DYNAMIC INFORMATION AND EXQUISITE PROCESSOR USING FPGAs AND XILINX

[1]C. Indupriya & [2]R. Senthamil Selvan

[1]II Year M.Tech, Department of Electronics and Communication Engineering, Chadalawada Ramanamma Engineering College, Tirupati, Andhra Pradesh.

[2]Assistant Professor, Department of Electronics and Communication Engineering, Chadalawada Ramanamma Engineering College, Tirupati, AndhraPradesh.

[1]indupriya.tpt@gmail.com

[2]senthamilselvan87@gmail.com

ABSTRACT

Soft-processors implemented on SRAM-based FPGAs are increasingly being adopted in on-board computing for space and avionics applications due to their flexibility and ease of integration. However, efficient component-level protection techniques for these processors against radiation-induced upsets are necessary otherwise as system failures could manifest. A register file is one of the critical structures that stores vital information the processor uses related to user computations and program execution. In this paper, we present a fault tolerance technique for the register file of a microprocessor implemented in Xilinx SRAM-based FPGAs. The proposed scheme leverages the inherent implementation redundancy created by the FPGA design automation tools when mapping the register file to on-chip distributed memory. A parity-based error detection and switching logic are added for fault masking against single-bit errors. The proposed scheme has been implemented and evaluated in low RISC, a RISC-V ISA soft-processor implementation. The effectiveness of the proposed scheme was tested using fault injection. The fault masking overhead required in terms of FPGA resources was much lower than a traditional Triple Modular Redundancy protection. Therefore, the proposed scheme is an interesting option to protect the register file of soft processors that are implemented in Xilinx FPGAs. Field programmable gate arrays (FPGAs) provide designers with the ability to quickly create hardware circuits. Increases in FPGA configurable logic capacity and decreasing FPGA costs have enabled designers to more readily incorporate FPGAs in their designs. FPGA vendors have begun providing configurable soft processor cores that can be synthesized onto their FPGA products. While FPGAs with soft processor cores provide designers with increased flexibility, such processors typically have degraded performance and energy consumption compared to

hard-core processors. In this paper, we study the implementation of soft-core processors in FPGAs, and some of the decisions and design tradeoffs which must be made during the design process. Triple-modular redundancy (TMR), which consists of three identical modules and a voting circuit, is a common architecture for soft-error tolerance. However, the original TMR suffers from two major drawbacks: the large area overhead and the vulnerability of the voter. In order to overcome these drawbacks, we propose a new complementary dual-modular redundancy (CDMR) scheme for mitigating the effect of soft errors. Inspired by the Markov random field (MRF) theory, a two-stage voting system is implemented in CDMR, including a first stage optimal MRF structure and a second-stage high-performance merging unit. The CDMR scheme can reduce the voting circuit area while saving the area of one redundant module. The design is done in Xilinx ISE tool.

Keywords: FPGAs (Field programmable gate arrays), CDMR (complementary dual-modular redundancy), MRF (Markov random field), TMR (Triple-modular redundancy), Xilinx ISE,SRAM (Static Random Access Memory)

I.INTRODUCTION

The successful deployment of satellite missions based on commercial SRAM-based FPGAs has not only shown the feasibility of their usage but also the sensitivity-levels of the reconfigurable fabric resources to radiation effects [1][2]. Therefore, motivating a resource-level analysis of radiation effects and its consequences along with design-mitigation approaches at a modular-level to enhance system-level reliability. This bottom-up approach to reliable electronic systems design needs to consider the heterogeneous composability of the state-of-theart reconfigurable fabrics and the mapping of computational structures to it by the design automation tools. Applying such bottom-up design and test approach to soft-processors implemented on SRAM-based FPGAs is an interesting research direction [3]. In particular, combined with the recent trends towards a standard open Instruction Set Architectures (ISA) [4], reliability in soft processors could be investigated and mitigated from the circuit and system level to architectural level and from operating system (OS) to application layers. Reliability techniques and re-configurability approaches could be integrated along the way to improve the fault-tolerance, fault diagnosis and recovery aspects.

II.CONVENTIONAL SYSTEM

Protection of the Register File in Soft-processors

Our proposed methodology leverages the inferred architectural redundancy by the FPGA design automation tools when mapping a processor register file [5]. This existing redundancy is augmented with an efficient error detection and error masking method to provide TMR like capabilities at considerably low area and delay overhead. The following sections illustrates our proposed method in details.

Read-port Address Switching Based Fault Masking

As stated before, the Xilinx design tool implements a dual port register file as two separate physical copies, RD1 and RD2, each one containing all the register values Although the presented technique will be explained in depth in the next sections, a general overview is presented in this subsection so that the readers can get the top level approach in advance. The strategy of the presented technique is as follows. If an instruction reads two operand registers and one of the parity checks is wrong, that means that the register associated to that parity check has suffered an error. If, after this, the two operands are swapped in the read lines, then their actual values would be retrieved from the opposite register copy, thus avoid ing the instance that has suffered the error. Since we are assuming a single error mechanism, there will always be at least one valid copy of every register in the system [6]. An example of this technique is shown in Fig. 1. On the left, register r2 in RD1 (whose initial correct value was 16) has been affected by an error in its least significant bit, changing its content to 17. A read operation of the register reflects this (using a parity check), but its second copy in RD2 is error-free. An error detection logic is added to detect such an error and control the switching of read address lines. In the example, before the switching, r2 was read from the first copy RD1, but after switching r2 is now read from the error-free second copy RD2, thus avoiding the wrong value. Similarly, notice that r10 was read before from RD2, but it is swapped to be read from RD1. In this case the output of r10 is still valid, since none of its copies has suffered any error. With this technique, no matters where the error affects, the system is able to swap the read lines and get a good copy of both registers. Notice that with this procedure faults are masked, although not corrected [7]. This is not a problem because the register file is frequently updated and the corrupted content would be replaced by another value. The implementation of this scheme in Xilinx FPGAs and the error detection and switching logic is described in the following subsections.

