AN EFFICIENT ARCHITECTURE FOR DETECTION OF MULTIPLE BIT UPSETS IN PROCESSOR REGISTER FILES

BY WEN YUEH

A thesis submitted to the
Graduate School—New Brunswick
Rutgers, The State University of New Jersey
in partial fulfillment of the requirements
for the degree of
Master of Science
Graduate Program in Electrical and Computer Engineering

Written under the direction of
Prof. Tapan J. Chakraborty
and
Prof. Michael L. Bushnell
and approved by

_____

_____

_____

New Brunswick, New Jersey
May, 2009

ABSTRACT OF THE THESIS

An Efficient Architecture for Detection of Multiple Bit Upsets in Processor Register Files

by Wen Yueh

Thesis Directors: Prof. Tapan J. Chakraborty

and

Prof. Michael L. Bushnell

With the semiconductor industry transitioning into the next generation of deep submicron technology such as 40 nm or 32 nm CMOS technology, transistors are becoming more vulnerable to malfunction due to soft errors. Due to the reduction in the supply and threshold voltages of the transistors in this smaller geometry, soft errors can affect the state of multiple numbers of transistors simultaneously. Hence, the traditional fault model of a *single event upset* (SEU) due to a soft error needs to be revisited and a new fault model and an associated fault tolerant architecture for circuit structures in deep submicron technology based on *multiple-bit upsets* (MBUs) needs to be developed. In this reseach work, we propose a novel fault tolerant architecture for the register files in a processor that can detect the MBUs in them efficiently. In this proposed method, we compute and store the *Cyclic Redundancy Check* (CRC) bits of a complete register which can be 32 bits or 64 bits wide, when new data is loaded into it. The CRC bits are of smaller bit size and are stored in a soft error protected memory structure using

well known conventional soft error protection mechanisms such as *error-correcting codes* (ECCs), etc., for memory structures. When data from a register is read, first the new CRC value is computed based on the existing data value stored in that register and compared against the original CRC value stored previously in the soft error tolerant memory structure. If there is a difference in these two CRC values an error is flagged as it shows that the data is corrupted due to a soft error either by an SEU or MBUs during the time interval between the last write and the current read operation for that register. This operation is done for reading every register in the register file. Although this method introduces timing and area overheads, they are tolerable and this method of detection scales with the increase in the number registers in the register file. Finally, we present simulation results regarding the fault detection capability of this proposed method.

Acknowledgements

I am thankful to Profesor Tapan J. Chakraborty for his valuable guidance and advice as my adviser. I am also thankful to Professor Michael L. Bushnell for his cooperation, which made this work possible. I am also thankful to him for his valuable inputs to this work. I am indebted to Jordan Aubry, Jayant Silva, and my many student colleagues for their feedback and comments to improve my work.

Dedication

To my parents, my grandfather, and all my teachers.

Table of Contents

List of Tables

List of Figures

Chapter 1
Introduction

Building a reliable product from unreliable sources has been an open problem in many engineering fields. For the field of electrical engineering, this problem was once thought to be solved with CMOS technology. However, as the size of transistors is scaled down with each successive chip generation, the overall issue of reliability reappears. Noise from various sources that were once considered to be tolerable has become a concern for hardware reliability. As the new generation of nano scaled sensors and devices emerge, fault tolerant hardware becomes critical in ensuring reliable operations.

1.1 Causes of Soft Errors

As opposed to hard errors, soft errors do not create permanent circuit defects in the hardware. They are temporary errors that occur in operational time. High energy particles such as α particles and neutrons travel through the silicon substrate creating election and hole pairs. The excess carriers then disrupt normal transistor operations until they recombine or diffuse to well taps. Soft errors are the manifestation of these disruptions. The charge that is necessary to cause an upset of a node logic value is defined as the *critical charge*. The charge collection processes are mainly dominated by a drifting process known as *charge funneling* [22]. A funnel is created when a particle enters the depletion region, and the depletion layer is ionized by the particle's plasma track. This removes the screening effect of the depletion layer and a potential difference along the particle's track guides minority carriers in the plasma drifting upwards. The

funneling effect enhances the charge collection at the particle entry position as shown in Figure 1.1. From the standpoint of a *multiple-bit upset* (MBU) caused by single events, charge funneling is beneficial because it increases charge collection at the struck node and decreases the amount of charge diffusing to neighboring nodes [12].

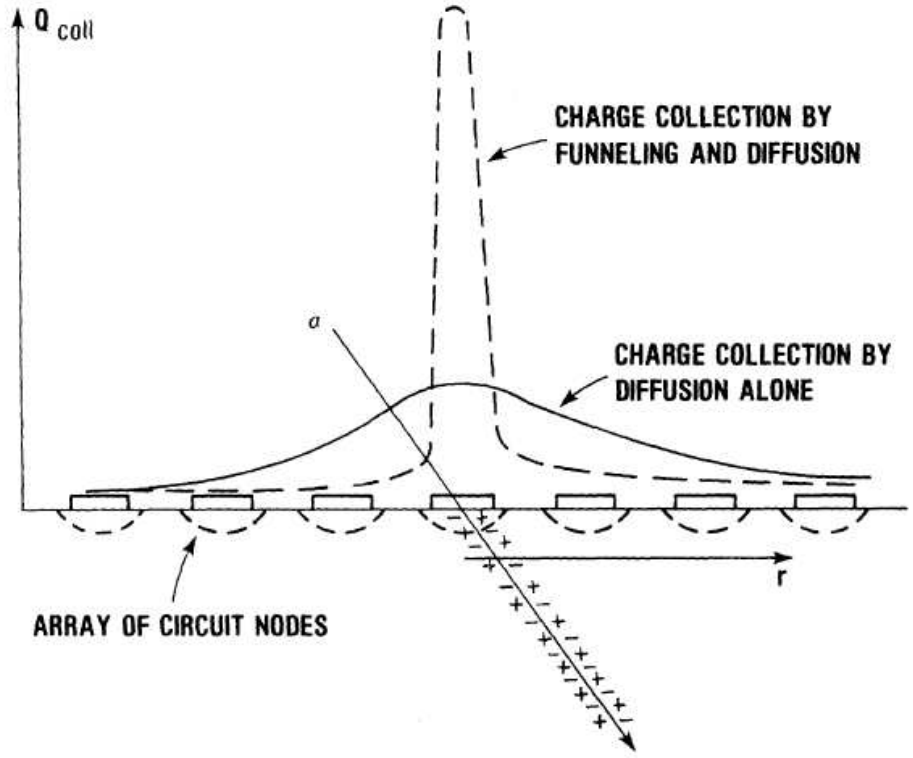

Figure 1.1: Funneling effect of a particle strike [22]

1.2 Sources of Radiation

In a terrestrial environment, the sources of high energy particles are radioactive decay of packaging materials, cosmic radiation, or a combination of both [1]. One significant source of high energy particles comes from radioactive decay of the device and packaging materials. The reaction emits α particles, which are a significant source of ionizing radiation. The α particle is composed of a doubly

Table 1.1: Alpha particle emissivities of various materials [1].

Material	Emissivity (α/cm^2-hr)
Fully Processed Wafer	< 0.001
30μm thick Cu Metal	< 0.002
20μm thick AlCu Metal	< 0.001
Mold compound	< 0.024 to < 0.002
Flip Chip Underfill	< 0.002 to < 0.001
Eutectic Pb-based Solders	< 7.200 to < 0.002

ionized helium atom ($^4He^2$) that carries kinetic energy within the range of 4-9 MeV. While the primary sources of α particles are the packaging materials, materials used to fabricate the chip, and materials used during the fabrication process can also contribute to these impurities. Although radioactive decay can produce other by-products such as β particles, they generally do not carry enough energy to cause a soft error. Thus, α particles are still the primary source of soft errors. The α particle emission rates of some key production materials are summarized in Table 1.1.

A second significant source of radiation is from the ever-present cosmic rays. The required energy for a primary cosmic ray to be able to affect a circuit at ground level energy is 1 to 17 GeV. The required energy varies due to a particle's trajectory interacting with Earth's magnetic field. Usually, a particle will react with the Earth's atmosphere and create a shower of secondary particles. Upon reaching Earth's surface the cascade is mainly composed of third to seventh generation particles. At terrestrial altitudes, less than 1% of the primary flux reaches sea level [33]. The particles that arrive will mainly be muons, protons, neutrons, and pions. Among these particles, neutrons are most likely to enter terrestrial altitudes, since pions and muons are short-lived and protons and electrons are charged particles, which lose energy through Coulombic interactions with the atmosphere. As neutrons only react to the *strong interaction* nuclear force and the interaction has a limited range of 10^{-13} cm, they tend to preserve their energy until the strong interaction takes place. Due to these facts, 97% of the flux at sea level is primarily neutrons. These high energy neutrons react with the silicon

lattice through neutron-induced silicon recoil. In this case, the high energy neutron transfers its kinetic energy to the silicon nucleus. With enough energy, the silicon nucleus breaks into smaller fragments and generates a burst of charge. It is also possible for neutrons to hit an unstable boron isotope and induce nuclear fission. This type of reaction allows neutrons with energies as low as 15 eV to trigger an ionizing reaction [2].

1.3 Types of Soft-Errors

Soft errors can be further categorized into SEUs and *Single Event Transients* (SETs). SEUs upset a sequential element and alter stored values; SETs affect the combinational circuitry and cause glitches and delays. In sub-micron scaled DRAMs, there have been observations of *Multiple Bit Upsets* (MBUs) where multiple bits in a word are corrupted by distinct particles, or *multiple-cell upsets* (MCUs) where multiple bits in a word are corrupted by a single particle. For an SEU to take place, there must be enough excess charge in the circuit node to violate a transistor's noise margin. Once the collected charge exceeds the *critical charge*, the transistor's operation is disrupted. Different types of particle strikes will have different responses. An α particle strike will follow a linear energy transfer equation. The particle gradually loses energy as it goes through the silicon substrate and creates electron and hole pairs. If the collected charge exceeds the critical charge, there will be an SEU. Although high energy neutrons do not normally interact with silicon, if they collide with a silicon nucleus while traveling through the silicon substrate they generate a burst of charge. When this happens, the collected charge is more likely to exceed the critical charge.

1.4 Issues of Reliability and Validity

A soft error does not necessarily always cause a system failure. If a fault occurs in a cache bit that has not been read before it is overwritten with new data, it would not affect the system performance. If the corrupted data is indeed read by

the system, it still might not affect the logical outcome. The combinational logic would possibly mask the error propagation. But if soft errors remain undetected they manifest themselves as *silent data corruptions* where they ultimately affect the system's final response without any warnings. To prevent such corruption from happening, we need additional mechanisms to correct or detect errors in the system. As the size of transistors is scaled down, the noise margin of the transistors becomes lower, and the density of transistors increases. A lower noise margin can result in a lower critical charge, which means the threshold energy of an SEU is reduced accordingly. The densely packed transistors also become more vulnerable to soft errors because the carriers generated by high energy particles can travel to different transistors before they recombine. Alternatively, the particle directly strikes through multiple cells and creates multiple soft errors. A single particle can now affect multiple nodes and cause MBUs, which are currently only a concern in densely packed memory cells, in a state machine in the logic circuit.

1.5 Contributions of the Research

We develop a fault tolerant scheme for registers that uses the CRC to achieve better performance than the common Hamming code in MBU detection. The CRC can either be protected by a Hamming code or left unprotected. As compared to IBM's *symbol code*, it is also slightly better in the ability to handle burst errors. In random error detection, the IBM symbol code missed 25.8% fewer random errors than the CRC code. In burst error detection, our stronger CRC code missed 0.5% fewer errors than the IBM SSC-DSD code. Although on average the burst error detection rates of both techniques are almost equivalent, the CRC code can withstand burst error lengths of 10 bits before a miss while the IBM symbol code can only withstand 9 bits. The CRC code has 3.9% less hardware overhead compared with the IBM symbol code. Thus, if chip real estate is a greater concern and multiple faults are modeled as MCUs, a CRC can be chosen instead of IBM's technique. Compared with the widely accepted distance 4 Hamming code,

our chosen CRC code is 6.0% smaller in hardware overhead than the distance 4 Hamming code. The random error detection miss rate of the CRC code is 1% less than the distance 4 Hamming code and the burst error detection miss rate is 88% less than the distance 4 Hamming code. Also, for burst error detection, the CRC can withstand a maximum of 5-bit burst errors while the distance 4 Hamming code can only withstand 3-bit burst errors. Furthermore, CRCs are more flexible to satisfy different design constraints, since the CRC polynomials can be chosen to detect arbitrary burst error lengths.

1.6 Thesis Organization

The thesis is organized as follows. Chapter 2 introduces related and prior work in the field. Chapter 3 introduces the reason for choosing code redundancy for our fault tolerant approach. Chapter 4 uses the idea of code redundancy and demonstrates a proof of concept. Chapter 5 details the implementation and experimental results. Chapter 6 concludes the research and proposes further research and development.

Chapter 2
Prior Work

There have been many prior techniques to reduce soft errors. Approaches to the problem include developing better fabrication processes, resizing transistors, modifying circuit topologies, and using error correcting and detecting codes. These approaches have their strengths and limitations. It is possible to combine different approaches to complement each technique's weakness. A fault tolerant system may incorporate one or more of these approaches to achieve a lower failure rate. However, in some of these techniques, they are optimized for SEUs and are not built to handle MBUs. To build systems that tolerate MBUs, there is a need to rethink a new design optimized for MBUs.

2.1 Techniques for Fabrication Processes

Baumann introduces several common methods to reduce soft errors due to α particles [1]. Since the majority of α particles come from the radioactive decay of packaging materials, one approach that reduces soft errors would be to use high-purity materials and screen for low alpha emission. However, due to the limitation of the current detection method that resolves α emission concentrations up to 0.001 α/hr-cm^2, it is insufficient to identify material emissions below this level [21]. Another approach would be to identify high α particle emitting materials and to have design rules for the chip floor plan to locate sensitive components away from them [1]. This technique works if the package has defined high emissivity materials such as solder bumps and the chip has logic cores that are significantly more sensitive than the other cores. Using package shielding with polyimide thin

films is another technique to guard against α particles [11]. The films coated over the finished chip prior to bonding and encapsulation can also bring α particles to a stop or at least slow them down. Extensive analysis on the source and energy of the impurity is helpful in choosing film thickness, to avoid worsening the *soft error rate* (SER) [28]. The reason for this is that the non-linearity in the α particle's charge generation rate reaches maximum ionization just before the particle stops and comes to a reset in the bulk lattice. Hence the *sensitive volume* (SV), which is the well dimension under the transistor, collects maximum charge.

The same transistor dimension in different fabrication technologies often shows different resistance against soft errors. The *Silicon on Insulator* (SOI) technology is known to have a lower SER than bulk technology, because of a lower junction capacitance and better noise isolation. SOI technology is also a more MBU resistant technology, because it does not suffer charge sharing and well charge collection-induced bipolar effects that are seen in the bulk technology. The reduction of the substrate thickness in SOI also lowers the SV depth. With a smaller SV, collected charge decreases when high energy particles intersect the silicon lattice, which results in a lower error rate [9]. In recent research on SOI SRAM design, in 65 nm nodes the SER is reduced as much as 75% when compared to bulk technology [8].

2.2 Techniques for Transistor Sizing

When shrinking transistors reduce the node's critical charge, it is reasonable to selectively resize nodes to improve SERs in sequential nodes. The technique resizes nodes that are not on circuit critical paths to improve the soft error rate (see Karnik *et al.* [19]). Since the extra capacitance is not on the critical path, there are no noticeable delay penalties and the method has only 3% power penalty. The improvement has been observed to be as high as 3× at the cell level.

