

# Soft-error Mitigation at the Architecture-Level Using Berger Codes and Instruction Repetition

E. J. Ossi, *Student Member*, D. B. Limbrick, *Student Member IEEE*, W. H. Robinson, *Member, IEEE*, B. L. Bhuva, *Senior Member, IEEE*

**Abstract**— Soft-error mitigation using design techniques at the architecture-level can overcome the limitations of process and circuit-level mitigation techniques in nanometer technologies. Two implementations are presented that enable error recovery in an arithmetic logic unit (ALU). One implementation achieves high reliability without regard to performance penalty, and the other achieves an improvement in reliability with the error rate having no effect on performance. The merits and simulation results of these implementations are discussed.

**Index Terms**— Single Event, Arithmetic Logic Unit, Soft Error Mitigation, Berger Code

## I. INTRODUCTION

Advanced integrated circuits (ICs) with reduced operating voltages and higher transistor densities exhibit increased sensitivity to soft errors. As such, soft errors have become a major reliability problem for military, space, and commercial electronic systems [1]. For older technologies, hardening against single events (SE) was achieved through process modifications to reduce the charge collected at a circuit node [2]. As the minimum feature sizes on ICs reached sub-micron dimensions, such approaches became less cost-effective, and the circuit designs were used to eliminate the single-event transients from the circuit [3-5]. Both of these approaches have been adequate to mitigate soft-errors for technology nodes up to 130 nm. However, in advanced technologies beyond the 130 nm technology node, these approaches have become ineffective against single events due to lower nodal capacitances, lower supply voltages, and close proximity of devices to each other [6]. Lower capacitances and supply voltages have resulted in very low charge requirements to cause an upset, while close proximity of devices has resulted in multiple devices collecting charges due to a single ion hit. These factors have resulted in a very complex response to single-events for advanced IC designs [7] necessitating higher-level approaches to manage the soft errors within a system.

Such architecture-level design approaches can provide a means of error recovery by monitoring the data transfer among

architecture-level blocks and eliminating erroneous data for improved system performance. In this paper, two architecture-level approaches are presented to handle the incorrect data generated by the presence of the soft errors. The performance penalty for each of the techniques is presented in terms cycles per instruction (CPI), clock frequency, and the number of logical units required.

## II. BACKGROUND

Most digital circuits can be viewed as sequential circuits made of latches and combinational circuits as shown in Fig. 1. However, most circuits do not have the error-detection module shown. A particle strike anywhere in the circuit may yield an incorrect output captured in the output latches. Circuit-level mitigation techniques use multiple copies of the same circuit or increased nodal capacitances to mitigate soft errors with significant area, speed, and power penalties. Architecture-level mitigation approaches, on the hand, seek to impose very little penalty with comparable soft error performance.

The first step for mitigation is detection of the error. Most approaches use additional circuits to relate input data to output data and detect any discrepancies resulting from particle strikes. These approaches compute a check value based on the circuit inputs and compare it to a check value computed from the circuit outputs. Many encoding algorithms have been proposed to compute the check values that relate the input data to output data with only a few check bits [8]. For example, a Berger check prediction (BCP) for a self-checking ALU that can detect errors in both logic and arithmetic operations was

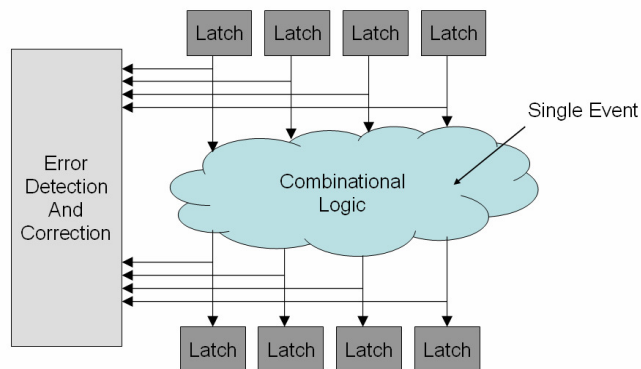


Fig. 1. Abstracted view of sequential logic components

Manuscript received March 11, 2009. This work was supported in part by NSF CAREER Award CCF-0747042.

E. J. Ossi, D. B. Limbrick, W. H. Robinson, B. L. Bhuva are with the Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, TN 37235 USA (edward.j.ossi @ vanderbilt.edu, daniel.b.limbrick @ vanderbilt.edu, william.h.robinson @ vanderbilt.edu, bharat.bhuva @ vanderbilt.edu).

discussed in [9]. By using such error detection and check bits, hardware requirements can be significantly reduced compared to a dual or triple mode structure, or a two-rail checker [10]. In this paper, Berger code is used to demonstrate the effectiveness of proposed architecture-level techniques. However, any of the numerous algorithms in the literature may be used for varying degrees of effectiveness and implementation cost [9,11].

