electronics

MDPI

Article

Fine-Grain Circuit Hardening Through VHDL Datatype Substitution

Maria Muñoz-Quijada, Samuel Sanchez-Barea, Daniel Vela-Calderon and Hipolito Guzman-Miranda *

Department of Electronic Engineering, Universidad de Sevilla, Camino de los Descubrimientos s/n, 41092 Sevilla, Spain; mamuki92@gmail.com (M.M.-Q.); samuelsanchezbarea@gmail.com (S.S.-B.); danvelcal96@gmail.com (D.V.-C.)

* Correspondence: hguzman@us.es; Tel.: +34-954-481-298

Received: 30 November 2018; Accepted: 23 December 2018; Published: 25 December 2018

Abstract: Radiation effects can induce, amongst other phenomena, logic errors in digital circuits and systems. These logic errors corrupt the states of the internal memory elements of the circuits and can propagate to the primary outputs, affecting other onboard systems. In order to avoid this, Triple Modular Redundancy is typically used when full robustness against these phenomena is needed. When full triplication of the complete design is not required, selective hardening can be applied to the elements in which a radiation-induced upset is more likely to propagate to the main outputs of the circuit. The present paper describes a new approach for selectively hardening digital electronic circuits by design, which can be applied to digital designs described in the VHDL Hardware Description Language. When the designer changes the datatype of a signal or port to a hardened type, the necessary redundancy is automatically inserted. The automatically hardening features have been compiled into a VHDL package, and have been validated both in simulation and by means of fault injection.

Keywords: radiation hardening; hardening by design; TMR; selective hardening; VHDL

1. Introduction

1.1. Background

Ionizing radiation affects the normal operation of electronic circuits. Different kind of effects may produce both physical degradation of the components, like TID (Total Ionizing Dose) or DD (Displacement Damage), or corruption of the logic values stored in the circuit, such as SEU (Single Event Upset), SET (Single Event Transient) or MBU (Multi-Bit Upset) [1]. The former category of effects, known as hard errors, are destructive in nature and must be protected against by using specific technology approaches. Soft errors, on the other hand, induce modifications in the internal states of the circuits, which may or may not then propagate both inside the circuit architectures and to their primary outputs. Errors propagating to the primary outputs of a circuit may escalate to external systems and produce device errors, subsystem failures or even catastrophic mission failures. These soft errors can be mitigated by inserting logic protections in the designs [2,3].

1.2. Problem of Interest

These logic protections can be inserted at different steps during the design flow. Typically, these protections are inserted either during the synthesis process or just after the synthesis process has completed, but before the placement and routing steps. These approaches require design teams to implement changes to their design flows, either by including specific proprietary synthesis tools or

extra post-synthesis netlist manipulation software, both of which have to be adapted and configured for the mission requirements, which demands extra effort from the designers.

When developing hardware modules cores that are expected to need some selective protections, but not full redundancy, it would be desirable to include the information on which elements should be hardened in the module code itself, in a non-synthesizer-specific way, since different developers and projects may choose or require different synthesis tools. An ideal situation would allow the designer to easily specify in the VHDL (Very High Speed Integrated Circuit Hardware Description Language) source code which elements should be hardened, with minimal code modifications.

1.3. Literature Survey

There are multiple types of protections that can be inserted in a digital circuit [4], from which the most common one is the full triplication of single memory elements, which is known as Triple Modular Redundancy or TMR. TMR is typically preferred to DMR (Dual Modular Redundancy) since the former can detect and correct single errors, but the latter has only detection, but no correction capabilities. The tradeoff for this is that TMR uses more area and power (around a 3.2× factor, instead of a ~2.1× factor for DMR, compared with the unhardened design [5]). TMR can be applied at both flip-flop level or module level, but DMR is more typically applied at module level.

Selective hardening is a more recent technique that involves identifying the most sensitive modules of a design (for example, by means of fault injection), and then applying TMR only to those modules. This way, a better tradeoff between area/power increase and error mitigation is achieved, since modules that do not contribute much to the Architectural Vulnerability Factor (AVF) of the design [6] are left unmodified and their power/area will not be affected by the aforementioned ~3.2× factor [7,8].

Hardening techniques can be applied during the synthesis process. An example of this are the protections inserted by some proprietary synthesizers that allow hardening of full modules, or even applying local TMR attributes to the specific signals that need to be hardened. The Synopsys Synplify pro [9] and Mentor Precision Hi-rel [10] synthesizers are examples of this.

Another way of inserting mitigation schemas into the designs is to perform post-synthesis netlist manipulation, for example using software such as the Xilinx XTMRtool [11] and the BYU (Brigham Young University) EDIF (Electronic Design Interchange Format) tools [12]. The former allows full module hardening in a Xilinx-specific design flow, and the latter is a software suite that can insert both TMR and DWC (Duplicate With Compare) for the user-selected elements. Mitigation elements may also be manually inserted in the post-synthesis netlist, but this process is error-prone and thus not recommended.

Approaches that insert protections during the synthesis process, or just after it, work at the RTL (Register-Transfer Level) netlist abstraction level and thus do not consider physical implementation aspects that may affect the robustness of the implemented design. Depending on whether the target technology on which the digital design will be implemented is an FPGA (Field Programmable Gate Array) or an ASIC (Application-Specific Integrated Circuit), other complementary approaches can be used at the place and route level to improve the robustness of the implemented design, for example physically separating the redundant copies of a hardened element, which improves tolerance to Domain Crossing Errors (DCE) [13]. For the ASIC design of the hardened microprocessor HERMES [14], both DMR and TMR techniques were implemented, depending on which processor block was to be hardened, and the replicated redundancy domains were physically separated during the circuit layout design phase. Another approach in fine-grain techniques is the one proposed on [15], in which design flip-flops are replaced by self-correcting rad-hard by design (RHBD) flip-flops after synthesis, and triplication is performed on spatially separated regions during the placement phase. For FPGA designs, actions can be taken during the placement and routing implementation stages, such as inserting redundant routing connections [16] or using reliability-oriented place and route algorithms to physically separate the redundant copies and avoid single points of failure [17]. Unused FPGA

resources may also be employed for error detection: [18] proposes the use of carry propagation chains, which is a common FPGA resource, as a way to create fine-grained comparators to detect bit upsets, which is complemented by the use of coarse-grain checkers that can determine whether the detected upsets did actually propagate to the main module outputs.

1.4. Scope and Contribution of This Paper

Of the previous approaches that work at the RTL netlist level, there is no single approach that allows for both easy insertion of mitigations by performing minimal modifications in the HDL code, and independence from the synthesis tool. In-code fine-grain selection of which elements should be hardened, that propagates to both arithmetic/logic operations performed, and flip-flops used to store them, would be desirable.

