ISSN XXXX XXXX © 2017 IJESC

Research Article Volume 7 Issue No.5

Self-Immunity Technique to Improve Register File Integrity against Soft Errors

P. Anitha[1], Tella kumari Padmalatha[2], K. Durgaprasad[3]
Department of Electronics and communication Engineering
Institute of Aeronautical Engineering, India

Abstract:
In the early days of computers, "glitches" were an accepted way of life. Since then, as computers have become more reliable (and more relied upon), glitches are no longer acceptable – yet they still occur. One of the most intractable sources of glitches has been the transient "bit-flip", or soft memory error: a random event that corrupts the value stored in a memory cell without damaging the cell itself. Soft errors* in electronic memory were first traced to alpha particle* emissions from chip packaging materials. Since then, memory manufacturers have eliminated most alpha particle sources from their materials, changed their designs to make them less susceptible (e.g., moved ball-grid bumps farther away from memory cells), and even added shielding (usually internal die coatings). Tests and standards have been developed to measure and improve the resistance of memory chips to alpha particles – but soft errors have not disappeared.

I. INTRODUCTION

Soft errors* in electronic memory were first traced to alpha particle* emissions from chip packaging materials. Since then, memory manufacturers have eliminated most alpha particle sources from their materials, changed their designs to make them less susceptible (e.g., moved ball-grid bumps farther away from memory cells), and even added shielding (usually internal die coatings). Tests and standards have been developed to measure and improve the resistance of memory chips to alpha particles – but soft errors have not disappeared. Further testing, mostly performed by avionics and space organizations, pinpointed a more pernicious source of soft errors: cosmic rays*. At ground level, cosmic radiation is about 95% neutrons and 5% protons. These particles can cause soft errors directly; they can also interact with atomic nuclei to produce troublesome short-range heavy ions. Cosmic rays cannot be eliminated at their source, and effective shielding would require meters of concrete or rock. To eliminate the soft memory errors that are induced by cosmic rays, memory manufacturers must either produce designs that can resist cosmic ray effects or else invent mechanisms to detect and correct the errors.

Register file soft errors

Register file (RF) is extremely vulnerable to soft errors, and traditional redundancy based schemes to protect the RF are prohibitive not only because RF is often in the timing critical path of the processor, but also since it is one of the hottest blocks on the chip, and therefore adding any extra circuitry to it is not desirable. Device scaling trends dramatically increase the susceptibility of microprocessors to soft errors. Further, mounting demand for embedded microprocessors in a wide array of safety critical applications, ranging from automobiles to pacemakers, compounds the importance of addressing the soft error problem. To bridge the gap, there is an aggravated need of techniques to increase the register file integrity against soft errors with a small effect on both area and power overhead. The paper addresses this challenge by introducing a novel technique, called *Self-Immunity* to improve the resiliency of register files to soft errors, especially desirable for processors that demand high register file integrity under stringent constraints.

Soft error background and terminology

1.1.1. MTBF and FIT

Vendors express an error budget at a reference altitude in terms of Mean Time Between Failures (MTBF). Errors are often further classified as undetected or detected. The former are typically referred to as *silent data corruption* (SDC); we call the latter *detected unrecoverable errors* (DUE). Note that detected recoverable errors are not errors. Forexample, for its Power4 processor-based systems, IBM targets 1000 years system MTBF for SDC errors, 25 years system MTBF for DUE errors that result in a system crash, and 10 years system MTBF for DUE errors that result in an application crash. Note that the processor MTBF must be significantly higher than the system MTBF, particularly for large multiprocessor systems. Another commonly used unit for error rates is FIT (Error in Time), which is inversely related to MTBF. One FIT specifies one failure in a billion hours. Thus, 1000 years MTBF equals 114 FIT (109 / (24×365×1000)).

A zero error rate corresponds to zero FIT and infinite MTBF. Designers usually work with FIT because FIT is additive, unlike MTBF. To evaluate whether a chip meets its soft error budget—possibly via the use of error protection and mitigation techniques—microprocessor designers use sophisticated computer models to compute the FIT rate for every device—RAM cells, latches, and logic gates—on the chip. The effective FIT rate for a structure is the product of its raw circuit FIT rate and the structure's *vulnerability factor*, i.e., an estimate of the probability that a circuit fault will result in an observable error.