Karnik *et al.* analyze the node capacitance and related *capacitance-voltage* (CV) curves [19]. One pMOS and one nMOS transistor are connected in parallel

to form a capacitor. The reason for using both pMOS and nMOS transistors in parallel is that the CV response changes the signal voltage swings between logic high and logic low. To compensate for such non-linearity of MOSFETs, the two complementing MOSFET capacitances are lumped together with a parallel connection. This extra capacitor is then added on the non-critical path of a cell. In the case of latch design, it is added within the feedback node between two feedback inverters. The benefit of doing this is that we add a low-pass filter to the circuit, which filters out SETs.

Karnik *et al.* also proposed an algorithm to insert these error hardened latches without noticeable performance penalties [19], since 70% of the circuit paths are non-critical. They propose an insertion equation as follows:

$$Inserting\ Priority = \alpha * \sum_{i=1}^{|C^S|} SERQ_i - \beta * \left[\sum_{i=1}^{|C^O+C^C+C^S|} C_{gate}(i) \right] \tag{2.1}$$

where C^O are output cells, C^C are the combinatorial cells, and C^S are the sequential cells in a given block. $SERQ$ is the SER tolerance and C_{gate} is the gate capacitance, and α and β are parameters to balance between SERs and active power. The selection algorithm will rank the latches based on the above equation and select the least critical latch for insertion. As the insertion will modify the maximum and minimum delay slacks, the program iteratively re-evaluates the delay criticality and finds the next suitable latch.

Similarly, *Soft-Error Filtering* (SEF) is a technique to reduce soft errors by filtering radiation induced noise pulses [29]. The physical implementation of SEF inserts low pass filters in front of latches and filters any transient fault from the combinational logic. This technique effectively filters *single event transients* (SETs) but does not protect the latch if the fault occurs when the latch is opaque. Thus, this technique cannot protect latches from SEUs where the soft error occurs when the latch is opaque and holding the data. The latch will forget its previous value and obtain the erroneous charge. Thus, although this technique shows a typical overhead of 17%, it cannot protect against all types of soft errors.

While SEF does not protect against SEUs, the methodology used in the paper

gains our attention. The technique analyses a latch as a *resistor-capacitor* (RC) circuit and treats *set* and *reset* lines as signal integrators. The RC time constant of the integrators is the key to filtering out soft errors. Within given process parameters, properly sized pMOS and nMOS capacitors will be attached on the *set* and *reset* lines to achieve the desired filtering effect. The technique is called *double-filtering*, since the feedback stage of the latch takes differential outputs of the *set* and *reset* signals. The authors propose an optimizing equation as follows:

$$\frac{U}{S-1} = e^{(2-S)U} \tag{2.2}$$

The variable S is a performance measure called the *security margin*. It represents the factor of setup time over the longest transient error a latch tolerates. The factor U is the error pulse width over the RC time constant of the filter. Finding the optimum U for a given S will determine the latch delay and the capacitance on the *set* and *reset* lines.

This method of providing the soft error immutability is solely based on manipulating the node critical charge against errors. Obtaining properly weighted transistor sizes could be difficult in different fabrication technologies. The efficiencies of these selective scaling techniques also depend on the circuit functions. Without a proper analysis of electrical properties of the circuit it could be difficult to determine the merit of these techniques.

2.3 Techniques for Circuit Topology

The *Dual Interlocked storage CEll* (DICE) is a design hardening technique to reduce soft errors at the circuit level [6]. A DICE cell uses hardware redundancy in a latch to ensure a copy of non-corrupted data after an SEU. It also decouples the n-tree and p-tree feedback inputs of each output node. This configuration prevents the corrupted data propagating over adjacent transistors and allows the correct data to restore the corrupted sets. Unlike the traditional *triple-modular redundancy* (TMR) technique, the DICE cell only uses approximately 40% area

overhead compared with an unprotected cell and does not need an additional voter in the design.

The DICE design relies on the principle of *dual node feedback control*. Referring to Figure 2.1, each charge storing node $X0$, $X1$, $X2$, and $X3$ is driven by two neighboring transistors and drives two other neighboring transistors. Two different types of transistors form feedback loops in the opposite direction. Transistors $N0...N3$ are n-type and form a counter-clockwise feedback loop. Transistors $P0...P3$ are p-type and form a clockwise feedback loop. To flip the stored charge, two nodes on the opposite diagonals must flip simultaneously. When an upset pulse occurs in only one node, then the cell will recover as follows (assuming node $X0$ is affected victim):

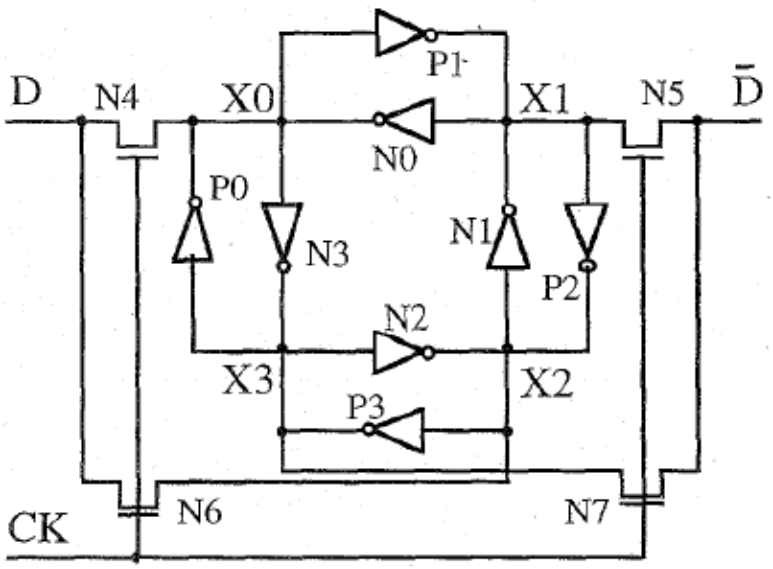

Figure 2.1: Principle of the dual interlocked storage cell [6]

- Nodes $X0...X3$ store 1010
- Node $X0$ flips from logic 1 to logic 0 due to a soft error
- $P1$ turns on; $N3$ turns off
- Node $X1$ voltage settles between logic 0 and 1; $X3$ remains unaffected

- $P2$ turns partially off
- Node $X2$ remains unaffected
- $P0$ recovers $X0$; $N1$ recovers $X1$

Another case:

- Nodes $X0...X3$ store 0101
- Node $X0$ flips from logic 0 to logic 1 due to a soft error
- $N3$ turns on; $N1$ turns off
- Node $X3$ voltage settles between logic 0 and 1; $X1$ remains unaffected
- $N2$ turns partially off
- Node $X2$ remains unaffected
- $N0$ recovers $X0$; $P3$ recovers $X3$

A design used in *National Aeronautics and Space Administration* (NASA) space applications was proposed [18]. The design separates the nMOS and pMOS storage nodes so that electrons are collected in n-diffusions and holes are collected in p-diffusions. This reduces the chance of a particle simultaneously affecting a pair of storage nodes. The design of Figure 2.2 shows an implementation of the described SEU-tolerant latch. The design stores charges in nodes PP, QP, NN, and QN. If any node holds an incorrect logic level due to a soft error, the C-switch at the output Q maintains the previous correct logic. For example, suppose that a logic 0 is to be stored in the latch, and node NN is driven to an incorrect logic state. The latch blocks an erroneous state change as follows:

- Nodes PP, QP, NN, and QN store 0, 1, 0, and 1, respectively
- Node NN flips from logic 0 to logic 1 due to a soft error
- $N3$ turns on; $P4$ turns off

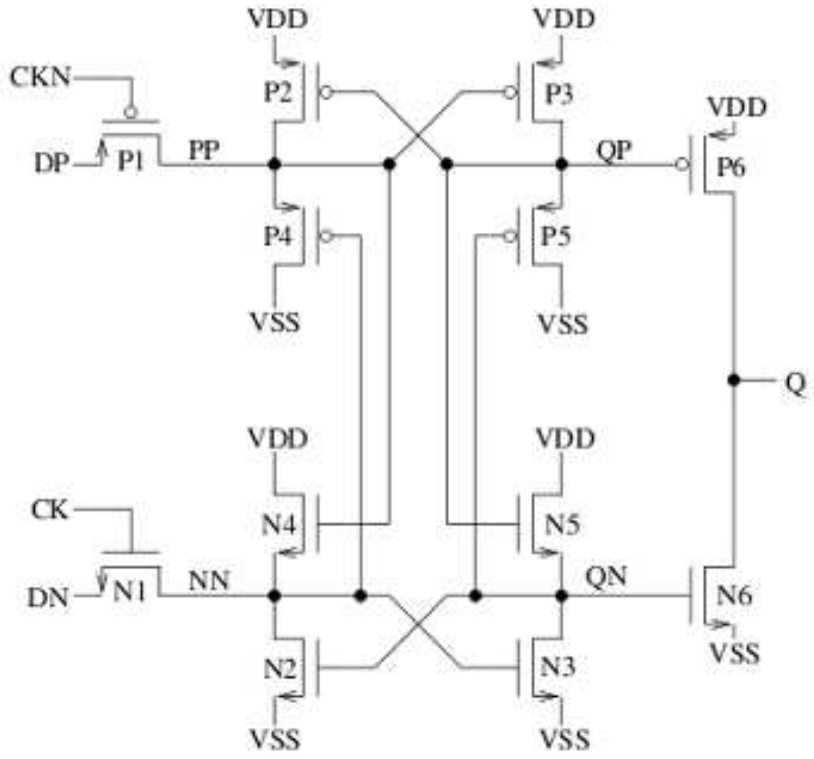

Figure 2.2: NASA SEU tolerant latch [18]

- Node QN's voltage settles at logic 0; PP remains unaffected
- $P5$ turns on, $N6$ turns off
- Node QP voltage settles at logic 1; Q remains unaffected

Here is another case of Q holding logic 1, while node PP is driven to an incorrect state 0:

- Nodes PP, QP, NN, and QN store 1, 0, 1, and 0, respectively
- Node PP flips from logic 1 to logic 0 due to a soft error
- $P3$ turns on; $N4$ turns off
- Node QP voltage settles at logic 1; NN remains unaffected
- $N5$ turns on, $P6$ turns off
- Node QN voltage settles at logic 0; Q remains unaffected

The work of NASA claims that there are only two cases in all possible state changes, since a node that is driven by an nMOS transistor has a monotonic drop from logic 1 to logic 0 upon a soft error, and a monotonic rise if a pMOS transistor is affected [20]. The latch design relies on the parasitic charge on the output node to preserve the previous correct state when an error occurs. Unlike the DICE design, the circuit does not actively correct an erroneous node charge. The latch design incorporates a *temporal separation technique* to reduce SET faults. The technique reduces the transient fault by delaying an input line with an additional buffer. The latch treats a SET pulse as two successive SEU pulses and stops their propagation individually. In more recent research [20], the cell shows 34× higher resilience against soft errors compared with unprotected cells. This technique further expands the charge-collecting technique and creates a soft error hardened cell that preserves charges in three nodes. The innovative arrangements of transistors also block erroneous charge propagation. The design effectively combines the benefits of two designs [6, 18].

Shadow latching is a topological approach to soft error detection in flip-flops [13]. The flip-flop design contains a regular master-slave flip-flop and a parallel pulse latch. The augmented pulse latch toggles synchronously with the flip-flop. Alternatively, it toggles with a slightly delayed clock to guard against SETs from the combinational circuit. The latch follows the flip-flop operations at normal operation with little performance penalty, and it is thus called a shadow latch. Upon a soft-error occurrence, the logic value of the shadow latch and the flip-flop will feed into a meta-stability detector and generate an error interrupt. This technique only detects but does not correct soft errors.

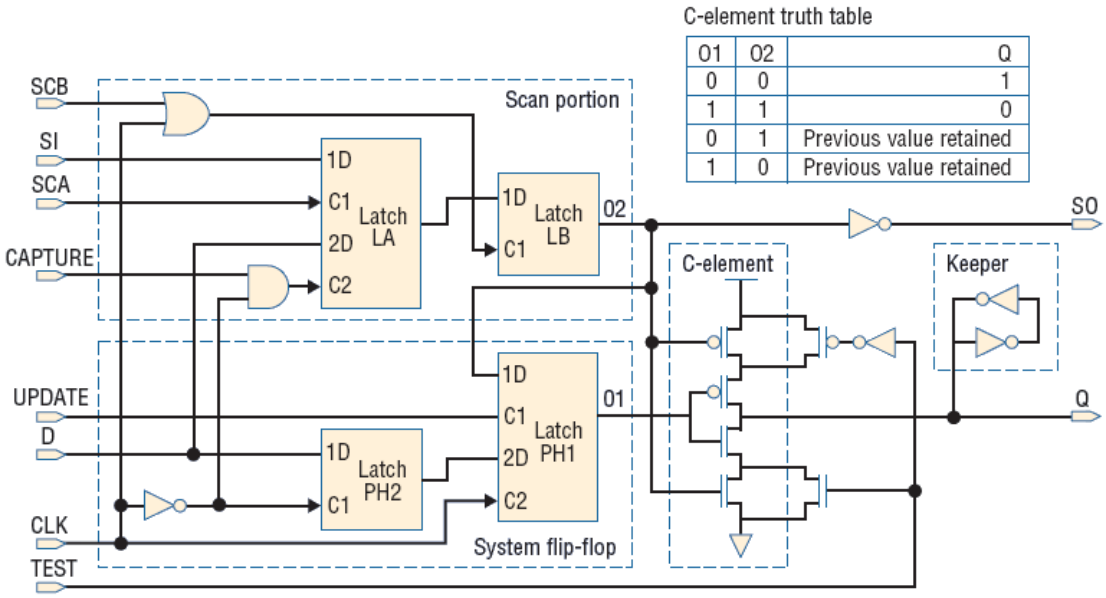

Figure 2.3: The BISER design with a C-element truth table [23]

2.4 Techniques for Reusing Scan Latches

Extending the idea of shadow latching, *Built-In Soft Error Resilience* (BISER) redesigns normal scan latches and reuses the scan latch as shadow latch to achieve a lower area overhead [23]. Referring to Figure 2.3, if one of the BISER latches should fail, a C-element at the output node will block the transmission of the

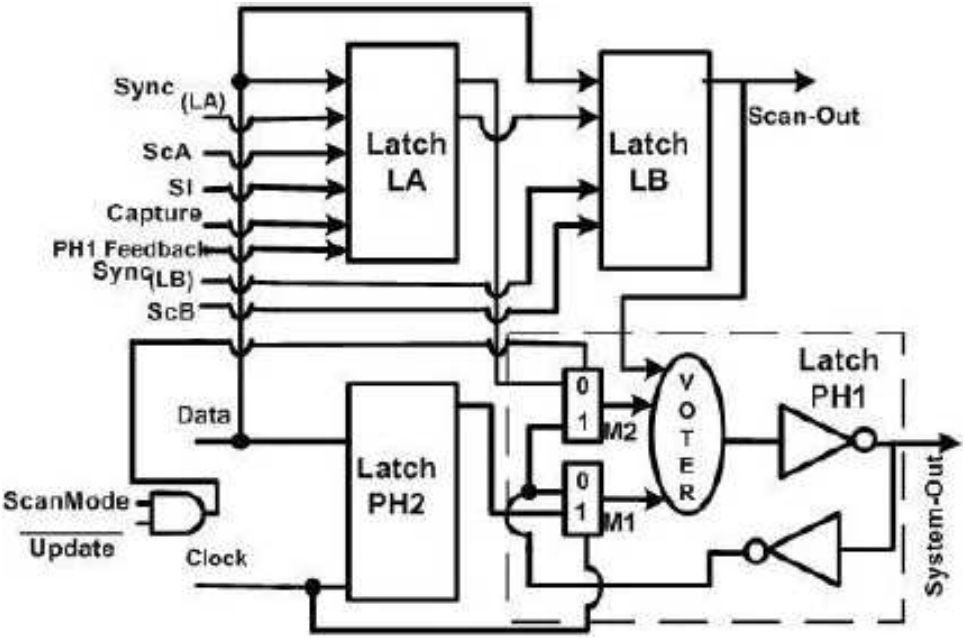

Figure 2.4: The XSEUFF2 scan flip-flop design [25]

incorrect value and a keeper will preserve the last known consensus value. Effectively, it interrupts error propagation as well as detects errors. Mitra *et al.* claim that even though the BISER flip-flop uses 218% of the power, the cell area overhead is only 8% compared to an unprotected scannable flip-flop. While a scannable flip-flop has two unused latches in the functional mode, it is possible to perform a correction with majority voting instead of simple detection and metastable state blocking. Oliveira *et al.* presented another design that utilizes the scan latch [24]. The design modifies existing scan latches and uses TMR and voting. An improvement of the design was made later [25], which can protect against not only SEUs but also *single event transients* (SETs). The improved design is the XSEUFF2 flip-flop of Figure 2.4. The design has 37% hardware overhead and uses 95% more power compared to the standard scan flip-flop. It samples the combinational input at 3 distinctly different times around the clock edge, and then votes to determine the result. Although the design has larger area

overhead and has similar power consumption to Mitra's work, the XSEUFF2 design relies on static logic outputs and uses no dynamic logic, unlike Mitra. Also, the overall overhead in the circuit reduces to 15% to 20% with careful placement of protected cells. This protection also greatly reduces the *window of vulnerability* (WoV) – the time period when a SET is latched in to the flip-flop and results in an SEU. The design incorporates the *temporal separation* technique, which was introduced earlier in the work of NASA, to sample the combinational logic output at different phases of the clock. This is an effective time redundancy approach to the current SET problem. Roy *et al.* further optimize the BISER design at the transistor level [16]. Referring to Figure 2.5, the scan cell is named ESFF-SEC. The ESFF-SEC cell is 62% of the original BISER cell size and uses only 79% of the BISER cell power. The cell design is also capable of performing delay test by toggling the $Hold$ signal with controlled timing.

