ACADEMIE DE MONTPELLIER

UNIVERSITE MONTPELLIER II
SCIENCES ET TECHNIQUES DU LANGUEDOC

THESE

Présentée à l'Université de Montpellier II Sciences et Techniques du Languedoc
pour obtenir le **DIPLOME DE DOCTORAT**

Spécialité: **Microélectronique**
Formation Doctorale: **Système Automatiques et Microélectroniques**
Ecole Doctorale: **Information, Structures, Systèmes**

ARCHITECTURE HYBRIDE TOLERANTE AUX FAUTES POUR L'AMELIORATION DE LA ROBUSTESSE DES CIRCUITS ET SYSTEMES INTEGRES NUMERIQUES

(A HYBRID FAULT-TOLERANT ARCHITECTURE FOR ROBUSTNESS IMPROVEMENT OF DIGITAL INTEGRATED CIRCUITS AND SYSTEMS)

par

Duc Anh TRAN

Soutenue le **21 Décembre 2012**, devant le Jury compose de:

Mme.	Lirida NAVINER, Professeur, Télécom ParisTech	Rapporteur
M.	Matteo SONZA REORDA, Professeur, Politecnico di Torino	Rapporteur
M.	Régis LEVEUGLE, Professeur, Grenoble INP	Examinateur
M.	Arnaud VIRAZEL, Maître de conférences, Université Montpellier II	Encadrant
M.	Serge PRAVOSSOUDOVITCH, Professeur, Université Montpellier II	Directeur de thèse
M.	Patrick GIRARD, Directeur de recherche CNRS, LIRMM, Montpellier	co-Directeur de thèse

Acknowledgements

I would like to express my sincere thanks and appreciations to all the people who have contributed to my researches, as well as to the writing of this thesis.

First of all, I am grateful to my thesis co-directors, Prof. Serge Pravossoudovitch and Dr. Patrick Girard, who have brought up the thematic of my researches and given me the opportunity to join their team. I would like to express my sincere thanks to my supervisor, Dr. Arnaud Virazel, for his patience, motivation and continuous supports. His guidance has allowed me to surpass many difficulties, from the very first steps of my research to the writing of this thesis.

Besides, I would like to thank Prof. Hans-Joachim Wunderlich for his invaluable advices and suggestions, which have allowed significant accelerations in several steps of my research. My sincere thanks also go to Dr. Alberto Bosio, Dr. Aida Todri, Dr. Luigi Dilillo and Mr. Michael Imhof for their contributions to my papers and journals.

I would like to thank my thesis committee: Prof. Régis Leveugle, Prof. Lirida Naviner, Prof. Matteo Sonza Reorda, for their encouragements, insightful comments and constructive questions.

I take this opportunity to record my sincere gratitude to all the colleagues at LIRMM, who have given me an ideal working place during the last three years.

Last but not least, I would like to thank my family for their love and supports during all my life. My special thanks to Hai Anh Trinh, my beloved wife for her encouragements as well as her contributions to the review of this thesis.

Duc Anh Tran.

Contents

Introduction 1

Chapter 1 Contexts and Motivations 4

1.1 Robustness of digital systems 5

1.1.1 CMOS technology 6

1.1.2 Classification of failure mechanisms 8

1.1.3 Discussion 13

1.2 Fault-tolerant architectures 13

1.2.1 Fundamentals 14

1.2.2 Hardware redundancy 15

1.2.3 Information redundancy 18

1.2.4 Timing redundancy 20

1.2.5 Hybrid fault-tolerance 21

1.2.6 Discussion 23

1.3 Robustness improvement of digital systems 23

1.3.1 Fault-tolerance in memories 23

1.3.2 Fault-tolerance in logic circuits 25

1.3.3 Discussion 27

1.4 Summary 28

Chapter 2 The Hybrid Fault-Tolerant Architecture 29

2.1 Principles of hybrid fault-tolerance 30

2.2 Error detection 31

2.2.1 Concurrent Error Detection 31

2.2.2 Parity codes 32

2.2.3 Duplication/Comparison 33

2.2.4 Conclusion 39

2.3 Transient error correction 40

2.3.1 Input register 40

2.3.2 Reset signal 42

2.3.3 Transient error correction mechanism 42

2.3.4 Control logic and timing constraints 44

2.3.5 Conclusion 47

2.4 Permanent error correction 47

Contents

2.4.1 Input de-multiplexer 48
2.4.2 Output multiplexer 49
2.4.3 Reconfiguration finite state machine 51
2.4.4 Control logic and timing constraints 53
2.5 Summary 55
Chapter 3 Evaluation of the Hybrid Fault-Tolerant Architecture 57
3.1 Context 58
3.2 Architecture description 59
3.2.1 Hybrid Fault-Tolerant Architecture 60
3.2.2 TMR architecture 60
3.2.3 Discussion 62
3.3 Logic synthesis 62
3.3.1 Dynamic CMOS standard cell creation 63
3.3.2 Combinational logic synthesis 64
3.3.3 Redundant modules synthesis 66
3.3.4 Fault-tolerant architecture synthesis 72
3.4 Timing behavior of hybrid fault-tolerant architecture 73
3.4.1 Comparator simulation 74
3.4.2 Control logic simulation 76
3.4.3 Hybrid fault-tolerant architecture simulation 78
3.4.4 Discussion 80
3.5 Power simulation 80
3.6 Summary 81
Chapter 4 Extended Usage of the Hybrid Fault-Tolerant Architecture 82
4.1 Aging phenomenon 83
4.1.1 Lifetime improvement 83
4.1.2 Usage of FSMs 84
4.1.3 Discussion 85
4.2 Application of the hybrid fault-tolerant method in pipeline architectures 86
4.2.1 Basic of pipeline architecture 86
4.2.2 Fault-tolerance for pipeline architecture 87
4.2.3 Hybrid fault-tolerant design for pipeline architecture 90
4.2.4 Conclusion 92
4.3 SEU protection 92

4.3.1 SEU protection techniques ... 92
4.3.2 SEU protection for the hybrid fault-tolerant architecture ... 94
4.3.3 Discussion ... 96
4.4 Summary ... 96
Conclusion ... 97
Appendix A ... 99
A1. Combinational logic extraction ... 99
A2. RTL descriptions of the hybrid fault-tolerant architecture ... 100
Top-level module without fault injections ... 100
Top-level module with fault injections ... 101
Input register ... 102
Input demultiplexer ... 103
Output multiplexer ... 104
Output register ... 107
Pseudo-dynamic comparator ... 107
Control logic module ... 108
A3. RTL descriptions of TMR architectures ... 114
Top-level module of Partial TMR architecture ... 114
Top-level module of Full TMR architecture ... 114
Word-voter ... 115
Scientific Contributions ... 117
References ... 118
List of Figures ... 122
List of Tables ... 124

Introduction

The Moore's law is known as the best description of the Complementary Metal Oxide Semiconductor (CMOS) technology evolution [MOO65]. Established in 1965, it predicted that as a result of continuous scaling in transistor feature sizes, number of devices in Integrated Circuits (IC) would double every eighteen months. This evolution allows the production of smaller and cheaper ICs with more and more functionalities. Furthermore, smaller devices are faster and consume less energy. Consequently, CMOS evolution has enabled to transition of digital systems from specialized applications to ubiquitous mass products. Today, these systems can be found in almost every modern electrical device such as cars, television sets, personal computers, cellular phones, etc.

While supporting the need for competitive mass products, CMOS evolution also influences the reliability of digital systems [ITR11]. Different factors are responsible for transient and permanent faults that affect robustness of digital circuits and systems. First of all, a high integration density provokes a high defect density. Together with aging phenomenon, it may cause permanent defects that result in hard errors during circuit operations. Besides, nanometer-scale devices are more vulnerable to cosmic radiations and interference phenomenon, which may cause transient faults. These faults are observed at circuit outputs as soft (single event-upsets SEUs affecting sequential elements and single event-transients SETs affecting combinational logic) and timing errors (additional delays in combinational logics that cause timing constraint violations in sequential elements of logic circuits). In advanced CMOS technology nodes, these problems affect not only critical systems that require high reliability, like circuits used in spatial or medical domains, but also consumer electronic systems. Therefore, robustness improvement becomes a crucial requirement for CMOS electronic circuits and systems.

Robustness improvement of digital circuits and systems is getting more and more difficult for every introduced CMOS technology node because treating faults at physical level by adjusting manufacturing process parameters is no longer feasible. Therefore, fault-tolerance techniques, which deal with faults at design level, have become essential to fulfill the required robustness of future digital CMOS circuits and systems. These techniques employ information, timing and hardware redundancies to guarantee correct operations despite the presence of faults [KOR07].

In memory part of digital systems, the use of fault-tolerant techniques has been proven necessary and efficient. Information (error detection and correction codes, [KOR07]) and hardware (spare memory words, columns and cells [SCH01, NIC03, NIC05, SU05]) redundancies are generally employed to deal with transient and permanent faults in memories. However, fault-tolerance in random logic circuits of digital systems remains a challenge. These circuits are composed of combinational logic and sequential elements such as latches and flip-flops. Different techniques have been proposed to protect the sequential part from hard errors and/or SEUs [LYO62, ERN03, ZHA06, DAS09, IMH11]. Some of these techniques are also efficient for SETs and timing errors that occurred in the combinational part of logic circuits [ERN03, DAS09]. Besides, in order to protect this part from hard errors, Triple Modular Redundancy (TMR) architecture has been proven to be an efficient method [VIAL08, VIAL09]. Although each type of error has several corresponding solutions, combining all these techniques for robustness improvement may require very high level of redundancy and thus, not be applicable in mass products.

In the state-of-the-art solutions presented above, beside fault-tolerance capability, area overhead is the main optimization criterion. In [VIAL08, VIAL09], authors have introduced manufacturing yield enhancement as a new goal. Besides, power consumption is also a rising issue in advanced CMOS

technology nodes. In fact, as fault-tolerance becomes necessary in mass products, limiting power consumption increase of these techniques is one of the key factors in digital design. However, this criterion has only been studied in fault-tolerant communication [PUL07], but not for random logic circuits.

Given the new requirements in fault-tolerance field, this manuscript studies the possibility to combine different types of redundancy in a hybrid fault-tolerant architecture, which allows the detection and correction of all transient and permanent errors in combinational part of logic circuits. Besides fault-tolerance capability, we also reach optimized costs in both silicon area and power consumption compared to existing solutions.

In the proposed hybrid fault-tolerant architecture, information redundancy consists of duplicating combinational part of logic circuits and comparing their outputs to detect all kind of errors. Timing redundancy, which performs re-computation of affected input vector, allows transient errors correction at low silicon area costs. Finally, hardware redundancy that requires one additional combinational logic module enables permanent error correction via re-configuration. As only two out of three redundant combinational logics are running in parallel, the proposed architecture offer about 33% power consumption saving compared to TMR architectures, while having similar silicon area. Besides, this solution can be used in several contexts such as: 1) Dealing with aging phenomenon; 2) Protecting pipeline architectures from hard, SETs and timing errors; and 3) Being combined with register-level SEU protection techniques to tolerate faults in both combinational and sequential parts of logic circuits at optimized area overhead.

The following of this manuscript is divided in four chapters:

- Chapter 1 details the contexts and motivations of our research. The first part presents various advantages of CMOS evolution, as well as how diverse kinds of fault and errors affect robustness of digital circuits and systems in advanced technology nodes. The second part of this chapter studies the principle of different fault-tolerant techniques, classified in four categories depending on their employed redundant resources. Finally, the last part provides an overview on state-of-the-art solutions for permanent and transient errors in different parts of digital systems.
- Chapter 2 proposes a hybrid fault-tolerance architecture targeting permanent and transient faults in combinational part of logic circuits. This architecture is built step-by-step as three fault-tolerance levels. The first level consists of using information redundancy to detect all kinds of errors, regardless of their nature. The second fault-tolerance level adds timing redundancy to correct transient errors. Finally, the third level completes the hybrid fault-tolerant architecture with hardware redundancy, which allows permanent errors correction.
- Chapter 3 consists of evaluating the proposed hybrid fault-tolerant architecture. This method is compared to TMR architectures in order to prove its advantages in terms of fault-tolerance capability, area overhead and power consumption. The evaluations are performed using simulations done with Electronic Design Automation (EDA) tools. Fault-tolerance techniques are implemented for combinational part of ISCAS'85 and ITC'99 benchmarks circuits [ISCAS85, ITC99]. The resulted architectures are then mapped on a 45nm standard cells library [NOCL]. The final netlists are then used to simulate the architecture's behavior in different error occurrence scenarios, as well as to estimate their power consumption.
- Chapter 4 proposes extended usages of the hybrid fault-tolerant architecture for various applications. First of all, we study how the proposed architecture can be used in the context of aging phenomenon: 1) to improve lifetime of digital logic circuits and 2) to optimized the fault-tolerance scheme in case of high error occurrence rates. Then, we investigate the

possibility to use the hybrid fault-tolerant technique in pipeline architectures. The objective is to add hard errors tolerance to state-of-the-art solutions, which only tolerate SETs and timing errors in combinational part of these architectures. Finally, we propose to use the hybrid fault-tolerant architecture in combination with a register-level SEU protection technique and thus, provide an advanced solution for faults in both combinational and sequential parts of logic circuits.

Chapter 1

Contexts and Motivations

Chapter 1 Contexts and Motivations 4
1.1 Robustness of digital systems 5
1.1.1 CMOS technology 6
1.1.2 Classification of failure mechanisms........ 8
1.1.3 Discussion 13
1.2 Fault-tolerant architectures 13
1.2.1 Fundamentals 14
1.2.2 Hardware redundancy 15
1.2.3 Information redundancy 18
1.2.4 Timing redundancy 20
1.2.5 Hybrid fault-tolerance 21
1.2.6 Discussion 23
1.3 Robustness improvement of digital systems 23
1.3.1 Fault-tolerance in memories 23
1.3.2 Fault-tolerance in logic circuits 25
1.3.3 Discussion 27
1.4 Summary 28

Digital systems have transitioned from specialized applications to ubiquitous mass products. Today, a consumer mobile phone may contain a processor with a higher computing power than a super computer in the early 1990s. This revolution has been conducted by the evolution of CMOS (Complementary Metal Oxide Semiconductor) technology which allows the production of smaller and cheaper Integrated Circuits (IC) with higher performance and lower power consumption. While supporting the need for competitive mass products, this evolution also influences the reliability of digital systems. In particular, increasing apparition rate of faults and errors during manufacturing processes and ICs' lifetime make robustness one of the upcoming key requirements in many application areas, including safety critical applications and mass products. However, improving digital system reliability is getting more and more difficult for every introduced technology node because treating faults on the physical level by adjusting manufacturing process parameters is no longer feasible. Therefore, fault-tolerance techniques which deal with faults at the design level have become essential to fulfill the required robustness of future digital CMOS circuits and systems.

In this chapter, we study different issues of technology evolution with regards to robustness of digital systems, and explain how fault-tolerance can be a solution for these problems. The chapter is organized as follows. In the first section, we present the evolution of CMOS technology and how it can impact the reliability of digital systems by inducing faults in ICs. These faults are classified into different categories depending on their duration (transient, intermittent and permanent faults) and their impacts on ICs' operation (hard, soft and timing errors). Then, in the second section, we study the principle of fault-tolerance techniques which allow systems to operate correctly despite the presence of faults. Four categories (hardware, information, timing and hybrid fault-tolerance) are detailed with concrete architectural examples. Finally, in the third section, we study different existing solutions which tolerate faults in both memories and logic circuits of integrated systems.

1.1 Robustness of digital systems

Robustness of digital systems is their ability to cope with anomalies during execution, *i.e.* providing good results despite the presence of faults. Different sources of fault can be grouped into five categories, depending on their occurrence during the lifetime of a system:

- Design errors: errors during design phases, which result in incorrect hardware implementation of system specifications.
- Manufacturing defects: failures during fabrication phases, which modify logic function of the system or degrade its functional characteristics.
- Operation errors and malicious attacks: human errors, unintentional or voluntary, to operate the system under abnormal conditions.
- Interference phenomena: interactions of the digital system with its environment during operation.
- Physical degradations: aging phenomena which degrade components of the digital system.

In the scope of this thesis, we aim to deal with physical faults [LAP95] which are caused by manufacturing defects, interference phenomena and physical degradations. This section presents how CMOS technology evolution increases the apparition frequency of these problems as well as their impacts on digital circuits and systems. It is divided into two sub-sections. The first sub-section details key points of CMOS technology evolution which is used to manufacture IC, the material blocks that built digital systems. We analyze also consequences of each technology progress with regards to reliability of these systems. Then, in the second sub-section, we detail how different factors, such as manufacturing defects, variability, interference and aging phenomena, result in fault in ICs, which may lead to failures of

digital systems. These faults and errors will then be classified according to their impact on ICs' functionality.

1.1.1 CMOS technology

Technology evolution

In 1965, Gordon Moore made a prediction in [MOO65] that the number of transistors in an IC roughly doubles every two years. This statement has become the guide line for the entire semiconductor industry and therefore leads to CMOS technology evolution.

The most important evolution of CMOS technology is the downscaling in feature size of manufacturing processes. This parameter refers to the minimum dimension of a Metal-Oxide-Semiconductor Field-Effect Transistor (MOSFET) that can be built on an IC. It has been reduced from 10µm in 1971 to 22nm in 2011. And the ITRS predicts that this progression will continue for at least another decade [ITR11]. Figure 1.1 shows this evolution with square symbols that represent existing manufacturing processes and triangular symbols that represent predicted technology nodes [INT10, ITR11].

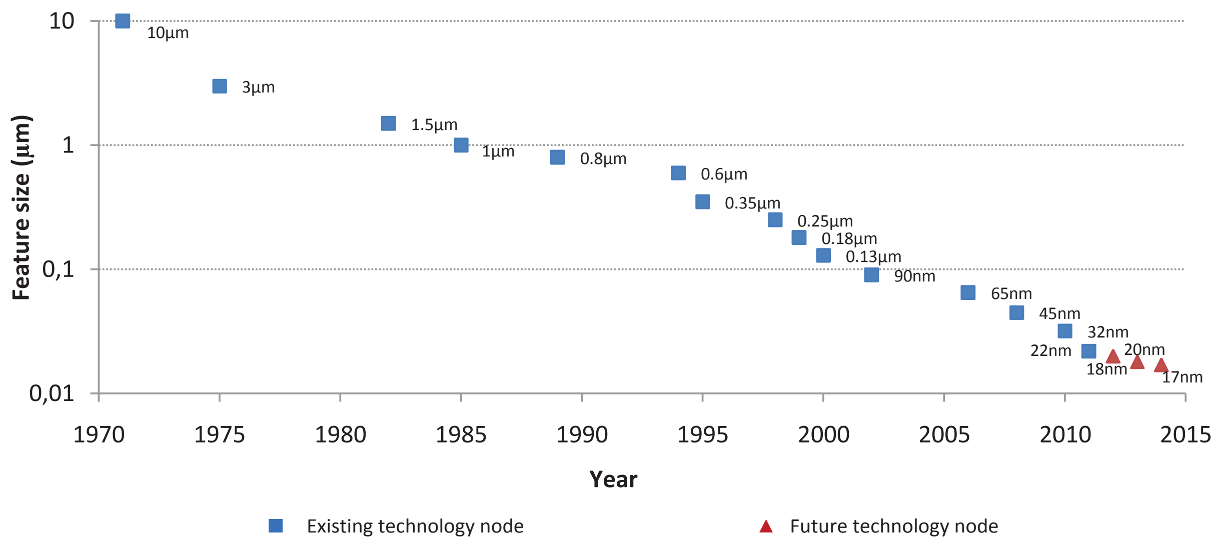


Figure 1.1 CMOS Technology Nodes

By downscaling feature size, CMOS evolution allows integration of more and more transistors on an IC. In 1972, the first Intel microprocessor was introduced with only 2300 transistors [INT10]. In 2011, the six-core microprocessor Core i7 (Sandy Bridge-E) contained more than 2.27 billion transistors. This corresponds to a compounding annual growth rate of more than 40% over 40 years. With this incredible growth rate, illustrated in Figure 1.2, the industry has transformed from Small-Scale Integration (SSI, up to 10 logic gates per IC) through Medium-Scale Integration (MSI, up to 1000 gates per IC) and Large-Scale Integration (LSI, up to 10,000 gates per IC), to today Very Large-Scale Integration (VLSI) with many millions of logic gates per IC.

Miniaturization of CMOS technology also offers manufacturing cost reduction which is the most important factor in the semiconductor industry. Historically, Cost Per Function (CPF) of ICs, i.e. cost per transistor, decreased by an average of 29% each year. This means CPF is halved every two years. In 2011, it is estimated at 5.5 micro-cents per transistor. And the ITRS predicts that this reduction rate of 29% will continue in the next decade [ITR11].

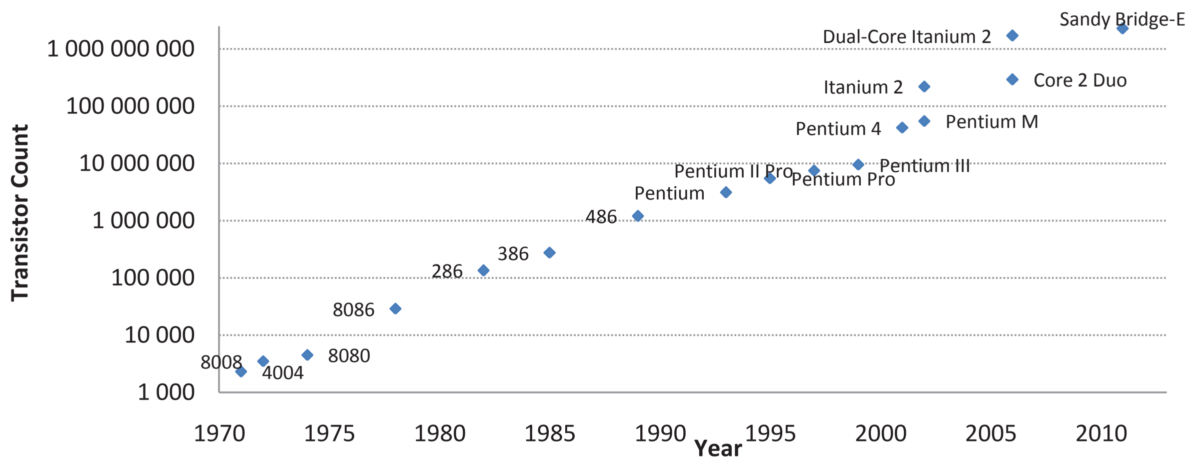


Figure 1.2 Transistor Counts of Intel Microprocessors, Data Source: [INT10]

Beside higher integration level and lower manufacturing cost, steady downscaling efforts in CMOS technology can be explained by Dennard's scaling theory [DEN74]: smaller transistors are faster and consume less power. Between two technology nodes, main dimensions of MOSFETs, i.e. channel length/width and oxide thickness, are scaled with the same factor κ. Consequently, channel resistances remain unchanged while gate capacitances are reduced by κ. Hence, transistor delays are also scaled by the same factor. As transistors become faster, ICs can be operate at higher frequencies. Figure 1.3 shows clock frequency change in Intel microprocessors since 1971. We see that the frequency doubled almost every 34 months. However, in 2005, the frequency scaling process has reached the power wall limit at about 3 GHz. In fact, higher switching activity of transistors leads to higher power consumption. Even though these small transistors do not consume much, hundred millions of them are switching at the same time in less than five hundred millimeter square IC. This results in significant power density that must be limited to avoid breakdown of physical materials.

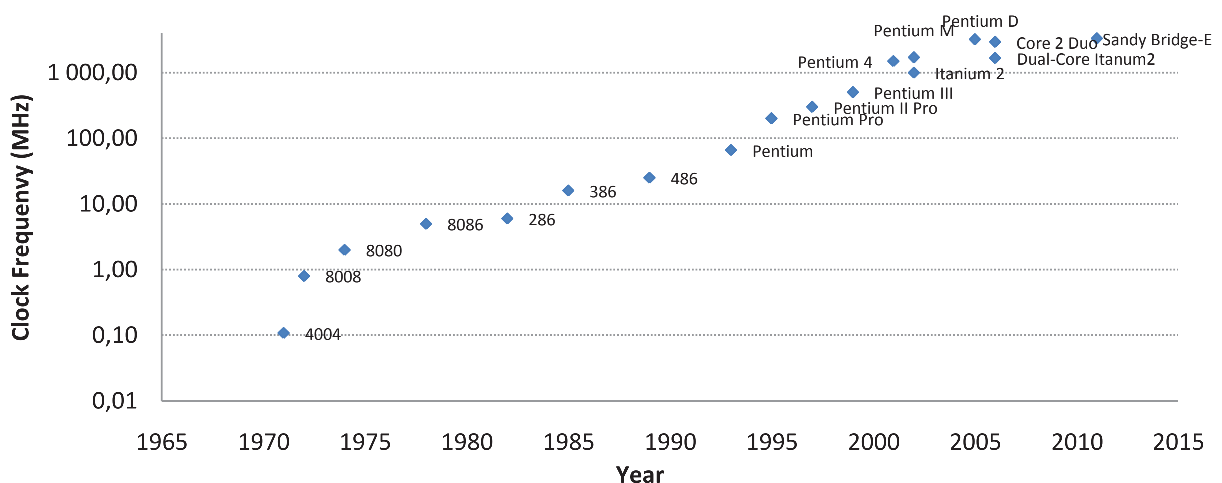


Figure 1.3 Clock Frequencies of Intel Microprocessors, Data Source: [INT10]

To reduce power dissipation of CMOS devices, the supply voltage V_{dd} is also scaled, because dynamic power is proportional to V_{dd}^2 while leakage power is proportional to V_{dd}. However, voltage scaling only started in the late 80s because the industry had settled on 5V supplies in the early 70s to be compatible

with bipolar Transistor-Transistor Logic (TTL) [CRI07]. As power dissipation became unsustainable, this standard finally collapsed. V_{dd} was scaled within few years, first to 3.3V then to 2.5V, etc. In 2011, supply voltage of high-performance ICs was at 0.9V and predicted to be reduced to 0.66V in 2021 [ITR11].

Reliability

Reliability of digital circuits and systems is defined as their ability to perform required functions under stated conditions and for a specified period of time [STA11]. In other word, circuits and systems have reliability issues when they fail to operate correctly and to provide expected results. These events are called failures.

While offering many advantages, each new CMOS technology node is facing reliability issues [ITR11]. As the miniaturization trend approaches physical limits of operation and manufacturing, failures may occur at all phases of digital systems' lifetime: infant mortality, working life or wearout period [GIR10]. The bathtub curve in Figure 1.4 [GIR10] shows how different types of failure affect ICs during its lifetime. There are three types of failure; each of them has major effects during one the three phases:

- Early failures that dominate infant mortality phase are mostly due to manufacturing issues. While transistors size shrinks, manufacturing processes are more difficult to control and thus, more likely to cause defects [ITR11]. Furthermore, in nanoscale CMOS technology, transistors are so small that printing errors below the wavelength of light and variations in the discrete number of dopant atoms have major effects on their performance.
- Random failures happen during systems' working life. As transistors become smaller, as well as their supply voltage, they are more vulnerable to interference phenomena such as cosmic radiation. Furthermore, variability caused by manufacturing imperfections may also affect chips performance and reliability.
- In the last phase of systems' lifetime, wearout failures may decrease their reliability. These failures are caused by aging phenomena such as metal and oxide wearout, hot carriers injection or electromigration-related defects [GIR10].

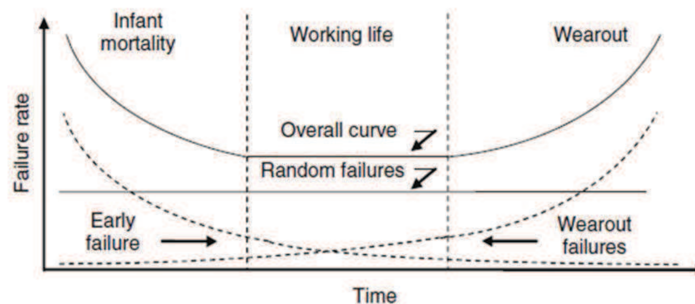


Figure 1.4 Failure Rate during Digital Systems' Lifetime, Source: [GIR10]

Reliability problems described above affect digital systems via faults and errors occurrences in ICs. In the next sub-section, we detail how each type of fault and error affects chips' operation.

1.1.2 Classification of failure mechanisms

Before analyzing further different factors responsible for failures of digital systems, let's define three important terminologies used in thesis: *failure*, *defect*, *fault* and *error* [BUS02]. As stated in the previous sub-section, *failures* are deviations of a digital system from compliance with its specification during a period of time. Failures are caused by *defects*, which are "unintended difference between the implemented hardware and its intended design". These defects can be represented at abstracted

function level as *faults*. Finally, *errors* are the manifestation of faults during system operations under the form of wrong output signals. Note that not all faults lead to errors because some of them may be masked.

To clarify these terminologies, we consider as example an adder inside a microcontroller. Suppose that due to manufacturing defects, the adder's carry output line is shorted to the supply voltage V_{dd}. Consequently, this output remains at logic-1 regardless of the adder's input operands. We say that the adder is affected by a stuck-at-1 *fault*. When this adder is used, the fault only causes an *error* when the carry line is supposed to have been at logic-0 instead of logic-1. In this case, the error may lead to a *failure* of the microprocessor.

In the previous sub-section, we have seen that there are four main reasons for failures in digital systems: manufacturing defects, variability, interference and aging phenomena. In fact, each of these factors has different impacts on system devices and therefore can create various types of fault.

Manufacturing defects

In IC manufacturing, different processes can be responsible for defects on fabricated products: implantation, etching, deposition, planarization, cleaning, lithography, etc. From the International Technology Roadmap for Semiconductors (ITRS,[ITR11]), several contaminations and mechanisms are defect causes in ICs: a) Airborne Molecular Contamination (AMC); b) process induced defects, such as scratches, cracks, particles, overlay faults and stresses; c) process variations, such as variations in doping profiles or layer thicknesses; d) deviation from design, due to pattern transfer from the mask to the wafer; and e) diffusion of atoms through layers and in the semiconductor bulk material. Different defect types in digital CMOS ICs caused by these manufacturing imperfections are illustrated in Figure 1.5.

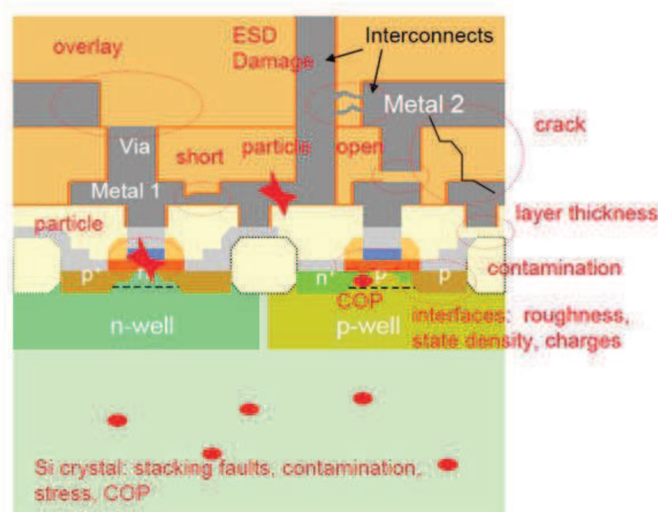


Figure 1.5 Different types of manufacturing defect, Source: [ITR11]

There are manufacturing defects that can modify the structure of digital circuits. For example, in Figure 1.5, the "short" defect creates a connection at Metal 1 level, between the gate and the drain of the PMOS transistor. This type of defect may permanently change the logic function of the circuit which then leads to failures of digital systems. Consequently, they are also responsible for manufacturing yield loss.

Meanwhile, other defects, such as variation in doping profile or deviation of channel length from design specification, do not change the logical function of devices. However, they modify functional

behavior of these components such as their switching delays. ICs containing this type of defect operate with degraded timing characteristics and may cause random failure of digital systems. This will be detailed further below.

Variability

From manufacturing processes to operating environment, there are three principal sources of variation that determine an IC's behavior: process variation, supply voltage and operating temperature. They are also known under the term Process, Voltage and Temperature (PVT) variation. Significant efforts are spent during the design phase to guarantee that digital systems operate correctly under high range of PVT variations. However, in nanoscale technologies, this goal becomes more and more challenging:

- Process variations: Due to manufacturing imperfections, transistors and interconnects in ICs are subject to variations in film thickness, lateral dimensions and doping concentrations [BER99]. For transistors, most important variations are channel length L and threshold voltage V_{th}. For interconnects, most important variations are line width/spacing, metal/dielectric thickness and contact resistance [WES10]. In digital circuits, variations of these parameters cause different timing behaviors. For example, longer channel transistors are slower while thicker metals lead to faster connections.
- Supply voltage V_{dd}: Digital systems are designed to operate at a nominal supply voltage. But during operations, this parameter may vary for different reasons, such as IR-drops along supply rails, di/dt noise and tolerances of the voltage regulator. These variations also affect timing characteristics of digital circuits. In fact, [BAK10] has proven that IC speed is roughly proportional to V_{dd}
- Operating temperature: During their lifetime, ICs may be subject to temperature variations from freezing to boiling. For example, a military IC may have to work correctly between -55°C and 125°C [WES10]. Furthermore, there are also high temperature variations inside of the IC. In [HAR01], simulation results showed that for the Intel Itanium 2 microprocessor, temperature at the execution core is higher than 100°C while memory caches in the periphery are below 70°C. It has been proven that these variations also have impacts on propagation delay of CMOS ICs [KUM06].

Although PVT variations usually do not change the logic function of ICs, they are the source of timing degradations in logic circuits, which create non-operational digital systems. Consequently, in sequential systems where timing characteristics are critical factors, these variations may also lead to errors. Therefore, these problems must be dealt with, in order to improve robustness of digital systems.

Interference phenomenon

During their lifetime, ICs continuously interact with the operating environment. They may be subject to radiation strikes, electrical noises from crosstalk or electromagnetic interferences with other running circuits in proximity. Among these interactions, radiation strikes are the most important sources of error in CMOS ICs. They create Soft errors which affect memories, registers and combinational logics of digital systems.

Soft errors are triggered when high-energy alpha particles strike an IC. These particles can be found in cosmic rays, or can also be emitted by impurities in packaging material. The mechanism of error creation is illustrated in Figure 1.6 sourced from [BAU05]. When a particle hits a silicon atom, it can induce fission, shattering the atom into charged fragments that continue traveling through the substrate. Therefore, a cylindrical track of electron-hole pairs is formed (Figure 1.6-a). When the ionization track comes close to the depletion region, the electric field rapidly collects carriers and creates a current/voltage glitch at that

node. Note that a tunnel shape extending high field depletion region deeper into substrate is formed (Figure 1.6-b). This collection phase completes within tens of picoseconds, and another phase follows, in which diffusion begins to dominate the collection process (Figure 1.6-c). Figure 1.6-d shows the corresponding current pulse resulting from these three phases.

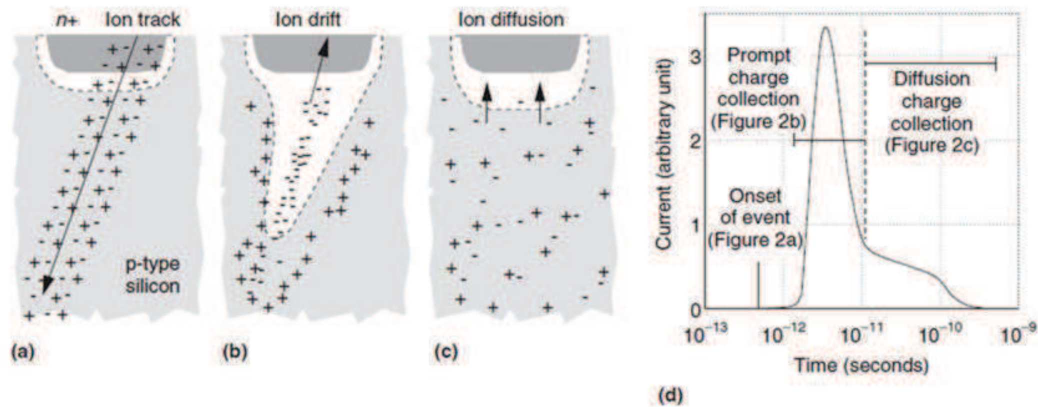


Figure 1.6 Soft Error Mechanism, Source: [BAU05]

With the mechanism described above, soft errors create voltage glitches at struck nodes. In combinational logic parts of digital systems, these glitches are called Single-Event-Transient (SET). If the glitches are captured by registers, they will change stored values in these elements (bit-flip). This behavior is called Single-Event-Upset (SEU).

Note that to completely flip state of a node, a minimum quantity of charge $Q_{critical}$ must be collected. This value depends on the capacitance and the voltage of the node. In nanoscale CMOS technology, both gate capacitances and supply voltage are downscaled. This feature explains why soft-errors have more and more impacts on digital circuit operations, and therefore must be prevented in order to improve robustness of digital systems.

Aging phenomena

Different phenomena such as hot carriers injection, temperature variations, oxide wearout and electromigration are responsible for aging of ICs' components, which may lead to defects in digital systems [GIR10]. Their two most important impacts on circuits are: oxide and interconnect wearouts.

During ICs' lifetime, gate oxides are subject to stress and gradually wear out. Consequently, the threshold voltage shift reduces the speed of transistors. As for PVT variations, this modification in timing characteristics of components may cause digital systems to fail. There are three main mechanisms responsible for oxide wearout of CMOS ICs:

- Hot Carriers Injection (HCI): As transistors switch, high-energy ("hot") carriers are occasionally injected into the gate oxide. These carriers are trapped in the oxide, and therefore change the current-voltage characteristics of the device. Note that, as electrons have higher mobility, they account for the most of the hot carriers. Consequently, the current in NMOS transistors decrease while the current in PMOS transistors increase. When the NMOS becomes too slow, the ICs may stop working correctly and cause system failures.
- Negative Bias Temperature Instability (NBTI): This problem concerns mostly PMOS transistors because they almost always operate at elevated temperature with strong negative bias (current gate voltage is at 0 while drain and source voltages are at V_{dd}). In this situation,

dangling bonds called traps develop at the Si-SiO2 interface. As traps form, the threshold voltage V_{th} increases, reducing the drive current, and making transistors slower. This phenomenon becomes one of the major causes for temporal reliability degradations in nanoscale CMOS technologies where oxide thickness is aggressively downscaling [PAU07].

- Time-Dependant Dielectric Breakout (TDDB): With an electric field applied across the gate oxide, the gate current gradually increases. After sufficient stresses, this can result in catastrophic dielectric breakdown that short-circuits the gate and cause system failures [WES10].

In digital ICs, electromigration caused by high unidirectional current flowing through wires is the main responsible for interconnect wearout. In fact, when the current density is sufficiently high, it will drift the metal ions in the direction of the electron flow. Consequently, metal atoms are displaced gradually during circuits' lifetime. Therefore, resistance of interconnect may vary, which leads to modification in timing characteristics of ICs. In extreme cases, metal wearout can also cause the formation of voids (i.e. open in the metal line) which change the logic function of circuits.

Classification

As we have discussed previously, manufacturing defects, variability, interference and aging phenomena may create different fault types in ICs. Although have different causes, these faults also have some common impacts on circuits. In the following, we classify faults into categories, depending on their duration as well as the nature of errors they may cause.

Based on duration, faults can by classified into three groups: permanent, transient and intermittent [KOR07].

- Permanent faults: As their names indicate, these faults are irreversible. Once they have occurred, none will vanish. The most common sources of permanent faults are manufacturing defects. These faults can also be caused by aging phenomena at the end of circuits' lifetime when the device starts to wear out.
- Transient faults: These are faults that cause IC components to malfunction during a short period of time. Unlike permanent faults, they disappear after that time and devices return to correct operation. Principle sources of transient faults are variability and interference phenomena.
- Intermittent faults: These faults happen now and then during ICs' operation. They never disappear completely like transient faults, but they do not occur continuously like permanent fault either. However, intermittent faults often precede the occurrence of permanent faults. Aging phenomenon is the main cause of this reliability issue.

In the scope of this thesis, we propose robustness improvement solutions for permanent and transient faults. Depending on their duration, intermittent faults can also be treated as one of these two types. For example, if an intermittent fault appears during two or more consecutive clock cycles in logic circuits, it will be considered as a permanent fault. Otherwise, solutions for transient faults will be applied.

Faults can also be classified by their resulting errors in ICs. There are three types of error: hard, soft and timing.

- Hard errors: These are permanent errors which change permanently the logic function of ICs. A typical example is an error caused by a stuck-at-fault. Normally, permanent faults are responsible for this type of error.
- Soft errors: As we have seen previously, soft errors are caused by transient faults. Depending on the place of error occurrence (in memories, combinational logics or registers), they may lead

to bit-flipping (in memories and registers) or voltage glitches (in combinational logics). SET in combinational logic part and SEU in registers are the most common soft errors observed in digital circuits.

- Timing errors: Unlike hard and soft errors, components that suffer from timing error still provide correct logic outputs. However, they have higher delays between input and output signal establishments. Transient faults induced by PVT variability, manufacturing defects and aging phenomenon are responsible for this type of error.

Table 1.1 summarizes different types of fault and error, as well as the four main reasons for these reliability issues. The first column presents different phenomena that induce faults and errors. The two other columns show possible errors created by these phenomena: the second column contains errors induced by permanent faults while the third column shows errors created by transient faults.

	Permanent	**Transient**
Manufacturing defect	Hard error	Timing error
Variability		Timing error
Interference		Soft Error
Aging phenomenon	Hard error	Timing error

Table 1.1 Faults and Errors in Digital Systems

1.1.3 Discussion

In this section, we have seen that CMOS technology evolutions allow the realization of more complex systems at lower cost and with higher performance. At each new technology node, feature sizes of transistor are downscaled allowing us to integrate more devices in one chip. Besides, these small transistors are faster, consume less power and are cheaper to manufacture. This explains why the semiconductor industry keeps scaling CMOS technology further despite the fact that reliability of digital systems has become a more and more important issue.

However, each new technology node is facing reliability problems. Different factors are responsible for transient and permanent faults in integrated circuits. These faults may induce errors and cause digital systems to fail. First of all, smaller devices are more difficult to fabricate. This leads to a higher rate of manufacturing defects and a lower manufacturing yield. Furthermore, if these defects are not detected during production test, they may cause hard errors during ICs' operation. Secondly, defect free ICs may suffer from Process-Voltage-Temperature variations which are responsible to timing errors. During their lifetime, ICs are also affected by interference phenomena. Smaller transistors are more vulnerable to radiation effects which cause soft errors in both memories and logic circuits. Finally, aging phenomena are responsible to hard and timing error at the end of circuits' lifetime.

Given the importance of CMOS technology in recent information technology revolutions, it is necessary to solve its reliability issues when emergent technologies are not ready for mass production. Fault-tolerant architectures which allow correct operation of digital systems despite the presence of faults may be a promising solution.

1.2 Fault-tolerant architectures

Previous section has shown that permanent and transient faults in ICs must be treated in order to improve the robustness of digital systems. This can be achieved by: i) improving manufacturing

processes to reduce defects and variability; ii) putting more constraints in circuit utilization, maximum operating voltage for example, to avoid aging phenomenon; or iii) redesigning digital circuits in a way that they can operate correctly despite the presence of faults.

In nanoscale CMOS, where device dimensions are of atomic size, improving manufacturing processes becomes extremely difficult. Furthermore, manufacturing improvements must be revised at each new technology node. Likewise, constraints in circuit utilization are not easy to apply and do not solve random failures. That is why in this thesis we aim to improve robustness of digital circuits and systems at the design level, using fault-tolerance methods.

This section is divided into five sub-sections. In the first sub-section, we present the fundamental of fault-tolerance. After studying the principles of this technique, we classify different types of fault-tolerant architecture into several categories, depending on which redundant resources they employ. Then, in the four remaining sub-sections, we present different fault-tolerant architectures corresponding to each type of redundancy: hardware, information, timing and hybrid.

1.2.1 Fundamentals

As we have seen in the last section, to cause failure of digital systems, faults must trigger errors in ICs. In the case where a fault exists, but does not cause any logic or timing faulty operations of circuits, we say that they are tolerated. An example of a tolerated fault is shown in Figure 1.7. In a fault free case, the circuit in Figure 1.7 provides an output $z = a + b + \overline{c}$. Suppose that due to manufacturing defects, the node *x* of the circuit is shorted to the ground. Consequently, *x* is always at logic-0 regardless of the inputs *a*, *b* and *c* (stuck-at-0 fault). However, even in this case, the output remains $z = a + b + \overline{c}$ and the fault never triggers an error.