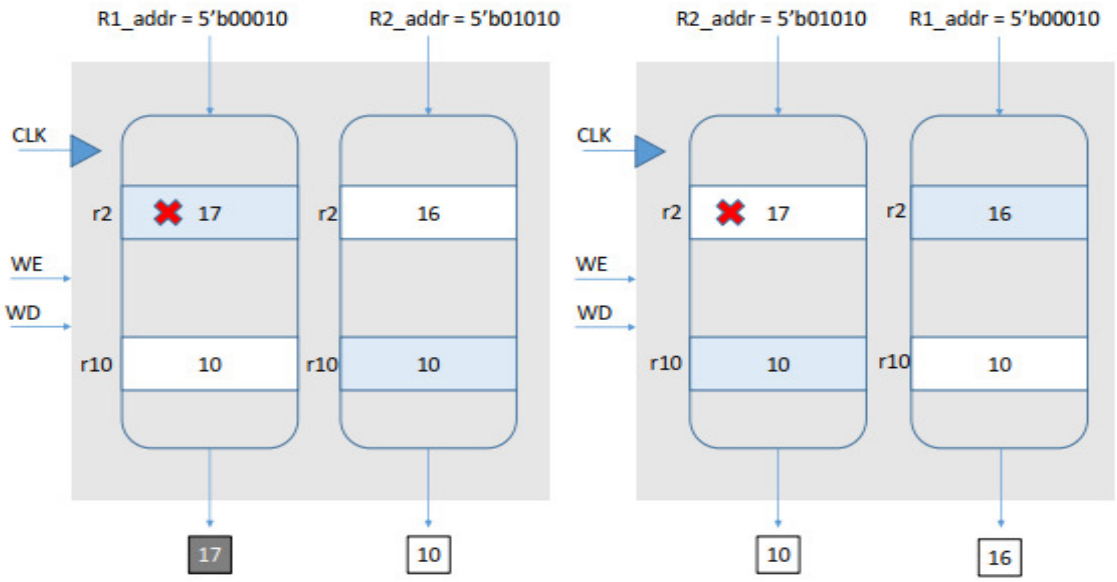


Fig.2.0.Read-port Address Switching Based Fault Masking

Register File Implementation in Xilinx FPGAs Register files are an integral part of any processor architecture and are vital to provide support for instructions operating on multiple registers simultaneously. Usually, most operations require two registers as inputs. Therefore, simultaneous readings of two values from the register file is an important consideration. Since memories available inside SRAM-based FPGAs are designed with one read port, design automation tools replicate the same data in more than one instance of the architectural memory primitives to support multi-read ports [8,9].

In Fig. 2, the Logical view, illustrates the scheme for a dual (read) port register file that has a word width of 64 bits and depth of 32 registers. This view shows the content organization with respect to individual registers arranged horizontally in rows and then divided into equal size chunks vertically. Each chunk is housed by a fundamental block annotated in the logical view. Consequently, each fundamental building block acts as a smaller register file of 32 registers (r0, r1, …, r31 in Fig. 2, Logical view) with word width of 6 bits (3 LUTs each having 2 bits outputs). Therefore, the entire register file contents are divided into equal-size chunks of 6 bits denoted as C1, C2… C11 in Fig. 2 and stored across multiple fundamental blocks. The physical view in Fig. 2 illustrates that the two instances of the FPGA memory are used (COPY A and COPY B), each one connected to different read address lines (RA1 and RA2 respectively) but the same write port. It has to be mentioned that the write operation is synchronous but the read operation is asynchronous. Furthermore, the Block 1 in the Logical View is mapped

to the FPGA as Block 1 for COPY A and Block 1' for COPY B. This process is repeated for Block 2 and onwards until reaching the Block 11, which is mapped in the same way but with less resources. This mapping results in eleven (for COPY A and for COPY B) such memories to be used to compose the entire register file [10].

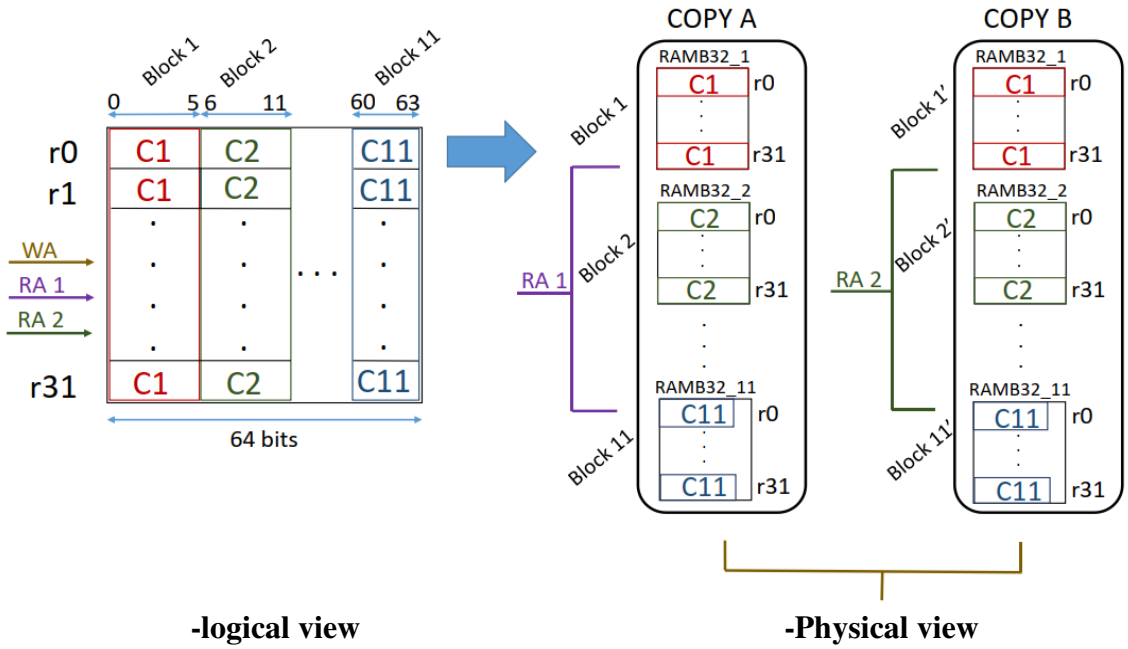
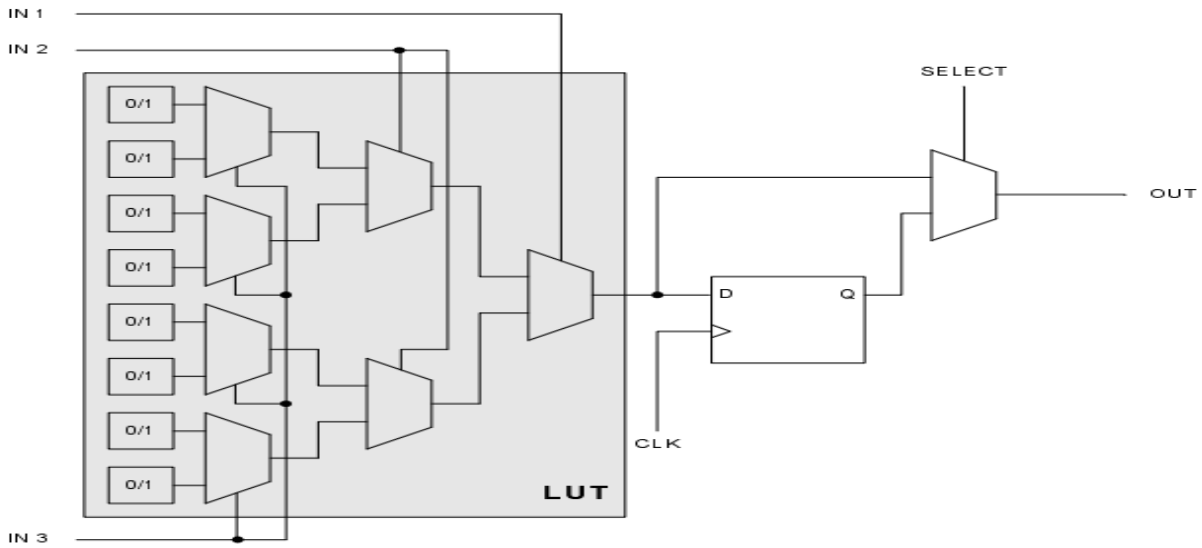