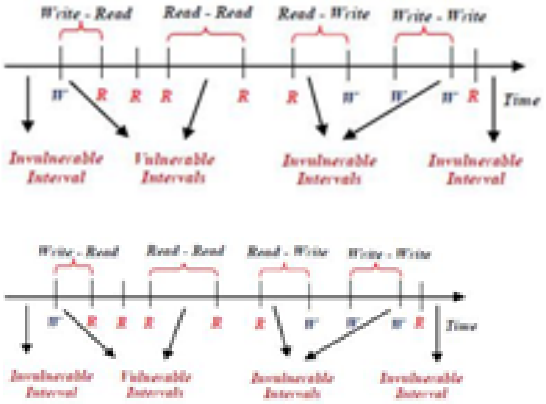
The overall FIT rate of the chip is calculated by summing the effective FIT rates of all the structures on the chip.

1.1.2Vulnerability Factors

The effective FIT rate per bit is influenced by several *vulnerability factors* (also known as *derating factors or soft error sensitivity factors*). In general, a vulnerability factor indicates the probability that an internal fault in a device's operation will result in an externally visible error. For example, when a level-sensitive latch is accepting data rather than holding data, a strike on its stored bit may not result in an error, because the erroneous stored value will be overridden by the (correct) input value. If the latch is accepting data 50% of the time, this effect results in a *timing vulnerability factor* for the latch of 50%. For simplicity, we assume this timing vulnerability factor is already incorporated in the raw device fault rate. The computation of the device fault rate also includes some circuit-level vulnerability factors.

$$Vulnerability(reg) = \frac{\sum Vulnerable\ Interval\ Time}{\sum LifeTime}$$

$$Vulnerability(reg) = \frac{\sum Vulnerable\ Interval\ Time}{\sum LifeTime}$$



3. PROPOSED SELF-IMMUNITY TECHNIQUE

We propose to exploit the register values that do not require all of the bits of a register to represent a certain value. Then, the upper unused bits of a register can be exploited to increase the register's immunity by storing the corresponding SEC Hamming Code without the need for extra bits. The Hamming Code is defined by *k*, the number of bits in the original word and *p*, the required number of parity bits (approximately $\log_2 k$). Thus, the code word will be $(k + \log_2 k + 1)$. In our proposed technique, the optimal value of k is the value which guarantees that *w*, the bit-width of the register file, can cover both k, the required number of bits to represent the value, and the corresponding ECC bits of that value. In other words, the value and its ECC should be stored together within the bit-width of a register. Consequently, the following condition should be valid $(k + \log_2 k + 1 \leq w)$. Thus, the optimal value of k is 26 in 32-bit architectures and 57 in 64-bit architectures. For instance, when studying 32-bit architectures, where each register can represent a 32-bit value, we may exploit the register values, which require less than or equal to 26 bits by storing the corresponding ECC bits in the upper unused six bits of that register to enhance the register file immunity against soft errors1. We call this technique *Self-Immunity* and we call such values "26-bit" values. On the other hand, we call register values which need more than 26 bits to be represented "over-26-bit" register values. Figure-3.1 shows the percentage of register values usage for different applications of the MiBench Benchmark compiled for MIPS architecture.

Figure.1. "26-bit" register values and "over-26-bit" register values in different benchmarks

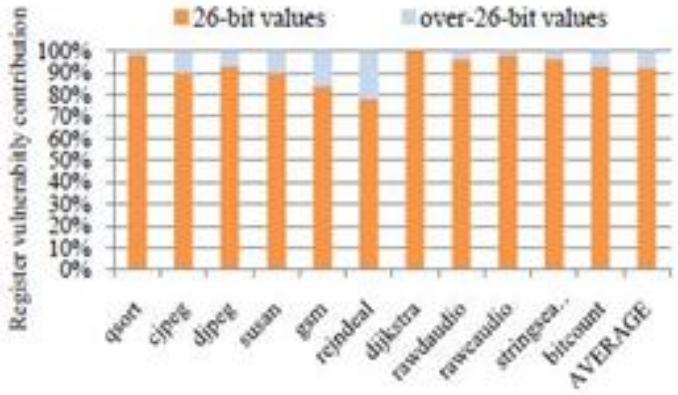


Figure.2. the fraction of vulnerable intervals of "26-bit" register values and "over-26-bit"register values in different benchmarks.

Architecture for Our Technique

The key challenge in distinguishing whether the ECC bits are embedded in the register value or not, is that the processor does not have sufficient information to make this decision when reading a value from a register. Consequently, we need to distinguish "26-bit" register values from "over-26-bit" register values. To do that, a *self-π* bit is associated with each register and we initially clear all *self-π* bits to indicate the absence of any *Self-Immunity*. For the sake of simplicity, we explain the proposed architecture with the required algorithms in two different steps.

