Logic Soft Errors in a Parallel CISC Decoder

Eric L. Hill, *Member, IEEE,* and Mikko H. Lipasti, *Senior Member, IEEE*

***Abstract*—The instruction decoder is one of the most complex and least regular logic structures in a modern processor that attempts to process multiple variable-length CISC instructions per cycle. This structure consumes a significant amount of area and is heavily utilized, making it vulnerable to logic soft errors. This paper analyzes a parallel decoder using gate-level modeling and statistical fault injection, and finds that the conventional single-event upset (SEU) approach is inadequate for modeling the effects of soft errors in this circuit. We also show that different sections of the decoder design have very different fault propagation characteristics, and discuss the suitability of various approaches for protecting such circuits.**

***Index Terms*—Combinational logic circuit fault tolerance, pipeline processing, parallel architectures, computer reliability**

I. Introduction

Transient faults in logic (also known as logic soft errors) are becoming an increasingly pressing problem for future designs. This paper performs a case study of a large, complex logic block (a parallel CISC instruction decoder) in order to characterize fault propagation in such circuits and better understand the most effective approaches for mitigating logic soft errors. Decoder logic was specifically targeted for characterization because the instruction decode stage within a typical microprocessor pipeline is highly utilized and complex (specifically in architectures using a variable length instruction set). Our experiments yield two significant results. First, our results show that the artifacts caused by logic soft errors cannot always be modeled by SEUs, which is an assumption commonly made by studies on reliability performed at higher levels of abstraction [1][2][3]. Second, transient faults originating in different sections of the decoder have different propagation characteristics. These characterization results can be used to understand which protection techniques would best suit complex logic blocks like instruction decoders, and also to improve the accuracy of fault models existing at higher levels of abstraction.

II. Decoder Design

The decoder implemented for this work decodes z80 instructions. The z80 architecture is commonly used for embedded micro-controllers, and is thus substantially simpler (in terms of ISA complexity) than architectures prevalent in the high performance general purpose microprocessor design space. While z80 is substantially simpler than x86, several commonalities do exist between the two ISAs. First, both x86 and z80 have variable length instructions [4][5]. Second, z80 was originally designed to be binary compatible with the 8080 ISA, an predecessor to x86 [5]. In addition to this, this decoder implementation is inspired by industry published descriptions of parallel decoder designs[6], mapping a single z80 instruction to potentially many RISC operations. For these reasons, we feel the decoder implemented for this study has a similar functional structure (and thus similar transient fault propagation characteristics) to a parallel x86 instruction decoder within a high performance microprocessor.

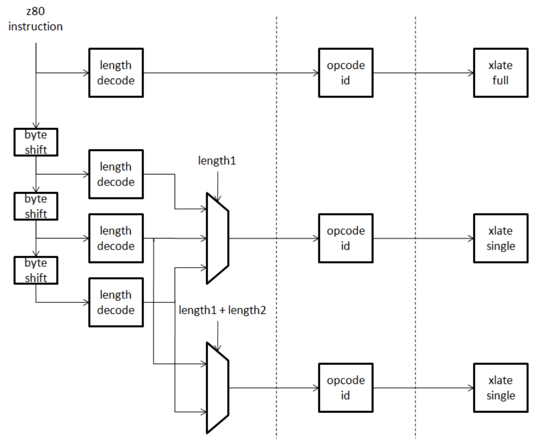

Fig. 1: z80 Decoder Block Diagram.

A block diagram of the parallel z80 decoder is shown in Figure 1. The decoder is capable of decoding up to three z80 instructions per cycle, and each instruction can be translated into up to 12 RISC operations. The decoder is pipelined into 3 stages: speculative length decode, opcode identification, and translation. The boundaries between each of the stages are denoted by the dotted lines in Figure 1. Decoding multiple instructions in parallel can be difficult for ISAs featuring variable instruction lengths. The most difficult aspect of parallel decode in these cases is identifying the start of each new instruction following the first instruction, because the starting point of each instruction is dependent on the length of its predecessors.