The original Berger code proposed an encoding scheme to create a check symbol using the number of 0's in the data word represented as a binary number. The check symbol is calculated based on the number of 0's and 1's in the data word. The comparison of input and output check bits is carried out using a formula associated with the specific operation.

For example, when adding two numbers  $X$  and  $Y$ , let  $X=1001$  and  $Y=1010$  with  $cin=0$ . The sum is  $S=0011$  with  $cout = 1$ . The internal carry bits are  $C=1000$ . The number of zeroes for each is  $Sc=2$ ,  $Xc=2$ ,  $Yc=2$ , and  $Cc=3$ . The BCP formula for addition is  $Sc=Xc + Yc - Cc - cin + cout$ , or  $Sc = 2 + 2 - 3 + 1$ . This is equal to 2, which is equal to the number of zeroes in the result of the addition operation [10]. The hardware required for the Berger check symbol calculator is shown in Fig. 2.

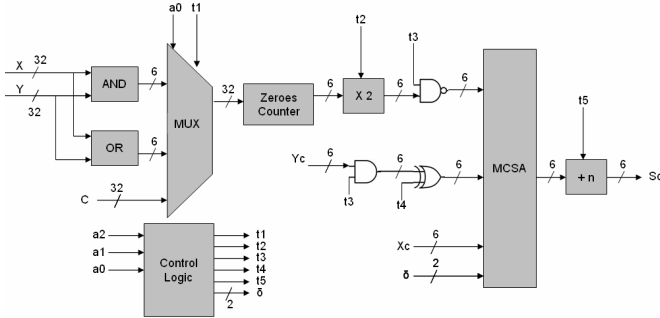


Fig. 2. Block Diagram of Berger Check Symbol Calculator

One drawback of the BCP algorithm discussed above is that it can only detect unidirectional errors. Unidirectional errors imply all errors are transitions from a logic '1' to a logic '0' (or all errors are transitions from logic '0' to logic '1'). If both a '1' and a '0' are flipped in the data word, the error will not be detected [11] and other algorithms must be employed that can detect bidirectional errors

The check bit length  $k$  is given by  $k = \log_2(n+1)$ , where  $n$  is the number of bits in the original data word [10]. For example, a 32-bit ALU would require a 6-bit Berger Check Symbol.

### III. IMPLEMENTATION

#### A. Implementation

At the architecture-level, the main difficulty is developing a technique to eliminate the soft error once it is detected. The approaches to error correction mostly involve overwrite or recalculation. For overwrite, corrupted bit(s) are identified and

correct values are overwritten over the incorrect bits. This can be accomplished by using complicated check codes that require large computation overhead and are costly, but can automatically correct the data. Another approach for overwrite is to use three copies of the hardware in parallel and vote on incorrect data. Approaches for recalculation essentially recalculate the incorrect data assuming that recalculated data will be correct. Recalculation approaches require very little overhead and can be as robust as overwrite approaches. For this project, the recalculation approach with two different implementations is investigated. These two implementations differ in their basic approach to error correction. The first approach only intervenes when an error is detected and repeats the instruction; the second approach always repeats the instruction without any penalty to the operating frequency. The first approach will be better suited to an environment where the number of errors expected is very low, while the second approach will be better for an environment where the number of soft errors expected is high. Details of both the implementations are given below.

#### B. Need-based-Intervention Implementation

This implementation repeats the instruction during which the soft error occurred. The block diagram is shown in Fig. 3. The Berger Check Calculator block is responsible for raising a flag whenever a soft error is detected. Upon detection of the soft error, the clock to the entire system is suspended for one clock cycle. This results in all registers holding their values for an additional clock cycle. This effectively repeats the previous instruction until the Single Event Transient (SET) pulse has dissipated. If the SET pulse is longer than one clock cycle, the clock suspension will last as long as the SET pulse affects the output data.

The Clock Manager block receives the system clock and provides local clock signals to the sub-circuit being hardened. For the present case, the sub-circuit is the ALU with all the associated input and output registers. The Clock Manager holds the clock only when a soft error is detected resulting in insignificant penalty on performance as long as the number of soft errors experienced by the design is low.