Writing into a register- Figure3.3 illustrates that whenever an instruction writes avalue into a register it checks the upper six bits of that value if they are '0' or not. If they are (26-bit register value case), the corresponding *self-π* bit is set to '1' indicating the existence of *Self-Immunity*. The ECC value is generated and stored in the upper unused bits of the register. Hence, the data value and its ECC are stored together in that register. In the second case (over-26-bit register value), the corresponding *self-π* bit is set to '0' and the value is written into the register without encoding.

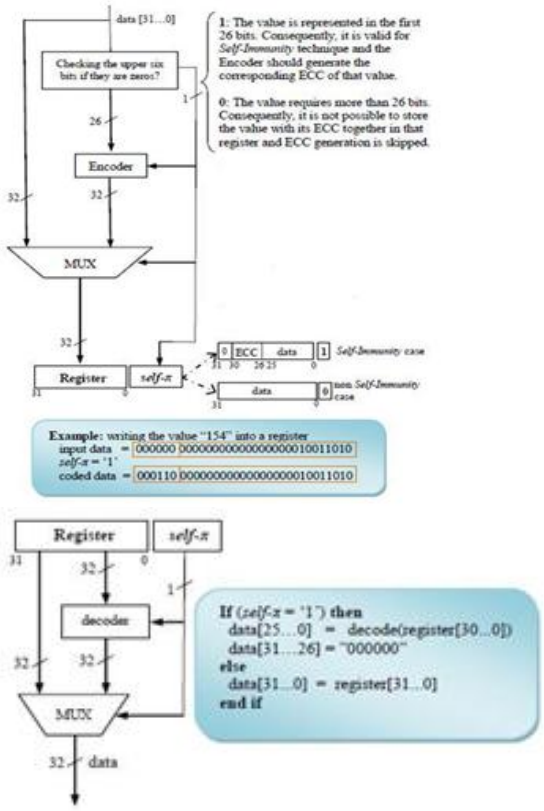


Figure.3.architectural support for reading a register value: Micro.

4. VLSI TECHNOLOGY

Gone are the days when huge computers made of vacuum tubes sat humming in entire dedicated rooms and could do about 360 multiplications of 10 digit numbers in a second. Though they were heralded as the fastest computing machines of that time, they surely don't stand a chance when compared to the modern day machines. Modern day computers are getting smaller, faster, and cheaper and more power efficient every progressing second. But what drove this change? The whole domain of computing ushered into a new dawn of electronic miniaturization with the advent of semiconductor transistor by Bardeen (1947-48) and then the Bipolar Transistor by Shockley (1949) in the Bell Laboratory.

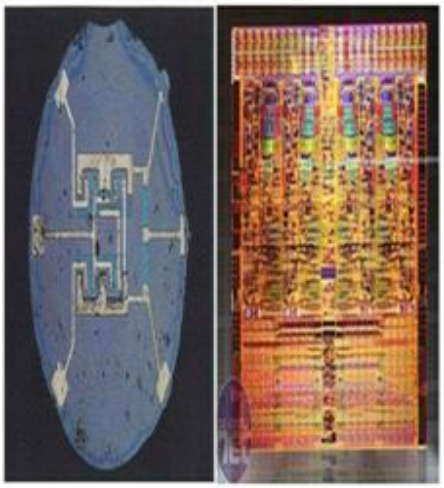


Figure.4. A comparison: first planar IC(1961) and Intel nehalem quad core die

Since the invention of the first IC (Integrated Circuit) in the form of a Flip Flop by Jack Kilby in 1958, our ability to pack more and more transistors onto a single chip has doubled roughly every 18 months, in accordance with the Moore's Law. Such exponential development had never been seen in any other field and it still continues to be a major area of research work.

Introduction of HDL

Hardware description languages (HDLs), mainly to describe logic equations to be realized in programmable logic devices (PLDs).in the 1990s, HDL usage by digital systems designers accelerated as PLDs, CPLDs, and FPGAs became inexpensive and common place. Designers turned to HDLs as a means to design individual modules within a system-on-chip.The important innovations in HDLs occurred in the mid-1980s, and were the developments of VHDL and VERILOG HDL became popular. There are several steps in an HDL based design process, often called the design flow. These steps are applicable to any HDL based design process and are shown in figure.