Fig.2.1 Implementation of a register file in Xilinx FPGAs

III.FPGA PROGRAM FLOW

FPGA technology FPGA devices are programmable to implement arbitrary user logic. To support this programmability, FPGA devices contain three types of resources: logic blocks, I/O blocks, and programmable interconnection. Most FPGAs contain logic blocks that consist of a lookup table (LUT), and a flip-flop. A LUT can be programmed to implement any logic function of its inputs. A LUT with n inputs is called an n-LUT. An n-LUT is internally implemented as a set of 2-to-1 multiplexers, functioning as a 2 n -to-1 multiplexer. Multiplexer inputs are programmable, while select lines are used for inputs of the implemented function [11]. Research has shown that 4-LUTs are an optimal choice for FPGAs [11]. Figure 2.1 shows an example of a logic block consisting of a 3-LUT, and a flip-flop. An 8-to1 multiplexer in a LUT is implemented using 2-to-1 multiplexers [12]. Therefore, the propagation delay from inputs to the output is not the same for all the inputs. Input IN 1 experiences the shortest propagation delay, because the signal passes through fewer multiplexers than signals IN 2 and IN 3. Since a LUT can implement any function of its input variables, inputs to the LUTs should be mapped in such a

way that the signals on a critical path pass through as few multiplexers as possible. Logic blocks also include a flip-flop to allow the implementation of sequential logic. An additional multiplexer is used to select between the LUT and the flip-flop output. Logic blocks in modern FPGAs [12] are usually more complex than the one presented here [13].



Simple logic block structure

Each logic block can implement only small functions of several variables. Programmable interconnection, also called routing, is used to connect logic blocks into larger circuits performing the required functionality [20]. Routing consists of wires that span one or more logic blocks. Connections between logic blocks and routing, I/O blocks and routing, and among wires themselves is programmable, which allows for the flexibility of circuit implementation. Routing is a very important aspect of FPGA devices, because it dominates the chip area and most of the circuit delay is due to the routing delays [11]. I/O blocks in an FPGA connect the internal logic to the outside pins. Depending on an actual device, most pins can be configured as input, output, or bidirectional. Devices supporting more than one I/O standard allow configuration of different pins for different standards [12]. Programmability of FPGAs is commonly achieved using one of three technologies: SRAM cells, antifuses, and floating gate devices. Most devices use SRAM cells. The SRAM cells drive pass transistors, multiplexers, and tri-state buffers, which in turn control the configurable routing logic and I/O blocks [14]. Since the content of SRAM cells is lost when the device is not powered, the configuration needs to be reloaded into the device on each power-up. This is done using a configuration device that loads the configuration stored in some form of non-volatile memory. Programmability of FPGAs comes at a price. Resources

necessary for the programmability take up chip area and consume power. Designing a complex system targeting FPGAs would be virtually impossible without CAD tools.

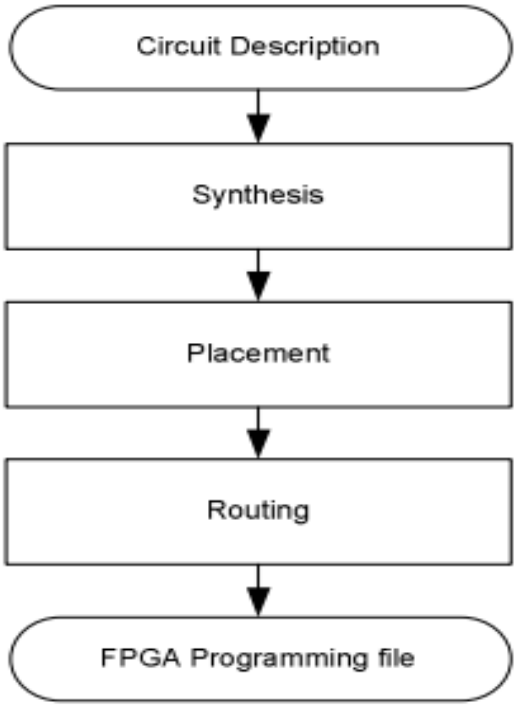


Fig.3.4 FPGA Program Flow

The placement algorithm maps logic blocks from the netlist to physical locations on an FPGA. If the clustering was performed during the synthesis step, clustered LUTs are mapped to physical clusters. Logic block placement directly influences the amount of routing resources required to implement the circuit [15-18]. A placement configuration that requires more routing resources than is available in the corresponding portion of the device cannot be routed. Hence, the circuit cannot be implemented in an FPGA with that placement configuration, and a better placement must be found, if one exists. Since the number of possible placements is large, metaheuristic algorithms [19-20] are used.

IV.PROPOSED SYSTEM

Proposed Protection Technique

The proposed register file protection technique is presented in Fig. 2.1. It can be noted that the register file implementation is modified to include a parity computation, storage and comparison module along with several multiplexers at the input and output of the register file. The parity module takes the advantage of the spare bits in the register file implementation for storage of the parity bits.

The proposed scheme is very efficient because all the LUTs inside the fundamental memory block are connected to the read/write ports. Therefore, the existing interconnections are utilized for synchronous parity computation, storage and retrieval with limited overhead. It can be noted from Fig. 3 that at the writing phase in the register file, a XOR-reduction is performed by X1 on the written word and a parity bit at a corresponding location is saved in the parity memory. Although, for sake of illustration, the parity memory in Fig. 3 is shown separately, it is implemented inside the same FPGA mapped register file memory structure. Therefore, the parity memory is not adding any resource overhead due to the exploitation of unused bits in the FPGA mapped register (spare bits are utilized) as discussed in the previous subsection). The error detection and masking logic is designed to deal with two distinct cases, i.e. when an instruction is using two separate operand registers (like ADD R1,R2,R3) and when an instruction is using the same register e.g. AND R1, R2, R2 which is a common method to implement a MOV instruction in most RISC processors). In the first case, at the reading stage, the parity is calculated for the output words by X2 and X3, and compared with the stored parity bit. In case of a mismatch, a parity-check flag is generated by O1. This parity check-flag is further sampled by the negative edge triggered flip-flop FF2, inserted to break the combinational feedback cycle necessary to avoid multiple writing of the register contents. This flip-flop is followed by two multiplexers M5 and M6. These multiplexers are inserted to deal with the aforementioned two cases. In the first case (different registers), the same_signal will be logic 0 leading to the selection of FF2 output while in the second case (same register), the same_signal will be logic 1 resulting in selection of outputs from FF1 and FF3. Note that these flip-flops are directly sampling the inequality operators which means that these flip-flops reflect which of the copy is faulty to subsequently switch address and output lines for the faulty one only using the multiplexers at the inputs (M1 and M2) and outputs (M3 and M4) of the memory.

