FARHAD: a Fault-Tolerant Power-Aware Hybrid Adder for High-Performance Processor

by

Mohammad Hossein Hajkazemi
B.Sc., Shahed University, 2008

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

© Mohammad Hossein Hajkazemi, 2013
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.

Supervisory Committee

FARHAD: a Fault-Tolerant Power-Aware Hybrid Adder for High Performance Processor

by

Mohammad Hossein Hajkazemi
B.Sc, Shahed University, 2008

Supervisory Committee

Dr. Amiarali Baniasadi (Department of Electrical and Computer engineering)
Supervisor

Dr. Nikitas Dimopoulos (Department of Electrical and Computer engineering)
Departmental Member

Abstract

Supervisory Committee
Dr. Amiarali Baniasadi (Department of Electrical and Computer engineering)
Supervisor
Dr. Nikitas Dimopoulos (Department of Electrical and Computer engineering)
Departmental Member

This thesis introduces an alternative Fault-Tolerant Power-Aware Hybrid Adder (or simply FARHAD) for high-performance processors. FARHAD, similar to earlier studies, relies on performing add operations twice to detect errors. Unlike previous studies, FARHAD uses an aggressive adder to produce the initial outcome and a low-power adder to generate the second outcome, referred to as the checker. FARHAD uses checkpoints, a feature already available to high-performance processors, to recover from errors. FARHAD achieves the high energy-efficiency of time-redundant solutions and the high performance of resource-redundant adders. We evaluate FARHAD from power and performance points of view using a subset of SPEC'2K benchmarks. Our evaluations show that FARHAD outperforms an alternative time-redundant solution by 20%. FARHAD reduces the power dissipation of an alternative resource-redundant adder by 40% while maintaining performance.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I owe my deepest gratitude to my supervisor Dr. Amirali Baniasadi for all the technical advice, motivation and support he provided me with during my studies in Canada.

I am also pleased to thank my supervisory committee member, Dr. Nikitas Dimopoulos for his valuable advice and technical feedback. In addition, I feel lucky that I could make nice friends at the University of Victoria who made it easy to deal with difficulties on this journey.

Dedication

To my parents and my sisters
for their love and support

Chapter 1

Introduction

Although most trends in VLSI design including technology scaling, lowering voltage level, increasing working frequency and using denser chips help achieve better performance they also result in higher vulnerability to transient and permanent *faults*. For instance, employing smaller transistors or lowering supply voltage results in smaller critical charge, known as Q_{crit} (an amount of charge needed for SRAM or DRAM to be flipped). Therefore, shrinking the size of a transistor or lowering the supply voltage makes the device more sensitive to high-energy and alpha particles leading to higher transient fault rate [1, 2].

Moreover, technology scaling increases the number of transistors per die, and consequently raises the probability of having faults. Similarly, having more on-chip transistors results in higher power dissipation and consequently more generated heat. If the increase in heat causes the temperature to last high for a long period of time, it makes the chip more prone to intermittent faults [18].

We usually consider physical defects including a broken wire or a broken transistor as a fault. Such defects can manifest themselves as errors. For example if an induced charge in a memory cell toggles a bit from 1 to 0, or vice versa, the occurred fault shows itself as an error. If this error is not masked or recovered, it would become visible to the consumer and result in failure [7]. Likewise, failures can show themselves in a higher level of abstraction, from a simple miscalculation to a devastating system crash.

Modular and temporal redundancies are the two main conventional approaches proposed to make designs immune against faults. The main problem of these techniques

is that they come with significant performance, area or power overhead [3]. While modular redundancy usually comes with power/area overhead, temporal redundancy suffers from performance loss. The goal of many recent studies has been to reduce these overheads [4-5].

In this work we investigate fault tolerant methods, particularly those focusing on arithmetic units including adders; study their pros and cons and then introduce our own approach.

1.1 Faults

This section enlists causes and impacts of different categories of faults depending on their duration. We also study faults in two different logics including combinational and sequential.

1.1.1 Fault Duration and Source

Faults occurring in semiconductor devices can be classified into three different categories: permanent or hard fault, intermittent fault, transient fault or soft errors.

- ***Permanent faults*** happen at the same place of a component every time used. A component can get faulty during the fabrication process due to incorrect metallization, contaminated silicon surface, etc [6]. Moreover hard faults can occur in a module after manufacturing due to many reasons including thermal cycling, electro migration and stress migration of interconnects. These faults are usually dependent on the material used in the component [6].
- ***Intermittent faults*** are those faults that occur repeatedly but not continuously. They occur as a result of some physical phenomena such as chip overheating. For example as the chip temperature fluctuates, some wire connections may disconnect temporarily [1] [7]. Therefore, sometimes an error may happen and

sometimes may not. Generally speaking, these faults manifest themselves at the same place but not each time the component is used.

- ***Transient faults*** or single event transients occur in components randomly and rarely. Unlike the permanent faults, they do not happen every time the component is being used. There are two main sources for this kind of fault: high energy particles which are produced when cosmic rays hit atmosphere and alpha particles which are emitted due to natural decay of radioactive isotopes [7]. As soon as each of these particles strikes a semiconductor chip, they dislodge some amount of charge which results in flipping a cell or altering the value a gate has produced in a circuit. In this work we focus on transient faults and introduce an adder which can tolerate soft errors.

1.1.2 Faults Impacts

Not only a single memory cell or gate can be affected by transient faults, but also two adjacent memory cells or gates value can be altered by them. With aggressive technology scaling, radiation-induced soft errors can lead to Multiple Event Transients (METs) in combinational logic and Multiple Bit Upsets (MBUs) in sequential logic [8-11].

As the distance between junctions in CMOS technology decreases, the chance for an energetic particle to affect two adjacent logic gates or memory cells increases [9]. An energetic particle hitting a combinational logic can affect two or multiple physically-adjacent logic gates leading to multiple transients, also called METs. If an energetic particle hits multiple flip-flops, it can invert multiple values, leading to MBUs.

1.1.3 Faults in Sequential and Combinational logic

Storage structures are one of the main parts of digital systems. Any faults in these structures may result in destructive system failure and system crash.

For many years, due to masking phenomenon including logical and electrical, transient errors were thought to occur less in combinational logic than memory cells [2].

In logical masking the altered signal has no effect on the output. For instance, if the first input of an OR gate becomes faulty, the fault does not propagate to the output if the second input is "1". In electrical masking, the altered signal is not powerful enough and becomes weaker due to gates electrical features to a point that it does not affect the output.

However, recent studies show that combinational logics including ALU are highly susceptible to transient errors [2]. Like a memory cell which may be flipped by a transient voltage caused by neutron or alpha particles, any combinational logic node may confront a transient fault. This may result in a soft error as the fault propagates through the logic gates and gets latched by a sequential logic somewhere in the circuit [12].

1.1.4 Fault Tolerance Metric

Quantitatively, there are different metrics including availability, reliability, mean time to failure, meantime between failures, failure in time and architectural vulnerability factor that can be used to evaluate the reliability of a design [7]. In this section we briefly explain each.

- **Availability** of a design at time *t* is the probability of working properly for the design at time *t* [7].

- **Reliability** of a design at time t shows the probability for that design to work with no error till time *t* [7]. For more details see section 4.5.2.
- **Mean time to failure** (MTTF) is the average time to have faults in a design. Although, designers try to have a higher MTTF, but since it is as an average number, it is not a reliable metric for comparing two different designs. That is because it does not give any information about the time of fault's occurrence. Therefore it is possible to have a design with a smaller MTFF but higher variance compared to another one with higher MTFF but smaller variance. Certainly, a design with higher variance faces fault sooner than the other one, which may not be acceptable for[7].
- **Failure in time** shows the number of faults that happen in one billion (10^9) hours. This is proportional to MTTF inversely [7].
- **Architectural vulnerability factor** or simply AVF is a recently proposed metric which evaluates the fault-tolerance of micro-architectural designs against transient faults [30].

  AVF classifies the system states into 2 groups including Architecturally Correct Execution (ACE) and un-ACE. ACE states are those units for which faults would result in wrong execution of instruction. For example, program counter (PC) is always an ACE state; any fault in PC results in an incorrect instruction execution. On the other hand, any faults in un-ACE units just may cause performance loss. For instance, any soft error in branch predictor results is misprediction and consequently performance loss. But it is not architecturally visible and will not lead to wrong execution of instructions.

Besides the above mentioned structures, there are also many parts that are ACE just for a fraction of times. The AVF of these units is calculated based on the number of un-ACE bits and the time these un-ACE bits reside in the system. In order to calculate the AVF of the whole system, the same method should be used.

1.2 Adders

The adder is one of the most commonly used units in digital circuits. Many processor operations including addition, subtraction and comparison rely on the adder to produce results. Moreover almost all processor types, i.e., high-performance, embedded and DSP processors, use adders in their organization.

Figure 1.1 shows the frequency of ADD operations in a subset of SPEC'2k applications as measured by SimpleScalar 3.0 toolset [13]. On average, 65% of the operations taking place in these applications use the adder unit. Accordingly, designing a reliable adder can impact the overall reliability considerably.

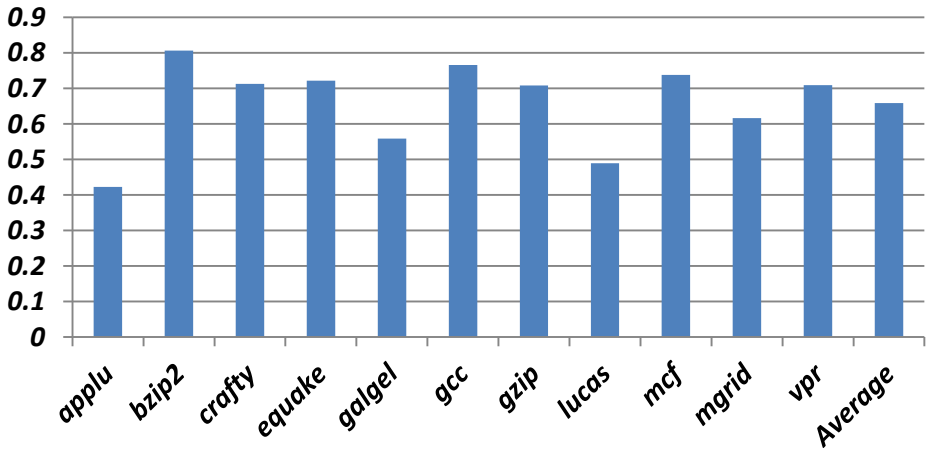

Figure 1.1: Frequency of add operations in a subset of SPEC'2K application

Previous studies have introduced many adder implementations including ripple carry adder (RCA), carry skip adder, carry select adder and etc. [29]. The ripple carry adder (RCA) is among the most straightforward implementations of an adder. RCA, while being simple, is very slow. The low performance of this implementation is the result of the long delay associated with propagating the carry signal through a number of full-adders and in a sequential manner. Figure 1.2 illustrates a 32-bit RCA implementation.