This paper proposes a new technique for performing selective, fine-grain circuit hardening, that allows designers to include the information on which combinatorial and sequential elements should be hardened in the VHDL code. In order to be selective, the technique allows designers to individually choose which elements of the VHDL code to harden. To be useful for designers, the technique only implies minimal code substitution and does not change the functionality of the design in absence of soft errors. The technique is also portable between different VHDL synthesizers and does not require the use of post-synthesis tools to generate the hardened netlist.

The difference between the proposed technique and proprietary approaches such as [9–11] is that the proposed technique can be used across different synthesizers. Also, while [11] must harden complete modules, our technique allows selection of which elements are to be hardened.

Since VHDL allows for both Behavioral and RTL descriptions, the technique can work at both abstraction levels and thus its scope does not include physical layout techniques, but it can be complemented with them.

1.5. Organization of the Paper

The paper is structured as follows: Section 2 describes the proposed approach, with the developed datatypes and operators. Section 3 describes how the approach was verified, both in simulation, to check functional correctness of the hardened designs, and by means of fault injection, to check the correctness of the protection implementations. Finally, the discussion and conclusions are presented in Section 4.

2. The Triple_logic Package

In this article, we propose a new approach to implement fine-grain circuit hardening for digital designs by just changing the datatype of the object to be hardened. By changing the object types, the implementation changes accordingly to introduce the desired redundancy. The designer can then select which nodes of the circuit should be hardened, thus creating redundancy domains for the critical parts of the design. Figure 1 shows a redundant branch of a design, and represents graphically how to pass from a non-hardened domain to a hardened domain, where redundant operations and data storage are performed, and back to the non-hardened domain. It must be noted that, in this context, domain crossing refers to user data passing from the non-hardened to the hardened domain or vice versa, and not to the propagation of errors between redundant copies of the design elements.

We have compiled all the new datatypes and hardening functionality in a VHDL package for ease of use and minimal VHDL code modification. An important feature of the package is avoiding the scenario present in Figure 2, where the robustness of the hardened domain is jeopardized by a Single Point of Failure introduced by premature voting inside the hardened domain. To avoid this situation, transitions between hardened and non-hardened domains are determined by the datatypes of the intervening operands. For example, if an operation receives two hardened operands and must return a non-hardened result, a voter will be inserted, but if the result data type is of a hardened type, no voter will be implemented.

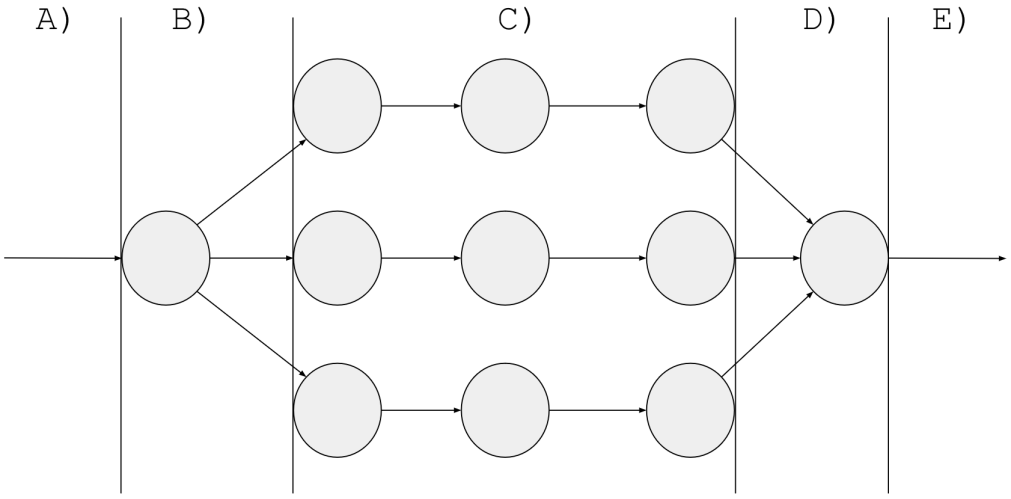

Figure 1. Domain crossing between non-hardened and hardened domains. Each element in the graph may represent either a combinatorial operation or a memory element. (**A**) Non-hardened domain. (**B**) Crossing to hardened domain. (**C**) Hardened domain. (**D**) Crossing to non-hardened domain. (**E**) Non-hardened domain.

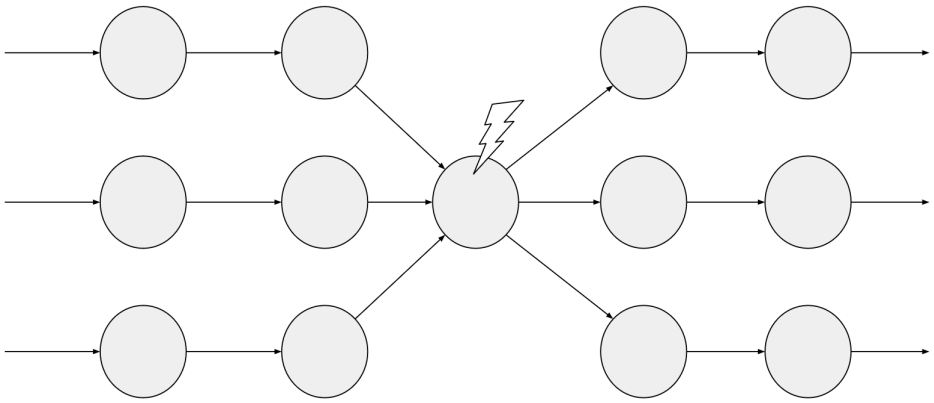

Figure 2. Single Point of Failure introduced inside a redundant domain by premature voting.

2.1. Data Types

Before implementing the automatic hardening functionality mentioned before, the new hardened data types that will compose the hardened domains must be defined. Since the most used standard data types are based on the `std_logic` data type, defined in the `std_logic_1164` package of library `IEEE`, a `triple_logic` datatype has been defined that comprises three `std_logic` values. By defining a vector of `triple_logic` values, the `triple_logic_vector` is created. `triple_unsigned` and `triple_signed` are hardened vectors with numeric interpretation, just as their non-hardened counterparts. Finally, a `triple_integer` contains three integers, whose range can be parametrized if using the IEEE Std.1076-2008 revision of the language, more widely known as VHDL-2008 [19]. Table 1 shows the equivalence between hardened and non-hardened data types.

The package defines logic and arithmetic operators for the new datatypes, and for mixed operations between these and the already existing ones. The operator and function overload capability of VHDL will allow an operation (for example, a sum) to receive any combination of datatypes in its input and return operands, and the relevant implementation will be automatically selected depending on the actual data types.