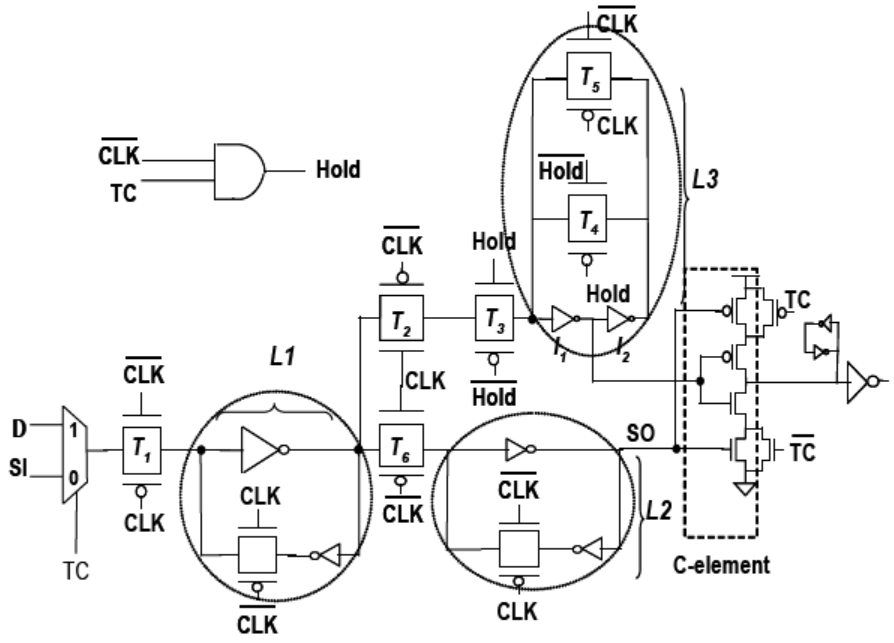

Figure 2.5: The ESFF-SEC scan flip-flop design [25]

2.5 Techniques for Code Redundancy

The code redundancy solution to soft error tolerant hardware has been used in random access memory technologies and error prone storage devices. It is also widely used in digital communications, guarding reliable messages across noisy channels. In memory, *single error correction and double error detection* (SEC-DED) Hamming codes can guard against a single error in a word, or can detect double errors in the same word. There are other codes that have better mathematical properties and are capable of correcting multiple faults, such as the work of Chen *et al.* [10] described in Section 2.5.5. These codes with multiple-bit error correction capabilities generally have more hardware overhead compared to the ones that correct fewer bits.

2.5.1 Hamming Distance

The *Hamming distance* of a pair of binary symbols is the number of bits they differ in. For example, symbols 001 and 010 differ by the lower two bits, so the Hamming distance of these two symbols is 2. Symbols 100 and 000 differ by the highest bit, so the Hamming distance of these symbols is 1. Another way to view the distance between a pair of symbols is how many bits of minimal transitions the symbol must make to reach the other symbol. In our first example, symbol 001 starts from 001, moves to 011 (or 000), and reaches 010. Thus, the minimal Hamming distance is 2. In some error correcting techniques, mapping erroneous codewords to a valid codeword directly uses the minimal Hamming distance for likelihood arbitration. For a code with two valid symbols 000 and 111, upon receiving an invalid code symbol 110, it can be simply decoded as 111 using the Hamming distance concept. This assumes that a code 111 having one bit flip to 110 is more likely to happen than a code 000 having two flips to 110. The ability of detecting and correcting errors with minimal Hamming distance is given in Table 2.1.

The previous code example corrects 1 error, but the code in fact is capable of

Table 2.1: Minimal Hamming distances and their properties [17]

Minimal Distance	Meaning
1	Uniqueness
2	Single error detection
3	Single error correction
4	Single error correction plus Double error detection
5	Double error correction, etc.

detecting 2 errors without the correction. The detectable error will then be the shortest Hamming distance between any valid symbol pairs. This should not be confused with the correction plus detection technique.

2.5.2 Hamming Codes

The Hamming code is a well-known error correcting code for memory systems. The basis of Hamming code correction is finding a systematic code with a Hamming distance of 3 [17]. A Hamming code is a *perfect code* in that each code symbol maintains minimal Hamming distance and the code symbol volumes exactly fill the code space. In the original work by Hamming, *single error detecting* (SED) codes, *single error correcting* (SEC) codes, and SEC-DED codes are introduced. The single error detecting code is a simple even *parity check*. The code has n binary digits with the first $n-1$ positions as information bits and the n-th position as a check bit. Checking whether the previous $n-1$ bits contain even or odd numbers of 1's, we assign position n with a 0 or 1 to ensure that the entire expression contains an even number of 1's. If an error occurred, then we will observe an odd number of 1's. As a result, an even number of errors will not be detected, but every odd number of errors will be detected. Such a code has $n-1$ message positions and 1 check position. It forms a $(n, n-1)$ block code with the redundancy of this code being:

$$\frac{n}{n-1} = 1 + \frac{1}{n-1} \tag{2.3}$$

The single error detecting code does not give information on where the errors occur in the data bits. Thus, there is a need to construct a single error correction code with the ability to identify where the error occurs and correct such an error. Since there are m information positions, a k position check must be able to represent every location in $m + k$ bit positions to identify the error location, so that:

$$2^k \geq m + k + 1 \tag{2.4}$$

should be satisfied. Writing $n = m + k$, Hamming obtains the (n, m) Hamming code redundancy equation:

$$2^m \leq \frac{2^n}{n+1} \tag{2.5}$$

Upon solving the above equation for all message lengths under a desired m, the check positions are determined by observing the relation between n and k. When n increments and k changes, the particular position n should be a check bit. Then, the binary representation of the locations will indicate which checkers perform parity checks on what bits. For example, check bit 0, check bit 1, and check bit 2 will check location 7 (0111). Check bit 1 and check bit 3 will check the location 10 (1010). Hamming also introduces the single error correcting plus double error detecting code by adding an extra parity check, treating all previous message bits and Hamming check bits as message bits. It effectively increases the Hamming distance of the code to distance 4.

2.5.3 Cyclic Redundancy Checks

The CRC is a type of checksum that calculates the remainder of a polynomial. This type of cyclic code is very effective at burst error detection. Depending on the code polynomial, the code can detect multiple individual errors and multiple burst errors. Peterson and Brown reported some important facts on these polynomials and their abilities to detect multiple faults are given [27]. Transmitting n digits of code polynomial with k information digits and $n - k$ check digits, a generator polynomial $P(X)$ of degree $n - k$ satisfies the following theorems:

Theorem 2.5.1 *A cyclic code generated by any polynomial $P(X)$ with more than one term detects all single errors.*

Theorem 2.5.2 *Every polynomial divisible by $1 + X$ has an even number of terms. Hence, they detect any odd number of errors.*

Theorem 2.5.3 *A code generated by the polynomial $P(X)$ detects all single and double errors if the length n of the code is no greater than the exponent e to which $P(X)$ belongs.*

Theorem 2.5.4 *A code generated by $P(X) = (1 + X)P_1(X)$ detects all single, double, and triple errors if the length n of the code is no greater than the exponent e to which $P_1(X)$ belongs.*

Theorem 2.5.5 *Any cyclic code generated by a polynomial of degree $n-k$ detects any burst-error of length $n - k$ or less.*

Theorem 2.5.6 *The fraction of bursts of length $b > n - k$ that are undetected is $2^{-(n-k)}$ if $b > n - k + 1$, $2^{-(n-k-1)}$ if $b = n - k + 1$.*

Theorem 2.5.7 *The cyclic code generated by $P(X) = (1+X)P_1(X)$ detects any combination of two burst-errors of length two or less if the length of the code, n, is no greater than e, the exponent to which $P_1(X)$ belongs.*

Theorem 2.5.8 *The cyclic code generated by $P(X) = (X^c+1)P_1(X)$ will detect any combination of two bursts $E(X) = X^iE_1(X) + X^iE_2(X)$, provided that $c+1$ is equal to or greater than the sum of the lengths of the bursts, $P_1(X)$ is irreducible and of the degree at least as great as the length of the shorter burst, and provided that the length of the code is no greater than the least common multiple of c and the exponent e to which $P_1(X)$ belongs.*

A properly chosen cyclic code's minimal Hamming distance is equivalent to the SEC-DED code's minimal distance. The difference is that a CRC checker

is designed for error detection only. Without a syndrome decoding mechanism, the polynomial simply detects triple errors. With the two codes being essentially equally good at random error detection, the benefit of CRCs over Hamming codes would be the ability to detect burst errors. If the number of check bits $n - k$ is greater than the code's minimal Hamming distance, then the CRC's ability to detect burst errors would have an advantage over Hamming codes based on Theorem 2.5.5.

A CRC code is generated by multiplying the code polynomial by the generator polynomial $P(x)$. Using a *Linear Feedback Shift Register* (LFSR) to construct a CRC generator requires an n-bit shift register where n is the order of the generator polynomial. The data input will be XORed with the shift register output and feedback to the input of the first shift register S_0. Then, we insert XOR logic gates before the register polynomial terms where the $P(x)$ coefficient is not zero but we ignore the highest polynomial term and the constant 1 term. For example, a polynomial $P(x) = x^3 + x^2 + 1$ will form the LFSR in Figure 2.6. Removing the highest polynomial term and the constant term, we obtain the x^2 term and we should insert a XOR gate before S_2. The current CRC bits are stored in the S_n registers. This implementation is a simple serial CRC checker that uses the least amount of hardware but takes many cycles to encode and decode the CRC. A parallel CRC can be created through unrolling the LFSR sequential logic or iteratively using constrains to find the parallel CRC generator matrix with the least number of XOR gates [7, 31].

Referring to Figure 2.6, to encode the data sequence, we shift in $u(t)$ serially, where $u(t)$ is a vector of serialized data and t is the index of the current data bit. Alternatively, t is the discrete time steps of the clock. To initialize the checker, we initialize t, S_0, S_1, and S_2 to zeros. At every clock cycle we increment t by 1 time until $u(t)$ is fully shifted into the checker. Then, we stop the shifting and retrieve S_0, S_1, and S_2. These three bits are the CRC check bits of the data sequence $u(x)$. The CRC code word will be the CRC check bits appended to $u(t)$. To perform CRC decoding, we shift the first d bits of the CRC code word

into the same encoder. If the resultant S_0, S_1, and S_2 are equal to the remaining $p - d$ bits of the CRC code word, then there is no error during decoding. Let d be the size of vector $u(t)$ and p be the size of the CRC code word. The delay of encoding will be d clock periods when encoding, and the delay of decoding will be $d + 1$ clock periods. In some designs, the shift register can only access appended check bits after all the data bits are shifted into the checker. For these designs, the decoding delay will be p clock periods.

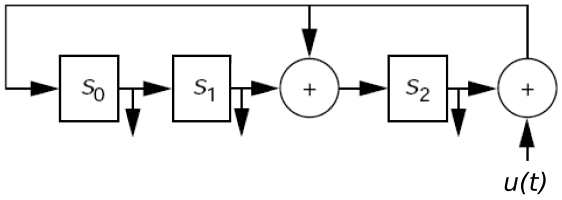

Figure 2.6: The LFSR for $P(x) = x^3 + x^2 + 1$. The data shifts in through $u(t)$ the and CRC check bits are retrieved from outputs of registers S_0, S_1, and S_2.

2.5.4 Bose-Chaudhuri-Hocquenghem Codes

Bose-Chaudhuri-Hocquenghem (BCH) codes were discovered by Hocquenghem in 1959, and independently by Bose and Ray-Chaudhuri in 1960 [5]. For a cyclic code with polynomial degree m and its maximum correctable errors to be t, one important discovery about this class of codes is that for any arbitrary choices of m and t there exists a code of length $2^m - 1$, which is capable of correcting any combinations of t errors with no more than mt redundant digits. In the worst case, a BCH code has $2^m - mt - 1$ data bits and mt check bits. Alternatively, such a code detects $2t$ random combination of errors. This fact gives an upper bond on the required bits for error correction and covers a wide range of data transmission rates and error-correcting abilities [26]. The BCH codes are cyclic, and encoding can be done with a shift register similar to the CRC. Constructing a BCH code polynomial $f(x)$ in field $GF(2^m)$ is simple. The polynomial $f(x)$

would be the *least common multiple* (LCM) of the first $2t-1$ unique irreducible polynomials $p_i(x)$:

$$f(x) = \text{LCM}\left[p_1(x) \times p_3(x) \times p_5(x) \times \ldots \times p_{2t-1}(x)\right] \tag{2.6}$$

Since the factors are irreducible, the least common multiple of them will be simply their product. Even though individual factors are irreducible, it is possible to have duplicated products as the primitive root α^i happens to be the same as α^j in a finite field. In these case, the duplicated terms should be omitted. Reed-Solomon codes and Reed-Muller codes are proven to be equivalent to a subcategory of the BCH code [26].

2.5.5 IBM's Research on Symbol Codes

IBM has done research on symbol correcting codes specifically targeting memory systems [10]. While a SEC-DED Hamming code operates on binary fields, the same concept can be extended to b-bit symbols to form a *single symbol correction plus double symbol detection* (SSC-DSD) code. In the parity check matrix, each entry would be a submatrix of a $b \times b$ zero matrix or a power of the companion matrix T of a primitive polynomial of degree b. A companion matrix is a square matrix with last column defined by a polynomial $p(t) = c_0 + c_1 t + \ldots + c_{n-} t^{n-1} + t^n$ and a sub-identity matrix of size $(n-1) \times (n-1)$ starting from the second row:

$$T(p) = \begin{bmatrix} 0 & 0 & \ldots & 0 & -c_0 \\ 1 & 0 & \ldots & 0 & -c_1 \\ 0 & 1 & \ldots & 0 & -c_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 1 & -c_{n-1} \end{bmatrix} \tag{2.7}$$

The parity check matrix can be viewed as a distance 4 Hamming code that operates on distinct symbols instead of on the binary field. The parity check matrix can be transformed into a systematic form after proper elementary row operations. The transformation is beneficial as it removes duplicated columns and

makes the code more compact.