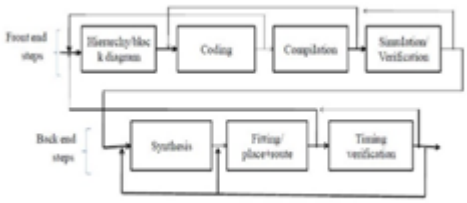


Figure.5. Steps in an HDL Based Design Flow

In any design, specifications are written first. Specifications describe the functionality, interface and overall architecture of the digital circuit to be designed. The next step is the actual writing of HDL code for modules, their interfaces and their internal details. After the code has written we have to compile the code, this step is known as compilation. Here the HDL compiler analyzes the code for syntax errors and also checks it for compatibility with other modules which it relies.

HDL Tool Suites

- HDL tool suite really has several different tools with their own names and purposes:
- A text editor allows writing, editing and saving an HDL program. It often contain HDL specific features, such as recognizing specific file name extensions and recognizing HDL reserved and comments and displaying them in different colors.
- The compiler is responsible for parsing the HDL program, finding syntax errors and figuring out what the program really says.
- A synthesizer or synthesis tools targets the design to a specific hardware technology, such as FPGA, ASIC etc...
- The simulator runs the specified input sequence on the described hardware and determines the values of the hardware¡¦s internal signals and its outputs over a specified period of time.

- The output of the simulator can be include waveforms to be viewed using the waveform editor.
- A schematic viewer may create a schematic diagram corresponding to an HDL program, based on the intermediate-language output of the compiler.
- A translator targets the compilers intermediate language output to a real device such as PLD, FPGA OR ASIC.

5. VHDL STRUCTURE

The VHDL structure or model is shown in figure. A single component model is composed of one entity and one of more architecture. The entity represents the interface specification (I/O) of the component. It defines the components external view, sometimes referred to as its pins .The architecture(s) describe(s) the internal implementation of an entity.

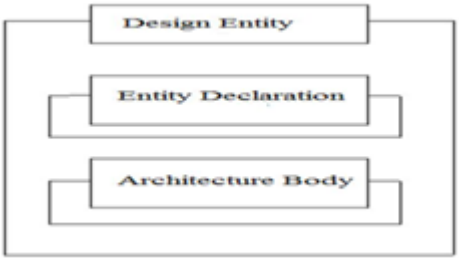


Figure.6. VHDL Structure

Verilog HDL

Verilog, standardized as IEEE 1364, is a hardware description language (HDL) used to model electronic systems. It is most commonly used in the design and verification of digital circuits at the register transfer level of abstraction. It is also used in the verification of analog circuits and mixed signal circuits. Hardware description languages such as Verilog differ from software programming languages because they include ways of describing the propagation time and signal strengths (sensitivity). There are two types of assignment operators; a blocking assignment (=), and a non-blocking (<=) assignment. The non-blocking assignment allows designers to describe a state-machine update without needing to declare and use temporary storage variables. Since these concepts are part of Verilog's language semantics, designers could quickly write descriptions of large circuits in a relatively compact and concise form. At the time of Verilog's introduction (1984), Verilog represented a tremendous productivity improvement for circuit designers who were already using graphical schematic capture software and specially written software programs to document and simulate electronic circuits. A Verilog design consists of a hierarchy of modules. Modules encapsulate design hierarchy, and communicate with other modules through a set of declared input, output, and bidirectional ports. Internally, a module can contain any combination of the following: net/variable declarations (wire, reg, integer, etc.), concurrent and sequential statement blocks, and instances of other modules (sub-hierarchies). Sequential statements are placed inside a begin/end block and executed in sequential order within the block. However, the blocks themselves are executed concurrently, making Verilog a dataflow language. Verilog's concept of 'wire' consists of both signal values (4-state: "1, 0, floating, undefined") and signal strengths (strong, weak, etc.). This system allows abstract modelling of shared signal lines, where multiple sources drive a common net. When a wire has multiple drivers, the wire's (readable) value is resolved by a function of the source drivers and their strengths.

Summary:

1. Verilog is based on C, while VHZDL is based on Pascal and Ada.
2. Unlike Verilog, VHDL is strongly typed.
3. Unlike VHDL, Verilog is case sensitive.
4. Verilog is easier to learn compared to VHDL.
5. Verilog has very simple data types, while VHDL allows users to create more complex data types.
6. Verilog lacks the library management, like that of VHDL