Table 1. Equivalence between non-hardened and hardened data types.

Non-Hardened	Hardened
`std_logic`	`triple_logic`
`std_logic_vector`	`triple_logic_vector`
`unsigned`	`triple_unsigned`
`signed`	`triple_signed`
`integer`	`triple_integer`

2.2. Hardened to Non-Hardened Domain Crossing

Once all data types and operations have been defined, special consideration must be taken into how to pass data between the non-hardened and hardened domains. The function/operator overload capability of VHDL allows for this domain crossing to be performed automatically for all operator results, but when making a single assignment without any operations this cannot be automatically done, as VHDL is strongly typed and thus the assignment operator cannot be overloaded. We have developed two functions for these cases: a `vote()` function to pass from the hardened domain to the non-hardened domain (Figure 3), and a `triple()` function to perform the opposite operation (Figure 4).

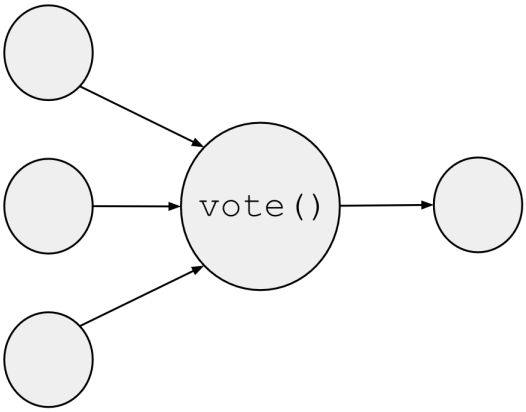

Figure 3. Graphic illustration of `vote()` function.

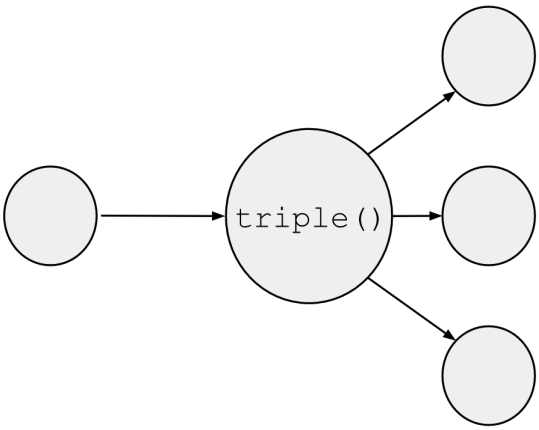

Figure 4. Graphic illustration of `triple()` function.

Both functions, `vote()` and `triple()`, are overloaded so that the user can pass every equivalent data type from the non-hardened domain to the hardened domain, and vice versa, with the same two functions.

2.3. Developed Functionality

After the development of the datatypes and the `vote()` and `triple()` functions, logic, arithmetic and comparison operators were developed for these datatypes.

2.3.1. Operator List

The operators developed for the hardened datatypes are logic (AND, NAND, OR, NOR, XOR, XNOR), comparison (= [is equal], /= [is not equal], > [greater than], >= [greater or equal], < [lower than], <= [lower or equal] and arithmetic operators (+ [addition], - [substraction], * [multiplication], / [division]). Since not every operator is available for every non-hardened datatype (for example, `std_logic_vector` does not have numerical interpretation, and integers do not support bitwise operations), not all operators have been implemented for all datatypes. The list of implemented operators is shown in Table 2.

The assignment operator (<= for signals, := for variables) may not be overloaded since VHDL is strongly typed.

Table 2. List of implemented operators.

Datatype	Logic	Equality/Inequality	Rest of Comparison Operators	Arithmetic
`triple_logic`	yes	yes	yes	no
`triple_logic_vector`	yes	yes	no	no
`triple_unsigned`	yes	yes	yes	yes
`triple_signed`	yes	yes	yes	yes
`triple_integer`	no	yes	yes	yes

2.3.2. Operator Variants

Due to operator overload, for each of the operators, we have developed a number of variants. This way, domain crossing is performed by automatically choosing the appropriate operator variant, which is done by the synthesis tools and simulators. For example, the statement A <= B + C will assign a hardened or non-hardened value to A depending on its data type. For unary operators, there are four combinations according to whether the operand and result are hardened or not. For binary operators, there are eight possibilities. All these possibilities are shown in Table 3. Of course, the possibilities that correspond to all values in the non-hardened domain are already defined in the `std_logic_1164` or `numeric_std` packages so they do not need to be defined again.

Table 3. Operator Variants.

Unary Operators		
Operand	**Result**	
unhardened	unhardened	
unhardened	hardened	
hardened	unhardened	
hardened	hardened	
Binary Operators		
Left Operand	**Right Operand**	**Result**
unhardened	unhardened	unhardened
unhardened	unhardened	hardened
unhardened	hardened	unhardened
unhardened	hardened	hardened
hardened	unhardened	unhardened
hardened	unhardened	hardened
hardened	hardened	unhardened
hardened	hardened	hardened

The current implementation of the hardening functionality includes all operator variants in the same VHDL file, but those operator variants could also be separated into different files, in case the designer wants to automate domain crossing in one direction but not on the other. In that case, the functions that automatically cross from the unhardened to the hardened domain, the functions that automatically cross from the hardened domain to the unhardened one, and the functions that operate only on the hardened domain would be defined in different files. This way, the user could choose one of these four possibilities, depending on which files are included:

1. Automatically cross domains from the unhardened to the hardened one, but manually use the `vote()` function to go back to the unhardened domain.
2. Automatically cross domains from the hardened to the unhardened one, but manually use the `triple()` function to go back to the hardened domain.
3. Automatically perform all domain crossing operations. In this case, qualified expressions of VHDL may be needed to solve ambiguity in some cases. For example, the statement `B <= not (not A);` becomes ambiguous, because even if A and B are known types, the innermost `not` operator does not know whether it should return a hardened or unhardened result. This is resolved by specifying the desired return type for the intermediate operations, for example: `B <= not std_logic'(not A);`.
4. Manually perform all domain crossing operations.

2.3.3. Hardening Finite State Machines

Hardening Finite State Machines (FSMs) is not trivial when FSMs use an enumerated data type, which is a common practice. A custom solution can be implemented for each FSM, by defining a `decode()` and `triple()` function for the hardened version of their state datatype. Both functions are used for domain crossing: when decoding the state of the FSM, the first function returns the correct state, after correcting errors, and when assigning a new state, the second function converts the enumerated constant to a hardened value. This is a needed tradeoff in order to have fine-grain hardening with minimal code modifications, since on every possible state many signals may be assigned, and the designer may not want to harden all of them.