Symbol codes are specially designed codes targeting two different modes of failures in memory systems: chip failures and bitwise failures. Matching the symbol polynomial degree b to the chip input width, a SSC-DSD is able to withstand one failed chip and is still capable of detecting one symbol error from the remaining good chips. For IBM's G3 and G4 servers, a code with *4-bit correction capability* (S4EC) was implemented [30]. The (76, 64) S4EC/DED code that was implemented on the G3 and G4 servers is designed to ensure that all single-bit failures of one chip occurring in the same doubleword as a 1-to-4-bit errors on a second chip are detected.

2.5.6 Fault Tolerant Systems

In many mainframe computers, fault tolerant design is necessary to achieve reliable computing. In IBM's enterprise grade servers, there are many mechanisms to ensure data integrity: instantaneous error detection, fault isolation, and on-line repair [30]. It is common to have duplicated chips or module pairs to perform redundancy-checks. In the G5 server design, the processors have duplicated instruction fetch/decode units and execution elements for error detection. These *steplocked* pairs will perform identical operations and the final results will be examined by an error correcting module. The design allows a data comparison cycle overlapped with the instruction execution pipeline, so the checker performs operations, transparently without incurring cycle-time penalties. The processor supports online repair through an *error-correcting code* (ECC) protected register file. When an error occurs in registers or during instruction executions, the processor will reset and return to the last check-pointed state. If the error is transient, the processor will resume and the retry will be successful. If the retry fails, the failing processor will halt its own operation and transparently another working processor will retry and continue the application. In the G5 memory hierarchy, every level of cache has adequate protections. The L1 cache has write

through design, and a parity check on each byte is sufficient to protect the duplicated data. In the L2 cache, the data is protected by an ECC. In the L3 cache, a (72,64) SEC-DED code protects the system against catastrophic failures. Here, the stronger (76,64) S4EC-DED code used in G3 and G4 is no longer in use in G5 due to the excessive overhead of such an implementation.

Integrating fault tolerant designs in every microprocessor module would require longer design and verification cycles. It becomes less attractive to build custom designed fault tolerant processors for large systems. On the other hand, it is possible to build identical high-end processors and to use n-redundancy processor cores checking for majority outputs. The Intel Pentium and Hewlett Packard NotStop architectures used such inter-processor checks for data integrity [30]. In the Hewlett Packard *Non-Stop Advanced Architecture* (NSAA), the fault tolerant cluster uses processor pairs or TMR configurations in Itanium servers to check for consensus or majority processor outputs. Then, the result will be stored through a *system area network* (SAN) via messages under CRC checksum protection [3]. The NSAA design does not require traditional lockstepped processors where two processors must agree on every instruction, as the result is checked only through a carefully designed I/O interface and allows non identical commodity microprocessors to work together.

In embedded systems there is also a need for low cost and high error tolerant devices that work in extreme environments. An embedded soft error tolerant ARM processor was proposed [4]. The design uses both a *Register Value Cache* (RVC) and previously-introduced shadow latches. The RVC guards the processor register file and shadow latches guard processor state registers. The RVC block relies on locality of reference, maintaining duplicate copies of the most recently used register data. The RVC contains CRC check bits in case of disagreement between the cached value and the register value. It conducts a CRC check on the previously-read value. If the CRC check fails, then the error signal is asserted and the buffered value from the register file is identified as the correct value, otherwise the buffered value from the cache is assumed to be the correct value. For state

registers, every shadow latch incurs power and area overhead on the design. The placement of these hardened registers is determined after performing a Monte Carlo fault injection simulation. The top n registers that are more likely to have faults are replaced by shadow latch protected cells.

2.6 Summary – Best of Prior Methods

So far, we investigated many important approaches to soft errors. Approaches to the problem include developing better fabrication processes, resizing transistors, modifying circuit topologies, and using error correcting and detecting codes. These approaches have their strengths and limitations. Many of these designs model the soft error as a single SEU. Especially in the circuit topology category, this hinders the development of MBU tolerant cell designs. In techniques for fabrication processes and transistor sizing, the designs are too process dependent and cannot easily adapt to technologies other than the silicon fabrication process. Considering the scalability, error correcting codes are the most flexible solution for MBUs. The best prior work is XSEUFF2 due to the reason that it protects the cells with static logic while most of the other approaches use dynamic logic in their design. In the next chapter, we will detail the reason why some techniques that are superior in one fault model can be inferior in another fault model scenario.

Chapter 3
Multiple Bit Upsets

The prior work on soft error tolerance focuses on mitigating SEUs and SETs. These techniques are very effective in recovering from or masking these types of errors. Nevertheless, many designs cannot easily adapt to tolerate MBUs. The problems generally reside in the excessive hardware overhead. Some designs are also limited by cell level optimization and cannot reliably guard against MBUs unless they are completely redesigned. Much of the prior work assumes that the possibility of two errors in a single cell is extremely unlikely [33]. However, the result of more recent research shows that simultaneous errors in 90 nm memory cells can be twice as high as expected [15]. The possibility of MBUs further increases if ions intersect with the bulk at larger angles. For smaller technology nodes, it is reasonable that high energy particles are capable of traveling through several well regions before they come to a full stop. Figure 3.1 shows two flip-flop designs in 90 nm and 70 nm. The 70 nm node technology cell is roughly 3.3 $\mu m \times 9$ μm and the 90 nm cell is roughly 4.2 $\mu m \times 11.2$ μm. This results in areas of 29.7 μm^2 and 47.04 μm^2, respectively. The ratio of areas is 1.5838. It suggests a much larger affected radius upon a particle strike even for two successive technology nodes. New soft error mitigation techniques should be able to reliably handle MBU problems. Our research also takes a step forward and focuses on designing MBU tolerant hardware.

The conventional fault detection techniques store charges in multiple nodes and check for the consensus logical level of these nodes. Most of these topological approaches require redesigning for different technology nodes and lead to a scalability problem. Only later designs such as BISER and shadow latching allow

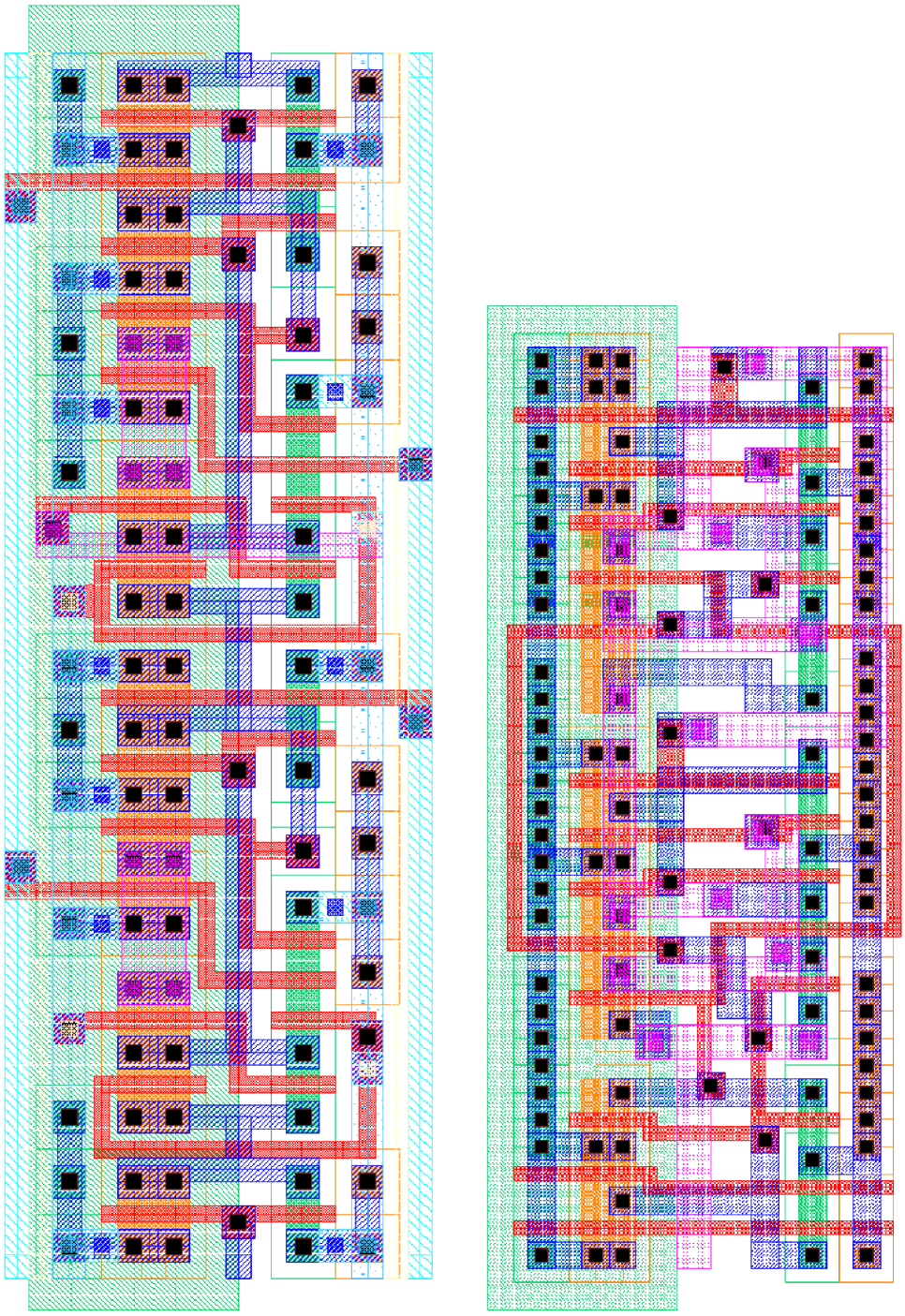

Figure 3.1: Relative sizes of flip-flops in 90 nm and 70 nm. The left cell is a flip-flop for the 90 nm node and the right one is for the 70 nm node.

easier adaptation of multiple-bit detection and correction. The design can be extended by adding more latches for comparisons in each cell. This provides a simple tradeoff between the chip real estate and the desired hardware reliability. Using Mitra *et al.*'s work [23] as an example, we can estimate the transistor overhead. To detect MBUs, there must be at least one master latch to hold the correct logic value. Thus, for an n-redundancy fault detecting flip-flop, it can, at most, tolerate $n-1$ SEUs. The number of transistors needed for fault detection will be the count of transistors in each master latch, the slave latch, and the C-element. The latch design has 8 transistors each and the C-element will have $2 \times n + 4$ transistors, where n is the number of redundancies. To correct multiple-bit upsets, the nodes with correct logic levels must win by majority voting. Thus, there must be at least $n/2 + 1$ latches to hold the correct logic values. For an n-redundancy fault correcting flip-flop, it can, at most, tolerate $(n-1)/2$ SEUs. The transistors that are needed for fault detection will be the sum of transistors in each master latch, the slave latch, and the majority voter. The latch design has 8 transistors each and the majority voter will have $5 \times n$ transistors. As shown in Figure 3.2, the number of transistors in each cell grows linearly with the number of tolerable faults. The delay overhead will also increase as the voter grows in size.

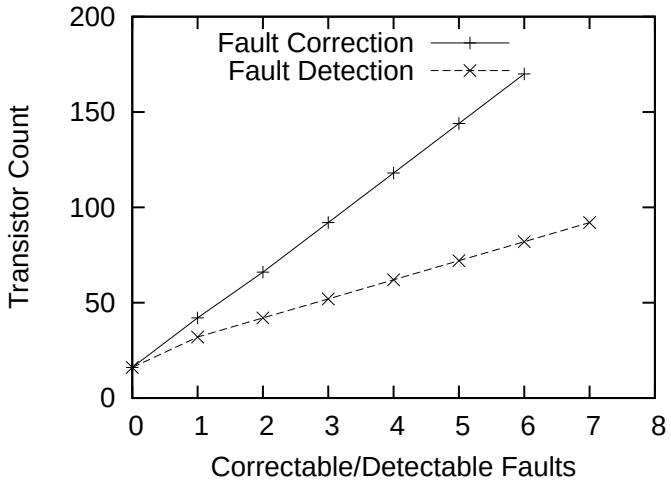

Figure 3.2: Protected flip-flop's transistor count

3.1 Spacial Locality of Multiple Cell Upsets

There is no better solution than using n-redundancy checking for a multiple-ion-induced MBU. However, we can exploit the locality property of a subcategory of MBU that is called a *mulitiple-cell upset* (MCU). In a MCU, the failing bits are usually adjacent [14]. The reason for MCUs appearing as a cluster of multiple upsets is because they are the result of a single ion affecting multiple nodes. MCUs are difficult to correct with conventional error correcting techniques. Because most topological fault tolerant techniques rely on the neighboring nodes to recover a faulty node charge, the overhead can be enormous for n-redundancy techniques. We find the code redundancy approaches to be more attractive. Instead of storing the current state in multiple nodes within the same register cell, the encoding process encodes a number of registers and stores code information in distinct cells. In smaller registers, the MCU effects are more severe because the percentage of MCUs in MBUs is given by:

$$E_{MCU}\% = e^{-E_{SRP} \times \frac{AdjCell}{N}} \tag{3.1}$$

This equation is an adaptation of the work of Gasiot *et al.* [14]. Here E_{SPR} is the number of SEUs recorded after irradiation, $AdjCell$ is the number of cells around each SEU that are inspected to detect a MCU, and N is the size of the memory array. $AdjCell$ is also a function of cell spacing, which relates to cell dimensions. When the error rate and the cell spacing remain constant, if the size of a memory array reduces, then the percentage of MCUs increases. The percentage of localized errors in a smaller memory array such as a register file is higher and this necessitates the codes that correct burst errors.

3.2 Conclusion

It is infeasible to modify flip-flops bit-by-bit to make them all fault-tolerant to MBUs. The solution does not scale. Using code theory on a group of flip-flops and taking advantage of the locality property of MCUs is a better approach. In the

next chapter, we will introduce a hybrid approach that combines both traditional cell level SEU tolerance and a simple parity check on group of cell.

Chapter 4
MCU-Tolerant Hardware with Hybrid Techniques

4.1 Proposed MCU-Tolerant Design

We chose a hybrid technique to implement our registers. The technique uses SEU correcting scan flip-flops to correct all SEUs, and uses a parity checker to detect MCUs. The implementation of this detection and correction hardware is different from the conventional SEC-DED Hamming code. The implementation guarantees to correct any combinations of MBUs as long as they occur in different flip-flop cells, and to detect a MCU in n protected registers, where n is the flip-flop count in a parity group. The group size of n directly affects the MCU detection ability of our technique because a parity check will miss an even number of errors. If n is larger, the possibility of double MCUs increases. It is possible to implement other types of checkers to detect MCUs. Codes such as Hamming codes and CRCs are suitable substitutes to a simple parity check. These codes are fairly simple to compute and can detect multiple inter-cell errors. However, one of the drawbacks of using these more advanced codes would be the checker complexity and the hardware overhead.