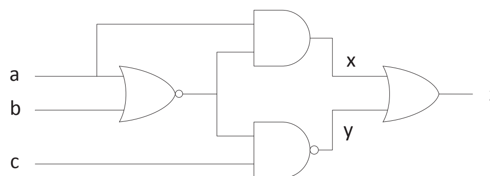


Figure 1.7 Example of a Tolerated Fault

The principle of fault-tolerance techniques is to exploit and manage redundancy to tolerate faults in circuits and systems. Redundancy is the property of having more than the minimal resource necessary to perform an operation [KOR07]. For example, in the circuit of Figure 1.7, the AND gate and the OR gate are redundant because we can remove them and use the node *y* as primary output without modifying the logic function of the circuit. Fault-tolerances are traditionally used to deal with online faults, i.e. faults that occur during the working life of ICs. However, it has been proven in [FAN06, VIA08, VIA09] that they could also tolerate manufacturing defects and thus help improving yield.

There are two ways to deal with faults: error masking and error detection/correction. In the first method, errors are masked by the redundant resources. Therefore, faults that are responsible for these errors become transparent at system level. In the second method, the tolerance process is divided in two phases: i) error detection and ii) error correction.

Redundancy is the core of fault-tolerance techniques. There are four sources of redundancy: hardware, information, timing and software [KOR07].

- Hardware redundancy: This kind of redundancy consists of integrating extra hardware into the circuit. An example is the Triple Modular Redundancy (TMR) structure where there are three identical circuits running in parallel to mask faults.

- Information redundancy: The principle of this redundancy is to generate additional information to detect and correct errors at circuit outputs. The best-known examples for this technique are error detection and correction codes.
- Timing redundancy: This redundancy attempts to tolerate faults by using additional computation time. For example, we can repeat a calculation several times and compare the results to detect errors.
- Software redundancy: Mainly used to prevent software failures, this redundancy can also be used to tolerate hardware faults in ICs. For example, let's consider two programs (software) that realize the same function. During their executions, each program uses only one part of the hardware resources. Therefore, there are hardware faults that affect only one of the two programs. Consequently, these faults can be detected by comparing results generated by the two softwares.

Beside these four types of redundancy, there exists a technique called hybrid fault-tolerance. It consists of combining different types of redundancy in the same fault-tolerant architecture in order to benefit from their advantages and overcome their drawback.

In the scope of this thesis, we do not study software redundancy which requires particular knowledge on interactions between software and hardware parts of digital systems. Different examples of fault-tolerant architecture using the other three redundancies and the hybrid technique are detailed in the following sub-sections.

1.2.2 Hardware redundancy

Hardware is the most used redundant resource in the field of fault-tolerance. Many hardware fault-tolerance techniques are employed widely in various applications, from consumer electronics to space satellites [MCH01]. In this sub-section, we present three important examples of this technique: M-of-N system, Duplex system and Neumann multiplexing architecture.

M-of-N system

An M-of-N system is composed of N modules running in parallel and a voter [SIE75]. The modules receive a common input and realize the same operation. They can either be identical or different implementations of the same logic function. The voter has the following functionality. It receives all outputs of the N modules and compares them. If there are at least M identical outputs then the voter returns the common value of these outputs. Otherwise, the system fails. Note that to guarantee a correct operation of the system, even with fault occurrences, there must not be two different sets, each account more than M identical outputs. Consequently, we need:

$$M > \frac{N}{2} \quad (1.1)$$

Due to the condition above, the function of the voter is called majority vote. Usually, N is an impair number: N=2.k-1, while M is equal to k.

A widely used M-of-N system is the Triple Modular Redundancy (TMR) architecture [LYO62]. In this particular case, N=3 and M=2. The TMR architecture is illustrated in Figure 1.8. Note that when only one module of the TMR is faulty, the voter returns the common output of other two fault-free modules. Therefore, single and multiple errors in one module are masked in this fault-tolerance technique. Furthermore, in [VIA08, VIA09], the authors have shown that a TMR architecture can also tolerate an important set of multiple faults in its three modules.

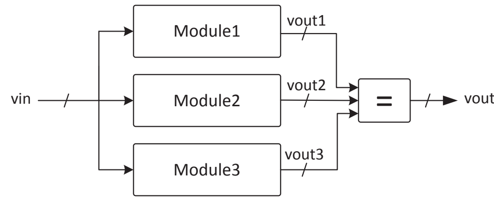


Figure 1.8 TMR Architecture

One variation of M-of-N systems is the unit-level modular redundancy architecture. Using this fault-tolerance technique, we apply replication and voting at the sub-system (unit) level [LYO62]. Figure 1.9 shows a subsystem-level TMR architecture applied for an original circuit combined of 4 units. One advantage of this architecture compared to the TMR is that the voter is no longer a critical element. In fact, in Figure 1.8, even if the three modules are fault-free, a single fault in the voter (a stuck-at-fault at one of its output bits for example) may cause failures of the entire architecture. This is no longer a problem in Figure 1.9 where there are always three voters working in parallel.

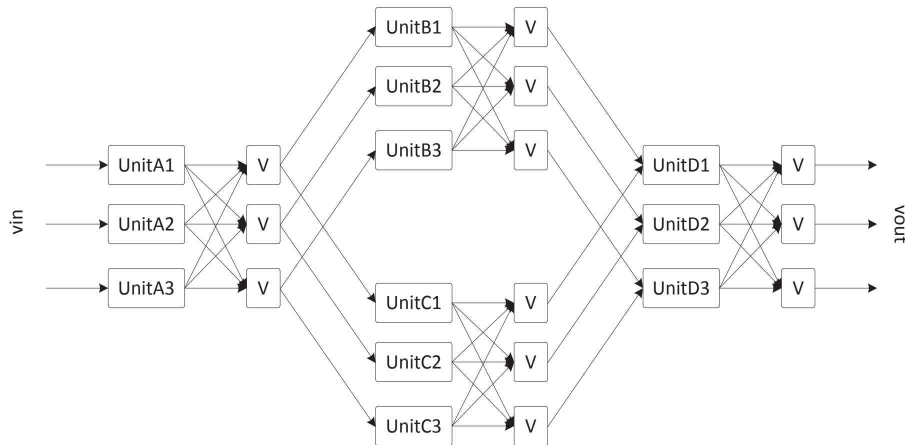


Figure 1.9 Subsystem-level TMR Architecture

Duplex system

Duplex (or Duplication/Comparison) is an error detection method widely used in fault-tolerant architectures [KOR07]. Figure 1.10 illustrates a Duplication/Comparison architecture. It consists of using two modules running in parallel and a comparator. As for M-of-N systems, the two modules can either be identical or different, but must realize the same function. The comparator compares outputs of modules in order to determine the presence of any errors.

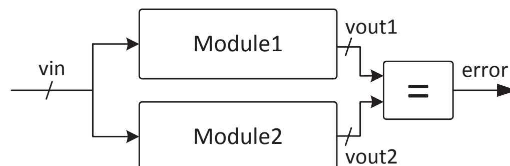


Figure 1.10 Duplication/Comparison Architecture

Compared to M-of-N systems, duplex system has the advantage of using less hardware redundancy. However, it only detects errors but does not correct them. Consequently, this technique is usually combined with other methods such as timing or information redundancy to form a complete hybrid fault-tolerant architecture [FOR09, TAH95]. This technique will be discussed more in details in the later sub-section.

NAND multiplexing system

In 1956, Von Neumann was the first person to consider using redundant components to tolerate defected devices. He proposed an architecture called NAND multiplexing (or Von Neumann multiplexing) system [NEU56].

The principle of NAND multiplexing architecture is similar to unit-level modular redundancy system. Each processing unit is replicated N_{bundle} times, forming a bundle of units. However, instead of majority voters, a bundle of wires is used to connect two successive bundles of units.

Figure 1.11 shows the example of a NAND multiplexing architecture, where the processing unit is a XOR gate and N_{bundle}=3. The architecture has two stages: Executive and Restorative. The Executive stage performs the function of the processing unit (XOR function) while the Restorative stage is used to reduce degradations caused by errors in both inputs and faulty devices. To carry out these operations:

- The Executive stage contains N_{bundle}=3 copies of the XOR gates, and a random permutation module (U). With this structure, signals of input bundles A and B are randomly paired before being connected to the processing units. Output of the Executive stage is Bundle C which carries the computation result.
- The Restorative stage is made using the same technique. Signals of Bundle C are duplicated at the beginning of this stage. After being permuted, these signals are connected to N_{bundle}=3 NAND gates. Outputs of these NAND gates form Bundle D which carries inverted value of Bundle C. The same process is used one more time to invert Bundle C, and produce output Bundle Z that carries the computation result.

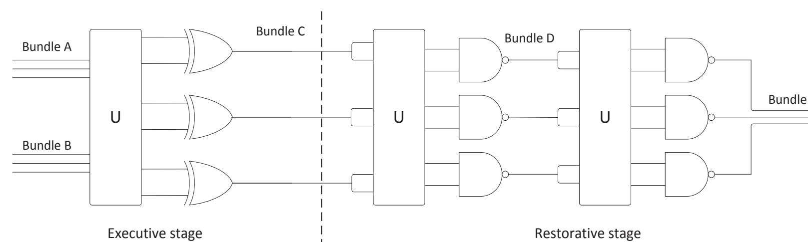


Figure 1.11 NAND Multiplexing Architecture of a XOR Unit

In [NEU56], the authors have shown that using this architecture, digital system can operate correctly even with an individual device failure rate of about 0.01. However, this solution requires enormous hardware redundancy level (about 10^3-10^4 times silicon area requirement compared to original circuits).

Pros and cons

The most important advantage of hardware fault-tolerance techniques is their fault-tolerance capability. Duplex systems can detect all single and multiple faults arriving in one of its modules while M-of-N and NAND multiplexing systems also offer the possibility to mask errors induced by these faults. Furthermore, these fault-tolerant architectures can be effective for both transient and permanent faults,

making them the most used in critical systems that require high level of reliability such as aircraft or space satellite.

Hardware redundancy is expensive in both silicon area and power consumption because it requires parallel operation of two or more copies of the circuits. That explains why in the past, these solutions were employed only by applications where the reliability is a critical factor. In advance CMOS technologies, where billions of transistors that can be put on a single chip, the area cost becomes less important, especially because techniques such as TMR can also improve manufacturing yield [VIA08, VIA09]. However, power consumption is still an important factor that needs to be improved when using hardware redundancy.

1.2.3 Information redundancy

The most common form of information redundancy is coding, which consists of inserting redundant bits (check bits) into the data inside digital systems. This redundancy can be realized under the form of extra stored bits in memories, or supplementary output signals in logic circuits. The additional information is used to verify the correctness of data at each stage, before it propagates further in the system. In some fault-tolerance techniques, the redundant information also allows correction of faulty data.

In a memory, information redundancy is implemented as additional check bits, stored with data bits. Figure 1.12-a shows an example of information fault-tolerant architecture for memories. The architecture is divided into two stages corresponding to the data writing and reading operations of the memory. During data writing, a coder calculates the check bits from the data. A code-word containing both data and check bits is then stored in the memory. During data reading, a decoder separates data and check bits from the code-word. Before providing the data bits to subsequent stages of the digital system, a checker verifies their correctness using the check bits.

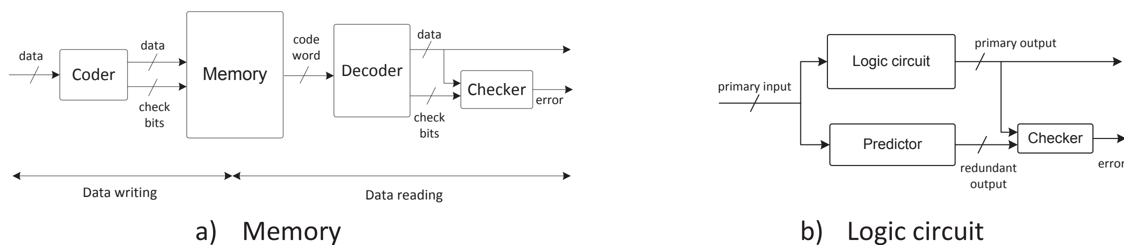


Figure 1.12 Information Fault-Tolerant Architectures for Memories and Logic Circuits

Figure 1.12-b presents a fault-tolerant architecture for logic circuits. In this architecture, redundant outputs are calculated by a predictor. This is done in parallel with the primary output computation of the logic circuit. Then, a checker compares redundant and primary outputs to validate a correct operation of the logic circuit. Note that the predictor does not calculate redundant bits from outputs of the logic circuit like a coder, but from the primary input bits. Therefore, its logic function may be much more complex than the former module.

Various types of code are used in the field of information redundancy. Among them, parity codes, arithmetic codes, Berger codes and cyclic codes are the most common uses.

Parity codes

Single-bit parity code is the simplest of all. It consists of adding one redundant bit (check bit) to the data bits. This additional bit contains the parity of the data. For example, in an even parity code, the

check bit is equal to logic-0 if and only if there is an even number of logic-1 bits in the data. In this case, the check bit can be calculated by performing a logical sum of all data bits. Then, to verify the correctness of the data, we only have to sum all bits of the code-word. The result is logic-0 if the code-word is correct and logic-1 otherwise. Note that single-bit parity code can detect all single faults, at both data bits and check bit.

Although it is very simple, single-bit parity code has a limited fault-tolerance capability. First of all, it cannot detect multiple faults. For example, if due to soft errors, two bits of the code-word stored in the memory are flipped, then the logical sum of all bits will remain intact and consequently, the error will not be detected. Secondly, even in the good case, it only helps detecting errors but not correcting them.

To overcome the first limitation, one solution is to divide the data into separate groups of bits and to use one single-parity check bit for each of them. As each group has a smaller number of bits, the probability of having multiple faults in one group is also smaller. Therefore, the error detection capability is improved. However, better error detection is achieved with a higher level of redundancy (number of check bits added). In the extreme case, one check bits is added to each data bits. For logic circuits, this fault-tolerance solution is equivalent to the duplex system presented previously.

The second limit of single-bit parity code is more difficult to resolve. To be able to correct errors, we also need multiple check bits. However, the groups of bits must not be separated. And it's the dependence between these groups that allows the correction of errors. Due to their complexity, in the field of fault-tolerance, error correction codes usually offer only detection of two erroneous bits with correction of one bit. These codes are called Single Error Correction – Double Errors Detection (SEC-DED) codes. One example is the Hsiao codes that are widely used in memories [HSI70].

Berger and Cyclic codes

With higher integration, the probability of multiple error occurrence increases significantly. This leads to the need of better detection and correction codes. For memories, Berger and cyclic codes are efficient solutions:

- Berger codes [BER61, DE94] allow the detection of unidirectional errors, i.e. errors where all detected bits are flipped in the same direction from logic-1 to logic-0 or vice-versa. Compared to single-bit parity code, they have higher fault-tolerance capability because they can detect unidirectional errors affecting multiple bits and all single-bit errors. Compared to SEC-DEC code, Berger codes do not allow error correction. Besides, they also have the disadvantage of lacking of bidirectional double error detection.
- Cyclic codes are non-separable: data and check bits are mixed together, and extracting data from the code-word requires decode operations that may induce additional delays. Despite this drawback, these codes are the most used in memory fault-tolerance because they allow detection and correction of multiple bits with low information redundancy [KOR07].

Arithmetic codes

While parity, Berger and cyclic codes are the most used in memories, arithmetic codes are better suited to logic circuits. They employ arithmetic properties that are preserved under a set of arithmetic operations, such as addition or multiplication, to detect errors. Their advantage relies on the simple implementation of the predictor module (Figure 1.12-a) when used for arithmetic circuits [RAO70, RAO77, TAH95, FOR09]. However, these solutions are not applicable for random logic circuits.

Pros and cons

Compared to hardware redundancy, information redundancy has the advantage of using much less additional resources. Parity, Berger and cyclic codes are adapted for fault-tolerance in memories. In

these cases, the number of check bits is significantly smaller than the number of data bits while coders, decoders and checkers can be implemented easily using small number of simple logic gates. Furthermore, advanced parity and cyclic codes also allow correction of single or multiple defected bits. However, implementation of these techniques for logic circuits is more complex, especially in the generation of redundant outputs from primary inputs. For these circuits, arithmetic codes are more suitable. Nevertheless, these codes can only be used for arithmetic operations such as addition or multiplication.

1.2.4 Timing redundancy

The principle of timing redundancy is, when an error is detected, to repeat the faulty operation several times before delivering the final good result. Computations are performed by the same processing units and their results are stored in different registers. These values are then compared in order to detect and correct errors.

Figure 1.13-a shows an example of using timing redundancy to detect errors at logic circuits. In this architecture, each operation is repeated once by the processing unit Module. A control module and a demultiplexer are used to store the results in two output registers Reg1 and Reg2. These outputs are compared at the end of the calculation, using a comparator. If a transient error occurs at one of the two computations then it will be detected by this comparison. The functionality of this architecture is similar to the duplex system presented in Figure 1.10, but it only requires one instead of two modules. Hence, it allows significant reduction of area overhead and power consumption. However, the timing fault-tolerant architecture can only detect transient errors. In fact, if Module suffers from a permanent fault then both computation results stored in Reg1 and Reg2 will be affected the same way. Consequently, the errors will not be detected by the comparator.

Error detection can also be added to the architecture above by doing more than two computations for each input data and then using a majority voter to mask transient errors [HSU94, GAL98]. Figure 1.13-b shows an example where each operation is repeated twice. This architecture is called Time shared Triple Modular Redundancy (TTMR) because it has similar functions compared to TMR architecture. Note that like the structure in Figure 1.13-a, although it offers significant cost reductions compared to TMR architecture, TTMR can only tolerate transient faults.

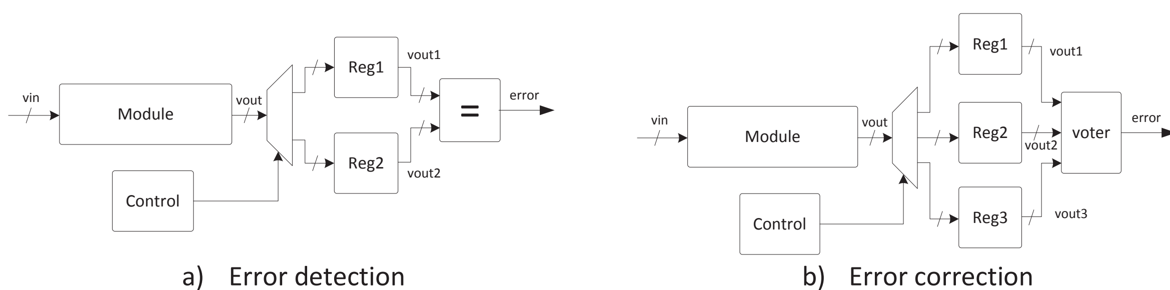


Figure 1.13 Timing Fault-Tolerant Architectures

Compared to two other types of redundancy, timing redundancy has smaller area and power consumption costs. However, they have significant delay costs because every calculation is repeated several times, even in fault-free cases. Furthermore, as each operation is repeated, their total energy consumption also increases several times.

1.2.5 Hybrid fault-tolerance

As we have seen in previous sub-sections, each type of redundancy has different advantages and disadvantages regarding their fault-tolerance capability, silicon area and power consumption. The principle of hybrid fault-tolerance technique is to combine these redundancies to complement their benefits and costs.

Information and Timing

One important drawback of timing redundancy is the lack of permanent fault detection capability. Meanwhile, information redundancy offers low cost detection for both permanent and transient faults, but it is not adapted for logic circuits. Combining information and timing redundancies allows to overcome these issues.

Figure 1.14 shows an example where information and timing redundancies are combined to detect transient and permanent faults at logic circuits. Compared to the timing fault-tolerant architecture in Figure 1.13-a, this architecture does not re-compute the primary input data X but its encoded value C(X), where C is the logic function of the coder. Both data X and C(X) are selected for computations by an additional multiplexer. Then, the computation result of C(X) is decoded before being compared with the computation result of X. The processing unit Module is proven fault-free if no difference is detected when comparing the results.

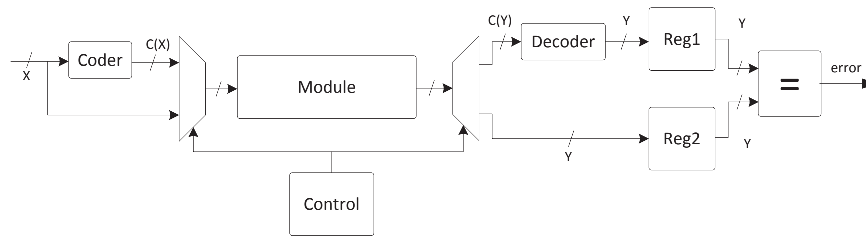


Figure 1.14 Information-Timing Hybrid Fault-Tolerant Architecture for Error Detection

Note that this simple architecture works under the hypothesis that the computation result of C(X) is C(Y) where Y represents the computation result of X. If we call F the logic function of Module then we need:

$$F\big(C(X)\big) = C(Y) = C(F(X)) \qquad (1.2)$$

One concrete application of the architecture in Figure 1.14 is error detection in alternating logic circuits [RAY75]. They are circuits that satisfy $F\big(\overline{X}\big) = \overline{F(X)}$. For example, a circuit receiving three input bits a, b, c, which calculates $\overline{\mathrm{a}}\oplus\overline{\mathrm{b}}\oplus\overline{\mathrm{c}}$, is an alternating circuit because $\overline{\mathrm{a}}\oplus\overline{\mathrm{b}}\oplus\overline{\mathrm{c}} = \overline{\mathrm{a}\oplus\mathrm{b}\oplus\mathrm{c}}$. For this type of logic circuit, we can use complimentary code $C(X) = \overline{X}$ to fulfill equation (1.2). Consequently, both coder and decoder can be easily implemented using inverters. Besides, this architecture has also been proven to be efficient for stuck-at-fault detection [RAY75].

Although being efficient for both transient and permanent faults detection, the architecture in Figure 1.14 does not offer error correction. For this purpose, Figure 1.15 proposes another way to combine information and timing redundancies.

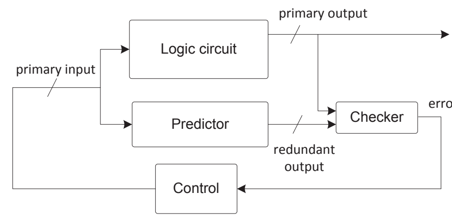
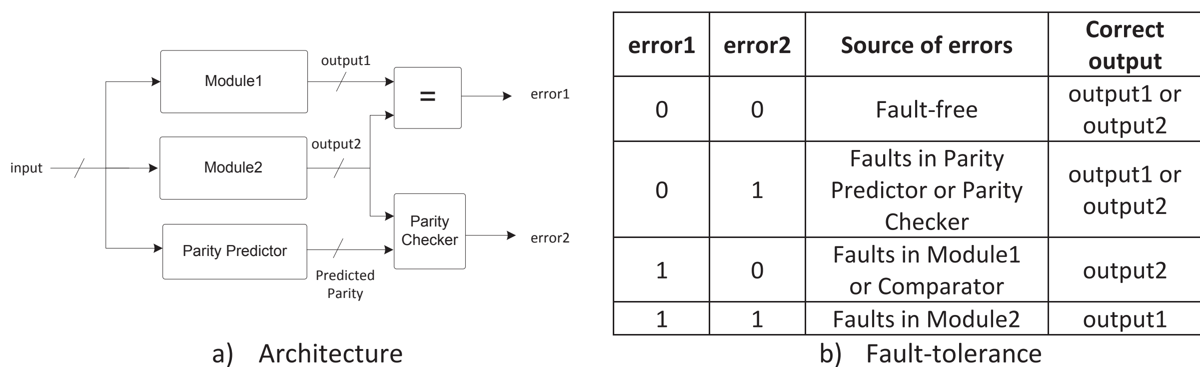


Figure 1.15 Information-Timing Hybrid Fault-Tolerant Architecture for Error Correction

Compared to the information fault-tolerant architecture shown in Figure 1.12-b, the new architecture employs an additional control module to manage re-computation by re-injecting affected primary input data in case an error is detected by the checker. As error correction is done by timing redundancy, we can use simple error detection codes as information redundancy. Hence, this method helps reducing area overhead thanks to the complexity of the predictor.

Information and Hardware

Although it offers efficient error correction capability, the principal drawback of hardware redundancy is its high costs in area and power consumption. To improve this, we can combine information and hardware redundancies. Figure 1.16 shows an example for such architecture [ALM03].



error1	error2	Source of errors	Correct output
0	0	Fault-free	output1 or output2
0	1	Faults in Parity Predictor or Parity Checker	output1 or output2
1	0	Faults in Module1 or Comparator	output2
1	1	Faults in Module2	output1

b) Fault-tolerance

Figure 1.16 Hybrid Architecture Combining Duplication/Comparison and Parity Codes

The architecture in Figure 1.16-a combines duplication/comparison method (hardware redundancy, Figure 1.10) and parity codes (information redundancy) to detect and correct errors in logic circuits. The processing unit is duplicated (Module1 and Module2). A comparator detects mismatches between their outputs (*output1* and *output2*). If Module1, Module2 and the comparator are fault-free then signal *error1* is at logic-0. If one and only one of these modules is affected by faults then *error1* turns to logic-1. The correctness of *output2* is also verified by a parity predictor (Parity Predictor) and a checker (Parity Checker). If these two modules are fault-free and *output2* is correct then *error2* is at logic-0. If there are faults in one of the modules Module2, Parity Predictor or Parity Checker then error2 switches to logic-1.

With the architecture above, all single and multiple faults affecting any single module are detected and tolerated. The table in Figure 1.16-b shows how this is done: the first two columns present values of *error1* and *error2*; the third column shows which module can be affected given the value of the error signals; the last column indicates which output is correct and should be used as primary output of the architecture.

The advantage of the presented architecture is that for some particular logic circuits, the parity predictor may have smaller silicon area than the processing units. Consequently, the complete architecture may have smaller area overhead than that of TMR architecture.

Note that in the example above, parity codes are used as information redundancy. However, this solution can also be applicable for other error detection codes such as arithmetic codes, Berger codes, etc. Depending on the type of logic circuit, one type of code can be more suitable than the others if it allows simpler implementations of the predictor and the checker. In some cases where the predictor is much smaller than the processing units, the resulting fault-tolerant architecture has smaller area overhead compared to TMR architecture while offering comparable error detection/correction capability.

1.2.6 Discussion

In this section, we have studied the principle of fault-tolerance technique. Employing different redundant resources allows digital circuits to operate correctly despite the presence of permanent and/or transient faults. This is done by masking or detecting/correcting errors induced by faults.

We have also classified fault-tolerant architectures in four categories depending on which redundancies are used. Hardware fault-tolerant architectures use one or more copies of the original circuit to detect or mask errors. Although providing efficient fault-tolerance for both transient and permanent faults, this solution requires considerably high costs in silicon area and in power consumption. Information fault-tolerant architectures employ codes to detect and correct errors created by faults. This method is efficient for memories thanks to their regular structure. For logic circuits, information redundancy may lead to high area overhead due to complex implementations of code predictors. Timing fault-tolerant techniques consist of using re-computation to detect and correct transient errors. Although having small area overhead, this method results in high calculation delays because both fault-free and faulty operations are repeated several times. To enhance the advantages and overcome the drawbacks of the three techniques above, hybrid fault-tolerance methods employ different redundancies at the same time to deal with faults. These techniques are efficient for logic circuits and can be used to tolerate both permanent and transient faults while optimizing area overhead and power consumption.

1.3 Robustness improvement of digital systems

In previous sections, we have seen how faults and errors are responsible for failures in digital systems. We have also studied how fault-tolerance techniques can be used to deal with these issues and thus, improve digital system robustness. In this section, we detail how fault-tolerant architectures are used to protect different parts of digital systems from faults and errors.

The section is divided in two parts. In the first sub-section, we study fault-tolerance in memories while in the second sub-section, we investigate robustness improvement in logic circuits.

1.3.1 Fault-tolerance in memories

In 2003, the Semiconductor Industry Association (SIA) reported that memory cores presented more than 70% silicon area of a digital system and that this ratio would increase to more than 90% after 2012 ([ITR03], Figure 1.17). In 2011, they predicted that this trend will continue in the future [ITR11]. This explains why reliability of memory is a very important factor in the semiconductor industry.

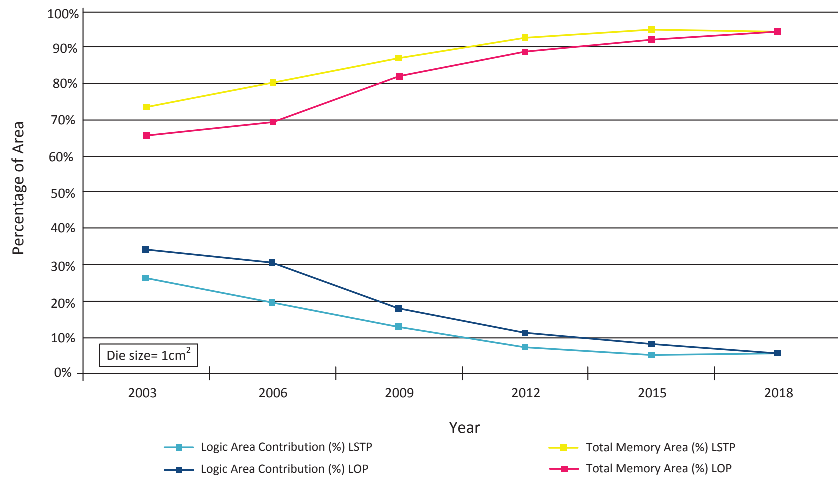


Figure 1.17 Logic/Memory Composition of System-on-Chip, Source: [ITR03]

Robustness of memory circuits are affected the most by two types of error: soft errors caused by interference and radiation phenomena, permanent errors caused by manufacturing defects and aging phenomenon [SU05].

Soft errors

Table 1.2 shows Soft Error Rate (SER) in memories for different CMOS technology nodes. This rate is presented in number of failures in one million hours per megabit (FIT/Mb). It is shown that SER increases as the technology advances. In this table, we can also see that the percentage of Multiple Bits Upsets (MBU) on total SER increases very fast and will reach 100% in 2016. These trends explain why today, error detection/correction codes are systematically used in memories.

Year	2007	2008	2009	2010	2013	2016	2019	2022	2024
Technology node (nm)	65	55	50	45	35	25	18	13	10
SER (FIT/Mb)	1150	1150	1150	1200	1250	1300	1350	1400	1450
Percentage of MBU on total SER	16%	16%	16%	32%	64%	100%	100%	100%	100%

Table 1.2 Soft Error Rate in Embedded Memory, Source [ITR11]

Permanent error

Although information redundancy can tolerate both soft and permanent errors, the effectiveness of this method decreases during systems' lifetime. In fact, as memories age, the number of permanently defected bits increases. If this number exceeds the detection capability of codes then failures will occur. The most used solution for this problem is hardware redundancy.

Hardware fault-tolerance in memories is implemented under the form of spare memory cells, words and columns [SCH01, NIC03, NIC05, SU05]. Traditionally, memory repair is performed right after the fabrication phase. External test equipments are used to detect and localize faults. Then defected elements (memory cells, words and columns) are replaced by redundant resources using different techniques such as laser beams, electrical fuses or anti-fuses. For embedded memory, external test and repair have been replaced by Built-In Self-Test (BIST) and Built-In Self-Repair (BISR) [NIC05, SU05]. Additional modules are implemented so that the memory circuit can perform test and repair itself. Note

that, BIST/BISR can be run anytime during the lifetime of memories to tolerate both manufacturing defects and faults created by aging phenomena.

1.3.2 Fault-tolerance in logic circuits

In general, a logic circuit is composed of combinational and sequential parts. At each moment, outputs of the combinational part depend only on its present inputs, while outputs of the sequential part also depend on its previous inputs. In other words, the sequential part has memory while the combinational part does not. Besides, operations of the sequential part are synchronized by clock signals. Consequently, the logic circuit is subject to two timing constraints:

- Setup time: Inputs of the sequential part must be established an amount of time (called setup time) before the clock events.
- Hold time: Inputs of the sequential part must be held steady during an amount of time (called hold time) after the clock events.

There are different types of logic circuits, depending on how the combinational and the sequential parts are connected. In the scope of this thesis, we study logic circuits with the architecture presented in Figure 1.18. In this architecture, the combinational part consists of a Combinational Logic module (CL) while the sequential part is made of an input register (Reg_in) and an output register (Reg_out). Primary input *PI* of the logic circuits is captured by Reg_in. Then, output *vin* of this register is used to feed CL. Finally, Reg_out captures output *vout* of CL to provide primary output *PO*. The registers are synchronized by a clock signal *CLK*. Note that registers can be made by different elements such as flip-flops or latches. In this thesis, we consider only edge sensitive designs which employ D flip-flops.

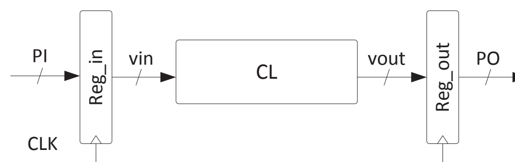


Figure 1.18 Logic Circuit Architecture

There are three types of error that can affect the logic circuit in Figure 1.18:

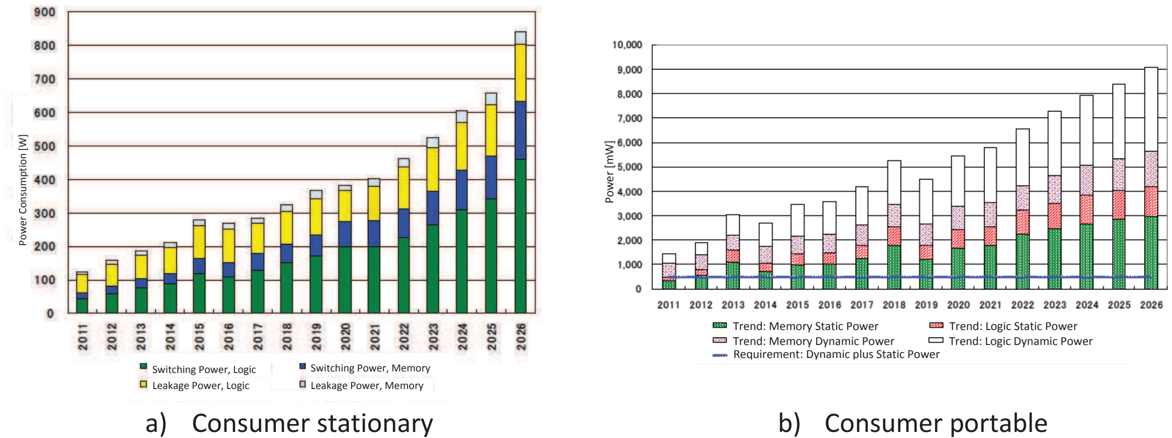
- Hard errors: These errors may modify logic functions of both combinational and sequential parts, and change the logic value of *vin*, *vout* and *PO*.
- Soft errors: Two types of soft error may cause incorrect operations of the logic circuit. The first type is SEU which modifies the logic values (bit flipping) stored in the registers. The second type is SET which may induce glitches at output *vout* of CL and thus, responsible for timing violations (setup and hold times of Reg_out).
- Timing errors: These errors cause additional calculation delays in CL which may also lead to timing violation in Reg_out.

In the following parts, we detail different solutions proposed in the literature to protect registers and the combinational logic from the mentioned errors.

Hard error protection

As we have seen in the previous section, TMR architecture is an efficient fault-tolerance solution for hard errors. However, this method usually leads to high area and power consumption overhead (more than 200%). To overcome this problem, one solution is to triplicate only parts of the logic circuit that are more vulnerable to hard errors [DAS09].

Although occupy small silicon area compared to memories, logic circuits majorly contribute to power consumption in digital systems. Figure 1.19 sourced from [ITR11] shows that this trend will continue in the future, especially for consumer stationary applications. Consequently, power consumption will be an important drawback of TMR architecture in advance technology nodes.



a) Consumer stationary b) Consumer portable

Figure 1.19 Power Consumption Trends of System-on-Chip

SEU protection

To protect registers from SEU, TMR architecture is also an effective solution. [WAN03] proposes to use subsystem-level TMR (Figure 1.9) to harden D flip-flops of the registers. These flip-flops are made of master and slave latches. Each latch is hardened in an asynchronous manner (Figure 1.20). Although effective for both hard errors and SEU, this method has an area overhead of about 400%.

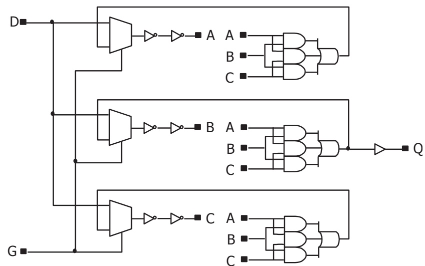


Figure 1.20 Latch Hardening Using TMR, Source: [WAN03]

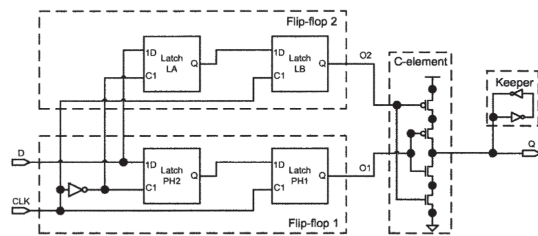


Figure 1.21 Flip-flop Hardening Using C-element, Source: [ZHA06]

To overcome the high costs of TMR, [ZHA06] proposes BISER (Built-In Soft Error Resilience) architecture using C-elements to tolerate SEU in D flip-flops. In this solution (Figure 1.21), the hardened flip-flop is composed of two D flip-flops, a C-element and a keeper. As shown in its truth table (Table 1.3), the C-element acts like an inverter when its inputs *O1* and *O2* are identical. If due to a SEU, one input bit is flipped, then output *Q* remains at its previous value retained by the keeper. Consequently, the soft error is tolerated.

O1	O2	Q
0	0	1
1	1	0
0	1	Previous value retained
1	0	Previous value retained

Table 1.3 Truth Table of a C-element

Compared to TMR solution, solution in Figure 1.21 has lower silicon area and power consumption costs. However, it does not allow the correction of hard errors because if *O1* or *O2* suffers from stuck-at-faults then the value of Q will never change.

SET and timing error protection

Although caused by different phenomena, SET and timing errors have the same effect on operations of logic circuits: outputs of the combinational logic are not stable, violating setup and hold time of the output register. Consequently, to detect these errors, one solution is to discover glitches or invalid transitions that occur at inputs of the register.

In [ERN03], the authors propose RAZOR flip-flop architecture (Figure 1.22), which allows the detection of SET and timing errors. Each RAZOR flip-flop is augmented with a shadow latch controlled by a delayed clock (*clk_del*). This delayed capture assures that the shadow latch stores a correct value even if SET or timing errors occur during the main flip-flop capture (at *clk* event). Outputs of the main flip-flop and the shadow latch are compared to detect errors. In case an error is detected, a multiplexer is used to replace the faulty value of the main flip-flop by the correct value stored in the shadow latch.

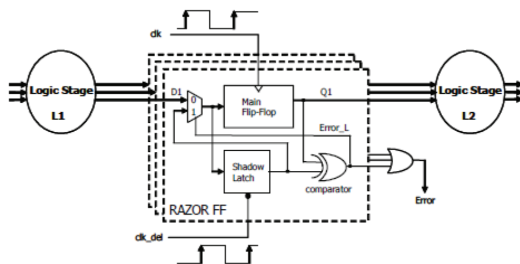


Figure 1.22 RAZOR Flip-flop, Source: [ERN03]

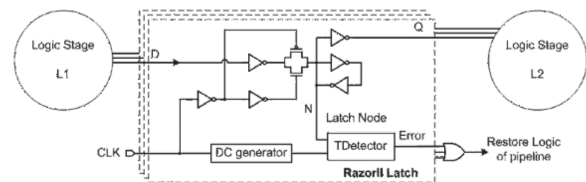


Figure 1.23 RAZOR II Latch, Source: [DAS09]

Figure 1.23 illustrates another technique proposed in [DAS09] which employs RAZOR II latches to detect errors. In this architecture, between two consecutive computations, outputs of the combinational logic L1 are supposed to be stable during a timing window (the detection window). Spurious transitions caused by SET and timing errors during this window may be discovered by a transition detector (TDetector). To assure that TDetector reports only invalid transitions, a digital clock generator (DC generator) is used to define the detection window.

1.3.3 Discussion

In memory part of digital systems, the use of fault-tolerant architectures has been proven necessary and efficient. Information (error detection/correction) and hardware (spare memory words, columns and lines) redundancies are employed to deal with both transient and permanent faults. However, fault-tolerance in logic circuits of digital systems remains a challenge for future technology nodes. Different

fault-tolerant architectures have been proposed but none of them are effective for both transient (SEU, SET, timing errors) and permanent (hard errors) faults in logic circuits:

- TMR architecture may be used to tolerate both permanent faults in combinational part and SEU in sequential part of logic circuits. However, this solution is vulnerable to timing errors which may arrive at inputs of the combinational logic.
- BISER structure allows SEU tolerance in registers with low area overhead than TMR. The tradeoff is that this structure cannot tolerate permanent errors in logic circuits.
- RAZOR and RAZOR II architectures deal with both SEU in registers and SET/timing errors in combinational logics. However, these solutions only detect problems. Errors correction is done by re-computation. Furthermore, like BISER, both architectures are vulnerable to permanent errors.

Beside fault-tolerance capability, power consumption of logic circuits is a rising issue in advanced technologies. As these circuits contribute the major consuming parts of digital systems, limiting their power budget is one of the key factors in digital design. However, existing fault-tolerant architectures are only optimized in term of area overhead.

1.4 Summary

In this chapter, we have discussed the importance of CMOS technology evolutions that have allowed the realization of more complex systems at lower costs and with higher performance. We have also seen that advanced technology nodes are facing reliability issues. Manufacturing defects, variability, interference and aging phenomena induce more and more transient and permanent faults which cause digital systems to fail. In order to continue taking advantage of new CMOS technology nodes, we must improve robustness of digital systems by dealing with hard, soft and timing errors.

To solve reliability problem of digital systems, one solution is to use fault-tolerant architectures. These architectures employ redundant resources to guarantee correct operation of circuits despite the presence of faults. There are three type of redundancy: hardware, information and timing redundancy. Each type fault-tolerance method has different pros and cons with regards to errors. Hardware fault-tolerance is efficient for both transient and permanent errors, but they often require high area overhead and power consumption. Information fault-tolerance requires less redundant resources but is only adapted for memories and arithmetic circuits. Timing fault-tolerance is the most cost effective solution in term of area overhead but it increases significantly IC delays and can only tolerate transient errors. To improve robustness of digital circuits and systems, the three types of redundancy can be employed together to compromise their pros and cons. This technique is called hybrid fault-tolerance.

While information and hardware redundancy provide efficient robustness improvement for memories, fault-tolerance in logic circuits remains a challenge for future technology nodes. Different fault-tolerant architectures have been proposed, but none of them are effective for both transient (SEU, SET, timing errors) and permanent (hard errors) faults in logic circuits. Besides, these techniques are only optimized in term of area overhead whilst power consumption of logic circuits is becoming a more and more important factor.

In the rest of this thesis, we study the use of hybrid fault-tolerance techniques for robustness improvement of logic circuits. Our objective is to tolerate both transient and permanent errors in these circuits. Besides, both silicon area and power consumption of the solutions are subject to optimizations. Finally, we aim to provide a "plug and play" solution which can be applied without modification in the implementation of the logic circuits.

Chapter 2

The Hybrid Fault-Tolerant Architecture

Chapter 2 The Hybrid Fault-Tolerant Architecture 29
2.1 Principles of hybrid fault-tolerance 30
2.2 Error detection 31
2.2.1 Concurrent Error Detection 31
2.2.2 Parity codes 32
2.2.3 Duplication/Comparison 33
2.2.4 Conclusion 39
2.3 Transient error correction 40
2.3.1 Input register 40
2.3.2 Reset signal 42
2.3.3 Transient error correction mechanism 42
2.3.4 Control logic and timing constraints 44
2.3.5 Conclusion 47
2.4 Permanent error correction 47
2.4.1 Input de-multiplexer 48
2.4.2 Output multiplexer 49
2.4.3 Reconfiguration finite state machine 51
2.4.4 Control logic and timing constraints 53
2.5 Summary 55

We have seen in the previous chapter that different types of fault, transient and permanent, affect normal operations of logic circuits. Consequently, redundancy resources are added to tolerate these faults and thus, improve circuits' robustness. Each type of redundancy is suitable for some particular fault categories. In this chapter, we propose a new architecture, which combines information, timing and hardware redundancies to tolerate both transient and permanent faults of digital systems. This solution, called hybrid fault-tolerant architecture, will be able to detect and correct hard, soft and timing errors which occur in combinational part of logic circuits.