These functions can be made generic for every enumerated datatype if using VHDL-2008, and can be used with the rest of the package when using a VHDL-2008 capable synthesizer. When full TMR is not needed in the state registers, the technique allows the user to implement his own EDAC (Error Detection and Correction) functions to encode and decode the FSM state instead of triplicating it, for example by defining the functions `encode()` and `decode()` to add Hamming codes to the state registers.

2.4. Usage Examples

A couple of usage examples follow. Figures 5 and 6 show a hardened multiplexer and a hardened generic-width counter, with minimal code modifications, which are underscored. For the designer, it is clear from the signal and port datatypes which objects belong to the hardened domain.

To prevent the synthesizer from removing the redundancy, attributes can be applied to the tripled registers. The name of the specific attribute depends on the chosen synthesis tool, for example, when using Synopsys Synplify the attribute `syn_preserve` can be used, whereas in Xilinx XST (Xilinx Synthesis Technology) the relevant attributes are called `keep` and `equivalent_register_removal`.

```
entity mux2to1 is
    port ( input_l : in  triple_logic;
           input_r : in  triple_logic;
           sel     : in  triple_logic;
           output  : out triple_logic);
end mux2to1;

architecture arch of mux2to1 is
begin
  comb: process (input_l, input_r, sel)
  begin
    if (sel = '0') then
      output <= input_l;
    else
      output <= input_r;
    end if;
  end process;
end arch;
```

Figure 5. Hardened 2-to-1 multiplexer. Note that the equality comparison operator is overloaded, so `sel` can be compared to `'0'`.

```
architecture arch of contparam is
       signal reg_i, p_reg_i: triple_unsigned (N-1 downto 0);
begin

  comb: process (reg_i, enable, updown)
  begin
      if (enable = '1') then
              if (updown = '1') then
                      p_reg_i <= reg_i + 1;
              else
                      p_reg_i <= reg_i - 1;
              end if;
      else
              p_reg_i <=reg_i;
      end if;
  end process;

  sinc: process (clk,rst)
  begin
      if (rst = '1') then
              reg_i <= (others => (others => '0'));
      elsif (rising_edge(clk)) then
              reg_i <= p_reg_i;
      end if;
  end process;

        data_out <= std_logic_vector(vote(reg_i));

end arch;
```

Figure 6. Hardening an N-bit counter architecture. Note that the only modifications are the change in the datatype of the internal count, its reset value, and the voting for the primary output, which belongs to the non-hardened domain.

3. Package Verification

To check the correct behaviour of the package, a number of test cases have been generated. Both basic functionality and designs of increasing levels of complexity have been tested. Synthesis, simulation and fault injection results have been obtained to verify that not only the inserted protections mitigate effectively against SEU, but also that the added functionality does not change the expected circuit functionality in the absence of SEU.

Synthesis has been performed with Xilinx ISE (Integrated Synthesis Environment) 14.7 and Synopsys Synplify v4.2. The simulations have been performed with Xilinx ISim (ISE Simulator)

version 14.7. The fault injection campaigns have been performed with the FT-Unshades2 (Fault Tolerance—Universidad de Sevilla Hardware Debugging System) fault injection platform [20], version 3.10, working in ASIC mode, which means injections are performed in the user flip-flops.

The Yosys Open SYnthesis Suite [21] has been used to formally verify design equivalence between the hardened and non-hardened versions of the smaller designs, described below, such as counter and shiftreg. The formal equivalence checker tries to solve a boolean satisfiability problem (abbreviated SAT). In this case, the solver must check if there is any input combination that would make the outputs of the hardened and unhardened design differ, and prove by induction that the design outputs will not differ at any time in the future, for any possible set of input vectors. For some of the other designs, even if full formal equivalence cannot be demonstrated because of their complexity, hundreds of induction steps have been performed without any equivalence error being encountered. The simulations also show that the output of the hardened and unhardened versions of all designs are the same, when no SEU are being injected.

3.1. Primitive Verification

To validate the smallest package functionality, a number of test cases have been generated, which have been checked both in simulation, checking correct behaviour against transient errors, and by reviewing the generated netlist topologies. To check the primitives, synthesis has been performed with the XST synthesizer, but results are expected to be reproducible with any other VHDL synthesizer. No optimization of the inserted protections has been detected when synthesizing with XST, but in the case of these optimizations happening with other synthesizers, VHDL attributes can be added to the hardened signals to avoid removal of the hardening elements. Figures 7 and 8 show the synthesized netlist and a short simulation of one of the developed primitives.

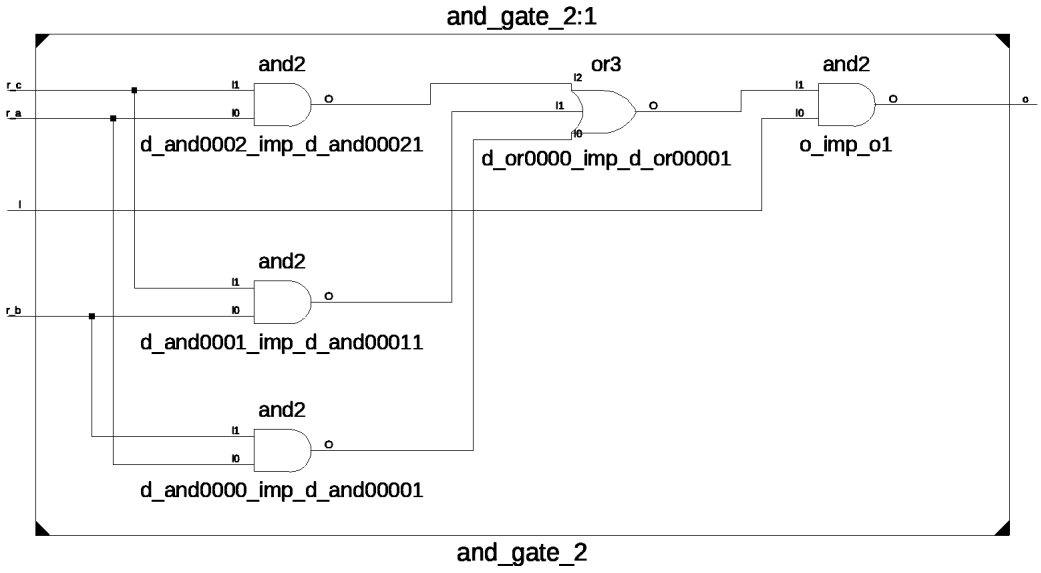

Figure 7. Internal logic structure of AND gate with right port hardened.

