga, James M. Womack, 2007.

[3] NSF Center for High-Performance Reconfigurable Computing (CHREC) Deptartment of Electrical and Computer Engineering Brigham Young University, *SEU Mitigation and Validation of the LEON3 Soft Processor Using Triple Modular Redundancy for Space Processing* Michael Wirthlin, Andrew Keller, Chase McCloskey, Parker Ridd, 2012

[4] Springer, *Electronics System Design Techniques for Safety Critical Applications* Luca Sterpone, 2008

[5] IEEE, *Designing fault-tolerant techniques for SRAM-based FPGAs*, Wali, I., Virazel, A., Bosio, A. et al. J Electron Test (2016) 32: 147. 2004

[6] Springer, *A Hybrid Fault-Tolerant Architecture for Highly Reliable Processing Cores*, Priyanka P. Ankolekar, Roger Isaac, and Jonathan W. Bredow

[7] IEEE, *Multibit Error-Correction Methods for Latency-Constrained Flash Memory Systems*, F. G. de Lima Kastensmidt,G. Neuberger, R. F. Hentschke,L. Carro,R. Reis, 2010

[8] CERN, *Technical Design Report of the Inner Tracking System (ITS)*, 1999.

[9] Michael Wirthlin, *High-Reliability FPGA-Based Systems: Space, High-Energy Physics, and Beyond*, IEEE Vol.103, No.3, 2015.

[10] Xilinx, *UltraScale Architecture Libraries Guide*, v1.5, 2017.

[11] Xilinx, *UltraScale Architecture CLB User Guide*, v1.5, 2017.

[12] Xilinx, *Preliminary Product Specification*, v3.2, 2018.

[13] Xilinx, *UltraScale Architecture Memory Resources*, v1.9, 2018.

[14] Xilinx, *UltraScale Architecture Clocking Resources*, v1.6, 2017.

[15] AVNET, *Kintex UltraScale KU040 Development Board, Hardware User Guide*, v1.0, 2015.

[16] CERN, *GBT-SCA, the slow control adapter ASIC for the GBT system user manual*, v8.2, 2017.

[17] Andrew Waterman, Krste Asanovic SiFive Inc., 2CS Division, EECS Department, University of California, Berkeley, *The RISC-V Instruction Set Manual Volume I: User-Level ISA*, v2.2, 2017.