The chapter is organized as follows: In the first section, we present principles of the hybrid fault-tolerance solution. It is divided into three phases: error detection using information redundancy, transient error correction using timing redundancy and permanent error correction using hardware redundancy. Then, in the three following sections we study in detail these phases. For each phase, we propose a complete fault-tolerant architecture with logic implementations of additional modules as well as their control logic and timing constraints.

2.1 Principles of hybrid fault-tolerance

Our objective is to tolerate transient and permanent fault in logic circuits which consist of combinational logic (CL) and input/output registers (Figure 1.18). As described in 1.3.2, there are different efficient methods to protect registers from hard and soft errors. Therefore, in this chapter, we consider only hard, soft and timing errors, which may occur in CL part of the circuits. The integration of SEU protection into the proposed architecture will be studied further in Chapter 4.

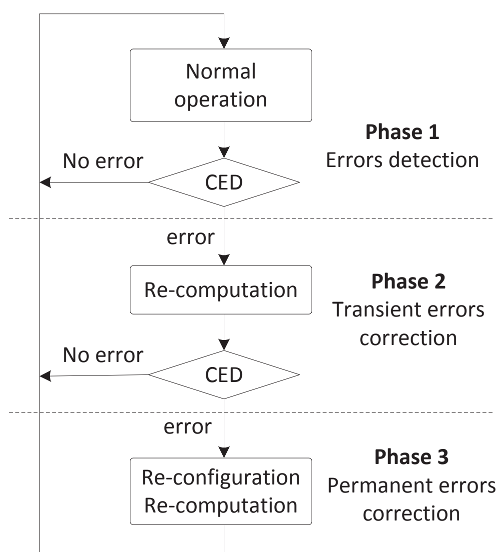


Figure 2.1 Principles of Hybrid Fault-tolerance

The hybrid fault-tolerant architecture uses error detection/correction method to deal with faults. The fault-tolerance operation is divided into three phases (Figure 2.1). In the first phase, we use Concurrent Error Detection (CED) techniques to detect errors, regardless of their nature. In a fault-free case, the architecture continues to operate normally in Phase 1. If errors are detected, the second phase will be activated. The architecture will re-compute the affected input vector in order to tolerate transient faults. If errors disappear after the re-computation, the architecture will return to its normal operation. If errors remain, the architecture will then enter the third phase. A re-configuration will replace hardware which

may contain permanent faults by fault-free resources. A re-computation will then allow the architecture to come back to normal operation.

In the following sections, we construct the complete hybrid fault-tolerant architecture through three stages. At each stage, we integrate one of the three phases presented above to obtain:

- An error detection architecture capable of detecting transient and permanent errors.
- A transient error correction architecture that detects both type of error and tolerates transient errors.
- A permanent error correction architecture, *i.e.* the complete hybrid fault-tolerant architecture, which detects and tolerates both transient and permanent errors.

2.2 Error detection

2.2.1 Concurrent Error Detection

Concurrent Error Detection (CED) methods are widely used to enhance logic circuits' reliability [MIT00a] by continuously checking for errors during its operation. Their principle is illustrated in Figure 2.2:

- From an input vector *vin*, the circuit CL must realize a function f and produce an output vector *vout*:

$$vout = f(vin) \tag{2.1}$$

- The CED employs an additional Predictor module to predict some particular characteristics *C* of a fault-free *vout*. Let's call the Predictor's logic function *P* and its output vector *CB*. We will have:

$$CB = P(vin) = C(f(vin)) \tag{2.2}$$

  or:

$$P = C \circ f \tag{2.3}$$

- Finally, a Checker will make sure that the output *vout* has the characteristics predicted, which means verifying that:

$$C(vout) = P(vin) \tag{2.4}$$

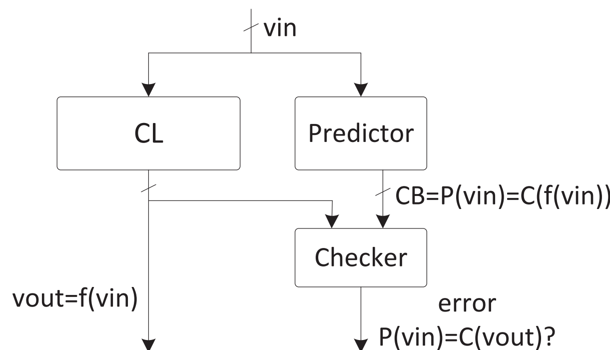


Figure 2.2 Concurrent Error Detection Scheme

Depending on the characteristic function *C,* we have different CED techniques [MIT00a], such as parity codes [DE94, TOU97, NIC97], duplication/comparison [SEL68], Berger codes [BER61, DE94], Bose-Lin codes [BOS85], or arithmetic codes [FOR09, TAH95, RAO77]. Berger and Bose-Lin codes can only

detect unidirectional errors while the use of arithmetic codes requires sufficient knowledge of the logic function *f*. As our architectures target general logic circuits, we will only consider parity codes and Duplication/Comparison scheme.

2.2.2 Parity codes

Predictor synthesis

As we have seen in 1.2.3, single-bit parity code is the simplest way to detect errors in memories because its implementation only requires an additional bit (code-word) for a data word. Even with more advanced techniques, such as Hsiao codes [HSI70], the cost remains relatively low. For example, 10-bit Hsiao codes are able to detect double-errors in 120-bit data word. In order to use parity codes for logic circuits, we also use additional code bits for their output vector. The generation of these parity bits by a predictor must be performed independently to the logic circuits so that a single fault can not affect both the output and the parity code-words.

In [KO04], the authors suggested a predictor synthesis using AND/XOR expressions and Davio's expansion theorem. But the proposed method targets only FPGA implementations. Another method is to use logic synthesis tools to build parity predictors. This synthesis flow is described in Figure 2.3. For a combinational logic circuit CL, the predictor is obtained by adding a logic structure XORT which can calculate parity check bits *CB* from circuit outputs *vout*. The simplest way is to use XOR-trees to realize XORT. Then, a logic synthesis tool is run to generate the gate level description of parity predictors.

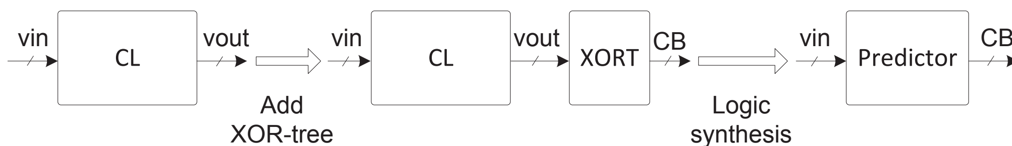


Figure 2.3 Parity Predictor Synthesis Flow

Area overhead

Circuit	Single-bit parity code	Hsiao code
c17	176%	na
c432	113%	133%
c499	60%	149%
c880	116%	151%
c1355	124%	148%
c1908	114%	143%
c2670	119%	156%
c3540	101%	114%
c6288	103%	109%
c7552	113%	139%

Table 2.1 Parity Predictors' Area

We used the synthesis flow described above to build predictors for single-bit parity and Hsiao codes (with the minimum number of check bits). The targeted circuits are from ISCAS'85 benchmark [ISCAS85]. The logic synthesis is performed by Synopsys Design Compiler [DCSYN] using a 90nm technology from

STMicroelectronics. Results are presented in Table 2.1, where the predictors' area ("Single-bit parity code" and "Hsiao code") is presented in percentage of the original circuit. We note that in most case, the predictors are larger than the original logic circuits.

Common-mode failures

Parity codes are vulnerable to common-mode faults (CMF), i.e. faults that may cause more than one erroneous bits. For example, single-bit parity code can only detect an impair number of CL output faulty bits. In [MIT00a, TOU97], the authors proposed a CED systems based on parity codes that deal with CMFs. Although the proposed techniques promise small parity predictors, they require a re-synthesis of the combinational logic CL into separate logic cones which increase its total area significantly. Furthermore, these techniques are not applicable because we target a "plug and play" fault-tolerance method.

Discussion

We have seen in this sub-section that parity predictors have similar silicon area compared to original logic circuits. Thus, using parity code in CED leads to an area overhead equivalent to a Duplication/Comparison scheme. Besides, we will see in Chapter 3 that the Duplication/Comparison technique allows better fault detection of CMFs. Therefore, in our hybrid fault-tolerant architecture, we will use this technique for errors detection.

2.2.3 Duplication/Comparison

Error detection scheme

Figure 2.4 illustrates our error detection method using the Duplication/Comparison technique detailed in 1.2.2. Our targets are logic circuits composed of a combinational logic CL and input/output registers *Reg_in/Reg_out* (Figure 1.18). Compared to Figure 1.18, the combinational logic CL is duplicated (CL1 and CL2). These two copies realize the same logic function as CL but they can be implemented differently. Both CLs are fed by output *vin* of the input register. However, only output *vout1* of CL1 will feed the output register. Output *vout2* of CL2 is instead used to validate data integrity of the complete architecture. This validation is performed by comparing the primary output *PO* with *vout2* during a comparison window. If the two vectors matches up during this time window, *error* stay at logic-0 indicating "No error detected". If they are different, *error* will switch to logic-1 alerting the problem.

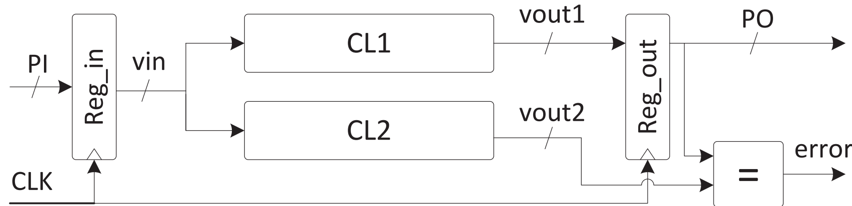


Figure 2.4 Duplication/Comparison Scheme for Logic Circuits

Note that the comparison is placed after data capture of the output register. In this configuration, error detection process will finish after *CLK* rising edge when the next computation has be launched. Therefore re-computation phase (details in 2.3) will take two instead of one *CLK* cycles: one to calculate new input and one to repeat the affected input.

An alternative solution is to finish the comparison before the data capture (by comparing *vout2* with *vout1* instead of *PO*). At *CLK* rising edge, either new computation will be launched if no error is detected or re-computation is activated otherwise. Although only one CLK cycle is taken for transient error correction, this solution requires a longer *CLK* period, even for fault-free operation, because delay time of the comparator must be taken into account with CLs calculation time.

Supposing that fault-free operations happen most of the time, we will choose to keep the architecture in Figure 2.4 which operates at the same *CLK* period as the original logic circuit (Figure 1.18).

The *error* signal

The comparator itself consists of two stages: a local (bit to bit) comparison and a global comparison which accumulates all local comparisons into a one-bit-signal *error*. The basic structure of a comparator is presented in Figure 2.5. In the first stage, XOR gates are used to realize *n* bit-to-bit comparisons between *vout1* and *vout2*. Then, the *n* signals *Ci* (i=1..*n*) are combined in the second stage by an OR-tree, in order to provide the *error* signal. The presented circuit is called static comparator because it is made of static CMOS gates.

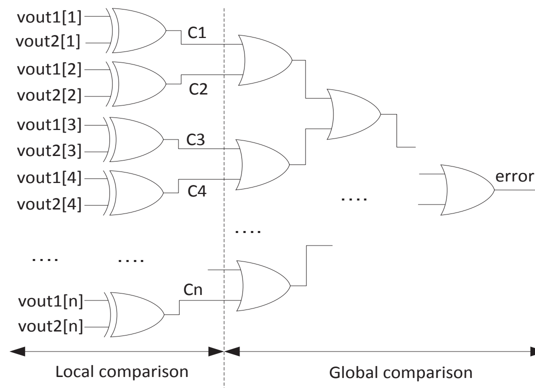


Figure 2.5 Static Comparator Structure

While combinational logics CL1 and CL2 realize the same logic functions, their timing characteristics often differ. In fact, CL1 and CL2 might be structurally different due to area/power optimizations. Even if they are structurally identical, both circuits are likely to be affected by intra-die variations. The different timing characteristics of the CLs conduct to differences between *vout1* and *vout2* during the computation time t_{CL} of the CLs. Consequently, *error* signal is not stable during that period. In addition, the comparator itself has a propagation delay t_{COMP} during which glitches may occur at the *error* signal. Therefore, *error* must be used only during a stable period called "comparison window", after t_{CL} + t_{COMP} and before the next computation.

In a fault-free context, *error* is constantly at logic-0 during the comparison window. If a hard error occurs, the problem is detectable since *error* will change to a logic-1. For soft and timing errors, the detection may be effective since glitches and delay transitions are observable at *error* signal during the comparison window. Fault-free and faulty (hard, soft and timing errors) cases are highlighted in Figure 2.6.

Although it is capable of detecting hard, soft and timing errors that affect *CLs*, the static comparator has a drawback: electrical mask. In fact, soft and timing errors result in small differences (glitches or late transitions) between *vout1* and *vout2* during the comparison window. These slight differences might be

filtered out by the XOR gates. If they are not, small glitches produced by the local comparison might also be masked by the multi-layer Or-tree. This results in undetectable errors. In order to solve this problem, we integrate dynamic CMOS gates to the comparator structure illustrated in Figure 2.5.

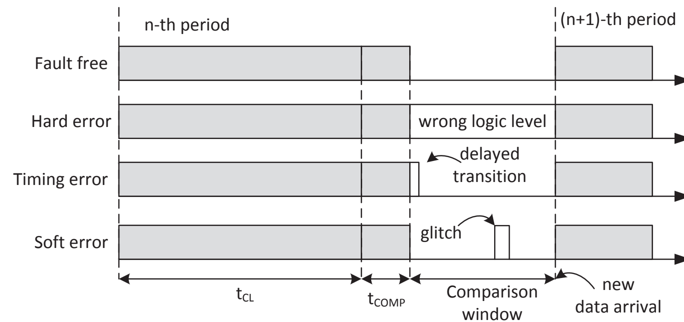


Figure 2.6 Error Signal

Dynamic CMOS

Dynamic or clocked CMOS gates are used to increase speed, decrease power dissipation while reducing the complexity of combinational logic circuits [BAK10]. The basic idea is to use capacitive inputs of the MOSFET to store a charge and thus remember a logic level for later use.

Figure 2.7 illustrates the principle of a dynamic gate using a pull-down network (PDN). In this Figure, *CL* presents the input capacitance of the next logic stage or of an output inverter. When the clock ϕ is at logic-0 (pre-charge phase), T1 is opened while *T2* is closed. *CL* node is therefore charged to VDD and *output* is at logic-1. When ϕ switches to logic-1 (evaluation phase), *T1* is closed while *T2* is opened. Depending on the NMOS logic, the pre-charged capacitance *CL* might be discharged to *GND* (*output* switches logic-0) or stay at *VDD* (*output* remain logic-1).

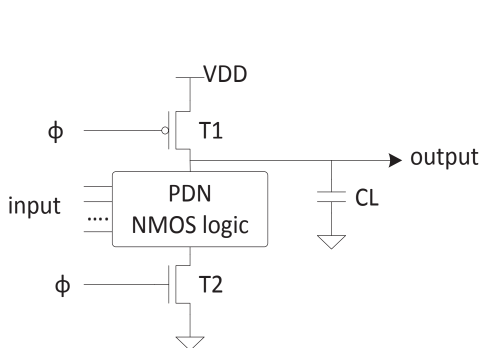


Figure 2.7 Dynamic CMOS Logic

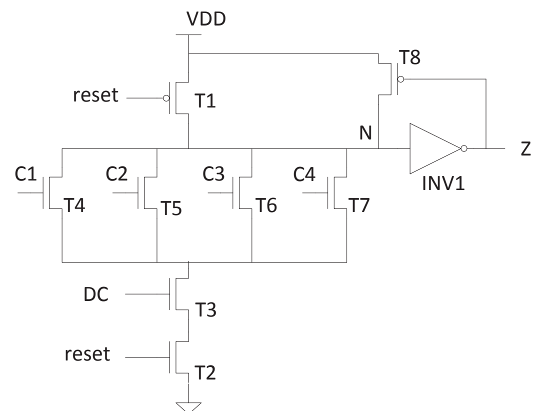


Figure 2.8 Dynamic OR

Compared to static CMOS structures, the dynamic CMOS presented in Figure 2.7 does not have a pull-up network made of PMOS logic. This simplification gives dynamic CMOS gates with higher switching speed and lower power consumption.

In Figure 2.8, we present the concrete example of a 4-input dynamic OR (DOR). The gate is controlled by *reset* and *DC* signals. During the pre-charge phase, *reset* is at GND. Input capacitance of inverter *INV1*

is pre-charged to VDD, which puts output *Z* to logic-0. During the evaluation phase, *reset* and *DC* are both at VDD, which means T1 is closed while T2 and T3 are opened. If all inputs Ci (i=1..4) are at logic-0 then the PDN is closed and N is kept at logic-1 while Z remains at logic-0. If at least one of the inputs Ci turns to logic-0 then a discharge current path exists. Consequently, N is pulled down to VDD and Z switches to logic-1. Note that once the discharge happens, Z will remain at logic-0 until the next pre-charge phase.

During the evaluation phase, dynamic CMOS logic suffers from leakage currents. Even if the PDN is closed, these currents still slowly discharge node *N* and causes a false logic value at output *Z*. Therefore, we need a “keeper” to maintain *N* at logic-1 when the PDN is closed. Besides, the keeper must be weak enough so that when the PDN is opened, node *N* can be pulled down to logic-0. In [DAS09], the authors proposed a “weak keeper” formed by a two-inverter loop. However, in order to reduce the area overhead, we decided to use a feedback transistor T8 as proposed in [BAK10].

Pseudo-dynamic comparator

There are different ways to use dynamic CMOS gate to improve the comparator. One method consists of using the transition detector proposed in [DAS09]. This detector, presented in Figure 2.9, is capable of detecting all transition arriving at input *N* during the high phase of clock *DC*. By adding this detector to every output bits of the CLs, we can detect small glitches at these signals during the comparison windows and thus detect soft and timing errors. However, this method leads to a very high area overhead as the output number of CLs is normally high. We can also combine output signals of CLs into a one-bit signal using a parity tree before employing the transition detector, as suggested in [PAL11]. Although less costly in term of area overhead, this method also has electrical masking problem caused by the parity tree itself.

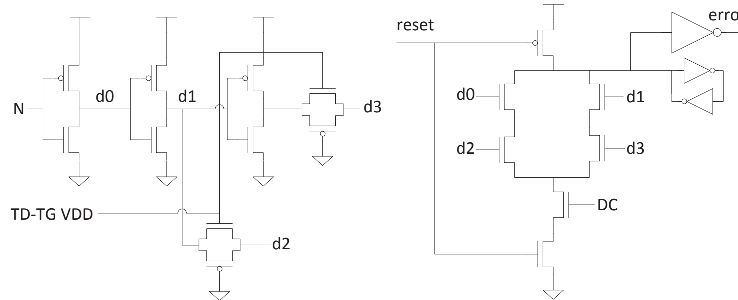


Figure 2.9 Transition Detector Structure

Another approach to implement the pseudo-dynamic comparator is to use dynamic gates inside the static comparator structure. This is achieved by replacing one part of the static comparator with dynamic CMOS logic. The obtained circuit is called pseudo-dynamic comparator.

Figure 2.10 shows our proposed pseudo-dynamic comparator. Similarly to the static comparator presented in Figure 2.5, the pseudo-dynamic comparator has two stages: local and global comparison. The local comparison stage consists of static XOR gates whilst the global comparison stage is an OR-tree combining dynamic and static gates. The first layer of the Or-tree is made of dynamic OR gates (DOR) while other layers use static OR gates. Compared to the static comparator, the pseudo dynamic comparator has two more inputs, which control *reset* and *DC* inputs of the DOR gates. For the reason of clarity, these control signals are not presented in Figure 2.10.

As mentioned above, DOR gates' outputs are stable at logic-0 due to the keeper during the pre-charge phase (*reset*'s low phase). During the evaluation phase (*reset* and *DC*'s high phase), if *vout1* and *vout2* are stable and identical then all signals Ci (i = 1..n) are also at logic-0 which maintain DOR outputs unchanged. Therefore, *error* remains constant logic-0 representing a fault-free case. Otherwise, if *vout1* and *vout2* differs, there will be either a constant logic-1 (presence of hard error) or glitches (presence of soft or timing error) at the Ci signals. At least one of the DOR outputs will then turn to logic-1, which makes *error* signal switch to logic-1 indicating "Error detected". The *Error* signal will remain at logic-1 until the next *reset* signal is applied. Note that *DC* defines comparison window of the comparator during its high phase.

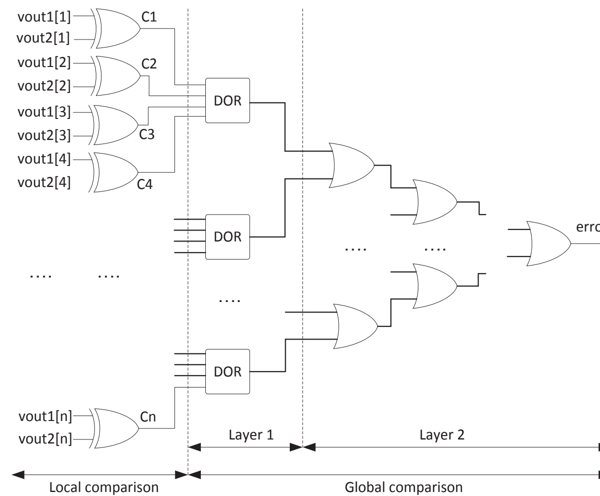


Figure 2.10 Pseudo-Dynamic Comparator Structure

In the pseudo-dynamic comparator structure (Figure 2.10), only the first layer of the global comparison stage is implemented using dynamic gates. In fact, when a glitch is captured by these gates, their outputs remain logic-1 even if the glitch has disappeared. Therefore inputs of the second layer are stable signals which will not be filtered by the OR tree. Consequently, static gates of this layer can also guaranty correct operation of the global comparison.

As error detection is place right after the XOR gates, our pseudo-dynamic comparator is more sensitive to small glitches caused by soft and timing errors compared to the static comparator. This improvement will be proven in Chapter 3. Besides, our architecture also promises lower power consumption. In fact, the global comparison stage is only active during the comparison window. Moreover, in fault free conditions, which happen most of the time, DOR's gate outputs are at constant logic-0, which means that the Layer 2 of the global comparison stage (Figure 2.10) does not consume dynamic power. Finally, we will show in the next chapter that our dynamic OR gate can be implemented with no area overhead compared to a static gate. Therefore our pseudo-dynamic comparator obtains better performance without introducing any area penalty.

Figure 2.11 present the complete error detection architecture using the pseudo-dynamic comparator. Compare to Figure 2.4, a control module is added to generate *DC* and *reset* signals, which command this comparator. Its logic implementation will be discussed in the next part.

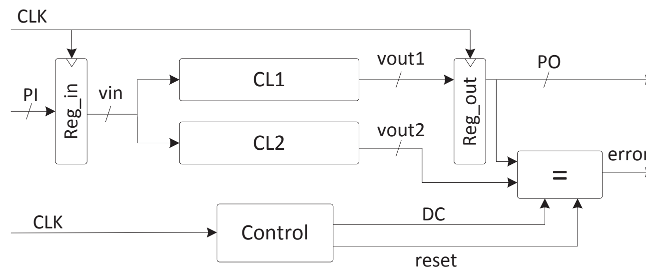


Figure 2.11 The Complete Error Detection Architecture

Control logic and timing constraints

In order to use the pseudo-dynamic comparator for error detection, we must control the comparison window by generating a *DC* signal. The comparison must start (*DC* at logic-1) only when both *PO* and *vout2* are stable. Otherwise, valid transitions (while combination logics are calculating) may be flagged as errors. Applied to our architecture in Figure 2.11:

- The comparison must start when both outputs *PO* of register Reg_out and *vout2* of combinational logic CL2 are established. During a fault-free operation, CL1 and CL2 must finish their calculation before the *CLK* capture edge. After this CLK edge, Reg_out will add a delay t_{dff} before *PO* is established, where t_{dff} represents the CLK-to-Q delay of output D flip-flops. Consequently, this condition requires the comparison windows (*DC* high phase) to begin later than t_{dff} after *CLK* rising edge.
- The comparison must finish before *PO* and *vout2* start changing value. When both signals are established, *PO* will only vary at next *CLK* capture edge. However, as the previous *CLK* capture edge has released a new input *vin*, *vout2* will start changing value after a delay t_{short} compared to this edge (t_{short} represents the short path of CL2). Therefore, to satisfy this condition, the comparison window (*DC* high phase) must finish earlier than t_{short} after *CLK* rising edge.

Besides, after each comparison, *reset* signal must be applied so that the pseudo-dynamic comparator is ready for new error detection. This must happens between each comparison window and the next one.

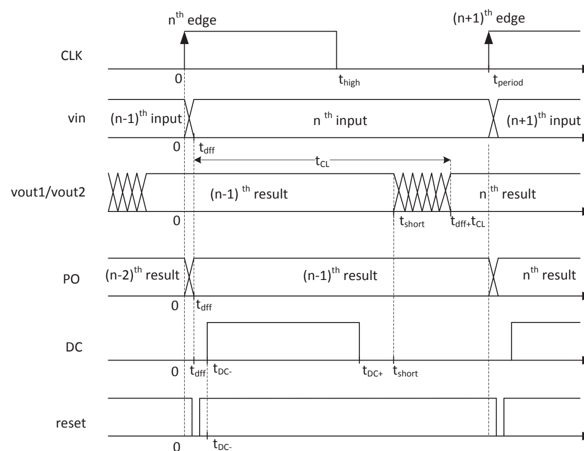


Figure 2.12 Timing Constraints for Error Detection Scheme

Figure 2.12 presents a correct timing scheme for our error detection method:

- After each *CLK* rising edge, input *vin* of CLs and primary output *PO* of the complete architecture are unstable during the switching time t_{dff} of Reg_in/Reg_out. Note that we suppose here that both registers are made of same D flip-flops and thus have similar delays. After this short duration, *vin* and *PO* remain stable until the next CLK rising edge.
- At n^{th} *CLK* edge (t=0), *vout1* and *vout2* are stable at the $(n-1)^{th}$ computation result. These signals remain stable until $t=t_{short}$ and then start varying. They return to stable state (n^{th} result) at $t=t_{dff}+t_{CL}$ where t_{CL} represents the maximum between the computation times of combinational logics CL1 and CL2.
- *DC* signal defines the comparison window of the Comparator during its high phase between t_{DC-} and t_{DC+} after each CLK rising edge. The timing constraints for this signal are:

$$t_{dff} < t_{DC-} < t_{DC+} < t_{short} \tag{2.5}$$

- *Reset* signal is activated (logic-0) each period before the comparison window which means earlier than t_{DC-} after each CLK rising edge.

Figure 2.13 illustrates a simple implementation of the control module which generates *DC* and *reset* signal in Figure 2.12. In Figure 2.13-a, *DC* is generated using a buffer bufDC which adds a delay t_{bufDC} to *CLK* signal. In order to respect the condition (2.5), we must have:

$$t_{bufDC} > t_{dff} \tag{2.6}$$

and

$$t_{short} > t_{bufDC} + t_{high} \tag{2.7}$$

Condition (2.6) can be obtained by adjusting the delay of bufDC while condition (2.7) requires either t_{high} reduction by controlling *CLK* duty cycle or t_{short} increase by adding delay buffers to LCs' short path [DAS09]. In Figure 2.13-b, *reset* is created using a glitch generator combined of a buffer bufRComp, an inverter invRComp and an NAND gate. Timing constraint of *reset* is obtained by fine-tuning delay time of these gates.

Figure 2.13 Control Module for Error Detection Architecture

Note that the *reset* signal will be applied every CLK cycle, regardless of errors occurrence. Therefore, *error* is at logic-0 at the beginning of each period. If this signal turns to logic-1 during n^{th} period then the $(n-1)^{th}$ computation result is not correct.

2.2.4 Conclusion

The complete error detection architecture studied in this section is presented in Figure 2.11. Using Duplication/Comparison, it allows both transient and permanent error detections. Moreover, this concurrent error detection method is more adapted for logic circuits with better detection capability and lower area cost compared to coding techniques. Furthermore, to improve the detection of SET and timing error, we use pseudo-dynamic comparator which detects small glitches better than classic static comparators. We have also proposed control logics and different timing constraints for the error detection scheme (Figure 2.13).

2.3 Transient error correction

As we have seen in the last chapter, transient errors only affect circuits during a short period of time. Therefore, by re-computing the affected input vectors when these errors have been removed, we will obtain a correct operation. The advantage of this method is that it does not require particular knowledge of the logic circuit and hence, respects our "Plug and Play" constraint. Moreover, this timing redundancy is only used when an error is detected. Thus, it will not affect circuit's performance during fault-free operations which happen most of the time. Finally, integrating the re-computation scheme into our existing error detection architecture does not require much area overhead compared to fault masking techniques such as TMR.

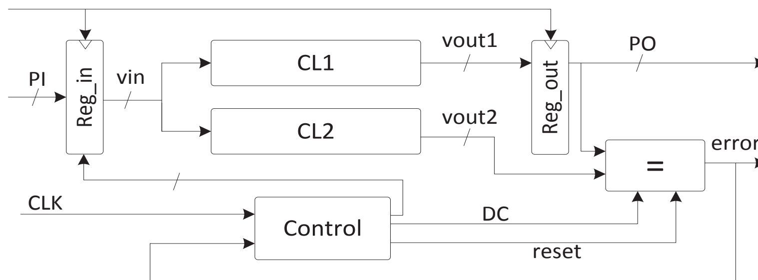


Figure 2.14 Transient Error Correction Architecture

Figure 2.14 illustrates an implementation of the re-computation technique integrated into the duplication/comparison error detection scheme presented previously. Compared to the last structure, new control signals are introduced to drive the input register Reg_in. In a fault-free case, the structure works exactly like the one presented in Figure 2.11. However, if an error is detected, the Control logic will be informed by the *error* signal and the structure will enter into the "Transient Error Correction" phase. The new signals will order Reg_in to repeat the affected input vector. Hence, a re-computation will tolerate transient errors, and the complete structure will then return to normal operation.

2.3.1 Input register

As we have seen in 2.2.3, the pseudo-dynamic comparator is placed after Reg_out so that the logic circuit's performance is not affected during fault-free operations. However, this configuration leads to a difficulty for the re-computation scheme. Let's suppose that when our circuit is computing the nth input vector, an error occurs at CL1. The faulty n^{th} output will be captured at $(n+1)^{th}$ CLK rising edge. At the same time, Reg_in has passed a new input vector to CLs. Consequently, when the error is detected at $(n+1)^{th}$ period, the nth input is no longer available for a re-computation. This problem is illustrated in Figure 2.15.

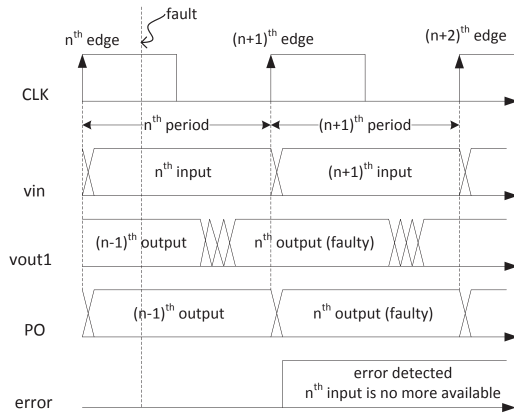


Figure 2.15 Re-computation Problem

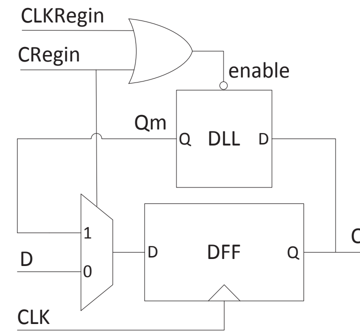
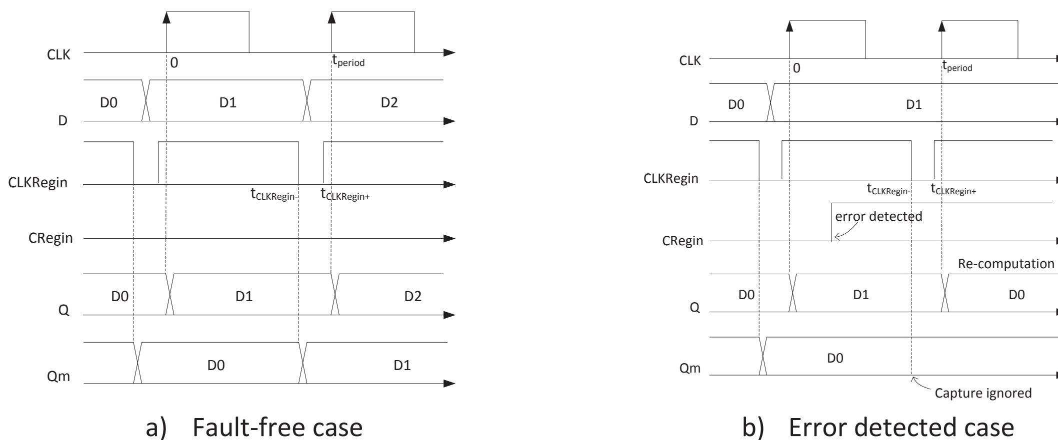


Figure 2.16 Modified D Flip-Flop mDFF for Re-computation

In order to use re-computation technique for transient error correction, we must resolve the problem above. One solution is to use additional memories to preserve previous input data until the computation result has been proven correct. As our architecture targets logic designs using D flip-flop based registers, we propose in Figure 2.16 a modified D flip-flop mDFF that satisfies this requirement.

In Figure 2.16, a low level sensitive D latch (DLL) is used to store previous input in *Qm* when the original D flip-flop (DFF) has captured a new value for *Q*. The time period during which *Qm* is maintained in DLL is defined by a new clock signal *CLKRegin* and a control signal *CRegin*. The last signal controls also a 2:1 multiplexer which decides whether the new data *D* or the memorized value *Qm* will be passed to *Q* at the next *CLK* rising edge. Note that while the additional latch and the 2:1 multiplexer must be added for each modified flip-flop, the OR gate which control *enable* input of DLL can be shared between them.



a) Fault-free case b) Error detected case

Figure 2.17 Modified D Flip-Flop's Function

Figure 2.17 explains how a modified D flip-flop works in: a) Fault free case and b) Error detection case:

- In a fault free case (Figure 2.17-a), *CRegin* remains at logic-0. Hence, the multiplexor always selects input *D*. Consequently, at *CLK* rising edge ($t=t_{period}$), the main flip-flop DFF captures new data from *D* and hence, *Q* switches from D1 to D2. Before this moment, the prior data D1

has been captured by DLL during its transparent window (low phase of *CLKRegin)* from $t_{CLKRegin-}$ to $t_{CLKRegin+}$. This data is stored in DLL as long as *CLKRegin* remains logic-1.

- In the situation of Figure 2.17-b, a fault becomes active while CLs are computing D0. The erroneous result is then detected during the next CLK period, between t=0 and $t=t_{period}$. CRegin will turn to logic-1 during this period so that the *enable* input of DLL is also kept at high level. Consequently, D1 is not captured during the next CLKRegin low phase from $t_{CLKRegin-}$ to $t_{CLKRegin+}$. Therefore, data D0 will be maintained by DLL. At the next CLK rising edge ($t=t_{period}$), the multiplexer will select this data for re-computation.

Our method requires two CLK cycles for transient error correction: the first cycle consists in raising *error* signal while the second cycle is for re-computation. Also note that in Figure 2.17, when the error is detected, input *D* does not change from D1 to D2 and hence, D1 remains available after the error correction process. This must be controlled at system level and will be discussed further in Chapter 4.

The complete transient error correction architecture using modified input register is presented in Figure 2.18. The additional control signals *CRegin* and *CLKRegin* are also driven by the control logic which generates *DC* and *reset* for the pseudo-dynamic comparator. A *resetControl* signal is added which allow initialization of the control module.

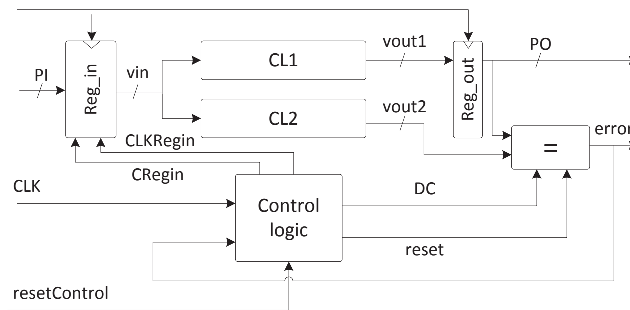


Figure 2.18 The Complete Transient Error Correction Architecture

2.3.2 Reset signal

With the error detection mechanism presented in the last section, *error* signal is reset to logic-0 at the beginning of every *CLK* cycle. This is necessary because the architecture computes new input vector at each period, regardless of errors occurrence. However, when we integrate errors correction to the architecture, this periodic *reset* signal is no longer validate. In fact, as explained previously, each time an error is detected for n^{th} input, the combinational logics will lost one CLK cycle running the $(n+1)^{th}$ input before re-computing the prior one. Once the error is corrected, the architecture will return to normal operation and run the $(n+1)^{th}$ input. Consequently, at system level, the $(n+1)^{th}$ result must be ignored when error correction mechanism takes place. One solution is to keep *error* signal at logic-1 indicating invalid output during this period. To do that, we need to dismiss (keep at logic-1) the periodic *reset* signal for one *CLK* period every time an error is detected. Furthermore, to minimize dynamic power consumption of the comparator, we can keep *reset* at logic-1 during normal operation and active it only when an error is detected (one period after the detection).

2.3.3 Transient error correction mechanism

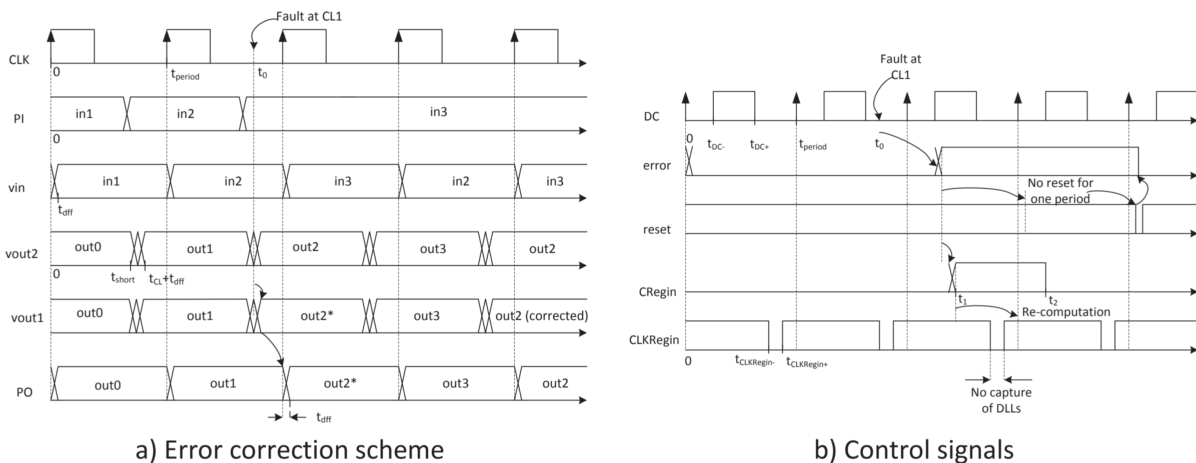
Figure 2.19-a illustrates the complete transient error correction mechanism of our architecture:

- As the first period is fault-free, we have the same waveforms as those of the error detection scheme. Note that even if Reg_in is modified, the registers still have similar delay time t_{dff}

because it depends mainly on the CLK-to-Q delay of the D flip-flops, which remains identical for both of them.

- During the second period, a transient fault occurs at combinational logic CL1 at t=t_0. After in2 computation, this fault causes an error at output *vout1* under the form of a faulty value of out2 (out2*).
- The erroneous output out2* is captured by Reg_out at the third CLK rising edge. Although the error is detected during this *CLK* cycle, a new calculation of input in3 has been launched at the beginning of the period. Consequently, both CLs work as if no error were detected and answer with out3.
- Error detection of the last cycle has triggered the error correction mechanism. Hence, during the fourth period, Reg_in passes input in2 to CLs again. The re-computation happens during this period. At the end of the cycle, both CLs provide the correct value out2.
- At the fifth period, Reg_out captures the corrected result out2 while Reg_in release input in3. The system returns to its normal operation mode.

Note that during the fourth and the fifth CLK edges, *PI* does not change value so that in3 remain available after the error correction process.



a) Error correction scheme b) Control signals

Figure 2.19 Transient Error Correction Mechanism

In Figure 2.19-b, we explain how the control signal manages the operations detailed above:

- For each period, *CLKRegin* turns to logic-0 between $t_{CLKRegin-}$ and $t_{CLKRegin+}$ after CLK rising edge so that at fault-free operation, prior input data will be captured by the additional latches DLL of Reg_in before its main flip-flops DFF pass to new input. The stored data will be available until the next *CLKRegin* falling edge.
- The transient fault (t=t_0) causes a difference between *PO* and *vout2* during the third *CLK* period. The pseudo-dynamic comparator detects this error during its comparison window and raises *error* signal to logic-1, triggering the error correction process.
- When *error* turns to logic-1, it makes *CRegin* switch to high level too (t=t_1). So, during the next *CLKRegin* low phase, data capture DLL is ignored. Therefore, previous input data in2 remains stored at these latches. Consequently, at the forth *CLK* rising edge, this stored input data is selected for re-computation because *CRegin* is still at logic-1. After this edge, *CRegin* returns to logic-0 (t=t_2) so that new input will be computed at the next period.
- As the error is detected at the third *CLK* period, *reset* signal of the pseudo-dynamic comparator is not activated (i.e. is kept at logic-1) until the fifth cycle. That's why at the third

and the fourth *CLK* cycle, *error* remains at logic-1 indicating that primary output *PO* must be ignored. The fact that error remains at logic-1 during these periods is also used at system level to control primary input flow *PI* so that no new input vector comes during the error correction process. At the fifth period, the comparator is reset and the architecture returns to normal operation.

2.3.4 Control logic and timing constraints

DC signal

Compared to the error detection architecture, *DC* signal in transient error correction scheme has the same constraints. Its high phase must be placed when:

- Primary output *PO* has been established (t_{dff} after CLK rising edges where t_{dff} represents the CLK to Q delay of output D flip-flops)
- Output *vout2* of combinational logic CL2 has not changed value (t_{short} after CLK rising edges where t_{short} represents the short path of CL2)

We can use the same control logic as in Figure 2.13-a to generate *DC* signal, i.e. use a buffer bufDC to delay the *CLK* signal. To satisfy the timing constraints, we must also assure (2.6) and (2.7) by adjusting the delay t_{bufDC} of the buffer, high phase duration t_{high} of *CLK* and short path t_{short} of the combinational logics.