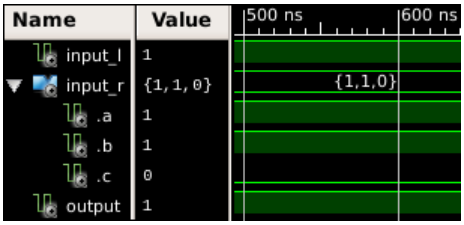

Figure 8. Simulation results of AND gate with right port hardened, with a transient error in its right input.

3.2. Designs Under Test

A number of VHDL designs with increasing levels of complexity have been chosen to validate the hardening capabilities of the package. For each of these designs, an SEU fault injection campaign has been performed with FT-Unshades2, in order to identify the most critical registers, which will be hardened by using the methodology proposed in this work. The hardened versions of the designs have also been subjected to fault injection campaigns, to check the effectiveness of the inserted protections. In order to validate that the technique can be used with different synthesizers, synthesis of both hardened and unhardened designs has been performed with Xilinx ISE and Synopsys Synplify. Finally, synthesis results have also been obtained with the NanoXplore NanoXmap synthesizer version 2.9.1, but the results of this synthesis cannot be tested in the current version of FT-Unshades2, since this synthesizer targets the NanoXplore NG-MEDIUM FPGA and the current version of FT-Unshades2 uses a Virtex-5 FPGA.

- counter

  An 8-bit up counter with an enable signal.

- shiftreg

  An 8-bit shift register. In this example, flip-flops turn into shift registers when synthesis is made with XST so they are optimized even if "keep" attribute is set. To avoid this, "Shift Registers Extraction" and "Equivalent Register Removal" synthesis options have been unselected for this design.

- simple_fsm

  A 4-state simple state machine design, described specifically for this work. In the unhardened version, when using the default synthesis options, XST uses binary codification for synthesis. However, keeping the default synthesis options, in the hardened version, one-hot codification is used for synthesis, so the number of FF (flip-flops) increases from 2 to 12 (4 bits, triplicated). This is the worst case in terms of area overhead, but it can be controlled by the user, by specifying the desired FSM encoding during synthesis. For example, the user can change the FSM encoding from binary to one-hot when hardening the design, in order to reduce the area overhead of hardening the FSM state register, if the timing constraints allow for a slower state decoding. Table 4 shows flip-flop usage for this design in all its possible variants.

Table 4. Simple_fsm state flip-flops.

FSM Encoding	Unhardened	Hardened
one-hot	4	12
binary	2	6

  For the fault injection experiments both versions (hardened and unhardened) of the simple_fsm design have been synthesized using one-hot encoding, when synthesizing with XST, and binary encoding, when synthesizing with Synplify, to show that the FSM hardening can be performed independently of the encoding.

- adder_acum

  A simple adder-accumulator design that accumulates the sum of 8-bit input values into a 20-bit register.

- fifo

  A generic 256-bit depth and 32-bit width FIFO (First in, First Out) memory buffer with Empty and Full flags [22].

- fft

  Fast Fourier Transform module for usage on FPGA devices [23].

- fir_ri

  A low pass FIR (Finite Impulse Response) filter [24].

- pcm3168

  An I^2S interface designed for the PCM3168 audio interface from Texas Instruments [25].

- 8051

  A VHDL model of The Intel 8-bit 8051 micro-controller [26]. This design, which has more complexity than the others, has been tested with a simple program written in C.

3.3. Experimental Results

Injection campaigns have been performed for all the test designs, and their results have been analyzed by comparing the number of flip-flops, AVF and lines of code changed between hardened and unhardened designs. AVF has been estimated by making N injections in a set of FF and dividing the number of injections that produce output errors by the number of total injections (N). For each injection, the complete test vectors are executed by the design.

Designs with a low percentage of total FFs and less number of clock cycles like counter, shiftreg, simple_fsm, fir_ri or adder_acum have been tested with exhaustive campaigns. However, designs with a higher occupancy and more clock cycles, like pcm3168, fft, fifo or 8051 have been tested with random campaigns checking that the number of injections performed on these campaigns is enough to assure less than 5% of error, with a confidence level of 99%, according to [27].

For each design, Table 5 shows the name of registers with damages due to the injections performed in every campaign. The results of these campaigns have been analyzed to determine the AVF of each register with damage, as it can be seen in the fourth column of the table.

To determine which registers will be hardened, the percentage of FFs in the register by FFs in the design has been calculated and those that have more percentage of FF with a higher AVF have been selected to be hardened.

Tables 6 and 7 show results for hardened designs synthesized with XST and Synplify respectively. A comparison between hardened and unhardened designs versions has been done to check the effectiveness of the package. The first column contains the name of the hardened version of the design (in bold) followed by the name of the hardened registers. In the second one, the number of different code lines between the hardened and unhardened versions and the resulting percentage against the total lines in the design are shown. Third and fourth columns present the AVF both for the complete design and each register and the number of FFs obtained in each version.

Table 8 compares synthesis results with three synthesis tools (XST, Synplify and NanoXmap), showing that the proposed technique can be used with different synthesizers, avoiding vendor lockdown.

An estimation of the power consumption for each design is also shown in Tables 9 and 10 for XST and Synplify synthesis respectively. As power consumption depends on which target technology is going to be used, we have made this estimation using the XPower analyzer tool from Xilinx [28], assuming that designs will be implemented for an FPGA, specifically the Virtex-5

XC5VFX70T. According to this, results presented show logic (flip-flops and lookup tables) and signal (interconnections) power consumption estimation both for the unhardened and the hardened versions, and the increase incurred by using our approach, in absolute value and percentage.

Table 5. Designs under test, synthesized with Synplify, with register signals classified by percentage of total number of flip-flops and Architectural Vulnerability Factor.

Design: counter Total Flip-flops: 8			
Signals	**Flip-flops**	**% FFs**	**AVF**
reg	8	100	98.60
Design: shiftreg Total Flip-flops: 8			
Signals	**Flip-flops**	**% FFs**	**AVF**
reg	8	100	97.54
Design: simple_fsm Total Flip-flops: 2			
Signals	**Flip-flops**	**% FFs**	**AVF**
state_FSM	2	100	68.75
Design: adder_acum Total Flip-flops: 8			
Signals	**Flip-flops**	**% FFs**	**AVF**
acc_value	20	100	97.50
Design: pcm3168 Total Flip-flops: 95			
Signals	**Flip-flops**	**% FFs**	**AVF**
s_bit_clk1	1	1.05	91.84
s_counter_bit	2	2.11	95.83
s_counter_lr	5	5.26	74.31
s_lr_clk	2	2.11	91.38
v_lr_clk_enable	1	1.05	58.18
DATA_L	24	25.26	40.04
DATA_R	24	25.26	60.63
s_current_lr	1	1.05	52.38
shift_reg	24	25.26	16.54
s_parallel_load	1	1.05	51.22
counter	5	5.26	3.77
DOUT	2	2.11	83.67
Design: fifo Total Flip-flops: 34			
Signals	**Flip-flops**	**% FFs**	**AVF**
looped	1	0.03	100.00
Tail	8	23.52	45.00
Head	8	23.52	60.00
Design: fft Total Flip-flops: 929			
Signals	**Flip-flops**	**% FFs**	**AVF**
o_im	9	0.97	100.00
o_re	9	0.97	100.00