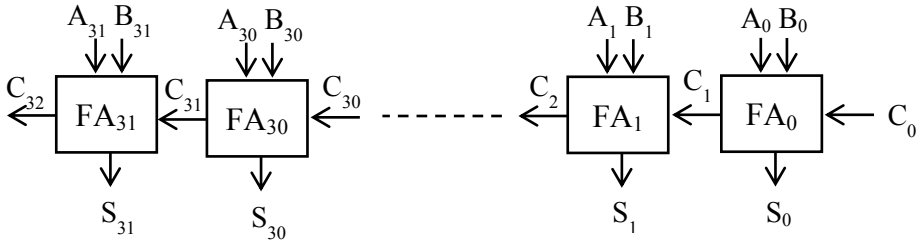

Figure 1.2: 32-bit RCA implementation.

As Figure 1.2 shows, every full adder (FA) output is dependent on its previous FA result. Therefore $S_{31\ and}\ C_{32}$ do not become ready till A_0, B_0 and C_0 propagate through all the FA blocks and represent themselves as an input carry (i.e., C_{31}) to the last FA block (i.e., FA_{31}). This causes the RCA low efficiency.

Carry Look-ahead (CLA) is a classic solution for this problem. CLA adder relies on extra logic to produce carry signals fast and without the timing overhead associated with the ripple carry. This delay reduction, however, comes with extra hardware overhead, which in turn results in considerable power dissipation. In Figure 1.3 we present the CLA adder design. As depicted, CLA adder is composed of 3 main blocks including *P & G Generator*, *Carry Generator* and *Sum Generator* blocks. These blocks are based on equations below:

$P_i = A_i \oplus B_i, \qquad G_i = A_i B_i \quad (1)$

$C_{i+1} = G_i + P_i C_i \quad (2)$

$S_i = P_i \oplus C_i \quad (3)$

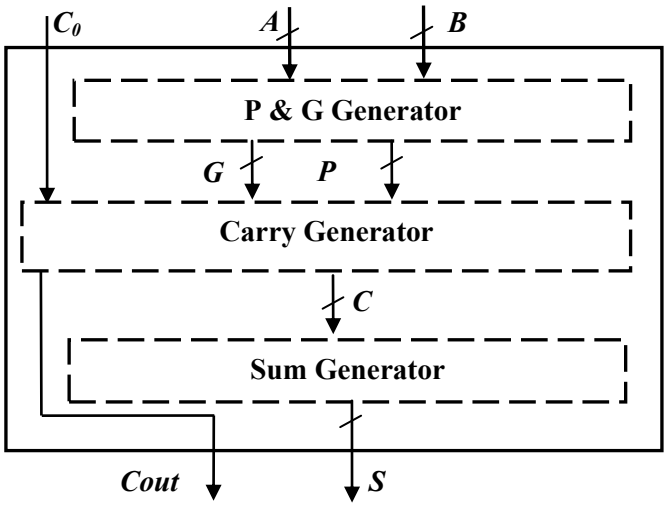

Figure 1.3: CLA adder.

The blocks work in serial, but each block generates all its output at the same time. As equation 1 reports, in theory all Ps and Gs can be generated in one level of logic. On the other hand equation 2 can be expanded as follows:

$C_2 = G_1 + P_1 C_1$

$C_3 = G_2 + P_2 C_2 = \ G_2 + P_2 G_1 + P_2 P_1 C_1$

$C_4 = G_3 + P_3 C_3 = \ G_3 + P_3 G_2 + P_3 P_2 G_1 + \ P_3 P_2 P_1 C_1$

$\vdots$

$C_{32} = G_{31} + P_{31} C_{31} = \ G_{31} + P_{31} G_{30} + P_{31} P_{30} G_{29} + \ P_{31} P_{30} P_{29} C_{28} + \cdots$

Therefore if all Ps and Gs become available simultaneously, carries can be generated in two logic levels. Thus the final output can be produced in four logic levels theoretically.

Note that due to physical implementation constraints, CLA adder cannot be implemented in four levels of logic.

Beside CLA adder there are also many other fast adders including Ling adder [25], Kogge-Stone adder[26], Brent-Kung [27] adder and Han-Carlson adder [28] from which most rely on CLA adder for implementations.

1.3 Error Detection Methods

Conventionally, there are two main approaches in building fault-tolerant designs. The first approach relies on modular redundancy [3]. It uses three equal modules and a majority voter to recover from possible errors. This approach comes with significant power and area overhead. The second approach relies on temporal redundancy, i.e., it takes three executions of a task to prepare fault-free results. Although this approach has less area and hardware complexity, it suffers from performance overhead.

Furthermore, previous studies on fault-tolerant adders have introduced many designs including TMR (i.e. triple modular redundancy) [3], QTRA [5], TEPS [14], and lazy error detection [15]. While TMR uses three full-size similar adders connected to a majority voter to correct any possible errors, QTRA uses three one-fourth size adders to do so but in four consecutive iterations. Lazy error detection technique detects an error using the time gap between execution and commit time and by employing an area-optimized checker. TEPS takes advantage of the unused most significant bits of the adder for re-computation and thus error detection and correction. More details of each method can be found in Chapter 2.

Although each of the mentioned approaches has their own drawbacks, some of these adders achieve reliability at the expense of power and area overhead (e.g., TMR). Such

solutions employ high performance and power hungry units to reproduce execution units' outputs. This aggressive approach is inefficient from the energy point of view as errors occur rarely, making calculating the second output often unnecessary.

One way to reduce the associated overhead is to use a simple, slower and hence more power efficient unit to reproduce the execution unit output. This approach can provide the same level of reliability while imposing much less power overhead.

1.4 Contributions

In this thesis, we address both power and performance issues by introducing a **Fa**ult-tole**r**ant Power-Aware **H**ybrid **Ad**der (FARHAD). FARHAD uses two adders to produce the add operation result twice. Both outcomes are compared and a mismatch detects an error. This is achieved by using a power-efficient adder (i.e., ripple carry adder) alongside a conventional fast adder (i.e., carry look-ahead adder). The fast adder calculates the outcome which will be immediately used by the processor. The slow adder calculates the outcome and has it compared to the outcome produced earlier. The processor ignores matching outcomes without interrupting program flow. A mismatch triggers a recovery process.

We study the impact of infrequent consecutive additions on our system and take appropriate measures to address them. We evaluate FARHAD from different aspects and show that FARHAD comes with higher reliability (quantitatively and qualitatively) but dissipates 59% less power compared to TMR. We also show that FARHAD outperforms QTRA by 20%.

1.5 Thesis Organization

The rest of the thesis is organized as follows. In Chapter 2 we review the previous works. In Chapter 3 we propose our technique, FARHAD. In Chapter 4 we present the results. Finally, in Chapter 5 we offer our conclusions.

Chapter 2

Related Works

In this chapter we discuss previous fault tolerant approaches applied to digital blocks and in particular adders. Employing physical and temporal redundancies are two common approaches used in error detection and correction. Both approaches are applied to protect hardware against transient and permanent errors.

Using multiple copies of the target module is a simple solution for both detecting and correcting errors [7]. DMR (dual modular redundancy) and TMR (triple modular redundancy) are the most well-known ones. Figure 2.1 illustrates the architecture of DMR.

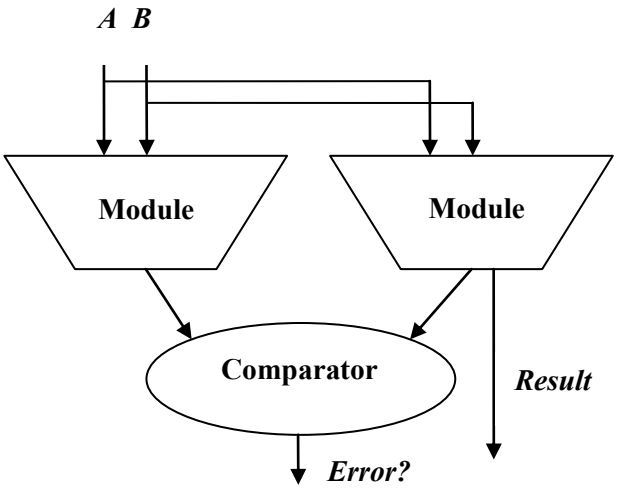

Figure 2.1: Dual modular redundancy.

As Figure 2.1 shows, using an extra identical module alongside with a comparator makes the design capable of detecting errors. Both modules generate results

independently. Then the comparator checks whether the produced results are the same or not. Any mismatch shows that the operation result is erroneous.

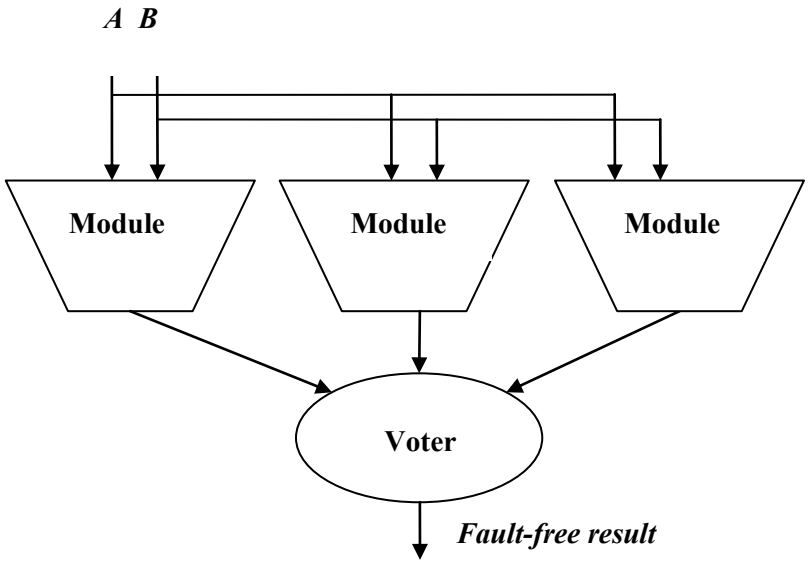

Figure 2.2: Triple modular redundancy.

Adding another replica and replacing the comparator with a majority voter converts DMR to TMR which is capable of correcting errors. This is shown in Figure 2.2. Moreover, Figure 2.3 illustrates a 1-bit majority voter that is used in TMR.

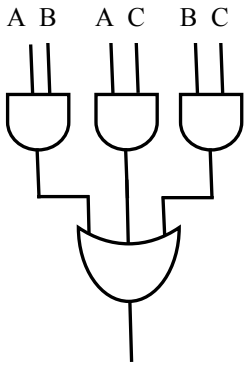

Figure 2.3: 1-bit majority voter.

As Figure 2.3 reports, a 1-bit majority voter receives three binary inputs from which at least two are the same. Then it decides the value of similar inputs and bypasses it to the output. In other words, as it is assumed to have only 1 fault at a time, a 1-bit majority voter picks an amount (0 or 1) which has been produced at least two times of three by the three replicas. Although DMR and TMR are implemented easily, they require extra logic and hence come with considerable power and area overhead.

One way to reduce this overhead is to reuse the module (temporal redundancy) for error detection and correction. In temporal redundancy the computation is done using the same module two and three times to detect and correct the errors respectively. In this case a register is needed to store each computation result for later comparison.