A naive approach for decoding variable length instructions in parallel would be to speculatively perform all decoding tasks from all possible starting positions for each instruction slot, and then select the correct decoded instructions to used based on the results of the first instruction (for the second slot), and the first and second instruction (for the third slot). While this would be a valid solution to the problem, it would require a significant amount of wasted computation, as six candidate instructions would be speculative decoded for just three slots in the best case. To circumvent this overhead, conventional parallel decoders adopt a strategy where the starting point of

Manuscript received January 8, 2010. This work was supported by NSF grant CCF-0702272.

Eric L. Hill was with University of Wisconsin–Madison when this work was performed. He is now with Intel Corp. (e-mail: eric.l.hill@intel.com).

Mikko H. Lipasti is with University of Wisconsin–Madison (e-mail: mikko@engr.wisc.edu).

each instruction slot is determined first (a process denoted as length decoding), allowing the other decoding tasks to be performed in a non-speculative manner. Dividing decode up in the manner described is advantageous in that the only speculative logic needed is in the length decode stage. Also, while there are six possible starting points for z80 instructions, in Figure 1 there are only four length decoders. The reason for this is that the results of speculative length decode starting from the third and fourth bytes can be used for instructions potentially in the second and third slots. The hardware for length decoding is relatively simple (as there are only four possible instruction lengths) and is thus an attractive candidate for speculation.

In the final two stages of decode, opcodes are first identified and then translated into RISC operations. Inspired by the decoder implementation described in [6], the translation logic for the first instruction slot can translate any instruction, while the translate logic for the other two slots (denoted as xlate single) can only handle z80 instructions which are translated into a single RISC operation. This means that the decoder can produce up to 14 RISC operations per cycle. The decoder also includes three valid bits for reach instruction slot, as well as 14 valid bits for each possible RISC operation produced.

III. Soft Error Characterization

In order to understand how transient faults propagate through the decoder, experiments were performed using gate-level statistical fault injection, following the methodology of [7]. The decoder described in the previous section was implemented in Verilog, and synthesized into a netlist of standard logic cells providing a benchmark circuit for analysis. In order to provide input stimulus to the circuit, a trace of z80 instructions was captured. During fault injection, transient pulses were injected at gate outputs and simulated over time to determine if a flip-flop at the end of the pipestage containing the gate would be corrupted. If this is the case, and the corrupted flip-flop was at the end of an intermediate pipestage, additional simulation was done to determine if the error would logically propagate to a primary output of the circuit. A diagram showing all of the possible outcomes that can occur is shown in Figure 2. Only the outcomes C (transient fault on a gate in the last pipestage corrupts a primary output) and F (transient fault on a gate in the last pipestage corrupts an intermediate flip-flop, and the result propagates to a primary output). All other cases are either timing window or logically masked. Our analysis does not consider the effects of electrical masking. In this experiment, 30,000 faults were injected into combinational gates exclusively.

IV. Results

Each slice of the pie chart shown in Figure 3 represents the fraction of injected faults which result in a particular outcome. All of the possible outcomes (and which node in Figure 2 they correspond to) that can occur when a fault is injected into combinational logic are shown in Table I. Of all the faults injected into combinational logic gates, only 7.1 % (the LSERR and ISLER cases) result in errors. Probably the most interesting attribute of this particular circuit is its

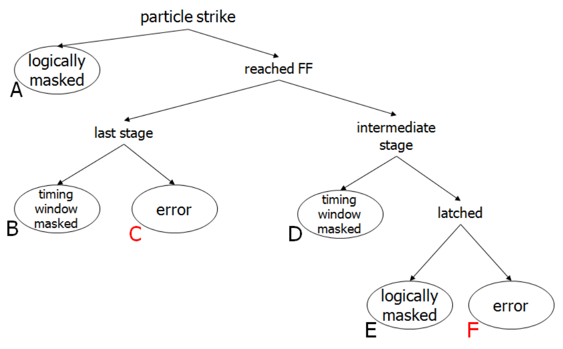

Fig. 2: Combination Logic Soft Error Fault Model.