Table 5. *Cont.*

Design: fir_ri Total Flip-flops: 86			
Signals	**Flip-flops**	**% FFs**	**AVF**
N_bit_reg/Q	9	0.97	95.97
shift_reg	9	0.97	93.40
Design: 8051 Total Flip-flops: 1396			
Signals	**Flip-flops**	**% FFs**	**AVF**
/alu_op_code/	4	0.29	75.00
/alu_src/	8	0.59	15.79
/p0_out_c/	8	0.59	100.00
/p1_out_c/	8	0.59	100.00
/p2_out_c/	8	0.59	100.00
/p3_out_c/	8	0.59	100.00
/ram_wr	1	0.07	100.00
/U_CTR/exe_state/	3	0.22	100.00
/U_CTR/reg_pc_7/	8	0.59	75.00
/U_RAM/iram/	1024	75.13	5.34
/U_RAM/sfr_acc/	8	0.59	50.00
/U_RAM/sfr_psw/	8	0.59	50.00
/U_RAM/sfr_sp	8	0.59	20.00
/U_RAM/sfr_tmod/	8	0.59	20.00

Table 6. Hardened versions of the designs under test, synthesized with XST.

Design	**Code Modif.**		**AVF**			**FF**		
	(Lines)	**(%)**	**Unhardened**	**Hardened**	**Decrease (%)**	**Unhardened**	**Hardened**	**Increase (%)**
counter_v2	4	10.53	99.09	0.00	100.00	8	24	200.00
reg			99.09	0.00	100.00			
shiftreg_v2	5	11.90	86.89	0.00	100.00	8	24	200.00
reg			86.89	0.00	100.00			
simple_fsm_v2	49	73.13	53.13	0.00	100.00	4	12	200.00
state_FSM			53.13	0.00	100.00			
adder_acum_v2	8	21.05	97.50	0.00	100.00	20	60	200.00
i_acc_value			97.50	0.00	100.00			
pcm3168_v2	44	8.40	41.16	4.14	89.94	90	234	160.00
DATA_L			40.57	0.00	100.00			
DATA_R			56.00	0.00	100.00			
shiftreg			17.43	0.00	100.00			
fifo_v2	13	13.00	18.38	18.27	0.61	8243	8275	0.39
Tail			100.00	0.00	100.00			
Head			100.00	0.00	100.00			
fft_v2	42	8.73	1.53	1.02	33.33	387	723	86.82
o_im			100.00	0.00	100.00			
o_re			100.00	0.00	100.00			
fir_ri_v2	18	16.82	72.66	0.00	100.00	80	240	200.00
N_bit_reg/Q			72.70	0.00	100.00			
8051_v2	31	0.47	99.71	1.30	98.72	1327	1365	2.86
/U_CTR/reg_pc_7/			100	0.00	100.00			
/U_RAM/sfr_acc/			100	0.00	100.00			
/U_RAM/sfr_psw			100	0.00	100.00			

Table 7. Hardened versions of the designs under test, synthesized with Synplify.

Design	Code Modif.		AVF			FF		
	(Lines)	(%)	Unhardened	Hardened	Decrease (%)	Unhardened	Hardened	Increase (%)
counter_v2	6	15.79	98.60	0.00	100.00	8	24	200.00
reg			98.60	0.00	100.00			
shiftreg_v2	5	11.90	85.57	0.00	100.00	8	24	200.00
reg			97.54	0.00	100.00			
simple_fsm_v2	53	79.10	68.75	0.00	100.00	2	6	200.00
state_FSM			68.75	0.00	100.00			
adder_acum_v2	11	28.95	97.50	0.00	100.00	20	60	200.00
i_acc_value			97.50	0.00	100.00			
pcm3168_v2	42	8.02	38.51	3.93	89.79	95	244	156.84
DATA_L			40.04	0.00	100.00			
DATA_R			60.63	0.00	100.00			
shift_ref			17.19	0.00	100.00			
fifo_v2	13	13.00	17.55	17.48	0.40	34	87	155.88
Tail			45.00	0.00	100.00			
Head			60.00	0.00	100.00			
fft_v2	26	5.41	2.42	2.15	11.16	929	1061	14.21
rot2bf_im			100.00	0.00	100.00			
rot2bf_re			100.00	0.00	100.00			
fir_ri_v2	18	16.82	93.61	0.00	100.00	86	240	179.07
N_bit_reg/Q			95.97	0.00	100.00			
8051_v2	34	0.51	8.37	1.35	83.87	1396	1445	3.51
/U_CTR/reg_pc_7/			75	0.00	100.00			
/U_RAM/sfr_acc/			50	0.00	100.00			
/U_RAM/sfr_psw			50	0.00	100.00			

Table 8. Comparison of synthesis results. Data marked with an asterisk (*) corresponds to the synthesizer implementing an internal memory with Flip-flops instead of inferring a Block RAM.

Design	FF XST	FF Synplify	FF NanoXmap
counter	8	8	8
counter_v2	24	24	24
shiftreg	8	8	8
shiftreg_v2	24	24	24
simple_fsm	4	2	4
simple_fsm_v2	12	6	12
adder_acum	20	20	20
adder_acum_v2	60	60	60
pcm3168	90	95	91
pcm3168_v2	234	244	187
fifo	8243 *	34	23
fifo_v2	8275 *	87	55
fft	387	929	447
fft_v2	723	1061	783
fir_ri	80	86	80
fir_ri_v2	240	240	224
8051	1327	1396	1339
8051_v2	1365	1445	1359

Table 9. Power consumption estimation of hardened and unhardened versions of the designs under test, synthesized with XST.

Design	Unhardened Power (mW)		Hardened Power (mW)		Increase	
	(Logic)	(Signal)	(Logic)	(Signal)	Total	Percentage (%)
counter	0.10	0.30	0.33	0.97	0.90	225.00
shiftreg	0.00	0.00	0.00	0.00	0.00	N/A
simple_fsm	0.00	0.02	0.02	0.04	0.04	200.00
adder_acum	0.00	0.01	0.00	0.01	0.00	0.00
pcm3168	0.19	0.33	0.22	0.42	0.12	23.08
fifo	0.18	11.83	0.27	12.79	1.05	8.74
fft	1.55	4.13	2.56	5.68	2.56	45.07
fir_ri	0.00	0.19	0.03	0.27	0.11	57.89
8051	1.35	9.70	1.32	10.90	1.17	10.59

Table 10. Power consumption estimation of hardened and unhardened versions of the designs under test, synthesized with Synplify.

