FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Building Robust GALS Circuits

Fault-Tolerant and Variation-Aware Design Techniques for Reliable Circuit Operation

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der technischen Wissenschaften

by

Jakob Lechner
Registration Number 0226071

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger

The dissertation has been reviewed by:

(Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger)	(Prof. Jens Sparsø)

Wien, 29.04.2014

(Jakob Lechner)

Erklärung zur Verfassung der Arbeit

Jakob Lechner
Zur Spinnerin 22/11, 1100 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

I would like to thank Prof. Andreas Steininger for his excellent supervision of my PhD research, his invaluable technical and scientific insights during our discussions and his support and encouragement, which never failed to boost my motivation. He significantly influenced my academic career at an early stage with his remarkable lecture on digital circuit design. In this lecture he was able to spark my interest for the field, which ultimately led to the completion of this thesis.

Many thanks have to be given to my colleagues and friends from the Embedded Computing Systems group, Markus Ferringer, Matthias Függer, Marcus Jeitler, Alex Kößler, Robert Najvirt, Rameez Naqvi, Thomas Polzer, Thomas Reinbacher, Varadan Savulimedu Veeravalli and Martin Zeiner. I want to thank them for all the helpful and inspiring discussions, the collaboration in research projects and the late hours spent together when a paper submission deadline was approaching. The great team spirit and companionship is what makes this research group special.

Furthermore, I would like to thank Prof. Ulrich Schmid for the opportunity to perform my research as a part of the FATAL project[1]. Thank you also has to be said to Traude Sommer for her administrative support and her efforts to remind me to go home when time was getting late.

With great appreciation I also want to thank Prof. Alex Yakovlev from the Newcastle University. Due to his support I was able to join his research group as a guest researcher for half a year. This visit[2] was a fantastic opportunity to meet and work with excellent researchers and allowed me to perform the final steps towards the completion of my thesis.

A big thank you goes to all my friends who spent time with me outside of the university and helped me to forget about my research once in a while and enjoy my free time.

Finally, I want to express my deepest gratitude to my parents, my brother and my sister for their lifelong support and love! Thank you!

[1]This work received funding from the Austrian Science Foundation (FWF): FATAL project, no. P21694.

[2]The visit in Newcastle was supported with a Marietta Blau Grant of OeAD – GmbH, funded by the Austrian Federal Ministry of Science and Research (BMWF).

Abstract

Digital integrated circuits perform computations following a well-defined functional specification. Any deviation from the circuit's expected behaviour breaks the boundaries of this specification and can therefore lead to unknown and unwanted circuit states, miscomputations and ultimately a service failure. Today transient faults are responsible for most of the circuit failures. They are typically triggered by adverse external influences like radiation, electromagnetic interference or variations of supply voltage and ambient temperature during operation. Fault-tolerance is the capability of a circuit to maintain its correct operation despite of such effects.

In this thesis we investigate fault-tolerance mechanisms applied to GALS-style circuits. GALS, short for *globally asynchronous locally synchronous*, is a design paradigm, which partitions a complex circuit into several locally synchronized modules and performs data transfers between these modules by means of asynchronous handshake channels. This is a systematic approach to reduce timing dependencies in circuits and therefore ease their physical implementation. Based on the fundamental structure of GALS circuits, this thesis is split into a part dedicated to fault-tolerant computation in synchronous modules and a part, where mechanisms for reliable data transfers in asynchronous communication channels are explored.

For fault-tolerant computation we propose two new *modular redundant* circuit architectures. The key concept in both cases is to apply full replication to all circuit elements of the targeted GALS module, including the clock source and the clock tree. In contrast to conventional modular redundant circuits, replicated units can therefore be operated with independent clock signals. This simplifies circuit timing and increases flexibility with respect to the physical partitioning of redundant components. State restoration is performed at dedicated checkpoints, which are interwoven with regular computation cycles. Stoppable clocks, commonly used in GALS circuits for safe data exchange among locally synchronous modules, provide the underlying framework for executing the recovery process at these checkpoints. While the proposed architectures share the same basic idea, we engage two different methods for state exchange and majority voting during the recovery and compare their area, performance and reliability properties.

In the second part of the thesis we advocate for the combination of *delay-insensitive* and *error detecting codes* to build asynchronous communication channels between GALS modules. This approach integrates *variation tolerance* and *fault tolerance* and therefore provides a comprehensive form of robustness. First we present a systematic analysis of common delay-insensitive codes to determine their resilience against transient faults and what capabilities associated error detecting codes need to have to mitigate a certain number of such faults during a transmission. Based on these theoretical results, implementations for various encoder and decoder circuits are developed, which can offer protection against single or double faults.

Kurzfassung

Digitale integrierte Schaltungen führen Berechnungen auf Basis einer genau definierten Spezifikation durch. Jegliche Abweichung vom erwarten Verhalten einer Schaltung kann diese Spezifikation verletzten und daher zu unbekannten und unerwünschten Schaltungszuständen, Fehlberechnungen und letztendlich zum Totalausfall der Schaltung führen. In den allermeisten Fällen sind heutzutage transiente Fehler, ausgelöst durch ungünstige externe Einflüsse wie Strahlung, elektromagnetische Interferenzen oder Spannungs- und Temperaturvariationen, verantwortlich für solche Ausfälle. Kann eine Schaltung ihre korrekte Funktion trotz dieser Effekte aufrecht erhalten, so spricht man von einem fehlertoleranten Design.

