A SOFT ERROR MITIGATION SCHEME TO INCREASE THE RESILIENCE OF REGISTER FILE

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Technology

In

VLSI Design & Embedded Systems

By

MADHUKER DHULAPELLY

Roll No: 209EC2125

Under the guidance of

Prof. S. K. Das

Department of Electronics & Communication Engineering

National Institute of Technology

Rourkela

2013

National Institute Of Technology
Rourkela

CERTIFICATE

This is to certify that the thesis entitled, **"A SOFT ERROR MITIGATION SCHEME TO INCREASE THE RESILIENCE OF REGISTER FILE** "submitted by MADHUKER DHULAPELLY in partial fulfillment of the requirements for the award of Master of Technology degree in **Electronics and Communication Engineering** with Specialization in "**VLSI Design & Embedded Systems**" during session 2012-2013 at National Institute of Technology, Rourkela (Deemed University) and is an authentic work by him under my supervision and guidance.

To the best of my knowledge, the matter embodied in the thesis has not been submitted to any other university/institute for the award of any Degree or Diploma.

Date:

Prof. S. K. Das
Dept. of ECE
National Institute of Technology
Rourkela-769008
Email: dassk@nitrkl.ac.in

National Institute Of Technology
Rourkela

DECLARATION

I hereby declare that this thesis entitled "**A SOFT ERROR MITIGATION SCHEME TO INCREASE THE RESILIENCE OF REGISTER FILE**" submitted to National Institute Of Technology, Rourkela for the award of the degree of Master of Technology is a record of original work done by me under the guidance Prof. S.K. Das and that it has not been used, they have been acknowledged.

Place: ROURKELA

Date:

Signature

Acknowledgment

This work would have not been possible without the guidance and the help of several individuals who in one way or another, contributed and extended their valuable assistance in the course of this study.

My utmost gratitude to **Prof. S. K. Das**, my dissertation adviser whose sincerity and encouragement I will never forget. **Prof. Das** has been my inspiration as I hurdle all the obstacles in the completion this research work and has supported me throughout my project work with patience and knowledge whilst allowing me the room to work in my own paradigms.

Sincere thanks to **Prof. S. K. Patra, Prof. K. K. Mohapatra, Prof. S. Meher, Prof. D.P.Acharya, Prof.S. Ari and Prof. P. Singh** for their constant cooperation and encouragement throughout the course. I also extend my thanks to entire faculties and staffs of Dept. of Electronics & Communication Engineering, National Institute of Technology, Rourkela who have encouraged me throughout the course of my Master's Degree.

I would like to thank to Mr.Durgaprasad for his guidance and endless support in solving queries and advices for betterment of this work as a second supervisor.

And finally thanks to my parents, my elder brother (who has been a constant source of inspiration) whose faith, patience and teaching had always inspired me to work upright in my life. Without all these beautiful people my world would have been an empty place.

Madhuker Dhulapelly

madhu468@gmail.com

Abstract

Register files are essential and integral part of any microprocessor architecture. Soft errors in the register file can quickly spread to various parts of the system and result in silent data corruption. Traditional redundancy based schemes to protect the register file are prohibitive because register file is often in the timing critical path of the processor. Since it is one of the hottest blocks on the chip, adding any extra circuitry to it is not desirable. For embedded systems under severe cost constraints, where power, performance, area and reliability cannot be simply compromised, we propose a soft error reduction technique for register files.

This thesis introduces a soft error mitigation scheme, called Self-Immunity to increase the resilience of the register file from soft errors, by using unused bits of the register file. It is desirable for processors that demand high register file integrity under stringent constraints.

This thesis explains the implementation of our proposed technique to protect the register file from soft errors. And show the best overall results compared to state-of-the-art in register file vulnerability reduction with minimum impact on the area and power.

List of Abbreviations

RF	Register File
ECC	Error Correction Code
MTBF	Mean Time between Failures
SDC	Silent Data Corruption
DUE	Detected Unrecoverable Errors
FIT	Error in Time
AVF	Architectural Vulnerability Factor
TMR	Triple Modular Redundancy
RVF	Register Vulnerability Factor

Contents

List of Figures

Chapter-1

Introduction

1.1 Introduction

Soft error is defined as a wrong signal or data stored in electronic memory. A soft error will damage the data being processed but it will not damage any hardware of the system. Soft errors are classified as chip level soft errors and system level soft errors. Chip level soft errors occur due to emission of alpha particles into the chip when the radioactive atoms in the chip's material decay. The emitted alpha particle have a positive charge and kinetic energy, it can hit a memory cell and change the state to a different value. But it does not damage the actual structure of the chip because of a tiny atomic reaction. System-level soft error occurs due to noise. When the data is on a data bus and hit with a noise phenomenon, then computer tries to interpret the noise as a data bit. The bad data bit also saved in memory and later it will cause problems.

Occurrence of a soft error in a circuit depends on the energy of the incoming particle, the location of the strike, geometry of the impact, and the logic design of the circuit. And also it depends on the critical charge. The critical charge , Qcrit is defined as the minimum electron charge disturbance required to changing the logic level of the circuit. The critical charge is depends on both voltage and capacitance. Continuous shrinking in chip feature size and supply voltage, decreases Qcrit. Thus, the importance of reduction of soft errors increases as chip technology advances.

Over the last decade, the technology scaling has raised soft errors to become one of the major sources for processor crashing, regardless of the increasingly complex architectures in the nano scale era. Soft errors caused by charged particles are dangerous primarily in high atmospheric conditions, where heavy alpha particles are available [1]. However, trends in today's nanometer technologies such as aggressive shrinking have made low-energy particles, which are more than high-energy particles, cause sufficient charge to provoke soft errors. Furthermore, there is a exact prediction that soft errors will become a cause of an unacceptable error rate problem in the near future even in earthbound applications [2]. Researchers have

mainly and traditionally focused on mitigating soft errors in memory and cache structures [4], [5], [13], due to their large sizes.

On the other hand, relatively little work had been conducted for register files although they are very susceptible against soft errors [8]. Despite the overall rather small area footprint of the register file, it is accessed more often than any other architectural component [6] [9]. Thus, the corrupted data in any register may propagate rapidly throughout the other parts of processor, leading to severe system reliability problems [6]. In fact, soft errors in register files can be the cause of a large number of system failures [10]. Recently, Blome et al. [8] showed that a considerable amount of faults that affect a processor usually come from the register file. Though, some processors protect their registers with Error Correction Code (ECC) [11], but such solutions may be prohibitive in certain applications (like embedded) due to the significant impact in terms of area and power [14]. Moreover, power consumption was conventionally a major concern in embedded systems due to their considerable effects on the system.

