Nowadays Embedded Systems have became a part of human life. The most important part of an embedded system is the embedded processor. The performance of embedded processor determines the performance of embedded system. An embedded processor is a Reduced Instruction Set Computer (RISC). A RISC processor uses load-store architecture, fixed length instructions and pipelining. In load-store architecture, load instruction reads data from memory and writes it to a register, data processing instructions process data available in registers and write result to a register, and store instruction copies data from register to memory. Pipelining is an implementation technique in which multiple instructions are overlapped in execution. The reduction of transistors size has allowed the increase of microprocessors speed and the decrease of their size and supply voltage, but at the cost of augmenting the incidence of faults. This reduction causes a higher rate of transient faults, commonly provoked by temporary environmental conditions such as electromagnetic interference, or cosmic or internal radiation. Even, radiation may now affect multiple locations. Also, changes in the manufacturing processes have increased the rate of permanent faults. This type of fault is produced by irreversible physical changes in a chip. Recently, intermittent faults have emerged as a new source of trouble in deep submicron integrated circuits. With the scaling of complementary metal-oxide-semiconductor (CMOS) technology to the submicron range, designers have to deal with a growing number and variety of fault types. In this way, intermittent faults are gaining importance in modern very large scale integration (VLSI) circuits. The presence of these faults is increasing due to the complexity of manufacturing processes (which produce residues and parameter variations), together with special aging mechanisms. This work presents a case study of the impact of intermittent faults on the behavior of a reduced instruction set computing (RISC) microprocessor. We have carried out an exhaustive reliability assessment by using verilog-based fault injection. In this way, we have been able to modify different intermittent fault parameters, to select various targets, and even, to compare the impact of intermittent faults with those induced by transient and permanent faults.

KEYWORDS — RISC, Intermittent, Faults, Processor

I. INTRODUCTION

Nowadays Embedded Systems have became a part of human life. The most important part of an embedded system is the embedded processor. The performance of embedded processor determines the performance of embedded system. An embedded processor is a Reduced Instruction Set Computer (RISC). A RISC processor uses load-store architecture, fixed length instructions and pipelining. In load-store architecture, load instruction reads data from memory and writes it to a register, data processing instructions process data available in registers and write result to a register, and store instruction copies data from register to memory. Pipelining is an implementation technique in which multiple instructions are overlapped in execution. The reduction of transistors size has allowed the increase of microprocessors speed and the decrease of their size and supply voltage, but at the cost of augmenting the incidence of faults. This reduction causes a higher rate of transient faults, commonly provoked by temporary environmental conditions such as electromagnetic interference, or cosmic or internal radiation. Even, radiation may now affect multiple locations.

Also, changes in the manufacturing processes have increased the rate of permanent faults. This type of fault is produced by irreversible physical changes in a chip. Recently, intermittent faults have emerged as a new source of trouble in deep submicron integrated circuits. With the scaling of complementary metal-oxide-semiconductor (CMOS) technology to the submicron range, designers have to deal with a growing number and variety of fault types. In this way, intermittent faults are gaining importance in modern very large scale integration (VLSI) circuits. The presence of these faults is increasing due to the complexity of manufacturing processes (which produce residues and parameter variations), together with special aging mechanisms. This work presents a case study of the impact of intermittent faults on the behavior of a reduced instruction set computing (RISC) microprocessor. We
have carried out an exhaustive reliability assessment by using verilog-based fault injection. In this way, we have been able to modify different intermittent fault parameters, to select various targets, and even, to compare the impact of intermittent faults with those induced by transient and permanent faults.

The proposed design can find its applications in automation, high configured robotic work-stations such as, portable pong gaming kits, smart phones, Vender Machines, ATMs, bottling plant, etc. Bottles start filling from the right side and boxes start to move from the left side. Here four tracks of bottles are used simultaneously therefore packing is made of four bottles. When bottle reaches to the fourth position, box moves to the first position. After that, bottle is dropped in the box and hence, box moves one position ahead. In this way, when box is at the fifth position, signal ‘lb’ is set to ‘1’ indicating to lift the box.

The main purpose of the study presented in this work is to analyze the influence of intermittent faults in the behavior of a RISC microprocessor. The system target is the Plasma microprocessor. It has a 64-bit microprocessor without interlocked pipeline stages (MIPS) architecture with a four-stage pipeline. The Verilog model of Plasma is described at RT and logic abstraction levels.