CLKRegin signal

The additional clock *CLKRegin* must guarantee a valid capture of the prior data by DLLs before the input register releases new value. Therefore, three timing constraints must be satisfied:

- First of all, the data capture of additional latches DLL must happen when the prior data is still available. As illustrated in Figure 2.16, these additional latches are placed at output of the main flip-flops DFF. Consequently, to insure this timing constraint, the transparent window of DLLs (*CLKRegin* low phase) must be placed before the capture edge of DFFs (*CLK* rising edge) after which new data has arrived at Q.
- Then, a new data capture must happen only when the computation result of the last data has been proven correct by the pseudo-dynamic comparator. As the comparator has a delay t_{Comp}, the falling edge of *CLKRegin* must take place at least t_{Comp} after *DC* falling edge.
- Finally, the signal Q must be stable (at prior data value) during the setup and hold time ($t_{setupDLL}$ and $t_{holdDLL}$) of latches to assure a good data capture. As Q changes value at *CLK* rising edge and the falling edge of *CLKRegin* will be placed when Q is established, this constraint means that:
  - *CLKRegin* low phase duration must be at least t_{setup}
  - *CLKRegin* rising edge must happen at least t_{hold} before *CLK* rising edge

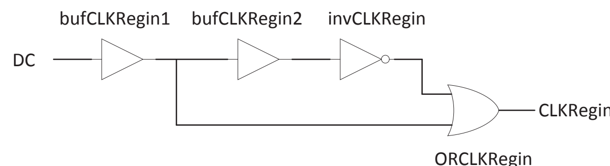


Figure 2.20 CLKRegin Generator for Transient Error Correction Architecture

In Figure 2.20, we propose a simple generator for CLKRegin. The main idea is to use a glitches generator to create *CLKRegin* low phase while *DC* is at logic-0. The buffer bufCLKRegin1 adds a delay

$t_{bufCLKRegin1}$ to *DC* so that *CLKRegin* falling edge happens at least t_{Comp} after the comparison window. Thus, we must have:

$$t_{bufCLKRegin\ 1} > t_{Comp} \tag{2.8}$$

The duration of *CLKRegin* low phase is assured by bufCLKRegin2. As this phase must last at least t_{setup}, the delay $t_{bufCLKRegin2}$ must satisfy:

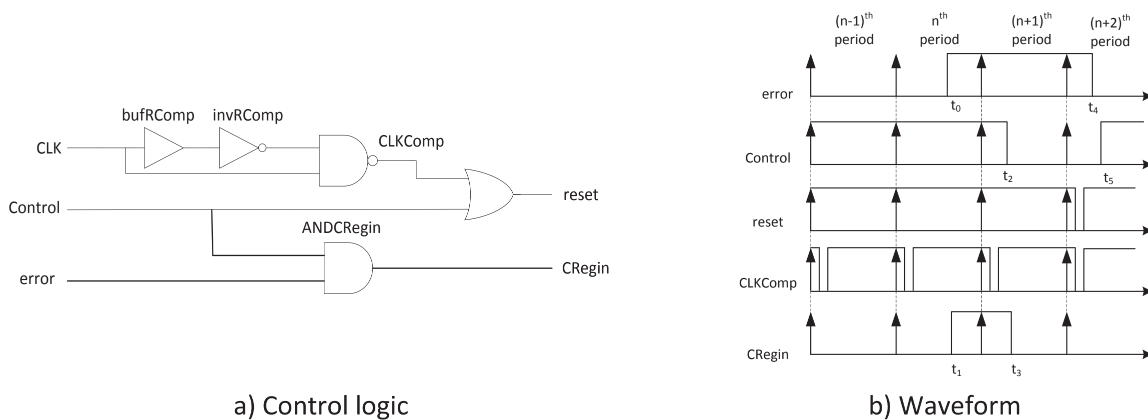
$$t_{bufCLKRegin\ 2} > t_{setup} \tag{2.9}$$

Finally, we must assure that *CLKRegin* rising edge happens at least t_{hold} before *CLK* rising edge. As in our generator, CLKRegin rising edge corresponds to the falling edge of DC after being delayed by the buffers, the inverter and the OR gate, this condition means that:

$$t_{bufDC} + t_{high} + t_{bufCLKRegin\ 1} + t_{bufCLKRegin\ 2} + t_{invCLKRegin} + t_{ORCLKRegin} + t_{hold} < t_{period} \tag{2.10}$$

Reset signal

As we have seen before, once an error is detected, we will wait for one *CLK* cycle before applying a reset to the pseudo-dynamic comparator. To do that, we can employ the circuit presented in Figure 2.13-b to generate a periodic signal *CLKComp*, and then introduce an additional *Control* signal to create *reset* from *CLKComp*. The method is illustrated in Figure 2.21.



a) Control logic b) Waveform

Figure 2.21 Reset and CRegin Generator for Transient Error Correction Architecture

We see in Figure 2.21-b that during normal operation (before (n-1)th period), *Control* is at logic-1. When an error is detected at nth period ($t=t_0$), *Control* remains at logic-1 until $t=t_2$ to keep *reset* at high level during (n+1)th period. It switches to logic-0 ($t=t_2$) before the (n+2)th *CLK* rising edge so that a reset will be applied to the comparator during this period. After the re-computation, at $t=t_5$, *Control* returns to logic-1 and the architecture comes back to normal operation.

For a correct function of the architecture, *Control* must satisfy two conditions:

- First of all, its falling edge ($t=t_2$) must happen after the rising edge of *CLKComp* during the (n+1)th period so that no reset is applied for this cycle.
- Then, its rising edge ($t=t_5$) must also happen after the rising edge of *CLKComp* during the (n+2)th period to make sure that a reset is applied correctly.

CRegin signal

We have explained previously that *CRegin* controls both prior data capture by additional latches DLL and new data capture by main flip-flops DFF. When an error is detected at n^{th} *CLK* cycle for $(n-1)^{th}$ output:

- *CRegin* turns from logic-0 to logic-1 to disable the nth input captured by the latches. Thus, *CRegin* rising edge must take place before the beginning of this capture at $t_{CLKRegin-}$.
- *CRegin* remains at logic-1 until the next *CLK* rising edge so that the $(n-1)^{th}$ input stored at DLLs (*Qm* outputs of the latches) will be selected by the multiplexers. After this edge, CRegin must return to logic-0 so that n^{th} input data (*D* inputs of the modified flip-flops) will be selected for subsequent calculations.

CRegin signal can be easily generated using the *Control* signal discussed before. Figure 2.21 shows how we can do that by simply combining *error* and *Control* using an AND gate ANDCRegin. We see that at $t=t_0$, when an error is detected, *error* turns to logic-1 while *Control* remains at logic-0 until $t=t_2$ after the $(n+1)^{th}$ *CLK* rising edge. Consequently, CRegin rises to logic-1 at $t=t_1$, just after t_0. This rising edge must happen before the beginning of CLKRegin low phase at $t_{CLKRegin-}$ after the CLK n^{th} edge. When *Control* is switched to logic-0 at $t=t_2$, *CRegin* also comes back to logic-0 at $t=t_3$ during the $(n+1)^{th}$ period. Therefore, at $(n+2)^{th}$ the architecture can return to normal operation. Note that before *Control* comes back to logic-1 ($t=t_5$), *error* has been reset to logic-0 at $t=t_4$ and hence, *CRegin* remains at logic-0.

To guarantee a correct function of the architecture, we must assure that:

- The rising edge of *CRegin* happens earlier than $t_{CLKRegin-}$ after *CLK* n^{th} rising edge. For each period, the latest error can be detected at t_{DC+} after *CLK* rising edge and the comparator has a delay t_{Comp}. Therefore, this condition means that:

$$t_{DC+} + t_{Comp} + t_{ANDCRegin} < t_{CLKRegin-} \tag{2.11}$$

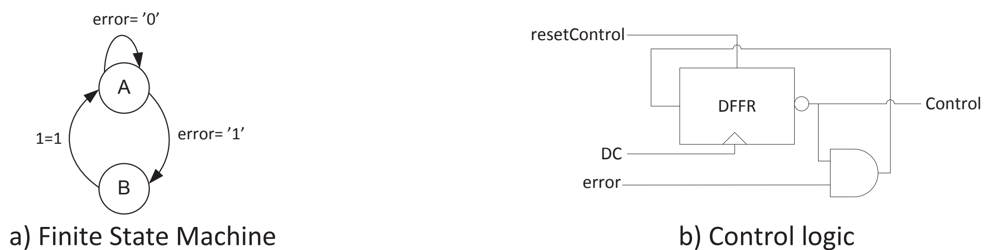
  where $t_{ANDCRegin}$ represents the delay of ANDCRegin. As $t_{CLKRegin-}$ is controlled by the buffer bufCLKRegin1, we can satisfy the equation above by controlling this gate so that:

$$t_{DC+} + t_{Comp} + t_{ANDCRegin} < t_{bufCLKRegin\ 1-} < t_{CLKRegin-} \tag{2.12}$$

- *CRegin* remains at logic-0 after $t=t_3$, which means that *error* falling edge (t=t4) must happen before *Control* falling edge (t=t5).

Control signal

As explained previously, in fault-free operations, *Control* is stable at logic-1. When an error is detected (*error* at logic-1), this signal drops to logic-0 during the next period and then returns to logic-1 one cycle later. This characteristic can be easily described by the finite state machine (FSM) in Figure 2.22-a where A represents the normal operation (error='0') when *Control* is at logic-1 and B represents the cycle after an error is detected (error='1'). Note that from B, regardless of the value of *error* (1=1), the system will return to normal state A at the next clock edge.



a) Finite State Machine　　b) Control logic

Figure 2.22 Control Signal Generator for Transient Error Correction Architecture

To realize the FSM described above, we will need a clock signal. As both rising and falling edges of *Control* must happen after the rising edge of *CLKComp* (in two adjacent periods), one option is to use *DC* signal which has the same characteristics. Moreover, by adjusting the delay of *DC* compared to *CLK*, we can also guarantee that *Control* falling edge will take place after the falling edge of *error* signal (Figure 2.21).

In Figure 2.22-b, we propose a logic circuit that realizes the FSM. It is composed of an AND gate and a D flip-flop (DFFR) whose asynchronous reset input is driven by *resetControl* signal. *Control* signal is connected to the output $\overline{\mathrm{Q}}$ of the flip-flop. Therefore, *Control* will be at logic-0 (or logic-1) if a logic-1 (or logic-0) is captured by the flip-flop. During the initialization phase of the FSM, *resetControl* is at logic-0, and *Control* will be set to logic-1 (state A). During normal operations, *error* is at logic-0 and output of the AND gate will remain at low phase too. At DC rising edge, this logic-0 will be captured by the flip-flop which makes *Control* remain at logic-1 (state A). If an error is detected then *error* will rise to logic-1 and thus, input of DFFR will turn to logic-1 too. Therefore, at new DC edge, logic-1 is captured which pulls *Control* to logic-0 (state B). Consequently, the output of the AND gate returns to logic-0 regardless of the value of *error*. At next DC edge, this logic-0 will be captured and hence, Control will come back to logic-1 (state A).

Complete control logic for the transient error correction architecture is presented in Figure 2.23.

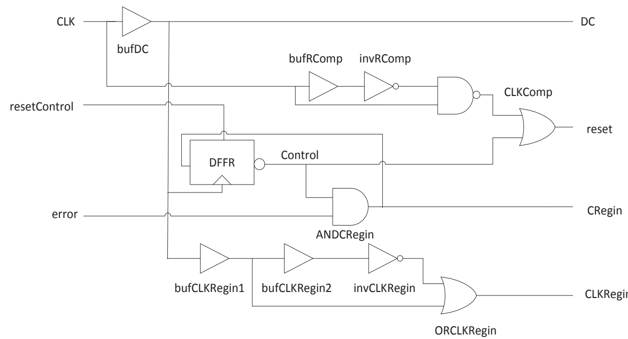


Figure 2.23 Control Module for Transient Error Correction Architecture

2.3.5 Conclusion

The complete transient error correction architecture studied in this section is presented in Figure 2.18. Using a timing redundancy based fault-tolerance method, the architecture needs two CLK cycles to tolerate transient errors. The *error* signal is kept at logic-1 during these two periods to inform the system so that new primary inputs will be hold until the correction process is done. Besides, the input register is modified to capture prior input data necessary for re-computation. Control signals for this register and that of the pseudo-dynamic comparator are produced by the control module in Figure 2.23. New timing constraints have been studied to assure good operation of the whole architecture.

2.4 Permanent error correction

In the previous section, we explained how to use timing redundancy to tolerate transient faults in logic circuits. Although simple and effective for temporary errors, this method does not work for faults which last for more than two clock cycles such as permanent or intermittent faults. For example, let's suppose that aging phenomenon causes a permanent stuck-at-fault at combinational logic CL1. When the fault becomes active, Duplication/Comparison mechanism will detect errors. However, when the affected vector is re-computed, the fault remains in CL1 and causes a new error. Consequently, the

complete architecture stops working for this input vector. Note that even though CL2 still operates correctly, it does not help tolerating faults because there will always be mismatches between outputs of CLs which lead to error detection.

In order to resolve the problem above, our solution is to integrate hardware redundancy in the existing architecture. The idea is to replace the affected combinational logic by a third one CL3 before performing a new re-computation. This time, as both CLs operate correctly, the fault will be tolerated. Figure 2.24 shows how we combine this permanent error correction technique and the ongoing transient error correction structure to form a hybrid fault-tolerant architecture.

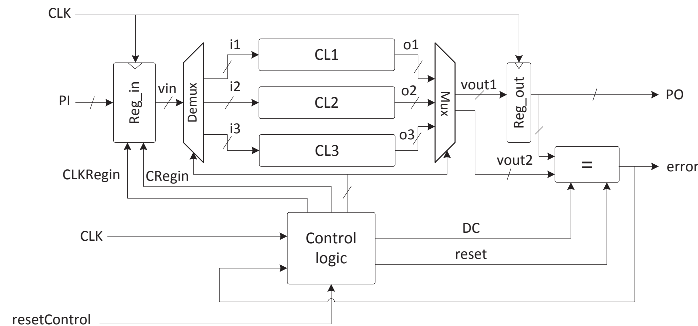


Figure 2.24 The Hybrid Fault-Tolerant Architecture

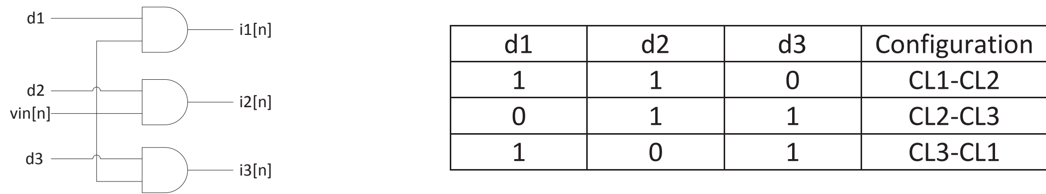
Compared to the transient error correction architecture presented in Figure 2.18, the hybrid fault-tolerant architecture has three instead of two copies of the combinational logic (CL1, CL2 and CL3). However, during normal operation, only two CLs (CL1 and CL2) run in parallel while the third one (CL3) is put on standby. Consequently, only two out of three CLs consume dynamic power. This is how our proposed method saves power consumption compared to the TMR technique. When an error is detected, the architecture will perform a re-computation which tolerates transient faults. In the case where errors persist after re-computation, a re-configuration will replace one of the two running combinational logics (CL1 for example) by the third one (CL3). If the replaced CL1, which is now on standby, is the faulty one then this re-configuration has eliminated the faults and the system can return to normal operation after a new re-computation. Otherwise, we will need a new re-configuration to finally put the affected CL2 on standby and use CL1 and CL3 to tolerate the fault.

The method described above works with the help of two additional modules (Figure 2.18) Demux and Mux which represent respectively an input demultiplexer and an output multiplexer. Their roles are selecting two running CLs while keeping the third one on standby. These modules receive control signals from the control logic which now uses a new finite state machine (FSM) to decide which CLs run and which CL does not.

In the following sub-section, we present details of Demux, Mux, FSM and the control logic.

2.4.1 Input de-multiplexer

In the proposed hybrid fault-tolerant architecture, the input demultiplexer Demux has two functions. First of all, it selects two functioning combinational logics (CL1 and CL2 for example) by driving input signal *vin* to their input vectors (*i1* and *i2*). Then, it must keep the third circuit (CL3) in standby by applying logic-0 to all of its input bits.



d1	d2	d3	Configuration
1	1	0	CL1-CL2
0	1	1	CL2-CL3
1	0	1	CL3-CL1

Figure 2.25 Elementary input demultiplexer

To realize the function above, Demux consists of many elementary demultiplexers eDmux, one for each input bit. In Figure 2.25 we propose a gate-level schematic for this circuit. The proposed eDmux is made of three AND gates which all receive n^{th} bit of *vin* (*vin[n]*) as one input. Their remaining inputs are driven by control bits *d1*, *d2* and *d3* provided by the control logic. In the table of Figure 2.25, we see that to put a combinational logic, CL1 for example, on operation (or standby), we only have to keep its corresponding control bit d1 at logic-1 (or logic-0).

Note that even at standby state, combinational logic still consumes leakage power. To optimize this kind of power consumption, one idea is to find out an input vector for which the leakage power of CLs is the smallest. Then, we can modify the Demux so that it applies this input vector while putting the CLs on standby. Figure 2.26 shows an example of an optimized Demux for three-bit input vectors *vin*. The control bits used for this Demux are the same as those in Figure 2.25. Using this demultiplexer, when a circuit is put on standby, its input vector will be kept at '010' instead of '000'.

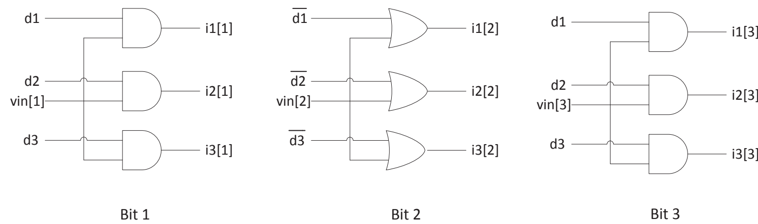


Figure 2.26 Example of an Optimized Input Demultiplexer

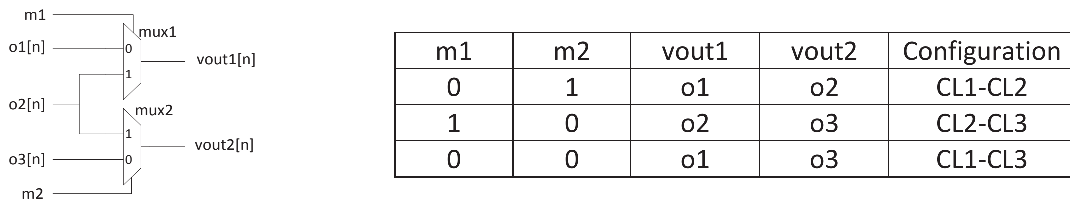
2.4.2 Output multiplexer

While the input demultiplexer selects two running circuits (CL1 and CL2 for example) by applying *vin* to their inputs (*i1* and *i2*), the output multiplexer Mux must drive their output vectors (*o1* and *o2*) to its outputs *vout1* and *vout2*. These vectors will then be compared to detect errors. *Vout1* is also captured by the output register Reg_out to provide the primary output *PO*.

As for the input demultiplexer, Mux is also combined of several elementary multiplexers eMux, one for each primary output bits of the hybrid fault-tolerant architecture. In the flowing, we will present different ways to implement eMux.

Method 1

In Figure 2.27, we present a simple elementary multiplexer eMux for the nth output bit. It is made of two 2:1 multiplexers mux1 and mux2 which are controlled by *m1* and *m2* signals. Output *vout1[n]* of mux1 is connected to *o1[n]* or *o2[n]* when *m1* is at logic-0 or logic-1, respectively. Meanwhile, output *vout2[n]* of mux2 is connected to *o3[n]* or *o2[n]* when *m1* is at logic-0 or logic-1, respectively. The table of Figure 2.27 summarizes different values of the control bits and the corresponding configurations.



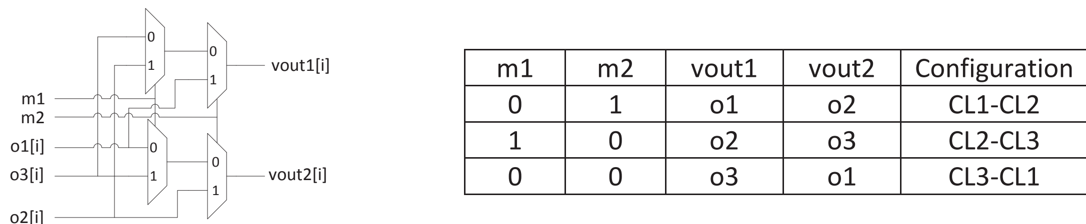
m1	m2	vout1	vout2	Configuration
0	1	o1	o2	CL1-CL2
1	0	o2	o3	CL2-CL3
0	0	o1	o3	CL1-CL3

Figure 2.27 Elementary output multiplexer – Method 1

The eMux presented above is very simple. But using this structure, *vout1* is never connected to *o3* while *vout2* cannot be connected to *o1*. This constraint will limit the performance of our hybrid fault tolerant architecture with regard to timing issues. Let us suppose that for a certain input vector, CL1 has a longer calculation time due to process variations. As a result, its output *o1* is established after *CLK* rising edge but still before the comparison window. When *o1* is connected to *vout1*, Reg_out always captures a faulty result. Therefore, there is only one configuration of Figure 2.27 that can operate correctly: CL2-CL3. A possible solution that allows error correction when CL1 is running consists in connecting *o1* to *vout2*. As *vout2* is used for comparison after *CLK* edge, this configuration allows additional time for CL1 to finish its computation.

Method 2

We propose another eMux that is composed of four 2:1 multiplexers in Figure 2.28. This circuit is control by the same signals *m1* and *m2* as in the previous method. The table of Figure 2.28 shows different configurations corresponding to various values of these signals. The first two configurations work exactly like in Method 1. The third configuration is slightly different. In fact, when both m1 and m2 are at logic-0 we will still have CL1 and CL3 work in parallel. However, output *o1* of CL1 is connected to *vout2* while output *o3* of CL3 is connected to *vout1*. Consequently, by using this bigger eMux, we can improve the performance of the hybrid fault-tolerant architecture.

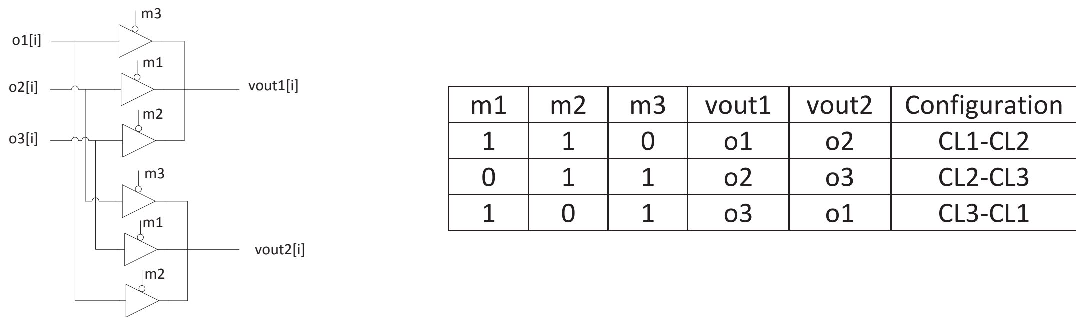


m1	m2	vout1	vout2	Configuration
0	1	o1	o2	CL1-CL2
1	0	o2	o3	CL2-CL3
0	0	o3	o1	CL3-CL1

Figure 2.28 Elementary output multiplexer – Method 2

Tri-state buffer

Another way to implement an elementary multiplexer is to use active-low tri-state buffers. Controlled by a signal *c*, these buffers have two states. When *c* is at logic-0, they work like normal buffers and hence, their outputs and inputs have the same logic values. However, when *c* is at logic-1, their output will be held in a high-impedance state (i.e. disconnected from the rest of the circuit). We can simply think of the tri-state buffer as a switch. The switch is opened when *c* is at logic-0 and closed when c is at logic-1.



m1	m2	m3	vout1	vout2	Configuration
1	1	0	o1	o2	CL1-CL2
0	1	1	o2	o3	CL2-CL3
1	0	1	o3	o1	CL3-CL1

Figure 2.29 Elementary output multiplexer – Tri-state buffer

Figure 2.29 explains how we can use active-low tri-state buffers to make an eMux. The circuit is controlled by three signal m1, m2 and m3. To make two circuits (CL1 and CL3 for example) run together, we must keep their respective control signals (m1 and m3) at logic-1 and the third one (m2) at logic-0. Therefore, we can use the same control signal as those of the input demultiplexer Demux (Figure 2.25). However, the control logic must assure that in no case, two control signals can be at logic-0 at the same time. Because otherwise, there will be more than one signal connected to the same bus, which is not allowed. The table of Figure 2.29 shows that with the proposed circuit, we can have the same configurations as Method 2, with optimized performance of the hybrid fault-tolerant architecture.

Note that we can also realize the same circuit using transmission gates (pass-gates) instead of tri-state buffers. However, while usually being more costly in term of silicon area, the tri-state buffers have advantages of output drive-strength compared to transmission gates.

2.4.3 Reconfiguration finite state machine

The Finite State Machine (FSM) manages the re-configuration of the hybrid fault-tolerant architecture by deciding which two circuits run in parallel. When an error is detected, two tolerant schemes are investigated:

- In the first scheme (FSM1), the architecture will not be re-configured when the first error occurs. Two working circuits will run the affected input vector one more time. If the error is transient then it will be corrected after this re-computation and hence, the architecture can return to normal operation. However, if the error remains, the FSM must re-configure the architecture before applying the re-computation. This time, the faulty circuit will be eliminated and the architecture can then operate correctly. Note that when an error is detected, we do not know which combinational logic was affected by faults. Consequently, it might take two re-configurations in order to tolerate permanent faults.
- The second fault-tolerant scheme (FSM2) consists of changing the configuration each time an error is detected. This method also takes one re-configuration to correct transient errors, and maximum two re-configurations to tolerate permanent faults in one of the CLs. However, supposing that the possibility of having permanent faults are equal for the three combinational logics, we have 50% of chance that one re-configuration is enough to eliminate the faulty one. Therefore, this method might be faster for permanent fault-tolerance.

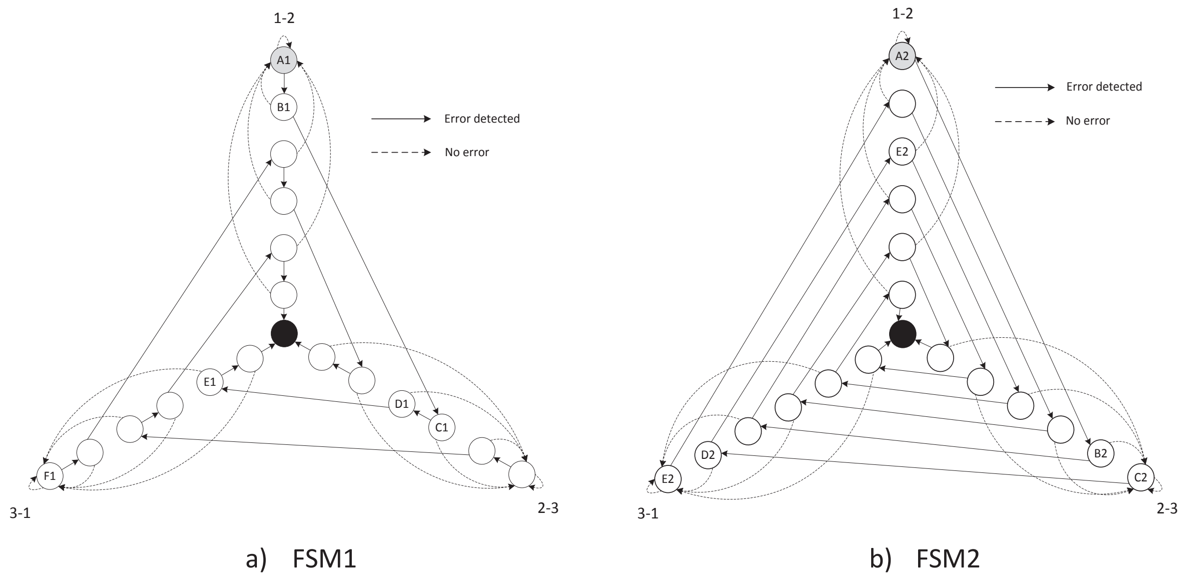


Figure 2.30 Finite state machine diagrams

Diagrams of both FSMs are illustrated in Figure 2.30 while their functioning examples are shown in Table 2.2 (FSM1) and Table 2.3 (FSM2).

In both diagrams of Figure 2.30, states of FSMs are represented by circles. The initial state is in gray while the final state (the one when no more correction/tolerance is possible) is in black. Transitions between different states at each clock edge are illustrated by arrows. Solid arrows correspond to error detected situations while dotted arrows are used when there is no error occurrence. There are three branches in each diagram, corresponding to the three possible configurations (1-2, 2-3 and 3-1). For example, 1-2 means that CL1 and CL2 are running in parallel while CL3 is on standby.

In Table 2.2 and Table 2.3, the first row indicates the computation cycle; the second row specifies the current configuration; the third row highlights the input sequence; the last row points out the active state in the corresponding diagram in Figure 2.30. In both examples, when the first transient error (T) occurs at the 3th cycle, input vector V3 is repeated for re-computation. However, the configuration remains 1-2 for FSM1 (state A1 and B1) while switches from 1-2 (state A2) to 2-3 (state B2) for FMS2. In both cases, the transient error is corrected after one re-computation. From 5th computation, a permanent fault (P2) is activated at CL2 by input vector V4. FMS1 realizes four faulty computations of V4 using configurations 1-2 and 2-3 before operating correctly with configuration 3-1 at the 9th computation. Meanwhile, FSM2 take only one computation cycle to switch from 2-3 to 3-1 at 5th period.

In the finite state machine diagrams above, the architecture stops working (black state in Figure 2.30) after six re-computations. This is our choice so that the hybrid fault-tolerant architecture can tolerate transient errors which occur once during the tolerance of a permanent error. This will be discussed further in Chapter 4. However, note that depending on the application, we can choose to stop the FSM after another number of re-computations.

Computation	1	2	3	4	5	6	7	8	9	10
Configuration	1-2	1-2	1-2	1-2	1-2	1-2	2-3	2-3	3-1	3-1
Input vector	V1	V2	V3	V3	V4	V4	V4	V4	V4	V5
Current state	A1	A1	A1	B1	A1	B1	C1	D1	E1	F1
			⇧		⇧	⇧	⇧	⇧		
			T		P2	P2	P2	P2		

Table 2.2 FSM1 Functioning Example

Computation	1	2	3	4	5	6	7	8	9	10
Configuration	1-2	1-2	1-2	2-3	2-3	3-1	3-1	3-1	3-1	3-1
Input vector	V1	V2	V3	V3	V4	V4	V5	V6	V7	V8
Current state	A2	A2	A2	B2	C2	D2	E2	E2	E2	E2
			⇧		⇧					
			T		P2					

Table 2.3 FSM2 Functioning Example

2.4.4 Control logic and timing constraints

Unlike the error detection architecture (Figure 2.11) and the transient error correction architecture (Figure 2.18), the complete hybrid fault-tolerant architecture has to insert two modules in the data path, between primary input *PI* and primary output *PO*: the input demultiplexer Demux and the output multiplexer Mux. This affects the calculation time of the complete structure. However, delays (t_{Demux} and t_{Mux}) of the two modules are normally negligible compared to the logic circuit's calculation time. It is because they are made of small elementary circuits (eDmux and eMux) running in parallel. Moreover, as the hybrid fault-tolerant is capable of tolerating timing errors, in non critical applications, we can use more aggressive timing for the combinational logics which may results in a total delay of the fault-tolerant architecture equal to the delay of the original logic circuit. To summarize, the additional modules Demux and Mux do not affect significantly functional frequency of the logic circuit.

In the hybrid fault-tolerant architecture, we still need control signals of the transient error correction architecture: *DC* and *reset* for the pseudo-dynamic comparator; *CRegin* and *CLKRegin* for the input register; *resetControl* for the control logic. We can reuse the same logic circuits proposed in the last section (Figure 2.23) for the new architecture:

- *DC* signal can be generated with respect to timing constraints in (2.6) and (2.7). However, in (2.7) we must take into account the delay of Demux and Mux. Hence, t_{short} will represent the short path between *vin* and *vout2*.
- *CLKRegin* must satisfy timing constraints in (2.8), (2.9) and (2.10).
- *Reset* and *CRegin* must respects (2.11) and (2.12).

Besides these four control signals, we will need to generate control bits of Demux and Mux in order to perform architectural re-configurations. To do that, first of all, we will need to realize the finite state machine FSM. Then, from the output of this FSM, we will produce these control bits.

Logic circuit of the finite state machine

To design logic implementations of the FSM from the state diagram in Figure 2.30, we must define firstly the clock signal that drives transitions of the FSM. As for the transient error correction scheme presented in the previous section, fault correction in the hybrid fault-tolerant architecture also requires two clock cycles. Suppose that a fault becomes active at $(n-1)^{th}$ CLK cycle, it will be detected (*error* turns

to logic-1) at nth period and the re-computation will take place at (n+1)th period. Consequently, the re-configuration must happen during nth cycle after the error detection. As the latest detectable error is at *DC* falling edge and the comparator has a delay t_{Comp}, the FSM must change state at least t_{Comp} after *DC* low phase. This condition is the same as that of the data captured by additional latches DLL of Reg_in. Therefore, we can use *CLKRegin* falling edge to drive the FSM.

We also need to define conditions that decide which state of the FSM will be active after each *CLKRegin* falling edge. Let's consider again the last example where a fault occurs at (n-1)th CLK cycle. As the re-configuration happens only at nth CLK cycle, we need a signal which has a fixed logic value during fault-free operation but changes to the opposite value at nth period. The switching must take place before *CLKRegin* falling edge of nth period and finish earlier than the next *CLKRegin* falling edge. *CRegin* is such a signal. In fact, during fault-free operations, *CRegin* is at logic-0. When the error is detected at nth period, it switches to logic-1 before the transparent phase of Reg_in's DLLs which begins at *CLKRegin* falling edge. Then, it returns back to logic-0 before the next *CLKRegin* falling edge. The example above is illustrated in Figure 2.31.

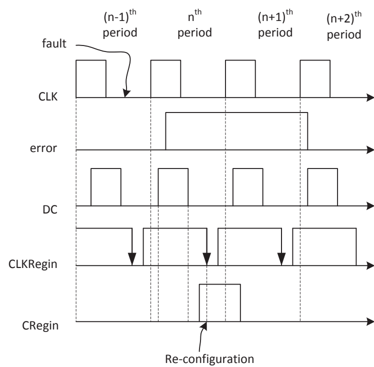


Figure 2.31 Clock and Condition Signals of the Finite State Machine

f1	f2	Configuration	Comments
0	1	1-2	LC1 and LC2 work together
1	0	2-3	LC2 and LC3 work together
0	0	3-1	LC3 and LC1 work together
1	1	"Final state"	FSM stops working

Table 2.4 Outputs of the Finite State Machine

As explained above, we will use *CLKRegin* falling edge to drive the FSM and *CRegin* as condition to define the next active state at each clock edge. If *CRegin* is at logic-0 at *CLKRegin* falling edge then the transition will correspond to dotted arrows ("No error") in the diagrams of Figure 2.30. Otherwise, it is defined by solid arrows ("Error detected") of these diagrams. Besides these two signals, we can also use the existing *resetControl* signal to define an asynchronous reset for the FSM during initialization of the architecture.

The fact that we use *CLKRegin* as the FSM clock signal imposes an additional constraint for this signal. From *CLKRegin* rising edge, the FSM outputs will be established after a delay t_{FSM} which represents the calculation time of the machine. Because these outputs must be ready before the next *CLK* rising edge where re-computation process starts, *CLKRegin* rising edge must happen at least t_{FSM} earlier than this moment. This condition is similar to the one described by (2.10) which guarantees that *CLKRegin* rising edge takes place at least t_{hold} before *CLK* rising edge. Therefore, it can be expressed by:

$$t_{bufDC} + t_{high} + t_{bufCLKRegin\ 1} + t_{bufCLKRegin\ 2} + t_{invCLKRegin} + t_{ORCLKRegin} + t_{FSM} < t_{period} \quad (2.13)$$

We have seen in the last sub-section that the FSM defines three possible configurations of the architecture as well as a final state when it stops working. Therefore, we will need two outputs bits *f1*

and *f2* to define these four situations. The values of these two bits and the corresponding configuration are presented in Table 2.4.

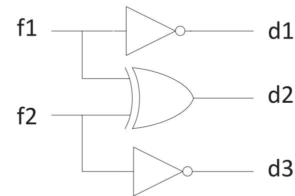
In order to obtain a concrete gate level implementation of the FSMs, we use VHDL descriptions of the modules and logic synthesis tools to generate the netlists. VHDL code for both FSMs of Figure 2.30 can be found in Appendix A.

Control signal of the input demultiplexer

To control the input demultiplexer Demux, we need to generate the control bits *d1*, *d2* and *d3* for its elementary module eDmux (table of Figure 2.25). To produce these bits, we use output signal *f1* and *f2* of the finite state machine (Table 2.4). Combining the two tables above, we obtain the truth table in Figure 2.32-a. Note that in the last row of this table, the “Final state” is reached by applying logic-0 to all control bits of Demux. This means that all CLs will be put on standby. The simple circuit in Figure 2.32-b realizes our proposed truth table using two inverters and one XOR gate.

f1	f2	Configuration	d1	d2	d3
0	1	LC1-LC2	1	1	0
1	0	LC2-LC3	0	1	1
0	0	LC3-LC1	1	0	1
1	1	“Final state”	0	0	0

a)Truth table



b) Circuit

Figure 2.32 Control Logic for Input Demultiplexer

Control signal of the output multiplexer

Depending on the method used for the output multiplexer Mux, we will need different control signals. For Method 1 (Figure 2.27) and Method 2 (Figure 2.28), we can see that control signals *m1* and *m2* correspond exactly to output *f1* and *f2* of the FSM (Table 2.4). Meanwhile, to use tri-state buffers for the Mux (Figure 2.29), we can use the same control bits *d1*, *d2* and *d3* of the input demultiplexer Demux.

2.5 Summary

In this chapter, we have developed a hybrid-fault tolerant architecture capable of detecting and correcting hard, soft and timing errors in combinational part of logic circuits. To obtain this objective, we proposed three architectures, corresponding to three phase of the hybrid fault-tolerance:

- Error detection architecture: Employing Duplication/Comparison CED technique (Information redundancy), this architecture detects both transient and permanent errors. To improve its error detection capability, a pseudo-dynamic comparator is proposed to deal with small glitches produced by soft and timing errors.
- Transient error correction architecture: Adding timing redundancy to the error detection architecture, this architecture corrects transient error by re-computation. Our method employs modified input register capable of keeping one previous input vector at each clock cycle. This vector will be used for re-computation when an error is detected. While having the same operation frequency as the original logic circuit, the architecture requires two clock cycles to tolerate each transient error.
- Permanent error correction architecture (hybrid fault-tolerant architecture): A third copy of the combinational logic CL is added to the previous architecture to tolerate both transient and permanent errors. This CL is kept on standby state during normal operation. When

permanent errors are detected in one of the two running CLs, a re-configuration is done to replace the faulty CL by the third one. This process is performed by additional input demultiplexer and output multiplexer. Different re-configuration schemes have been studied with corresponding Finite State Machine.

For each one of the architectures above, we have proposed detailed logic implementation for additional modules as well as their control logic and timing constraints. The use of these architectures will depend on application fields of the original logic circuit. Tradeoff between their fault-tolerance ability and their silicon area, power consumption costs must be considered. These tradeoffs of the hybrid fault-tolerant architecture will be studied in the next chapter.

Chapter 3

Evaluation of the Hybrid Fault-Tolerant Architecture

Chapter 3 Evaluation of the Hybrid Fault-Tolerant Architecture 57
3.1 Context 58
3.2 Architecture description 59
3.2.1 Hybrid Fault-Tolerant Architecture 60
3.2.2 TMR architecture 60
3.2.3 Discussion 62
3.3 Logic synthesis 62
3.3.1 Dynamic CMOS standard cell creation 63
3.3.2 Combinational logic synthesis 64
3.3.3 Redundant modules synthesis 66
3.3.4 Fault-tolerant architecture synthesis 72
3.4 Timing behavior of hybrid fault-tolerant architecture 73
3.4.1 Comparator simulation 74
3.4.2 Control logic simulation 76
3.4.3 Hybrid fault-tolerant architecture simulation 78
3.4.4 Discussion 80
3.5 Power simulation 80
3.6 Summary 81

In this chapter, we evaluate the hybrid fault-tolerant architecture presented in Chapter 2 using simulations with Electronic Design Automation (EDA) tools. The objective is to prove that this architecture can be used for pipeline-style logic circuits (Figure 1.18) regardless of their logic function, without any modification on the combinational part of circuits. This chapter also proves that implemented hybrid fault-tolerant architectures have the predicted fault tolerance ability with regard to transient and permanent faults. Besides, the hybrid solution is compared with TMR techniques in terms of area overhead and power consumption to highlight the pros and cons of each solution.

The chapter is organized as follows. The first section presents the concept of fault-tolerant architecture evaluation which is divided into four phases: RTL descriptions, logic synthesis, timing behavior and power consumption simulations. Then, in the four following sections we study in detail each of these phases. For each phase, we present different simulation steps and the required EDA tools. Important results of each simulation are then discussed to highlight the conformity of hybrid fault-tolerant methods with the objective defined in Chapter 2.

3.1 Context

Our evaluation of fault-tolerant architectures is based on simulations using EDA tools. The simulation process is divided into four phases:

- Architecture description: In this phase, we create a Register-Transfer Level (RTL) description of fault-tolerant architectures combining of logic circuits and redundant modules. These descriptions are written using Hardware Description Languages (HDL) such as Verilog or VHDL.
- Logic synthesis: This step consists of converting RTL descriptions into gate-level implementations (netlist). For this, we use logic synthesis tools that map abstract logic functions to concrete gates of standard cell libraries while optimizing silicon area and delays of the architecture. During synthesis, different timing constraints must be applied to guarantee correct operations of fault-tolerant architectures.
- Functional and timing behavior simulations: Using netlists generated in previous phases, we simulate functional and timing behaviors of architectures. Simulations are performed at transistor-level using SPICE or SPICE-like simulators. Different types of fault are injected during simulations to verify fault-tolerance capability of architectures.
- Power simulation: This phase is similar to the previous, except that no fault is injected. A set of random input vectors are run to compare the power consumption of different architectures. This comparison is performed by monitoring average and peak currents at power node VDD during simulation runs.

The complete simulation flow is illustrated in Figure 3.1 where we also specify different files needed in each phase. In the following subsections, we detail methods and simulation tools used for each simulation.

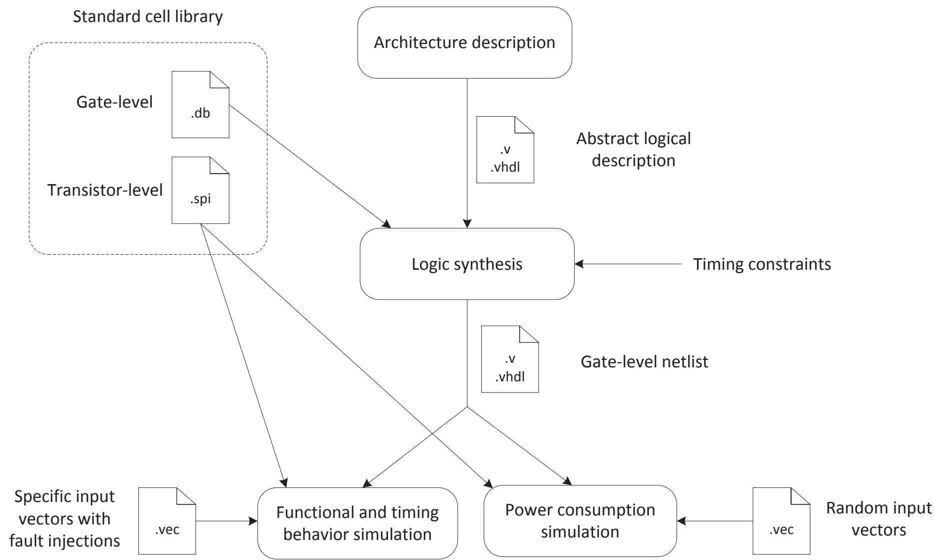


Figure 3.1 Fault-tolerant Architecture Evaluation Flow

3.2 Architecture description

For evaluation purpose, the hybrid-fault tolerant (Figure 2.24) and the TMR architecture (Figure 1.8) are used to tolerate transient and permanent faults in logic circuits whose structure is illustrated in Figure 1.18. Such logic circuits are created using combinational logic (CL) part of ISCAS'85 [ISCAS85] and ITC'99 [ITC99] benchmark circuits and input/output registers made of D flip-flops.