1.2 Register file soft errors

Register file (RF) is highly vulnerable to soft errors, and existing redundancy based soft error protection schemes to protect the RF are prohibitive because RF is often in the timing critical path of the processor. Since it is one of the important blocks on the chip, and adding any extra circuit to it is not desirable. Technology scaling trends increase the susceptibility of microprocessors to soft errors. The demand for embedded microprocessors in a wide array of safety critical applications, compounds the importance of the soft error problem.

To overcome these problems, there is a severe need of techniques to increase the register file integrity against soft errors with a small effect on both area and power overhead. This thesis addresses this challenge by introducing a technique, called Self-Immunity to protect the register file from soft errors, especially necessary for processors that require high register file integrity under stringent constraints.

This thesis will contribute the following things:

(1) A soft error mitigation technique for improving the immunity of register files against soft errors by storing the ECC in the unused bits of a register.

(2) Protecting register file by achieving high area and power saving with a slight degrading in the register file vulnerability reduction (7%) compared to a full protection scheme.

Chapter-2

Background

2.1 Soft error background and terminology

2.1.1 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 errors or detected errors. The former referred to as *silent data corruption* (SDC); we call the latter *detected unrecoverable errors* (DUE). Note that detected recoverable errors are not errors. 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 (Failure 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.

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*, defined as the probability that a circuit fault will result in an observable error. The total FIT rate of the chip is calculated by adding the effective FIT rates of all the structures on the chip.

Current predictions show that typical raw FIT rate numbers for latches and SRAM cells vary between 0.001 – 0.01 FIT/bit at sea level. The total FIT contribution of logic gates today is a negligible fraction of the FIT contribution from latches, so we concern ourselves only with faults due to strikes on latches and SRAM cells. In the future, if the contribution of logic becomes non-negligible, we could incorporate the effective FIT rate due to a logic block into the FIT rate of the latch that it feeds.

2.1.2 Vulnerability Factors

The effective FIT rate per bit is influenced by several *vulnerability factors.* A vulnerability factor shows 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.

The *architectural vulnerability factor* (AVF) expresses the probability that a visible system error will occur given a bit flip in a storage cell. The AVF can have a significant impact on the effective error rate of a processor. Prior studies with statistical fault injection into RTL models have demonstrated AVFs of 1%-10% for latches and 0% - 100% across a range of architectural and micro architectural state bits.

The earliest schemes of register file protection such as Triple Modular Redundancy (TMR) and ECC can achieve a high level of fault tolerance but they may not be suitable solutions in embedded systems due to their power and area overheads. Recently, Fazeli et al. [14] shown that protecting the entire register file with SEC-DED comes with about 20% power overhead. The proposed approach in [15] utilizes the Cross-parity check as a method for correcting multiple errors in the register files. Spica et al. [16] achieved a little gain (just 2%) in fault tolerance for caches if they increase the protection to Double Error Correction while the overhead for that gain is considerable.

Building on the concept of Architectural Vulnerability Factor (AVF), introduced by Mukherjee [3], Yan et al. [19] proposed the *Register Vulnerability Factor* (RVF) to describe the probability of a soft error in registers can be spread to other system components. In general, a value is written into a register, then it is read

frequently from the register, and later a new value is written into the register again. Thus, any soft error occurring during "write-write" or "read-write" intervals will have no effect on the system, because by the next write operation it will be automatically corrected. Whereas, the intervals "read-read" and "write-read" are considered as vulnerable intervals as shown in Figure-2.1. The RVF of a register is defined as the sum of the lengths of all its vulnerable intervals divided by the sum of the lengths of all its lifetimes [19].

$$\textbf{Vulnerability (reg)} = (\sum \textbf{Vulnerable Interval Time}) / (\sum \textbf{Life time})$$

Finally, the total vulnerability of the register file is taken as the sum of vulnerability of all registers [21].

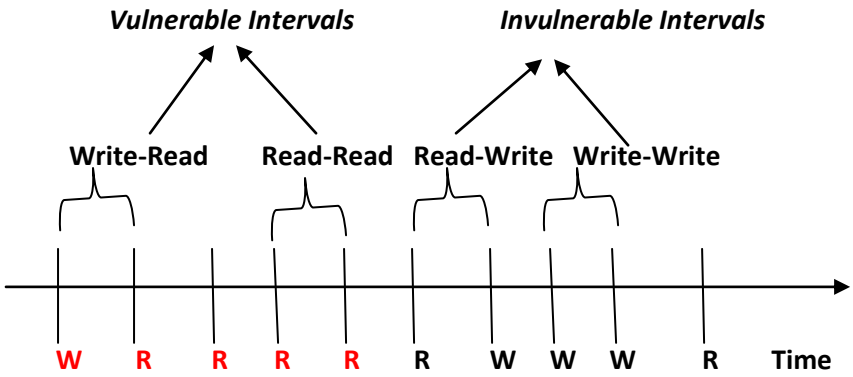

Figure-2.1: Different Register Access Intervals [19]

The pure software approach at compile level introduced by Yan et al. [19] re-schedules the instructions in order to decrease the RVF of a register file but the proposed technique is not always very effective because it may increase the execution cycles and even the RVF in some benchmarks [19]. In a bid to decrease the area and power penalties, Yan et al. [19] proposed a scheme to protect a subset of the registers instead of full protection schemes and modify the register allocation algorithm to assign the most sensitive registers against soft errors to the protected registers. The achieved RVF reduction is 23%, 41%, 67% and 93% for protecting 2, 4, 8 and 16 out of 32 registers respectively. Montesinos [9] make a decision of which register values

should be protected at runtime by hardware logic but the runtime prediction is very costly in terms of energy [22]. Lee et al. [7] presented a compile technique to reduce RVF by protecting a small part of memory and write the vulnerable register values in this memory by inserting load/store instructions but it increases both runtime and code size.

Another important approach is In-Register Replication "IRR" [17], which exploits the fact that a large fraction of register values are less than or equal to 16 bits wide for 32- bit architectures. Such values can be replicated in the same register for increasing the immunity against soft errors. The fundamental conflict is that, while increasing the immunity of the register file against soft errors by reducing the vulnerability of the register file, this reduction (either with full or partial protection schemes) increases the area and power overheads.

2.2 Register Vulnerability factor

While it is critical to protect the register file against soft errors, not all soft errors occurred in the register file can lead to visible system faults. Over-estimation of the soft error problem can result in over-design of the protection mechanisms, which will increase the reliability cost eventually. On the other hand, insufficient protection of register files will make the system unreliable and thus is useless. As a result, designers must accurately measure the probability that register soft errors can impact other system components and thus lead to erroneous final output. Recently, Mukherjee proposed the concept of architectural vulnerability factor (AVF), which is defined as the probability that a fault in a processor structure will lead to a visible error in the final output of program. In general, the AVF provides designers an accurate estimate of the soft error rate for various hardware components for making cost/reliability trade-offs. While the concept of AVF can also be applied to the register file, but it fails to exploit the fact that soft errors in the register file can be automatically overlapped by the new values written to the register file. If a value with soft errors is written before it is read, it will have no impact on the system output.