Design	Unhardened Power (mW)		Hardened Power (mW)		Increase	
	(Logic)	(Signal)	(Logic)	(Signal)	Total	Percentage (%)
counter	0.11	0.20	0.34	0.50	0.53	170.97
shiftreg	0.00	0.00	0.00	0.01	0.01	N/A
simple_fsm	0.00	0.03	0.01	0.04	0.02	66.67
adder_acum	0.00	0.01	0.00	0.02	0.01	100.00
pcm3168	0.13	0.48	0.18	0.61	0.18	29.51
fifo	0.11	0.52	0.20	1.33	0.90	142.86
fft	3.30	9.22	3.52	9.63	0.63	5.03
fir_ri	0.00	0.24	0.03	0.28	0.07	29.17
8051	1.75	16.80	1.97	16.60	0.02	0.11

The experimental results show that the selected registers can be hardened with the proposed approach, and that this protection is effective against SEU. When synthesizing the hardened designs with Synplify, it can be observed that the synthesizer not only triples the hardened flip-flops, but also may insert extra memory elements as a means of compensating the increased fan-out needs by the design, in a process known as timing-driven replication. It is very interesting to note that SEUs introduced in these new flip-flops do not produce errors in the output, which means that the relevant voting logic has also been propagated to these new memory elements, so the timing-driven replication does not negatively impact the effectiveness of the inserted protections. The AVF of the FIFO does not show much improvement, because only the flip-flops have been hardened, while SEU may affect the complete memory, which has many more sensitive elements.

Since the inserted redundancy is hardware redundancy and there are no time redundancy operations, the hardened designs take exactly the same number of clock cycles to perform their workload than their unhardened counterparts. Simulation execution time does not grow significantly: in small designs it varies less than a second, and in large designs it is less than 5%. This is coherent with what would be expected since, in the bigger designs, the entire design is not tripled, but only part of it, so the simulation time should not be tripled. Increments in simulation time for other designs will depend on what percentage of the design was hardened.

Finally, the power consumption increase of the hardened designs is in line with what is expected, according to the area increase of each design and an expected multiplication by a ~3.2 factor for each triplicated element.

4. Conclusions

A new approach to implement fine-grain circuit hardening, using datatype substitution, has been developed and validated. As a result, a VHDL package for selective circuit hardening by design has been developed as a new tool for mitigating soft errors on digital circuits, with minimal code modifications. The designer only has to select which signals or ports should be hardened and change their datatype accordingly. Some use of the `triple()` and `vote()` functions can be needed because of the strongly typedness of VHDL.

An interesting feature of this way of performing hardening by design is that the designer, after identifying the critical elements of his/her design using fault injection or other approaches, can embed in the source code of the module the information of which elements should be protected, thus eliminating the need to configure a second tool (such as a post-synthesis netlist processor) with the results of the vulnerability analysis.

Collaboration with synthesis tool vendors would improve the performance of the package to avoid some unwanted optimizations that may happen when performing multiple passes during the synthesis process, for example, when TMR flip-flops that would not be optimized, because correct signal attributes have been used, get converted to SRL16 primitives (Lookup tables used as Shift Registers) which in turn get optimized away. Another case of this is when hardened ports of internal modules get optimized by the synthesizer, because the attributes to avoid redundancy removal have been applied in the wrong object, since some synthesizers require these attributes to be placed in the ports to preserve, and others require them to be placed in the affected architecture. The ideal situation would be that the attributes that avoid redundancy removal could be applied to the hardened datatypes and inherited by all ports, signals and variables of that datatype.

Future work may also include implementing different hardening schemas by using the datatype substitution technique, such as hamming encoding for FSMs or approximate TMR.

5. Licensing

The `triple_logic` package is licensed under the GNU Lesser General Public License (LGPL) v3.0. The code can be downloaded from the website http://ftu.us.es/triplelogic .

Author Contributions: Conceptualization, H.G.-M.; methodology, H.G.-M., M.M.-Q. and D.V.-C.; validation and bug fixing: M.M.-Q., S.S.-B. and H.G.-M., formal analysis (equivalence checking), H.G.-M.; simulation, M.M.-Q., S.S.-B. and H.G.-M.; fault injection, M.M.-Q., D.V.-C.; analysis, M.M.-Q., D.V.-C.; investigation, M.M.-Q., S.S.-B., D.V.-C. and H.G.-M.; supervision, H.G.-M.; project administration, H.G.-M.

Funding: This work was supported by the Spanish Ministerio de Economía y Competitividad, through the project "Diseño de sistemas digitales robustos frente a radiación mediante componentes y tecnologías comerciales" (RENASER3), project reference ESP2015-68245-C4-2-P. This work was also partly supported by the European Commission, through the project "VEGAS: Validation of European high capacity rad-hard FPGA and software tools", project ID 687220.

Acknowledgments: The authors would like to thank: Clifford Wolf from Symbiotic EDA for kindly providing a VHDL-capable version of the Yosys Open SYnthesis Suite. Edouard Lepape, Hervé Baier and Mohamed Gountaf from NanoXplore for kindly providing a version of NanoXmap and promptly answering our support queries. Xilinx University Program (XUP) for kindly providing the ISE software. European Space Agency for funding the development of the FT-Unshades2 Fault Injection Platform (ESA contract 4200022981/09/NL/JK), which has been used to validate the results of this work. Finally, the authors would like to thank José M. Hinojo from Universidad de Sevilla for his support with multiple tool setup and licensing issues, and Luis Sanz for his advice on automating the generation of the different implementations of the test designs and meaningful conversations on how to better structure the operation overload capabilities of the package.