4.2 Experiment

We grouped 4 bits of SEU correcting scan flip-flops into a parity check group. The design is shown in Figure 4.1. Counting the transistors versus 4 regular scan flip-flops resulted in a hardware overhead of 82.5%. The scan flip-flops were connected as a 4-bit shift register. The variable delay buffers between flip-flops

were components that emulated the combinational logic between states. Signals $Capture$, $Update$, and $Test$ were scan control signals. In the operation mode, flip-flops read the $Data$ input and write the output Q to $System\_Out$. In the scan mode, flip-flops read SI and write output to $Scan\_Out$. Since the SEU correcting flip-flops were capable of correcting all SEUs in each cell, there was a need to inject more than two SEUs to a cell. So, two error signals SCA and SCB were wired into each cell. $Q1$, $Q2$, and $Q3$ were the latched outputs of the first, second, and third flip-flops, respectively. In the experiment only the third cell had SCA and SCB signals internally wired to the shadow latches. Figure 4.2 shows a SET response of the system. At 550 ns on the x axis, $Data$ switched almost simutainously with Clk and this signal change during the flip-flop's hold time is captured as a SET. The $Error$ signal was asserted accordingly. Figure 4.3 shows the SEU response of the system. At 390 ns on the x axis, a single error pulse entered the third flip-flop. The flip-flop corrected the error immediately. Figure 4.4 shows an MCU response of the system. At 390 ns, two errors in a SEU correcting flip-flop resulted in an erroneous correction in the third flip-flop. The parity check detected the problem and asserted the $Error$ signal indicating there was an uncorrectable error in the group. These simulations were performed with 70 nm transistor models and we used the Cadence SpectreS simulator for analog simulations. We have proven that combining SEU scan latches with the parity group can mask or at least detect MBUs.

4.3 Design Limitations

The hybrid design that uses SEU tolerating flip-flops and a parity checker is a simple concept to handle MBUs. However, the detection ability of MCUs is limited to one cell error per group. The hardware overhead is kept under 100% but there is a need for better checkers with the detection abilities covering multiple errors. In order to keep the detection ratio high, it is inevitable that one must have small parity groups. This leads to a further wiring problem when error signals

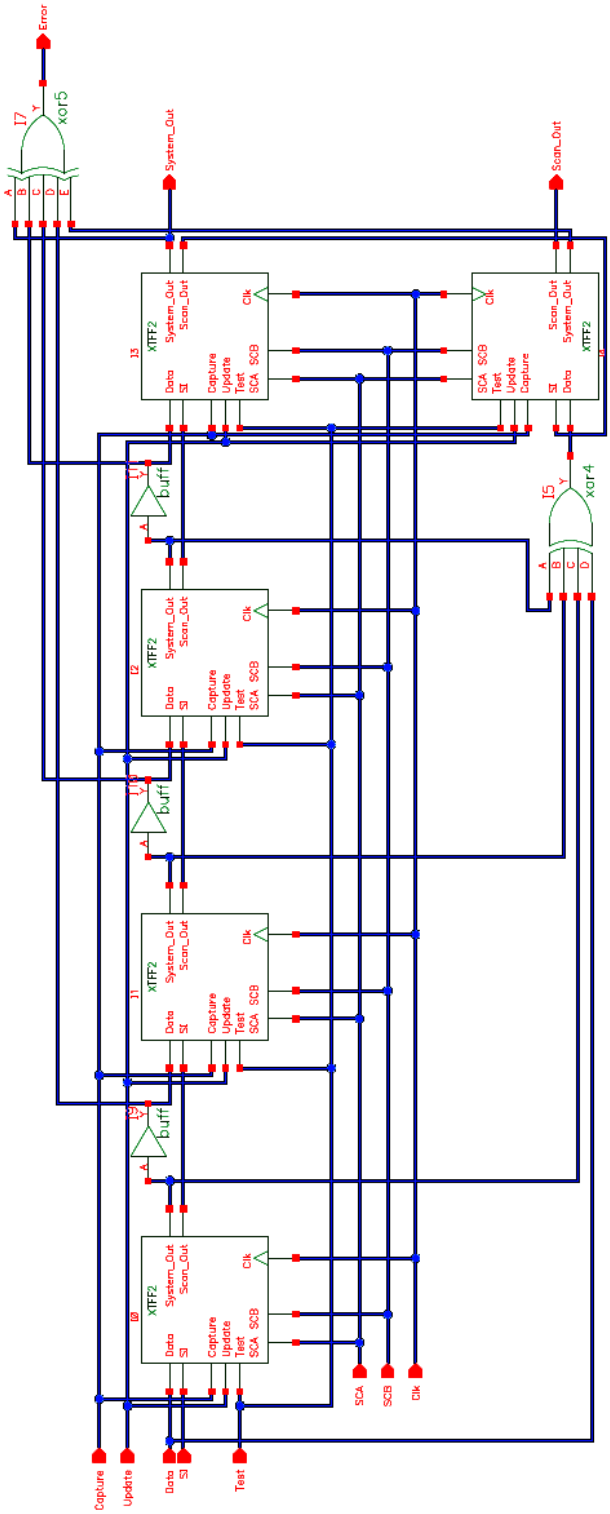

Figure 4.1: 4-bit shift register with MCU protection with a single even parity bit

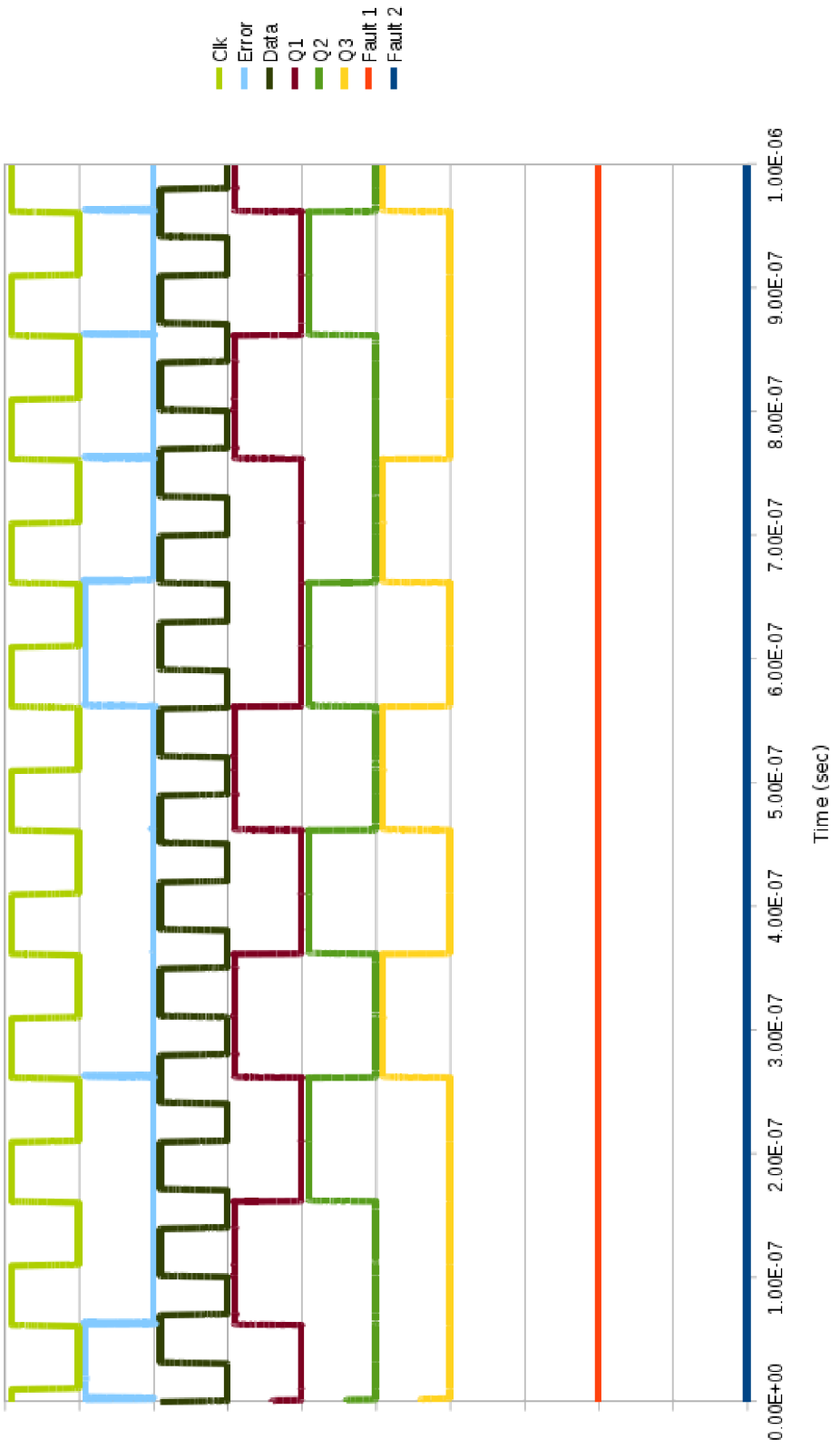

Figure 4.2: A SET affecting system outcomes

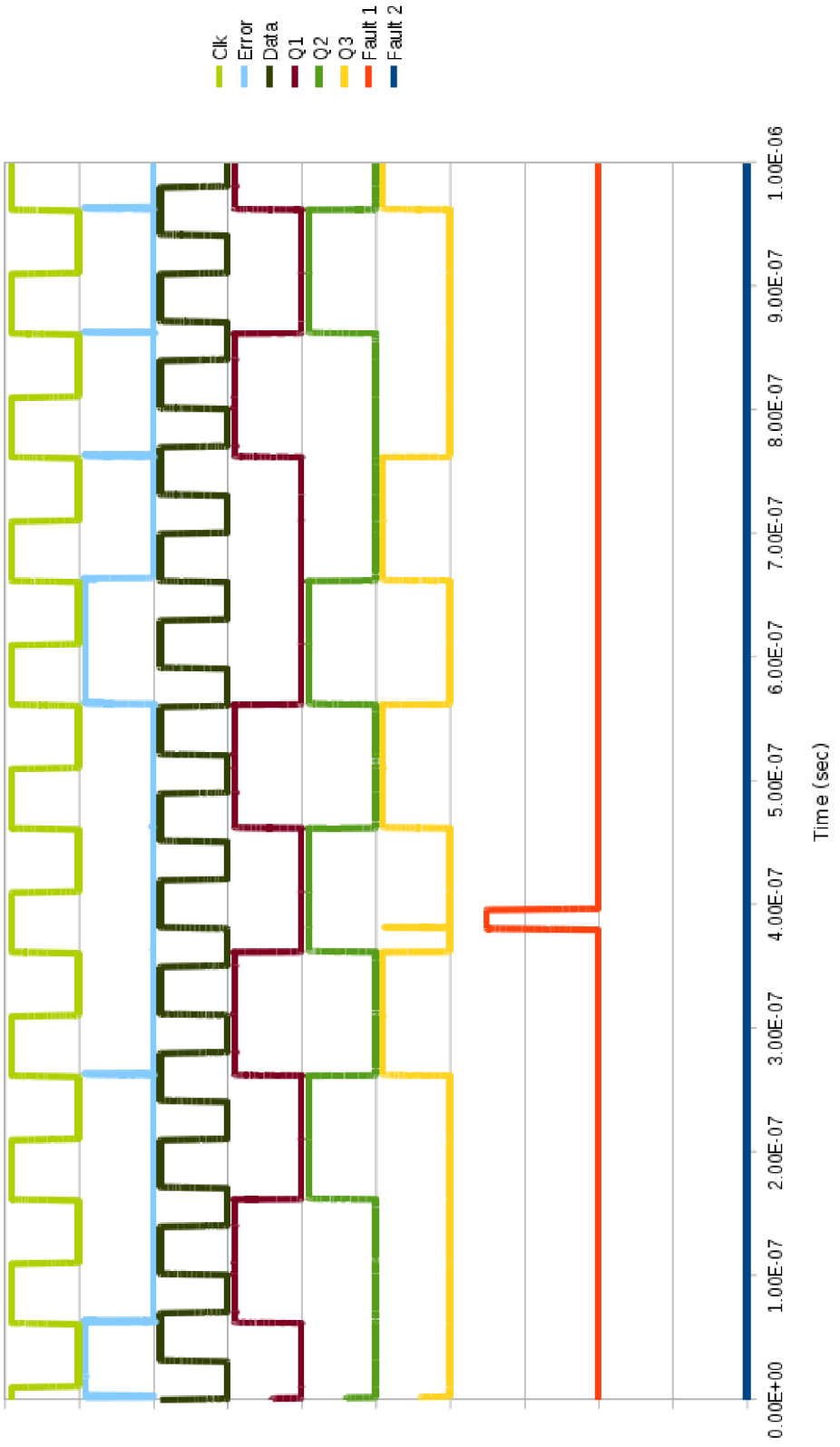

Figure 4.3: A SEU affecting system outcomes

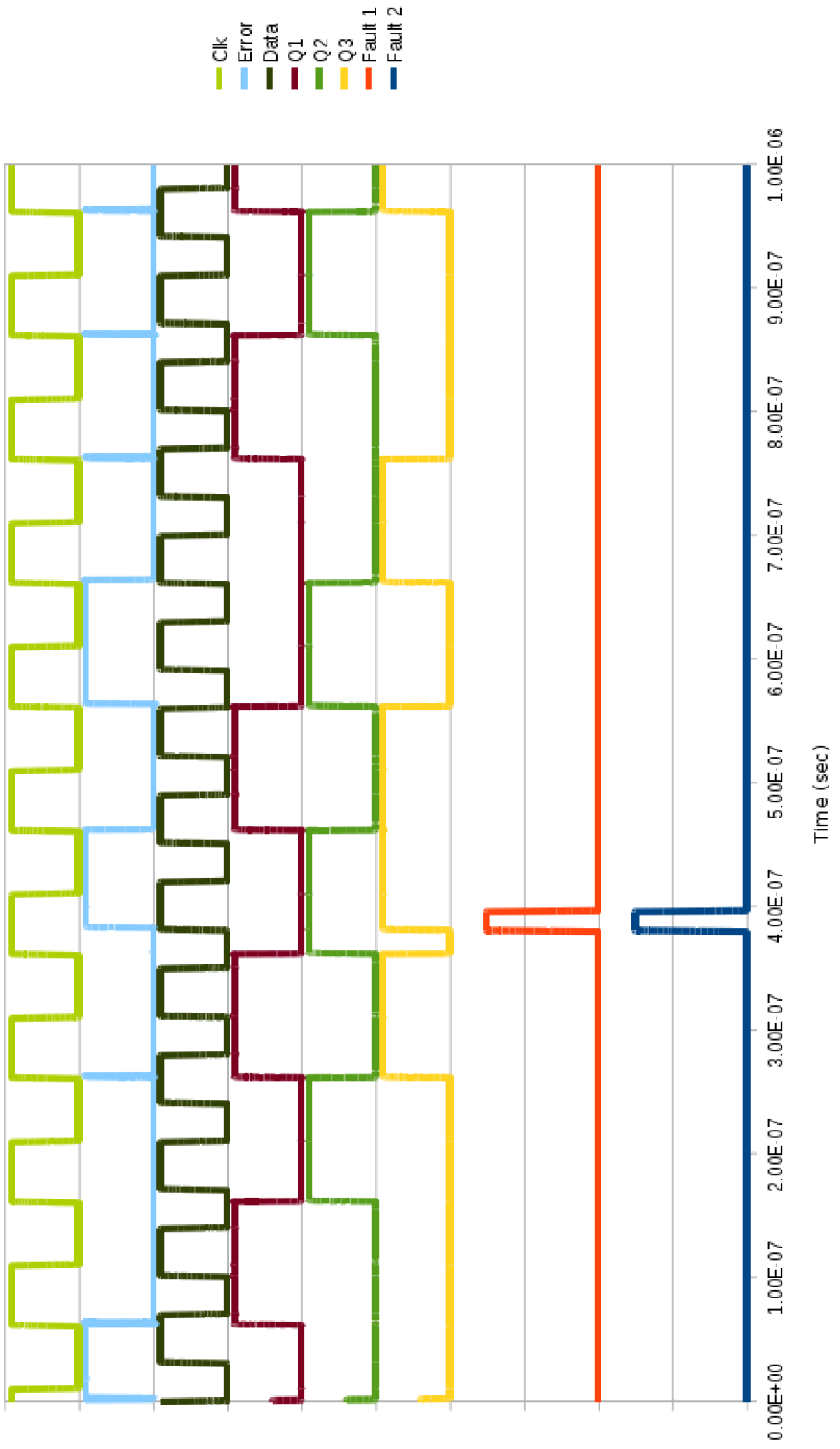

Figure 4.4: A MCU affecting system outcomes

from each group must wire to reset logic in the system. Also, if the checker can detect many MCUs, it is possible to use regular flip-flops instead of SEU tolerant flip-flops and letting the check code handle all of the errors.