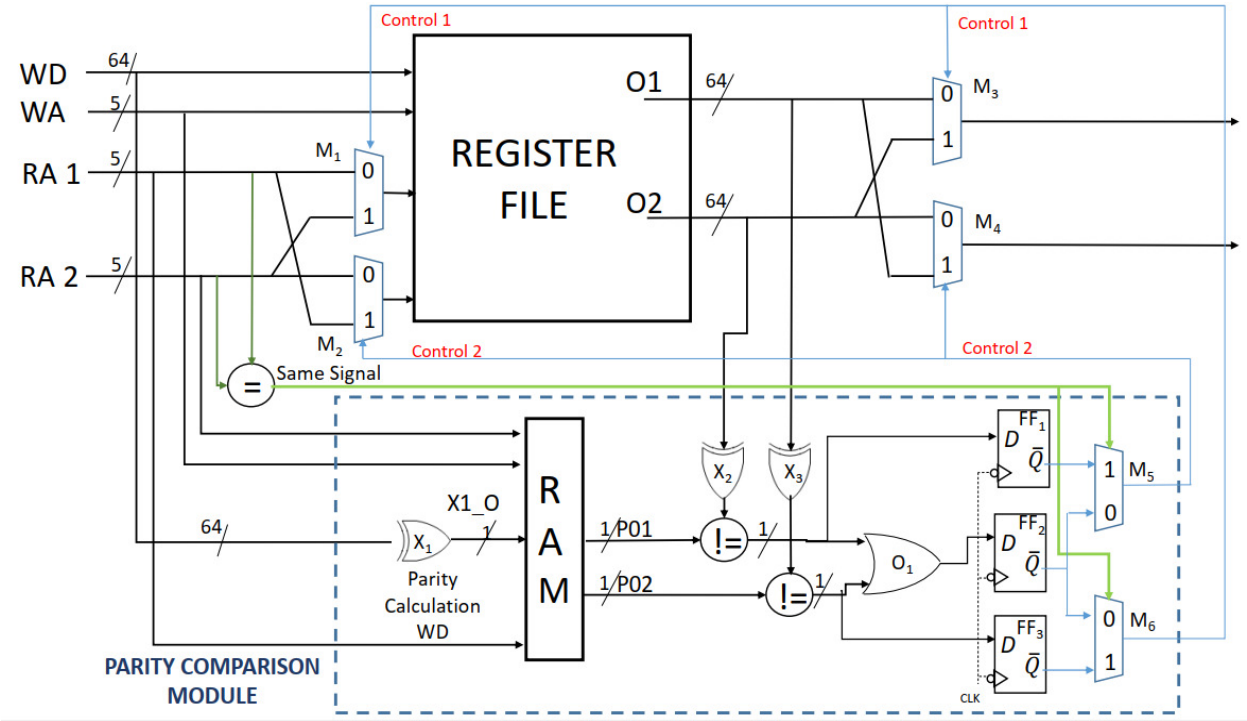


Fig.4.0.Proposed Error masking method

V.RESULTS AND DISCUSSION

The specifications we are going to see may be different for different foundry and technology.

- Design Example (3 Levels): NOR Gate
- Logic Design
- Circuit Design
- Layout Design

Microwind / DSCH NOR Example: NOR Gate Logic

- Open the Schematic Editor in Microwind (DSCH3). Click on the transistor symbol in the symbol Library on the right.
- Instantiate NMOS or PMOS transistors from the symbol library and place them in the editor window.
- Instantiate 2 NMOS and 2 PMOS transistors.
- Connect the drains and sources of transistors.
- Connect Vdd and GND to the schematic.
- Connect input button and output LED.

Now we have NOR schematic ready.

- Use your logic simulator to verify the functionality of your schematic.
- The next step is to simulate the circuit and check for functionality.
- Click on, Simulate -> Start simulation.
- This brings up a Simulation Control Window.
- Click on the input buttons to set them to 1 or 0. Red color in a switch indicates a '1'. As shown,

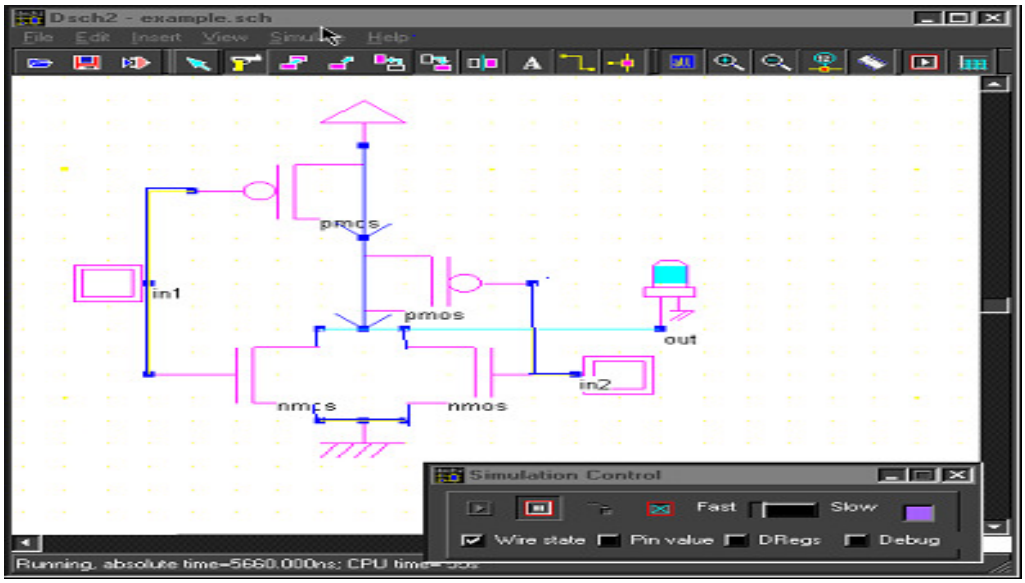


Fig.5.0. NOR schematic

The simulation output can be observed as a waveform after the application of the inputs as above. Click on the timing diagram icon in the icon menu to see the timing diagram of the input and output waveforms.