One of the problems with using temporal redundancy is that it can only detect or correct transient faults but not permanent faults. As explained in Section 1.1.1, permanent faults occur at the same place of a component each time the component is used. Therefore, if the occurring fault is permanent, it cannot be detected and corrected by using temporal redundancy since the result would be the same for all three iterations. But, in case of any transient faults, the re-computation results would be different and making it possible to detect the fault. Many approaches were introduced to solve this problem. RE-computing with SWapped Operands (RESWO) [16] and RE-computing with Rotated Operands (RERO) [17] are two examples.

In RESWO, first the input operands are applied to the module and the result is saved. Then, the operands upper and lower haves are swapped, the computation is done again and the results are saved. Finally the results are swapped back and compared with the primary result to detect any probable errors. The same technique is used in RERO. The

difference in RERO is that the operands are not swapped; they are rotated by a specific number of bits instead.

Although temporal redundancy methods including RESWO and RERO resolve the power and area overhead of physical redundancy and also capable of detecting permanent faults, due to serial re-executions they come with considerable performance overhead.

RE-computing with Duplication With Comparison (REDWC) [18] was proposed to improve performance in RESWO. Instead of using the whole module twice, it uses two half sized replicas in two consecutive iterations. In the first iteration the first half of the operand is fed to the replicas. In the second iteration the replicas do the computations for the second half. In both iterations the results are compared. As REDWC approach does not need any swapping or shifting operation, its circuit comes with less delay and less complexity compared to RESWO and RERO.

By adding another replica and dividing the operands to three portions, HPTR (Hardware Partition in Time Redundancy) extends REDWC's error detection capability to error correction ability [19]. But as operands are divided by three, an extra iteration would be needed which will result in performance degradation. Both HPTR and REDWC apply modular and temporal redundancy simultaneously in their methods.

HPTR suffers from two problems. First, usually operand sizes are not dividable by three, so the module should be padded, which results in more resources, area and more power dissipation. Second, not all resources including the multiplexer control signals are utilized in all iterations.

QTRA was introduced to overcome the utilization and resource overhead problem of HPTR [5]. Similar to HPTR, QTRA takes advantage of both modular and temporal

redundancy. In QTRA operands are divided to four parts. Therefore four iterations are needed for the operation to complete. In each of the iterations, three replicas do the addition and then the correct outcome is selected by the majority voter. While each iteration can be done in a shorter time, the extra number of iterations results in performance loss. Figure 2.4 shows the QTRA architecture.

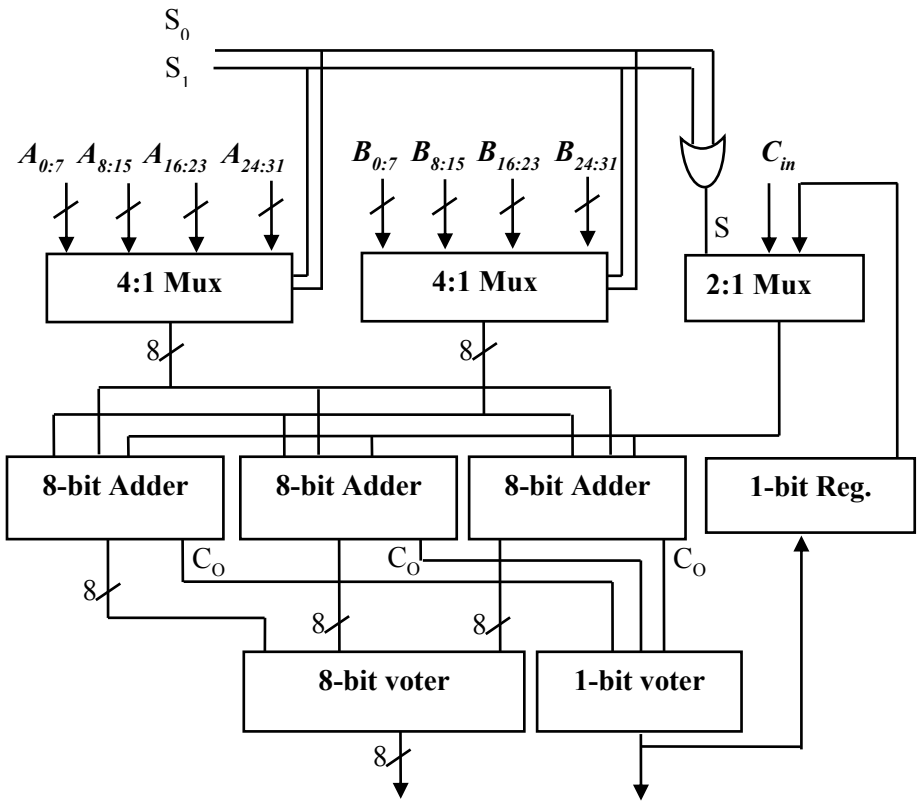

Figure 2.4: 32-bit QTRA.

As Figure 2.4 depicts, in addition to adders and majority voters, 3 multiplexer and a 1-bit memory cell is needed. Depending on the iteration turn, the two 4:1 multiplexers let the appropriate operand portions to be fed to the adders. 1-bit memory cell is employed to save the output carry of each iteration result. The saved output carry is used for the next iteration as an input carry. Furthermore, the 2:1 multiplexer is in charge of choosing the

proper input carry for the adders in each of the iterations. For the first iteration C_{in} should be chosen while for the rest of the iterations the memory cell content should be passed to the adders.

An alternative approach to detecting errors in arithmetic operations and particularly multiplications is modulo (residue) checking scheme [7, 20]. This method utilizes smaller arithmetic units as the replica to detect errors. Figure 2.5 depicts how module checking works.

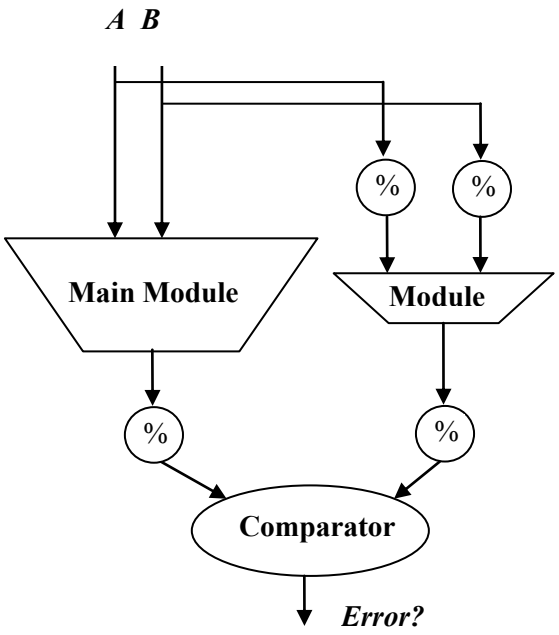

Figure 2.5: Modulo checking.

As Figure 2.5 shows, in modulo checking, first the modulus operation is applied to both operands. Since the results are smaller than the real operands, smaller operator is utilized to do the operation to produce the checker. Then the modulus operation is performed on both the actual result and the checker. Finally, similar to DMR, these results are compared together to find faults.

On the negative side and depending on the modulus operands size, there is a chance that both checker results and the actual results match even in the presence of an erroneous primary result [7]. One way to address this is to increase the modulus operands size, which results in more hardware and power dissipation.

TEPS (Transient Error Protection Utilizing Sub-word Parallelism) [14] and Lizard [4] are two recent techniques introduced to protect arithmetic units against transient and permanent faults, respectively. They both take advantage of narrow-width operands to protect the ALU from transient and permanent errors. Narrow-width operands are operands that contain a large number of consecutive zeros or ones from their most significant bit. If both operands are narrow-width, a big portion of resources would not be utilized. For example for an 8-bit addition in a 32-bit ripple-carry adder, twenty four of the full adders do not participate in the operation. Therefore, the free resources can be used for re-computation and thus error detection and correction. According to TEPS and Lizard, a big portion of additions are narrow-width. To provide better insight, in Figure 2.6 we report the rate of narrow-width operations for the SPEC'2K applications studied here. This Figure shows the number of data bits necessary to use in integer add operations.

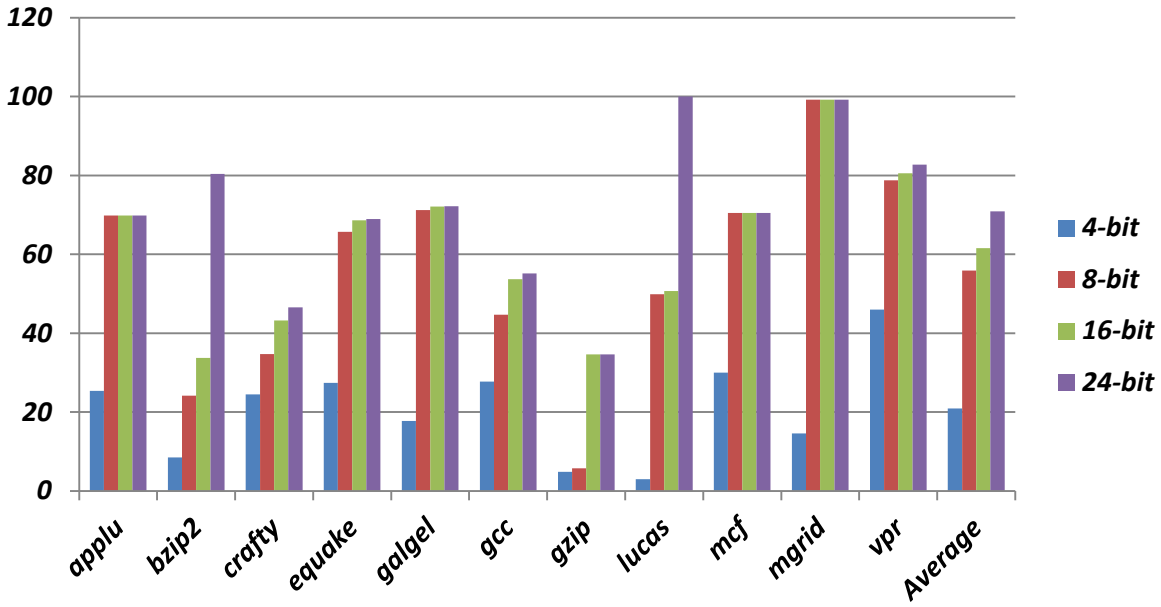

Figure 2.6: Bars from left to right report the frequency of add operations requiring four, eight, 16 and 24 bits for 32-bit adder.

As reported in Figure 2.6 for 32-bit adders, on average, 20.3%, 55.8%, 61.5%, and 70.9% of add operations require less than eight, 10, 16 and 24 bits, respectively.

TEPS relies on the observation reported in Figure 2.6, and takes advantage of the free resources for re-computation. For example, as shown in Figure 2.6, on average for 61.5% of the additions, TEPS can use the 16 upper bits for re-execution while doing the main execution as well. While TEPS comes with acceptable area and power dissipation overhead, it increases the critical path delay of ALU by 17%. Moreover, it comes with 12% performance penalty since there is still a portion of additions which are not narrow-width. ALU should be allocated to these additions for another time to produce the checker.