Toward the goal to measure register file susceptibility to soft errors accurately and quantitatively, we define the register vulnerability factor (RVF) to be the probability that a soft error in registers can be propagated to other system components (i.e., functional units, memory). In contrast to the concept of AVF, which focuses on the effect of soft error propagation, the RVF concentrates on the probability of soft error propagation to other hardware elements. It should be noted that even if a soft error occurred in the register file is consumed by an instruction; it may still not affect the final output since this instruction may be miss-speculated. Obviously, the RVF and the AVF can be combined to select the most cost-effective techniques to increase the register file reliability against soft errors. Since processors only employ a limited number of architecture registers while programs typically use a large number of values, multiple values can be stored in the same register as long as their lifetimes do not overlap. In general, a value is first written into a register, then it is read by one or more times and finally another value is written into the same register, which finishes the lifetime of the old value and begins the lifetime of the new value.

As depicted in figure-2.1, we can divide the accesses to register files into four different patterns (or intervals), namely, the write-read (W-R), read-read (R-R), read-write (R-W) and write-write (W-W) patterns (note that the read/write mentioned in the thesis refers to the corresponding operations on register values, including but not limited to the load/store instructions, which operate on the data from the memory hierarchy). Among these four patterns, the register file is only susceptible to soft errors during the W-R and R-R intervals. In contrast, the soft errors occurred during the R-W and W-W intervals can be overlapped by the latter write operations, and hence will not impact other system components. It is widely accepted that fault-inducing particle strikes are randomly and uniformly distributed, therefore, the probability that a soft error in registers can be propagated to other system components can be computed as the average ratio that the register values are exposed to the susceptible intervals (i.e., W-R and R-R), as described in Equation below. In this Equation, RVi represents any register value, the Susceptible Time (RVi) represents the time intervals that RVi is exposed to the susceptible intervals (i.e., W-R and R-R intervals for RVi), and the Lifetime (RVi) represents the lifetime of RVi, which is time interval between the time that a register is allocated for RVi and the time it is

overlapped by another value. Since both the Susceptible Time (RVi) and Lifetime (RVi) can be easily obtained from a performance simulator, by using these values we can compute the RVF.

RVF= (∑Susceptible Time (RVi)) / (∑Lifetime (RVi))

The RVF indicates the probability that register soft errors can spread to other hardware elements and thus impact the system output. The higher the RVF, the lower the register file reliability, and hence more expensive techniques are needed to fight soft errors. Measuring the RVF is not only useful to understand the reliability requirement of register files accurately for avoiding both the over-protection or under-protection, it also opens up avenues for software (e.g., compiler) to enhance register file reliability by reordering the read/write operations to reduce the RVF. In contrast, traditional software optimizations mainly focus on performance. Therefore, the RVF allows compilers to consider both performance and reliability to optimize the register access patterns. Such a software based approach has no hardware overhead, which is fundamentally different from traditional space redundancy or information redundancy techniques.

Methods to reduce register vulnerability factor are:

(a) Rescheduling of Instructions to Reduce RVF.
(b) Reliability oriented Register Assignment with Partial ECC Protection.

A simple way to reduce RVF is to find heavily executed loops, identify unused registers in them, and save or restore the registers before or after the loops. Anyhow, with such an ad-hoc method, it is not only hard to achieve optimal results but also very cumbersome to handle complex control flows, recursive functions, and even function calls. Furthermore, an intra-procedural optimization has a fundamental weakness that it can protect even unnecessary intervals, which significantly lowers the efficiency of ad-hoc methods. We approach this as an optimization problem: given a performance bound, what is the set of program points in which to insert save/restore operations so that the transformed program will achieve the minimum RVF with minimal code size overhead? This is inherently an inter-procedural problem, since

register save/restore operations can easily affect other functions along the program paths, not only in terms of functionality but also more in terms of RVF, and also because identifying long vulnerable intervals will necessarily demand considering more than one functions. Other challenges include devising simple yet effective save/restore operations, inserting them not overly but just enough to guarantee the code size, the program correctness, and accurately estimating their effect on performance, and RVF, in the midst of complex control flows and function calls.

2.3 Summary

With the shrinking feature size, lower supply voltage, higher density and frequency, soft errors are becoming an increasing challenge for microprocessor design. To protect processors against soft errors, the first step is to understand the vulnerability of different hardware components to soft errors. Based on the accurate estimate of the reliability requirement, the most cost-effective technique can be selected to meet the pre-defined reliability goal, which is of particular importance for embedded systems with cost constraints.

Chapter-3

Self-Immunity Technique

3.1 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 particular value. Then, the upper unused bits of a register can be used to increase the register's immunity by storing the corresponding SEC Hamming Code [11] without the requirement for extra bits. The Hamming Code is indicated 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 $(k + \log_2 k + 1)$ [23]. In our proposed technique, the optimal value of k is the value that 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 ECC bits of the value. The value and its ECC bits should be stored together within the bit-width of a register. Consequently, the condition $(k + \log_2 k + 1 \leq w)$ should be valid. Thus, in 32-bit architectures the optimal value of k is 26 and in 64-bit architectures k is 57.

For instance, when studying 32-bit architectures, where each register represent a 32-bit value, we may exploit the register values, which require less than or equal to 26 bits by storing the ECC bits in the upper unused six bits of that register to enhance the register file immunity against soft errors. We call this technique *Self-Immunity* and we call such values "26-bit" values. On the other hand, we call the 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 [12] compiled for MIPS architecture.

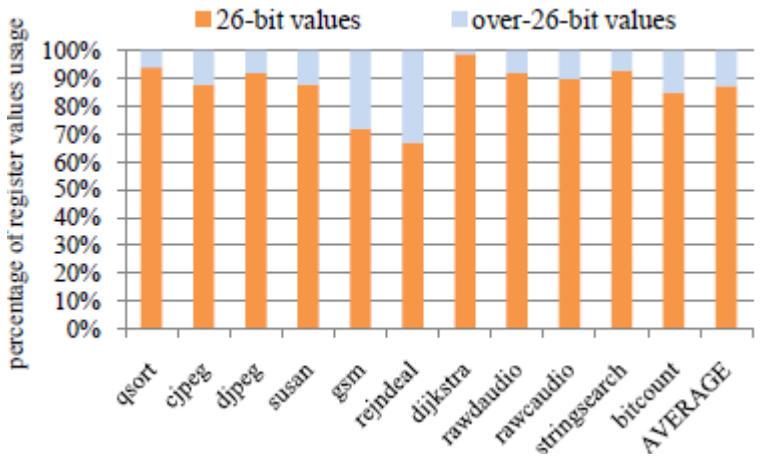

Figure-3.1: "26-bit" register values and "over-26-bit" register values in different benchmarks [24]

As it can be noticed, in all benchmarks most of the register values are "26-bit" values. In other words, the upper six bits of 88% of the stored data in the register file are actually unused. Consequently, we can store the corresponding ECC in these available bits and increase the register's immunity. In addition to the previous key observation, the contribution of "26-bit" register values in the total vulnerable intervals is much more than the contribution of "over-26-bit" register values. In Figure3.2, the fraction of vulnerable intervals of each benchmark is reported. As is demonstrated, the fraction of vulnerable intervals of "26- bit" values is 93% on average.