To exercise the main elements of the microprocessor (memory, registers, buses, arithmetic and logic unit (ALU) and control unit (CU)), the bubble sort sorting algorithm has been used. In this way, we have injected intermittent faults into the storage elements (the register bank, and the random-access memory (RAM)), the buses, and the combinational logic of the ALU and CU.

II. PROPOSED SYSTEM

The design of 32-bit RISC processor incorporates various design blocks like Arithmetic Logic Unit (ALU), Accumulator, Program Counter (PC), Instruction Register (IR), Memory, Control Unit (CU), and additional logic.

The design incorporates some the following issues.

- It handles 32 bit data, 28 bit address
- Uses fixed instruction format of length 32 bit.
- Size of opcode is of 4 bit, handling 15 instructions.
- Has 256 memory locations
- 32-bit registers (IR, ACC)
- Implements 2-staged pipelining i.e. overlap of fetch and execute cycles.
- Has two addressing modes, Register addressing and memory addressing modes
- No interrupts and No conditional branches
- Data that it handles is unsigned integer type.

The next sections of this chapter presents the schematic diagrams of individual modules, explanation of their function, flow chart to implement the function, VHDL description of the individual modules. In the last section the top order module Block diagram, functioning, VHDL description of the top module is given. Let us consider an instance when some information is stored in the memory. Now when the system is switched on, CPU is initialized. In order to fetch an instruction, as a result the program goes to the location in the memory that is pointed out by the program counter. After some instance, the instruction from the memory is put on the data bus. This cycle is called the instruction fetch cycle. The instruction is now available at the data bus. At next instance; the instruction is loaded into the instruction register. This is called the instruction load cycle. In this cycle the 4 msb’s of the instruction are separated and put in the opcode register and are loaded to control unit as well as ALU. The rest of the bits are sent out as Irout. The outputs of the instruction register and the program counter are connected to a mux. During the negative edge of the fetch signal, the output of the instruction register is selected, while the output from the program counter is selected during the positive edge of fetch cycle. Now when the fetch signal goes low the mux selects the output from the instruction register and it points to the location of the operand. Now the operand present in the location is placed on the data bus. After an instruction is fetched the program counter is incremented. It points to the next location. Now the operand is available at the ALU. The operand is taken in by the ALU and operates on it. Now the result is available at acc1 at positive edge of exclel.
During the negative edge of exclk, the result at the Acc1 register is placed on the data bus, which is sent and loaded into the accumulator for any further operations. If the data has to be stored into the memory, then during this clock cycle, Rd and Wr has to be 0 & 1 respectively. As a result the accumulator is connected to the memory and the value in the accumulator is sent back to a location in the memory through a module named Buffer. A characteristic of RISC processor is their ability to execute one instruction per clock cycle.

III. IMPLEMENTATION PROCESS

Encoder

An -bit codeword $c$, which encodes a $k$-bit information vector $i$ is generated by multiplying the $k$-bit information vector with a $k \times n$ bit generator matrix $G$; i.e., $c = iG$.

EG-LDPC codes are not systematic and the information bits must be decoded from the encoded vector, which is not desirable for our fault-tolerant approach due to the further complication and delay that it adds to the operation. However, these codes are cyclic codes.

Note the identity matrix in the left columns.

$G = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1
\end{bmatrix}$

$G$ is the generator matrix for the (15, 7, 5) EG-LDPC in systematic format

In this figure $i = (i_0, i_1, i_2, \ldots , i_6)$ is the information vector and will be copied to $(c_0, \ldots , c_6)$ bits of the encoded vector $c$, and the rest of encoded vector, the parity bits, are linear sums (XOR) of the information bits. If the building block is two-inputs gates then the encoder circuitry takes 22 two-input XOR gates. Once the XOR functions are known, the encoder structure is very similar to the detector structure shown in Fig. 3.

$$i_0, i_1, i_2, i_3$$

*$C_0, C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8, C_9*$

FIG 3: Structure of an encoder circuit for the (15, 7, 5) EG-LDPC code; $i_0$ to $i_6$ are 7-bit information vector. Each of the XOR gates generates one parity bit of the encoded vector. The codeword consists of seven information bits followed by eight parity bits.