Lizard uses an approach similar to TEPS. In lizard, the 32-bit adder is divided to four 8-bit adders. As a big portion of additions require less than eight bits, three sub-adders

can adequately correct any occurring fault. Lizard comes with time overhead, as it requires two cycles for non-narrow-width operations. One of the advantages of Lizard is that upon diagnosing permanent faults, it isolates the faulty sub-adder and continues working.

Furthermore, there are two other studies which take advantage of processor speculation mechanism to recover from faults [15][20]. The approaches rely on *lazy error detection*. In these methods, the result of the add operation is tested by a *checker*. Error detection is done between the execution and commit time.

Chapter 3

FARHAD

In this chapter we introduce a low-power alternative for fault-tolerant adders used in high-performance processor. We utilize a power-aware adder alongside the main adder and compare their outcomes, to detect faults. In order to recover from probable faults we take advantage of check pointing, an already available feature of modern processors.

3.1 Introduction

Almost all fault-tolerant approaches proposed for arithmetic units including adders take advantage of modular or temporal redundancy in their scheme. This comes with an unavoidable re-execution of the operation which consequently results in more energy consumption. Therefore, reducing power is one of the serious challenges for architects and IC designers while designing fault-tolerant units.

As explained in Section 1.2, the adder is one of the most commonly used components in ALU. It accounts for a significant portion of ALU power dissipation. Therefore, in order to achieve high performance most high performance processors comprise fast adder in their design. But due to high complexity and large number of gates, usually fast adders come with more power dissipation than slow ones. Figure 3.1 shows the power dissipation of two already discussed adders, i.e., CLA adder and an RCA per addition, for a subset of SPEC'2K benchmark when used independently.

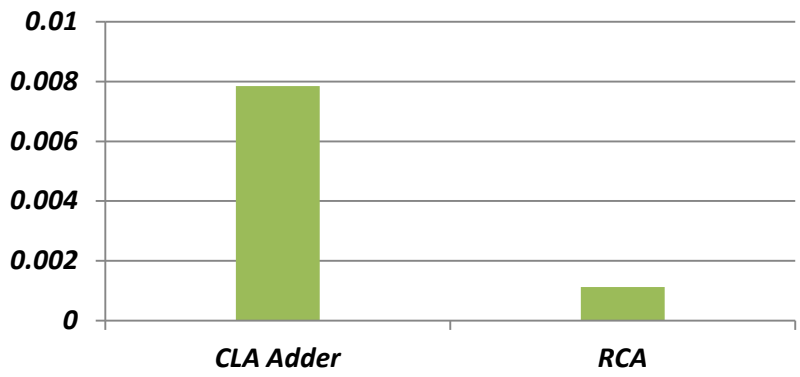

Figure 3.1: Power dissipation of CLA adder vs. Ripple Carry Adder (mW).

As Figure 3.1 shows the power dissipation of the CLA adder is seven times higher than RCA. Based on this observation and considering the point that faults happen rarely, it is not worthwhile to use the same fast adder for re-execution of the operation to find faults. Motivated by this observation, we use a simple RCA in parallel to a CLA adder to detect possible errors. We show that by using checkpointing [22], which is already available to high-performance processors, it is possible to recover from errors without compromising performance.

Our two underlying adders come with different latencies. CLA adder is faster than RCA and thus has to wait till RCA's result becomes ready for comparison. This does not harm performance under FARHAD since FARHAD takes advantage of the gap between pipeline stages and overlaps the RCA delay with this gap (i.e., dispatch to commit time). In other words, while RCA is busy with calculations, CLA adder's outcome is sent to units waiting for the addition result.

Note that there are other timing complexities including occasions when two add instructions are dispatched consecutively. In this work we address these complexities and

explain how FARHAD deals with these issues. We show that the overall associated performance cost of these timing issues is negligible.

3.2 Overview

Figure 3.2 shows FARHAD's architecture. FARHAD uses CLA as a fast but power hungry adder. In order to detect possible errors FARHAD uses an RCA to execute the operation twice. Performing an operation twice and comparing the two independent outcomes enhances reliability significantly.

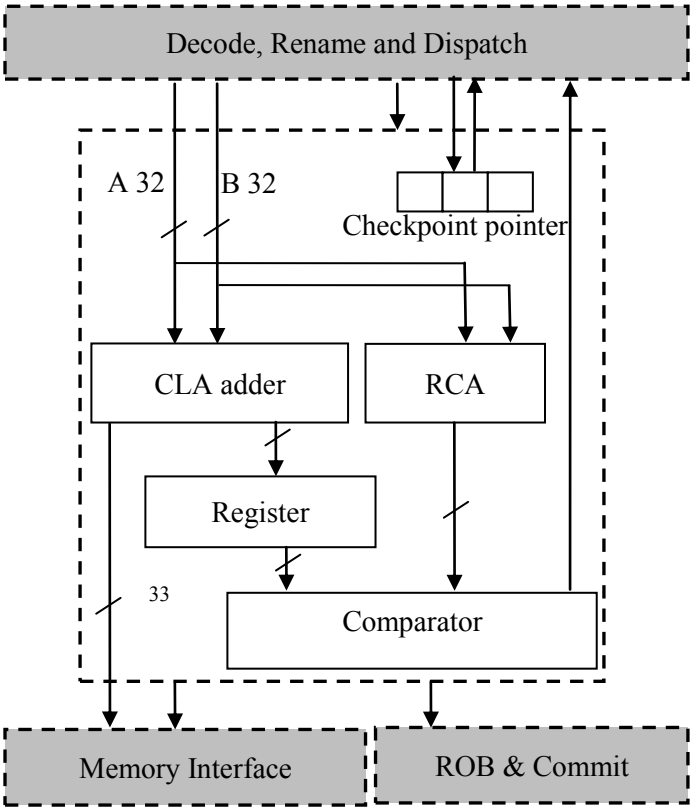

Figure 3.2: FARHAD architecture.

We refer to the two outcomes as the *actual outcome* (produced by CLA) and the *checker* (produced by RCA). While the actual outcome has to be produced fast to

maintain performance, the checker can be decided more slowly (and hence more power efficiently).

In order to maintain reliability, the actual outcome has to be compared to the checker prior to the instruction's retirement. Upon a mismatch (which occurs rarely), the operation must be redone and the processor must recover from the mistake. Current modern processors come with effective recovery and checkpointing mechanisms which we employ at a low extra expense (i.e., area cost of a 3-bit register). The slower checker has to be produced before the instruction commits to avoid performance penalty. In our system, the checker is produced early enough and therefore there is no performance penalty associated with accurate outcomes (see Figure 3.2).

FARHAD comes with the overhead of four extra components, including an *RCA* adder, a comparator, a 3-bit checkpoint pointer and a 32-bit register.

It should be noted that we use RCA and carry look-ahead (CLA) adder as examples of low-power but slow and fast but power hungry adders. The idea behind FARHAD, however, can be applied to any pair of adders with characteristics similar to ripple carry and CLA adders.

3.3 Microarchitecture

3.3.1 RCA, CLA Adder and Comparator

As explained earlier, FARHAD relies on two different adders with different latencies to generate the actual outcome and the checker. Then they are compared using a comparator. The comparator result decides whether the associated instruction can retire. Figure 3.3 represents the gate level implementation of the comparator.

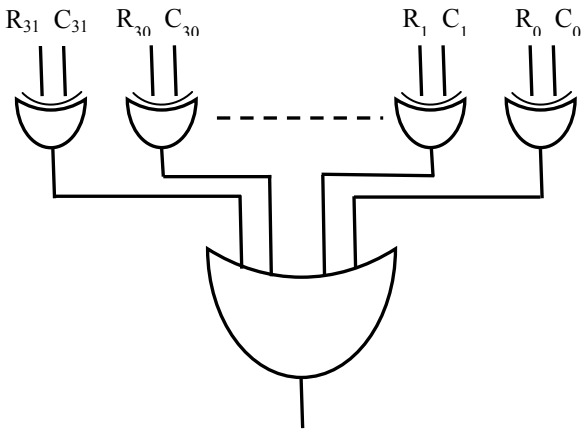

Figure 3.3: Gate level implementation of the comparator.

As Figure 3.2 shows, CLA adder's result is bypassed to the next stage to assure that CPU is not stalled while error detection is in process. In other words, in FARHAD fault detection is done in the background using a copy of the outcome while the processor is performing its regular tasks. More details are explained in Section 3.3.4.

3.3.2 Checkpoint Pointer

Modern processors use check points to recover from branch mispredictions [22]. Checkpoints store the information required to cleanup and restart operation after a branch misprediction

Beside control-flow speculation, FARHAD also relies on this feature as a part of its recovery mechanism. Whenever a fault is detected by FARHAD, it re-executes all the instructions from the most recent branch, using data provided by checkpoints. As checkpointing is already available to the processor, FARHAD only needs a way to access it. To this end, we use a 3-bit memory cell called *checkpoint pointer*. Checkpoint pointer is in charge of saving the checkpoint for the add instruction being checked. This

checkpoint is the same checkpoint associated with the most recent branch instruction. Our system supports eight checkpoints requiring a 3-bit checkpoint pointer.

A checkpoint which has been assigned to an addition should not be released till the CLA adder's outcome has been validated. Therefore, FARHAD does not allow a checkpoint to be released when its corresponding branch is resolved.

When an error occurs, FARHAD takes advantage of this feature, flushing the pipeline back to the most recent branch instruction. This comes with the cost of re-executing the accurately executed instructions between the branch and the add operation. The cost is negligible as such errors occur rarely.

3.3.3 Register

As RCA is slower than the CLA adder, the CLA adder's result is saved so it can be compared with the checker. A 32-bit register is responsible to latch the CLA adder's result (more details in Section 3.4).

3.4 Timing

In this sub-section we investigate and address the timing issues FARHAD faces. In all the studied conditions in this sub-section we assume 90nm technology.

3.4.1 RCA and CLA Adder Different Latencies

Since CLA adder and RCA come with different delays, the actual outcome and the checker are produced not at the same time. Therefore, to make comparison possible, FARHAD requires an extra register to save the CLA adder's result.

As explained in Section 1.2, CLA adder can be implemented in four logic levels. In other words, in theory, CLA adder's critical path is composed of four gates. On the other hand RCA consists of serially connected full adders which result in a long chain of gates.

Therefore, there should be a big difference between RCA and CLA adder's latencies. But, due to physical implementation constrains including fan-in and fan-out, the 32-bit CLA adder can never be implemented in four levels of logics. For instance, the output carry needs an OR gate with 32-bit inputs to be implemented. Assuming that each gate can have a maximum of four inputs, the OR gate is implemented in three level gates. Therefore the total delay of the CLA adder would be more than what we expect. Assuming 90nm technology, our observation shows that there is only one clock cycle difference between a 32-bit RCA and a 32-bit CLA adder. Figure 3.4 reports the relative delay of 32-bit and 64-bit RCA compared to 32-bit and 64 bit CLA adders in different technologies respectively.