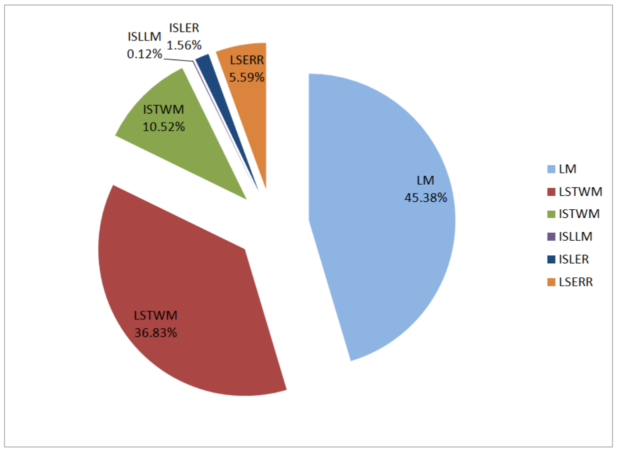

Fig. 3: z80 Logic Fault Outcome Breakdown.

logical masking behavior. Looking at Figure 3, 45.4 % of the injected faults result in the LM outcome, meaning they are logically masked before ever reaching a flip-flop. This result is particularly interesting when it is contrasted with the logical masking behavior of transients that do corrupt a flip-flop within an intermediate pipestage (the ISLER and ISLLM outcomes). Looking at these two cases, our results show that after an intermediate flip-flop is corrupted by a transient fault, there is a 92.8 % chance that error will eventually propagate to a primary output.

Outcome	Description
LSERR (C)	Error latched at output FF
LSTWM (B)	Timing window-masked at output FF
ISLER (F)	Error latched at interm. FF
ISLLM (E)	Logically masked after interm. FF
ISTWM (D)	Timing window-masked at interm. FF
LM (A)	Logically masked before interm. FF

TABLE I: Possible Outcomes for Combinational Logic Transient Fault Injection.

The cause of this distinctly different observed logical masking behavior stems from the functional structure of the decoder.

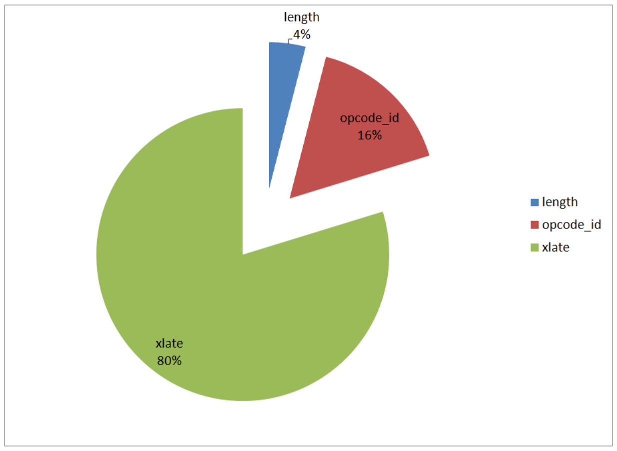

Fig. 4: z80 Error Origin.

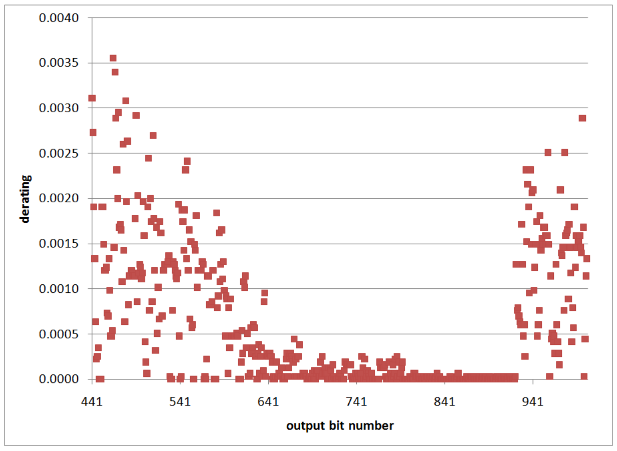

Fig. 5: z80 Decoder Output Derating per Bit.