7. SIMULATION RESULTS

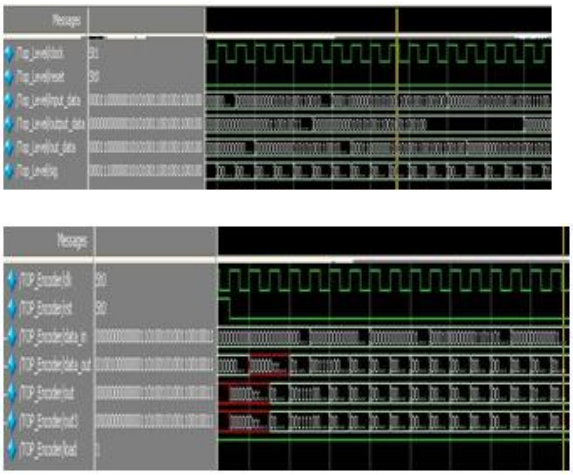


Figure.7.Test wave forms for Encoding block

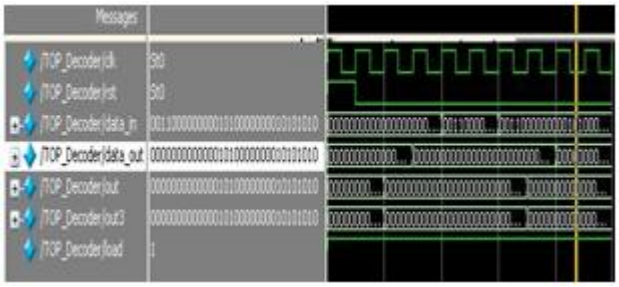


Figure.8. Test wave forms for Decoding block

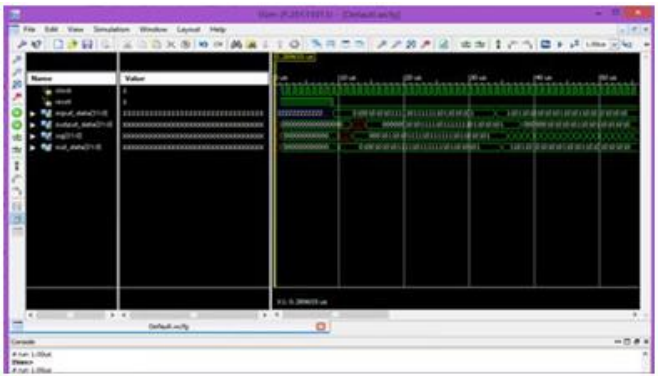


Figure.9.Test wave forms for Top-level block

OUTPUT VERIFICATION:

Input:
Clk - leading edge 1
Trailing edge 0
Time period 1 us
Reset 1
Run
Reset 0
Input 00001010101010101010101010101010
Run
Input
10101010
11001010
00110101
10011001
run

INTRODUCTION TO FPGA

FPGA contains a two dimensional arrays of logic blocks and interconnections between logic blocks. Both the logic blocks and interconnects are programmable. Logic blocks are programmed to implement a desired function and the interconnections are programmed using the switch boxes to connect the logic blocks. To be more clear, if we want to implement a complex design (CPU for instance), then the design is divided into small sub functions and each sub function is implemented using one logic block. Now, to get our desired design (CPU), all the sub functions implemented in logic blocks must be connected and this is done by programming the internal structure of an FPGA which is depicted in the following figure 5.1.

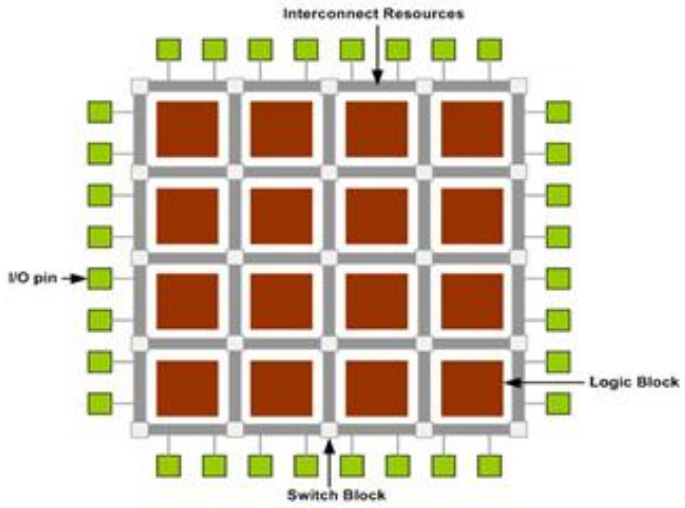


Figure.10. FPGA interconnections

FPGAs, alternative to the custom ICs, can be used to implement an entire System On one Chip (SOC). The main advantage of FPGA is ability to reprogram. User can reprogram an FPGA to implement a design and this is done after the FPGA is manufactured. This brings the name "Field Programmable. "Custom ICs are expensive and takes long time to design so they are useful when produced in bulk amounts. But FPGAs are easy to implement within a short time with the help of Computer Aided Designing (CAD) tools (because there is no physical layout process, no mask making, and no IC manufacturing).