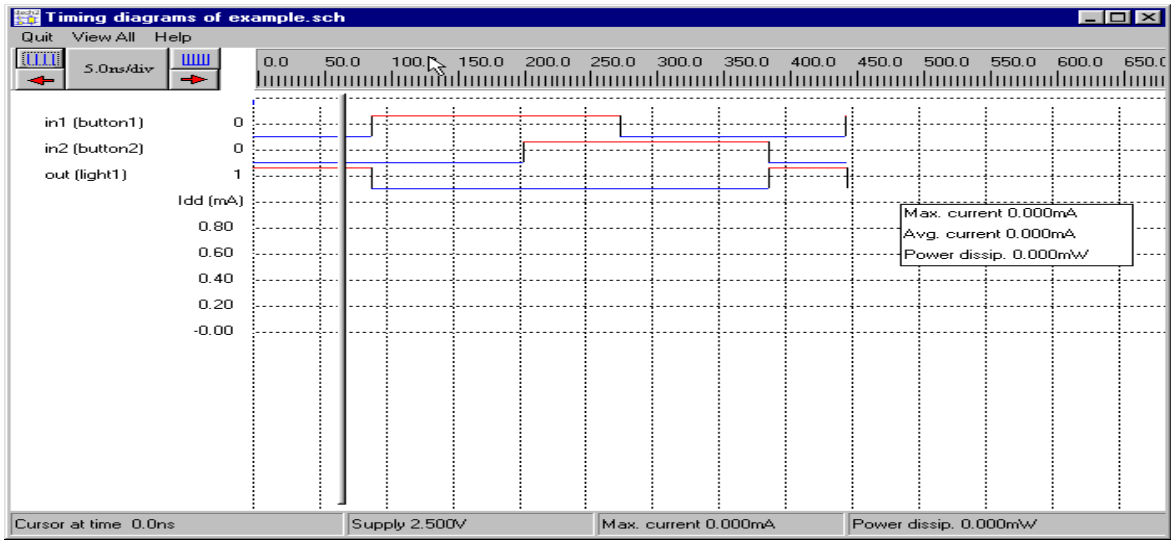


Fig.5.1. Timing Diagram

Simulate your system with your hand calculated transistor sizes.

- Click File -> Make Verilog File. The Verilog, Hierarchy and Netlist window appears. This window shows the verilog representation of NOR gate.
- Click OK to save the Verilog as a .txt file.

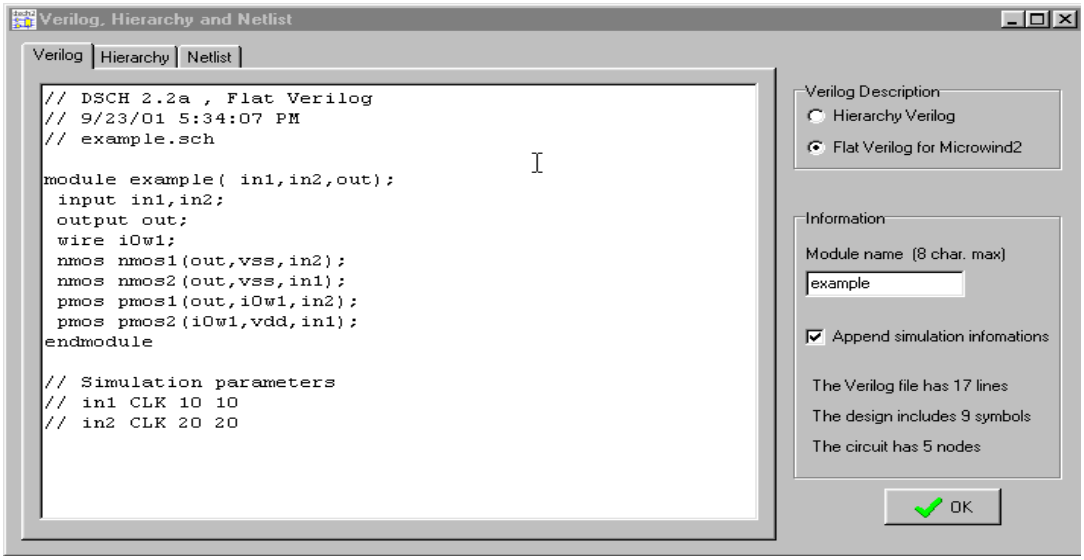


Fig.5.2.Verilog Text File

Microwind / DSCH NOR Example: Circuit Design

- Open the layout editor window in Microwind. Click *File -> Select Foundry* and select *X.rul*.This sets your layout designs in X technology.

- Click on *Compile -> Compile Verilog File.* An *Open* Window appears Select the .txt verilog file saved before and open it.

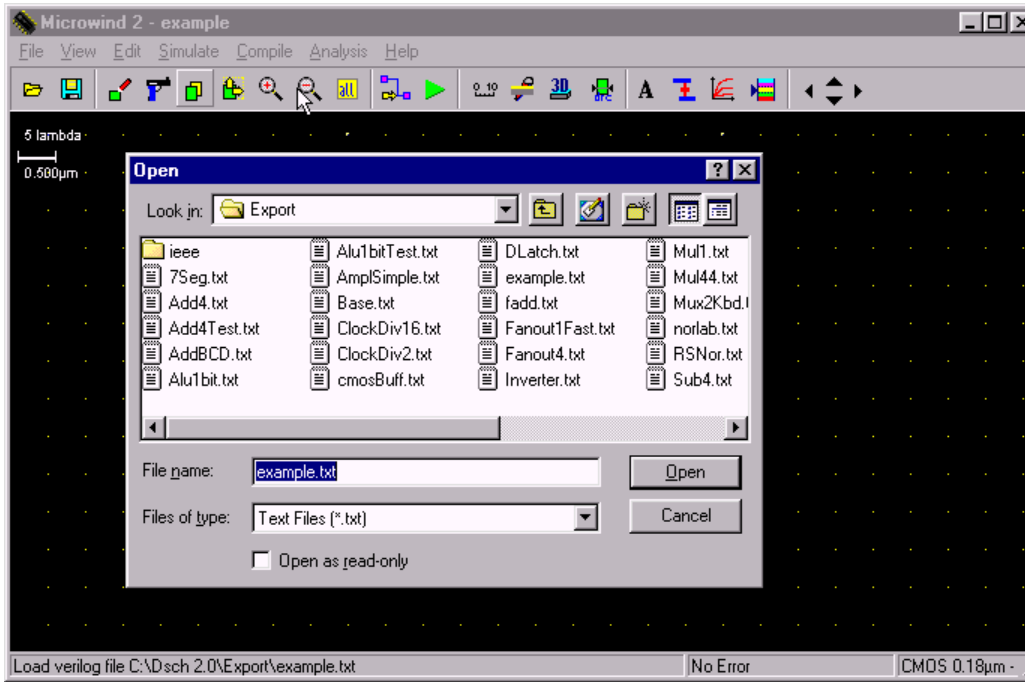


Fig 5.3.Verilog File Saving Method

After selecting the .txt file, a new window appears called Verilog file.Click on Size on the right top menus. This shows up the NMOS and PMOS sizes. Set the sizes according to choice.