As stated previously, the decoder logic is partitioned into length decode, opcode identification, and translate stages. In this design, many outputs of each internal stage directly influence the functionality of the succeeding stage. For example, opcode identification is performed in the second pipeline stage by looking at the opcode byte (whose position is determined by length decoding) as well as the instruction length (not necessary, but used to narrow the space of candidate opcodes). Because of this implementation, bit flips resulting from particle strikes in the length decode stage will likely result in incorrect opcodes being identified, and thus incorrect RISC operations being generated at the output of the decoder. A similar situation occurs in the translation stage. Any corruption in the opcode identifiers generated at the output of the second stage is likely to cause an error in translation. If the decoder was pipelined into a larger number of stages, the fraction of faults resulting in ISLLM outcomes (where the fault is still logically masked after corrupting an intermediate flip-flop) observed should intuitively increase. This is not the case in our design, because the ranks of flip-flops were placed at coarse functional boundaries.

In Figure 4, the errors observed during fault injection are broken down according to the pipeline stage the fault originated in. The pie chart shown in this figure has 3 slices: length (representing the length decode stage), opcode_id (representing the opcode identification stage), and xlate (representing the translation stage). From this chart, it is clear that the majority of errors observed originate in the translation stage of the decoder. This is the largest stage of the pipeline in terms of area, so statistically a larger fraction of faults are injected into its gates.

Another interesting phenomenon explored in this case study was the sensitivity of decoder output bits to particle strikes in combinational logic. One of the original stated motivations for this paper was to explore the choice made by many studies at higher abstraction levels to always model the artifacts of particle strikes as SEUs. This choice has two implicit assumptions: first, all output bits are equally likely to be corrupted, and second, that only a single bit is corrupted by a particle strike.

In order to examine the validity of the first assumption the derating per output bit, calculated by dividing the number of times each output bit was corrupted by the total number of faults injected, was measured and displayed in the scatter plot shown in Figure 5. Each point in this plot represents the probability that a particular output bit's value will be incorrect due to a transient fault. The y-axis in figure represents the derating per bit of each output while the x-axis is the number assigned to that particular node. Looking at this plot, it is clear that some output bits are significantly more likely to be corrupted than others.

As was stated previously, the implemented decoder design is capable of translating the instruction in the first slot into up to 12 RISC operations, and the instructions in the second and third slots into a single RISC operation. The majority of the instructions in the trace used as input stimulus for this experiment had short translations, which is why the bits on the left hand side of the graph (which correspond to the translation RISC operations for the first instruction slot) and the bits on the extreme right hand side of the graph (which correspond to RISC operations for the second and third instruction slot) have higher derating values.

In addition to looking at which output bits were most likely to be corrupted, the number of output bits corrupted simultaneously by a single injected fault was also studied. The pie chart shown in Figure 6 shows a breakdown of the observed errors in the decoder classified by fault origin (xlate, opcode_id, length) and how many bits were simultaneously corrupted (s - single, m - multiple). Looking at Figure 6, nearly 60% of the injected faults result in a single output being incorrect, with a vast majority of those cases originating in the translate stage. This case is denoted by the slice labeled xlate_s in the pie chart. Apart from this, multi-bit errors comprise 40% of the overall errors observed, including the majority of cases originating from the length decode and opcode identification stages.