Some disadvantages of FPGAs are, they are slow compared to custom ICs as they can't handle vary complex designs and also they draw more power. Xilinx logic block consists of one Look Up Table (LUT) and one Flip-Flop. An LUT is used to implement number of different functionality. The input lines to the logic block go into the LUT and enable it. The output of the LUT gives the result of the logic function that it implements and the output of logic block is registered or unregistered output from the LUT.

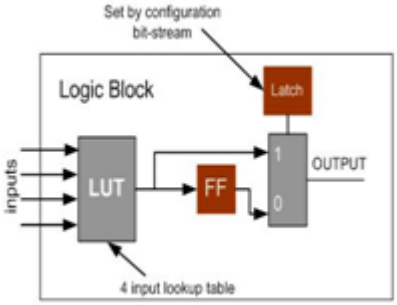


Figure.11.shows a 4-input LUT based implementation of logic block

FPGA Design Flow

In this part of tutorial we are going to have a short intro on FPGA design flow. A simplified version of design flow is given in the flowing diagram.

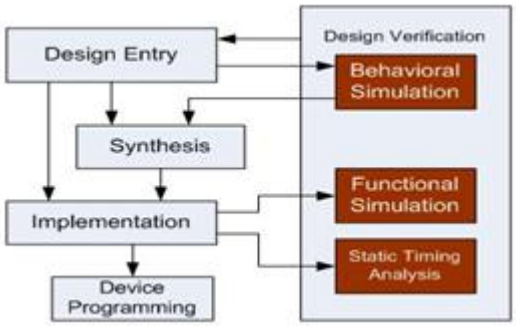


Figure.12.FPGA Design Flow

Synthesis

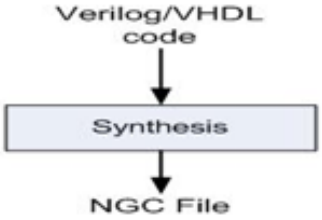


Figure.13.Synthesis

FPGA Synthesis

The process that translates VHDL/ Verilog code into a device netlist format i.e. a complete circuit with logical elements (gates flip flop, etc…) for the design. If the design contains more than one sub designs, ex. to implement a processor, we need a CPU as one design element and RAM as another and so on, then the synthesis process generates netlist for each design element Synthesis process will check code syntax and analyze the hierarchy of the design which ensures that the design is optimized for the design architecture, the designer has selected.

The resulting netlist(s) is saved to an NGC (Native Generic Circuit) file (for Xilinx® Synthesis Technology (XST)).

Implementation

This process consists of a sequence of three steps

- Translate
- Map
- Place and Route

Translate:

Process combines all the input netlists and constraints to a logic design file. This information is saved as a NGD (Native Generic Database) file. This can be done using NGD Build program. Here, defining constraints is nothing but, assigning the ports in the design to the physical elements (ex. pins, switches, buttons etc) of the targeted device and specifying time requirements of the design. This information is stored in a file named UCF (User Constraints File). Tools used to create or modify the UCF are PACE, Constraint Editor Etc.

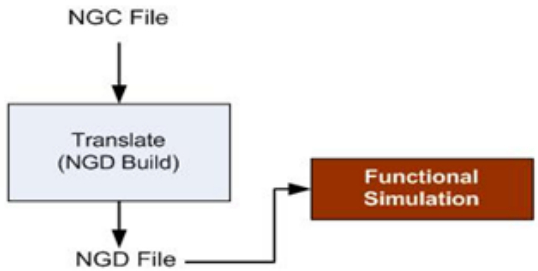


Figure.14. FPGA Translate

Map:

Process divides the whole circuit with logical elements into sub blocks such that they can be fit into the FPGA logic blocks. That means map process fits the logic defined by the NGD file into the targeted FPGA elements (Combinational Logic Blocks (CLB), Input Output Blocks (IOB)) and generates an NCD (Native Circuit Description) file which physically represents the design mapped to the components of FPGA. MAP program is used for this purpose.

RTL Schematic

The RTL (Register Transfer Logic) can be viewed as black box after synthesize of design is made. It shows the inputs and outputs of the system. By double-clicking on the diagram we can see gates, flip-flops and MUX.

Figure.15. Schematic with Basic Inputs and Output

Here in the above schematic, that is, in the top level schematic shows all the inputs and final output of design.