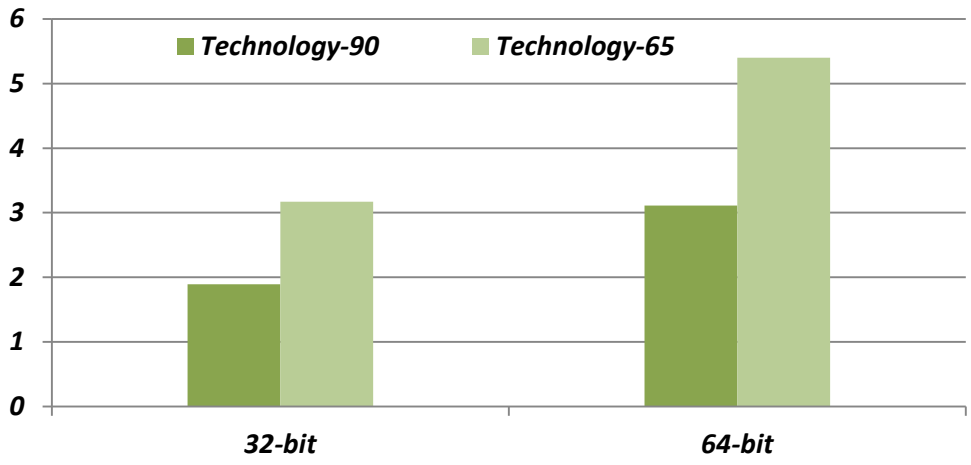

Figure 3.4: Relative delay for 32-bit and 64-bit RCA compared to 32-bit and 64-bit CLA adder.

As presented in Figure 3.4, RCA latency is 1.9 and 3.17 times of CLA adder delay in 90nm, and 3.11 and 5.4 in 65nm CMOS process technology for 32-bit and 64-bit addition respectively.

As FARHAD uses a bypass path, the CPU does not need to wait for the fault detection system to verify the correctness of the operation. While the fault detection system is working, a copy of the CLA adder's result is sent to the next pipeline stage. Accurately produced results, therefore face no additional penalty.

3.4.2 Issue to Commit Time Interval

It is also essential that the fault detection system produces the checker early enough to compare against the actual outcome before the instruction leaves the pipeline. In a typical processor, as well as the one used in this work, there is a *write-back* stage between issue and commit stages, taking at least two cycles. This interval gives enough time to the RCA to prepare its result.

Moreover, instructions reaching the commit stage sometimes have to wait for additional cycles so earlier instructions can commit first. As a result, the checker is produced in time to avoid any additional stalls. Figure 3.5 shows the timing diagram of FARHAD.

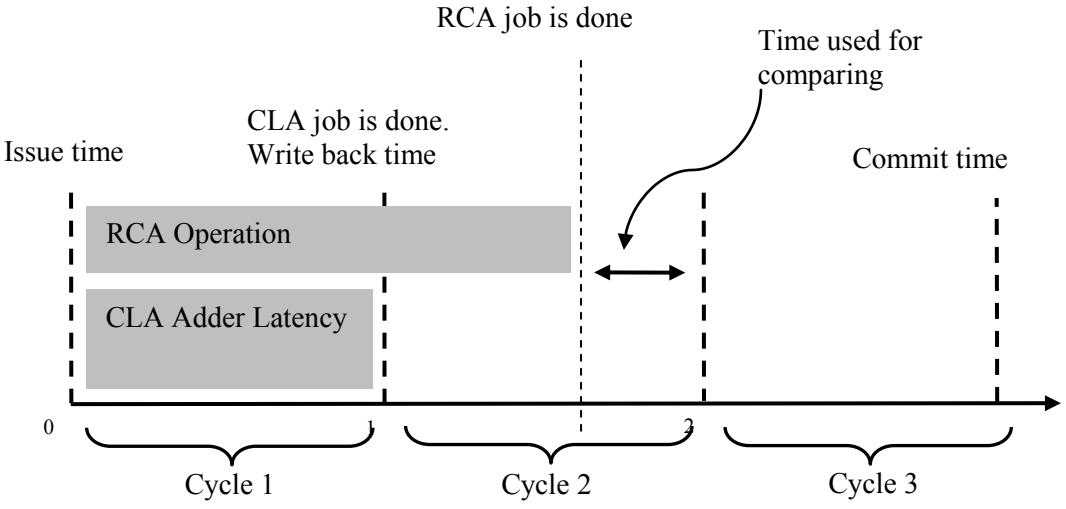

Figure 3.5: FARHAD's timing diagram.

As Figure 3.5 illustrates, based on latencies reported in Figure 3.4, while CLA adder operation is done in one cycle, the RCA needs an additional cycle. As RCA delay is less than two cycles, sufficient time remains for comparing the results. As a result the interval between the issue to commit time is adequate for FARHAD to decide whether the addition is erroneous or not.

3.4.3 Consecutive Additions

One of the challenges FARHAD faces is dealing with consecutive additions (i.e., two additions in consecutive cycles). This happens due to different delays of RCA and CLA adder. Since CLA adder needs only one cycle to perform, it can handle any number of additions in consecutive cycles. But, as RCA needs two cycles to complete its job, any successive additions should wait for the adder to finish its operation. The RCA cannot execute a new addition when busy performing the previous addition.

To address this issue, FARHAD stalls dispatching instructions to ALU upon seeing an immediate second addition. FARHAD keeps the second addition in the ready list and issues it in the next cycle. Our study shows that consecutive additions happen infrequently. According to our observations, just four of the 11 studied benchmarks in this work have consecutive additions [i.e., *bzip2*, *crafty*, *gcc* and *vpr*]. The performance penalty associated with the required additional stalls is less than 0.01%.

Moreover, we propose a solution for applications in which there is a big number of consecutive additions. In these cases, stalling dispatching of the instructions would result in a significant performance loss. A reasonable solution to this problem is employing a buffer at FARHAD's RCA input to store the operands while RCA is busy. Moreover, since the comparator needs both RCA and CLA adder's results, the register at CLA

adder's output should be replaced with a buffer to save the CLA adders' results for future comparison; the comparison of the current CLA adder's result at the output buffer and the RCA's result which has not been produces yet. The buffer size depends on the number of additions which come successively in consecutive cycles. Since the RCA needs 2 times of CLA adder clock cycles to perform the addition, after *n* clock cycles with *n* consecutive add operations half of the additions (n/2) have not been allocated to the RCA, and thus the operands should be waiting in the buffer. Therefore, the size of the buffer is half of the number of additions that have arrived successively in consecutive cycles.

Having many consecutive additions in successive cycles and using a buffer at RCA input and CLA output comes with two drawbacks. First, since the RCA's results are not prepared on time, many CLA adder's outcomes have to wait to be validated. This causes many instructions including those which have dependency on the additions to commit with delay which can result in performance penalty. Second, since buffer utilizes more resources than a single register, adding it at the RCA input and replacing the CLA adder's register with a buffer comes with more area and power overhead.

3.4.4 Timing and the Number of Checkpoint Pointer

In this sub-section we investigate how many checkpoint pointers are needed in FARHAD. Since we consider only one checkpoint pointer in FARHAD, the apparent problem appears when a new addition shows up while the old addition has not been validated yet; thus the checkpoint pointer is in use and cannot be assigned to the new addition. To address this problem we consider two scenarios for the new addition. Either

a branch instruction happens first and then the new addition occurs, or the new addition occurs between the old addition and a new branch instruction.

In the first situation, when the new addition occurs after the new branch instruction, since there is at least one instruction between the old and the new addition (i.e., the new branch instruction itself), certainly according to Figure 3.5 the correctness of the first addition has been decided already. Therefore, the checkpoint pointer is free and can be used by the second addition to point to the new branch instruction. In this case, the associated checkpoint to the second branch instruction can be assigned to the second addition if only the result of the first addition had been fault-free. Otherwise, all the instructions including the second branch and the second addition will be flushed from the pipeline.

In the second situation, the second addition occurs before a new branch instruction (i.e., the new addition has occurred between the old addition and a new branch instruction). In this case, if the second addition arises immediately after the old addition it can use the same checkpoint as the previous adder is using. But if the second addition happens after one, two or more instructions, there would be enough time for the checkpoint pointer assigned to the first addition to be released. Therefore the second addition can use it.

3.5 FARHAD vs. Lazy Error Detection

Our work is different from lazy error detection in several ways. First, FARHAD and lazy error detection employ different logic circuits. While lazy error detection utilizes a checker cell, FARHAD uses a simple full adder. Figure 3.6 shows the checker used in lazy error detection method.

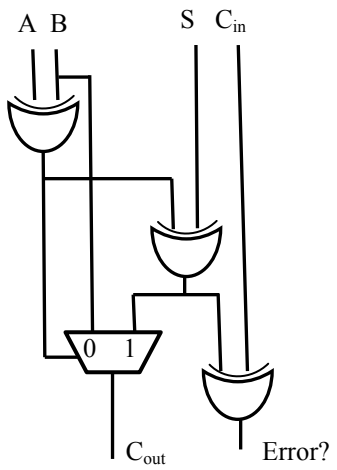

Figure 3.6: 1-bit lazy error detection checker.

We simulated the overhead associated with both solutions. Our measurements show that FARHAD comes with 12.5% less power overhead and 11% less area overhead compared to lazy detection.

Furthermore, in this work we provide deeper and more accurate analysis of performance and timing issues. We evaluate FARHAD under a more advanced technology and investigate scenarios that may slow down CPU's performance including consecutive additions. We also compare FARHAD to alternative fault-tolerant solutions and report power, performance, delay, area overhead and reliability. Moreover, and in addition to the quantitative results, we offer qualitative analysis of FARHAD's vulnerability. For qualitative analysis we consider various conditions under which SEU, SET, MBU and MET could occur. Unlike lazy error detection, we also suggest an error recovery solution and take the associated overhead into account.

In addition to the differences mentioned above, FARHAD can also be potentially used to recover from permanent adder errors. This is not available to lazy detection.

Chapter 4

Results

In this chapter we evaluate FARHAD from power, performance, area and vulnerability points of view. We compare FARHAD with two conventional methods i.e., TMR and QTRA. To provide better understanding, we also compare to an alternative processor, which uses two CLA adders in parallel referred to as *Dual-CLA*. Dual-CLA's architecture is the same as FARHAD's. The only difference is that it uses a CLA adder instead of RCA to detect faults.

4.1 Power Reduction per Addition

In Figure 4.1 we report the power reduction for FARHAD relative to other studied methods. As reported, on average, FARHAD comes with 41% and 60% power reduction compared to Dual-CLA and TMR respectively. Moreover it dissipates 44% more power relative to QTRA.

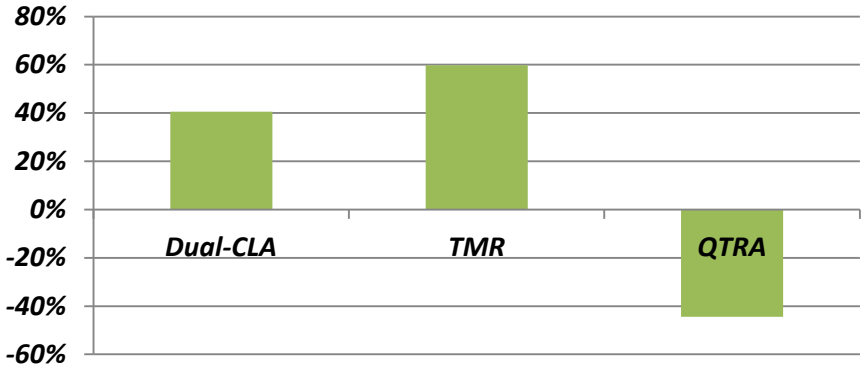

Figure 4.1: Power reduction of FARHAD relative to Dual-CLA, TMR and QTRA.