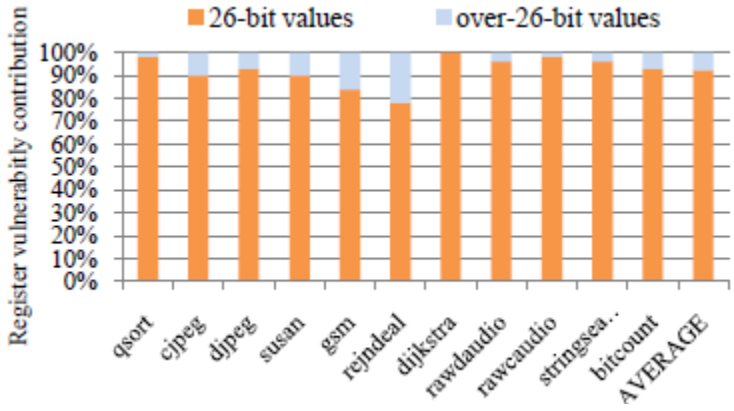

Figure-3.2: The fraction of vulnerable intervals of "26-bit" register values and "over-26-bit" register values in different benchmarks [24].

3.2 Problem Description

The goal of our technique is to reduce vulnerability of the register file with minimum impact on both area and power. Let *N* is the total number of registers and *V* is the vulnerability of a register, then the vulnerability of the register file is ($\Sigma^{N}_{i=1} V_i$). As the power overhead comes from accessing the encoder and decoder, it can be approximately modeled through the number of accesses [22]. Let *M* is the number of protected register values and *A* is the number of accesses, then the total power overhead can be estimated as ($\Sigma^{M}_{i=1} A_i$). Thus, our goal can be formulated as:

$$\textbf{Minimize } (V = \Sigma^{N}_{i=1} V_i), \textbf{ Minimize } (P = \Sigma^{M}_{i=1} A_i)$$

3.3 Architecture for Our Technique

The difficulty 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 while reading a value from a register. Hence, we need to classify "26-bit" register values from "over-26-bit" register values. To do this, 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.

3.3.1 Writing into a register- Figure3.3 illustrates that whenever an instruction writes a value 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 will be generated and stored in the upper unused bits of the register. Hence, the data bits and its ECC bits 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.

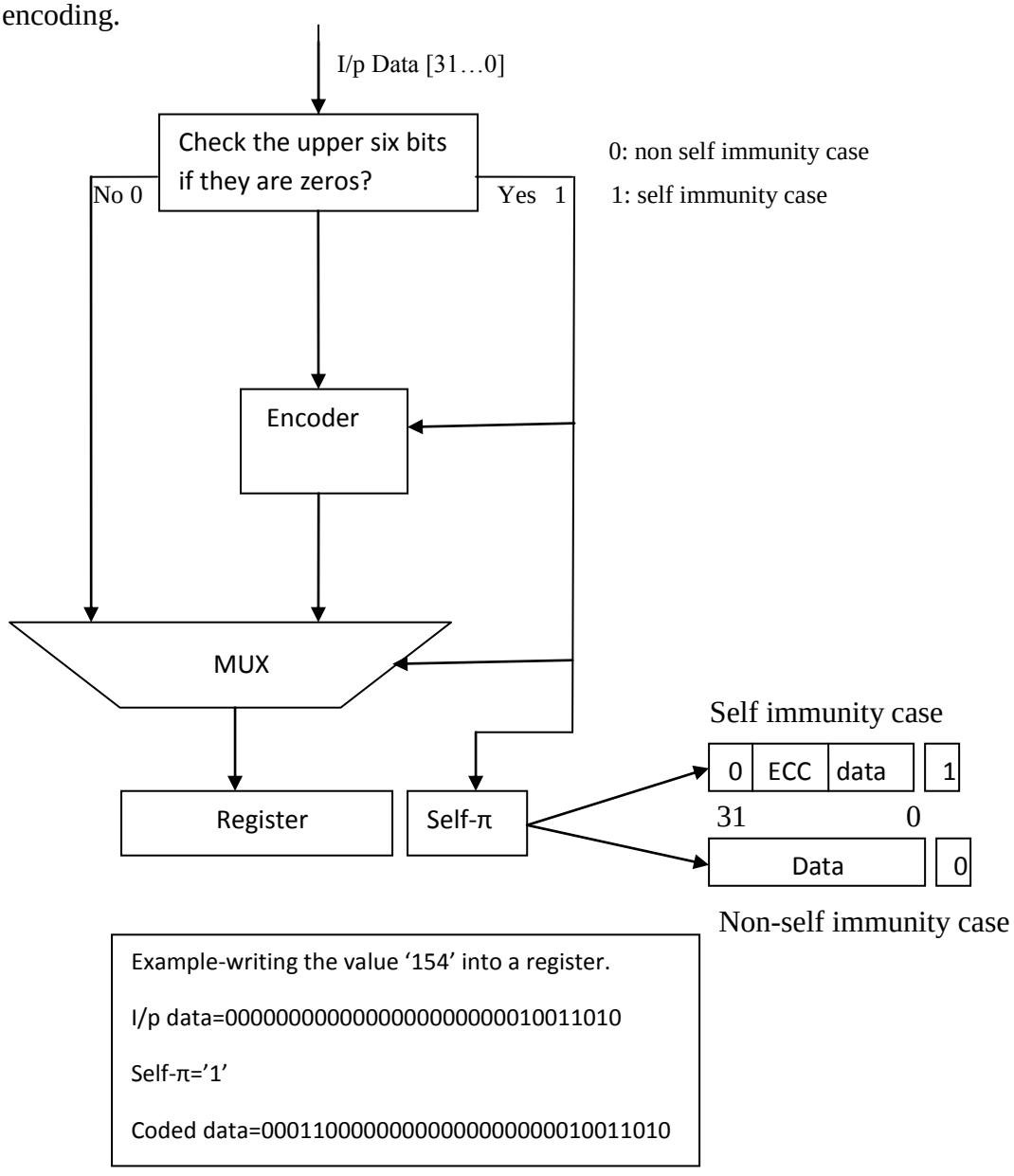

Figure-3.3: Micro-architectural support for writing a register value.