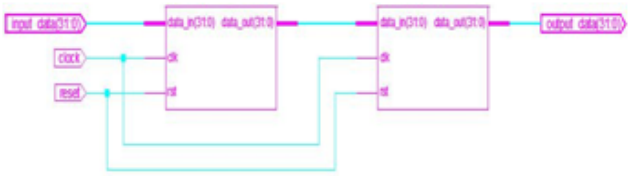


Figure.16. Blocks inside the Top Level Design

The internal blocks available inside the design includes encoder, decoder which were clearly shown in the above schematic level diagram. Inside each block the gate level circuit will be generated with respect to the modeled HDL code.

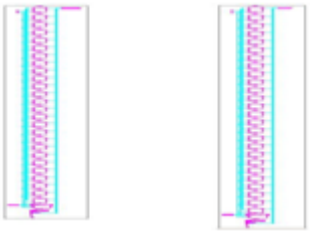


Figure.17. Blocks inside the Developed Encoder, Decoder Design

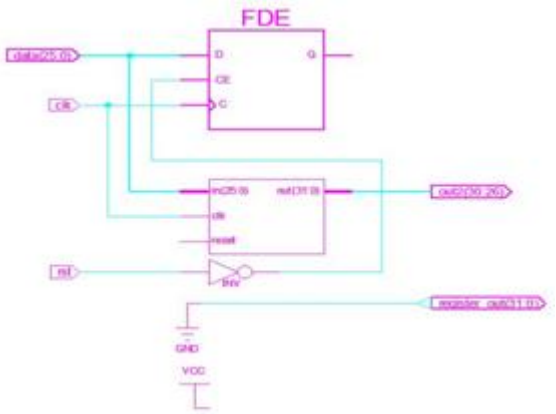


Figure.18. ECC Block inside the encoder Design

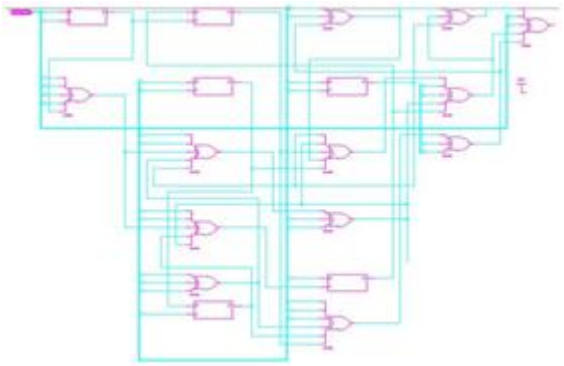


Figure.19. Encoder block inside the ECC Design

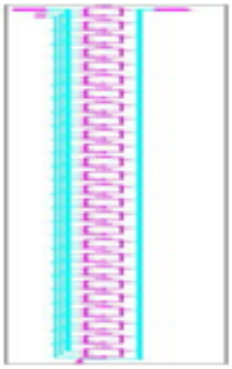


Figure.20. The decoder block

Synthesis Result

This device utilization includes the following.

- Logic Utilization
- Logic Distribution

- Total Gate count for the Design

Device Utilization Summary				
Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Flip Flops	104	9,312	1%	
Logic Distribution				
Number of occupied Slices	52	4,656	1%	
Number of Slices containing only related logic	52	52	100%	
Number of Slices containing unrelated logic	0	52	0%	
Number of bonded IOBs	60	232	25%	
Number of BUFGMUXs	1	24	4%	

The device utilization summery is shown above in which its gives the details of number of devices used from the available devices and also represented in %. Hence as the result of the synthesis process, the device utilization in the used device and package is shown above.

==========*

Final Report *

==========

Final Results

RTL Top Level Output File Name: Top_Level.ngr

Top Level Output File Name : Top_ Level

Output Format : NGC

Optimization Goal : Speed

Keep Hierarchy : NO

Design Statistics

IOs : 66

Cell Usage :

BELS : 2

GND : 1

INV : 1

FlipFlops/Latches : 104

FDE : 26

FDR : 78

Clock Buffers : 1

BUFGP : 1

IO Buffers : 59

IBUF : 27

OBUF : 32

Device utilization summary:

Selected Device : 3s500efg320-4

Number of Slices:	60 out of	4656	1%
Number of Slice Flip Flops:	104 out of	9312	1%
Number of 4 input LUTs:	1 out of	9312	0%
Number of IOs:	66		
Number of bonded IOBs:	60 out of	232	25%
Number of GCLKs:	1 out of	24	4%

Partition Resource Summary:

No Partitions were found in this design.