Although QTRA employs some extra logic including three multiplexers, 1-bit register and an OR gate, it uses three 8-bit CLA adders which are much smaller compared to

FARHAD's 32-bit CLA adders. Therefore, QTRA uses fewer resources in comparison to FARHAD and hence dissipates less power. This, however, is achieved at the expense of 20% performance loss compared to FARHAD. This is due to the fact that QTRA needs four cycles to execute each addition while FARHAD needs only one cycle.

Dual-CLA employs more resources compared to FARHAD. While FARHAD uses an RCA to generate the checker, Dual-CLA employs a CLA adder to do so. As already reported in Figure 3.4, RCA dissipates much less power compared to CLA adder. As a result, Dual-CLA comes with more power dissipation. Similarly, because TMR utilizes more resources (i.e., an extra CLA adder compared to Dual-CLA) it has the highest power dissipation among all methods.

4.2 Latency

In Figure 4.2 we report the relative latency of the adders studied in this work. Latency is important as it can impact CPU's clock frequency. Since almost all conventional fault-tolerant methods add extra resources to ALU's critical path, they can increase ALU latency if the additional delay requires an extra cycle. For instance, both TMR and QTRA impose a majority voter at the adder's output. Or QTRA uses multiplexers at its input.

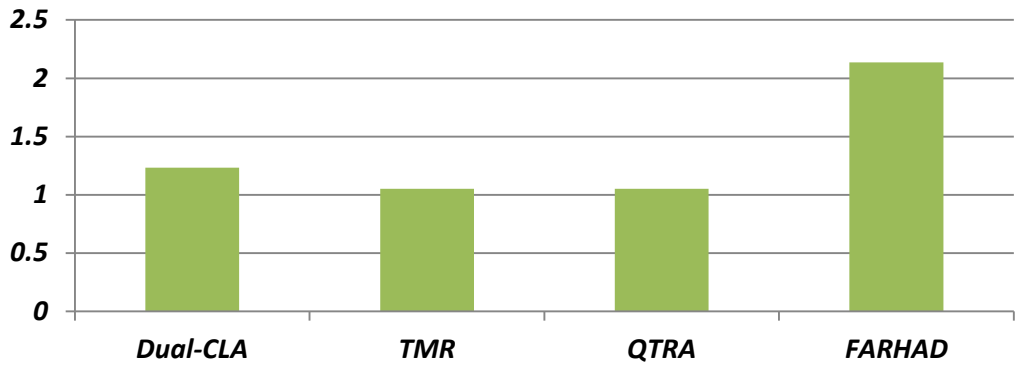

Figure 4.2: Relative Delay of FARHAD, TMR and Dual-CLA and QTRA compared to CLA adder.

As Figure 4.2 reports, while both TMR and QTRA have the minimum latency, FARHAD is the slowest adder. According to Figure 2.2, TMR uses three parallel CLA adders which are connected to a majority voter serially. As majority voter delay is equal to two logic levels (i.e., an XOR gate and OR gate), TMR's latency is slightly more than a CLA adder. QTRA's critical path uses a small size (8-bit) adder but requires extra resources including multiplexer and majority voter making it as slow as TMR. Both TMR and QTRA timing overhead is about 5%.

FARHAD includes a carry propagation chain and a majority voter, and therefore has the longest delay among all the methods. But as Figure 3.2 depicts, the CLA adder's result is bypassed to the next stage and error detection is done in the background while the result is being used. Dual-CLA also uses the same technique and utilizes a bypass path to the next stage too.

Therefore both FARHAD and Dual-CLA's effective latencies are close to a CLA adder and slightly more than TMR and QTRA latencies. The reason that the DUAL-CLA and FARHAD's effective latencies are more than TMR and QTRA's delay is that they employ a comparator while TMR and QTRA use a majority voter. As Figure 3.3

illustrates the results of XOR gates are inputs for a single OR gate. Due to fan-in constrains more than one level of logic is used in physical implementation and hence the comparator delay would be more than majority voter latency.

FARHAD and Dual-CLA timing overhead is about 5%. As a result, in our simulations we consider one clock cycle for FARHAD, Dual-CLA and TMR. Additionally as QTRA needs four iterations, and its latency is almost equal to a CLA adder, QTRA's latency is four clock cycles.

4.3 Performance

Figure 4.3 reports the performance loss for processors using FARHAD and QTRA relative to an unprotected processor. Since the delay overheard of TMR is negligible, we consider one clock cycle for its latency, similar to an unprotected ALU's adder. As a result its performance is equal to an unprotected CPU. Therefore we do not include it in Figure 4.3.

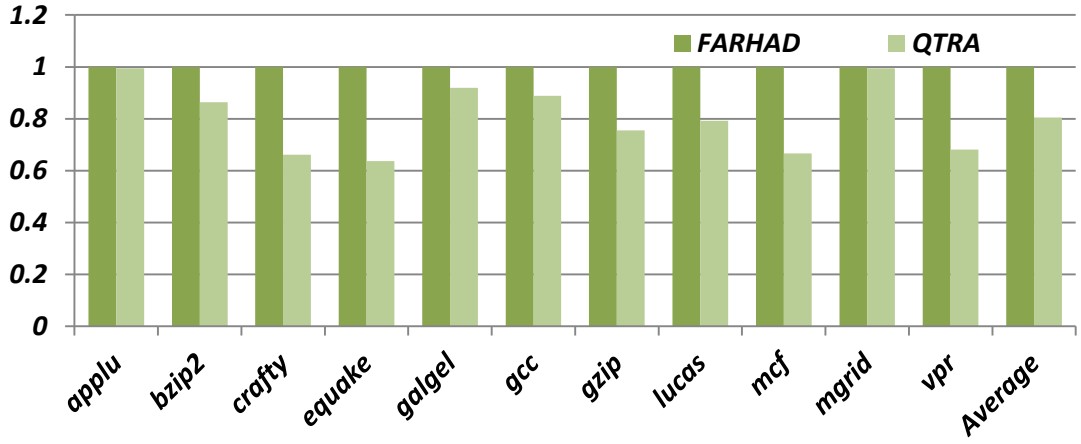

Figure 4.3: Relative performance of FARHAD and QTRA compared to an unprotected CPU.

As we explained in Section 3.4.3, there are only four benchmarks including *bzip2*, *crafty*, *gcc* and *vpr* for which consecutive additions occur. Our observations show that for these benchmarks, stalling the ALU can result in negligible performance loss (less than 0.1%). Therefore FARHAD has no negative side-effect on performance.

As Figure 4.3 reports, QTRA shows a significant average performance loss, i.e., 20%. That is because QTRA needs three more cycles to perform additions compared to FARHAD. This performance loss is consistent with our observation of add frequency in the studied applications. As Figure 1.1 shows a big portion of operations are additions. Therefore, by using more number of cycles for each addition QTRA comes with a large performance loss.

As Figure 4.3 reports, *applu, galgel and mgrid* have the least performance loss in QTRA. These benchmarks have the lowest number of additions. On the other hand, *equake*, *crafty* and *vpr* have the highest performance loss under QTRA. They have the most number of additions among their operations.

4.4 Area

Almost all fault-tolerant adders come with area overhead. Figure 4.4 reports the area of studied methods in this work compared to an unprotected CLA adder.

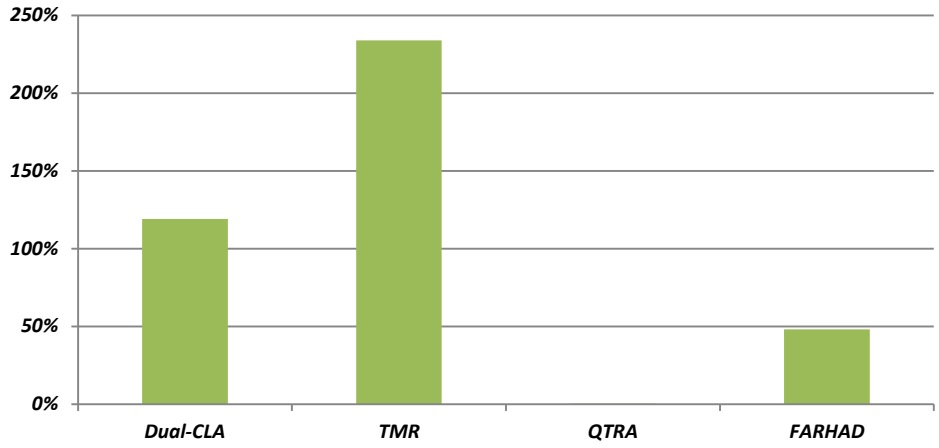

Figure 4.4: Area overhead of FARHAD, Dual-CLA and conventional methods.

As Figure 4.4 depicts, QTRA has the lowest area overhead. QTRA imposes extra resources including three multiplexers, two majority voters and a register; at the same time it utilizes 3 small adders (i.e. 8-bit) instead of a full size adder (i.e., 32-bit). Therefore its overall overhead is about zero.

FARHAD on average shows 48% area overhead, which is lower than TMR and Dual-CLA. That is because it uses RCA as the replica. RCA is less complex than CLA adder. Beside 32-bit comparator, Dual-CLA imposes one extra CLA. Hence its area overhead would be more than 100%. Dual-CLA's overhead is about 120%.

TMR has the worst area overhead (i.e., 233%) among all methods. That is because TMR employs three CLA adders and a 32-bit majority voter. Each of the extra adders comes with around 100% overhead.

4.5 Vulnerability Analysis

In this section we present vulnerability analysis from both qualitative and quantitative points of view.

4.5.1 Qualitative

In our qualitative analysis we study different scenarios in which SEU, SET, MBU, and MET occur. Since the probability of having two simultaneous particles hitting combinational or sequential logic is extremely low, in the rest of this analysis, we ignore the effect of having two particles hitting a single device at the same time. However, in our analysis, the effect of METs and MBUs caused by a single event (or an energetic particle) is taken into account.

Moreover, since any fault which occurs in wires propagate to a logic block; we study situations in which faults happen only in combinational or sequential logic.

- **TMR**

  SETs and METs may happen in any of TMR's components including the adders and the majority voter. The majority voter can correct errors occurring in the adders. An error happening in the majority voter, however, can propagate to the next logic block and may result in a system failure.

- **QTRA**

  As shown in Figure 4.5 QTRA has several components prone to SETs, METs and SEUs including multiplexers, adders, majority voter, 1-bit register and an OR gate.