ISCAS'85 benchmark contains only combinational logics and hence no modification is needed. However, ITC'99 benchmark circuits are circuits. Thus, we must remove sequential elements (D flip-flops) from these circuits before using them. For each D flip-flop removed, one primary input nPI[n] and one primary output nPO[n] will be added to the resulting CL. The new primary input corresponds to output *Q* of the removed flip-flop while the new output corresponds to its input *D*. The combinational part extraction is illustrated in Figure 3.2.

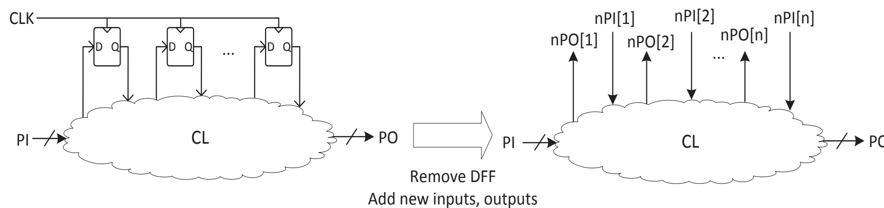


Figure 3.2 Combinational Logic Extraction from Sequential Circuits

Note that logic circuits created using extracted CLs and input/output registers do not have the same logic function as original benchmark circuits because all feedback signals have been removed. However, this does not affect our objective which consists of comparing different fault-tolerant architectures, regardless of CL functions.

In Appendix A, we illustrate an example of how Verilog netlists of CLs are created from original netlist of benchmark circuits.

To create RTL descriptions of fault-tolerant architectures, beside HDL codes of CLs, we also require descriptions of redundant modules as well as HDL codes of top-level modules. The creation of these modules is detailed in the following sub-sections.

3.2.1 Hybrid Fault-Tolerant Architecture

As illustrated in Figure 2.24, the hybrid fault-tolerant architecture employs three copies CL1, CL2 and CL3 of logic circuits' combinational part CL. For these modules, we can use the same CL Verilog descriptions extracted from benchmark circuit netlists (Figure 3.2). Note that using the same HDL code we can create different gate-level implementations during logic synthesis to simulate variability. This will be explained further in the next section.

Input register Reg_in, input demultiplexer Demux, output multiplexer Mux and output register Reg_out are created using identical sub-modules. Reg_in is made of nbInput identical modified flip-flops mDFF (Figure 2.16), where nbInput represents input number of CLs. Demux is made of nbInput elementary demultiplexers eDmux (Figure 2.25). Mux is made of nbOutput elementary multiplexers eMux (Figure 2.27, Figure 2.28 or Figure 2.29), where nbOutput represents output number of CLs. Reg_out is made of nbOutput D flip-flop. We use generic logic functions and behavioral Verilog to describe sub-modules, and then generate complete modules using nbInput and nbOutput as parameters.

HDL description of the control module is divided into three sub-modules:

- The first sub-module consists of control logic for transient error correction (Figure 2.23), which generates *DC*, *reset*, *CRegin* and *CLKRegin* signals from *CLK*, *resetControl* and *error*. To guarantee correct timing behavior of this sub-module, we do not use generic logic functions but specific gates from standard cell library. Note that this sub-module must be keep untouched during logic synthesis. This will be detailed further in the next section.
- The second sub-module is the Finite State Machine FSM. Two versions of FSM corresponding to state diagrams in Figure 2.30 can be described in behavioral Verilog. Output values *f1* and *f2* of this sub-module corresponding to different FSM states are shown in Table 2.4.
- The third sub-module receives FSM outputs and provides control signals for Demux and Mux modules that re-configure the hybrid fault-tolerant architecture in case of errors occurrence. It can be described with Verilog generic logic functions, using Figure 2.32.

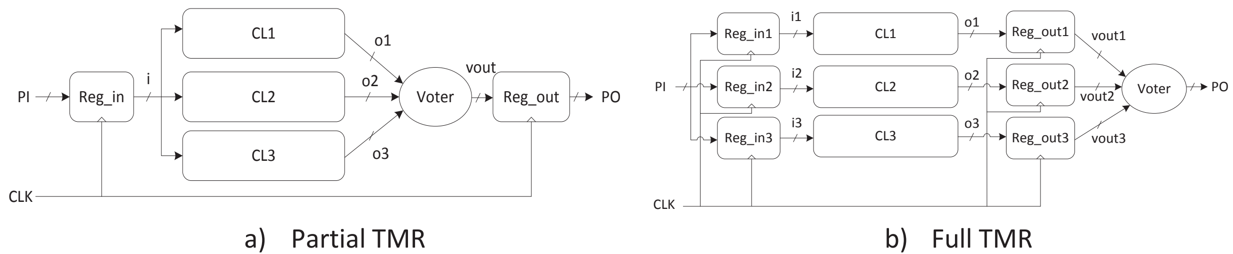
Unlike other modules, the pseudo-dynamic comparator (Figure 2.10) cannot be described in Verilog using generic logic functions. This is due to the dynamic characteristics of DOR gates. These gates must be created, added to the standard cell library and instanced together with generic XOR, OR gates in structural Verilog description of the dynamic comparator. The creation of DOR gates will be detailed further in this chapter.

After making all modules, we describe the complete hybrid fault-tolerant architecture (top-level) using structural Verilog. Examples of concrete HDL codes of each module and the complete architecture are presented in Appendix A.

3.2.2 TMR architecture

There are different methods to implement TMR architecture for logic circuits, depending on which part of this circuit is triplicated. In Figure 3.3, we present two TMR structures that will be compared with the hybrid fault-tolerant architecture. The first implementation (Partial TMR, Figure 3.3-a) consists of triplicating only combinational logic (CL) part of the logic circuit while the second one (Full TMR, Figure

3.3-b) requires triplications of both combinational and sequential parts. While having smaller area overhead, the Partial TMR solution introduces a module Voter in the data path of the structure. Consequently, this architecture requires longer *CLK* period which results in slower operation. Moreover, a timing error at the Voter will be captured by Reg_out without being tolerated. Problems above can be solved using Full TMR solutions. By putting the Voter after output registers, it preserves functioning speed of logic circuits while avoiding timing error. Furthermore, triplicated registers will be immune to single SEU because they can only affect at most one vector among *vout1*, *vout2* and *vout3*. Note that in Full TMR architecture, input registers are also triplicated so that timing errors caused by each register can also be tolerated.



a) Partial TMR b) Full TMR

Figure 3.3 TMR Structure for Logic Circuits

For comparison purpose, we use the same CLs as those previously created for the hybrid fault-tolerant architecture. Besides, we can also reuse output register Reg_out of this architecture for TMR solutions (Reg_out for Partial TMR and Reg_out1, Reg_out2, Reg_out3 for Full TMR).

Input registers of TMR architectures (Reg_in for Partial TMR and Reg_in1, Reg_in2, Reg_in3 for Full TMR) are simpler than that of the hybrid fault-tolerant structure. Only one D flip-flop is needed for each CL input in structural Verilog description of this module. Consequently, we can use the same HDL code as for output registers.

There are two types of voter that can be used for fault tolerance: bit-wise and word-wise voters. The first solution consists of independent bit-by-bit votes, while the second solution is based on vote of three whole input vectors. Table 3.1-a and b show examples that distinguish the two schemes. In both tables, the first three columns present input vectors of the voter while the forth column corresponds to its output vector. The word-wide voter has an additional output *error* which turns to logic-1 when and only when the vote is impossible (there are not a couple of identical input vectors). In both table, correct bits are in black while faulty bits are in red.

Input 1	Input 2	Input 3	Output
1100	1111	1100	1100
1100	1000	1010	1000

a) Bit-wise Voter

Input 1	Input 2	Input 3	Output	error
1100	1111	1100	1100	0
1100	1000	1010	xxxx	1

b) Word-wise Voter

Table 3.1 Bit-wise vs. Word-wise Voter

In Table 3.1-a and b, both voters work correctly in the first two cases where only one input vector is erroneous. In the second case where both *Input2* and *Input3* are faulty, the word-wise voter does not

found two identical inputs and thus, raises an *error* signal. Meanwhile, the bit-wise voter continues voting and answer a value. As the second bit of *Input2* and *Input3* are identical faulty logic-0, the second bit of *Output* is also faulty logic-0.

In [MIT00b], authors demonstrated that the word-wise voter is more suitable for fault-tolerance because it increases data integrity of TMR structures. Consequently, we will use this voter for our TMR architectures. Verilog description of such voter can be created using generic logic functions and the structure illustrated in Figure 3.4.

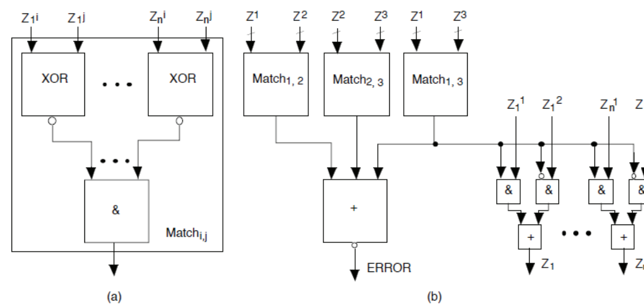


Figure 3.4 Word-Voter Architecture, Source: [MIT00b]

As for the hybrid fault-tolerant architecture, top-level descriptions of both TMR architectures are created using structural Verilog. Examples of concrete HDL codes of the modules and the complete architecture are presented in Appendix A.

3.2.3 Discussion

In this section, we have create RTL descriptions of hybrid fault-tolerant as well as Partial and Full TMR architectures for combinational part of ISCAS'85 and ITC'99 benchmark circuits. HDL codes of different modules and complete architectures are presented in Appendix A. These codes only specify logic functions and connections between modules regardless of the technology used to implement them. Area, timing and power consumption information of architectures can only be extracted after logic synthesis phase which will be detailed in the next section.

3.3 Logic synthesis

Logic synthesis is a process where we translate RTL descriptions of digital circuits and systems to concrete implementations using specific gates. For this, we require standard cell libraries which contain information about logic function, silicon area, power consumption and different delays of each gate. A synthesis tool is then used to translate HDL codes to gate-level netlist of circuits.

In the scope of this thesis, we use commercial synthesis tool Synopsys Design Compiler® [DCSYS] and the Nangate 45nm Open Cell Library (NOCL, [NOCL]) which contains standard cells of a 45nm technology specified by Predictive Technology Model (PTM, [PTM]).

Our logic synthesis flow for fault-tolerant architectures is divided into 4 steps:

- Dynamic CMOS standard cells creation: In NOCL, there are only static CMOS gates. However, to create pseudo-dynamic comparator (Figure 2.10) of the hybrid fault-tolerant architecture,

we require dynamic CMOS OR gates (Figure 2.8). These gates must be created using full custom design style, characterized, and added to the library.

- Combinational logic synthesis: To guarantee correct comparison of different architectures, the same gate-level netlists of CL must be use for the hybrid fault-tolerant and TMR structures. Consequently, this module must be synthesized independently, and then kept untouched during subsequent synthesis steps.
- Redundant module synthesis: In this step, we synthesize different versions of redundant modules of the hybrid fault-tolerant architecture. Then, we compare these versions in terms of silicon area and delay in order to decide which implementation should be used for the fault-tolerant architecture.
- Fault-tolerant architecture synthesis: This step consists of synthesizing the complete fault-tolerant architectures and comparing their silicon area.

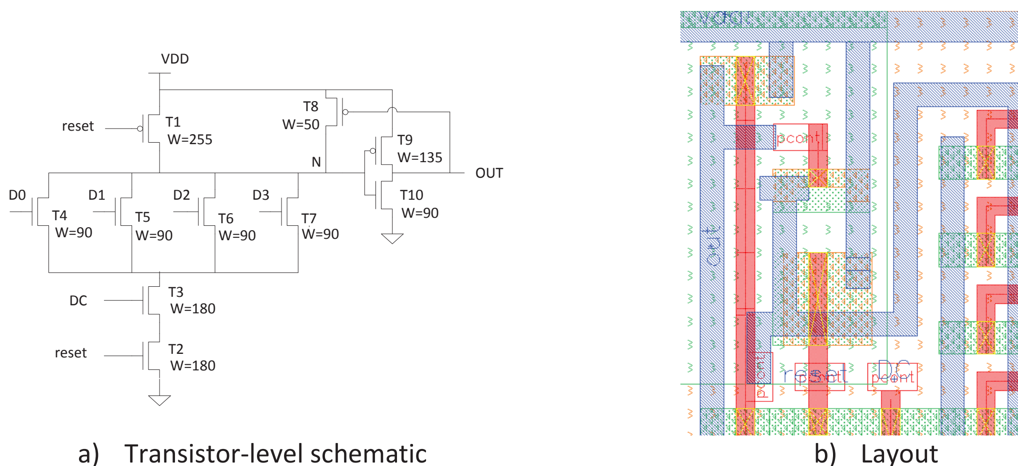
In the following sub-section, we detailed each step together with synthesis results.

3.3.1 Dynamic CMOS standard cell creation

As detailed in Chapter 2, we need dynamic OR gates to create pseudo-dynamic comparators. A 4-input dynamic OR gate (DOR4_X1) is proposed in [TRA12]. It is designed according to design and electrical rules of FreePDK process design kit [PDK], which was also used to create NOCL. DOR4_X1 transistors are also sized according to typical transistor dimensions of NOCL gates. These dimensions are illustrated in Figure 3.5-a. For each transistor, W represents its channel width in nanometer. All transistors have minimum channel length of 50nm.

Figure 3.5-b shows the layout of DOR4_X1 standard cell. The transistors from Figure 3.5-a are placed as follows: The small N-well in the upper left corner contains pull-up transistor T9 of the inverter, feedback transistor T8 as well as charge transistor T1; The Pwell at the bottom holds pull-down transistor T10 of the inverter together with discharge transistors T2 and T3; The right hand side implements NMOS transistors T4-T7 used for the inputs. With this design, DOR4_X1 has the same silicon area as a static 4-input OR gates (OR4_X1) from NOCL.

The layout in Figure 3.5-b is characterized to extract parasite parameters as well as area, power consumption and delay information. DOR_X1 can then be used in logic synthesis as other standard cells from NOCL.



a) Transistor-level schematic b) Layout

Figure 3.5 4-input dynamic OR gate DOR4_X1

Note that in the scope of this thesis, we use 4-logic-input DOR gates because the highest fan-in static OR gate in NOCL also has four inputs. However, higher fan-in DORs can also be used to optimize silicon area and power consumption of pseudo-dynamic comparators.

3.3.2 Combinational logic synthesis

During logic synthesis, different conditions must be specified to Design Compiler to guarantee correct timing behavior of combinational logic CLs used in stand-alone logic circuits as well as in fault-tolerant architectures.

Timing constraints in logic circuits

Delays of combinational logic CLs must respect both setup and hold time constraints of logic circuits' output register (Figure 1.18). While setup time violation can be easily avoided by increasing operational clock period, hold time violations require more effort to be corrected during synthesis.

Small delay paths (short paths) between CL inputs and outputs are responsible for hold time violation in output register of logic circuits. An example of short path is feedthroughs which are direct connections between CL inputs and outputs. In our CLs, feedthroughs may come from both original benchmark circuits' structure and the combinational part extraction. Let us consider the example of a logic circuit illustrated in Figure 3.6 in which a feedback signal connects output Q of the nth flip-flop to input D of the mth flip-flop (note that they may be the same flip-flop). After combinational logic extraction process (Figure 3.2), a feedthrough path is created between *nPI[n]* and *nPO[m]*.

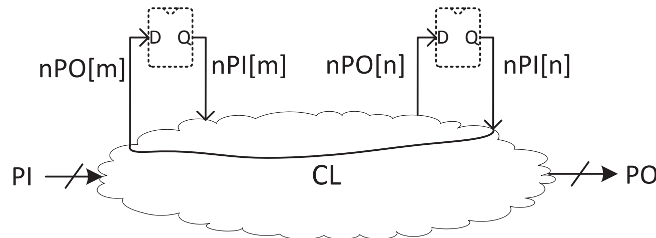


Figure 3.6 Feedthrough Path Created by Combinational Part Extraction

Hold time violations in logic circuits can be handled automatically by EDA tools. In our case, Design Compiler does this by inserting buffers or resizing gates to increase CL short path delays.

Timing constraints in fault-tolerant architectures

Beside hold time violations, CL short paths must also be dealt with to guarantee correct function of the pseudo-dynamic comparator in hybrid fault-tolerant architectures (Figure 2.24). In Chapter 2, we have seen that CL outputs must be held stable during the comparison window of this comparator (Figure 2.12). This condition is specified by equation (2.7), which defines the minimum CL short path delay.

Similar to previous timing constraints, (2.7) can also be handled during logic synthesis by specifying a minimum delay between all inputs and outputs of CLs using the command set_min_delay of Design Compiler. Note that CL short path fixing may lead to higher silicon area and power consumption of CL due to buffer insertions. However, it has been proven that these overhead are negligible for large circuits [ERN03]. Besides, we can also reduce the duty cycle of *CLK* signal in the hybrid fault-tolerant architecture to reduce the minimum delay defined by (2.7).

Logic synthesis results

As discussed above, during logic synthesis of CLs, minimum delay constraints are applied. For all circuits, this parameter is set at 2ns. These chosen values correspond to the 1.5ns hold time of typical D flip-flop DFF_X1 from NOCL library.

Table 3.2 shows logic synthesis result for CL part of biggest ISCAS'85 and ITC'99 benchmark circuits. The first three columns present CL characteristics: name of original benchmark circuit, input and output number of combinational part. The fourth column corresponds to area in square micrometer of synthesized CL with applied timing constraints. The last column presents maximum delay in nanosecond between inputs and outputs of CL.

Circuit	Nb. Input	Nb. Output	Area CL (μm^2)	Delay max (ns)
c5315	178	123	5312	11.19
c6288	32	32	2928	8.18
c7552	206	107	4798	8.67
b14s	278	300	15000	9.00
b15s	486	520	27189	11.57
b20s	523	513	28096	11.03
b21s	523	513	27956	10.26
b22s	768	758	41729	9.92

Table 3.2 Area and Delay of Synthesized Combinational Logic

Discussion

During logic synthesis of CLs, we have chosen important timing margins for short path delays, which lead to high area overhead between CLs synthesized with and without constraints. There are two reasons for this choice. First, due to important number of gates, our transistor-level power evaluations are done at low SPICE-level in order to reduce simulation time. Consequently, large timing margins are needed to prevent simulation errors due to the lost of precision. Second, all evaluations are done at front-end stages of digital design flow, before place and route. Therefore, wire delays between gates are not taken into account. This means short paths are under-estimated during synthesis, and hence, more buffers are required to compensate the difference.

The important area overhead induced by short path correction is mainly due to our choice of benchmark circuits, and does not represent area overhead of fault-tolerant architecture compared to stand-alone logic circuits. In fact, in [ERN03] and [DAS09], similar minimum delay constraints are performed for state-of-the-art processors with negligible area and power overhead.

Redundant combinational logics

An important phenomenon that must be taken into account during logic synthesis of redundant CLs is variability. We have seen in previous chapters that this phenomenon may result in different timing characteristics of identical CL modules in the same fault-tolerant architecture. These variations cause CLs' outputs to differ during transient phase of computation, and therefore affect power consumption of the pseudo-dynamic comparator and the voter. To simulate this phenomenon, various minimum delays are defined for different CL copies in a fault-tolerant architecture so that buffers are inserted unequally, and result in variations of CL timing characteristics.

Note that the same set of timing constraints is used to synthesize CLs in TMR and the hybrid fault-tolerant architectures to guarantee fair comparisons. For all architectures, the minimum path delay of three CL copies CL1, CL2 and CL3 (Figure 2.24, Figure 3.3) are set at 2ns, 2.2ns and 2.1ns respectively.

Table 3.3 shows synthesized area of combinational logics with different timing constraints. The first column presents original benchmark circuits' name while the three next column detail silicon area of CL1, CL2 and CL3 in square micrometer. The fifth column shows average area of the three CL copies used in fault-tolerant architectures. The last column details maximum delay in nanosecond of CLs, which define operating frequency of fault-tolerant architectures.

Circuit	Area CL1 (μm^2)	Area CL2 (μm^2)	Area CL3 (μm^2)	CL (μm^2)	Delay max (ns)
c5315	5312	5737	5579	5543	11.88
c6288	2928	3090	3004	3007	8.18
c7552	4798	5246	5005	5016	9.44
b14s	15000	16969	15889	15953	10.59
b15s	27189	29644	28227	28353	12.11
b20s	28096	30322	28699	29039	11.48
b21s	27956	30325	29020	29100	10.42
b22s	41729	44520	42977	43075	11.54

Table 3.3 Area and Delay of Synthesized Redundant Combinational Logics

3.3.3 Redundant modules synthesis

In this subsection, we detailed logic synthesis results of redundant modules. Different versions of these modules, whose structure are presented in Chapter 2, are compared in term of area overhead and delay.

Registers

We have seen in Chapter 2 that there are two types of register used in fault-tolerant architectures. The first type consists of using one D flip-flop for each input/output bits. Output register Reg_out of hybrid fault-tolerant architectures (Figure 2.24) as well as input registers Reg_in, Reg_in1, Reg_in2, Reg_in3 and output registers Reg_out, Reg_out1, Reg_out2, Reg_out3 of TMR architectures (Figure 3.3) belong to this type. The second type of register is made of modified D flip-flops (Figure 2.16) that enable re-computation possibility in hybrid fault-tolerant architectures.

In Table 3.4, we present synthesized input and output registers' area. The first three columns correspond to characteristics of original logic circuits: name, CL input and output numbers. The fourth column shows input register area of hybrid fault tolerant architectures while the fifth column presents input register area of TMR architectures. The last column details output register area of all architectures. All synthesized areas are expressed in square micrometer.

We can see in Table 3.4 that due to additional latches and multiplexers, second type registers (Reg_in of hybrid fault-tolerant architectures) are twice larger than first type registers (Reg_in of TMR architectures) of the same input number.

Circuit	Nb. Input	Nb. Output	Reg_in Hybrid (μm^2)	Reg_in TMR (μm^2)	Reg_out (μm^2)
c5315	178	123	1623	805	556
c6288	32	32	292	145	145
c7552	206	107	1879	931	484
b14s	278	300	2539	1257	1357
b15s	486	520	4440	2198	2351
b20s	523	513	4769	2365	2320
b21s	523	513	4769	2365	2320
b22s	768	758	7008	3473	3428

Table 3.4 Area of Synthesized Input and Output Registers

Input demultiplexer

Input demultiplexers Demux of hybrid fault-tolerant architectures (Figure 2.24) are made of elementary demultiplexers eDmux (Figure 2.25), one for each CL input bit. Structure of eDmux after logic synthesis is presented in Figure 3.7. Compared to Figure 2.25, AND gates are replaced by NOR gates and inverters to reduce eDmux area. Note that inverters that generate *cd1*, *cd2* and *cd3* from *d1*, *d2* and *d3* signals can be shared between eDmuxes of a Demux. To enhance drive strength of *cd1*, *cd2* and *cd3* signals provided by the shared logic, additional buffers are automatically inserted by Design Compiler for high input number CL.

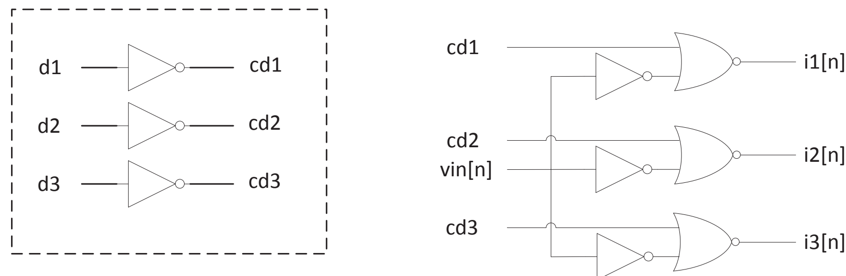


Figure 3.7 Synthesized Elementary Input Demultiplexer

Beside silicon area, delay of input demultiplexer Demux is also an important factor. There are two types of Demux delay that have influences on operations of hybrid fault-tolerant architectures:

- The first type is IN/OUT delay that consists of delay between data inputs (*vin*) and outputs *i1*, *i2*, *i3* of this module (Figure 2.24). For each eDmux in Figure 3.7, it corresponds to the delay between *vin[n]* and *i1[n]*, *i2[n]*, *i3[n]*. This type of delay increases data path delay between input and output registers of hybrid fault-tolerant architectures. Consequently, it affects computation speed of architectures during fault-free operation.
- The second type is SELECT/OUT delay that exists between control bits *d1*, *d2*, *d3* and outputs of Demux. Beside inverters and NOR gates between *d1*, *d2*, *d3* and *i1[n]*, *i2[n]*, *i3[n]* (Figure 3.7), inserted buffers at outputs of shared logic also contribute to this delay. This type of delay does not exist during fault-free operations when *d1*, *d2* and *d3* are stable. Consequently, it does not influence computation speed of hybrid fault-tolerant architectures.

However, it defines the time needed for re-configuration of the architectures when errors occur.

Logic synthesis results of Demuxes are presented in Table 3.5. The first and second columns of Table 3.5 correspond to original logic circuit name and CL input number. The third column shows synthesized area in square micrometer of Demux. The two last columns present IN/OUT and SELECT/OUT delays in picoseconds.

Circuit	Nb. Input	Demux (μm^2)	IN/OUT (ps)	SELECT/OUT (ps)
c5315	178	561	70	340
c6288	32	95	70	160
c7552	206	650	70	390
b14s	278	892	70	280
b15s	486	1555	70	380
b20s	523	1673	70	390
b21s	523	1673	70	390
b22s	768	2469	70	320

Table 3.5 Area and Delays of Synthesized Input Demultiplexer

In Table 3.5, synthesis results have proven our hypothesis that IN/OUT delay is negligible compared to maximum CL delay (Table 3.2). This allows hybrid fault-tolerant architectures to operate at almost the same frequency as standalone logic circuits. We can also observe that SELECT/OUT delay varies with CL input number, due to buffer insertion. However, this delay remains lower than 0.5ns. In further sections, we will prove that this value satisfies different timing constraints that guarantee correct re-configurations of hybrid fault-tolerant architectures before re-computation phase.

Output multiplexer

In Chapter 2, we have seen that there are three methods to implement output multiplexer Mux of the hybrid fault-tolerant architecture (Figure 2.24), corresponding to three elementary output multiplexers eMux presented in Figure 2.27 (Method 1), Figure 2.28 (Method 2) and Figure 2.29 (Method 3).

Table 3.6 compares synthesized area of Muxes created using the three methods. Name and CL output number of original logic circuits are presented in the first two columns. The three next columns show areas the three Mux versions. The two last columns detail area overhead of Method 2 and Method 3 compared to Method 1. Both overheads are expressed in percentage of Method 1 Mux area.

In Table 3.6, we can see that Method 1 provides significantly smaller Muxes area compared to other methods. Area overhead of Method 3 compared to Method 1 is 157% for all circuits while overhead of Method 2 compared to Method 1 is 73% for largest CL output numbers.

Note that the constant ratio between area of Method 3 and Method 1 Muxes is due to the fact that for each method, the area of synthesized Muxes is proportional to CL output number. This is because the structures in Figure 2.27 (Method 1) and Figure 2.29 (Method 3) are optimized. No logic sharing among eMuxes is possible and hence, total Mux area is equal to CL output number time the area of an eMux.

Circuit	Nb. Output	Area (µm^2)			Area overhead	
		Method 1	Method 2	Method 3	2/1	3/1
c5315	123	458	789	1178	72%	157%
c6288	32	119	195	306	64%	157%
c7552	107	398	675	1025	70%	157%
b14s	300	1117	1921	2873	72%	157%
b15s	520	1936	3345	4979	73%	157%
b20s	513	1910	3299	4912	73%	157%
b21s	513	1910	3299	4912	73%	157%
b22s	758	2823	4879	7259	73%	157%

Table 3.6 Area of Synthesized Output Multiplexer

For Method 2, Mux area does not vary linearly with CL output number. In fact, after logic optimization, the structure of eMux in Figure 2.28 is transformed into that structure in Figure 3.8. In a Mux, the logic part that generates *cm1*, *cm2* and *cm3* signals from *m1* and *m2* signals can be shared among eMuxes. However, when CL output number increases, buffers are inserted to preserve drive strength of *cm1*, *cm2* and *cm3* signals. This explains the non linear variation of Method 2 Mux area. Note that when CL output number is large, area of shared logic and inserted buffers are negligible compared to total Mux area. For this reason, we have the same area overhear between Method 2 and Method 1 Muxes for high fan-out CLs.

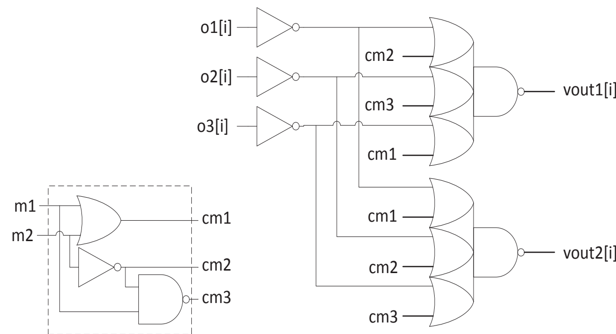


Figure 3.8 Synthesized Elementary Output Multiplexer – Method 2

Beside area overhead, delays between inputs and outputs of Muxes are also an important selection criterion. Similar to input demultiplexer Demux, there are two types of Mux delay that affect functions of hybrid fault-tolerant architectures. The first type is delays between data input *o1*, *o2*, *o3* and output *vout1*, *vout2* of Mux (Figure 2.24). These IN/OUT delays affect hybrid fault-tolerant architectures during fault-free operations. They increase data path between input and output registers and thus, reduce functional *CLK* frequency of the architectures. The second type consists of delays between selection bits and outputs of Mux. These SELECT/OUT delays increase switching configuration time of hybrid fault-tolerant architectures when errors occur.

Table 3.7 shows maximum delays estimated by Design Compiler (in picoseconds) for synthesized Muxes for c6288, b14s and b22s benchmark circuits. In this table, the second line presents name and CL output number of logic circuits. The three next lines correspond to different delays of Muxes created

with Method 1, Method 2 and Method 3. For each of these lines, second, third and fourth columns show IN/OUT delay while the last three columns present SELECT/OUT delay.

	IN/OUT (ps)			**SELECT/OUT (ps)**		
Circuit	c6288 (32)	b14s (300)	b22s (758)	c6288 (32)	b14s (300)	b22s (758)
Method 1	100	100	100	120	120	120
Method 2	130	130	130	560	810	760
Method 3	90	90	90	280	280	280

Table 3.7 Delays of Synthesized Output Multiplexer

From Table 3.7, we can see that IN/OUT delays of all methods do not vary with CL output number because there is no logic sharing between data inputs and outputs of eMuxes. IN/OUT delay of Method 3 (90ps) is smaller than that of Method 1 (100ps) and Method 3 (130ps). However, differences between these delays are negligible compared to maximum delays of CLs which are at order of 10ns.

Table 3.7 also shows that SELECT/OUT delay of Method 1 and Method 3 are constant while delay of Method 2 varies with CL output number. This is due to buffers that are inserted to increase drive strength of *cm1*, *cm2* and *cm3* signals in Method 2 (Figure 3.8). For this reason, SELECT/OUT delay of this method is significantly higher than that of Method 1 and Method 3. However, this comparison does not taken into account buffers that need to be inserted to enhance drive strength of *m1*, *m2* and *m3* signals when using Method 1 and Method 3. Consequently, to keep a fair comparison, we consider SELECT/OUT delay of Method 2 without inserted buffers. After Design Compiler, this delay is only 190ps.

Although Method 1 is the most cost effective and provides fastest Muxes, we have seen in Chapter 2 that it limits the performance of the hybrid fault-tolerant architecture with regard to timing errors. Between two other methods, Method2 has smaller area overhead and lower SELECT/OUT delay. For this reason, despite its delay which is slightly higher IN/OUT compared to Method 3, in subsequent simulation, Method 2 Muxes are used for hybrid fault-tolerant architectures.

Comparator

Using the 4-input dynamic OR gate DOR4_X1 presented in Figure 3.5, we create pseudo-dynamic comparators for hybrid fault-tolerant architectures. Silicon area of these comparators is compared with that of traditional static comparators whose structure is shown in Figure 2.5.

Table 3.8 shows area comparison between synthesized comparators. The first two columns show CL characteristics: original benchmark circuit name and output number. The two third and fourth columns detail synthesized area in square micrometer of static and pseudo-dynamic comparators used to detect errors at CL outputs. The last column presents area overhead of pseudo-dynamic comparator compared to static comparators. This overhead is measured in percentage of static comparators' area.

Although DOR4_X1 is designed with the same area cost as a 4-input static OR (OR4_X1) from NOCL library, pseudo-dynamic comparators still require some area overhead compared to static comparators. This is mainly due to the fact that DOR4_X1 has a dynamic logic function that cannot be optimized during logic synthesis.

Circuit	Nb. Output	Static (μm^2)	Pseudo-dynamic (μm^2)	Area overhead
c5315	123	251	260	4%
c6288	32	65	67	3%
c7552	107	219	227	4%
b14s	300	622	638	3%
b15s	520	1062	1096	3%
b20s	513	1046	1082	3%
b21s	513	1046	1082	3%
b22s	758	1551	1608	4%

Table 3.8 Area of Synthesized Comparator

Although they require small area overhead of 3%-4%, pseudo-dynamic comparators have higher error detection capability and lower power consumption compared to static comparators. This will be proven in subsequent sections. Moreover, since both comparator areas are negligible with regard to CLs' area (Table 3.3), this overhead is insignificant compared to total area of fault-tolerant architectures.

Control logic

As stated in previous section, control logic module of hybrid fault-tolerant architectures is divided into three sub-modules:

- The first sub-module generates *DC*, *reset*, *CRegin* and *CLKRegin* signals that control input register and pseudo-dynamic comparator of hybrid fault-tolerant architectures. To guarantee correct timing between these signals, instead of using Design Compiler to do logic synthesis, we simply replace generic logic gate in Figure 2.23 by concrete instance of standard cells from NOCL library. Different cells with various drive strength are used to modify delays between signals. Buffers are also inserted for the same purpose. During logic synthesis of hybrid fault-tolerant architectures in further step, this sub-module must be keep untouched to avoid any area optimization that may modify its timing characteristics. In Appendix A, we provide the implementation that has been proven work for our evaluation logic circuits which have similar CL delays. The verification, which is performed using SPICE-like simulation of complete fault-tolerant architecture with fault injection, is detailed in further sections. The total area of the mentioned sub-module is $59\mu m^2$.
- The second sub-module corresponds to the FSM that defines configuration of hybrid fault-tolerant architectures (Figure 2.24). Output value of this sub-module is presented in Table 2.4. Unlike other redundant modules, FSM implementations do not depend on original logic circuits. Two versions of FSM correspond to state diagrams in Figure 2.30 are synthesized. Area (in square micrometer) and maximum delay (in picoseconds) of synthesized FSMs are presented in Table 3.9. We can see that the FSMs have similar silicon areas which are both negligible compared to CL areas (Table 3.3). Their delays of about 500ps contribute to the re-configuration time of hybrid fault-tolerant architectures in case of error occurrence. It will be proven in further sections that these delays satisfy different time constraints established in Chapter 2. In subsequent simulation, we use FSM2 as case study to control hybrid fault-tolerant architectures.

	Area (μm^2)	Delay(ps)
FSM 1	93.896	500
FSM 2	101.080	560

Table 3.9 Area and Delay of Synthesized Finite State Machine

- The third sub-module generates control signals d1, d2, d3 of input demultiplexer Demux and m1, m2 of output multiplexer Mux from outputs f1, f2 of the second sub-module. As we use the method in Figure 2.28 to implement Mux, f1 and f2 can directly be used as control signals m1 and m2 respectively. Control logic circuit for d1, d2 and d3 (Figure 2.32) is synthesized by Design Compiler. The resulted logic has a silicon area of 3μm^2 and a maximum delay of 70ps which are both negligible compared to area and delay of CLs.

Voter

We have seen that there are two possible voter implementations for TMR architectures. Synthesized area and maximum delay of both implementations are presented in Table 3.10 for different logic circuits. In this table, the first two columns show name and CL output number of original logic circuits. The third column presents CLs' average area in fault-tolerant architectures (Table 3.3). The three next columns detail area of synthesized bit-wise and word-wise voters as well as area overhead of word-wise solutions. This overhead is expressed in percentage of bit-wise voters' area. The three final columns correspond to maximum delay of bit-wise and word-wise voter together with delay overhead of word-wise implementations. The delay overhead is also calculated in percentage of bit wise-voters' maximum delay.

Circuit	Nb. Output	CL (μm^2)	Area			Delay		
			Bit-wise (μm^2)	Word-wise (μm^2)	Overhead	Bit-wise (ps)	Word-wise (ps)	Overhead
c5315	123	5543	393	984	150%	90	740	722%
c6288	32	3007	102	255	150%	90	610	578%
c7552	107	5016	341	858	152%	90	740	722%
b14s	300	15953	958	2447	155%	90	920	922%
b15s	520	28353	1660	4199	153%	90	1170	1200%
b20s	513	29039	1637	4139	153%	90	1130	1156%
b21s	513	29100	1637	4139	153%	90	1130	1156%
b22s	758	43075	2419	6152	154%	90	1190	1222%

Table 3.10 Area of Synthesized Voters

From Table 3.10, we can see that word-wise voters are significantly larger and slower than bit-wise solutions. However, while voter areas remain small compared to CL areas, additional delays induced by voters may have important influences on *CLK* frequency of TMR architectures. In fact, since CL maximum delays are at orders of 10ns (Table 3.2), word-wise voters may cause TMR architectures to operate at 7%-10% lower speed compared to standalone logic circuits. Despite these drawbacks, in further simulations, we will use word-wise voter which offer higher reliability for TMR architectures [MIT00b].

3.3.4 Fault-tolerant architecture synthesis

Using implementation of different modules established in previous sub-section, we synthesize fault-tolerant architectures for ISCAS'85 and ITC'99 benchmark circuits. Synthesis results are presented in Table 3.11.

In Table 3.11, the first three columns detail characteristics of original benchmark circuits: name, CL input and output numbers. The fourth column shows average area of three CL copies using in fault-tolerant architectures (Table 3.3). The fifth column presents area of hybrid fault-tolerant architectures while the sixth column details area of Partial TMR solutions. Area overhead of hybrid fault-tolerant technique compared to Partial TMR is shown in the next column. This overhead is expressed in percentage of Partial TMR architectures' area. The two last columns correspond to area of Full TMR architectures and area reduction of hybrid fault-tolerant structures compared to them. This reduction is calculated in percentage of Full TMR architectures' area.

Circuit	Nb. Input	Nb. Output	CL (μm^2)	Hybrid (μm^2)	Partial TMR		Full TMR	
					Area (μm^2)	Overhead	Area (μm^2)	Reduction
c5315	178	123	5543	20322	18973	7,1%	21695	6,3%
c6288	32	32	3007	9931	9567	3,8%	10147	2,1%
c7552	206	107	5016	18886	17322	9,0%	20152	6,3%
b14s	278	300	15953	54776	52919	3,5%	58147	5,8%
b15s	486	520	28353	96960	93808	3,4%	102906	5,8%
b20s	523	513	29039	99377	95941	3,6%	105311	5,6%
b21s	523	513	29100	99561	96125	3,6%	105495	5,6%
b22s	768	758	43075	147234	142279	3,5%	156081	5,7%

Table 3.11 Area of Synthesized Fault-Tolerant Architectures

We can observe that area overheads of hybrid fault-tolerant architectures compared to Partial TMR solutions are negligible for largest ITC'99 benchmark circuits (about 3.5%). This is not the case for c5315 and c7552 circuits of ISCAS'85 benchmarks. This can be explained by additional area of input register, input demultiplexer and output multiplexer in hybrid fault-tolerant architectures which are important compared to CLs' size.

Compared to Full TMR architectures, the area reduction realized using hybrid fault-tolerant technique is of about 6% except for c6288 which has small CL input and output numbers.

3.4 Timing behavior of hybrid fault-tolerant architecture

The objective of this section is to verify the correct function of synthesized hybrid fault-tolerant architectures. It is divided into three parts:

- Comparator simulation: This part consists of using SPICE simulations to study glitch detection capability of the pseudo-dynamic comparator. Comparisons with static comparator are performed to justify our choice of this module in hybrid fault-tolerant architectures.
- Control logic simulation: This part verifies the correct timing of different control signals generated by synthesized logic modules. Due to the large size of this module, SPICE-like simulations performed by Synopsys NanoSim [NNSIM] are used to reduce simulation time.
- Hybrid fault-tolerant architecture simulation: In this part, we use SPICE-like simulations to verify the correctness of hybrid fault-tolerant architectures' operations in case of error occurrence. During these simulations, glitches are injected at CL outputs to simulate different type of faults.

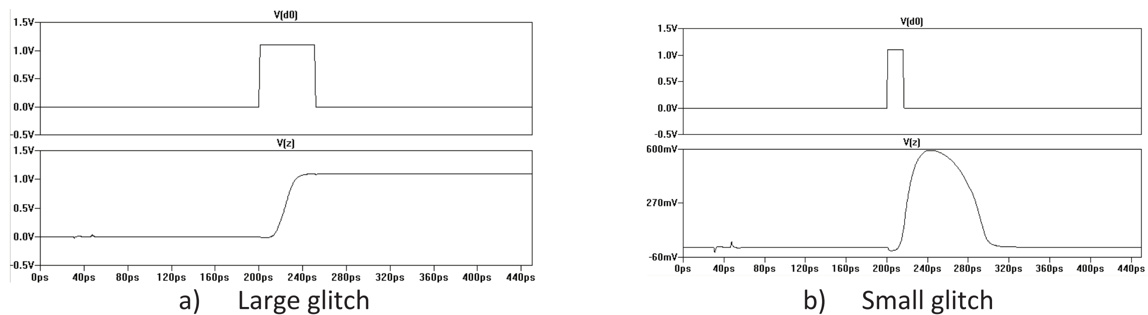
3.4.1 Comparator simulation

Dynamic OR

We have seen in Chapter 2 that the sensitivity of DOR gates with regard to glitches is the key factor that determines detection capability of pseudo-dynamic comparators. To evaluate this sensitivity, we perform SPICE simulations of the DOR4_X1 gate presented in Figure 3.5. These simulations are performed by Linear Technology LTSpice [LTSPICE].

In the simulations, the DOR4_X1 gate is reset once at t=50ps (logic-0 at *reset* input). Its evaluation phase (logic-1 at *DC* input) is set between t=100ps and t=350ps. As the four inputs *D0*, *D1*, *D2* and *D3* of DOR4_X1 are symmetric, we only apply glitches at *D0* while the others are kept at logic-0 during the entire simulation time. These 0-1-0 glitches are applied at t=200ps (during the evaluation window).

Figure 3.9 shows simulation results for two glitches with different durations: Δ1=50ps (large glitch, Figure 3.9-a) and Δ2=15ps (small glitch, Figure 3.9-b). Waveforms of input *D0* and output *Z* are presented as V(d0) and V(z), respectively. We can observe that when a large glitch appears at *D0*, *Z* turns from logic-0 to logic-1. After a transient phase which takes about 35ps, *Z* remains at high level. In the case of a small glitch, *Z* also changes value but then returns to logic-0 when the glitch disappears. This can be explained by the fact that the glitch duration is too small for *Z* to completely switch to logic-1. Consequently, it is pulled down to logic-0 by the feedback transistor T8 (Figure 3.5) when *D0* has returned to logic-0. In fact, our simulations show that DOR4_X1 gate can detect a minimum glitch size of 16ps.



a) Large glitch b) Small glitch

Figure 3.9 Glitches Detection Capability of DOR Gate

Another factor that may affect DOR4_X1 function is the reset duration necessary to pull down its output *Z* to logic-0 after error detection. Figure 3.10 presents our simulation results where a reset of 50ps is applied when *Z* is at stable logic-1. We can observe that this duration is enough to reset DOR4_X1. In further simulations, we will prove that the synthesized control logic is able to provide such *reset* signal.

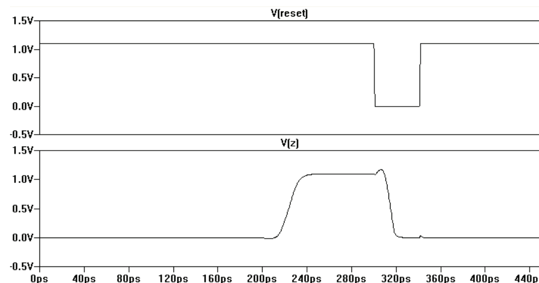


Figure 3.10 Reset of DOR Gate

Pseudo-dynamic versus static comparator

To compare error detection capability of pseudo-dynamic (Figure 2.10) and static comparators (Figure 2.5), we use SPICE simulations for two comparators of two 4-input vectors. The Local comparison stage of both comparators is made of four 2-input XOR gates XOR2_X1 from NOCL. In addition, a 4-input OR gate OR4_X1 from NOCL and a 4-input DOR gate DOR4_X1 created according to Figure 3.5 are used as Global comparison stage of the static and pseudo-dynamic comparators.