==========

TIMING REPORT

NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE. FOR

ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT GENERATED AFTER PLACE-and-ROUTE.

Clock Information:

Clock Signal	Clock buffer(FF name)	Load
Clock	BUFGP	104

Asynchronous Control Signals Information:

No asynchronous control signals found in this design

Timing Summary:

Speed Grade: -4

Minimum period: 1.319ns (Maximum Frequency: 758.150MHz)

Minimum input arrival time before clock: 5.014ns

Maximum output required time after clock: 4.283ns

Maximum combinational path delay: No path found

Timing Detail:

All values displayed in nanoseconds (ns)

The below figure shows the place and route of the design, the area occupation as well in the FPGA.

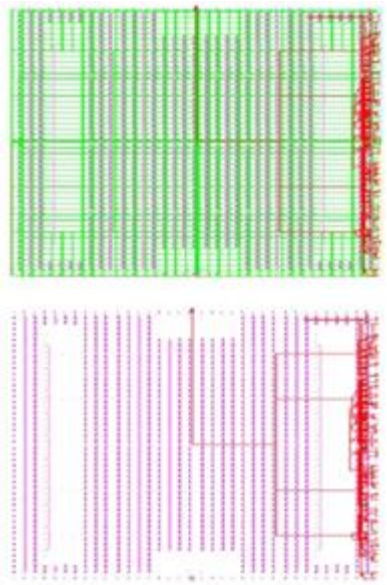


Figure.21. The area occupation of the Design in different colors

7. CONCLUSION

For embedded systems under stringent cost constraints, where area, performance, power and reliability cannot be simply compromised, we propose a soft error mitigation technique for register files. Our experiments on different embedded system applications demonstrate that our proposed *Self-Immunity* technique reduces the register file vulnerability effectively and achieves high system fault coverage. Moreover, our technique is generic as it can be implemented into diverse architectures with minimum impact on the cost. It can be concluded that this technique achieves the best overall result compared to state-of-the-art in register file vulnerability reduction.

8. FUTURE SCOPE

Cyber security needs much more attention. Given human limitations and the fact that agents such as computer viruses and worms are intelligent, network-centric environments require intelligent cyber sensor agents (or computer-generated forces) which will detect, evaluate and respond to cyber-attacks in a timely manner. Even though computational intelligence techniques have been widely used in the field of computer security and forensics, there are certain ethical and legal problems that arise as the technology rapidly expands. Some of these problems are privacy concerns or power issues on the ethical side or questions of due process on the legal side. A wide range of both ethical and legal questions come up in the light of the potential autonomy of this technology. Questions like "to what extent can an artificial neural network replace human judgment", "to what degree do we want to allow technology to take human roles" or "what legal precedent can be applied to machines" will need to be answered

9. REFERENCES

[1]. Greg Bronevetsky and Bronis R. de Supinski," Soft Error Vulnerability of Iterative Linear Algebra Methods," in the 22nd annual international conference on Supercomputing, pp. 155-164, 2008.

[2]. J.L. Autran, P. Roche, S. Sauze, G. Gasiot, D. Munteanu, P. Loaiza, M. Zampaolo and J. Borel, "Real-time neutron and alpha soft-error rate testing of CMOS 130nm SRAM: Altitude versus underground measurements," in ICICDT '08, pp. 233–236, 2008.

[3]. S.S. Mukherjee, C. Weaver, J. Emer, S.K. Reinhardt and T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," in International Symposium on Micro architecture (MICRO-36), pp.29- 40, 2003.

[4] T.J. Dell, "A whitepaper on the benefits of Chippkill-Correct ECC for PC server main memory," in IBM Microelectonics division Nov 1997.

[5]. S. Kim and A.K. Somani, "An adaptive write error detection technique in on-chip caches of multi-level cache systems," in Journal of microprocessors and microsystems, pp. 561-570, March 1999.

[6]. G. Memik, M.T. Kandemir and O. Ozturk, "Increasing register file immunity to transient errors," in Design, Automation and Test in Europe, pp. 586-591, 2005.

[7].Jongeun Lee and AviralShrivastava, "A Compiler Optimization to Reduce Soft Errors in Register Files," in LCTES 2009.

[8]. Jason A. Blome, Shantanu Gupta, ShuguangFeng, and Scott Mahlke, "Cost-efficient soft error protection for embedded microprocessors," in CASES '06, pp. 421–431, 2006.

[9]. P. Montesinos, W. Liu, and J. Torrellas, "Using register lifetime predictions to protect register files against soft errors," in Dependable Systems and Networks, pp. 286–296, 2007.