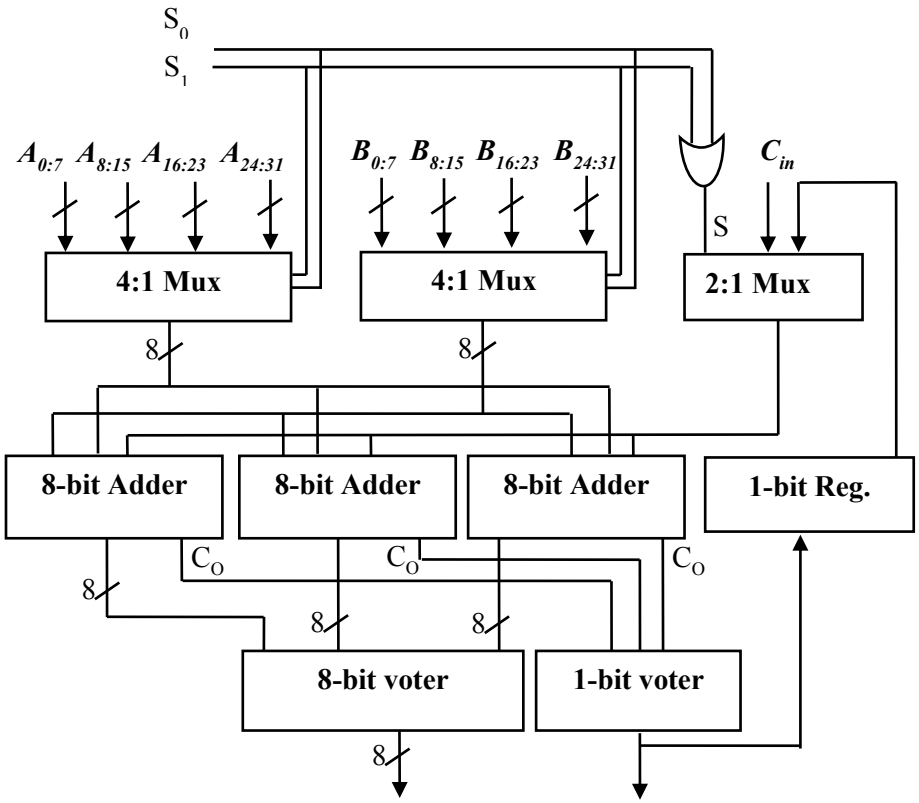

Figure 4.5: 32-bit QTRA.

As Figure 4.5 depicts, the multiplexers responsible to select the operand bit slice (4:1 multiplexers) use the same select wire. So any SET or MET in these wires and multiplexer logic can result in feeding the adders wrong but similar inputs. As adders receive the same inputs, they produce the same results. Hence, the majority voter detects no soft errors and propagates wrong data to the next stage. The same happens for the 2:1 multiplexer if any SET or MET occurs. If any soft error of any kinds happens in wires connected to the OR gate or the OR gate itself, it will result in selecting the wrong input in the multiplexer. This fault is not detectible by the majority voter. Thus, the error propagates to the next stage again. Moreover, since the register output is connected to the 2:1 multiplexer input, any SEU in the register

passes through the adders and the majority voter. Therefore this error also propagates to the next stage without being corrected.

The majority voter corrects any SET or MET initiated in the adders blocking the error from reaching the next stages. However, SETs and METs occurring in the majority voter can transmit a wrong data to the next logic path.

- **FARHAD**

As Figure 4.6 shows, FARHAD has components prone to soft errors including CLA adder, RCA, register, comparator and checkpoint pointer.

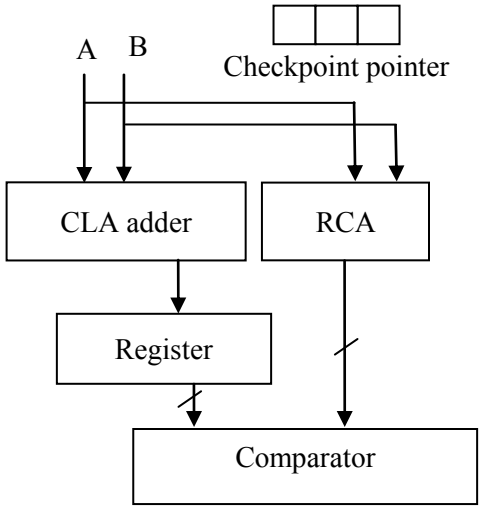

Figure 4.6: FARHAD's data path.

The comparator identifies any SETs or METs occurring in CLA adder and RCA. SEUs and MBUs in registers are detectable by the comparator. The comparator detects faults in any of the above three components.

The effect of SETs and METs in the comparator can be studied under two different conditions: 1) CLA adder and RCA produce similar results but the comparator mistakenly reports a mismatch. As a result the processor flushes the pipeline back to the most recent branch instruction unnecessarily. Consequently we pay a negligible

performance penalty. Unlike TMR and QTRA, the produced soft error does not propagate to the next stage. 2) RCA and CLA adder produce different results but the comparator mistakenly reports a match due to an SET or MET. Therefore, erroneous data propagates to the next stage. The second scenario is extremely unlikely, as it is very rare to have two particles hitting a single device at the same time. Moreover, it is very unlikely for two SETs (i.e., the one in the adder and the one in the comparator) to occur at the logic locations associated with the same bit.

As we save the checkpoint associated with the last branch instruction in the checkpoint pointer, in case of any SEUs and MBUs, the checkpoint associated with an earlier branch instruction if is not released is saved in checkpoint pointer. Therefore, even if an SET or MET happen to other components including adders or register, no failure occurs; the processor flushes instructions and loses small performance. Nevertheless, an error in checkpoint pointer can cause us problems in case that another fault already has happened in FARHAD's components and the faulty checkpoint pointer points to a branch which has been committed. Note that both explained scenarios happen too rarely as the probability of having 2 faults at the same or in a small period of time is extremely low. Otherwise, regardless of checkpoint pointer content, the addition result is fault-free and there is no need for recovery process if there is only a fault in checkpoint pointer.

In the light of the above observations, QTRA is the most vulnerable method. TMR comes second. Dual-CLA, while having architecture similar to FARHAD, has higher number of gates and therefore is more vulnerable. We conclude from our qualitative analysis that FARHAD has the lowest vulnerability among all studied methods.

4.5.2 Quantitative

Reliability shows how probable it is for a component to survive till time *t* [7]. Reliability depends on a parameter referred to as failure rate,λ, which is defined as follows:

$$R(t) = e^{-\lambda t} \quad (1)$$

Failure rate is expressed using equation (2) where π_P, π_Q, π_L, π_T and π_E are the pin, quality, learning, temperature and environment factors, respectively. C_1 and C_2 are the complexity factors which depend on the number of gates and pins used in the device. More details can be found in [4].

$$\lambda = (C_1\pi_T + C_2\pi_T)\pi_P\pi_Q\pi_L \text{ Failures/}10^6 \text{ Hours} \quad (2)$$

In order to estimate reliability we assume that beside C_1 and C_2 all other factors are the same for all methods. Moreover, since FARHAD and other studied methods are composed of some sub-blocks, first the reliability of each sub-block is estimated. Then depending on how the sub-blocks are connected the total reliability is calculated.

Equation 3 and 4 are used to estimate reliability of systems which have components in serial and parallel respectively [31].

$$RR(t) = 1 - [(1 - R_1(t)) * (1 - R_2(t)) * \ldots (1 - R_n(t)) \quad (3)$$

$$R(t) = R_1(t) * R_2(t) * \ldots R_n(t) \quad (4)$$

Figure 4.7 shows reliability estimate for FARHAD along with the other methods using (1), (2) (3) and (4). While the Y axis shows reliability, the X axis shows time in 10^6 hours.

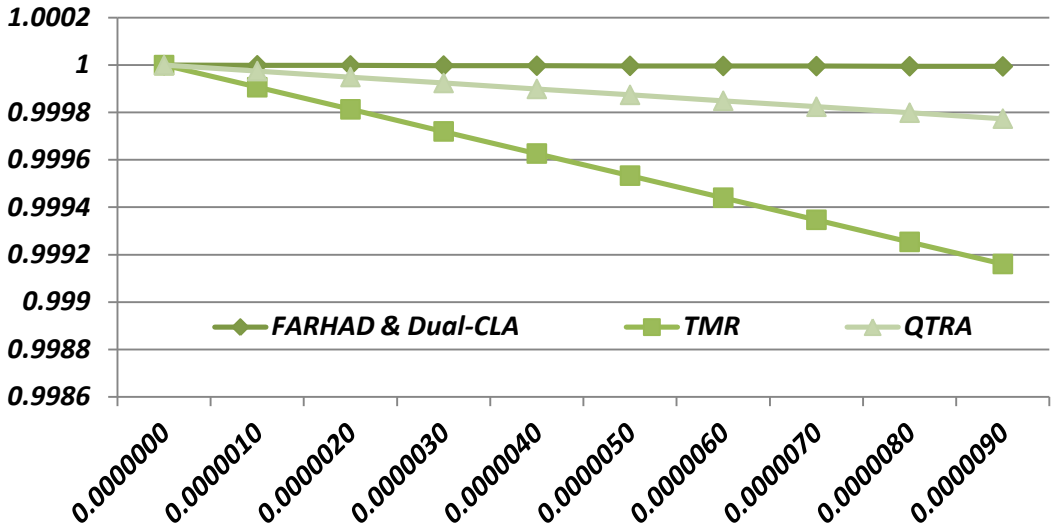

Figure 4.7: Reliability estimation.

As Figure 4.7 reports, FARHAD and Dual-CLA have the highest reliability. QTRA and TMR come next. Since the complexity factors of FARHAD and Dual-CLA are similar, their failure rates and thus reliabilities are equal.

Chapter 5

Conclusion and Future work

In this work we introduced FARHAD as a low-power fault-tolerant adder for high performance processor. Previous studied have introduced many approaches to protect arithmetic units including adder against faults. But the proposed methods suffer either from high energy consumption and area overhead or huge performance loss. Since FARHAD is employed in high-performance processors, its purpose is to reduce the power dissipation and area overhead of previous methods while coming with no performance penalty.

FARHAD uses dual modular redundancy for error detection but not in the conventional way. Unlike previous methods, which use the same adder as the replica to produce the checker to find the errors, FARHAD uses an energy-efficient but slow adder. It results in less power dissipation and less area overhead compared to other studied approaches including TMR. On the other hand the slow adder does not impact overall performance negatively, as it is working in the background. The main adder's result is generated on time and is passed to the other parts of the processor.

For error recovery, FARHAD relies on checkpointing, also used by high-performance processors to recover from branch misprediction. In case of any errors, FARHAD flushes the pipeline back to the most recent branch instruction.

As the error detection process of FARHAD is performed in parallel to processor execution, FARHAD comes with no performance penalty. FARHAD reduces power overhead significantly compared to modular redundant solutions.

5.1 Future Work

In this work we proposed an alternative to protect adder against faults. However, applying FARHAD's idea on multiplier and divider can be considered as a future work to introduce a fault tolerant ALU.

Moreover in this work we analyzed the reliability of all studied methods and FARHAD qualitatively and quantitatively. Injecting fault in FARHAD and other studied methods would be the next step to show how reliable FARHAD is.