3.3.2 Reading from a register: For read operations, to distinguish between a *Self-Immunity* case and a non *self-Immunity* case the *self-π* bit is used. In the self immunity case, the data value and the corresponding ECC bits are stored together in that register and consequently the value should be decoded. In the non self immunity case, the stored value is not encoded as a result there is no need to decode the value as shown in Figure-3.4.

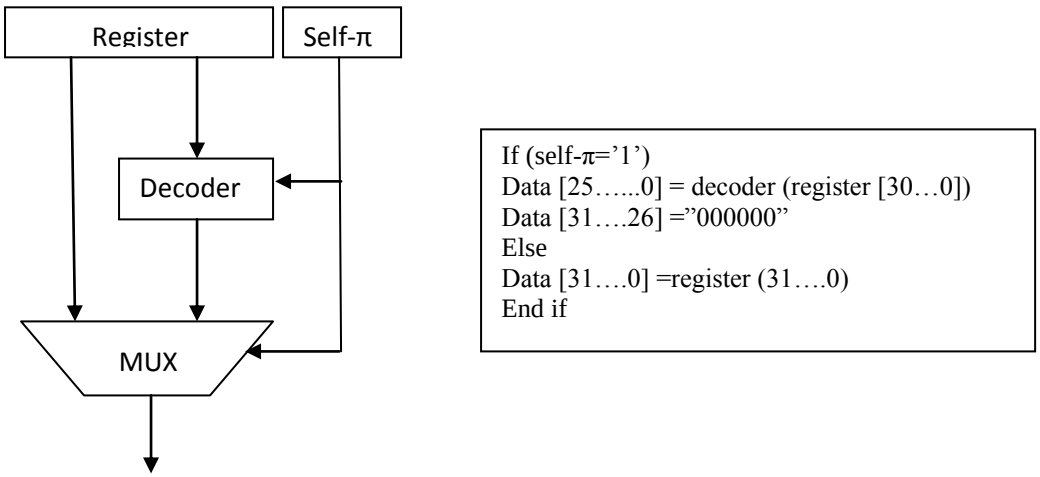

Figure-3.4: Micro architectural support for reading a register value.

3.4 Implementation details

Since the probability of multiple bit-errors is largely lower than the single bit-error [20], a single bit-error model has been considered. In this fault injection environment, faults will be injected on the fly while the processor executes an application. In the each fault injection simulation, one of the 32 registers is chosen randomly and a bit in that register is selected randomly and then flipped. Observe that a *write* operation clears out the previous injected error into that register by storing a new value. Likewise, by using a uniform distribution, a random cycle is selected as the time that soft error occurs. This guarantees that the faults will be provoked only when the program is executed [20]. Since an injected fault might create an infinite loop, a watchdog timer was implemented for the required number of execution cycles. When the cycle count exceeds two times the number of cycles in the fault-free case, we stop the simulation.

Towards evaluating our proposed technique, to take into account different possible scenarios for register utilization, we use different applications from MiBench Benchmark compiled for MIPS architecture [12]. Simulations were conducted by using the MIPS model simulator [18]. When a simulation terminates, the corresponding output information that consists of final results, the register file content, the time for execution and state of the processor are stored and used to classify the simulation. We exploit the following categories proposed in [10] [20] , to classify the simulation,

1. *Wrong Answer*: The application terminates normally but the results produced are not correct.
2. *Latent*: The application terminates normally and the results are correct but at the end of simulation the content of the register file is different from that of fault-free case.

3. *Effect-Less*: The application terminates normally and the results are correct, and the content of the register file is similar to that of fault-free case.

4. *Exception*: The processor detected the injected fault and generated an exception (e.g., invalid address exception).

5. *Timed-Out*: The application failed to terminate and produce results with a predefined time limit.

6. *Stalling*: The processor computed the expected results in a time greater than the time of fault-free case.

7. *Crashing:* The processor fails to terminate normally.

For a good comparison, we consider three models of the processor:

Base: a normal processor (without implementing any protection technique).

IRR: a fault tolerant model, where an In-Register Replication technique [17] is implemented. This technique has been chosen here because it tries to achieve a similar goal as our proposed technique.

SI: a fault tolerant model, where our proposed *Self-Immunity* technique is implemented.

Chapter-4

Simulation Results

As expected, our technique maintains very high levels of fault tolerance compared to the "Base" case. The self-immunity technique improves the register file integrity effectively by reducing largely the number of errors in each category. Also the number of errors reaches zero in some benchmarks. Since latent errors have no effect on the output of an application, they are less harmful. This means that we can safely add the "Latent" category to the "Effect-Less" category [20] since in both categories the final results are still completely correct. In this case the system fault coverage after implementing our technique reaches on average 93% and up to 100%.

Writing into a register: (self immunity case)

When writing a 26 bit value into a register then self immunity bit (load) set to '1' and the ECC value (encoded value) is generated by using encoder and stored in the upper six unused bits of the register. Hence, the data bits and its ECC bits are stored together in that register.

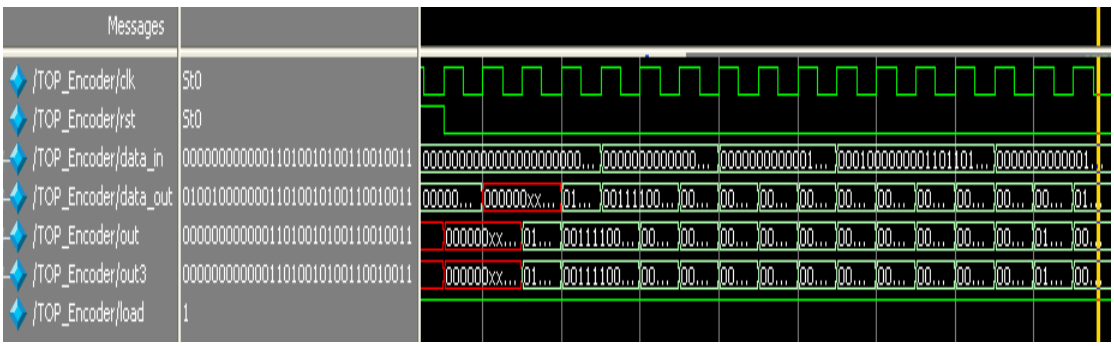

Figure-4.1: Test wave forms for Encoding block

Reading from a register:

The *self immunity bit (load)* is used to distinguish between a *Self-Immunity* case and a non *self-Immunity* case. If load ='1' then the data value and the corresponding ECC are decoded by using decoder.