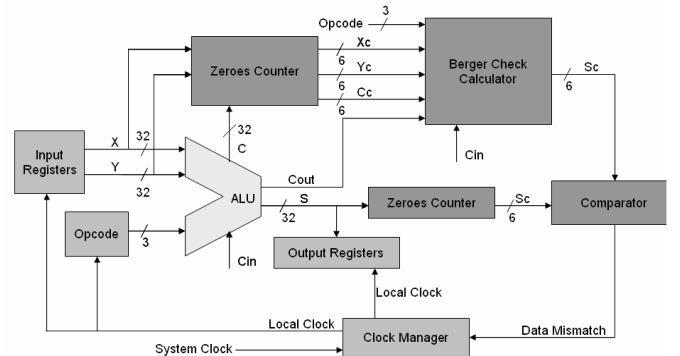


Fig. 3. Block Diagram of Need-Based-Intervention Implementation

### C. Always-Intervene Implementation

This implementation exploits temporal redundancy by running every ALU instruction twice in one clock cycle as shown in Fig. 4. The instruction executes once on the positive edge and once on the negative edge of the clock. In the first half clock cycle, the ALU performs the operation concurrently with the Berger Check Calculator and determines whether the operation executed as intended (absent an error). In the second half of the clock cycle, the ALU repeats the same operation and stores the result. If an error occurs in the first half of the clock cycle, the circuit will latch the result from the second half of the clock cycle. In all other cases, the ALU stores the result from the first half of the clock cycle. The basic assumption is that the soft error causing transients are shorter than half the clock period. Recent papers have shown that the number of short single-event pulses is orders of magnitude higher than that for longer ones for heavy-ion exposure [12, 13]. For such cases, most of the short errors will be detected and corrected. However, errors due to SET pulse longer than half a clock cycle may still get through the system.

The overriding circuitry was designed with a multiplexer and a series of registers. In order to run an instruction twice, two sets of input registers and output registers were used; one loads on the positive edge and the other loads on negative edge. The multiplexer determines which output register to use as the ALU output. The detection of the error is similar to the one described in need-based-intervention implementation.

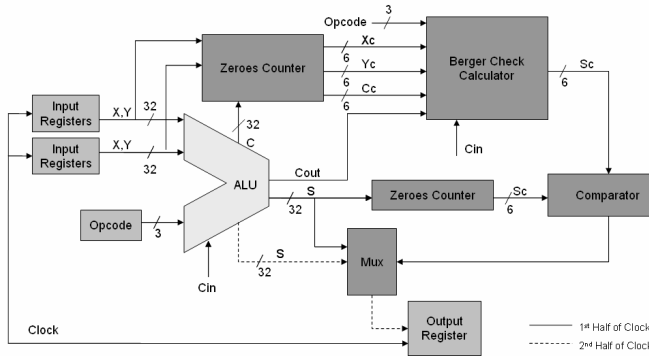


Fig. 4. Block Diagram of Always-Intervene Implementation

### IV. SIMULATION METHOD

A VHDL register transfer level (RTL) model was created for both designs and implemented in an Altera DE2 Development and Education Board which uses a Cyclone II Field Programmable Gate Array. The two BCP ALU designs were synthesized using Quartus II version 8.1 and the device targeted is EP2C35F672C6 from the Cyclone II family [14].

To simulate fault injection in our RTL model, XOR gates were used with the targeted node and fault injection signature as inputs using the ModelSim gate level simulation tool [14]. Faults were injected one at a time into each node for every possible instruction. The nodes were chosen based on their transparency to the output to attempt to eliminate logical

masking. Logic masking is the failure of an SET to cause an upset because it doesn't have a logical path the output. This provides a worst case fault injection profile limited only by a fault not occurring during a sensitive window and being latched..

The duration of each fault injected was varied using the IEEE.MATH\_REAL Library UNIFORM function to create a pseudorandom pulse width that varied between 0 and 125% of the clock period. The UNIFORM function was also used to randomly vary the position of the fault pulse with respect to the clock edge. A second ALU was run simultaneously to determine the functionally correct execution. At the end of each trial the results of each instruction were recorded and compared to the original results data. This data was used to quantify the inherent vulnerability of the ALU by determining a percentage of faults injected to errors recorded. The same procedure was performed on the BCP ALU designs. A similar fault injection method can be found in [15].