4.4 Conclusion

The parity bit protection alone is not enough for MBUs. It can only protect against odd numbers of errors, but not even numbers of errors. Combined with traditional cell-level SEU tolerant techniques, the parity bit is tolerant to odd numbers of MCUs. It uses low overhead, but it still provides insufficient protection against MBUs created by individual particles. In the next chapter, we introduce a protection scheme without using SEU tolerant flip-flops and providing sufficient protection from all MBUs.

Chapter 5
MBU-Tolerant Hardware Using CRCs

5.1 Proposed MBU-Tolerant Design

We choose the CRC for creating MBU tolerant registers because among all prior work, code redundancy designs are more promising for MBU detection considering the scalability and the ability to handle localized MCUs as mentioned in Section 3.1. We use the CRC for simple error detection and use software roll back to correct the corrupted state outcome. A high level block diagram is shown in Figure 5.1. In the design, the checker registers are appended to a group of registers. A pair of a CRC generator and a CRC checker will generate the check bits for the next cycle and check errors for the current cycle. Upon observing an error, the checker will raise the error signal *RegErr* high to indicate that the register has a non-matching data and check values. Responding to an error signal, the system enters the ABORT state and attempts a software roll back, then it returns to the previous state for a retry. Our design differs from Blome *et al.*'s *Register Value Cache* (RVC) [4], as we do not keep extra registers for the data recovery. This is because the RVC scheme requires one of two registers to hold the correct register value. In the case where both registers are corrupted, the checker will believe that the register file holds the correct value and the error will remain undetected. In this chapter, we will address errors in a contiguous sequence of bits as a *burst error*. The length of a burst error is the number of bit errors in a frame from the first error to the last, inclusive.

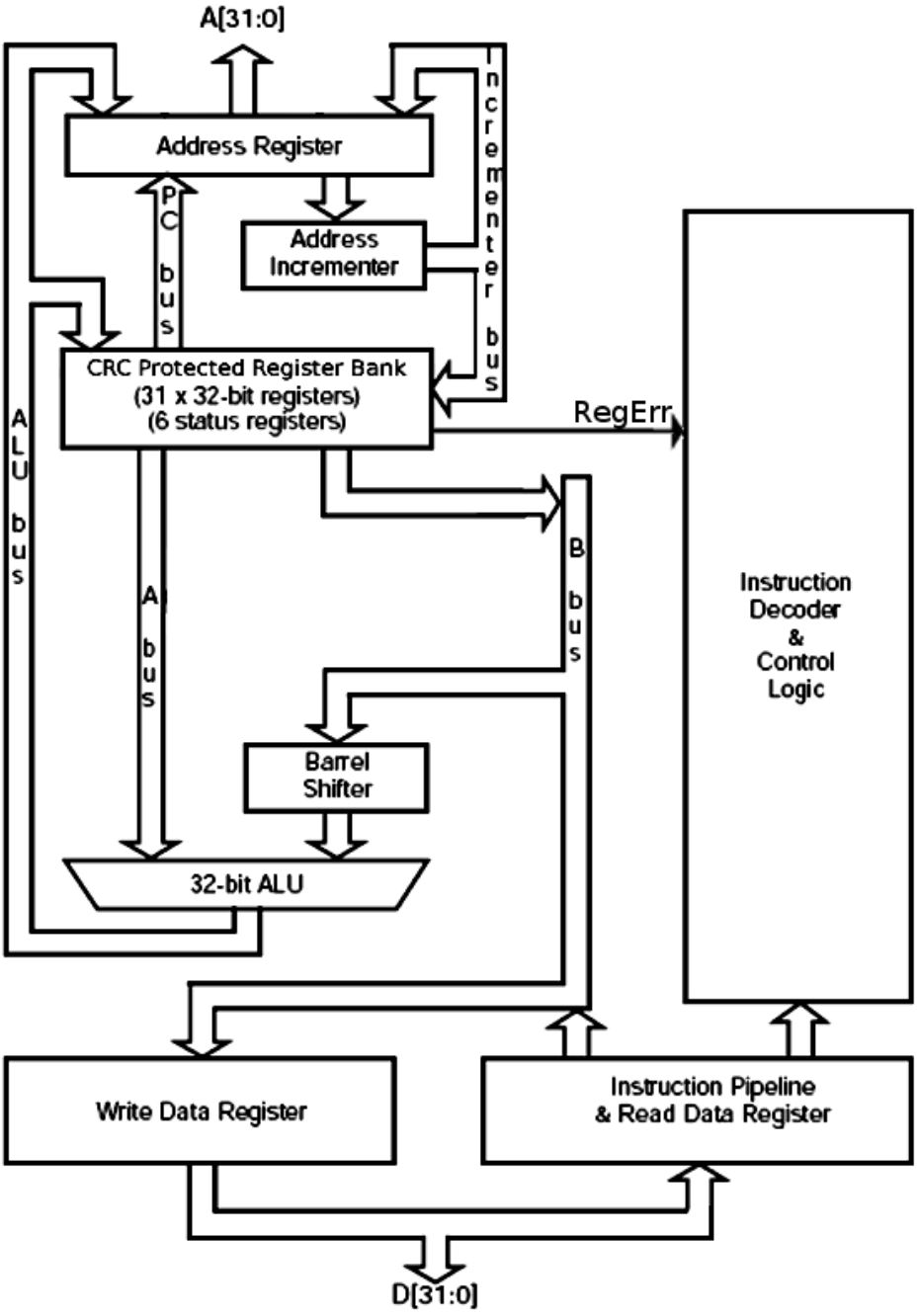

Figure 5.1: Block diagram of MBU detecting ARM processor

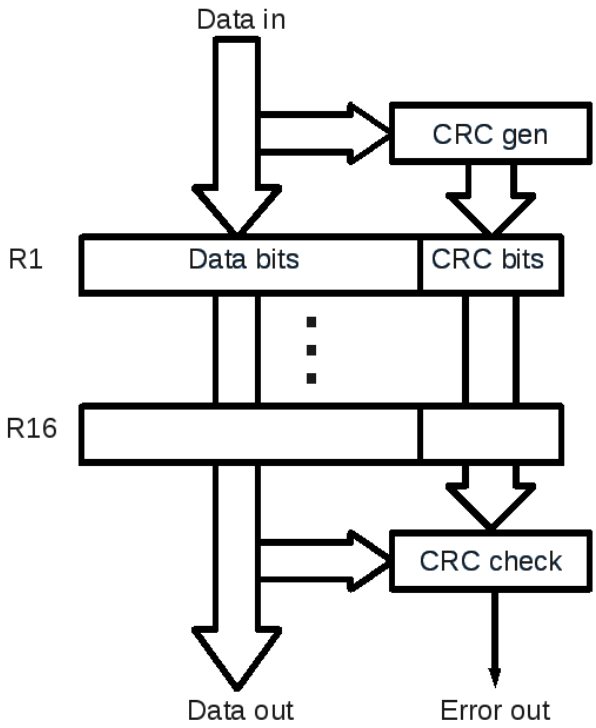

Figure 5.2: Architecture of MBU detecting registers

Architecture

We implemented an 32-bit ARM processor based on the ARMv4 archecture and instruction sets without the co-processor instructions. Figure 5.2 shows the architecture of our MBU detecting registers, where we compute a CRC on the register contents and store the CRC on-chip in bits attached to the register. If the check bits are protected from any errors, they always detect the theoretical *burst error* length. Thus, the CRC bits should be stored in soft error protected cells with the traditional ECC or topologically hardened registers. Fortunately, as observed from our experiments, the CRC code still functions near the optimal burst size detection without such protection. The degraded detectable burst size is typically one to two bits shorter than the theoretical one. It is a designer's decision whether to allow a better burst error detection, or to save chip area and power by not performing an ECC on the CRC check cells. To show a fair comparison against Hamming codes, we show experiments with the assumption that CRC check bits are equally vulnerable to soft errors as data bits are. Even without

an ECC on the CRC bits, all of our CRC checkers have excellent burst error protection. The algorithm for these registers for a write to R_n is to compute the CRC code $CRC_{reference}$ for that register, and store it in the register bits. When reading R_n, the hardware recomputes the CRC_n on the retrieved data, and compares it to the stored $CRC_{reference}$ in the register. The CRC checking can be pipelined, so that the retrieved register data is immediately forwarded to the processor or the controller using it, concurrently with CRC checking. When $CRC_n \neq CRC_{reference}$, the processor will enter the ABORT state. The controller pipeline is flushed, and within the ABORT state, a subroutine will perform software rolled back to the last checkpoint. The system then exits the ABORT state and returns to the previous processor state for a retry. With our design there is the additional expense of maintaining checkpoints of the computation, but the great virtue of this architecture is that delay overhead is only incurred when an error actually occurs, and not routinely. Also, this method avoids the heavy hardware overhead of error correcting codes, and is much more suitable for mobile electronic devices.

5.2 Experiment

We compared different fault detection codes to find the most suitable code for our purpose. The detection mechanism was optimized for soft error detection only, where the errors are intermittent and do not damage the physical hardware. Several popular error detection codes in memory systems were in the testbench for comparison. The designs then passed through the hardware compiler of Synopsys Design Vision and we estimated the checker overhead from the obtained *Register Transfer Level* (RTL) models. These experiments were the first step for us to make a decision on which candidate was more suitable for constructing a fault detecting ARM processor register. Later on, we implemented a fault detecting ARM processor in Verilog. Based on our initial findings, the system was then tested under various levels of soft errors and the responses were recorded. The

simulation was done mainly on a behavioral model and we used the *Verilog Procedural Interface* (VPI) to automatically insert faults to the hardware. The results obtained were grouped as random errors and burst errors. Within a register, the random error count was the count of total errors within a single register. We counted 2-bit flips on the same bit as two errors even if they nullified the fault. The register results were compared with an ideal register simulated with no error injected. After simulating for a hundred thousand iterations, we recorded the count of missed faults. In burst error simulations, the error ranges were further limited to the burst error length. Since a MCU does not necessarily affect every register cell because of the cell placement, we modeled MCUs differently from the standard definition. In our model, the furthest bit error pair occurring n bits apart would be counted as a length n burst error regardless of whether there were non-corrupted bits between the pair. The simulation ran for a hundred thousand iterations and the missed faults were recorded with a similar technique to measure random errors.

For Hamming codes, we picked the most common Hamming distance-3-code and distance-4-code. There was no implementation of the correction mechanism, but the codes simply detected the minimal simultaneous 2 or 3 bit errors, respectively. For the CRC, we picked some commonly-used CRC polynomials such as CRC-6-ITU ($P(X) = X^6 + X + 1$), CRC-7-SD ($P(X) = x^7 + x^3 + 1$), CRC-8-ATM ($P(X) = x^8 + x^2 + x + 1$), and CRC-12-TELECOM ($P(X) = x^{12} + x^{11} + x^3 + x^2 + x + 1$). For IBM's symbol code, we directly mapped the code table of the G4 server and obtained the SSC-DSD (76,64) code. The simulation was done by Verilog behavioral simulation. For research purposes, the faulty word group size is set to 8 bytes (64 bits) and the undetected fault was recorded for simultaneous upsets of 1 to 10 bits. We compared the results in Figure 5.3. On the log scale, if there is no undetected event, then it is not shown on the graph. Thus, all of the distance 2 codes will not have the first upset bit plotted since the error will always be detected, and all the distance 3 codes will not have the first

2 upset bits plotted, etc. Codes such as the CRC-12-TELECOM code and Hamming (72,64) code always detect odd errors, so any odd numbers of errors do not appear on the graph. The figure shows that SSC-DSD and CRC-12-TELECOM had the least number of undetected errors when the error count exceeded the codes' minimal Hamming distance. The result also shows that similar check bit lengths resulted in similar levels of undetected faults. All of the codes other than CRC-6-ITU had at least a distance-3 property, where all 2-bit errors were detected.

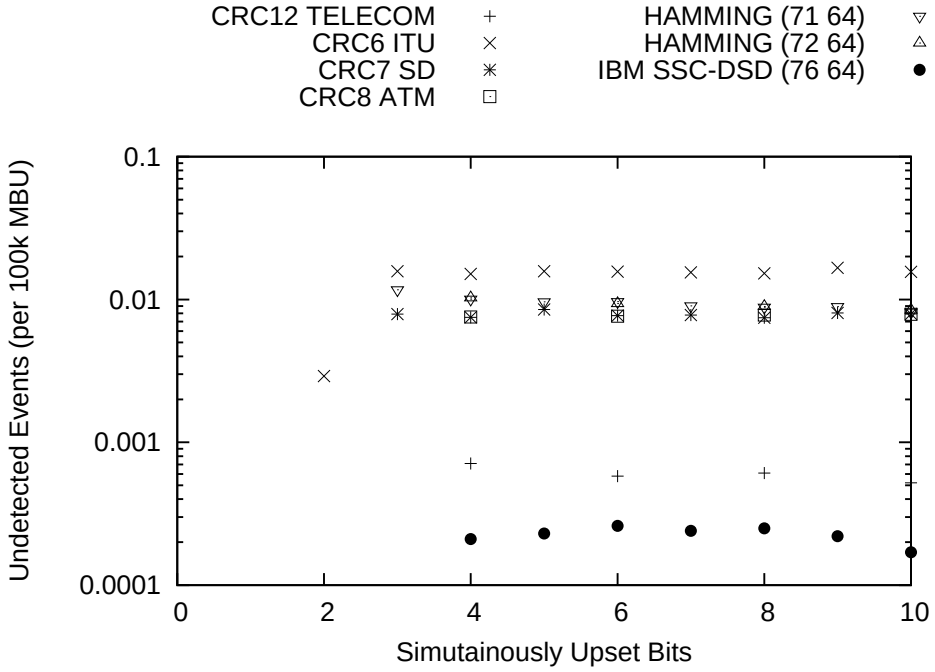

Figure 5.3: Random errors from 100K simulation runs

The next experiment was on burst errors. We set up the experiment so that check bits and data bits were both prone to soft error corruptions as in previous experiments. The burst starting position was randomly selected and there were cases where both data bits and check bits fell under the same burst error. In Figure 5.4, the CRC technique was obviously superior to both Hamming codes. On the log scale, if there is no undetected event, then it is not shown on the graph. Thus, we find the first point of a code that appears on the figure and subtract it by one to obtain the maximum guaranteed detectable burst error

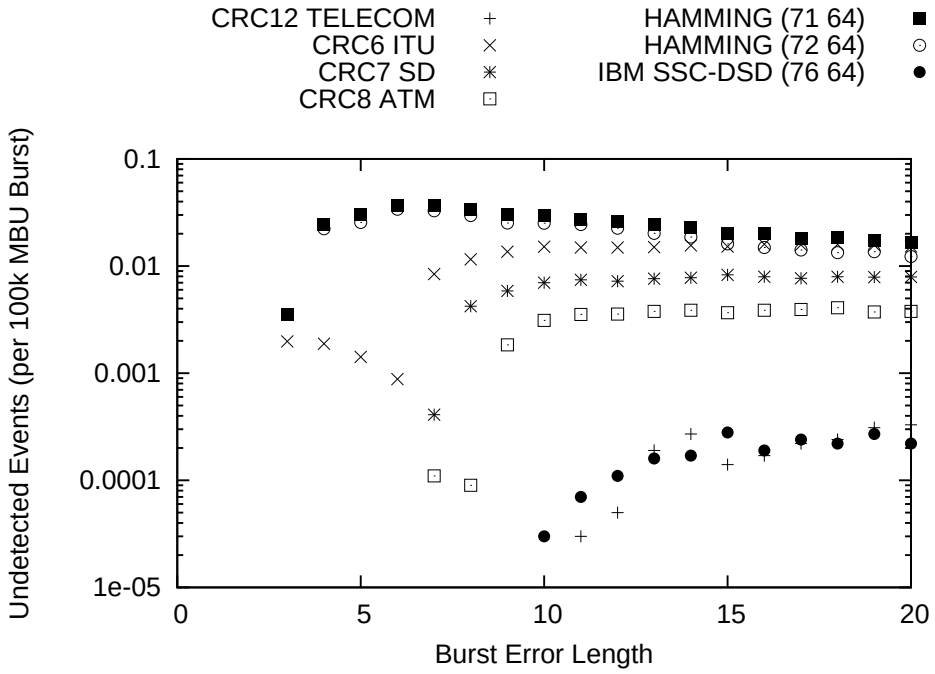

Figure 5.4: Burst errors from 100K simulation runs

length. For the CRC-12-TELECOM code, the burst error detection is better than for the IBM SSC-DSD (76,64) symbol code. Again, theoretically the CRC should have the ability to detect burst errors less than the order of the highest CRC polynomial, but since we allowed the check bits to be corrupted by a burst error, the performance was slightly degraded.