Fault Secure Detector

The core of the detector operation is to generate the syndrome vector, which is basically implementing the following vector-matrix multiplication on the received encoded vector $C$ and parity-check matrix $H$:

$$s = cH^T$$

Therefore each bit of the syndrome vector is the product of $C$ with one row of the parity-check matrix. This product is a linear binary sum over digits of $C$ where the corresponding digit in the matrix row is 1. This binary sum is implemented with an XOR gate. Fig. 4 shows the detector circuit for the (15, 7, 5) EG-LDPC code. Since the row weight of the parity-check matrix is $\rho$, to generate one digit of the syndrome vector we need a $\rho$-input XOR gate, or $(\rho - 1)$-input XOR gates. For the whole detector, it takes $n(\rho - 1)$ 2-input XOR gates. Table II illustrates this quantity for some of the smaller EG-LDPC codes. Note that implementing each syndrome bit with a separate XOR gate satisfies the assumption of Theorem I of no logic sharing in detector circuit implementation.

An error is detected if any of the syndrome bits has a nonzero value. The final error detection signal is...
implemented by an OR function of all the syndrome bits. The output of this input OR gate is the error detector signal.

![Fault-secure detector for (15, 7, 5) EG-LDPC code.](image1)

**Corrector:**

One-step majority-logic correction is a fast and relatively compact error-correcting technique. There is a limited class of ECCs that are one-step-majority correctable which include type-I two-dimensional EG-LDPC. In this section, we present a brief review of this correcting technique. Then we show the one-step majority-logic corrector for EG-LDPC codes.

**One-Step Majority-Logic Corrector:**

One-step majority logic correction is the procedure that identifies the correct value of each bit in the codeword directly from the received codeword; this is in contrast to the general message-passing error correction strategy (e.g., [5]) which may demand multiple iterations of error diagnosis and trial correction. Avoiding iteration makes the correction latency both small and deterministic. This technique can be implemented serially to provide a compact implementation or in parallel to minimize correction latency.

This method consists of two parts:

1) Generating a specific set of linear sums of the received vector bits and

2) Finding the majority value of the computed linear sums.

The majority value indicates the correctness of the code-bit under consideration; if the majority value is 1, the bit is inverted, otherwise it is kept unchanged. The theory behind the one-step majority corrector and the proof that EG-LDPC codes have this property are available in [1]. Here we overview the structure of such correctors for EG-LDPC codes.

![Serial one-step majority logic corrector structure to correct last bit (bit14th) of 15-bit (15,7,5) EG-LDPC code.](image2)
row density of EG-LDPC codes are $\rho$, each XOR gate that computes the linear sum has $\rho$ inputs. The single XOR gate on the right of Fig. 7 corrects the code bit $c_{n-1}$ using the output of the majority gate. Once the code bit $c_{n-1}$ is corrected the codeword is cyclic shifted and code bit $c_{n-2}$ is placed at $c_{n-1}$ position and will be corrected. The whole codeword can be corrected in $n$ rounds.

If implemented in flat, two-level logic, a majority gate could take exponential area. The two-level majority gate is implemented by computing all the product terms that have $[\gamma + 1/2]$ ON inputs and one

$$\left(\frac{\gamma}{\gamma + 1/2}\right)^{\text{input OR-term}}.$$ For example, the majority of 4 inputs, $a, b, c$, is computed with $3$ product terms and one 4-input OR-terms as follows:

Majority $(a, b, c) = ab + ac + bc$. 

\begin{equation}
\text{(2)}
\end{equation}

**Majority Circuit Implementation:**

Here we present a compact implementation for the majority gate using Sorting Networks [6]. The majority gate has application in many other error-correcting codes, and this compact implementation can improve many other applications.

A majority function of $\gamma$ binary digits is simply the median of the digits (where we define the median of an even number of digits as the $\gamma/2 + 1$ st smallest digit).

To find the median of the $\gamma$ inputs, we do the following:

1) Divide the $\gamma$ inputs into two halves with size $\gamma/2$;
2) Sort each of the halves;
3) The median is 1 if for $i = 1, 2, \ldots, \gamma/2$ the $i$ th element of one half and the $(\gamma/2 + 1 - i)$ th element of the other half are both 1.

We use binary Sorting Networks [6] to do the sort operation of the second step efficiently. An $n$-input sorting network is the structure that sorts a set of bits, using 2-bit sorter building blocks. Fig. 6(a) shows a 4-input sorting network. Each of the vertical lines represents one comparator which compares two bits and assigns the larger one to the top output and the smaller one to the bottom [see Fig. 6(b)]. The four-input sorting network, has five comparator blocks, where each block consists of two two-input gates; overall the four-input sorting network consists of ten two-input gates in total.