Bibliography

[1] C. Constantinescu, "Trends and challenges in VLSI circuit reliability," *Micro, IEEE* , vol.23, no.4, pp.14,19, July-Aug. 2003

[2] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on* , vol., no., pp.389,398, 2002

[3] B. W. Johnson, Design and Analysis of Fault-Tolerant Digital Systems. Reading, MA: Addison-Wesley Publishing Company, 1989

[4] S. Hong and S. Kim , "Lizard: Energy-efficient hard fault detection, diagnosis and isolation in the ALU," *Computer Design (ICCD), 2010 IEEE International Conference on* , vol., no., pp.342,349, 3-6 Oct. 2010

[5] W. J. Townsend, J. A. Abraham, and E. E. Swartzlander, "Quadruple Time Redundancy Adders," *Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on* , vol., no., pp.250,256, 3-5 Nov. 2003

[6] J. Srinivasan, S. V. Adve, P. Bose and J. A. Rivers, "The case for lifetime reliability-aware microprocessors," *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on* , vol., no., pp. 276- 287, 19-23 June 2004

[7] Daniel Sorin, Fault Tolerant Computer Architecture, Morgan & Claypool Publishers, 2009

[8]C. Rusu, A. Bougerol, L. Anghel, C. Weulerse, N. Buard, S. Benhammadi, N. Renaud, G. Hubert, F. Wrobel, and R. Gaillard, "Multiple event transient induced by nuclear reactions in cmos logic cells," *On-Line Testing Symposium, 2007. IOLTS 07. 13th IEEE International* , vol., no., pp.137,145, 8-11 July 2007

[9] D. Rossi, M. Omana, F. Toma, and C. Metra, "Multiple transient faults in logic: An issue for next generation ics?," *Defect and Fault Tolerance in VLSI Systems, 2005. DFT 2005. 20th IEEE International Symposium on* , vol., no., pp.352,360, 3-5 Oct. 2005

[10] J. A. Maestro and P. Reviriego, "Study of the effects of mbus on the reliability of a 150 nm sram device," *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE* , vol., no., pp.930,935, 8-13 June 2008

[11] D. Falgure and S. Petit, "A statistical method to extract mbu without scrambling information," *Nuclear Science, IEEE Transactions on* , vol.54, no.4, pp.920,923, Aug. 2007

[12] F. Wang, Yuan. Xie, R. Rajaraman and B. Vaidyanathan, "Soft Error Rate Analysis for Combinational Logic Using An Accurate Electrical Masking Model," *Dependable and Secure Computing, IEEE Transactions on* , vol.8, no.1, pp.137,146, Jan.-Feb. 2011

[13] D. Burger, T.M. Austin and S. Bennett (1996) Evaluating Future Microprocessors: The Simple Scalar Tool Set. Technical Report, University of Wisconsin-Madison, New York

[14] S. Hong and S. Kim, "TEPS: Transient Error Protection Utilizing Sub-word Parallelism," *VLSI, 2009. ISVLSI '09. IEEE Computer Society Annual Symposium on* , vol., no., pp.286,291, 13-15 May 2009

[15] M. Yilmaz, A. Meixner, S. Ozev and D. J. Sorin, "Lazy Error Detection for Microprocessor Functional Units," *Defect and Fault-Tolerance in VLSI Systems, 2007. DFT '07. 22nd IEEE International Symposium on* , vol., no., pp.361-369, 26-28 Sept. 2007

[16] B. W. Johnson, "Fault-Tolerant Microprocessor-Based Systems," *Micro, IEEE* , vol.4, no.6, pp.6,21, Dec. 1984

[17] J. Li and E. E. Swartzlander , "Concurrent error detection in ALUs by recomputing with rotated operands," *Defect and Fault Tolerance in VLSI Systems, 1992. Proceedings., 1992 IEEE International Workshop on* , vol., no., pp.109,116, 4-6 Nov 1992

[18] B. W. Johnson, J. H. Aylor and H. H. Hana, "Efficient use of time and hardware redundancy for concurrent error detection in a 32-bit VLSI adder," *Solid-State Circuits, IEEE Journal of* , vol.23, no.1, pp.208,215, Feb. 1988

[19] S. A. Al-Arian and M. B. Gumusel, "HPTR: Hardware partition in time redundancy technique for fault tolerance," *Southeastcon '92, Proceedings., IEEE* , vol., no., pp.630-633 vol.2, 12-15 Apr 1992

[20] I. Lee, M. Basoglu, M. Sullivan, D. Hyun Yoon, L. Kaplan and M. Erez, Survey of error and fault detection mechanism, University of Texas, April 2011

[21] A. Meixner, M. E. Bauer and D. J. Sorin, "Argus: Low-Cost, Comprehensive Error Detection in Simple Cores," *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on* , vol., no., pp.210,222, 1-5 Dec. 2007

[22] A. Moshovos, "Checkpointing alternatives for high-performance, power-aware processors," *Low Power Electronics and Design, 2003. ISLPED '03. Proceedings of the 2003 International Symposium on* , vol., no., pp.318,321, 25-27 Aug. 2003

[23] Standard Performance Evaluation Corporation, http://www.spec.org

[24] Data path and Building Block IPs, http://www.synopsys.com/dw/buildingblock.php

[25] H. Ling, “High Speed Binary Parallel Adder,” *Electronic Computers, IEEE Transactions on* , vol.EC-15, no.5, pp.799,802, Oct. 1966

[26] P. M. Kogge and H. S Stone, “A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations,” *Computers, IEEE Transactions on* , vol.C-22, no.8, pp.786,793, Aug. 1973

[27] R. P. Brent and H. T. Kung, “A Regular Layout for Parallel Adders” *Computers, IEEE Transactions on* , vol.C-31, no.3, pp.260,264, March 1982

[28] T. Han and D. A. Carlson, "Fast area-efficient VLSI adders" *Computer Arithmetic (ARITH), 1987 IEEE 8th Symposium on* , vol., no., pp.49,56, 18-21 May 1987

[29] Behrooz Parhami, Computer Arithmetic: Algorithms and hardware Design, Oxford University Press, New York, 2000

[30] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. 2003. “A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor,” *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on* , vol., no., pp.29,40, 3-5 Dec. 2003

[31] Koren Israel, C. Mani Krishna, Fault-Tolerant Systems, Morgan Kaufmann, 2007

Appendix A

Methodology

We use SimpleScalar 3.0 toolset with ALPHA configuration to model our processor for performance evaluation and study the timing issues that FARHAD may confront [13]. As We use a subset of SPECK'2K benchmarks compiled for ALPHA [21]. Table 1 reports the processor configuration used in this study. We run benchmarks for one billion instructions.

Table A.1: Simulated processor configuration.

Processor Core	
Fetch Queue/LSQ/RUU Size Decode/Issue/Commit Width Int. & FP Functional Unit	4/32/64 Instruction 4/4/4Instruction per cycle 4 ALU, 1 mult/div
Memory Hierarchy	
Memory Latency L1 D-cache, L1 I-cache L1 hit Latency Unified L2 cache L2 hit Latency	First_chuck: 512, Inter_chunck: 4 16 KB, 4-way 3 cycles 1 MB, 8-way 32 cycles
Branch Prediction	
Combinational Bimodal 2level BTB Misprediction penalty	meta size: 1024 2048 8 bits history and 1024 array 512, 4-way 3 cycles

All methods studied in this work are implemented using Verilog. We use Synopsys tool chain to synthesize the designs and measure the delay, area and power dissipation [22]. In order to have the best efficiency we use Synopsys DesignWare Building Block IPs to implement arithmetic units including adders [24]. While for CLA adder we use "cla" implementation, for RCA we use "rpl". All designs are compiled by Design power

compiler with medium mapping and area effort using TSMC 90nm CMOS process technology.

To measure power dissipation, we feed all methods operands obtained from the application benchmarks using SimpleScalar. Figure A.1 shows our power estimation flow.

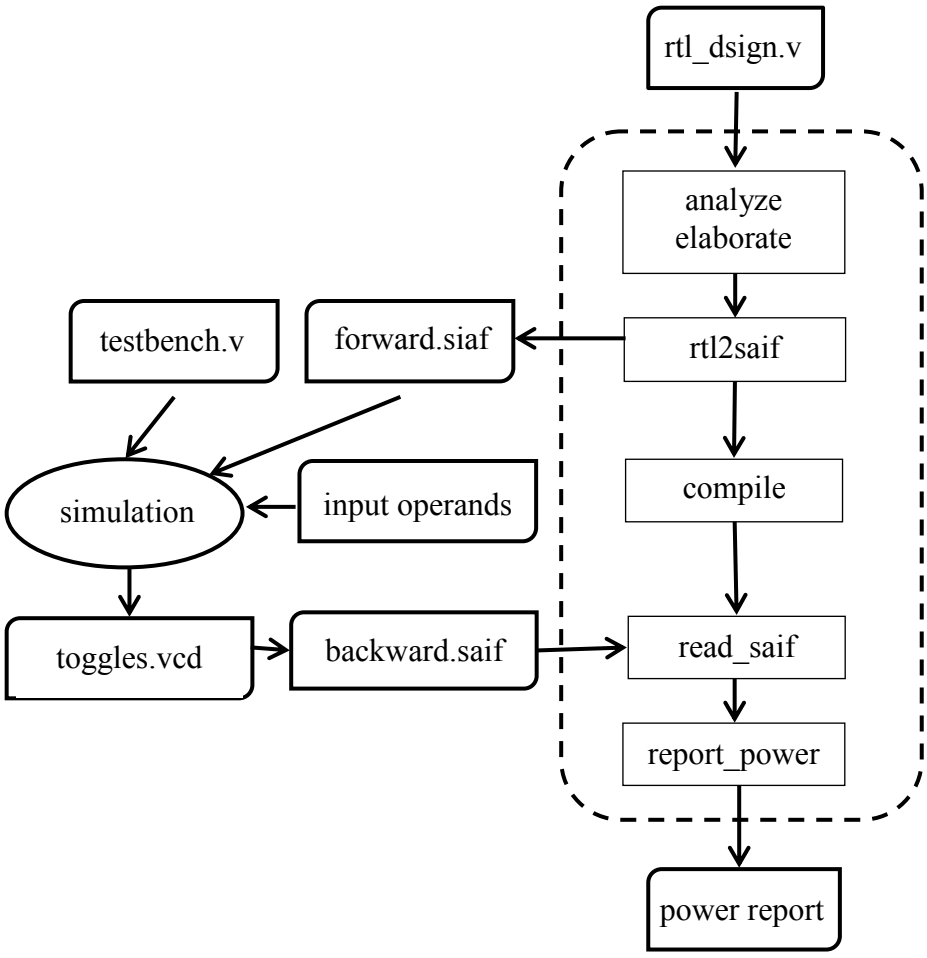

Figure A.1: Power estimation flow.

As Figure A.1 depicts, first we *analyze* and *elaborate* all our rtl files using Synopsys. "analyze" command check the Verilog syntax to find errors. "elaborate" command translate the design into a technology-independent design. Then we extract the *forward*

saif file (Switching Activity Interchange Format) from the rtl files. Afterwards, by using any tool capable of generating VCD including Modelsim and VCS, we simulate the whole design and dump the switching activities in VCD format. Then, by using *vcd2saif* command we generate the *backward saif* file which is readable by Synopsys to produce the power report.

As faults occur rarely, they have a negligible impact on performance and power. Therefore all performance and power measurements in this work assume the above.