### V. RESULTS AND DISCUSSION

The implementations were compared based on the impact of their designs on maximum clock frequency and the number of logic elements required. These were obtained from the Quartus II fitter summary and timing analyzer and are shown in Table 1. The clock frequency is decreased for the Need-Based-Intervention because the BCP requires the carry inputs from the ALU before it can compute the check symbol. The clock frequency in the Always-Intervene case is decreased because the result for the first half of the clock cycle is latched at the falling edge. This requires the operation be completed in half a cycle. A triple modular redundancy (TMR) scheme using three ALUs and a voting logic block would provide error detection and correction, but would require more logic blocks compared to the Need-Based-Intervention implementation. The loss in clock speed would have to be evaluated against the additional area and power used by the additional logic blocks would use. The number of logic elements required increases by 165% for Need-Based-Intervention implementation. The increase for Always-Intervene Implementation is 301%.

TABLE I  
SUMMARY OF SYNTHESIS RESULTS

	Maximum Switching Speed (MHz)	Number of Logic Elements
Unhardened ALU	40.88	278
Need-Based-Intervention	38.66	737
Always Intervene	30.24	1117

To further compare the performance of the two implementations, the execution time was calculated using the equation:

$$Execution\ Time = Instruction\ Count * Cycles\ Per\ Instruction\ (CPI) * Clock\ Rate \quad (1)$$

The Need-Based-Intervention implementation does not affect the CPI during normal operation. When an error occurs, the instruction is retried, meaning the total execution time is the time it takes to run the instruction the first time, detect an error, and re-execute the instruction. The number of cycles required for each instruction can be calculated as  $(2 + N)$ , where 2 cycles are required under normal operation to execute each instruction, and  $N$  is the number of successive errors (each error adding a cycle to the execution time). For every error detected and corrected, the execution time increases by 25.9 ns. The Always-Intervene implementation uses the second half of the clock cycle to calculate a backup result. Therefore, the CPI = 2 which is the same in normal operation as it is when an error is detected.

The calculation for execution time requires a fixed instruction count in order to compare the different implementations. For 10,000 instructions and no fault injection, the execution time increase for the Need-Based-Intervention implementation is 5.7%, and the increase for Always-Intervene implementation is 35.2% compared to the unhardened ALU.

The total number of instructions run for the fault injection simulation was 10,404. The percentage of injected faults to errors for the unhardened ALU was 12.6 %. These faults occurred during the sensitive window, the set-up-and-hold time of a latch. Faults that occurred in this window also caused the instruction to be repeated in the Need-Based-Intervention implementation. The Need-Based-Intervention approach did not show any errors, i.e., all injected errors were corrected. A fault injected into the Always-Intervene circuit resulted in an error 2.8 % of the time. The execution time for the Need-Based-Intervention case increased by 111.6% because of the number of additional cycles required to address the faults. The execution time for the unhardened ALU and Always-Intervene circuit were unaffected. Fault injection results are shown in Table 2.

TABLE 2  
SUMMARY OF FAULT INJECTION RESULTS

	Number of Injected Faults	Number of Observed Errors	Number of Total Instructions
Unhardened ALU	8118	1025	10404
Need-Based-Intervention	8118	0	10404
Always Intervene	8118	295	10404

The Always-Intervene implementation is vulnerable to SETs longer than half a clock cycle, since the error could be present on both halves of the clock cycle. This corresponds to a pulse width longer than 16.5 ns, which is half the the clock period of 33 ns. Given this sensitivity, this implementation is not advised for environments where the maximum SET pulse width is greater than the clock period. This would allow the SET to strike without causing an error on both the positive and negative edge of the clock.

Both implementations provide the capability of soft error

detection and correction. The vulnerability of the Need-Based-Intervention implementation is limited only by the inherent flaws in BCP circuits. The repeating of instructions eliminated the latching of soft errors, but at the cost of a performance penalty. In addition to this, there is an increased complexity in handling the instruction repetition. The repeating of the clock period would require external synchronization with the other components of the circuit. There would also have to be a watchdog timer to turn off the BCP circuit in the event of a static fault, otherwise the system would enter into an infinite stuck state.

The Always-Intervene implementation provides its single event protection at a reduced external complexity, but at a high performance penalty. The error rate required for the performance of the Always-Intervene implementation to surpass the Need-Based-Intervention implementation is approximately 55 errors per 100 instructions. While it does provide approximately a tenfold decrease in single event sensitivity, the increase in logic elements and the associated area and power increases is likely prohibitive and a Need-Based-Intervention method is better suited for most applications.