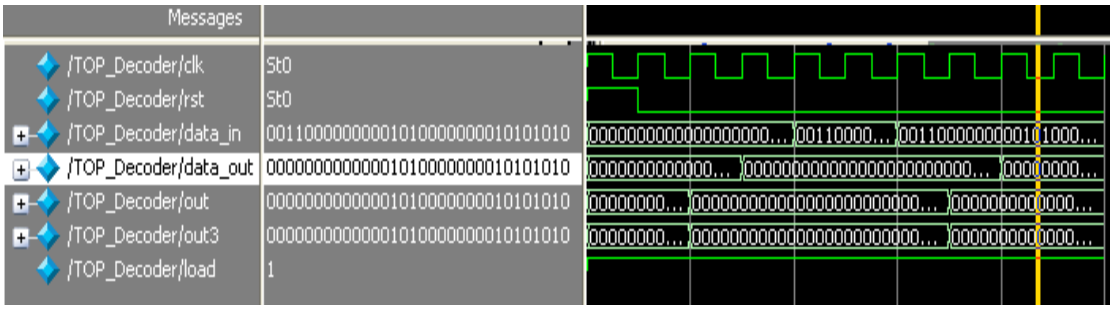

Figure-4.2: Test wave forms for Decoding block

(The above wave forms represent the intermediate values generated during simulation by encoder and decoder.)

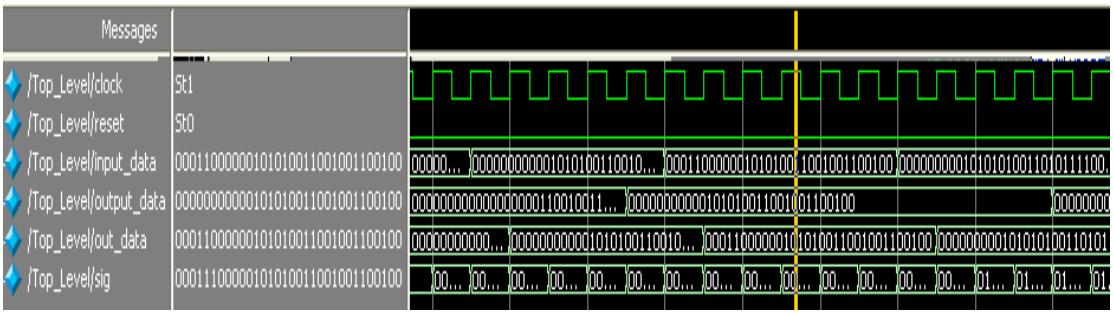

Figure-4.3: Test wave forms for Top-level block

(The above Wave form represents the actual input value and the output value.)

Effectiveness of our technique

In a full protection scheme, an ECC generation is performed with each *write* operation and similarly ECC checking is performed with each *read* operation. Our self immunity technique decides to protect the value depending if it is valid for *Self-Immunity*, then only it activates the ECC generator to compute the ECC. Otherwise, the ECC generation is skipped. Similarly, instead of always checking ECC, for every register *read* operation, our technique checks whether the ECC is being embedded in the register value or not and ECC checking is performed if embedded. On average 12% of the data will be stored in the register file without protection. As a result, our technique reduces *M* and it may lead to reduce the consumed power. As is when studying 32-bit architectures, 93% (on average) of the total *vulnerable intervals* are *vulnerable intervals* of valid register values for our technique. In other words, around 93% of *vulnerable intervals* will potentially be invulnerable. Thus, our technique guarantees to reduce the vulnerability of the register file considerably.

Potential Power Saving

In our proposed architecture, "over-26-bit" register values are neither encoded nor decoded and consequently the encoding and decoding operations are not performed with each read and write operation as it happens in a full protection scheme. This may reduce the power consumption of our proposed architecture because the encoding and decoding operations are performed only in the case of "26-bit" register values. Figure-3.5 demonstrates that on average 12% and 13% of the total number of read and writes operations, respectively, are occurred in the case of "over-26-bit" register values. As a result, our proposed technique may consume less power because the encoder and decoder are lesser times accessed.

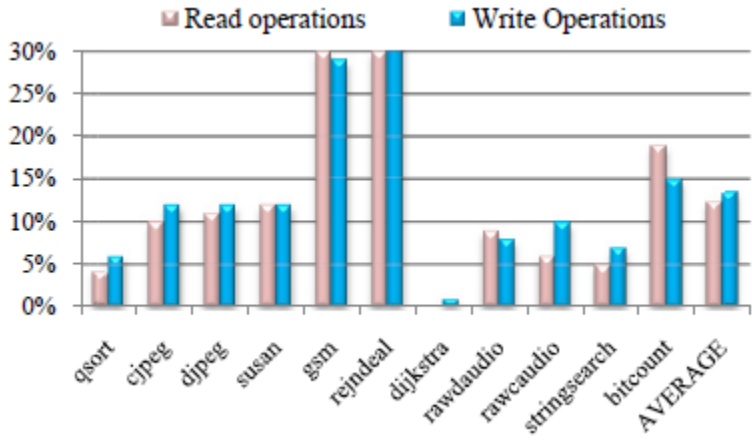

Figure-4.4: The percentage of read and write operations in the case of "over-26-bit" register values [24]

Since the input of the deployed encoder in our architecture is 26 bits instead of 32 bits, it generates 5 parity bits instead of generating 6 parity bits. The used decoder in our architecture takes 31 bits (26 bits for data + 5 bits for ECC) as input instead of taking 38 bits as input. In other words, our proposed technique uses a less complex encoder and decoder. This may also lead to a further saving in the terms of the power consumption. Finally, our proposed technique decreases the total number of bits of a protected register from 38 bits to 33 bits and as a result the consumed switching power is lower. In other words, the power saving is mainly due to the usage of a less complex ECC generator and checker, fewer ECC operations, and the absence of additional storage for ECC.

Chapter-5

FPGA Implementation

To investigate the advantages of using our technique in terms of area overhead against “Fully ECC” and against the partially protection techniques, we implemented and synthesized for a Xilinx XC3S500E different versions of a 32-bit, 32-entry, dual read ports, single write port register file. Once the functional verification is over, the RTL model is taken to the synthesis process using the Xilinx ISE tool. In synthesis process, the RTL model will be converted to the gate level net list mapped to a specific technology library. Many different devices were available in the Xilinx ISE tool in this Spartan 3E family,. In order to synthesis this design the device named as “XC3S500E” has been chosen and the package as “FG320” with the device speed such as “-4”.

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.