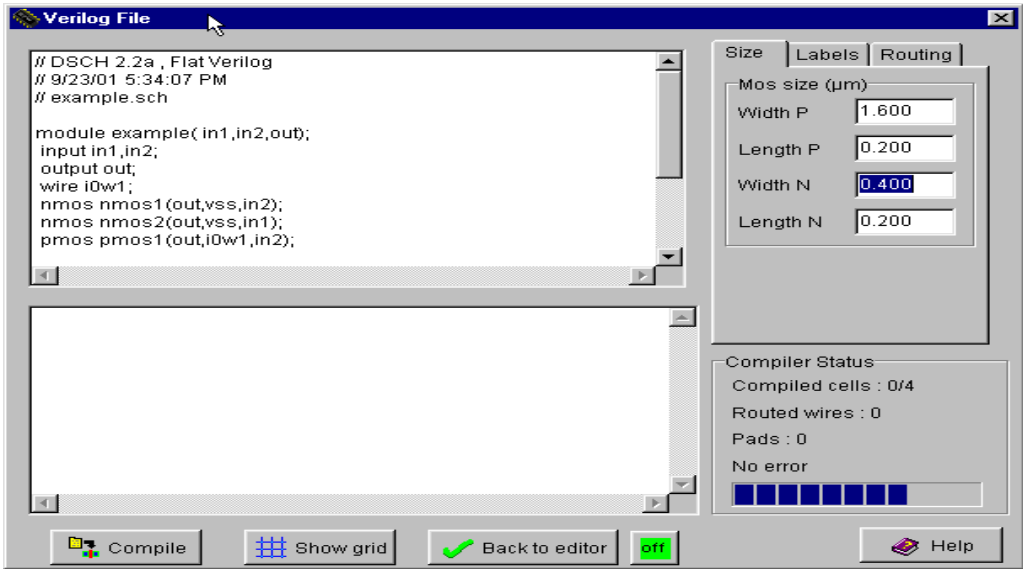


Fig.5.4.Size of NMOS and PMOS

- Click Compile and then Back to editor in the Verilog File Window. This creates a layout in layout editor window using automatic layout generation procedure.
- Add a capacitance to the output of the design. The value of the capacitance depends on your choice.
- Click on OK. The capacitance is shown on the left bottom corner with a value of 0.015fF.

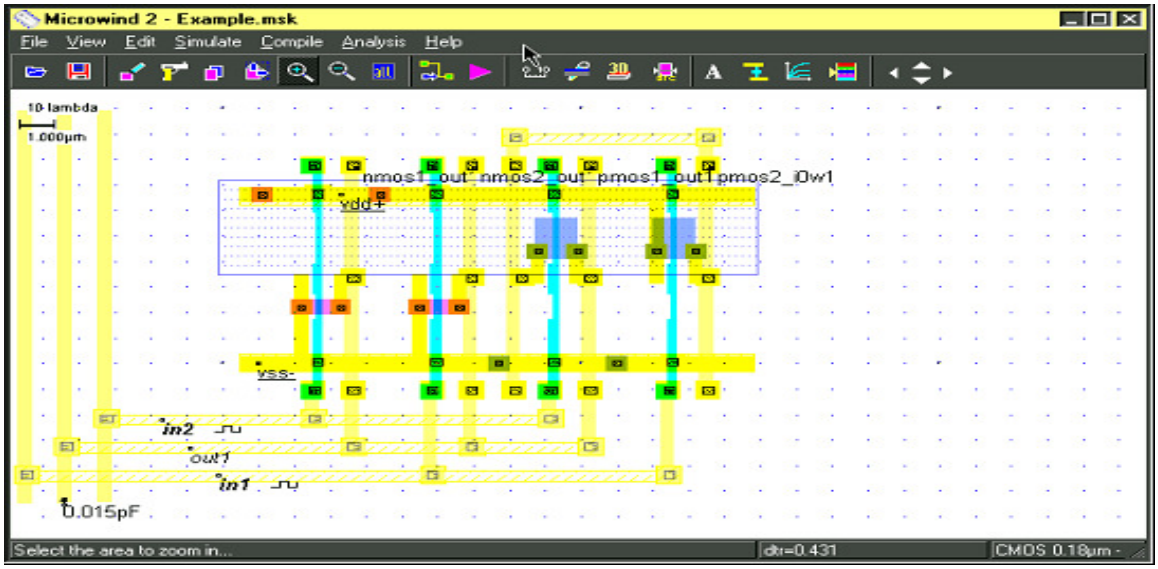


Fig.5.5. Design of Output Layout

- Click Simulate -> Run simulation. A simulation window appears with inputs and output, shows the tphl, tplh and tp of the circuit. The power consumption is also shown on the right bottom portion of the window.
- If you are unable to meet the specifications of the circuit change the transistor sizes. Generate the layout again and run the simulations till you achieve your target delays.

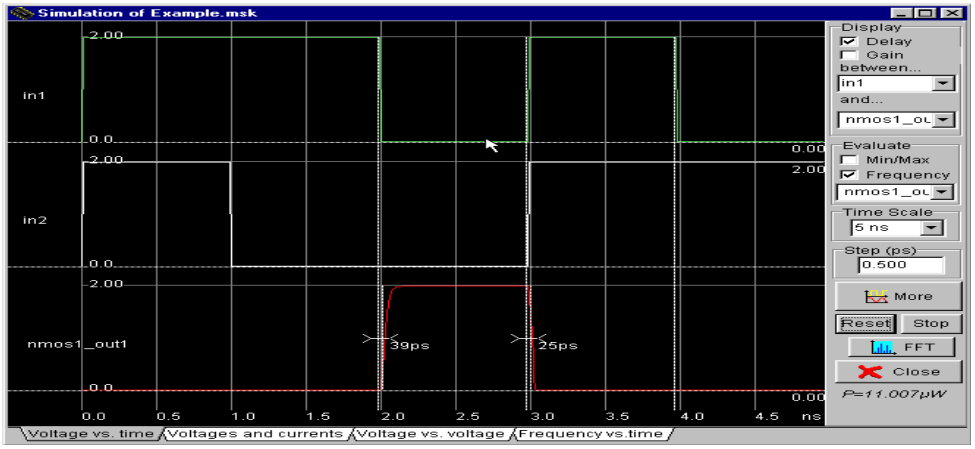


Fig.5.6.Generation of Delay sequence

Microwind / DSCH NOR Example: Layout Design

- Design the layout manually. Open the layout editor window in Microwind. Click File -> Select Foundry and select X.rul, Vdd and GND rails are of Metal1. The top rail is used as Vdd and the bottom one as GND. Click on Metal 1 in the palette and then creates the required rectangle in the layout window.
- The next step is to build the NMOS transistors. Click on the transistor symbol in the palette. Set the W, L of the transistor
- Then click on Generate device. The source of the transistor is connected to the GND rail. Create another NMOS and place it in parallel to the first NMOS device. We share the two devices' drain diffusions. A DRC check can be run by clicking on Analysis -> Design Rule Checker.
- The next step is to place two PMOS transistors in series. Place the PMOs transistor on layout close to the Vdd rail on the top. To construct two PMOS transistors in series, diffusions are shifted to a side and another poly line is added as second transistor. The diffusion is shared to save area and reduce capacitance.