5.3 Hardware Overhead

We should observe the hardware overhead of the different codes after knowing the efficiency of them. While we are aiming to detect errors in general purpose registers and pipeline registers, the register overhead can be drastically different from that of a memory system fault tolerant design. One major design difference is that the size of a processor register is significantly smaller than that of a memory bank. Constructing a syndrome decoder for the error correction might not be feasible in terms of hardware overhead. Also, the instruction and data are cached hierarchically in lower level memory devices. As long as they are accessible to

the processor, the data recovery can be done by rolling back and re-fetching the non-affected data. Most of the soft error rate is measured in the order of events per billion hours. It might be more economical to roll back and recompute results than to add extra hardware for correction. The performance gain of having dedicated correction hardware is negligible over billions of hours. Since the register size is small compared to a memory, the fault detection checker's size becomes significant. Unlike memory banks where designers only worry about the check bit length, we should consider the checker hardware overhead as well. We used Synopsys Design Vision to generate the hardware overhead of our checkers in Figures 5.6 and 5.7. The checker sizes have positive correlation with how many check bits each code requires. In our experiment, all of the CRC checkers are parallel checkers that can perform all checking in one cycle and not shift registers that require multiple cycles to compute results. The parallel CRC checker is designed based on the *time expension* of sequential circuits. We build a parallel CRC checker by unrolling the LFSR sequential logic to a combinational circuit. The design is similar to Sprachmann's technique [31], but we perform unrolling at the block level. In order to balance between the logic depth and logic fanout, we generate optimized checker blocks with a parallel CRC algorithm developed by Campobello *et al.* [7]. Then, we connect these blocks to form a complete parallel CRC checker. Figure 5.5 is a gate level CRC-7-SD code generator that is created by our procedures and optimized with Synopsys Design Vision. The Hamming codes are generated with Hamming's work [17] and then are optimized with Synopsys Design Vision's logic optimizing functions. For IBM SSC-DSD codes, we directly use the available generator matrix [10] and optimize the logic with Synopsys Design Vision.

The results show the strongest CRC code, the CRC-12-TELECOM code has 3.9% less hardware overhead compared with the IBM SSC-DSD code. In random error detection, the IBM SSC-DSD code misses 25.8% fewer errors than the CRC-12-TELECOM code. In burst error detection, the CRC-12-TELECOM code misses 0.5% fewer errors than the IBM SSC-DSD code. Although on average the

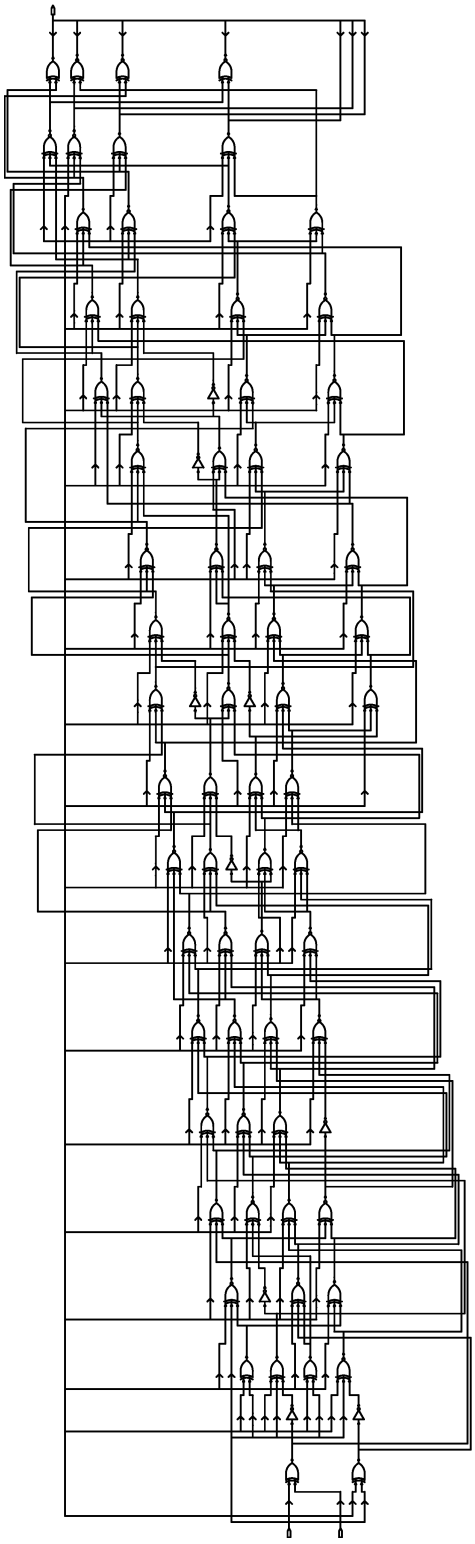

Figure 5.5: The CRC-7-SD code generator with 64-bit parallel inputs. The bottom ports are the 7-bit CRC seed in and the 64-bit data in; the top port is 7-bit CRC out.

burst error detection rates of both techniques are almost equivalent, the CRC-12-TELECOM can withstand a burst error length of 10 bits before missing an error while the IBM SSC-DSD code can only withstand 9 bits. The CRC-8-ATM code is compared with the distance 4 Hamming code, and has 10% more transistor overhead than the Hamming code. However, the CRC-8-ATM code misses 19.7% fewer errors and misses 88.2% fewer burst errors than a distance 4 Hamming code.

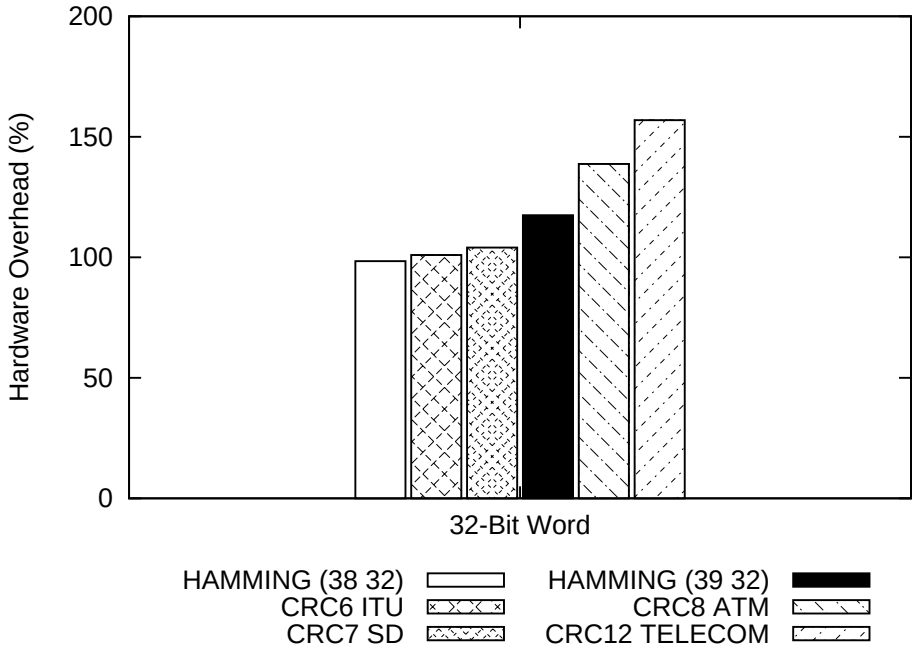

Figure 5.6: Overhead on a 32-bit word

5.4 Fault Detecting Processor

In our 32-bit ARM processor, we picked the CRC-7-SD code for general register protection. The CRC-7-SD code Hamming distance was equivalent to a distance-3 Hamming code. It used the same number of check bits as a distance 3 Hamming code but the checker overhead was slightly larger. The CRC-7-SD also detected up to 7-bit burst errors when CRC bits were protected, which was another benefit over a simple Hamming code. Unlike pipeline registers and state registers, in a register file it was possible to reduce the overhead by sharing checkers similarly to

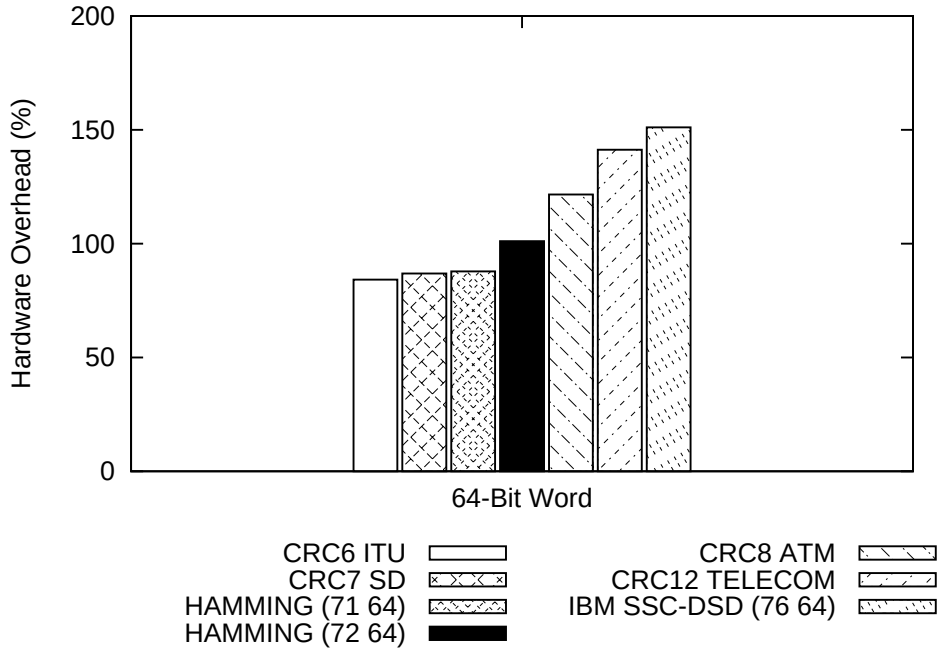

Figure 5.7: Overhead on a 64-bit word

a memory ECC design. This alleviated the 100% checker overhead problem in register file design. The register had two read ports and one write port and shared one CRC generator and two CRC decoders. We also protected the program counter, the current program state register, and saved program state registers individually. Together, we shared four CRC generators and five CRC decoders among 37 registers. We compared our CRC-7-SD with a distance-4 Hamming code and observed the processor's responses. To test the protected register processor, the processor calculated the Fibonacci number under soft errors. The instructions performed calculations of the largest 32-bit Fibonacci integer and saved the entry to the memory. The calculation took roughly 70 clocks to complete. We manually injected burst errors of length from 1 to 10. Without exhaustively simulating all possible error combinations, a Monte Carlo simulation was performed by injecting faults at a random cycle in a randomly picked register during each 70-clock calculation. Every 15 additions, we added 2 store instructions to save the current and previous Fibonacci numbers. When an error was detected, the processor entered ABORT state and performed 2 load instructions to load the stored current and

previous Fibonacci numbers. The routine would gracefully exit ABORT state for a retry if no more errors occurred during the roll back routine. The results were recorded in Figures 5.8 and 5.9, which show that the CRC was significantly better than the distance 4 Hamming code in burst error protection. In the random error simulations, the CRC performance was slightly better than the distance 4 Hamming code with an exception when the error is exactly 3 bits. The chosen polynomial CRC-7-SD was not a multiple of $X + 1$, and the CRC code in triple error detections was not as strong as a distance 4 Hamming code. The CRC-7-SD code hardware overhead is 6.0% smaller than for the distance 4 Hamming code. The random error detection miss rate of CRC-7-SD is 1% less than the distance 4 Hamming code and the burst error detection miss rate is 88% less than the distance 4 Hamming code.

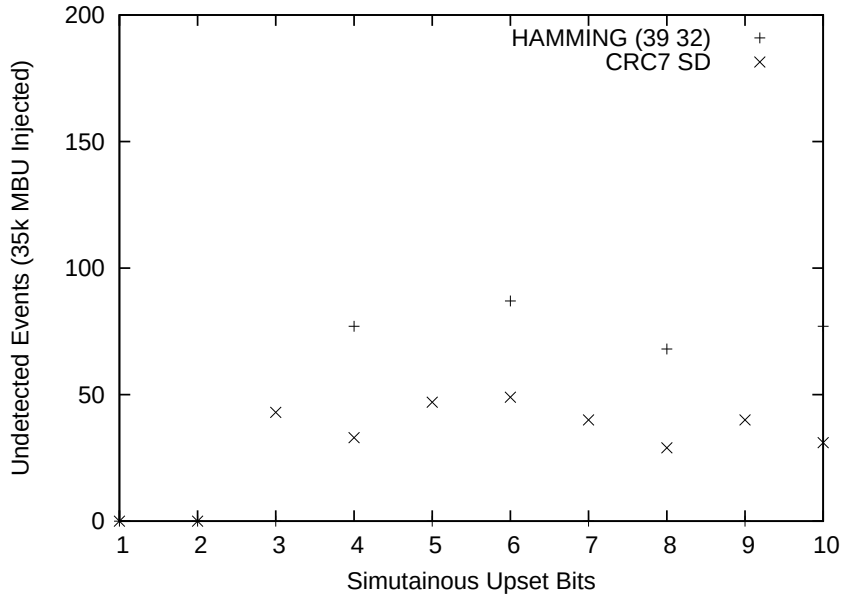

Figure 5.8: Random MBU affecting system outcomes

5.5 Conclusion

The CRC-12-TELECOM code has 3.9% less hardware overhead compared with the IBM SSC-DSD code. In random error detection, the IBM SSC-DSD code misses 25.8% fewer errors than the CRC-12-TELECOM code. In burst error

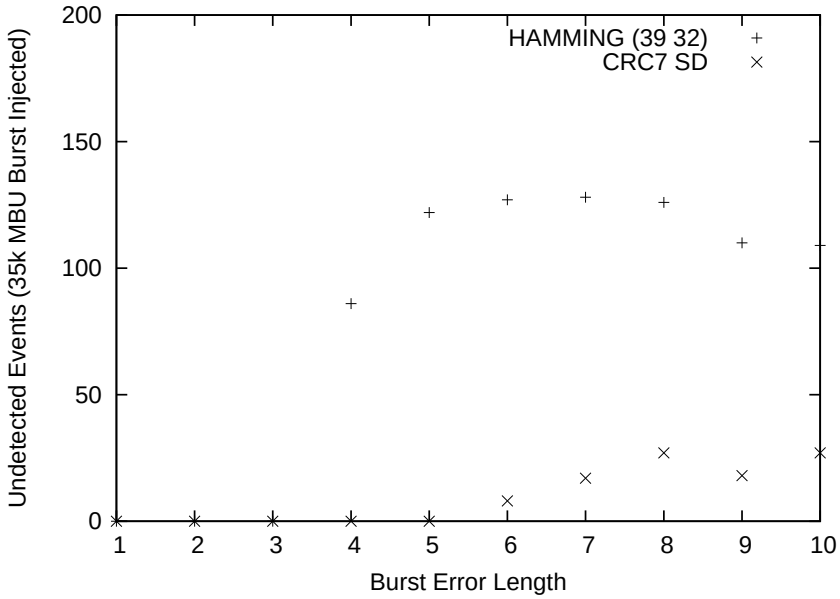

Figure 5.9: Burst MBU affecting system outcomes

detection, the CRC-12-TELECOM code misses 0.5% fewer errors than the IBM SSC-DSD code. Although on average the burst error detection rates of both techniques are almost equivalent, the CRC-12-TELECOM can withstand a burst error length of 10 bits before missing an error while the IBM SSC-DSD code can only withstand 9 bits. Regardless, these codes have a low miss rate, and the excess hardware overhead of them reaches nearly 150%. In the field of reliable enterprise server designs, IBM SSC-DSD codes, which were once commercialized in the G3 and G4 series, are no longer in use in the following G5 series due to the excessive overhead [30]. Thus, we use a distance 4 Hamming code as a benchmark for the rest of our experiments. The CRC-8-ATM code is compared with the distance 4 Hamming code, and has 10% more transistor overhead than the Hamming code. However, the CRC-8-ATM code misses 19.7% fewer random errors and misses 88.2% fewer burst errors than a distance 4 Hamming code. The CRC-7-SD code hardware overhead is 6.0% smaller than for the distance 4 Hamming code. The random error detection miss rate of CRC-7-SD is 1% less than the distance 4 Hamming code and the burst error detection miss rate is 88% less than the distance 4 Hamming code. The CRC checking can be pipelined, so

that the retrieved register data is immediately forwarded to the processor or the controller using it, concurrently with CRC checking. A 32-bit parallel CRC-7-SD checker has 11 gate delays on the critical path and a 64-bit one has 19 gate delays. If the pipelined logic's critical path is shorter than the CRC-7-SD checker's delay then the checker will affect the processor's performance regardless of whether the processor is pipelined.