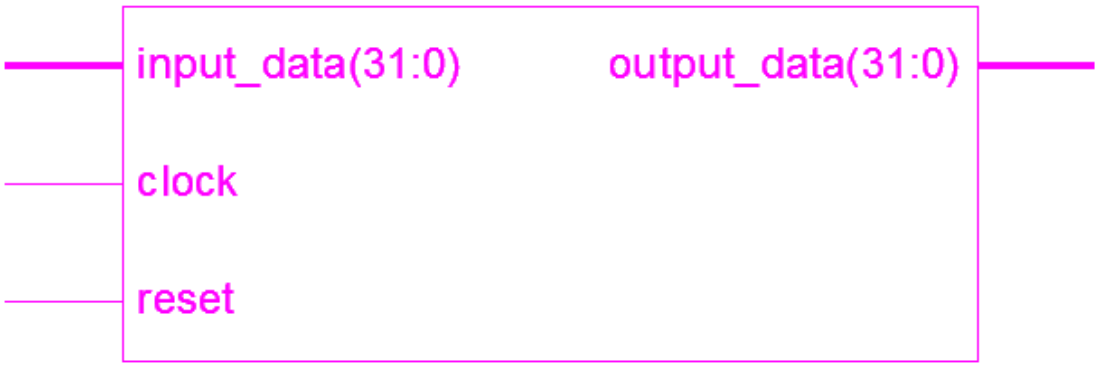

Figure 5.1: Schematic with Basic Inputs and Output

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

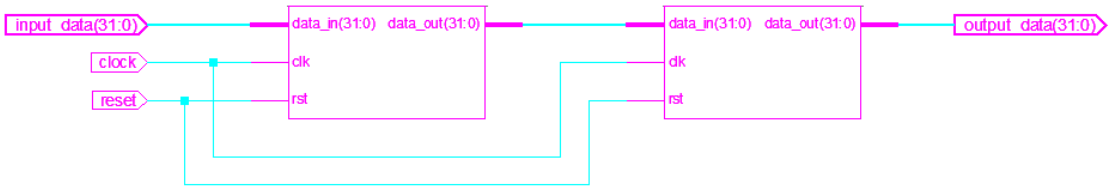

Figure 5.2: Blocks inside the Top Level Design

The internal blocks available inside the design includes encoder, decoder which was 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.