Conflicts of Interest: The authors declare no conflict of interest. The founding sponsors had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

ASIC	Application-Specific Integrated Circuit
AVF	Architectural Vulnerability Factor
DCE	Domain Crossing Errors
DD	Displacement Damage
DMR	Dual Modular Redundancy
DWC	Duplicate With Compare
EDAC	Error Detection And Correction
EDIF	Electronic Design Interchange Format
FF	Flip-flop
FIFO	First In, First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FT-Unshades	Fault Tolerance-Universidad de Sevilla Hardware Debugging System
HDL	Hardware Description Language
ISE	Integrated Synthesis Environment
ISim	ISE Simulator
LUT	Lookup Table
MBU	Multiple Bit Upset
RHBD	Rad-Hard By Design
RTL	Register-Transfer Level
SET	Single Event Transient
SEU	Single Event Upset
TID	Total Ionizing Dose
TMR	Triple Modular Redundancy
VHDL	Very High Speed Integrated Circuit Hardware Description Language
XST	Xilinx Synthesis Technology
Yosys	Yosys Open SYntesis Suite

References

1. Duzellier, S. Radiation effects on electronic devices in Space. *Aerosp. Sci. Technol.* **2005**, *9*, 93–99. [CrossRef]
2. Dominik, L. System mitigation techniques for single event effects. In Proceedings of the 2008 IEEE/AIAA 27th Digital Avionics Systems Conference, St. Paul, MN, USA, 26–30 October 2008; pp. 5.C.2-1–5.C.2-12.
3. Kuwahara, T.; Tomioka, Y.; Fukuda, K.; Sugimura, N.; Sakamoto, Y. Radiation effect mitigation methods for electronic systems. In Proceedings of the 2012 IEEE/SICE International Symposium on System Integration (SII), Fukuoka, Japan, 16–18 December 2012; pp. 307–312.
4. Lima-Kastensmidt, F.; Reis, R.A. *Fault-Tolerance Techniques for SRAM-Based FPGAs*, 1st ed.; Springer: Heidelberg, Germany, 2006.
5. Gomes, I.A.C.; Kastensmidt, F.G.L. Reducing TMR overhead by combining approximate circuit, transistor topology and input permutation approaches. In Proceedings of the 2013 26th Symposium on Integrated Circuits and Systems Design (SBCCI), Curitiba, Brazil, 2–6 September 2013; pp. 1–6.
6. Nair, A.A.; Eyerman, S.; Eeckhout, L.; John, L.K. A first-order mechanistic model for architectural vulnerability factor. In Proceedings of the 2012 39th Annual International Symposium on Computer Architecture (ISCA), Portland, OR, USA, 9–13 June 2012; pp. 273–284.
7. Martinez-Alvarez, A.; Cuenca-Asensi, S.; Restrepo-Calle, F.; Pinto, F.R.P.; Guzman-Miranda, H.; Aguirre, M.A. Compiler-Directed Soft Error Mitigation for Embedded Systems. *IEEE Trans. Dependable Secur. Comput.* **2012**, *9*, 159–172. [CrossRef]
8. Guzmán-Miranda, H.; Barrientos-Rojas, J.; López-González, P.; Baena-Lecuyer, V.; Aguirre, M.A. On the Structural Robustness Assessment of Wireless Communication Systems for Intra-Satellite Applications. *IEEE Trans. Nucl. Sci.* **2014**, *61*, 3244–3249. [CrossRef]
9. Synopsys. *FPGA Synthesis Attribute Reference Manual*; Synopsys: Mountain View, CA, USA, 2018.

10. Precision Hi-Rel Synthesis Software. Available online: http://www.mentor.com/products/fpga/synthesis (accessed on 20 December 2018).
11. Xilinx. TMRTool User Guide. Available online: https://www.xilinx.com/support/documentation/user_guides/ug156-tmrtool.pdf (accessed on 20 December 2018).
12. BYU EDIF Tools Home Page. Available online: http://reliability.ee.byu.edu/edif/ (accessed on 20 December 2018).
13. Quinn, H.; Morgan, K.; Graham, P.; Krone, J.; Caffrey, M.; Lundgreen, K. Domain Crossing Errors: Limitations on Single Device Triple-Modular Redundancy Circuits in Xilinx FPGAs. *IEEE Trans. Nucl. Sci.* **2007**, *54*, 2037–2043. [CrossRef]
14. Clark, L.T.; Patterson, D.W.; Ramamurthy, C.; Holbert, K.E. An Embedded Microprocessor Radiation Hardened by Microarchitecture and Circuits. *IEEE Trans. Comput.* **2016**, *65*, 382–395. [CrossRef]
15. Hindman, N.D.; Clark, L.T.; Patterson, D.W.; Holbert, K.E. Fully Automated, Testable Design of Fine-Grained Triple Mode Redundant Logic. *IEEE Trans. Nucl. Sci.* **2011**, *58*, 3046–3052. [CrossRef]
16. Kastensmidt, F.L.; Filho, C.K.; Carro, L. Improving Reliability of SRAM-Based FPGAs by Inserting Redundant Routing. *IEEE Trans. Nucl. Sci.* **2006**, *53*, 2060–2068. [CrossRef]
17. Sterpone, L.; Violante, M. A new reliability-oriented place and route algorithm for SRAM-based FPGAs. *IEEE Trans. Comput.* **2006**, *55*, 732–744. [CrossRef]
18. Nazar, G.L.; Carro, L. Fast error detection through efficient use of hardwired resources in FPGAs. In Proceedings of the 2012 17th IEEE European Test Symposium (ETS), Annecy, France, 28–31 May 2012; pp. 1–6.
19. IEEE Standard VHDL Language Reference Manual. In *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*; IEEE Computer Society: New York, NY, USA, 2009; pp. 1–626.
20. Mogollon, J.M.; Guzmán-Miranda, H.; Nápoles, J.; Barrientos, J.; Aguirre, M.A. FTUNSHADES2: A novel platform for early evaluation of robustness against SEE. In Proceedings of the 2011 12th European Conference on Radiation and Its Effects on Components and Systems, Sevilla, Spain, 19–23 September 2011; pp. 169–174.
21. Clifford Wolf. Yosys Open SYnthesis Suite. Available online: http://www.clifford.at/yosys/ (accessed on 20 December 2018).
22. VHDL Standard FIFO. Available online: http://www.deathbylogic.com/2013/07/vhdl-standard-fifo/ (accessed on 20 December 2018).
23. VHDL Implementation of FFT Algorithm(s). Available online: https://github.com/thasti/fft (accessed on 20 December 2018).
24. FPGA4student. A Low Pass FIR Filter for ECG Denoising in VHDL. Available online: https://www.fpga4student.com/2017/01/a-low-pass-fir-filter-in-vhdl.html (accessed on 20 December 2018).
25. I^2S Interface Designed for the PCM3168 Audio Interface from Texas Instruments. Available online: https://github.com/wklimann/PCM3168 (accessed on 20 December 2018).
26. Dalton Project, Department of Computer Science, University of California, "Synthesizable VHDL Model of 8051". Available online: http://www.cs.ucr.edu/~dalton/i8051/i8051syn/ (accessed on 20 December 2018).
27. Leveugle, R.; Calvez, A.; Maistri, P.; Vanhauwaert, P. Statistical Fault Injection: Quantified Error and Confidence. In Proceedings of the Design, Automation and Test in Europe Conference, Nice, France, 20–24 April 2009.
28. Xilinx. XPower Analyzer. Available online: https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/xpa_c_overview.htm (accessed on 20 December 2018).