Chapter 6
Conclusions and Future Work

The code redundancy technique for MBU detection can be very useful when errors are induced by a single α particle or cosmic ray. Using CRCs to detect errors and then recomputing from check points to correct the error is a simpler solution than using expensive MBU hardened register cells. Compared with other fault detecting codes, the CRC is also a better balance between hardware overhead and detection capabilities. The burst error capability shows that CRCs can be an effective substitute for Hamming codes in caches and register files. They have nearly equivalent checker overhead and a flexible check polynomial length that can be chosen during an early design cycle to match the system reliability constrains. Compared with the state of the art IBM SSD-DSD code, CRC polynomials can be chosen to have a better burst error protection but IBM still has the lead in random error detection. In random error detection, the IBM SSC-DSD code missed 25.8% fewer errors than the CRC-12-TELECOM code. In burst error detection, the CRC-12-TELECOM code missed 0.5% fewer errors than the IBM SSC-DSD code. Although on average the burst error detection rates of both techniques are almost equivalent, the CRC-12-TELECOM code can withstand a burst error length of 10 bits before a miss while the IBM SSC-DSD code can only withstand 9 bits. On the hardware overhead of these codes, our chosen CRC polynomial shows a lower checker overhead than the IBM SSD-DSD code. The CRC-12-TELECOM code has 3.9% less overhead compared with the IBM SSC-DSD code. Thus, if chip real estate is a higher concern and multiple faults are modeled as MCUs, a CRC can be chosen instead of the IBM SSD-DSD code.

Compared with the widely accepted distance 4 Hamming code, the CRC-7-SD

code used 6.0% less hardware than the distance 4 Hamming code. The random error detection miss rate of the CRC-7-SD code is 1% less than the distance 4 Hamming code and the burst error detection miss rate is 88% less than for the distance 4 Hamming code. Also, for burst error detection the CRC-7-SD code can withstand a maximum 5-bit burst error while the distance 4 Hamming code can only withstand a 3-bit burst error. A 32-bit parallel CRC-7-SD checker has 11 gate delays on the critical path and a 64-bit one has 19 gate delays. For a highly pipelined processor, the 64-bit CRC checker might be longer than the critical path. One way to solve the timing problem is to use two 32-bit parallel CRCs for a highly pipelined processor.

6.1 Future Work

Generally, a fault detecting register cannot protect against hard faults. The register with faulty hardware will continuously produce an erroneous outcome, and the software rollback will not correct the problem. The design can be improved so that the register file can protect against these stuck faults or other hard faults. With some modification, register renaming hardware can avoid mapping virtual registers to a faulty physical register. This is one way of making the system tolerate hard errors. There is also room for improvement in cell arrangements to minimize hardware overhead. In actual system layouts, the checkers' routing overhead should be considered and a partitioning algorithm should be derived to minimize this overhead.

Appendix A
User Manual

A.1 Programs

We used the Ruby programming language and interpreter [32] to execute these programs on a PC. All available programs are on the Sun UNIX Systems under the directory `/ece/under/wenyueh/code_gen/`.

crcgen_verilog.rb

Usage: **ruby crcgen_verilog.rb [CRC polynomial] [parallel bits]**

Example: **ruby crcgen_verilog.rb "1 0 1 1" 3**

A CRC Verilog code generator for an arbitrary datawidth and check polynomials. The program uses the standard algorithm [7] to generate CRC hardware. The CRC polynomial should be in an array of the polynomial coefficients. The first element should be the zeroth order of the polynomial and the last element should be the highest order of the polynomial. A polynomial $P(x) = x^3 + x^2 + 1$ should be entered as "1 0 1 1" separated with white space. Since the Ruby language contains a *smart parser*, entering "$x3 + x2 + 1$" will also work as expected.

hammingGen.rb

Usage: **ruby hammingGen.rb [parallel bits] [Hamming distance]**

A Hamming Code Verilog code generator for an arbitrary datawidth. The available Hamming distance is distance 3 or 4.

CheckGen.rb

Usage: **ruby CheckGen.rb [generator matrix file]**

Example: **ruby CheckGen.rb "hamming_12_8.csv"**

The program takes in any generator matrix and creates error detection hardware in Verilog. The generator matrix should be a *Comma-Separated Values* (CSV) file containing rows of generator matrix information. In the generator matrix, the first row is the parity check positions, which are separated by commas, of the first check bit, and the second row is the parity check positions of the second check bit, etc. If the check bit i checks position j, we fill in a 1 at the position (i, j) where i is the row position and j is the column position. Finally, we fill 0 in all unchecked positions. For example, the file format of the Hamming (12,8) code is:

1,0,1,0,1,0,1,0,1,0,1,0
0,1,1,0,0,1,1,0,0,1,1,0
0,0,0,1,1,1,1,0,0,0,0,1
0,0,0,0,0,0,0,1,1,1,1,1

autoRun.rb

Usage: **ruby autoRun.rb**

The program runs simulations in batch mode. To simulate the desired checkers, the user can simply drag and drop any previous program-created Verilog files to sub folder `.\codes` and the script will run a batch simulation automatically. The simulated results are time stamped, so invoking the batch script twice will not overwrite the previous result. The simulations

will be saved in folder `.\result`. Since the Ruby language is ported on various systems, these folders should exist and accessible to the user on the relative path of the autoRun.rb file. This prevent the complication of different systems handle file system differently.

A.2 Files

All avaliable files are under the directories `/ece/under/wenyueh/arm_processor_2009/`, `/ece/under/wenyueh/check_unprot_rand/`, and `/ece/under/wenyueh/check_unprot_burst/`.

ff_crc.v

This is Verilog code for a stand alone register with the CRC protection. This module is compatible with CRC checkers generated from the crcgen_verilog.rb script.

ff_hed.v

This is Verilog code for a stand alone register with the Hamming code protection. This module is compatible with Hamming code checkers generated from the hammingGen.rb script.

ff_sed.v

This is Verilog code for a stand alone register with the symbol code protection. This module is compatible with any code checkers generated from the CheckGen.rb script.

arm_*.v

This Verilog code is for parts of an ARM processor. They are necessary components for the autoRunError.rb script to simulate the full processor

with the protected register file.

memory.list

This is the instruction list for the processor. The legal instructions are in global.v. Thumb instructions and co-processor instructions are not available.

faultinject.c

This C library is compiled as a *Verilog Procedural Interface* (VPI) and interacts with Verilog modules. The library uses the Mersenne twister library to generate pseudo-random faults in registers for Monte Carlo simulations.

References

[1] R. C. Baumann. Soft Errors in Advanced Semiconductor Devices – Part I: the Three Radiation Sources. *IEEE Transactions on Device and Materials Reliability*, 1(1):17–22, Mar 2001.

[2] R. C. Baumann and E. B. Smith. Neutron-Induced Boron Fission as a Major Source of Soft Errors in Deep Submicron SRAM Devices. In *Proceedigs of the 38th Annual IEEE International Reliability Physics Symposium*, pages 152–157, 2000.

[3] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. NonStop Advanced Architecture. In *Proceedings of the International Conference on Dependable Systems and Networks. DSN 2005*, pages 12–21, June-July 2005.

[4] J. A. Blome, S. Gupta, S. Feng, and S. Mahlke. Cost-Efficient Soft Error Protection for Embedded Microprocessors. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems. CASES '06*, pages 421–431, New York, NY, USA, 2006. ACM.

[5] R. C. Bose and D. K. Ray-Chaudhuri. On a Class of Error Correcting Binary Group Codes. *Information and Control*, 3(1):68–79, March 1960.

[6] T. Calin, M. Nicolaidis, and R. Velazco. Upset Hardened Memory Design for Submicron CMOS Technology. *IEEE Transactions on Nuclear Science*, 43(6):2874–2878, Dec 1996.

[7] G. Campobello, G. Patane, and M. Russo. Parallel CRC Realization. *IEEE Transactions on Computers*, 52(10):1312–1319, Oct. 2003.

[8] E. H. Cannon, M. S. Gordon, D. F. Heidel, A. J. KleinOsowski, P. Oldiges, K. P. Rodbell, and H. Tang. Multi-Bit Upsets in 65nm SOI SRAMs. In *Proceedings of the IEEE International Reliability Physics Symposium. IRPS 2008*, pages 195–201, May 2008.

[9] E. H. Cannon, D. D. Reinhardt, M. S. Gordon, and P. S. Makowenskyj. SRAM SER in 90, 130 and 180 nm Bulk and SOI Technologies. In *Proceedings of the 42nd Annual IEEE International Reliability Physics Symposium*, pages 300–304, April 2004.

[10] C. L. Chen. Symbol Error-Correcting Codes for Computer Memory Systems. *IEEE Transactions on Computers*, 41(2):252–256, Feb 1992.

[11] T. H. Daubenspeck, J. P. Gambino, C. D. Muzzy, W. Sauter, and E. J. Sprogis. Post Bump Passivation for Soft Error Protection. US Patent #7348210, Filing Date:04/27/2005, Publication Date:03/25/2008.

[12] P. E. Dodd, F. W. Sexton, and P. S. Winokur. Three-Dimensional Simulation of Charge Collection and Multiple-Bit Upset in Si Devices. *IEEE Transactions on Nuclear Science*, 41(6):2005–2017, Dec 1994.

[13] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, Toan Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-36*, pages 7–18, Dec. 2003.

[14] G. Gasiot, D. Giot, and P. Roche. Alpha-Induced Multiple Cell Upsets in Standard and Radiation Hardened SRAMs Manufactured in a 65 nm CMOS Technology. *IEEE Transactions on Nuclear Science*, 53(6):3479–3486, Dec. 2006.

[15] D. Giot, P. Roche, G. Gasiot, and R. Harboe-Sorensen. Multiple-Bit Upset Analysis in 90 nm SRAMs: Heavy Ions Testing and 3D Simulations. *IEEE Transactions on Nuclear Science*, 54(4):904–911, Aug. 2007.

[16] A. Goel, S. Bhunia, H. Mahmoodi, and K. Roy. Low-Overhead Design of Soft-Error-Tolerant Scan Flip-Flops with Enhanced-Scan Capability. In *ASP-DAC '06: Proceedings of the 2006 Asia South Pacific Design Automation Conference*, pages 665–670, Piscataway, NJ, USA, 2006. IEEE Press.

[17] R. W. Hamming. Error Detecting and Error Correcting Codes. *The Bell System Technical Journal*, 29(2):147–160, April 1950.

[18] K. J. Hass, J. W. Gambles, B. Walker, and M. Zampaglione. Mitigating Single Event Upsets From Combinational Logic. In *Proceedings of the 7th NASA Symposium on VLSI Design*, October 1, 1998.

[19] T. Karnik, S. Vangal, V. Veeramachaneni, P. Hazucha, V. Erraguntla, and S. Borkar. Selective Node Engineering for Chip-Level Soft Error Rate Improvement [in CMOS]. In *Digest of Technical Papers of the Symposium on VLSI Circuits*, pages 204–205, 2002.

[20] Y. Komatsu, Y. Arima, T. Fujimoto, T. Yamashita, and K. Ishibashi. A Soft-Error Hardened Latch Scheme for SoC in a 90 nm Technology and Beyond. *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 329–332, 3-6 Oct. 2004.

[21] L. I. Lantz. Soft Errors Induced by Alpha Particles. *IEEE Transactions on Reliability*, 45(2):174–179, Jun 1996.

[22] F. B. McLean and T. R. Oldham. Charge Funneling in N- and P-Type Si Substrates. *IEEE Transactions on Nuclear Science*, 29(6):2017–2023, Dec. 1982.

[23] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K.S. Kim. Robust System Design with Built-In Soft-Error Resilience. *Computer*, 38(2):43–52, Feb. 2005.

[24] R. Oliveira, A. Jagirdar, and T. J. Chakraborty. A TMR Scheme for SEU Mitigation in Scan Flip-Flops. In *In Proceedings of the 8th International Symposium on Quality Electronic Design. ISQED '07*, pages 905–910, 26-28 March 2007.

[25] R. Oliveira, A. Jagirdar, and T. J. Chakraborty. Efficient Flip-Flop Designs for SET/SEU Mitigation with Tolerance to Crosstalk Induced Signal Delays. In *IEEE Workshop on Silicon Errors in Logic – System Effects. SELSE 3*, April 3-4, 2007.

[26] W. Peterson. Encoding and Error-Correction Procedures for the Bose-Chaudhuri Codes. *IRE Transactions on Information Theory*, 6(4):459–470, September 1960.

[27] W. W. Peterson and D. T. Brown. Cyclic Codes for Error Detection. *Proceedings of the IRE*, 49(1):228–235, Jan. 1961.

[28] P. Roche, G. Gasiot, K. Forbes, V. O'Sullivan, and V. Ferlet. Comparisons of Soft Error Rate for SRAMs in Commercial SOI and Bulk below the 130-nm Technology Node. *IEEE Transactions on Nuclear Science*, 50(6):2046–2054, Dec. 2003.

[29] Y. Savaria, N. C. Rumin, J. F. Hayes, and V. K. Agarwal. Soft-Error Filtering: A Solution to the Reliability Problem of Future VLSI Digital Circuits. *Proceedings of the IEEE*, 74(5):669–683, May 1986.

[30] L. Spainhower and T. A. Gregg. IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective. *IBM J. of Research and Development*, 43(5/6):863–873, November 1999.

[31] M. Sprachmann. Automatic Generation of Parallel CRC Circuits. *IEEE Design & Test of Computers*, 18(3):108–114, May 2001.

[32] S. Vinoski. Enterprise Integration with Ruby. *IEEE Internet Computing*, 10(4):91–95, July-Aug. 2006.

[33] J. F. Ziegler. Terrestrial Cosmic Rays. *IBM Journal of Research and Development*, 40(1):19–39, January 1996.