The histogram shown in Figure 7 represents a breakdown of how many output bits are incorrect in each multi-bit error case. The individual cases where different numbers of bits flip are represented on the x-axis, while the stacked bars indicate the fraction of total multi-bit output errors in that case originating from a particular pipeline stage. From this histogram it is clear

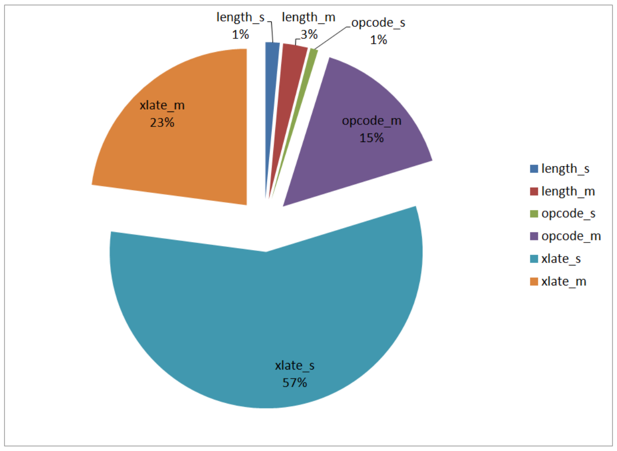

Fig. 6: z80 Output Error Characterization.

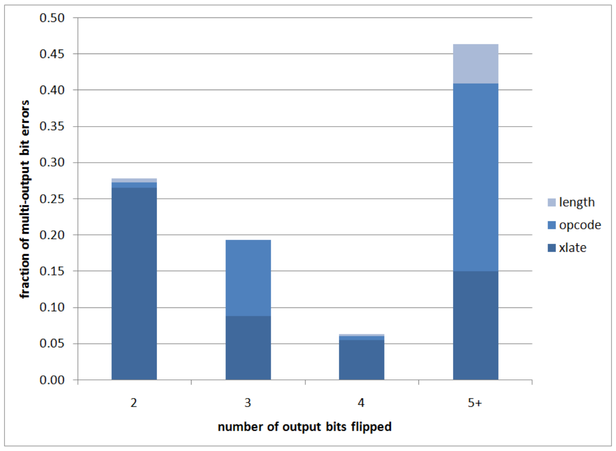

Fig. 7: Characterization of Multi-bit Output Errors for z80 Decoder.

that the cases where a higher number of bits are corrupted are more likely to originate from particle strikes in the length decode and opcode identification stages.

In summary, the results of the transient fault characterization performed on the z80 instruction decoder has shown that the assumptions made by many higher level reliability studies with respect to transient fault artifacts are not always valid. In particular, it has been shown that not only are output bits not corrupted with equal probability, but also that in many cases, multiple output bits can be corrupted simultaneously.

V. Discussion

Finding the most appropriate technique (or class of techniques) to use to protect a particular unit is greatly dependent on the functionality and gate level structure of the particular logic block in question. For this analysis, we classify different protection techniques into 3 broad categories and discuss their applicability to a CISC instruction decoder.

A. Macro-Level Replication Techniques

These techniques involve spatial or temporal replication, combined with result comparison in order to detect the presence of faults. In some cases, this replication can be global, at the system level [8], processor level [9], or at the thread level [10]. In other cases, replication can occur only on a functional unit level[11], or even in software[12]. The most natural application of a replication technique to an instruction decoder would be to either replicate the decoder (spatial redundancy) or perform the decode stage twice in the processor pipeline (temporal redundancy) and compare outputs in order to detect errors. While either of these approaches would catch all possible errors that could occur, their high area, power, and/or delay overheads makes them unattractive.

B. Property Based Checking Techniques

Property checking approaches detect and/or correct errors by verifying a chosen property of a computed result or stored value. Information redundancy techniques used for storage fall into this category, as well as code-based checking approaches used for logic blocks [13][14]. Additionally, techniques which check correctness [15][16][17] also fall into this category. In some cases, a property check failing only indicates a guess that an error occurred, meaning that there is a potential for performance-degrading false positives. Symptom-based approaches [18][1] fall into this category.