In the simulations, the pseudo-dynamic comparator is reset at t_0=50ps and t_4=400ps (logic-0 at *reset* input of DOR4_X1 gates) while its comparison phase is set between t_1=100ps and t_3=350ps (logic-1 at *DC* input of DOR4_X1 gates). Both comparators are used to compare two input vectors *A[3:0]* and *B[3:0]*. Input pair (*A0*, *B0*) is kept at logic-1 while the others are at logic-0. In addition, 0-1-0 glitches are applied at *A1* at t_2= 200ps to simulate error occurrence.

Figure 3.11 shows simulation results for a glitch of duration Δ=50ps. Waveforms of *reset* and *DC* inputs of the pseudo-dynamic comparator are present as V(reset) and V(dc), respectively. Plot V(a1) and V(b1) correspond to signals applied at input pair (*A1*, *B1*) of both comparators. Comparator outputs are shown in V(comp_s) and V(comp_pd) for the static and the dynamic comparators, respectively.

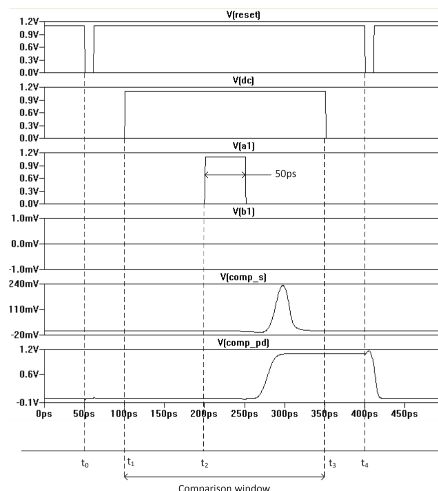


Figure 3.11 Detection Capability of Pseudo-Dynamic and Static Comparators

Figure 3.11 shows that the pseudo-dynamic comparator is able to detect the glitch of Δ=50ps while the static comparator filters it. By varying the glitch duration, we observe that the pseudo-dynamic comparator can detect glitches of 42ps while the static comparator detects only 55ps or larger glitches. Moreover, due to their un-symmetric internal structure, both comparators have detection capability that depends on the glitch form. Simulations with *B1* and *A1* kept at logic-1 and 1-0-1 glitches reveal that the pseudo-dynamic comparator can detect glitches of 44ps wide. In the same conditions, the static comparator can only detect larger than 58ps glitches.

Note that in the simulations above, Global comparison stages of both comparators contain only Layer 1 (Figure 2.10, Figure 2.5). However, Layer 2 is needed for comparators with higher input number. This layer which consists of static OR-tree may also filter glitches at Layer 1 outputs. Consequently, detection capability of larger static comparators may decrease. This degradation depends on the CMOS technology used to implement the gates. In our case, it remains small. In fact, for a static comparator of 1024-bit input vectors, the smallest glitch detectable is of 60ps size. For pseudo-dynamic comparators, the

sensitivity is not affected by electrical mask because DOR outputs are stable signal. Therefore, in all cases, pseudo-dynamic comparators always detect smaller glitches than static comparators.

3.4.2 Control logic simulation

The objective of this sub-section is to verify that timing characteristics of control signals provide by the synthesized control module satisfy different timing constraints established in Chapter 2. For this, we run SPICE-like simulations of this module with Synopsys NanoSim. In these simulations, we use a *CLK* signal of 10ns period whose high phase duration is 1ns (10% duty cycle). To initialize the module, its *resetControl* input is asserted (logic-0) during the first CLK period and then remains at logic-1 for the rest of the simulation. To simulate error occurrences, *error* signal is switched from initial logic-0 to logic-1 at t=32ns, during the third *CLK* period.

In Figure 3.12, waveform of *CLK*, *reset* and *DC* signals are presented as v(clk), v(reset) and v(dc). We can observe that *reset* is asserted (logic-0) during the fifth period (two *CLK* periods after error occurrence). This corresponds exactly to expecting timing behaviors shown in Figure 2.21-b. Besides, we have proven in the previous sub-section that low phase duration of more than 100ps allows *reset* signal to completely pull down *error* signal to logic-0 (Figure 3.10). This process takes about 50ps, which is smaller than the delay between *reset* falling edge and *DC* rising edge. Therefore, *error* signal is at correct low level during *DC* high phase which allows hybrid fault-tolerant architectures to return to normal operation.

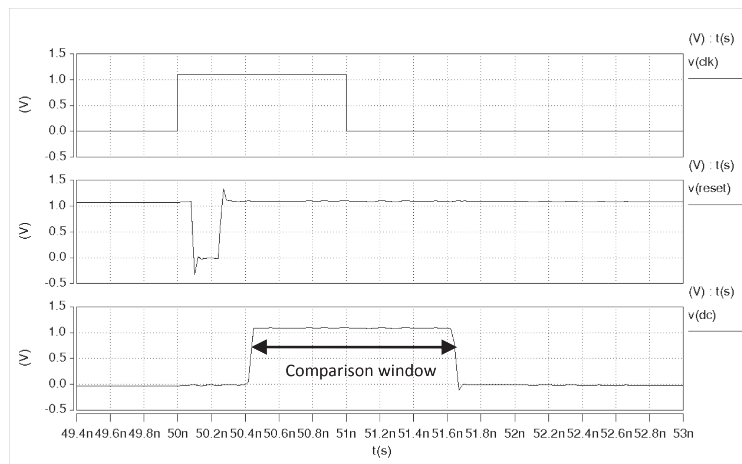


Figure 3.12 Generated Control Signals for Pseudo-Dynamic Comparator

Figure 3.12 also shows that the high phase of *DC* signal happens from 0.4ns to 1.6ns after each CLK positive edge. This guarantee that transient phase of CL outputs are not detected as errors because our CLs have minimum short path delays of 2ns.

Figure 3.13 shows simulation results of CLKRegin and CRegin signals that control hybrid fault-tolerant architectures' input registers. Waveform of *CLK*, *reset*, *CLKRegin* and *CRegin* are present as v(clk), v(error), v(clkregin) and v(cregin) respectively. We can see that *CLKRegin* low phase happens from 3ns to 3.7ns after each CLK rising edge. Consequently, there are 1.4 ns between the end of the comparison window (Figure 3.12) and CLKRegin falling edge. This gap allows *error* signals that are triggered at the end of the comparison window to reach high level and switch *CRegin* to logic-1 before *CLKRegin* negative edge. This condition is satisfied in our simulation where *CRegin* turns to high level at t=32.2ns while *CLKRegin* falling edge happens at t=33ns. Note that as constrained in Chapter 2 (Figure 2.21), *CRegin*

signal only remains at logic-1 during one *CLK* period and then returns to logic-0 before the next *CLKRegin* negative edge. Therefore *CLKRegin* and *CRegin* satisfy timing conditions that guarantee correct function of hybrid fault-tolerant architecture.

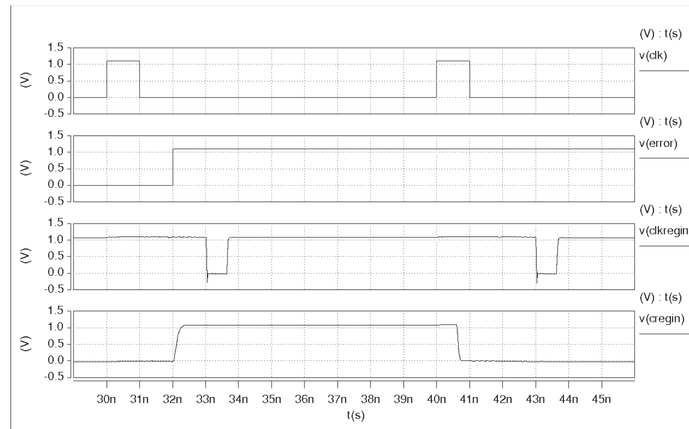


Figure 3.13 Generated Control Signals for Input Register

With correct timing of *CLKRegin* and *CRegin* signals, the control logic module is able to generate appropriate control signals *d1*, *d2*, *d3* and *m2*, *m1* to control input demultiplexer and output multiplexer of the hybrid fault-tolerant architecture. This is proven in Figure 3.14 where *CLK*, *error* and the control signals are presented as v(clk), v(error), v(d1), v(d2), v(d3), v(m1) and v(m2).

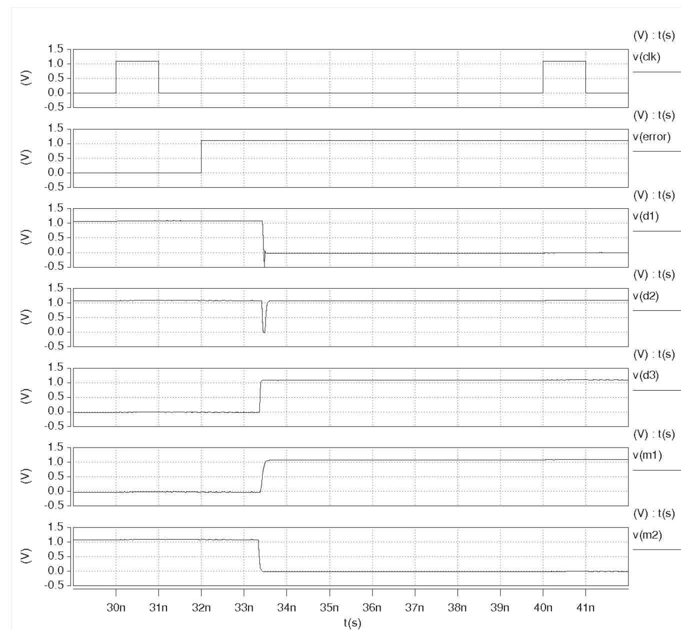


Figure 3.14 Generated Control Signals for Input Demultiplexer and Output Multiplexer

Figure 3.14 shows that when an error is detected at the third CLK period, the control signals are changed so that the hybrid fault-tolerant architecture is re-configured. In fact, (d1,d2,d3,m1,m2) switch

from logic (1,1,0,0,1), which means CL1 and CL2 are running in parallel, to logic (0,1,1,1,0) that orders CL2 and CL3 to run in parallel (Figure 2.25, Figure 2.28). Note that the re-configuration finishes before the end of the third period. Consequently, the architecture is ready for re-computation at the next period.

3.4.3 Hybrid fault-tolerant architecture simulation

The previous sub-section has proven that critical modules of the hybrid fault-tolerant architecture function correctly. In this sub-section, we study error detection capability of the complete architecture with regard to transient and permanent errors. The simulations in this sub-section are also SPICE-like simulations performed by NanoSim. The hybrid fault-tolerant architecture studied is the synthesized architecture for c6288 circuit of ISCAS'85 benchmark.

In the following simulations, the architecture is run with *CLK* signal of 10ns period and 10% duty cycle. Note that this period is larger than maximum CL delay of c6288 (Figure 3.2). This guarantees that no timing error can occur without fault injection. Before running different input vectors from t=70ns, the architecture is initialized so that its pseudo-dynamic comparator is reset and its configuration order CL1 and CL2 to run in parallel.

Simulation results of the fault-tolerant architecture in a fault-free case are presented in Figure 3.15. Signals shown in this figure correspond to that presented in Figure 2.24: *CLK* signal (clk); primary input (*PI*) and output (*PO*) vectors (pi[31:0] and po[31:0]); *error* signal (error); input vectors *i1*, *i2* and *i3* of CL1, CL2 and CL3 (i1[31:0], i2[31:0] and i3[31:0]); output vectors *vout1* and *vout2* of Mux (vout1[31:0] and vout2[31:0]). Logic value of vectors in this figure is expressed in Hexadecimal.

In Figure 3.15, *error* signal is stable at logic-0 confirming that the architecture is fault-free. Besides, we can see that only input *i1* and *i2* of CL1 and CL2 are receiving input value from *PI*. CL3 is at stand-by because its input *i3* is stable at 00000000_{16}. Finally, we can observe that outputs *vout1* and *vout2* of Mux remain stable for more than 2ns after each *CLK* rising edge, which guarantee correct function of the pseudo-dynamic comparator.

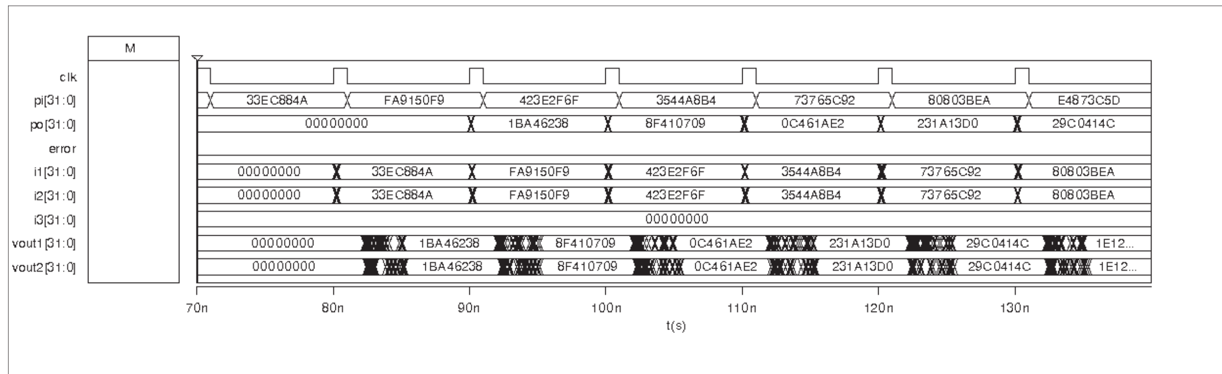


Figure 3.15 Hybrid Fault-Tolerant Architecture's Behavior in Fault-Free Case

In order to simulate function of the hybrid fault-tolerant architecture with error occurrence, we inject transient and permanent faults to output *o2* of CL2. For this, an additional 2-input XOR gate is inserted between CL2 and Mux. The XOR gate uses an additional input *fault* to flip the 0th output bit *o2[0]* of CL2. Modified RTL description of the hybrid fault-tolerant architecture is detailed in Appendix A.

To simulate an error caused by a transient faults (SEU, timing error) at CLs, a 0-1-0 glitch is injected to *fault* signal at the beginning of a period from t=90ns to t=100ns. Besides, primary input vector *PI* is kept

unchanged during two *CLK* periods of error detection and correction. Simulation results are shown in Figure 3.16. In addition to signals presented in Figure 3.15, this figure also shows *fault* signal (fault).

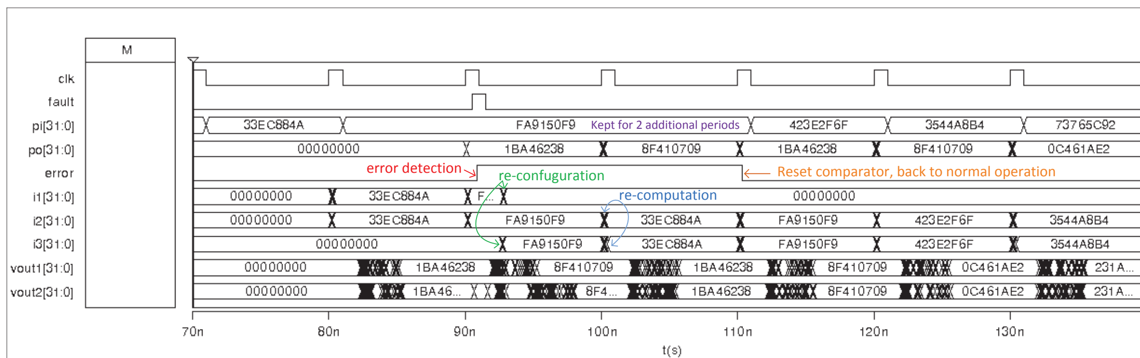


Figure 3.16 Hybrid Fault-Tolerant Architecture's Behavior with Transient Error Occurrence

In Figure 3.16, we can see that *error* signal turns to logic-1 signaling that *vout2* and captured *PO* have different values during the comparison window. During the same period between t=90ns and t=100ns, *CL1* is put in stand-by (stable logic 00000000_{16} at *i1*) while *CL3* is turn on (*i3* receives captured primary input). The re-configuration successfully finishes before the beginning of new *CLK* period. At next *CLK* positive edge (t=100ns), the previous value of primary input ($33EC884A_{16}$) is applied to *CL2* and *CL3*. This shows that the input register and the control logic module have correctly triggered a re-computation. Note that *error* remains at logic-1 during this period. The re-computation finishes before next *CLK* edge at t=110ns. As *vout2* and *PO* are identical, *error* returns to logic-0 signaling that the captured output is correct. The transient error is tolerated by the architecture with two additional *CLK* periods.

To simulate a permanent fault at CLs, *fault* signal is kept at logic-1 from t=90ns. Consequently, CL2 output is permanently affected. Simulation results are presented in Figure 3.17 with the same signal of Figure 3.16.

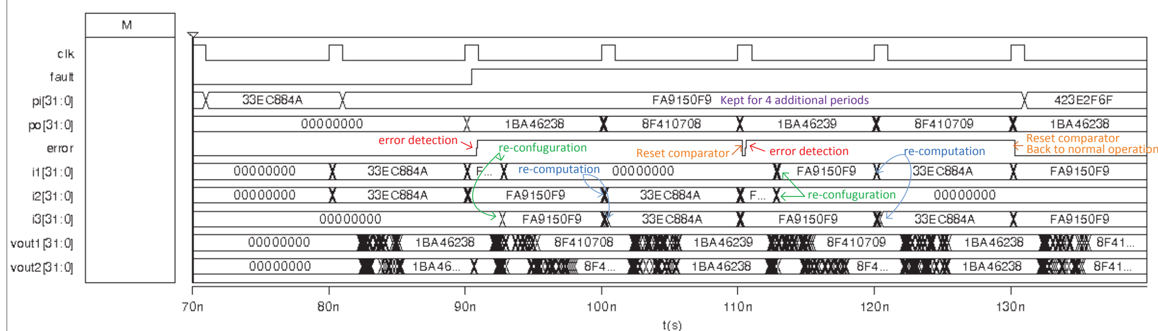


Figure 3.17 Hybrid Fault-Tolerant Architecture's Behavior with Permanent Error Occurrence

In the first two periods after error occurrence (from t=90ns to t=110ns), the architecture works exactly like in case of transient errors. It is re-configured so that CL2 and CL3 run in parallel to re-

compute the input vector 33EC8B4A. However, after CLK edge at t=110ns, *error* signal is briefly reset to logic-0, then return to logic-1, signaling that *PO* and *vout2* are not identical. This is caused by the permanent error that remains in CL2. Consequently, a new re-configuration is done during the period between t=110ns and t=120ns. CL2 is put on stand by while CL1 is turned on. After re-computation during next period, the permanent error is tolerated and *error* returns to logic-0 at t=130ns.

3.4.4 Discussion

In this section, we have studied timing behavior of pseudo-dynamic comparator, control logic module as well as complete hybrid fault-tolerant architecture. SPICE and SPICE-like simulation at transistors level of synthesized modules and architectures have shown that implemented hybrid fault-tolerant architectures respect all timing constraints specified in Chapter 2 and provide expected error correction capability for both transient (SET and timing errors) and permanent (hard errors) faults.

3.5 Power simulation

Beside error detection/correction capability, power saving is another important advantage of the hybrid fault-tolerant architecture compared to other solutions. In this section, we compare power consumption of the proposed architecture (Figure 2.24) with Partial and Full TMR architectures (Figure 3.3). In order to perform such comparison, we use NanoSim to run SPICE-like simulation of synthesized architectures obtained earlier. For each original logic circuit, the three corresponding fault-tolerant architectures are fed by the same set of 100 random input vectors. Average currents at power supply node VDD are monitored to deduce average power consumption of the architectures. Note that the number of random input vectors is chosen so that simulation results may represent typical power consumption of fault-tolerant architecture during normal operations, while simulations can be done with available resources of our simulators (time and memories).

Due to limit of time and memories available for simulation, we only evaluate power consumption of largest ISCAS'85 benchmark circuits. Table 3.12 presents results of these simulations. In this table, the first three columns correspond to name, input and output number of the benchmark circuits. The three next columns detail average power consumption of the hybrid fault-tolerant, the partial TMR and the Full TMR architectures. All power consumptions are calculated in milliwatt. The two final columns show average power saving of hybrid fault-tolerant architectures compared to Partial and Full TMR architectures. These values are expressed in percentage of corresponding TMR architecture's average power consumption.

Name	Nb. Input	Nb. Output	Average Power Consumption (mW)			Average Power Reduction	
			Hybrid	Partial TMR	Full TMR	Partial TMR	Full TMR
c5315	178	123	3.5	5.0	5.4	30.3%	35.4%
c6288	32	32	5.5	8.4	8.4	34.7%	35.3%
c7552	206	107	4.9	7.1	7.9	30.3%	37.6%

Table 3.12 Power Saving of Hybrid Fault-Tolerant Compared to TMR Architectures

In hybrid fault-tolerant architecture, only two CLs are running in parallel. This gives us about 33.3% power saving compared to three operating CLs in TMR architectures. In Table 3.12, we observe that the average power saving values are comparable to this ratio.

For c5315 and c7552 benchmark circuits, hybrid fault-tolerant technique reduces about 30% of power consumption compared to Partial TMR solution. This value, which is smaller than expected, may be explained by power consumption overhead introduced by redundant modules such as input demultiplexer, output multiplexer and shadow latches in input register of hybrid fault-tolerant architectures. This is not the case for c6288 because its input and output number are small, which lead to much less additional hardware in hybrid fault-tolerant architecture. Besides, we can see that for this circuit, power reduction value is slightly higher than expected. This may be explained by the fact that pseudo-dynamic comparators consume less than word voters.

For Full TMR solution, power reduction values that are higher than 33.3% are due to the fact that additional registers in TMR architectures consume more power than input demultiplexer and output multiplexer in hybrid fault-tolerant architectures.

3.6 Summary

In this chapter, we have evaluated the hybrid fault-tolerant architecture proposed in Chapter 2 using combinational part of ISCAS'85 and ITC'99 benchmark circuits. The evaluations have allowed us to:

- Create RTL descriptions of the hybrid fault-tolerant architecture that can be used for all benchmark circuits: These descriptions only take as parameters input and output numbers of logic circuits regardless of their logic functions. Consequently, the concept of plug-and-play has been proven for the proposed fault tolerance solution.
- Implement the hybrid fault-tolerant architecture in the 45nm technology library using commercial synthesis tool: By comparing our solution with Partial and Full TMR architectures, we show that our solution has small or even negative area overhead compared to TMR methods.
- Verify timing behavior and fault-tolerance capability of the hybrid fault-tolerant architecture: Using SPICE and SPICE-like simulations, we show that its synthesized critical redundant modules function correctly and all timing constraints for control signals specified in Chapter 2 are respected. We also prove that the hybrid fault-tolerant architecture can tolerate efficiently both permanent (hard) and transient (timing, SET) errors at its CLs.
- Monitor power consumption of the hybrid fault-tolerant architecture: Using SPICE-like simulation with random input vectors, we compare this architecture with Partial and Full TMR solution using different benchmark circuits. Simulation results show important dynamic power saving of about 30% for Partial TMR and 35% for Full TMR.

Despite its advantage of fault tolerance capability and power saving have been proven, the hybrid-fault-tolerant architecture only dealt with permanent and transient faults in CLs. Besides, its power saving consist only of dynamic power reduction. Improvement of the architecture is proposed in the next chapter.

Chapter 4

Extended Usage of the Hybrid Fault-Tolerant Architecture

Chapter 4 Extended Usage of the Hybrid Fault-Tolerant Architecture 82
4.1 Aging phenomenon 83
4.1.1 Lifetime improvement 83
4.1.2 Usage of FSMs 84
4.1.3 Discussion 85
4.2 Application of the hybrid fault-tolerant method in pipeline architectures 86
4.2.1 Basic of pipeline architecture 86
4.2.2 Fault-tolerance for pipeline architecture 87
4.2.3 Hybrid fault-tolerant design for pipeline architecture 90
4.2.4 Conclusion 92
4.3 SEU protection 92
4.3.1 SEU protection techniques 92
4.3.2 SEU protection for the hybrid fault-tolerant architecture 94
4.3.3 Discussion 96
4.4 Summary 96

In previous chapters, we have proposed a hybrid fault-tolerant architecture for robustness improvement of digital logic circuits and systems. This architecture targets stand-alone pipeline-style circuits, *i.e.* circuits that are combined of an input register, a combinational logic module and an output registers (Figure 1.18). The proposed technique is able to detect and correct hard, SETs and timing errors in combinational part of these circuits, with advantageous silicon area and power consumption costs compared to existing solutions. Beside these objectives, the hybrid fault-tolerant method can also be used in others contexts such as lifetime improvement of logic circuits and fault-tolerance of pipeline architectures. Furthermore, it can be combined with SEU protection techniques to provide an ultimate solution that target hard, soft and timing errors in all part of logic circuits. In this chapter, we investigate the possibility to extend the use of the hybrid fault-tolerant architecture for these objectives.

The rest of this chapter is divided in three sections. The first section consists in using the proposed hybrid fault-tolerant architecture to deal with aging phenomenon. It studies how re-configuration mechanism can be used for robustness improvement of circuits against material wearout. This section also discusses how the two FSMs proposed in 2.4.3 can be used with regards to different aging effects. The second section extends the concept of hybrid fault-tolerance architecture to pipeline architectures. The proposed technique is compared to state-of-the-art solutions, such as clock gating using Razor flip-flops or architectural replay using Razor II flip-flops. Finally, the last section discuss about the possibility to include SEU protection of sequential elements in the hybrid fault-tolerant architecture. It explains how register-level SEU tolerance techniques can be used for such purpose with optimized area overhead compared to bit-level solutions such as TMR, Razor or Razor II.

4.1 Aging phenomenon

We have seen in Chapter 1 that during the last phase of digital circuits and systems' lifetime, their reliability decreases because of increasing wearout failure rate (Figure 1.4). Aging phenomenon such as oxide or interconnect wearout may cause permanent defects, which result in hard errors during circuit operations. These errors may either reduce robustness of digital circuits and systems or make them not usable. In this section, we discuss how the hybrid fault-tolerant architecture can help increasing the useful life of logic circuits. We will also see how different FSMs proposed in 2.4.3 can be used with regards to different aging effects.

4.1.1 Lifetime improvement

In the context of this section, lifetime (or the useful life) of logic circuits is defined as the duration, during which these circuits operate correctly with all possible input vectors, supposing that there is no transient error occurrence. For logic circuits without redundant resource, this notion represents the total running time before appearance of the first permanent fault caused by aging phenomenon. After this moment, usage of these circuits must be limited to a subset of input vectors, which do not activate the fault and produce erroneous output.

Fault-tolerant architectures employ redundant resources to guarantee correct operation of logic circuits, despite the presence of faults. Consequently, they may also help increasing lifetime of digital logic circuits. The followings of this sub-section provide qualitative comparison between impacts of TMR and the hybrid fault-tolerant architecture on this factor. Note that this discussion only considers permanent faults in combinational logic modules of the architecture. Other redundant modules (voter, pseudo-dynamic comparator, control logic module, etc.) as well as sequential elements are supposed to be fault-free.

TMR architecture

As detailed in previous chapters, TMR architecture with word-wise voter (see 3.2.2) operate correctly as long as at least two of its combinational logic modules provide fault-free outputs. Consequently, all single and multiple permanent faults that affect only one CL can be corrected successfully. Moreover, multiple faults across modules can also be tolerated if they are never active at the same time. Therefore, lifetime of TMR architecture is measured by the total running time until at least two of its CLs are affected by permanent faults, which can be activated by the same input.

At every moment, all CLs of TMR architecture are running in parallel and hence, age at the same rate. Consequently, expected time before the first permanent fault caused by aging phenomenon appears at any of these modules is equal to lifetime of the original logic circuit. However, the probability that faults at different CLs can be activated by the same input vector is smaller than 100%. Depending on the concrete CL structures, this probability can be considerably small. In this case, significantly lifetime improvement can be archived using TMR architecture.

The hybrid fault-tolerant architecture

Similar to TMR techniques, the hybrid fault-tolerant architecture can be used as long as at least two of its CLs can operate correctly for each possible input vector. However, in this architecture, only two CLs are running in parallel while the remaining is on standby. The last CL does not have any switching activities and hence, ages slower than the others. If the hybrid fault-tolerant architecture is re-configured periodically then each CL only have to run 2/3 of the total running time of the architecture. Consequently, these modules suffer much less from aging phenomenon compared to CLs of TMR architecture. Therefore, the hybrid fault-tolerant architecture offers better solution for lifetime improvement of logic circuits.

In order to balance aging of CL modules in the hybrid fault-tolerant architecture, it must be re-configured periodically. This can simply be done using a counter which defines the number of CLK cycles after which a re-configuration is performed even is fault-free case.

4.1.2 Usage of FSMs

As second discussion on aging phenomenon, we analyze further the impact of using FSM1 and FSM2 proposed in 2.4.3 (Figure 2.30) on different fault-tolerance scenarios. We have seen that FSM1 only changes the configuration when two consecutive errors are detected while FSM2 changes the configuration each time an error occurs. Depending on current configuration and nature of errors, each FSM may require different number of re-configuration/ re-computation before errors are tolerated. As the hybrid fault-tolerant architecture suffers from aging phenomenon, error occurrence rate may increase significantly. Consequently, the capability to tolerate errors in a minimum number of re-configuration/re-computation of FSMs becomes an important factor.

This sub-section studies different examples of error occurrence during the computation of an input vector. Without losing generality, we suppose that the hybrid fault-tolerant architecture is at the configuration where CL1 and CL2 are running in parallel for the first computation of the input vector. Each time an error is detected (PE for permanent error, TE for transient error) during computation of the input vector, the architecture is re-configured. Re-configuration/re-computation repeats until all errors are tolerated (OK).

Let us consider the following notations used in the rest of this sub-section:

- Configuration j-k is the configuration when the jth and kth combinational logic modules of the hybrid fault-tolerant architecture are running parallel.
- Pi represents a permanent fault affecting the ith combinational logic module.

- Tjk represents a transient fault that occurs at first period when the configuration j-k is active.

Table 4.1 and Table 4.2 present different re-configurations done by FSM1 and FSM2 during the computation of an input vector, for seven error occurrence scenarios. In each table, the first row indicates the current configuration used at each computation. The remaining rows indicate the simulated scenarios. For example, scenario P_1-S_{23} means that CL1 is affected by a permanent fault and that a soft error occurs when the configuration is switched to 2-3.

Configuration	1-2	1-2	2-3	2-3	3-1	3-1
T12	TE	OK				
P1	PE	PE	OK			
P2	PE	PE	PE	PE	OK	
P3	OK					
P1-S23	PE	PE	SE	OK		
P2-S31	PE	PE	PE	PE	SE	OK
P3-S12	SE	OK				

Table 4.1 Re-configuration by FSM1

Configuration	1-2	2-3	3-1	1-2	2-3	3-1
T12	TE	OK				
P1	PE	OK				
P2	PE	PE	OK			
P3	OK					
P1-S23	PE	SE	PE	PE	OK	
P2-S31	PE	PE	SE	PE	PE	OK
P3-S12	SE	PE	PE	OK		

Table 4.2 Re-configuration by FSM2

The second row of both tables shows that only one re-configuration is needed for a transient error that occurs at the first computation.

The third to fifth rows of the tables show how re-configuration/re-computation is done for single permanent error that affects one of the CLs. We can see that FSM2, which re-configures at each error detection, allows faster fault-tolerance. Therefore, this FSM is suitable in case the hybrid fault-tolerant architecture suffers from aging phenomenon which creates high density of permanent faults.

In the three last scenarios, a transient error occurs after the previous hard error is tolerated. These scenarios correspond to the cases where aging phenomenon increase apparent rate of both permanent and transient errors equally. From the tables above, we can see that FSM1 is the suitable solution for efficient fault-tolerance in these cases.

4.1.3 Discussion

In this section, we have discussed effects of aging phenomenon on usage of the hybrid fault-tolerant architecture. We have seen that using periodical re-configuration to balance aging of combinational logic module in this architecture may help increasing circuit lifetime. This improvement is proven better than results archived using TMR architectures.

As second discussion, we have seen how different FSM versions can be use in case of high error apparent rate due to aging phenomenon. In the case where hard errors are dominant, FSM2, which re-configures the architecture at each error detection, is the suitable solution. However, if apparent rate of transient errors is also important then FSM1 may allow faster fault-tolerance scheme.

4.2 Application of the hybrid fault-tolerant method in pipeline architectures

In Chapter 1, we have seen that frequency scaling process in advanced CMOS technology nodes is limited by a “power wall”. Consequently, the semiconductor industry must find other solutions for speed improvement of digital circuits and systems. Pipeline architecture is one of the key methods used to make faster digital systems such as microprocessors. In subsequent sub-sections, we detail basics of pipeline methods, reliability issues in these techniques, as well as how the hybrid fault-tolerant architecture can be employed to resolve these problems.

4.2.1 Basic of pipeline architecture

In computer systems, pipelining is defined as “*an implementation technique whereby multiple instructions are overlapped in execution; it takes advantages of parallelism that exists among actions needed to execute an instruction*” [HEN07].

To clarify the definition above, Figure 4.1 illustrates the difference between pipeline architectures and stand-alone logic circuits.

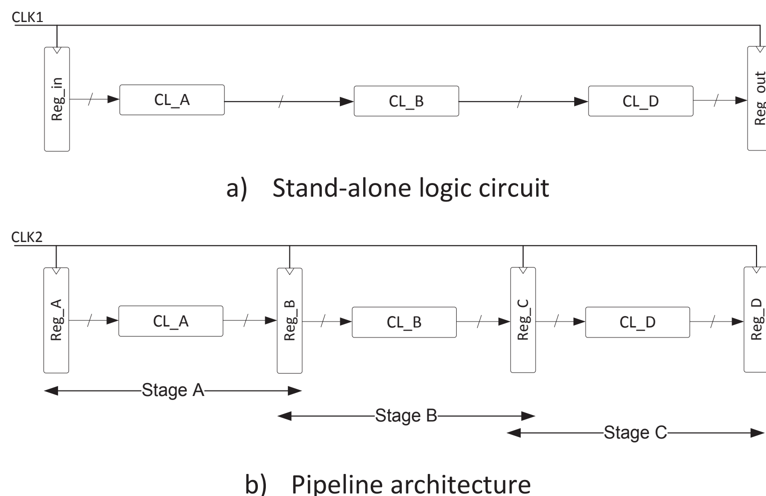


Figure 4.1 Stand-Alone Logic Circuit versus Pipeline Architecture

The stand-alone logic circuit in Figure 4.1-a is combined of an input register (Reg_in), an output register (Reg_out) and a combinational logic between these registers. The combinational part can be divided into three independent combinational logics connected in series (CL_A, CL_B and CL_C). This stand-alone circuit is cadenced by clock signal *CLK1*. The clock period t_{period} is defined by the sum of combinational logics' computation time.

Figure 4.1-b shows a pipeline implementation of the stand-alone circuit presented above. It is divided into three stages (A, B and C) corresponding to three combinational logic parts (CL_A, CL_B and CL_C).

Each stage consists of a combinational logic part, an input and an output registers. Moreover, output register of each stage is used as input register of its next stage. The pipeline architecture is cadenced by clock signal *CLK2* whose period is defined by the maximum computation time among CL_A, CL_B and CL_C. In the case where these logics have similar delay, *CLK2* period is one third of t_{period}.

With the presented structure, each nth task (or instruction) of the pipeline architecture is divided into three smaller actions An, Bn and Cn. These actions are consecutively executed in three pipeline stages A, B and C. Therefore, computation time for each instruction of the pipeline architecture is equal to three times of *CLK2* period. Consequently, the pipeline architecture have similar calculation delay compared to the stand-alone logic circuit. However, actions of different instructions can be overlapped in execution, which helps increasing the architecture throughput, *i.e.* the number of instruction done in a given time. This key concept of pipeline technique is illustrated by examples in Table 4.3.

CLK2 Period	1	2	3	4	5	6	7	8
Stage A	A1	A2	A3	A4	A5	A6	A7	A8
Stage B		B1	B2	B3	B4	B5	B6	B7
Stage C			C1	C2	C3	C4	C5	C6
Output				D1	D2	D3	D4	D5

Table 4.3 Operation of Pipeline Architecture

Table 4.3 presents normal operation of the pipeline architecture (Figure 4.1-b) in fault-free case. The first line of this table shows the number of *CLK2* period. The three next lines detail to actions being executed at Stage A, B and C of the architecture during corresponding periods. For example, during the second period, Stage A is executing action A2 of the second instruction while Stage B is performing action B1 of the first instruction. The last line of Table 4.3 corresponds to outputs of finished instructions that are be stored in output register Reg_D at each *CLK2* period.

We can observe in Table 4.3 that during fault-free operation, each instruction requires three *CLK2* periods to complete. For example, action A1 of the first instruction is started at the beginning of the first period while output D1 is only available at the beginning of the fourth period. As *CLK2* is three times faster than *CLK1* of the stand-alone circuit (Figure 4.1-a), this confirms that both architectures have the same calculation time. However, we can see in Table 4.3 that after seven *CLK2* cycles (from the beginning for the first period to the beginning of the eighth period) the pipeline architecture has finished five instructions. This work load takes five *CLK1* cycles or fifteen *CLK2* cycles for the stand-alone circuit to complete. Therefore, this example have shown that the pipeline have higher throughput than the stand-alone circuits.

By extending the previous example for a larger number of periods, we can prove that the pipeline architecture archives three times higher throughput compared to the stand-alone circuits during fault-free operations. Furthermore, if the combinational part of the stand-alone circuit is divided into N parts with similar delays, the corresponding pipeline architecture may have N times higher throughput.

4.2.2 Fault-tolerance for pipeline architecture

Error propagation

Similar to stand-alone logic circuits, pipeline architecture may also suffer from transient and permanent errors. Furthermore, in these structures, there are errors propagations between pipeline stages which require special re-computation schemes for error correction.

Table 4.4 illustrates an example for error propagation in the pipeline architecture of Figure 4.1-b. This table has the same lines as Table 4.3. However, it shows in bold red police the action executions affected by faults. Meanwhile, executions which receive faulty inputs caused by error propagation are presented in regular red police with an "*" symbol.

CLK2 Period	1	2	3	4	5	6	7	8
Stage A	A1	A2	A3	A4	A5	A6	A7	A8
Stage B		B1	B2	B3	**B4**	B5	B6	B7
Stage C			C1	C2	C3	C4*	C5	C6
Output				D1	D2	D3	D4*	D5

Table 4.4 Error Propagation in Pipeline Architecture

In the example of Table 4.4, we can see that during the fifth period, a fault becomes active at Stage B while it is performing action B4 of the fourth instruction. Consequently, at the next period, input vector for action C4 is affected by errors. These errors are then propagated to the output of the pipeline architecture. Therefore, output D4 of the fourth instruction is faulty. Note that in this example, we only consider transient faults. As a result, the pipeline architecture returns to normal operation at the eighth period when errors have reached the final register.

State of the art

Different works have studied fault-tolerance techniques for pipeline architectures, especially from timing errors at their combinational parts. Among them, the most prominent solutions are Razor [ERN03] and Razor II [DAS09]. We have seen in Chapter 1 that both architectures provide efficient SETs and timing errors detection at combinational part of stand-alone logic circuits. Razor method performs this detection by comparing circuit outputs with a reference value captured by shadow latches. Meanwhile, Razor II detects invalid transitions caused by errors at circuit outputs. While employing different error detection strategies, both Razor and Razor II techniques use timing redundancy for error correction in pipeline architectures. In order to deal with error propagation in these structures, Razor method uses global clock gating while Razor II technique performs architectural replay for re-computation. These detection/correction schemes are presented in flowing parts.

Global clock gating with Razor

Figure 4.2 shows how Razor and global clock gating methods are implemented for error Detection/Correction in the pipeline architecture of Figure 4.1-b.

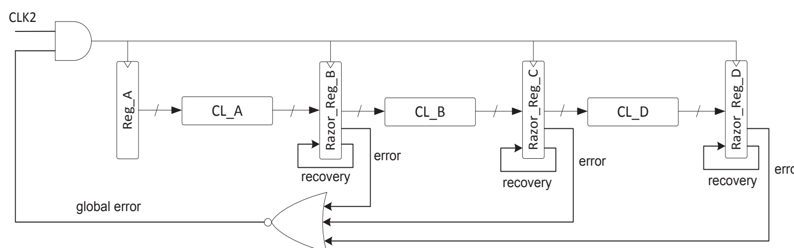


Figure 4.2 Razor and Global Clock Gating Implementation for Pipeline Architecture [ERN03]

Compared to the original pipeline structure, the architecture in Figure 4.2 employs Razor flip-flops for output register of all stages (Razor_Reg_B, Razor_Reg_C and Razor_Reg_D). Each register provide SETs and timing errors detection for the corresponding stage. Their error signals are combined by a NOR gate to form a global error signal. This signal is then used to disable global clock signal *CLK2* in case of error occurrence (one of the error signals at logic-1 and the global error signal at logic-0). While *CLK2* is gated, faulty values stored in Razor registers are recovered using correct values stored in their shadow latches (see Figure 1.22). Error signals are then reset to logic-0, which allows *CLK2* to control the registers. At next *CLK2* rising edge, all values stored in registers are correct. Hence, the architecture can return to its normal operation.

Table 4.5 shows how the previously described scheme helps correcting errors in the example of Table 4.4. As a fault become active at Stage B during the fifth period, its erroneous output is captured by Razor_Reg_C at the beginning of the sixth period. This Razor register detects this error during the same *CLK2* cycle. After error detection, the seventh *CLK2* rising edge is gated. Consequently, Reg_A, Razor_Reg_B and Reg_D conserve their previous value. Meanwhile, the correct value of Razor_Reg_C is restored from its shadow latches. Consequently, at the eighth *CLK2* cycle, all errors are corrected and the architecture returns to its normal operation.

Fault occurrence (5) — Error detection (6) — Clock edge gated (7) — Recovery (7)

CLK2 Period	1	2	3	4	5	6	7	8
Stage A	A1	A2	A3	A4	A5	A6	A6	A7
Stage B		B1	B2	B3	**B4**	B5	B5	B6
Stage C			C1	C2	C3	C4*	C4	C5
Output				D1	D2	D3	D3	D4

Table 4.5 Error Correction in Pipeline Architecture Using Razor and Clock Gating

As shown in the previous example, error detection/correction scheme using global clocking for Razor technique only requires one clock cycle. It is due to the fact that the correct output of pipeline stages can be restored from shadow latches of Razor registers. However, this is only applicable for SET and timing errors but not hard errors in combinational logics.

Architectural replay with Razor II

Figure 4.3 illustrates how Razor II registers and architectural replay are employed in the pipeline architecture of Figure 4.1-b. Similar to the structure in Figure 4.2, this solution also replaces pipeline stages' output registers by Razor II registers (Razor_II_Reg_B, Razor_II_Reg_C, Razor_II_Reg_D). Error signals which alert SET and timing error occurrences at each pipeline stage are then combined together in order to form a global error signal. However, instead of being used to disable *CLK2* after error occurrences, the global error signal is driven toward a system control module. Then, this module performs an architectural replay process, which consists of "flushing" the entire pipeline and restart from the first action of infected instruction. An example of such process is shown in Table 4.6.

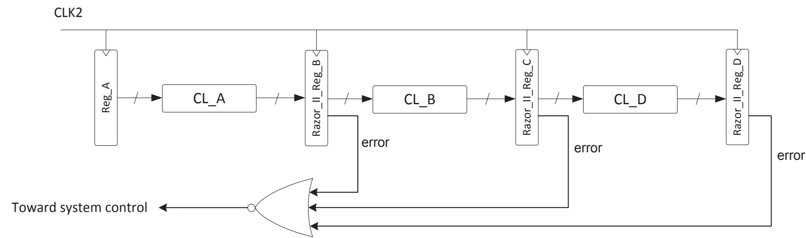


Figure 4.3 Razor II and Architectural Replay Implementation for Pipeline Architecture [ERN03]

Table 4.6 shows the use of architectural replay to tolerate errors in the example of Table 4.4. As a fault becomes active at Stage B during the fifth period, Razor_II_Reg_C register detects errors during the sixth period. Consequently, the global error signal turns to logic-0 and actives architectural replay process. At the seventh *CLK2* positive edge, Stage A re-executes the first action A4 of the (affected) fourth instruction. Meanwhile, other stages stop their current executions and wait for task propagation from the first stage. It takes two additional periods before the fourth instruction is propagated to Stage C. Then, the pipeline architecture returns to its normal operation.