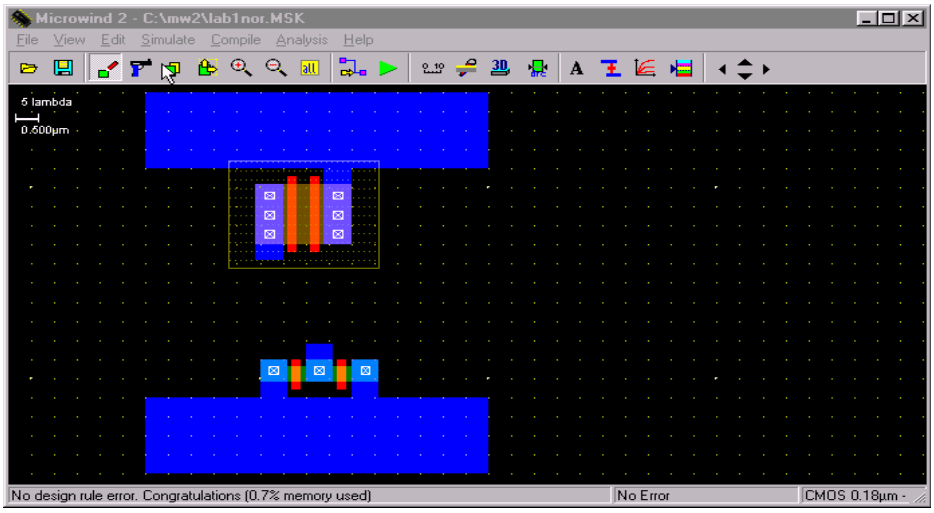


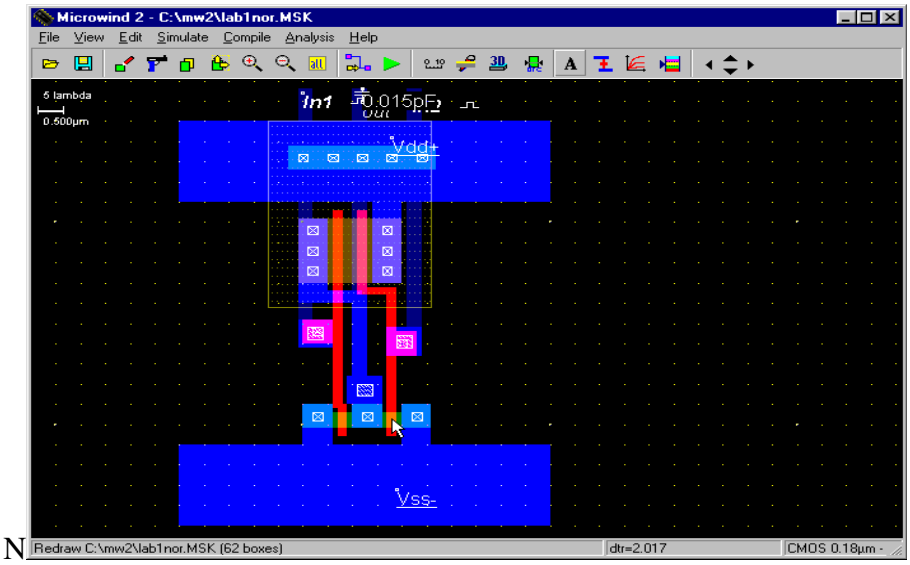
Fig.5.7. Rule Checker

- The next step is to connect the inputs and the output of the two transistors. Poly inputs is connected. Metal output is connected.
- The next step is to connect the poly to metal1 and then to metal2. The first symbol in the first row of the palette is the poly to metal1 contact.

- Then we connect the metal1 to metal2 contact to the previous contact. This is the 4th contact on the first row
- The next step is to connect the output Metal1 to Metal2. Once again use the 4th contact in the first row.
- Now we connect metal2 to the two inputs and one output and bring them to the top to go out of the cell. Observe the two inputs (left & right) and an output (middle) above the Vdd rail in dark blue color.

Now we label the inputs and output as In1, In2 and out. Click on Add a Pulse Symbol in the palette (5th from the right in the 3rd row). Then click on the metal2 of one of the inputs. A window appears. Change the name of the input signal. Insert a 01 sequences and click on Insert. The click on Assign. Similarly assign the 2nd input a pulse.

Select the Visible Node symbol from the palette (7th in the third row). Select it and click on the output. The 'Add a Visible Property' window appears. Change the label name to out. Select Visible in Simulation. Click on Assign.



N

Fig.5.8.Generation of Contacts using Rule Checker

Select Vdd Supply and GND from the palette (third row). Also click on the capacitor (3rd in 2nd row) symbol and add it to the output. Also, extend the pwell into the Vdd Rail. The click on Edit -> Generate -> Contacts. Select PATH and then in Metal choose Metal1 and N+ polarization.

To run the Simulation of your circuit, click on Simulate -> Start Simulation. Depending on the input sequences assigned at the input the output is observed in the simulation. The power value is also given.