The design of the decoder itself has significant implications with respect to how applicable a protection technique in this category would be. The majority of conventional CISC architectures translate instructions into RISC primitives in order to simplify the design of the execution hardware. One general attribute of RISC ISAs is their regularity, specifically in terms of which combinations of opcodes, source registers, and destination register can be combined to form valid instructions. From an error detection standpoint, an irregular ISA could be attractive because if many invalid opcode-register combinations exist, it is likely that a transient fault in the decoder logic could generate a translation of RISC instructions in an invalid format. Unfortunately, the underlying RISC ISA for the decoder implemented for this study is regular, precluding such a scheme from being effective.

C. Implementation-Level Techniques

Techniques in this category involve manipulating individual gates and/or transistors in order to improve soft error tolerance. Because techniques in this category generally manipulate low level components, some effort is needed to only modify a subset of components in order to ensure that other design goals are not sacrificed for the sake of reliability. For this reason, techniques in this category are often coupled with heuristics in order to make these decisions.

Many of the early implementation-level techniques provided reliability by resizing individual transistors within a circuit, increasing the drain capacitance (and thus the minimum amount of charge (Q_{crit}) needed to be generated by a striking particle in order to induce a fault)[19]. Another subset of techniques also exists in which the presence of transient activity is detected at the input of sequential elements [20][21][22][7]. Many of these techniques were originally proposed to enable varying degrees of **timing speculation**, where logic circuits are clocked at a frequency higher than what their critical paths would safely allow. These techniques are useful for SET detection because timing violations manifest themselves in the same manner as SETs arriving at flip-flop inputs: both cause data inputs to be unsettled at the end of the clock period, and can be detected by sampling data lines at two points in time and comparing the samples.

In contrast to property-based checking techniques, implementation level techniques are naturally suited to complex logic blocks like instruction decoders. Decoders have complex input to output mappings, and correctness can not be as easily verified as it can for regular arithmetic circuits. As an example, consider an integer multiplier and the z80 decoder implemented for this study. For the multiplier case, results can be checked by performing modulo arithmetic on the multiplicands (essentially performing a much smaller multiplication operation), as discussed in [23]. For the z80 decoder case, it is significantly harder to determine if a sequence of RISC operations produced at the output of the logic block matches the z80 instruction provided at the input. In addition to this, the logic within a decoder is likely to be significantly more random in structure than datapath logic, implying that some gates within such a logic block are more likely to logically mask faults. This is an attractive attribute because it implies that an implementation level technique can be effective while only manipulating a small subset of components. Timing speculation techniques where transient faults are captured at flip-flop inputs look to be a particularly good match for instruction decoders, specifically for the the pipestage partitioning used in our implementation. Our results in Section 4 show that while 45.4 % of injected faults are logically masked before reaching a flip-flop, only 7.2 % are logically masked after corrupting at least one flip-flop at the end of an intermediate pipestage. If a subset of critical flip-flops at the end of the length decode and opcode identification stages could be identified and protected, the overall soft-error vulnerability of the the decoder would be greatly reduced.

VI. Summary

In summary, this paper provides a detailed soft error vulnerability characterization for a parallel instruction decoder. The results of this analysis imply that an implementation-level technique would likely be the best strategy for proteching such a unit, and that many of the assumptions made by prior works regarding the artifacts produced by transient faults are not always valid. Future work will investigate how timing-level speculation techniques can be used to protect decoders, as well as how the observations regarding the occurrence of multi-bit errors and the likelihood of different output bits being corrupted can be incorporated into higher level fault models.

Acknowledgments

The authors wish to thank Ilhyun Kim for providing the original scalar Z80 decoder as a starting point for our parallel decoder.

References

[1] P. Racunas, K. Constantinides, S. Manne, and S. Mukherjee, "Perturbation-based Fault Screening," in *Proceedings of HPCA-13*, February 2007.

[2] J. Cook and C. Zilles, "A characterization of instruction-level error derating and its implications for error detection," in *Proceedings of DSN 2008*, June 2008.