Fault occurrence
Error detection
Architectural replay

CLK2 Period	1	2	3	4	5	6	7	8	9	10
Stage A	A1	A2	A3	A4	A5	A6	A4	A5	A6	A7
Stage B		B1	B2	B3	**B4**	B5		B4	B5	B6
Stage C			C1	C2	C3	C4*			C4	C5
Output				D1	D2	D3				D4

Table 4.6 Error Correction in Pipeline Architecture Using Razor II and Architectural Replay

Compared to clock gating technique, architectural replay do not require correct value restoration from shadows latches. This helps reducing area overhead of the solution. However, this technique requires higher re-computation time. In the example above, architectural replay requires three clock cycles to tolerate SETs and timings errors because errors are detected at the third stage of the pipeline architecture. In general case, error correction using this method takes n clock cycles if errors are detected at the nth pipeline stage. Besides, similar to Razor technique, Razor II method only allows detection/correction of transient faults in digital circuits.

4.2.3 Hybrid fault-tolerant design for pipeline architecture

As we have seen in previous sub-sections, Razor and Razor II techniques provide efficient solutions for transient fault-tolerance in pipeline architecture. However, they are not applicable for permanent faults created by manufacturing defect or aging phenomenon. One possible solution for this problem consists of combine these techniques with the use of TMR method for combinational part of pipeline architectures. However this solution significantly increases silicon area and power consumption of the architectures.

We have shown in previous chapter that the hybrid fault-tolerant architecture allows detection and correction of permanent faults at advantageous are overhead and power consumption compared to TMR methods. Besides, it also tolerates SETs and timing errors at combinational part of logic circuits. Consequently, using the hybrid fault-tolerant method in pipeline architectures may helps improving

robustness of these architectures with regards to hard, SETs and timing errors at reasonable costs. Figure 4.4 illustrates the implementation of this solution for the original pipeline architecture in Figure 4.1-b.

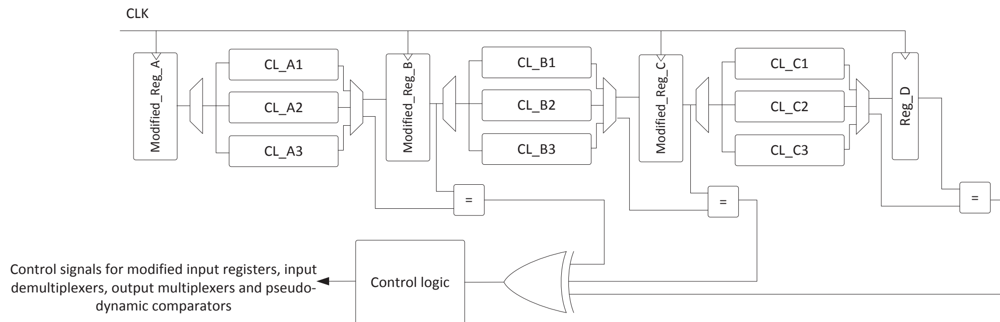


Figure 4.4 Hybrid Fault-Tolerant Implementation for Pipeline Architecture

In Figure 4.4, combinational part of each pipeline stage is triplicated (CL_Ai, CL_Bi, CL_Ci with i=1,2,3). Similar to the hybrid fault-tolerant architecture for stand-alone circuits, only two combinational logic copies of each stage are running in parallel at every moment. Others combinational logic parts are kept at standby mode and thus, do not consume dynamic power. The configuration of each stage is controlled by its input demultiplexer and output multiplexer (see Section 2.4 in Chapter 2). In this architecture, errors are detected by comparing output of the two running combinational logics at each stage (one before and one after capturing by corresponding output register). The comparison is performed by pseudo-dynamic comparators (see Sub-section 2.2.3 in Chapter 2), which allows better SETs and timing errors detections. As errors are detected after output capturing, input registers (Modified_Reg_A, Modified_Reg_B, Modified_Reg_C) of all stages are implemented using modified D flip-flop (see Sub-section 2.3.1 in Chapter 2). Error signals of all stages are combined by an OR-tree in order to provide a global error signal. This signal is used by the control logic module to perform re-configuration and re-computation of the complete architecture for error correction. For this purpose, this module provides control signals for all input registers, input demultiplexers, output multiplexers and pseudo-dynamic comparators. For the reason of clarity, these signals are not presented in Figure 4.4.

With the architecture presented above, transient and permanent errors are tolerated as follows. At each stage, error detection for execution results of nth period is performed at the beginning of the $(n+1)^{th}$ period. If an error is detected at any pipeline stage then the global error signal is turned to logic-1. Consequently, the control logic module forces all input registers to switch to their value of nth period, which are stored in their shadow latches (see Sub-section 2.3.1 in Chapter 2). This process restores the whole pipeline architecture to its last fault-free state before error occurrence and thus, prevents error propagations. Besides, during the same period, all pipeline stages are re-configured for permanent faults tolerance. This process must finish before the end of $(n+1)^{th}$ period. At the next clock edge, pipeline stages re-execute actions of nth period. If no error is detected, then the architecture returns to its normal operation. Otherwise, the previous processes are re-applied.

Table 4.7 presents error correction operation of the hybrid fault-tolerant pipeline architecture for the example in Table 4.4. As a fault become active during the fifth *CLK2* period at stage B, the affected result captured by Modified_Reg_C is detected at the beginning of the sixth period. During the same *CLK2* cycle, the architecture is re-configured. Depending on the FSM of the control logic module, some running combinational logics may be replaced by their redundant logics, which are currently on standby. Meanwhile, modified input registers of all stages switch their value to the input of the fifth *CLK2* cycle.

This re-configuration process finishes before the seventh *CLK2* rising edge. Consequently, during the seventh period, all stages re-execute the same action as in the fifth period. Note that at the seventh *CLK2* capture edge, Stage 3 has just been re-configured and its output signals are unstable. This explains the unknown value stored in output register Reg_D during this period. In this example, as the error is transient, the architecture returns to normal operation at the eight clock cycle.

Fault occurrence (period 5) / Error detection and re-configuration (period 6) / Re-computation (period 7)

CLK2 Period	1	2	3	4	5	6	7	8	9
Stage A	A1	A2	A3	A4	A5	A6	A5	A6	A7
Stage B		B1	B2	B3	**B4**	B5	B4	B5	B6
Stage C			C1	C2	C3	C4*	C3	C4	C5
Output				D1	D2	D3	unknown	D3	D4

Table 4.7 Error Correction in the Hybrid Fault-Tolerant Pipeline Architecture

4.2.4 Conclusion

In this section, we have studied the possibility to use the hybrid fault-tolerant architecture for robustness improvement of pipeline architectures. The principle of a complete fault-tolerance scheme has been proposed where transient errors correction requires two clock cycles, which is faster than architectural replay using Razor II technique. Moreover, this scheme based on re-configuration/re-computation allows permanent error correction, which is not possible using clock gating with Razor architecture. Finally, similar to the hybrid fault-tolerant architecture for stand-alone circuits, this architecture promises lower power consumption compared to TMR techniques.

4.3 SEU protection

In previous chapters, we have seen how different kinds of faults and errors affect robustness of logic circuits. Among them, hard, SETs and timing errors in combinational part as well as SEUs in sequential part are the most encountered. It has been proven that the hybrid fault-tolerant architecture provides efficient solution for faults and errors in combinational part of logic circuits, at advantageous silicon area and power consumption costs. In this section, we study the possibility to combine this architecture with SEU detection/correction methods in order to form more advanced fault-tolerance solutions.

4.3.1 SEU protection techniques

Different techniques have been proposed in the literature to protect sequential elements of logic circuits, *i.e.* latches and flip-flops, from hard errors and/or SEUs. As we have discussed in Chapter 1, the most prominent techniques include TMR, BISER, Razor, GRAAL and Razor II [LYO62, ZHA06, ERN03, NIC07, DAS09]. These methods consist of employing hardware and timing redundancy to tolerate SEUs by errors masking or detection/correction. Some of them even provide SETs and timing errors protection for combinational part of logic circuits [ERN03, NIC07, DAS09]. Although having different ways to deal with errors, these techniques all protect registers at bit-level, *i.e.* redundant resources are added to each latch and flip-flop of the register under protection. Consequently, silicon area and power consumption overhead grow linearly with register size.

In [IMH11], authors have proposed a novel method, which tolerates SEUs in sequential elements at register-level and hence, allows significant silicon area saving compared to bit-level techniques such as TMR, BISER and Razor. This technique employs both hardware (additional latches) and information redundancy (linear code) to detect flipped bits in level sensitive registers. Two correction schemes are proposed: 1) Re-computation or 2) Using bit-flipping latches. Figure 4.5 presents principle configuration for protection schemes.

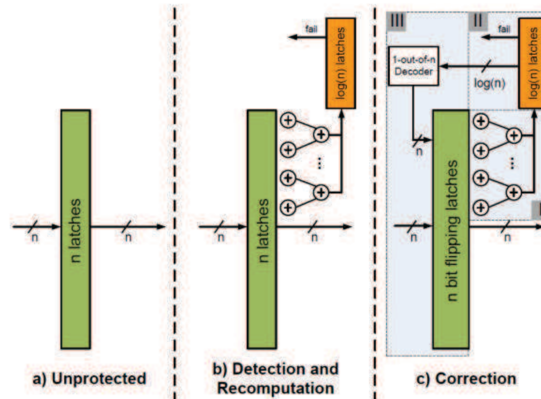


Figure 4.5 Register-Level SEU Protection [IMH11]

Figure 4.5-a shows an unprotected register, which is combined of n latches. Figure 4.5-b presents error detection scheme using linear code. The code word is computed from data of the register under protection using a XOR-tree and stored in redundant latches. During opaque phase of latches, the stored code word (called reference characteristic c_{ref}) is compared with the continuously computed code word (called current characteristic c_{cur}). If any mismatch is detected, a *fail* signal is used to triggered re-computation process. Note that SEUs can also arrive at redundant latches and cause faulty error detection. To avoid this problem, [IMH11] proposes to compute the parity of c_{ref} as $p(c_{ref})$ which is stored in an additional latch. This configuration, illustrated in Figure 4.6 allows the detection of all single SEU at any latches.

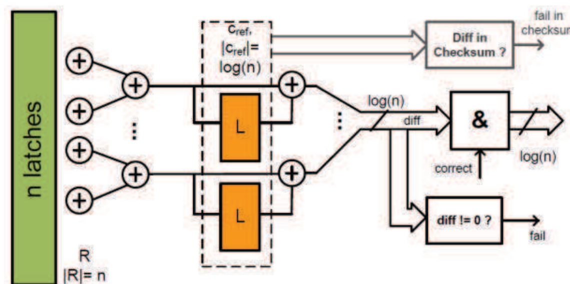


Figure 4.6 Detailed SEU Detection Scheme at Register-Level [IMH11]

In case re-computation is not feasible or is too time consuming, another error correction scheme is proposed in Figure 4.5-c. In this scheme, the n latches are replaced by n bit-flipping latches which are inherently able to invert their stored value. The difference *diff* between c_{ref} and c_{cur} is decoded and thus,

allow identification of the affected bits that must be inverted. Proposed design of the bit-flipping latch using transmission gates is presented in Figure 4.7.

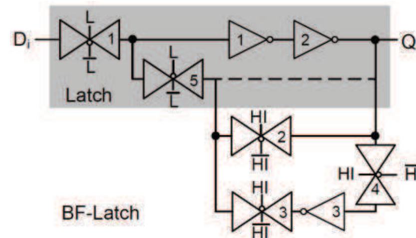


Figure 4.7 Bit-flipping latches for SEU Correction [IMH11]

This bit-flipping latch in Figure 4.7 is controlled by two signals: the main clock *L* and an additional signal *HI*. In fault free case, *HI* is kept at a constant value so that the second transmission gate (TG2) is opened while TG3 is closed. Consequently, the bit-flipping latch operates exactly like a conventional latch with clock *L*. In order to flip the stored bit during opaque phase of the latch (TG1 is closed while TG5 gate is opened), a short glitch must be applied at *HI*. The complement value of *Qi* is fed to the inverter chain formed by the first and second inverters (when *HI* changes state) and then stored in the latch (when *HI* returns to its normal state).

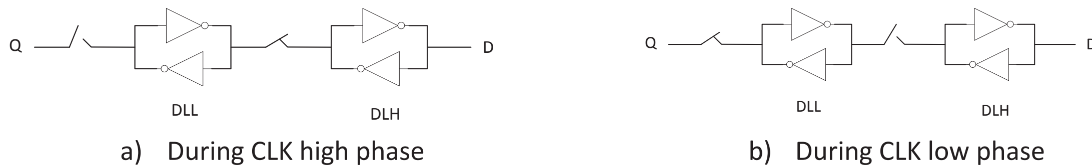
In [IMH11], authors propose the use of module-2 characteristic as error detection and correction code. This code allows not only error detection, but also error localization, which is needed to error correction using bit-flipping latches. Besides, it only requires $\log_2(n)$ check bits for information words of n bits. Consequently, only $\log_2(n)$ redundant latches are needed. This leads to significant silicon area saving compared to bit-level methods, especially when register size is important. For example, for 127-bit register, the proposed error detection scheme (Figure 4.5-b) only requires 127% area overhead compared to unprotected register. Error correction scheme (Figure 4.5-c) employs larger bit-flipping latches and thus have an area overhead of 183%. Meanwhile, the overhead is more than 300% for bit-level methods such as TMR, Graal or Razor [IMH11].

4.3.2 SEU protection for the hybrid fault-tolerant architecture

As the hybrid fault-tolerant architecture is able to detect and correct all kinds of fault in combinational part of logic circuits, adding SEUs protection to this scheme does not require bit-level methods such as Razor or Razor II. In fact, it can be combined with the fault-tolerance technique proposed in [IMH11], which allows optimization of the additional silicon area required for SEU protection. Sub-sequent parts of this sub-section details how this can be done for output and input registers of the hybrid fault-tolerant architecture.

Output register

Output register (Reg_out) of the hybrid fault-tolerant architecture is made of positive edge sensitive D flip-flops (DFFs). Each DFF is combined of a low level sensitive (DLL) and a high level sensitive D-latch (DLH). Both latches are controlled by the same clock signal *CLK*. Principle of a DFF during different *CLK* phases is illustrated in Figure 4.8. In this figure, a latch is presented as a switch followed by a memory point. The switch is opened during the latch's opaque phase, and closed during its transparent phase.



a) During CLK high phase b) During CLK low phase

Figure 4.8 D flip-flop function

During *CLK* high phase, DLL is opaque while DLH is transparent (Figure 4.8-a). Consequently, all SEUs that arrive at either DLL or DLH cause output D to flip. As a result, primary output vector *PO* of the hybrid fault-tolerant architecture (see Figure 2.24 in Chapter 2) switches value. Besides, in Chapter 2, we have seen that *PO* is continuously compared with output vector *vout2* of one running combinational logic during the comparison window (see Sub-section 2.2.3 in Chapter 2). As this window is set until the end of *CLK* high phase (see Figure 2.12 in Chapter 2), all changes in *PO* caused by SEUs during this period are detected by the pseudo-dynamic comparator. Re-computation process is then activated, which allows errors to be corrected.

During *CLK* low phase, DLL is transparent while DLH is opaque (Figure 4.8-b). Therefore, all SEUs that arrive at DLL during this period do not have any impact on operation of the hybrid fault-tolerant architecture. However, SEUs in DLH may cause output D to flip. This problem cannot be detected by the pseudo-dynamic comparator because its comparison window finishes at the beginning of *CLK* low phase (see Figure 2.12 in Chapter 2). Therefore DLH latches of the output register must be protected against SEU using the scheme proposed in [IMH11].

In order to use the register-level SEU protection technique for DLH latches of the output register, additional latches and error detection modules (XOR-tree for detection and correction code computation, comparator for error detection) are added following Figure 4.6. Besides, the new architecture still requires an SEU correction scheme, either by re-computation (Figure 4.5-a) or by using bit-flipping latches (Figure 4.5-b).

In Sub-section 2.3.1 of Chapter 2, we have seen that re-computation of the hybrid fault-tolerant architecture is only possible as long as previous input vector is still stored in shadow latch DLLs of its input register (see Figure 2.16 in Chapter 2). Figure 2.17 in Chapter 2 shows that this period finishes before the end of each *CLK* period (at *CLKRegin* low phase). Consequently, SEUs detected in DLH latches of the output register at the end of *CLK* low phase cannot be corrected by re-computation. Therefore, the only solution is to use bit-flipping latches (Figure 4.7) as DLH latches of the output register.

Input register

The hybrid fault-tolerant architecture use modified D flip-flop mDFFs (Figure 2.16) for its input register. Each mDFF is combined of a regular positive edge sensitive flip-flop DFF and a low level sensitive latch DLL.

The input register's DFFs also have principle schematic as presented in Figure 4.8. However, different than previous case, output of this register is not compared to any reference value during operations. Therefore, both DLL and DLH latches of each DFF must be protected against SEUs. Besides, as no re-computation is possible for this register, each latch must be replaced by bit-flipping latches (Figure 4.7).

Shadow latch DLLs of the input register stored previous input vector for re-computation. In case of single fault occurrence, if SEUs occur at these latches, then the rest of the architecture is supposed to be fault-free. As no re-computation is needed, the SEUs do not have any impact on data integrity of the hybrid fault-tolerant architecture. However, to protect the architecture from multiple faults, these latches also require the SEU tolerance method proposed in [IMH11].

4.3.3 Discussion

This section discusses about the possibility to add SEU protection in the hybrid fault-tolerant architecture. We have seen that bit-level techniques such as Razor and Razor II allow not only SEUs detection/correction but also SETs and timing errors tolerance. However, they all require very important additional redundant resources compared to register-level protection methods, which use information redundancy to detect SEUs in registers. Error correction can be performed either by re-computation or bit-flipping. As the hybrid fault-tolerant architecture is able to tolerate SETs and timing errors, we propose to combine it with the register-level SEU tolerance technique presented in [IMH11]. Besides, we have seen that only high level sensitive latches in output register of the hybrid fault-tolerant architecture require SEU protection whilst all latches in its input register must be protected.

4.4 Summary

In this chapter, we have extended the usage of the proposed hybrid fault-tolerant architecture in three axes:

- To dealt with aging phenomenon: As only two out of three combinational logic modules are running in parallel at any moments, the hybrid fault-tolerant architecture suffers less from aging phenomenon. Using a periodically re-configuration for this architecture allows all CLs to aging at smaller rates and thus, increases the useful life of the architecture compare to TMR method. Besides, we have seen that the FSM use for re-configuration also have impacts on fault-tolerance performance of the architecture. If the number of hard errors is dominant than re-configure the architecture at each error detections is the optimum methods. However, if the number of transient errors has the same importance than re-configuration after two consecutive detections is better solution.
- To tolerate faults in pipeline architectures: We have seen that beside SETs and timing errors, the hybrid fault-tolerant architecture also provides hard errors protection in combinational part of pipeline architectures, which is not possible for technique such as Razor and Razor II. Besides, its correction scheme using re-computation is proven to be faster than architectural replay method.
- To be combined with SEU protection technique: The hybrid fault-tolerant architecture detects and corrects all faults in combinational part of logic circuits. Consequently, we do not need bit-level SEU tolerance techniques such as Razor, Razor II, which also detect/correct SETs and timing errors. Instead, we can combine the hybrid fault-tolerant architecture with register-level technique, which use information redundancy to detect SEU. Error correction are performed using either re-computation (for low level sensitive latches of the output register) or using bit-flipping latches (for high level sensitive latches of the output register and all latches of the input register).

For each extension axes, qualitative studies for possible solution are provided. Quantitative studies, concrete implementations, as well as evaluations that confirm advantages of these solutions for each problem can be subjects of further researches.

Conclusion

Evolution of CMOS technology is one of the most important factors that are conducting the recent technological revolution. At each new technology node, transistor feature sizes are down scaled further, allowing the integration of more and more devices on chip. Together with frequency and power supply scaling, it allows the realization of more and more complex digital systems at lower costs and higher performance. These advantages explain why the semiconductor industry keeps scaling CMOS technology further despite the fact that reliability of digital systems has become an important issue.

Different factors are responsible for transient and permanent faults that degrade reliability of digital CMOS circuits and systems. First of all, manufacturing devices at nanometer scale is much more difficult, and thus leads to high rate of manufacturing defects. Together with aging phenomenon, these defects cause permanent fault in ICs. Secondly, PVT variations as well as aggressive timing requirements are responsible for increasing rate of timing errors. Finally, radiation and interference effects may cause more and more soft errors as small transistor are more vulnerable.

Given the importance of CMOS technology in recent information technology revolutions, it is necessary to solve problems of hard, soft and timing errors in advanced CMOS technology nodes. As targeting these issues at physical level is no longer feasible, fault-tolerant techniques, which deal with faults and errors at design level, become the best solutions. These techniques employ information, timing and hardware redundancy to guarantee correct operation of digital circuits and systems despite the presence of faults. Each type of redundancy has its pros and cons with regards to particular types of errors. Consequently, hybrid fault-tolerant methods, which combine the use of these redundancies, are one of the best solutions to targets different types of faults simultaneously.

In this manuscript, we have developed a hybrid fault-tolerant architecture that targets hard, soft and timing errors in order to improve robustness of digital circuits. The proposed method employs information redundancy for errors detection, timing redundancy for transient errors correction and hardware redundancy for permanent errors correction:

- Information redundancy is implemented under the form of duplication/comparison structure that detects error in combinational part of logic circuits. The detection is enhanced by the use of pseudo-dynamic comparator. This comparator, which employs dynamic CMOS gates to detect transitions at its input vector during a comparison window, allow the detection of hard, SETs and timing errors.
- Timing redundancy consists in performing re-computation for error correction. This is done with helps of a modified input register, which can store previous input vector until the corresponding output is proven correct by the pseudo-dynamic comparator. The re-computation scheme takes two clock cycles: one to restore affected input vector after error detection and one to re-compute this vector. This scheme allows the correction of SETs and timing errors in combinational part of logic circuits.
- Hardware redundancy requires a third copy of logic circuit's combinational part. This redundant module is used to replace the two other combinational logic modules in case of hard error occurrence. Note that, different than TMR technique, only two out of three CLs are running in parallel in the hybrid fault-tolerant architecture. The third one is put on standby and hence, does not consume dynamic power. Re-configuration of the hybrid fault-tolerant architecture is performed with help of a FSM. Two version of the FSM is proposed: 1) FSM1

consists in changing configuration only in case of two consecutive error occurrences, and 2) FSM2 re-configures the architecture each time an error is detected.

The proposed hybrid fault-tolerant architecture is evaluated using simulation with EDA tools. It is compared with TMR techniques using ISCAS'85 and ITC'99 benchmark circuits and a 45nm standard cell library. Simulation results have shown that the hybrid fault tolerant architecture is able to tolerate hard, SETs and timing errors. Compared to TMR architectures, it provides significant power saving of about 33% while having negligible area overhead. In fact, compared to TMR1 (Partial TMR) architecture where only combinational part of logic circuits is triplicated, the proposed method only requires from 3% to 9% additional silicon area. Meanwhile, it allows 2% to 6% area reduction compared to TMR2 (Full TMR) structure where sequential elements of logic circuits are also triplicated.

Beside advantages of fault-tolerant capability, silicon area and power consumption, the hybrid fault-tolerant also offer the possibility to deal with aging phenomenon. As only two out of three combinational logic modules are running in parallel at any moments, they all suffers less from aging phenomenon. By balancing running time of the modules, we can have them running only 2/3 of time compared to the same operations performed by original logic circuits or TMR structures. Consequently, the hybrid fault-tolerant architecture may have longer useful life. As a second discussion, we have seen that the two FSMs proposed may be used differently in various aging phenomenon effects: FMS2 is suitable for the case where hard errors occurrences are dominant while FSM1 is better solution if transient errors occurrences are of the same importance. Further fault modeling and qualitative analysis of aging phenomenon are required to confirm the qualitative studies above.

The hybrid fault-tolerant architecture is originally designed for stand-alone logic circuits. However, advanced digital systems such as microprocessors, pipeline architectures are used to increase system speed without frequency scaling, which is limited by the power wall. Existing fault-tolerant techniques such as Razor and Razor II allow the detection and correction of SETs and timing errors at combinational part of these architectures. However, none of them is effective for hard error. In this manuscript, we have qualitatively demonstrated that the hybrid fault-tolerant can be implemented for pipeline architectures, and provide protections for hard, SETs and timing errors. Concrete implementation as well as evaluation of this method may be the subject of further researches.

Finally, we have study the possibility to add SEU tolerance for sequential elements of the hybrid fault-tolerant architecture. As this method is already able to deal with SETs and timing errors, we can combined it with register-level SEU protection methods in order to reduce area overhead compared to bit-level techniques such as Razor and Razor II, which also target SETs and timing errors. Using the register-level method proposed in [IMH11], we have demonstrated that high level sensitive latches of the input register and all latches of the output registers must be protected. Reducing the area overhead of such architecture may also be studied further in other researches.

Appendix A

HDL Description of Fault-Tolerant Architectures

This appendix provides HDL codes that are used to describe different architectures used for evaluations in Chapter 3. It is divided in three sections. The first section consists of combinational logics extraction from original logic circuits. The second and third section detail HDL codes for the hybrid fault-tolerant and TMR architectures, respectively.

A1. Combinational logic extraction

As we have seen in Chapter 3, combinational parts are extracted from ITC'99 benchmark circuits by removing all D flip-flops from the original netlist (Verilog). For each flip-flop removed, a new primary output (nPO) and a new primary input (nPI) are added.

An example of combinational logic extraction is detailed below for circuit b01 of ITC'99 benchmark. In this example, differences in the extracted netlist compared the orginal netlist are in bold police.

Original netlist	Netlist of extracted combinational part
module b01 (LINE1, LINE2, OUTP_REG, OVERFLW_REG, CLK);	module b01 (LINE1, LINE2, OUTP_REG, OVERFLW_REG, CLK, **nPI1, nPI2, nPI3, nPI4, nPI5, nPO1, nPO2, nPO3, nPO4, nPO5**);
input LINE1,LINE2,CLK; output OUTP_REG,OVERFLW_REG;	input LINE1,LINE2,CLK; output OUTP_REG,OVERFLW_REG;
	input nPI1, nPI2, nPI3, nPI4, nPI5; **output nPO1, nPO2, nPO3, nPO4, nPO5**
//Begin sequential part	//Begin sequential part
dff dff_1 (STATO_REG_2_,U34,CLK);	**buf buf_1 (nPO1, U34);** **buf buf_2 (STATO_REG_2_, nPI1);**
dff dff_2 (STATO_REG_1_,U35,CLK);	**buf buf_3 (nPO2, U35);** **buf buf_4 (STATO_REG_1_, nPI2);**
dff dff_3 (STATO_REG_0_,U36,CLK);	**buf buf_5 (nPO3, U36);** **buf buf_6 (STATO_REG_0_, nPI3);**
dff dff_4 (OUTP_REG,U37,CLK);	**buf buf_7 (nPO4, U37);** **buf buf_8 (OUTP_REG, nPI4);**
dff dff_5 (OVERFLW_REG,U48,CLK);	**buf buf_9 (nPO5, U48);** **buf buf_10 (OVERFLW_REG, nPI5);**
//End sequential part	//End sequential part
//Begin combinational part	//Begin combinational part

```
……………

//End combinational part
endmodule
```

```
……………

//End combinational part
endmodule
```

A2. RTL descriptions of the hybrid fault-tolerant architecture

As detailed in Chapter 2, the hybrid fault-tolerant architecture is divided in different modules: input register Reg_in, input demultiplexers Demux, combinational logics CL1, CL2 and CL3, output multiplexer Mux, output register Reg_out, the pseudo-dynamic comparator, the control logic module (Figure 2.24). Consequently, HDL codes of the hybrid fault-tolerant architecture are composed of: different sub-modules and a top-level module that connect them together. Note that there is two version of top-level module: with and without fault injections.

Top-level module without fault injections

The following Verilog codes describe the fault-free top-level module of the hybrid fault tolerant architecture (Figure 2.24). Note this description use two parameters nbIn and nbOut, which correspond to input and output numbers of the original combinational logic.

```
module hybrid(CLK, resetControl, PI, PO, error);

    //parameter declarations
        parameter nbIn=0;           //input number of combinational part
        parameter nbOut=0;      //output number of combinational part

    //input declarations
        input CLK, resetControl;
        input [nbIn-1:0] PI;

    //output declarations
        output [nbOut-1:0] PO;
        output error;

    //internal signal declarations
        wire [nbIn-1:0] vin;
        wire CLKRegin, CRegin;
        wire d1,d2,d3;
        wire [nbIn-1:0] i1,i2,i3;
        wire [nbOut-1:0] o1,o2,o3;
        wire [nbOut-1:0] vout1,vout2;
        wire m1,m2;
        wire error,DC,reset;

    //input register Reg_in
        shadow_reg      #(nbIn)         (.CLK(CLK), .CLKRegin(CLKRegin), .CRegin(CRegin), .vin(PI), .vout(vin));

    //input demultiplexer Demux
        demux           #(nbIn)         (.vin(vin), .vout1(i1), .vout2(i2), .vout3(i3), .d1(d1), .d2(d2), .d3(d3));

    //combinational logic CLs
```

```
        cl1                              (.vin(i1), .vout(o1));
        cl2                              (.vin(i2), .vout(o2));
        cl3                              (.vin(i3), .vout(o3));

    //output multiplexer Mux
        mux             #(nbOut)         (.vin1(o1), .vin2(o2), .vin3(o3), .vout1(vout1), .vout2(vout2), .m1(m1),
        .m2(m2))

    //output register Reg_out
        simple_reg      #(nbOut)         (.CLK(CLK), .vin(vout1),.vout(PO));

    //pseudo-dynamic comparator
        comp            #(nbOut)         (.reset(reset), .DC(DC), .vin1(PO), .vin2(vout2), .vout(error));

    //control logic module
        control         #(nbOut)         (.CLK(CLK), .resetControl(resetControl), .error(error), .DC(DC),
        .reset(reset), .CRegin(CRegin), .CLKRegin(CLKRegin), .m1(m1), .m2(m2), .d1(d1) ,.d2(d2) , .d3(d3));

endmodule
```

Top-level module with fault injections

The following Verilog codes describe the top-level module of the hybrid fault tolerant architecture with fault injection. Differences with the fault-free version are in bold police.

```
module hybrid(CLK, resetControl, PI, PO, error);

    //parameter declarations
        parameter nbIn=0;           //input number of combinational part
        parameter nbOut=0;      //output number of combinational part

    //input declarations
        input CLK, resetControl
        input [nbIn-1:0] PI;
        input fault;

    //output declarations
        output      [nbOut-1:0] PO;
        output error;

    //internal signal declarations
        wire [nbIn-1:0] vin;
        wire CLKRegin, CRegin;
        wire d1,d2,d3;
        wire [nbIn-1:0] i1,i2,i3;
        wire [nbOut-1:0] o1,o2,o3;
        wire [nbOut-1:0] o2f;
        wire [nbOut-1:0] vout1,vout2;
        wire m1,m2;
        wire error,DC,reset;

    //input register Reg_in
        shadow_reg      #(nbIn)          (.CLK(CLK), .CLKRegin(CLKRegin), .CRegin(CRegin), .vin(PI), .vout(vin));

    //input demultiplexer Demux
        demux           #(nbIn)          (.vin(vin), .vout1(i1), .vout2(i2), .vout3(i3), .d1(d1), .d2(d2), .d3(d3));

    //combinational logic CLs
        cl1                              (.vin(i1), .vout(o1));
```

```
        cl2                                  (.vin(i2), .vout(o2));
        cl3                                  (.vin(i3), .vout(o3));

    //fault injection
        assign o2f[nbOut-1:1] = o2[nbOut-1:1];
        assign o2f[0] = fault^o2[0];

    //output multiplexer Mux
        mux             #(nbOut)        (.vin1(o1), .vin2(o2f), .vin3(o3), .vout1(vout1), .vout2(vout2), .m1(m1),
        .m2(m2))

    //output register Reg_out
        simple_reg      #(nbOut)        (.CLK(CLK), .vin(vout1),.vout(PO));

    //pseudo-dynamic comparator
        comp            #(nbOut)        (.reset(reset), .DC(DC), .vin1(PO), .vin2(vout2), .vout(error));

    //control logic module
        control         #(nbOut)        (.CLK(CLK), .resetControl(resetControl), .error(error), .DC(DC),
        .reset(reset), .CRegin(CRegin), .CLKRegin(CLKRegin), .m1(m1), .m2(m2), .d1(d1) ,.d2(d2) , .d3(d3));

endmodule
```

Input register

The following Verilog codes describe sub-module "shadow_reg", which is used in the top-level module of the hybrid fault-tolerant architecture for its input register Reg_in. This sub-module is composed of N modified D flip-flops (Figure 2.16), where N represents its input number.

```
//top-level module
module shadow_reg (CLK, CLKRegin, CRegin, vin, vout);

    //parameter declarations
        parameter N=0; //input number

    //intput declarations
        input CLK, CLKRegin, CRegin;
        input [N-1:0] vin;

    //output declarations
        output  [N-1:0] vout;

    //generate N modified D flip-flop (Figure 2.16)
        genvar i;
        generate
                for (i=0;i<N;i=i+1)
                begin
                        eRegin (.CLK(CLK), .CLKRegin(CLKRegin), .CRegin(CRegin), .D(vin[i]), .Q(vout[i]));
                end;
        endgenerate;
endmodule

//Modified D flip-flop (Figure 2.16)
module eRegin( CLK, CLKRegin, CRegin, D, Q );

     //input and output declarations
        input CLK, CLKRegin, CRegin, D;
        output Q;
```

```
    //internal signal declarations
        wire   enable;
        reg D_DFF,Q, Q_DLL;

    //main D flip-flop DFF
        always @(posedge CLK)
            begin: DFF
                Q=D_DFF;
            end;

    //shadow latch DLL
        always @(enable)
            begin: DLL
                if (enable==1'b0)
                begin
                    Q_DLL=Q;
                end;
            end;

    //multiplexer
        always @(D or Q_DLL or Cregin)
            begin: MUX
                if (CRegin==1'b1)
                    begin
                        D_DFF=Q_DLL;
                    end
                else
                    begin
                        D_DFF=D;
                    end;
            end;

    // OR gate
        or (enable, CLKRegin, CRegin);
endmodule
```

Input demultiplexer

The following Verilog codes describe sub-module "demux", which is used in the top-level module of the hybrid fault-tolerant architecture for its input demultiplexer Demux. This sub-module is composed N elementary demultiplexers (Figure 2.25), where N represents its input number.

```
//top-level module
module demux (vin, d1, d2, d3, vout1, vout2, vout3);

    //parameter declarations
        parameter N=0; //input number

    //input declarations
        input d1,d2,d3;
        input [N-1:0] vin;

    //output declarations
        output  [N-1:0] vout1,vout2,vout3;

    //generate N elementary input demultiplexers eDmux (Figure 2.25)
        genvar i;
        generate
        for (i=0;i<N;i=i+1)
```

```
        begin
            eDmux (vin[i], d1, d2, d3, vout1[i], vout2[i], vout3[i]);
        end;
        endgenerate;

endmodule
```

//elementary input demultiplexer eDmux (Figure 2.25)

```
module eDmux(vin, d1, d2, d3, i1, i2, i3);

    //input declarations
        input vin;          //data
        input d1,d2,d3;     //control bits

    //output declarations
        output i1,i2,i3;

     and gat1 (i1,vin,d1);
     and gat2 (i2,vin,d2);
     and gat3 (i3,vin,d3);
endmodule
```

Output multiplexer

The following Verilog codes describe sub-module "mux", which is used in the top-level module of the hybrid fault-tolerant architecture for its output multiplexer Mux. This sub-module is composed of N elementary multiplexers, where N represents its input number The following parts detail three possible versions of mux, correspond to three types of elementary multiplexer (Figure 2.27, Figure 2.28, Figure 2.29).

Method 1

//top-level module

```
module mux ( vin1, vin2, vin3, m1, m2, vout1, vout2);

    //parameter declarations
        parameter N=0; //input number

    //input declarations

        input m1,m2;                    //control bits
        input [N-1:0] vin1, vin2, vin3;  //data

    //output declarations
        output  [N-1:0] vout1,vout2;

    //generate N elementary output multiplexers eMux
        genvar i;
        generate
        for (i=0;i<N;i=i+1)
        begin
            eMux (.vin1(vin1[i]), .vin2(vin2[i]), .vin3(vin3[i]),.m1( m1), .m2(m2), .vout1(vout1[i]), .vout2(vout2[i]) );
        end;
        endgenerate

endmodule
```

//elementary output multiplexer eDmux using Method 1 (Figure 2.27)

```
module eMux (m1, m2, vin1, vin2, vin3, vout1, vout2);
```

```
	//input declarations
		input m1, m2;		//control bits
		input vin1, vin2, vin3;	//data

	//output declarations
		output vout1, vout2;

	//internal signal declarations
		wire x1,x2;

	mux2 ( .A(vin1), .B(vin2), .S(m1), .Z(vout1) );
	mux2 ( .A(vin3), .B(vin2), .S(m2), .Z(vout2) );

endmodule

//2:1 multiplexer
module mux2 (S, A, B, Z);
	input S, A, B;
	output Z;
	assign Z= (S & B)|(~S & A);
endmodule
```

Method 2

```
// top-level module
module mux ( vin1, vin2, vin3, m1, m2, vout1, vout2);

	//parameter declarations
		parameter N=0; //input number

	//input declarations

		input m1,m2;				//control bits
		input [N-1:0] vin1, vin2, vin3;	//data

	//output declarations
		output  [N-1:0] vout1,vout2;

	//generate N elementary output multiplexers eMux
		genvar i;
		generate
		for (i=0;i<N;i=i+1)
		begin
			eMux (.vin1(vin1[i]), .vin2(vin2[i]), .vin3(vin3[i]),.m1( m1), .m2(m2), .vout1(vout1[i]), .vout2(vout2[i]) );
		end;
		endgenerate

endmodule

//elementary output multiplexer eDmux using Method 2 (Figure 2.28)
module eMux (m1, m2, vin1, vin2, vin3, vout1, vout2 );

	//input declarations
		input m1, m2;		//control bits
		input vin1, vin2, vin3;	//data

	//output declarations
		output vout1, vout2;
```

```
        //internal signal declarations
                wire x1,x2;

        mux2 ( .A(vin3), .B(vin2), .S(m1), .Z(x1) );
        mux2 ( .A(vin1), .B(vin3), .S(m1), .Z(x2) );
        mux2 ( .A(x1), .B(vin1), .S(m2), .Z(vout1) );
        mux2 ( .A(x2), .B(vin2), .S(m2), .Z(vout2) );

endmodule

//2:1 multiplexer
module mux2 (S, A, B, Z);
        input S, A, B;
        output Z;
        assign Z= (S & B)|(~S & A);
endmodule
```

Method 3 (with tri-state buffers)

```
// top-level
module mux ( vin1, vin2, vin3, m1, m2, m3, vout1, vout2);

        //parameter declarations
                parameter N=0; //input number

        //input declarations

                input m1, m2, m3;                //control bits
                input [N-1:0] vin1, vin2, vin3;  //data

        //output declarations
                output  [N-1:0] vout1, vout2;

        //generate N elementary output multiplexers eMux
                genvar i;
                generate
                for (i=0;i<N;i=i+1)
                begin
                        eMux (.vin1(vin1[i]), .vin2(vin2[i]), .vin3(vin3[i]),.m1( m1), .m2(m2),.m3(m3), .vout1(vout1[i]),
                        .vout2(vout2[i]) );
                end;
                endgenerate

endmodule

//elementary output multiplexer eDmux using tri-state buffers (Figure 2.29)
module eMux (m1, m2, vin1, vin2, vin3, vout1, vout2 );

        //input declarations
                input m1, m2, m3;        //control bits
                input vin1, vin2, vin3;  //data

        //output declarations
                output vout1, vout2;

        tbuf (.S(m3), .A(vin1), .Z(vout1));
        tbuf (.S(m1), .A(vin2), .Z(vout1));
        tbuf (.S(m2), .A(vin3), .Z(vout1));

        tbuf (.S(m3), .A(vin2), .Z(vout2));
```

```
            tbuf (.S(m1), .A(vin3), .Z(vout2));
            tbuf (.S(m2), .A(vin1), .Z(vout2));

endmodule

//tri-state buffer
module tbuf (S, A, Z);
         input S, A;
         output Z;
         assign Z=(S)?1'bz:A;
endmodule
```

Output register

The following Verilog codes describe sub-module "simple_reg", which is used in the top-level module of the hybrid fault-tolerant architecture for its output register Reg_out. This sub-module is composed of N D flip-flops where N represents its input number.

```
//top-level module
module simple_reg (vin,vout, CLK);

         //parameter declarations
                parameter N=0; //input number

         //input declarations
                input CLK;          //clock
                input [N-1:0] vin;  //data

          //output declarations
                output [N-1:0] vout;
                reg [N-1:0] vout;

          always@(posedge CLK)
                begin
                        vout<=vin;
                end;

endmodule
```

Pseudo-dynamic comparator

The following Verilog codes describe sub-module "comp", which is used in the top-level module of the hybrid fault-tolerant architecture for its pseudo-dynamic comparator (Figure 2.10). This description uses a parameter N corresponding to size of the two input vectors to be compared.

```
//top-level module
module comp ( reset, DC, vin1, vin2, vout);

         //parameter declarations
                parameter N=0;    //number of bits per input vetor

         //input declarations
                input reset,DC;            //control signals
                input [N-1:0] vin1,vin2;   //data

         //ouput declarations
                output  vout;

          //internal signal declarations
                wire [N-1:0] oXor;                      //output of XOR gates
```

```
        wire [((N+3)-(N+3)%4)/4-1:0] oDor;            //output of DOR gates

    //local comparison stage (Figure 2.10)
        assign oXor= vin1^vin2;

    //generate DOR gates for the first layer of global comparison stage (Figure 2.10)
        genvar i;
        generate
        for (i=0;i<((N+3)-(N+3)%4)/4-1;i=i+1)
        begin
            DOR4_X1 ( .DC(DC), .RESET(reset), .D0(oXor[i*4]), .D1(oXor[i*4+1]), .D2(oXor[i*4+2]), .D3(oXor[i*4+3]),
            .OUT(oDor[i]));
        end
        for (i=((N+3)-(N+3)%4)/4-1;i<((N+3)-(N+3)%4)/4;i=i+1)
        begin
                if (N%4==0)
                begin
                    DOR4_X1 ( .DC(DC), .RESET(reset), .D0(oXor[i*4]), .D1(oXor[i*4+1]), .D2(oXor[i*4+2]),
                    .D3(oXor[i*4+3]), .OUT(oDor[i]));
                end
                if (N%4==1)
                begin
                    DOR4_X1 ( .DC(DC), .RESET(reset), .D0(oXor[i*4]), .D1(oXor[i*4]), .D2(oXor[i*4]),
                    .D3(oXor[i*4]), .OUT(oDor[i]));
                end
                if (N%4==2)
                begin
                    DOR4_X1 ( .DC(DC), .RESET(reset), .D0(oXor[i*4]), .D1(oXor[i*4]), .D2(oXor[i*4+1]),
                    .D3(oXor[i*4+1]), .OUT(oDor[i]));
                end
                if (N%4==3)
                begin
                    DOR4_X1 ( .DC(DC), .RESET(reset), .D0(oXor[i*4]), .D1(oXor[i*4+1]), .D2(oXor[i*4+2]),
                    .D3(oXor[i*4]), .OUT(oDor[i]));
                end
        end;
        endgenerate;

    //XOR tree for the second layer of global comparison stage (Figure 2.10)
        assign vout = |oDor[((N+3)-(N+3)%4)/4-1:0];

endmodule
```