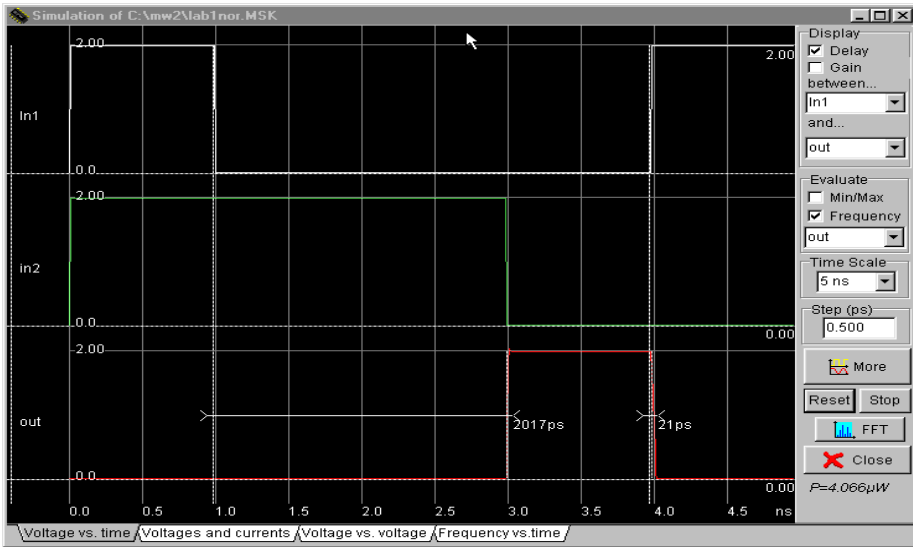


Fig.5.9.Generation of Power Signal

VI.CONCLUSION

This application presents an efficient technique to protect the register file of soft-processors implemented in Xilinx SRAM-based FPGAs. The method leverages the inferred redundancy incorporated by the design automation tools to come up with a low overhead approach to error detection and masking at considerably less cost than TMR. Our evaluations demonstrated that the delay introduced by the proposal is affordable in comparison to the processors clock period. The presented technique can be applied not only to register files in soft-processors but also to other similar dual-port memory structures implemented through distributed RAM resources on SRAM-based FPGAs.

The proposed scheme using parity based error detection is able to correct single bit errors. Future work will consider the protection against multiple bit errors by using the proposed technique with SEC or SEC-DED codes to perform error detection. This project presents the implementation of soft-core processors in FPGAs, and some of the decisions and design tradeoffs which must be made during the design process. Making informed decisions during the design process reduces the time required to design, implement, debug and test an FPGA soft

processor-based project. Important design factors are reviewed, common design oversights are discussed and softcores examples is presented. Soft processor design teams will benefit from a system-oriented design approach, which considers the long-term effects of design decisions at each design phase. With a solid understanding of the overall design cycle, development tools options, and benefits of key design trade-studies, the design team can avoid many common design mistakes and oversights resulting in a more efficient and flexible design cycle.

VII.REFERENCES

[1]. H. Quinn, D. Roussel-dupre, M. Caffrey and P. Graham, M. Wirthlin, K. Morgan, A. Salazar, T. Nelson, W. Howes, E. Johnson, J. Johnson, B. Pratt, N. Rollins, J. Krone, "The Cibola flight experiment", ACM Transactions on Reconfigurable Technology and Systems, Vol. 8, No. 1, Article 3, Feb 2015.

[2]. H. Quinn, P. Graham, K. Morgan, Z. Baker, M. Caffery, D. Smith, M. Wirthlin and R. Bell, "Flight experience of the Xilinx Virtex-4", IEEE Transactions on Nuclear Science Vol 60, Issue 4, pp 2682-2690.

[3]. N. A. Harward, M. R. Gardiner, L. W. Hsiao and M. J. Wirthlin., "Estimating soft processor soft error sensitivity through fault injection," in 2015 IEEE 23rd Annual International Symposium on Field Programmable Custom Computing Machines, May 2015, pp. 143– 150.

[4]. A. Waterman and others, The RISC-V Instruction Set Manual, may 2016.

[5]. Bairavasundaram, L. N., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., Goodson, G. R., and Schroeder, B., "An analysis of data corruption in the storage stack, "ACM Transactions on Storage, vol. 4, no. 3, pp. 8:1–8:28, Nov. 2008.

[6]. "7 series FPGAs memory resources (ver 1.12)," in Xilinx Document No. UG473, September" 2016. [Online]. Available: https://www.xilinx.com/

[7]. A. Manuzzato, S. Gerardin, A. Paccagnella, L. Sterpone, and M. Violante, "Effectiveness of TMR-based techniques to mitigate alphainduced SEU accumulation in commercial SRAM-based FPGAs," IEEE Transactions on Nuclear Science, vol. 55, no. 4, pp. 1968–1973, Aug 2008.

[8]. P. Reviriego, S. Pontarelli, J.A. Maestro, and M. Ottavi, "A method to construct low delay single error correction codes for protecting data bits only," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 32, no. 3, pp. 479–483, March 2013.

[9]. LowRISC. (2016, July) lowRISC trace debugger project. [Online].Available: http://www.lowrisc.org/docs/debug-v0.3/

[10]. R. Naseer, R. Z. Bhatti, and J. Draper, "Analysis of soft error mitigation techniques for register files in IBM cu-08 90nm technology," in 2006 49th IEEE International Midwest Symposium on Circuits and Systems, vol. 1, Aug 2006, pp. 515–519.

[11]. J. A. Blome, S. Gupta, S. Feng, and S. Mahlke, "Cost-efficient soft error protection for embedded microprocessors," in Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems. ACM, 2006, pp. 421–431.

[12]. G. Memik, M. T. Kandemir, and O. Ozturk, "Increasing register file Immunity to transient errors," in Design, Automation and Test in Europe, March 2005, pp. 586–591 Vol. 1.

[13]. H. Amrouch and J. Henkel, "Self-immunity technique to improve register file integrity against soft errors," in 2011 24th International Conference on VLSI Design, Jan 2011, pp. 189–194.

[14]. R. W. Hamming, "Error detecting and error correcting codes," The Bell System Technical Journal, vol. 29, no. 2, pp. 147–160, April 1950.

[15]. J. Yan and W. Zhang, "Compiler-guided register reliability improvement against soft errors," in Proceedings of the 5th ACM International Conference on Embedded Software, ser. EMSOFT '05. New York, NY, USA: ACM, 2005, pp. 203–209.

[16]. Reviriego, P., Demirci, M., Tabero, J., Regadío, A., and Maestro, J. A., "DMR +: An efficient alternative to TMR to protect registers in Xilinx FPGAs," Microelectronics Reliability, vol. 63, pp. 314 – 318, 2016.

[17]. P. Adell, G. Allen, G. Swift and S. McClure., "Assessing and mitigating radiation effects in Xilinx SRAM FPGAs," in 2008 European Conference on Radiation and Its Effects on Components and Systems, Sept 2008, pp. 418–424.

[18]. N. H. Rollins, "Hardware and software fault-tolerance of softcore processors implemented in SRAM-based FPGAs," Doctoral Dissertation, BYU 2012.

[19]. C. Hong, K. Benkrid, X. Iturbe and A. Ebrahim., "Design and implementation of fault-tolerant soft processors on FPGAs," in 22nd International Conference on Field Programmable Logic and Applications (FPL), Aug 2012, pp. 683–686.

[20]. "7 series FPGAs configurable logic block (ver 1.8)," in Xilinx Document No. UG474, Sep 2016, pp. 23–27.