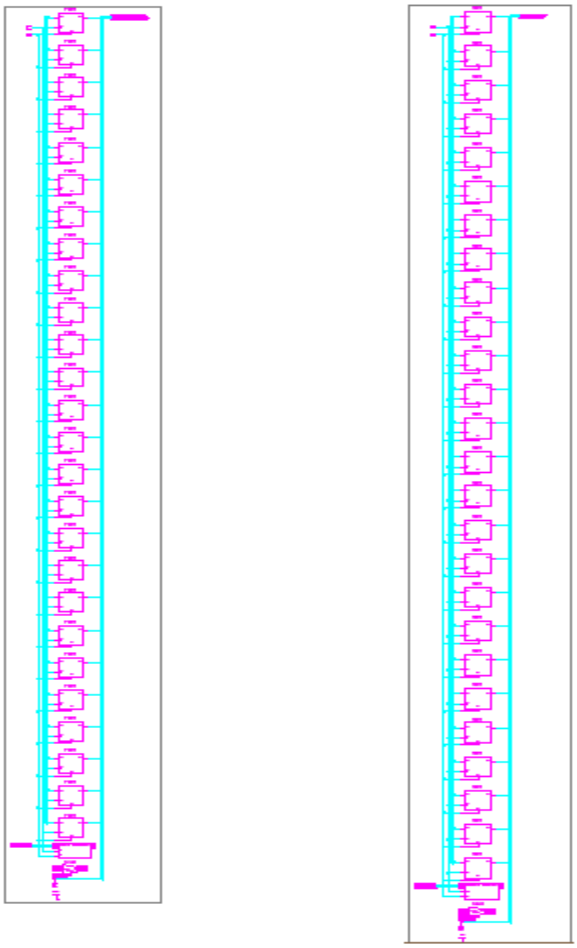

Figure 5.3: Blocks inside the Developed Encoder, Decoder Design

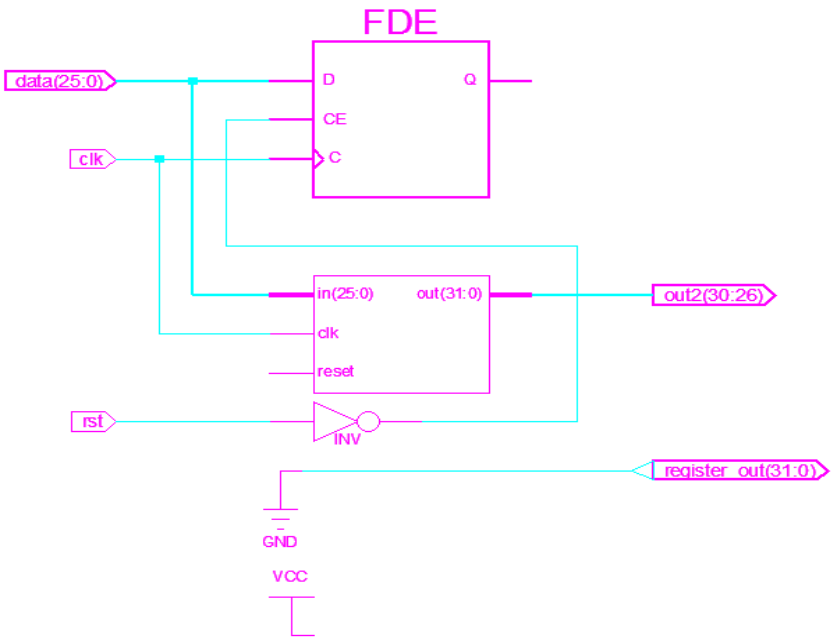

Figure 5.4: ECC Block inside the encoder Design

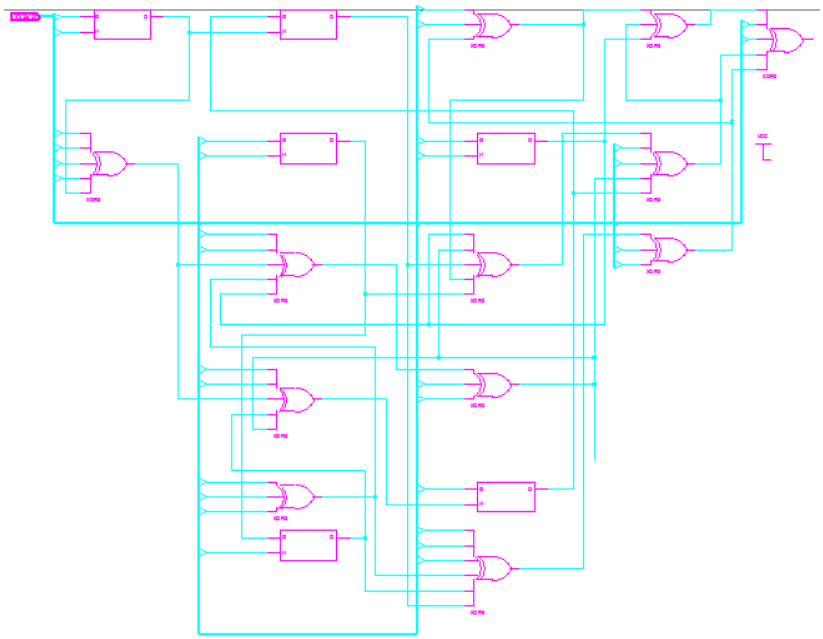

Figure 5.5: Encoder block inside the ECC Design

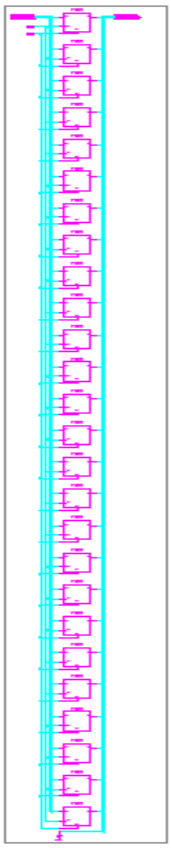

Figure 5.6: The decoder block

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
# Flip-flops/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 and the area occupation as well in the FPGA.

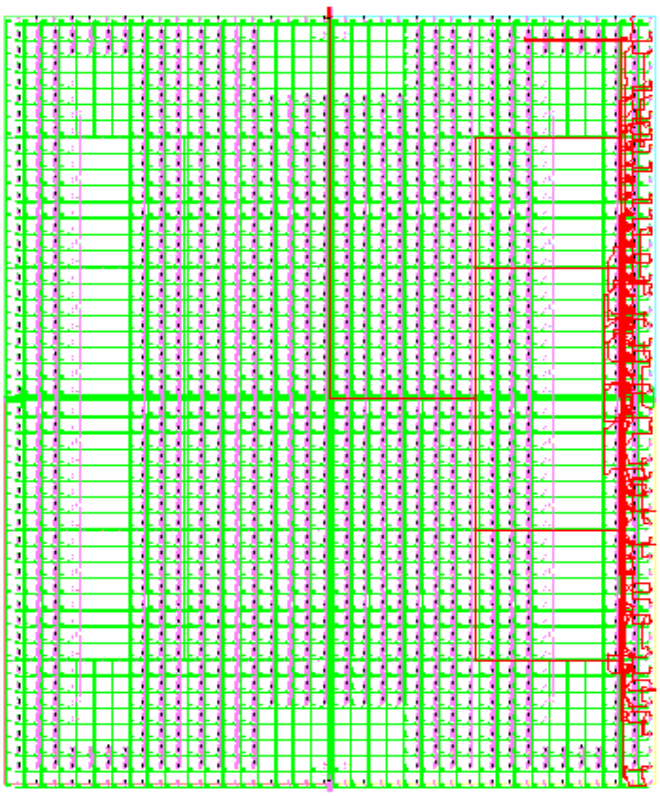

Figure 5.7: the area occupation of the Design

Chapter-6

Conclusion

Conclusion

For embedded systems under severe cost constraints, where power, performance, area and reliability cannot be simply compromised, we implemented a soft error mitigation technique for register files to improve the immunity to soft errors. Our experimental results on different embedded system applications demonstrate that our proposed *Self-Immunity* technique decreases the register file vulnerability effectively and achieves high system fault tolerance. Furthermore, our technique is generic so that 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.

Bibliography

[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 Microarchitecture (MICRO-36), pp.29- 40, 2003.

[4] T.J. Dell, “A whitepaper on the benefits of Chip kill-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 Aviral Shrivastava, “A Compiler Optimization to Reduce Soft Errors in Register Files,” in LCTES 2009.

[8] Jason A. Blome, Shantanu Gupta, Shuguang Feng, 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.

[10] M. Rebaudengo, M. S. Reorda, and M.Violante, “An Accurate Analysis of the Effects of Soft Errors in the Instruction and Data Caches of a Pipelined Microprocessor,” in DATE’03, pp. 602-607, 2003.

[11] I. Koren and C. M. Krishna, Fault-Tolerant Systems. San Mateo, CA: Morgan Kaufmann, 2007.

[12] MiBench (http://www.eecs.umich.edu/mibench/).

[13] T. Slegel et al, “IBM’s S/390 G5 microprocessor design,” in IEEE Micro, 19, pp. 12-23, 1999.

[14] M. Fazeli, A. Namazi, and S.G. Miremadi “An energy efficient circuit level technique to protect register file from MBUs and SETs in embedded processors,” in Dependable Systems & Networks 2009, pp.195–204, DNS’09.

[15] K. Walther, C. Galke and H.T. VIERHAUS, “On-Line Techniques for Error Detection and Correction in Processor Registers with Cross-Parity Check,” in Journal of Electronic Testing: Theory and Applications 19, pp.501-510, 2003.

[16] M. Spica and T.M. Mak, “Do we need anything more than single bit error correction (ECC)?,“ in Memory Technology, Design and Testing, Records of the International Workshop on 9-10, pp. 111– 116, 2004.

[17] M. Kandala, W. Zhang, and L. Yang, “An area-efficient approach to improving register file reliability against transient errors,” in Advanced Information Networking and Applications Workshops, AINAW '07, pp. 798–803, 2007.

[18] http://archc.sourceforge.net/.

[19] Jun Yan and Wei Zhang, “Compiler-guided register reliability improvement against soft errors,” in EMSOFT ’05, pp. 203–209, 2005.

[20] E. Touloupis, J.A. Flint, V.A. Chouliaras and D.D. Ward, “Efficient protection of the pipeline core for safety-critical processor-based systems,” in IEEE workshop on Signal Processing Systems Design and Implementation, pp. 188-192, 2005.

[21] Jongeun Lee and A. Shrivastava, “A Compiler-Microarchitecture Hybrid Approach to Soft Error Reduction for Register Files,” in Computer-Aided Design of Integrated Circuits and Systems, pp. 1018-1027, 2010.

[22] Jongeun Lee and A. Shrivastava, “Compiler-managed register file protection for energy-efficient soft error reduction,” in ASP-DAC, pp. 618–623, 2009.

[23] Riaz Naseer, Rashed Zafar Bhatti, and Jeff Draper, “Analysis of Soft Error Mitigation Techniques for Register Files in IBM Cu-08 90nm Technology,” in MWSCAS’06, pp. 515-519, 2006.

[24] Hussam Amrouch and Joerg Henkel, “Self Immunity Technique to Improve Register File Integrity against Soft Errors,” in 24th Annual Conference on VLSI Design, 2011.