Control logic module

The following HDL codes describe sub-module “control”, which is used in the top-level module of the hybrid fault-tolerant architecture for its control logic module. As discussed in Chapter 2, this sub-module is divided in three parts. The first part (“submodule1”) generates control signals for the input register and the pseudo-dynamic comparator (Figure 2.20, Figure 2.21). In order to meet different timing constraints of these signals, it is implemented using different buffers in the standard cell library [NOCL]. The second part (“submodule2”) corresponds to the FSM. There are two version of FSM as described in Figure 2.30. The third part (“submodule3”) generates control signals for the input demultiplexers and the output multiplexer from outputs of the FSM. Except of FSM modules which are described in VHDL, others modules are described in Verilog.

```
//top-level module
module control (CLK, error, resetControl, CLKRegin, CRegin, d1, d2, d3, m1, m2, DC, reset);
```

```
    //input declarations
        input CLK;                     //global clock signal
        intput error;                  //output of the pseudo-dynamic comparator
        input  resetControl;           //reset the complete architecture (Figure 2.24)

    //output declarations
        output CLKRegin, CRegin;       //control signals for input register Reg_in
        output d1,d2,d3;               //control signals for input demultiplexer Demux
        output m1,m2;                  //control signals for output multiplexer Mux
        output DC,reset;               //control signals for the pseudo-dynamic comparator

    //internal signal declarations
        wire f1,f2;                    //signal generated by the FSM (Table 2.4)

    //submodule1: clock generator (Figure 2.20, Figure 2.21)
        submodule1 sub1 (.CLK(CLK), .resetControl(resetControl), .error(error), .DC(DC), .reset(reset),
        .CRegin(CRegin), .CLKRegin(CLKRegin));

    //submodule2: FSM (Figure 2.30)
        submodule2 sub2 ( .CLKRegin(CLKRegin), .CRegin(CRegin), .resetControl(resetControl), .f1(f1), .f2(f2) );

    //submodule3: Control signals generator for input demultiplexers and output multiplexer (Figure 2.32)
        submodule3 sub3 (.f1(f1), .f2(f2), .m1(m1), .m2(m2), .d1(d1), .d2(d2), .d3(d3));

endmodule

//submodule1: clock generator (Figure 2.20, Figure 2.21)
//depending on different standard cell library , buf1 and buf2 in of this module are replaced by different buffer cells to
satisfy timing constraints stated in Chapter 2 [NOCL]
module submodule1 (CLK, resetControl, error, DC, reset, CRegin, CLKRegin);

    //input declarations
        input CLK, resetControl, error;
    //output declarations
        output DC, reset, CRegin, CLKRegin;

    // DC generator
        buf2 bufDC (.A(CLK), .Z(DC));
    // CLKRegin generator
        buf2 bufDC1 (.A(CLK), .Z(DC1));
        buf1 bufCLKRegin1 (.A(DC1), .Z(CLKRegin1));
        buf2 bufCLKRegin2 (.A(CLKRegin1), .Z(CLKRegin2));
        INV_X1 invCLKRegin (.A(CLKRegin2), .ZN(CLKRegin3));
        OR2_X1 ORCLKRegin (.A1(CLKRegin1), .A2(CLKRegin3), .ZN(CLKRegin));

    //CLKComp
        buf1 bufRComp (.A(CLK), .Z(CLKComp1));
        INV_X1 invRComp (.A(CLKComp1), .ZN(CLKComp2));
        NAND2_X1 andRComp (.A1(CLK), .A2(CLKComp2), .ZN(CLKComp));

    //Control
        DFFR_X1 DFFR (.CK (DC), .QN(Control), .RN (resetControl), .D(CRegin));

    //reset
        OR2_X1 orReset (.A1(CLKComp), .A2(Control), .ZN(reset));

    //CRegin
        AND2_X1 ANDCregin (.A1(Control), .A2(error), .ZN(CRegin));
```

```
endmodule
```

//submodule2 version FMS1 (Figure 2.30-a)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity fsm is
   Port ( CLKRegin : in  STD_LOGIC;
        CRegin : in  STD_LOGIC;
                    resetControl: in STD_LOGIC;
        f1 : out  STD_LOGIC;
        f2 : out  STD_LOGIC);
end fsm;

architecture behavior of fsm is

        type typeState is (A1,A2,A3,A4,A5,A6,B1,B2,B3,B4,B5,B6,C1,C2,C3,C4,C5,C6,STOP);
            -- State Ax: f1=0;f2=1; (CL1 and CL2)
            -- State Bx: f1=1;f2=0; (CL2 and CL3)
            -- State Cx: f1=0;f2=0;      (CL3 and CL1)
            -- State STOP: f1=1;f2=1;
        signal currentState,nextState : typeState;

begin

            state_reg: process (CLKRegin,resetControl)
                begin
                        if (CLKRegin'event and CLKRegin='0') then
                            if resetControl='0' then
                                currentState<=A1;
                            else
                                currentState <= nextState;
                            end if;
                        end if;

                end process;

            state_define: process (currentState,CRegin)
                begin
                        case currentState is
                            when A1 =>  f1<='0';f2<='1';
                                            if CRegin='0' then nextState <= A1; end if;
                                            if CRegin='1' then nextState <= A2; end if;
                            when A2 =>  f1<='0';f2<='1';
                                            if CRegin='0' then nextState <= A1; end if;
                                            if CRegin='1' then nextState <= B3; end if;
                            when A3 =>  f1<='0';f2<='1';
                                            if CRegin='0' then nextState <= A1; end if;
                                            if CRegin='1' then nextState <= A4; end if;
                            when A4 =>  f1<='0';f2<='1';
                                            if CRegin='0' then nextState <= A1; end if;
                                            if CRegin='1' then nextState <= B5; end if;
                            when A5 =>  f1<='0';f2<='1';
                                            if CRegin='0' then nextState <= A1; end if;
                                            if CRegin='1' then nextState <= A6; end if;
                            when A6 =>  f1<='0';f2<='1';
                                            if CRegin='0' then nextState <= A1; end if;
                                            if CRegin='1' then nextState <= STOP; end if;
```

```
when B1 =>  f1<='1';f2<='0';
            if CRegin='0' then nextState <= B1; end if;
            if CRegin='1' then nextState <= B2; end if;
when B2 =>  f1<='1';f2<='0';
            if CRegin='0' then nextState <= B1; end if;
            if CRegin='1' then nextState <= C3; end if;
when B3 =>  f1<='1';f2<='0';
            if CRegin='0' then nextState <= B1; end if;
            if CRegin='1' then nextState <= B4; end if;
when B4 =>  f1<='1';f2<='0';
            if CRegin='0' then nextState <= B1; end if;
            if CRegin='1' then nextState <= C5; end if;
when B5 =>  f1<='1';f2<='0';
            if CRegin='0' then nextState <= B1; end if;
            if CRegin='1' then nextState <= B6; end if;
when B6 =>  f1<='1';f2<='0';
            if CRegin='0' then nextState <= B1; end if;
            if CRegin='1' then nextState <= STOP; end if;



when C1 =>  f1<='0';f2<='0';
            if CRegin='0' then nextState <= C1; end if;
            if CRegin='1' then nextState <= C2; end if;
when C2 =>  f1<='0';f2<='0';
            if CRegin='0' then nextState <= C1; end if;
            if CRegin='1' then nextState <= A3; end if;
when C3 =>  f1<='0';f2<='0';
            if CRegin='0' then nextState <= C1; end if;
            if CRegin='1' then nextState <= C4; end if;
when C4 =>  f1<='0';f2<='0';
            if CRegin='0' then nextState <= C1; end if;
            if CRegin='1' then nextState <= A5; end if;
when C5 =>  f1<='0';f2<='0';
            if CRegin='0' then nextState <= C1; end if;
            if CRegin='1' then nextState <= C6; end if;
when C6 =>  f1<='0';f2<='0';
            if CRegin='0' then nextState <= C1; end if;
            if CRegin='1' then nextState <= STOP; end if;

when STOP =>  f1<='1'; f2<='1';
            nextState<=STOP;

    end case;
  end process;


end behavior ;
```

//submodule2 version FMS1 (Figure 2.30-b)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity fsm is
   Port ( CLKRegin : in  STD_LOGIC;
      CRegin : in  STD_LOGIC;
                resetControl: in STD_LOGIC;
      f1 : out  STD_LOGIC;
```

```
        f2 : out  STD_LOGIC);
end fsm;

architecture behavior of fsm is

        type typeState is (A1,A2,A3,A4,A5,A6,B1,B2,B3,B4,B5,B6,C1,C2,C3,C4,C5,C6,STOP);
            -- State Ax: f1=0;f2=1; (CL1 and CL2)
            -- State Bx: f1=1;f2=0; (CL2 and CL3)
            -- State Cx: f1=0;f2=0;      (CL3 and CL1)
            -- State STOP: f1=1;f2=1;
        signal currentState,nextState : typeState;

begin

            state_reg: process (CLKRegin, resetControl)
                begin
                            if (CLKRegin'event and CLKRegin='0') then
                                if resetControl='0' then
                                    currentState<=A1;
                                else
                                    currentState <= nextState;
                                end if;
                            end if;

                end process;

            state_define: process (currentState,CRegin)
                begin
                            case currentState is
                                when A1 =>  f1<='0';f2<='1';
                                                    if CRegin='0' then nextState <= A1; end if;
                                                    if CRegin='1' then nextState <= B2; end if;
                                when A2 =>  f1<='0';f2<='1';
                                                    if CRegin='0' then nextState <= A1; end if;
                                                    if CRegin='1' then nextState <= B3; end if;
                                when A3 =>  f1<='0';f2<='1';
                                                    if CRegin='0' then nextState <= A1; end if;
                                                    if CRegin='1' then nextState <= B4; end if;
                                when A4 =>  f1<='0';f2<='1';
                                                    if CRegin='0' then nextState <= A1; end if;
                                                    if CRegin='1' then nextState <= B5; end if;
                                when A5 =>  f1<='0';f2<='1';
                                                    if CRegin='0' then nextState <= A1; end if;
                                                    if CRegin='1' then nextState <= B6; end if;
                                when A6 =>  f1<='0';f2<='1';
                                                    if CRegin='0' then nextState <= A1; end if;
                                                    if CRegin='1' then nextState <= STOP; end if;


                                when B1 =>  f1<='1';f2<='0';
                                                    if CRegin='0' then nextState <= B1; end if;
                                                    if CRegin='1' then nextState <= C2; end if;
                                when B2 =>  f1<='1';f2<='0';
                                                    if CRegin='0' then nextState <= B1; end if;
                                                    if CRegin='1' then nextState <= C3; end if;
                                when B3 =>  f1<='1';f2<='0';
                                                    if CRegin='0' then nextState <= B1; end if;
                                                    if CRegin='1' then nextState <= C4; end if;
                                when B4 =>  f1<='1';f2<='0';
```

```
                                        if CRegin='0' then nextState <= B1; end if;
                                        if CRegin='1' then nextState <= C5; end if;
                        when B5 =>  f1<='1';f2<='0';
                                        if CRegin='0' then nextState <= B1; end if;
                                        if CRegin='1' then nextState <= C6; end if;
                        when B6 =>  f1<='1';f2<='0';
                                        if CRegin='0' then nextState <= B1; end if;
                                        if CRegin='1' then nextState <= STOP; end if;



                        when C1 =>  f1<='0';f2<='0';
                                        if CRegin='0' then nextState <= C1; end if;
                                        if CRegin='1' then nextState <= A2; end if;
                        when C2 =>  f1<='0';f2<='0';
                                        if CRegin='0' then nextState <= C1; end if;
                                        if CRegin='1' then nextState <= A3; end if;
                        when C3 =>  f1<='0';f2<='0';
                                        if CRegin='0' then nextState <= C1; end if;
                                        if CRegin='1' then nextState <= A4; end if;
                        when C4 =>  f1<='0';f2<='0';
                                        if CRegin='0' then nextState <= C1; end if;
                                        if CRegin='1' then nextState <= A5; end if;
                        when C5 =>  f1<='0';f2<='0';
                                        if CRegin='0' then nextState <= C1; end if;
                                        if CRegin='1' then nextState <= A6; end if;
                        when C6 =>  f1<='0';f2<='0';
                                        if CRegin='0' then nextState <= C1; end if;
                                        if CRegin='1' then nextState <= STOP; end if;

                        when STOP =>  f1<='1'; f2<='1';
                                        nextState<=STOP;

                end case;
        end process;


end behavior ;
```

// submodule3: Control signals generator for input demultiplexers and output multiplexer (Figure 2.32)

```
module submodule3 (f1,f2,m1,m2,d1,d2,d3);

        input f1,f2;

        output m1,m2,d1,d2,d3;

        assign m1=f1;
        assign m2=f2;
        assign d1=~f1;
        assign d2=f1^f2;
        assign d3=~f2;
endmodule
```

A3. RTL descriptions of TMR architectures

This section details Verilog description of Partial (Figure 3.3-a) and Full TMR architectures (Figure 3.3-b). Each architecture is combined of input registers, combinational logic modules, output registers and word voter. In TMR methods, all registers are made of D flip-flop. Consequently, “simple_register” sub-module described in previous section can be used. Therefore, the following sub-sections only detail Verilog descriptions for top-level modules (“TMR1” and “TMR2”) of the two TMR versions, as well as the word voter (“voter_tmr”) which is identical for both versions.

Top-level module of Partial TMR architecture

```
//top-level module
module TMR1 (PI, PO, CLK, error);

        //parameter declarations
            parameter nbIn=0;         //input number of combinational logic module
            parameter nbOut=0;        // output number of combinational logic module

        //input declarations
            input CLK;                //global clock signal
            input [nbIn-1:0] PI;      //primary input vector

        //output declarations
            output [nbOut-1:0] PO;    //primary output vector
            output error;             //error signal

        //internal signal declarations
            wire [nbIn-1:0] i;
            wire [nbOut-1:0] o1, o2, o3, vout;

        //input register
            simple_reg  #(nbIn)           reg_in (.CLK(CLK), .vin(PI), .vout(i));

        //combinational logics
            cl1                       cl1 (.vin(i), .vout(o1));
            cl2                       cl2 (.vin(i), .vout(o2));
            cl3                       cl3 (.vin(i), .vout(o3));

        //voter
            voter_tmr   #(nbOut)      voter (.vin1(o1), .vin2(o2), .vin3(o3), .vout(vout),.error(error));

        //output register
            simple_reg  #(nbOut)      reg_out (.CLK(CLK), .vin(vout),.vout(PO));

endmodule
```

Top-level module of Full TMR architecture

```
//top-level module
module TMR2 (PI, PO, CLK, error);

        //parameter declarations
            parameter nbIn=0;         //input number of combinational logic module
            parameter nbOut=0;        // output number of combinational logic module

        //input declarations
            input CLK;                //global clock signal
            input [nbIn-1:0] PI;      //primary input vector
```

```
    //output declarations
        output [nbOut-1:0] PO;   //primary output vector
        output error;            //error signal

    //internal signal declarations
        wire [nbIn-1:0] i1, i2, i3;
        wire [nbOut-1:0] o1, o2, o3
        wire [nbOut-1:0] vout1, vout2, vout3;

    //input registers
        simple_reg  #(nbIn)         (.CLK(CLK), .vin(PI), .vout(i1));
        simple_reg  #(nbIn)         (.CLK(CLK), .vin(PI), .vout(i2));
        simple_reg  #(nbIn)         (.CLK(CLK), .vin(PI), .vout(i3));

    //combinational logics
        cl1                     cl1 (.vin(i1), .vout(o1));
        cl2                     cl2 (.vin(i2), .vout(o2));
        cl3                     cl3 (.vin(i3), .vout(o3));

    //output registers
        simple_reg  #(nbOut)    (.CLK(CLK), .vin(o1),.vout(vout1));
        simple_reg  #(nbOut)    (.CLK(CLK), .vin(o2),.vout(vout2));
        simple_reg  #(nbOut)    (.CLK(CLK), .vin(o3),.vout(vout3));

    //voter
        voter_tmr   #(nbOut)    voter (.vin1(vout1), .vin2(vout2), .vin3(vout3), .vout(PO),.error(error));

endmodule
```

Word-voter

```
//top-level module
module voter_tmr (vin1, vin2, vin3, vout, error);

    //parameter declarations
        parameter N=0;

    //input declarations
        input [N-1:0] vin1, vin2, vin3;

    //output declarations
        output  [N-1:0] vout;
        output error;

    //internal signal declarations
        wire match12, match23, match31;
        wire [N-1:0] M31;

    //verify if there is at least two identical input vectors
    //using submodule Match: compare two vectors, returns logic-1 if they are identical and logic-0 otherwise
        Match #(N) gateMatch12 (.vin1(vin1), .vin2(vin2), .vout(match12));
        Match #(N) gateMatch23 (.vin1(vin2), .vin2(vin3), .vout(match23));
        Match #(N) gateMatch31 (.vin1(vin3), .vin2(vin1), .vout(match31));
        nor gateNor (error, match12, match23, match31);

    //generate output vector
        assign M31= {N {match31} };
        assign vout= (M31&vin1)|(~M31&vin2);
```

```
endmodule
```

//submodule Match: compare two vectors, returns logic-1 if they are identical and logic-0 otherwise

```
module Match (vin1, vin2, vout);

        parameter N=0;
        Input [N-1:0] vin1, vin2;
        output vout;
        wire [N-1:0] oXnor;
        assign oXnor= vin1~^vin2;
        assign vout=&(oXnor);

endmodule
```

Scientific Contributions

Journal

[TVL13] D. A. Tran, A. Virazel, A. Bosio, L. Dilillo, P. Girard, S. Pravossoudovitch, A. Todri, H. –J. Wunderlich, "A New Hybrid Fault-Tolerant Architecture for Digital CMOS Circuits and Systems", submitted to IEEE Tran. on VLSI Systems, October 2012.

Conferences

[VTS12] D. A. Tran, A. Virazel, A. Bosio, L. Dilillo, P. Girard, A. Todri, H. –J. Wunderlich, M. E. Imhof, "A Pseudo-Dynamic Comparator for Error Detection in Fault-Tolerant Architectures", Proc. of the IEEE VLSI Test Sym. (VTS'12), pg. 50-55, 2012.

[ATS11] D. A. Tran, A. Virazel, A. Bosio, L. Dilillo, P. Girard, S. Pravossoudovitch, H. –J. Wunderlich, "A Hybrid Fault-Tolerant Architecture for Robustness Improvement of Digital Circuits", Proc. of the IEEE Asian Test Sym. (ATS'11), pg. 136-141, 2011.

[ITC10] D. A. Tran, A. Virazel, A. Bosio, L. Dilillo, P. Girard, S. Pravossoudovitch, H. –J. Wunderlich, "Parity Prediction Synthesis for Nano-Electronic Gate Design", Proc. of the IEEE Int. Test Conf. (ITC'10), Poster.

Seminars and Workshop

[GDR11] D. A. Tran, A. Virazel, A. Bosio, L. Dilillo, P. Girard, S. Pravossoudovitch, H. –J. Wunderlich, "Architecture Tolérante aux Fautes pour la Robustesse des Circuits CMOS", Colloque National du GDR SOC-SIP, France, June 2011.

[JNR11] D. A. Tran, A. Virazel, A. Bosio, L. Dilillo, P. Girard, S. Pravossoudovitch, H. –J. Wunderlich, "Architecture Tolérante aux Fautes pour la Robustesse des Circuits CMOS", Journées Nationales du Réseau Doctoral en Micro-Nanoélectronique (JNRDM'11), France, May 2011.

[SET11] D. A. Tran, A. Virazel, A. Bosio, L. Dilillo, P. Girard, S. Pravossoudovitch, H. –J. Wunderlich, "A Hybrid Fault-Tolerant Architecture for Robustness Improvement of Digital Circuits", South European Test Seminar (SET'11), France, March 2011.

[GDR10] D. A. Tran, A. Virazel, A. Bosio, L. Dilillo, P. Girard, S. Pravossoudovitch, H. –J. Wunderlich, "Tolérance aux Fautes et Rendement de Fabrication", Colloque National du GDR SOC-SIP, France, June 2010.

[SET10] D. A. Tran, A. Virazel, A. Bosio, L. Dilillo, P. Girard, S. Pravossoudovitch, "Yield Enhancement using Fault-Tolerant Architectures", South European Test Seminar (SET'10), Austria, March 2010.

References

[ALM03] S. Almukhaizim, Y. Makris, “Fault Tolerant Design of Combinational and Sequential Logic Based on a Parity Check Code”, Proc. of the 18th IEEE Int. Sym. on Defect and Fault-tolerance in VLSI Systems (DFT '03), pg. 563-570, 2003.

[BAK10] R. J. Baker, “CMOS: Circuit Design, Layout, and Simulation”, Ed. Wiley-IEEE Press, 2010.

[BAU05] R. Baumann, “Soft errors in advanced computer systems”, IEEE Design & Test of Computers, Vol. 22, No. 3, pf. 258–266, May–June 2005.

[BER61] J. M. Berger, “A Note on Error Detection Codes for Asymmetric Channels”, Information and Control, Vol. 4, pg. 68-73, 1961.

[BER99] K. Bernstein, K. Carrig, C. Durham, P. Hansen, D. Hogenmiller, E. Nowak, and N. Roher, “High Speed CMOS Design Styles”, Kluwer Academic Publishers, 1999.

[BOS85] B. Bose, D. J. Lin, “Systematic Unidirectional Error-Detecting Codes”, IEEE Trans. Comp., pg. 1026-1032, November 1985.

[BUS02] M. L. Bushnell, V. D. Agrawal, “Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits”, Ed. Kluwer Academic Publishers, 2002.

[CRI07] Dale L. Critchlow, “Recollections on MOSFET Scaling”, IEEE SolidState Circuits Newsletter 2007, Vol. 12, Issue 1, pg. 19-22, 2007.

[DAS09] S. Das, C. Tokunaga, S. Pant, W. -H. Ma, S. Kalaiselvan, K. Lai, D. M. Bull, D. T. Blaauw, “Razor II: In Situ Error Detection and Correction for PVT and SER Tolerance”, IEEE J. of Solid-State Circuits, Vol. 44, Issue 1, pg. 32-48, January 2009.

[DCSYN] Synopsys Inc., Design Compiler® User Guide 2011.

[DE94] K. De, C. Natarajan, “RSYN: A System for Automated Synthesis of Reliable Multilevel Circuits”, IEEE Transactions on VLSI Systems, Vol. 2, No 2, pg. 186-195, 1994.

[DEN74] R. Dennard, F. H. Gaensslen, H. -N. Yu, V. L. Rideout, E. Bassous, A. R. Leblanc, “Design of Ion-Implanted MOSFETs with Very Small Physical Dimensions”, IEEE Journal of Solid State Circuits, Vol. SC-9, No. 5, pp. 256-268, October 1974.

[ERN03] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, T. Mudge, “Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation”, Proc. of the 36th Annual IEEE/ACM Int. Sym. on Microarchitecture (MICRO-36), pg. 7-18, December 2003.

[FAN06] L. Fang, M. S. Hsiao, “Bilateral Testing of Nano-scale Fault-tolerant Circuits”, Proc. of IEEE Int. Symp. on Defect and Fault-Tolerance in VLSI Systems, pg. 309-317, 2006.

[FOR09] R. Forsati, K. Faez, F. Moradi, A. Rahbar, “A Fault Tolerant Method for Residue Arithmetic Circuits”, Proc. of the IEEE Int. Conference on Information Management and Engineering, pg. 59-63, 2009.

[GAL98] W. L. Gallagher, E. E. Swartzlander Jr., “Error-Correcting Goldschmidt Dividers Using Time Shared TMR”, Proc. of the 13th Int. Sym. on Defect and Fault-Tolerance in VLSI Systems (DFT’98), pg. 224-232, 1998.

[GIR10] P. Girard, N. Nicolici, X.Wen, “Power-Aware Testing and Test Strategies for Low Power Devices”, Springer, 2010.

[HAR01] D. Harris, S. Naffziger, “Statistical clock skew modeling with data delay variations”, IEEE Trans. VLSI, Vol. 9, No. 6, pg. 888–898, December 2001.

[HEN07] J. L. Hennessy, D. A. Patterson, “Computer Architecture: A Quantitative Approach”, 4th Ed. Morgan Kaufmann Publishers, 2007.

[HSI70] M. Y. Hsiao, “A Class of Optimal Minimum Odd-Weight-Column SEC-DED Codes”, IBM J. of Res. and Develop., Vol. 14, No. 4, pg. 395-401, 1970.

[HSU94] Y. -H. Hsu, E. E. Swartzlander Jr., “Reliability Estimation for Time Redundant Error Correcting Adders and Multipliers”, Proc. of the IEEE Int. Workshop on Defect and Fault-Tolerance in VLSI Systems, pg. 159-167, 1994.

[IMH11] M. E. Imhof, H.-J. Wunderlich, “Soft Error Correction in Embedded Storage Elements”, Proc. of IEEE International On-Line Testing Symposium (IOLTS11), pp. 169-174, 2011.

[INT10] Intel Corporation, “The Evolution of a Revolution”, http://download.intel.com/pressroom/kits/IntelProcessorHistory.pdf.

[ISCAS85] F. Brglez and H. Fujiwara, “A neutral Netlist of 10 Combinatorial Benchmark Circuits and a Target Translator in FORTRAN”, Proc. of Int. Symposium on Circuits and Systems (ISCAS’85), pg. 695-698, 1985.

[ITC99] S. Davidson, “ITC’99 Benchmark Circuits - Preliminary Results”, Proc. of Int. Test Conf. (ITC’99), pg. 1125, 1999.

[ITR03] Semiconductor Industry Association (SIA), “International Technology Roadmap for Semiconductors (ITRS)”, 2003.

[ITR11] Semiconductor Industry Association (SIA), “International Technology Roadmap for Semiconductors (ITRS)”, 2011.

[KO04] S.-B. Ko, J-C. Lo, “Efficient Realization of Parity Prediction Functions in FPGAs”, Journal of Electronic Testing: Theory and Applications (JETTA), Vol. 20, Issue 5, pg. 489-499, October 2004.

[KOR07] I. Koren, C. M. Krishna, “Fault-Tolerant Systems”, Ed. Organ Kaufmann, 2007.

[KUM06] R. Kumar, V. Kursun, “Reversed temperature-dependent propagation delay characteristics in nanometer CMOS circuits”, IEEE Trans. Circuits & Systems, Vol. 53, No. 10, pg. 1078–1082, October 2006.

[LAP95] J. -C.Laprie, J. Arlat, J. -P. Blanquart, A. Costes, Y. Crouzet, Y. Deswarte, J. -C. Fabre, H. Guillermain, M. Kaâinche, K. Kanoun, C. Mazet, D. Powell, C. Rabéjac, P. Thévenod, “Guide de La Sûreté de Fonctionnement”, Cépaduès, 1995.

[LTSPICE] Linear Technology, LTSpice Getting Started Guide 2008.

[LYO62] R. E. Lyons, W. Vanderkulk, "The Use of Triple-Modular Redundancy to Improve Computer Reliability", IBM Journal of Research and Development, Vol. 6, Issue 2, pg. 200-209, April 1962.

[MCH01] J. McHale, "Actel Engineers Use Triple-Module Redundancy in New Rad-Hard FPGA", Military & Aerospace Electronics, 8 August 2001.

[MIT00a] S. Mitra, E. J. McCluskey, "Which Concurrent Error Detection Scheme to Choose?", Proc. of the IEEE Int. Test Conference, pg. 985-994, 2000.

[MIT00b] S. Mitra, E. J. McCluskey, "Word-voter: a new voter design for triple modular redundant systems", Proc. of the IEEE 18th VLSI Test Symposium, pg. 465-470, 2000.

[MOO65] G. E. Moore, "Cramming More Components onto Integrated Circuits", Electronics Magazine, Vol. 38, No. 8, April 1965.

[NEU56] J. V. Neumann, "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components", Automata Studies, ed. C. E. Shannon and J. McCarthy, pg. 43-98, Princeton University Press, 1956.

[NIC03] M. Nicolaidis, N. Achouri, S. Boutobza, "Dynamic Data-bit Memory Built-In Self- Repair", In Proc. of the 2003 IEEE/ACM International Conference on Computer-Aided Design (ICCAD '03), pg. 588-594, 2003.

[NIC05] M. Nicolaidis, L. Anghel, N. Achouri, "Memory Defect Tolerance Architectures for Nanotechnologies", J. of Electronic Testing, Vol. 21, Issue 4, pg. 445-455, August 2005.

[NIC07] M. Nicolaidis, "Graal: A New Fault Tolerant Design Paradigm for Mitigating the Flaws of Deep Nanometric Technologies", Proc. of IEEE International Test Conference (ITC07), pp.1–10, 2007.

[NIC97] M. Nicolaidis, R. O. Duarte, S. Manich, J Figueras, "Fault-Secure Parity Prediction Arithmetic Operators", IEEE Design and Test of Computers, Vol. 14, No. 2, pg. 60-71, 1997.

[NNSIM] Synopsys Inc., NanoSim® User Guide 2011.

[NOCL] Nangate, 45nm Open Cell Library v1.3, http://www.nangate.com, 2009

[PAL11] D. J. Palframan, N. S. Kim, M. H. Lipasti, "Time Redundant Parity for Low-Cost Transient Error Detection", Proc. of IEEE Design, Automation & Test in Europe, pg. 1-6, 2011.

[PAU07] B. C. Paul, K. Kang, H. Kufluoglu, M. A. Alam, K. Roy, "Negative Bias Temperature Instability: Estimation and Design for Improved Reliability of Nanoscale Circuits", IEEE Trans. On Computer-Aided Design of Integrated Circuits and Systems, Vol. 26, No. 4, pg. 743-751, April 2007.

[PDK] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F.Love, "FreePDK: An Open-Source Variation-Aware Design Kit", IEEE Int. Conf. on Microelectronic Systems Education, pg. 173-174, 2007.

[PTM] W. Zhao and Y. Cao, "Predictive technology model for nano-CMOS design exploration", ACM Journal on Emerging Technologies in Computing Systems , Vol. 3, No. 1, 2007.

[PUL07] A. Pullini, F. Angiolini, D. Bertozzi and L. Benini, "Fault Tolerance Overhead in Network-on-Chip Flow Control Schemes", in Proc. of Annual Symposium on Integrated Circuits and System Design, pp. 224-229, 2007.

[RAO70] T. R. N. Rao, "Bi-Residue Error-Correcting Codes for Computer Arithmetic", IEEE Transactions on Computers, Vol. 19, Issue 5, pg. 398–402, May 1970.

[RAO77] T. R. N. Rao, H. J. Reinheimer, "Fault-Tolerant Modularized Arithmetic Logic Units", Proc. of the June 13-16, 1977, National Computer Conference (AFIPS'77), pg. 703-710, 1977.

[REY75] D. A. Reynolds, G. Metze, "Fault Detection Capabilities of Alternating Logic", IEEE Transactions on Computers, Vol. 27, No. 12, pg. 1093-1098, December 1978.

[SCH01] V. Schöber, S. Paul, O. Picot, "Memory Built-In Self-Repair Using Redundant Words", in Proc. of the IEEE International Test Conference (ITC '01), pg. 995-1001, 2001.

[SEL68] F. Sellers, M-Y. Hsiao, L. W. Bearnson, "Error Detection Logic for Digital Computers", McGraw-Hill Book Company, 1968.

[SIE75] D. P. Siewiorek, "Reliability Modeling of Compensating Module Failures in Majority Voting Redundancy", IEEE Transactions on Computers, Vol. 24, pg. 525–533, May 1975.

[STA11] M. Stanisavljevic, M. Schmid, Y. Leblebici, "Reliability of Nanoscale Circuits and Systems", Springer, 2011.

[SU05] Chin-Lung Su, Yi-Ting Yeh, Cheng-Wen Wu, "An Integrated ECC and Redundancy Repair Scheme for Memory Reliability Enhancement", Proc. of the 20th Int. Sym. on Defect and Fault-Tolerance in VLSI Systems (DFT'05), pg. 81-92, 2005.

[TAH95] J. M. Tahir, S. S. Dlay, R. N. G. Naguib, O. R. Hinton, "Fault Tolerant Arithmetic Unit Using Duplication and Residue Codes", Integration, the VLSI Journal, Vol. 18, Issue 2-3, pg. 187-200, June 1995.

[TOU97] N. A. Touba, E. J. McCluskey, "Logic Synthesis of Multilevel Circuits with Concurrent Error Detection", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 16, No. 7, pg. 783-789, July 1997.

[VAL10] S. Valadimas, Y. Tsiatouhas, A. Arapoyanni, "Timing Error Tolerance In Nanometer ICs", Proc. of the 16th IEEE Int. On-Line Testing Symposium, pg. 283 – 288, 2010.

[VIA08] J. Vial, A. Bosio, P. Girard, C. Landrault, S. Pravossoudovitch and A.Virazel, "Using TMR Architectures for Yield Improvement", Int. Symp.on Defect and Fault-tolerance in VLSI Systems, pp. 7-15, 2008.

[VIA09] J. Vial, A. Virazel, A. Bosio, P. Girard, C. Landrault and S.Pravossoudovitch, "Is TMR Suitable for Yield Improvement?", IET Computers and Digital Techniques, vol. 3, No 6, pp. 581-592,November 2009.

[WAN03] J. Wang, W. Wong, S. Wolday, B. Cronquist, J. McCollum, R. Katz, and I. Kleyner, "Single Event Upset and Hardening in 0.15 μm Antifuse-Based Field Programmable Gate Array," IEEE Transactions on Nuclear Science, vol. 50, no. 6, pp. 2158–2166, 2003.

[WES10] N. H. E. Weste, D. M. Harris, "CMOS VLSI Design: A Circuits and Systems Perspective", Ed. Addison-Wesley, 2010.

[ZHA06] M. Zhang, S. Mitra, T. M. Mak, N. Seifert, N. J. Wang, Q. Shi, K. S. Kim, N. R. Shanbhag, S. J. Patel, "Sequential Element Design with Built-In Soft Error Resilience," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 14, No. 12, pg. 1368–1378, December 2006.

List of Figures

Figure 1.1 CMOS Technology Nodes 6
Figure 1.2 Transistor Counts of Intel Microprocessors, Data Source: [INT10] 7
Figure 1.3 Clock Frequencies of Intel Microprocessors, Data Source: [INT10] 7
Figure 1.4 Failure Rate during Digital Systems' Lifetime, Source: [GIR10] 8
Figure 1.5 Different types of manufacturing defect, Source: [ITR11] 9
Figure 1.6 Soft Error Mechanism, Source: [BAU05] 11
Figure 1.7 Example of a Tolerated Fault 14
Figure 1.8 TMR Architecture 16
Figure 1.9 Subsystem-level TMR Architecture 16
Figure 1.10 Duplication/Comparison Architecture 16
Figure 1.11 NAND Multiplexing Architecture of a XOR Unit 17
Figure 1.12 Information Fault-Tolerant Architectures for Memories and Logic Circuits 18
Figure 1.13 Timing Fault-Tolerant Architectures 20
Figure 1.14 Information-Timing Hybrid Fault-Tolerant Architecture for Error Detection 21
Figure 1.15 Information-Timing Hybrid Fault-Tolerant Architecture for Error Correction 22
Figure 1.16 Hybrid Architecture Combining Duplication/Comparison and Parity Codes 22
Figure 1.17 Logic/Memory Composition of System-on-Chip, Source: [ITR03] 24
Figure 1.18 Logic Circuit Architecture 25
Figure 1.19 Power Consumption Trends of System-on-Chip 26
Figure 1.20 Latch Hardening Using TMR, Source: [WAN03] 26
Figure 1.21 Flip-flop Hardening Using C-element, Source: [ZHA06] 26
Figure 1.22 RAZOR Flip-flop, Source: [ERN03] 27
Figure 1.23 RAZOR II Latch, Source: [DAS09] 27
Figure 2.1 Principles of Hybrid Fault-tolerance 30
Figure 2.2 Concurrent Error Detection Scheme 31
Figure 2.3 Parity Predictor Synthesis Flow 32
Figure 2.4 Duplication/Comparison Scheme for Logic Circuits 33
Figure 2.5 Static Comparator Structure 34
Figure 2.6 Error Signal 35
Figure 2.7 Dynamic CMOS Logic 35
Figure 2.8 Dynamic OR 35
Figure 2.9 Transition Detector Structure 36
Figure 2.10 Pseudo-Dynamic Comparator Structure 37
Figure 2.11 The Complete Error Detection Architecture 38
Figure 2.12 Timing Constraints for Error Detection Scheme 38
Figure 2.13 Control Module for Error Detection Architecture 39
Figure 2.14 Transient Error Correction Architecture 40
Figure 2.15 Re-computation Problem 41
Figure 2.16 Modified D Flip-Flop mDFF for Re-computation 41
Figure 2.17 Modified D Flip-Flop's Function 41
Figure 2.18 The Complete Transient Error Correction Architecture 42
Figure 2.19 Transient Error Correction Mechanism 43
Figure 2.20 CLKRegin Generator for Transient Error Correction Architecture 44

Figure 2.21 Reset and CRegin Generator for Transient Error Correction Architecture 45
Figure 2.22 Control Signal Generator for Transient Error Correction Architecture.............................. 46
Figure 2.23 Control Module for Transient Error Correction Architecture 47
Figure 2.24 The Hybrid Fault-Tolerant Architecture 48
Figure 2.25 Elementary input demultiplexer 49
Figure 2.26 Example of an Optimized Input Demultiplexer 49
Figure 2.27 Elementary output multiplexer – Method 1 50
Figure 2.28 Elementary output multiplexer – Method 2 50
Figure 2.29 Elementary output multiplexer – Tri-state buffer.. 51
Figure 2.30 Finite state machine diagrams 52
Figure 2.31 Clock and Condition Signals of the Finite State Machine.. 54
Figure 2.32 Control Logic for Input Demultiplexer... 55
Figure 3.1 Fault-tolerant Architecture Evaluation Flow 59
Figure 3.2 Combinational Logic Extraction from Sequential Circuits.. 59
Figure 3.3 TMR Structure for Logic Circuits.. 61
Figure 3.4 Word-Voter Architecture, Source: [MIT00b] 62
Figure 3.5 4-input dynamic OR gate DOR4_X1 63
Figure 3.6 Feedthrough Path Created by Combinational Part Extraction.. 64
Figure 3.7 Synthesized Elementary Input Demultiplexer... 67
Figure 3.8 Synthesized Elementary Output Multiplexer – Method 2 69
Figure 3.9 Glitches Detection Capability of DOR Gate 74
Figure 3.10 Reset of DOR Gate... 74
Figure 3.11 Detection Capability of Pseudo-Dynamic and Static Comparators................................... 75
Figure 3.12 Generated Control Signals for Pseudo-Dynamic Comparator 76
Figure 3.13 Generated Control Signals for Input Register 77
Figure 3.14 Generated Control Signals for Input Demultiplexer and Output Multiplexer 77
Figure 3.15 Hybrid Fault-Tolerant Architecture’s Behavior in Fault-Free Case 78
Figure 3.16 Hybrid Fault-Tolerant Architecture’s Behavior with Transient Error Occurrence 79
Figure 3.17 Hybrid Fault-Tolerant Architecture’s Behavior with Permanent Error Occurrence............ 79
Figure 4.1 Stand-Alone Logic Circuit versus Pipeline Architecture 86
Figure 4.2 Razor and Global Clock Gating Implementation for Pipeline Architecture [ERN03]............. 88
Figure 4.3 Razor II and Architectural Replay Implementation for Pipeline Architecture [ERN03] 90
Figure 4.4 Hybrid Fault-Tolerant Implementation for Pipeline Architecture.. 91
Figure 4.5 Register-Level SEU Protection [IMH11]... 93
Figure 4.6 Detailed SEU Detection Scheme at Register-Level [IMH11] 93
Figure 4.7 Bit-flipping latches for SEU Correction [IMH11].. 94
Figure 4.8 D flip-flop function 95

List of Tables

Table 1.1 Faults and Errors in Digital Systems 13
Table 1.2 Soft Error Rate in Embedded Memory, Source [ITR11] 24
Table 1.3 Truth Table of a C-element 27
Table 2.1 Parity Predictors' Area 32
Table 2.2 FSM1 Functioning Example 53
Table 2.3 FSM2 Functioning Example 53
Table 2.4 Outputs of the Finite State Machine 54
Table 3.1 Bit-wise vs. Word-wise Voter 61
Table 3.2 Area and Delay of Synthesized Combinational Logic 65
Table 3.3 Area and Delay of Synthesized Redundant Combinational Logics 66
Table 3.4 Area of Synthesized Input and Output Registers 67
Table 3.5 Area and Delays of Synthesized Input Demultiplexer 68
Table 3.6 Area of Synthesized Output Multiplexer 69
Table 3.7 Delays of Synthesized Output Multiplexer 70
Table 3.8 Area of Synthesized Comparator 71
Table 3.9 Area and Delay of Synthesized Finite State Machine 72
Table 3.10 Area of Synthesized Voters 72
Table 3.11 Area of Synthesized Fault-Tolerant Architectures 73
Table 3.12 Power Saving of Hybrid Fault-Tolerant Compared to TMR Architectures 80
Table 4.1 Re-configuration by FSM1 85
Table 4.2 Re-configuration by FSM2 85
Table 4.3 Operation of Pipeline Architecture 87
Table 4.4 Error Propagation in Pipeline Architecture 88
Table 4.5 Error Correction in Pipeline Architecture Using Razor and Clock Gating 89
Table 4.6 Error Correction in Pipeline Architecture Using Razor II and Architectural Replay 90
Table 4.7 Error Correction in the Hybrid Fault-Tolerant Pipeline Architecture 92

Architecture Hybride Tolérante aux Fautes pour l'Amélioration de la Robustesse des Circuits et Systèmes Intégrés Numériques

RESUME : L'évolution de la technologie CMOS consiste à la miniaturisation continue de la taille des transistors. Cela permet la réalisation de circuits et systèmes intégrés de plus en plus complexes et plus performants, tout en réduisant leur consommation énergétique, ainsi que leurs coûts de fabrication. Cependant, chaque nouveau nœud technologique CMOS doit faire face aux problèmes de fiabilité, dues aux densités de fautes et d'erreurs croissantes. Par conséquence, les techniques de tolérance aux fautes, qui utilisent des ressources redondantes pour garantir un fonctionnement correct malgré la présence des fautes, sont devenus indispensables dans la conception numérique.

Ce thèse étudie une nouvelle architecture hybride tolérante aux fautes pour améliorer la robustesse des circuits et systèmes numériques. Elle s'adresse à tous les types d'erreur dans la partie combinatoire des circuits, c'est-à-dire des erreurs permanentes (« hard errors »), des erreurs transitoires (« SETs ») et des comportements temporels fautifs (« timing errors »). L'architecture proposée combine la redondance de l'information (pour la détection d'erreur), la redondance de temps (pour la correction des erreurs transitoires) et la redondance matérielle (pour la correction des erreurs permanentes). Elle permet de réduire considérablement la consommation d'énergie, tout en ayant une surface de silicium similaire comparée aux solutions existantes. En outre, elle peut également être utilisée dans d'autres applications, telles que pour traiter des problèmes de vieillissement, pour tolérer des fautes dans les architectures pipelines, et pour être combiné avec des systèmes avancés de protection des erreurs transitoires dans la partie séquentielle des circuits logiques (« SEUs »).

Mots clefs : Robustesse, tolérance aux fautes, circuits logiques.

A Hybrid Fault-Tolerant Architecture for Robustness Improvement of Digital Integrated Circuits and Systems

ABSTRACT: Evolution of CMOS technology consists in continuous downscaling of transistor features sizes, which allows the production of smaller and cheaper integrated circuits with higher performance and lower power consumption. However, each new CMOS technology node is facing reliability problems due to increasing rate of faults and errors. Consequently, fault-tolerance techniques, which employ redundant resources to guarantee correct operations of digital circuits and systems despite the presence of faults, have become essential in digital design.

This thesis studies a novel hybrid fault-tolerant architecture for robustness improvement of digital circuits and systems. It targets all kinds of error in combinational part of logic circuits, *i.e.* hard, SETs and timing errors. Combining information redundancy for error detection, timing redundancy for transient error correction and hardware redundancy for permanent error corrections, the proposed architecture allows significant power consumption saving, while having similar silicon area compared to existing solutions. Furthermore, it can also be used in other applications, such as dealing with aging phenomenon, tolerating faults in pipeline architecture, and being combined with advanced SEUs protection scheme for sequential parts of logic circuits.

Keywords: Robustness, fault-tolerance, logic circuits.

Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier

LIRMM, UMR 5506 CC477, 161 rue Ada, 34392 Montpellier Cedex 5, France