[3] V. Reddy and E. Rotenberg, "Inherent Time Redundancy (ITR): Using Program Repetition for Low-Overhead Fault Tolerance," in *Proceedings of DSN*, June 2007.

[4] H. Pan and K. Asanović, "Heads and Tails: A Variable-length Instruction Format Supporting Parallel Fetch and Decode," in *CASES '01*, 2001.

[5] T. Scherrer, "Z80 Family CPU User Manual," `http://www.zilog.com/docs/z80/um0080.pdf`.

[6] D. Papworth, "Tuning the Pentium Pro Microarchitecture," *IEEE Micro*, vol. 16, no. 2, pp. 8–15, Apr 1996.

[7] E. L. Hill, M. H. Lipasti, and K. K. Saluja, "An accurate flip-flop selection technique for reducing logic ser," in *DSN*. IEEE Computer Society, 2008, pp. 128–136.

[8] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen, "NonStop Advanced Architecture," in *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks*, 2005.

[9] T. J. Siegel, E. Pfeffer, and J. A. Magee, "The IBM eServer z990 Microprocessor," *IBM J. Res. Dev.*, vol. 48, no. 3-4, pp. 295–309, 2004.

[10] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed Design and Evaluation of Redundant Multithreading Alternatives," in *Proceedings of ISCA-29*, May 2002.

[11] J. Ray, J. Hoe, and B. Falsafi, "Dual Use of Superscalar Datapath for Transient-fault Detection and Recovery," in *Proceedings of MICRO-34*, December 2001.

[12] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August, "SWIFT: Software Implemented Fault Tolerance," in *Proceedings of CGO*, March 2005.

[13] D. A. Anderson and G. Metze, "Design of Totally Self-Checking Check Circuits for m-Out-of-n Codes," *IEEE Transactions on Computers*, vol. 22, no. 3, pp. 263–269, 1973.

[14] J. W. Watterson and J. J. Hallenbeck, "Modulo 3 Residue Checker: New Results on Performance and Cost," *IEEE Transactions on Computers*, vol. 37, no. 5, pp. 608–612, 1988.

[15] V. Reddy, A. Al-Zawawi, and E. Rotenberg, "Assertion-Based Microarchitecture Design for Improved Fault Tolerance," October 2006.

[16] V. Reddy and E. Rotenberg, "Coverage of a Microarchitecture-Level Fault Check Regimen in a Superscalar Processor," in *Proceedings of DSN*, June 2008.

[17] S. A. Seshia, W. Li, and S. Mitra, "Verification-Guided Soft Error Resilience," in *Proceedings of Design Automation and Test in Europe (DATE)*, April 2007.

[18] N. Wang and S. Patel, "Restore: Symptom-Based Soft Error Detection in Microprocessors," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 3, pp. 188–201, July-Sept. 2006.

[19] Q. Zhou and K. Mohanram, "Transistor Sizing for Radiation Hardening," in *Proceedings of Reliability Physics Symposium*, April 2004.

[20] D. Ernst, N. S. Kim, S. Das, S. Pant, T. Pham, R. Rao, C. Ziesler, D. Blaauw, T. Austin, and T. Mudge, "Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation," in *Proceedings of MICRO-36*, November 2003.

[21] M. Nicolaidis, "Time Redundancy Based Soft-Error Tolerance to Rescue Nanometer Technologies," in *VTS '99: Proceedings of the 1999 17TH IEEE VLSI Test Symposium*, 1999.

[22] S. Mitra, M. Zhang, N. Seifert, B. Gill, S. Waqas, and K. S. Kim, "Combinational Logic Soft Error Correction," in *Proceedings of International Test Conference*, November 2006.

[23] A. Meixner, M. E. Bauer, and D. J. Sorin, "Argus: Low-Cost, Comprehensive Error Detection in Simple Cores," in *Proceedings of MICRO-40*, December 2007.