## VI. CONCLUSION

The theoretical ALU with Berger check prediction from [10] can detect all unidirectional errors in both logic and arithmetic operations. This was used as a proof of concept to present two architectural methods which provide single event upset detection and correction by analyzing and correcting the data as it passes between the latches and through the combinational logic. The first approach only intervened when an error was detected and repeated the instruction. The second approach always repeated the instruction on the falling clock edge and corrected the error without additional penalty to the circuit performance. The Need-Based-Intervention scheme corrected all injected faults, but at a performance penalty of  $(2 + N)$  cycles per error. The Always Intervene method corrected 97.2% of the faults, but suffered a reduction in performance due to a slowing of the clock. When compared to TMR, the Need-Based Intervention required less logical elements, but the Always-Intervene case required more elements due its latching of the result on both clock edges. Both implementations show the effectiveness of an architectural approach to recover from radiation-induced soft errors if process modifications or circuit techniques fail to prevent or correct an error

## REFERENCES

- [1] M. Santarini, "Cosmic radiation comes to ASIC and SOC design," EDN, 2005.
- [2] S. W. Fu, A. M. Mohsen, and T. C. May, "Alpha-particle-induced charge collection measurements and the effectiveness of a novel p-well protection barrier on VLSI memories," IEEE Transactions on Electron Devices, vol. 32, pp. 49–54, 1985.
- [3] M. P. Baze, J. C. Killens, R. A. Paup, and W. P. Snapp, "SEU hardening techniques for retargetable sub-micron digital libraries," 2002 Single Event Effects Symposium, Manhattan Beach, CA, 2002.

- [4] D. R. Alexander, D. G. Mavis, C. P. Brothers, and J. R. Chavez, "Design issues for radiation tolerant microcircuits in space," IEEE NSREC Short Course, 1996.
- [5] G. Anelli, M. Campbell, M. Delmastro, F. Faccio, S. Floria, A. Giraldo, E. Heijne, P. Jarron, K. Kloukinas, A. Marchioro, P. Moreira, and W. Snoeys, "Radiation tolerant VLSI circuits in standard deep submicron CMOS technologies for the LHC experiments: practical design aspects," IEEE Transactions on Nuclear Science, vol. 46, pp. 1690-96, 1999.
- [6] O. A. Amusan, A. F. Witulski, L. W. Massengill, B. L. Bhuvu, P. R. Fleming, M. L. Alles, A. L. Sternberg, J. D. Black, and R. D. Schrimpf, "Charge collection and charge sharing in a 130 nm CMOS technology," IEEE Transactions on Nuclear Science, vol. 53, pp. 3253-3258, 2006.
- [7] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," IEEE Transactions on Device and Materials Reliability, vol. 5, pp. 305-316, 2005.
- [8] S.S. Gorshe and B. Bose, "A self-checking ALU design with efficient codes," VLSI Test Symposium, 1996, Proceedings of 14th, 28 April-1 May 1996, pp.157 – 161.
- [9] D. Das and N. A. Toubia, "Synthesis of circuits with low-cost concurrent error detection based on Bose-Lin codes," Journal on Electronic Testing: Theory and Applications (JETTA), vol. 15, pp. 145-155, 1999.
- [10] J.C. Lo, S. Thanawastien, T.R.N Rao, M.S. Nicolaidis, "An SFS Berger check prediction ALU and its application to self-checking processor designs," Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 11, no. 4, April 1992, pp.525 – 540.
- [11] C. L. Chen and M. Y. Hsiao, "Error-correcting codes for semiconductor memory applications: A state-of-the-art review," IBM Journal of Research and Development, vol. 28, pp. 124–134, 1984.
- [12] B.L. Narasimham, B. L. Bhuvu, R. D. Schrimpf, L. W. Massengill, M. J. Gadlage, O. A. Amusan, W. T. Holman, A. F. Witulski, W. H. Robinson, J. D. Black, J. M. Benedetto, and P. H. Eaton, "Characterization of digital single event transient pulse-widths in 130-nm and 90-nm CMOS technologies," IEEE Transactions on Nuclear Science, vol. 54, pp. 2506-2511, 2007.
- [13] B. Narasimham, B. L. Bhuvu, W. T. Holman, R. D. Schrimpf, L. W. Massengill, A. F. Witulski, and W. H. Robinson, "The effect of negative feedback on single event transient propagation in digital circuits," IEEE Transactions on Nuclear Science, vol. 53, pp. 3285-3290, 2006.
- [14] <http://www.altera.com>
- [15] F. Lima, S. Rezgui, L. Carro, R. Velazco, and R. Reis, "On the Use of VHDL Simulation and Emulation to Derive Error Rates," Proc. Radiation Effects on Components and Systems Conf. (RADECS), 2001.