**FIG 6:** (a) Four-input sorting network; each vertical line shows a one-input Comparator. (b) One comparator structure.

To check the condition in the third step, we use $\gamma/2$ two-input AND gates followed by a $\gamma/2$ -input OR gate. Fig. 9 shows the circuit implementing the above technique to find the median value of 8 bits. It has two $\gamma/2$ -input (four-input) sorting networks followed by combinational circuitry, consisting of four two-input AND gates and a four-input OR gate, which can be implemented with three two-input OR gates. Therefore in total an eight-input majority gate implemented with sorting networks take 27 two-input gates; in contrast, the two-level implementation of this majority gate takes

$$\binom{8}{5} = 56$$

five-input ANDn gates and one 56-input OR gate.

**Serial Corrector**
As mentioned earlier, the same one-step majority-logic corrector can be used to correct all the \(n\) bits of the received codeword of a cyclic code. To correct each code-bit, the received encoded vector is cyclic shifted and fed into the XOR gates as shown in Fig. 5. The serial majority corrector takes \(n\) cycles to correct an erroneous codeword. If the fault rate is low, the corrector block is used infrequently; since the common case is error-free code words, the latency of the corrector will not have a severe impact on the average memory read latency. The serial corrector must be placed off the normal memory read path. This is shown in Fig. 7. The memory words retrieved from the memory unit are checked by detector unit. If the detector detects an error, the memory word is sent to the corrector unit to be corrected, which has the latency of the detector plus the round latency of the corrector.

**Parallel Corrector**

For high error rates [e.g., when tolerating permanent defects in memory words as well (see Section VIII)], the corrector is used more frequently and its latency can impact the system performance. Therefore we can implement a parallel one-step majority corrector which is essentially \(n\) copies of the single one-step majority-logic corrector. Fig. 1 shows a system integration using the parallel corrector. All the memory words are pipelined through the parallel corrector. This way the corrected memory words are generated every cycle. The detector in the parallel case monitors the operation of the corrector, if the output of the corrector is erroneous, the detector signals the corrector to repeat the operation. Note that faults detected in a nominally corrected memory word arise solely from faults in the detector and corrector circuitry and not from faults in the memory word. Since detector and corrector circuitry are relatively small compared to the memory system, the failure rate of these units is relatively low.

Assuming our building blocks are two-input gates, the number of \(\rho\)-input parity-check sums will require \(\gamma \times (\rho - 1)\) two-input XOR gates. The size of the majority gate is defined by the sorting network implementation. Table II shows the overall area of a serial one-step majority-logic corrector in the number of two-input gates for the codes under consideration. The parallel implementation consists of exactly \(n\) copies of the serial one-step majority-logic corrector.

**Bus correction module**

represents class of series .

<table>
<thead>
<tr>
<th>Example:</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>101</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0000 (No Change)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>0001 (No Change)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>2+1 = 0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>3+3 = 0110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>4+3 = 0111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>5+3 = 1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>6+3 = 1001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>7+5 = 1100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**IV. CONCLUSION AND FUTURE WORK**

In this paper, FPGA implementations of fault secure encoder and decoder for memory applications. Using this architecture tolerates transient faults both in the storage unit and in the supporting logic (i.e., encoder, decoder (corrector), and detector circuitries).
advantage of the proposed architecture is using this detect-and-repeat technique we can correct potential transient errors in the encoder or corrector output and provide fault-tolerant memory system with fault-tolerant supporting circuitry. And also takes less area compared to other ECC techniques and in this architecture there is no need of decoder because we use systematic generated matrix.

**Future work**

Faults secure encoder and decoder for memory applications is to protect the memory and supporting logic from soft errors. The proposed architecture tolerates transient faults both in the storage unit and in the supporting logic. Scope for further work is instead of memory we use nano memory which provides smaller, faster, and lower energy devices which allow more powerful and compact circuitry.

**V. RESULTS & DISCUSSIONS**

References

Reduced Intermittent Faults Using Coding Techniques In Risc Processor


Authors Profile:

Ms. VEMULA SOWJANYA LAKSHMI is pursuing her M. Tech in Department of Electronics and Communication Engineering at Gandhiji Institute of Science & Technology, Jaggaiahpet.

Mrs. CH.LAVANYA is an Assistant professor in the Department of Electronics and Communication Engineering at Gandhiji Institute of Science & Technology, Jaggaiahpet. She has 6 years of teaching experience and published several papers.