In dieser Arbeit untersuchen wir die Anwendung von Fehlertoleranz-Mechanismen im Kontext von sogenannten GALS-Schaltungen. Charakteristisch für GALS, abgekürzt für *global asynchron, lokal synchron*, ist die Aufteilung von komplexen Schaltungen auf mehrere lokal synchronisierte Module und der Einsatz von asynchronen Übertragungskanälen zum Datenaustausch. Dieser systematische Designansatz begrenzt Zeitabhängigkeiten auf einzelne Module und erleichtert somit die physische Umsetzung der Gesamtschaltung.

Im ersten Teil dieser Arbeit werden zwei neue Ansätze präsentiert, um GALS-Module mittels Mehrfachredundanz gegen Fehler zu schützen. Die Kernidee ist in beiden Fällen die vollständige Replikation aller Schaltungsbestandteile des entsprechenden GALS-Moduls, was insbesondere auch den Taktgeber und das Taktnetz einschließt. Im Gegensatz zu konventionellen redundanten Schaltungen, können replizierte Einheiten daher mit einem unabhängigen Taktsignal versorgt werden. Das vereinfacht das Zeitverhalten der Schaltungen und erhöht die Flexibilität bei der Aufteilung von redundanten Komponenten. Die Detektion und Korrektur von fehlerhaften Schaltungszuständen erfolgt zu bestimmten Kontrollpunkten, die zwischen den regulären Rechenvorgängen eingeschoben werden. Anhaltbare Taktgeneratoren, die häufig in GALS-Schaltungen für den sicheren Datenaustausch zwischen lokal-synchronen Modulen Verwendung finden, bilden die Basis für die Implementierung dieses Wiederherstellungsprozesses.

Zur zuverlässigen Kommunikation zwischen GALS-Modulen wird dann im zweiten Teil der Arbeit eine Lösung vorstellt, die zeit-insensitive (engl. delay-insensitive) und fehlerdetektierende Codes verbindet. Durch diesen Verband ist es dem Empfänger möglich sowohl Variationen im Zeitbereich, als auch Fehler im Wertebereich des übertragenen Signals zu tolerieren. Zunächst analysieren wir bekannte zeit-insensitive Codes systematisch auf ihre Widerstandsfähigkeit gegen Fehler und bestimmen die benötigte Stärke der komplementären fehlerdetektierenden Codes, um einer gewissen Anzahl von Fehlern während der Datenübertragung standzuhalten. Auf Basis dieser theoretischen Ergebnisse werden dann verschiedene Implementierungen für Kodierungs- und Dekodierungsschaltungen entwickelt und deren Eigenschaften evaluiert.

Contents

CHAPTER 1

Introduction

Integrated circuits have made their way into almost every aspect of our lives and have been able to fundamentally change the way how we work, spend our free time, obtain information and communicate. They can be found in general purpose hardware like classical personal computers, servers or notebooks, as wells as in mobile devices like tablets, smart phones or wearable computers. Mobile computers in particular have been transforming how we interact with the digital world in the past few years (and will potentially continue to do so in the years to come). Beyond that there is an abundance of applications in embedded systems, usually designed to perform a very specific function. These applications range from ordinary household appliances and consumer electronics like dishwashers, MP3 players, digital cameras, etc. to professional equipment used, e.g., in telecommunication or medical systems. Large numbers of embedded devices can also be found in all modern means of transportation like cars, aircraft or trains.

Semiconductor technology, the propellant behind this digital revolution and precursor of the information age, has undergone a remarkable and fascinating evolution in the past six to seven decades. Even though the first discoveries in the field of semiconductors date back to the 19th century [128], a pivotal moment for the emergence of integrated circuits, as we know them today, is the development of the transistor. In 1947 John Bardeen and Walter Brattain demonstrated the first transistor design at Bell Labs and only one year later William Shockley proposed an improved device, known as the bipolar junction transistor [93]. The fabrication of the first MOSFET device, just about a decade later in 1960, by John Atalla and Dawon Kahng then spurred an unprecedented technological race. Driven by advances in fabrication and process technology, transistor sizes kept shrinking continuously and the number of transistors that could be built into an integrated circuit grew exponentially. Just within fifty years industry went from chip designs with a couple hundred transistors to today's processors, graphics cards and FPGAs, containing several billions of transistor devices. Technology scaling and rising integration densities, allowing for powerful chip designs at reduced costs, were and still are the backbone of the semiconductor success story.

1.1 Circuit Timing

The progress of semiconductors in the last decades, however, has not been possible without severe problems to solve, without tremendous efforts both in academic and industrial research institutions to deepen our understanding of physical processes, to develop increasingly intricate fabrication methods, and figure out ways to deal with rising design complexities. As industry today is determined to continue scaling following *Moore's Law* [74], the battle to keep pushing the boundaries is still astir and with every technology node new challenges appear.

An increasingly severe problem of chip designs fabricated at 90 nm and below is circuit timing. Timing plays a key part in the overall design process because computations are usually synchronized with the help of a global clock signal, which controls all sequential elements in the circuit. This clock, being a periodic signal, enforces a rigid time base for the switching activities of all other signals. Thus, timing of every signal in such a *synchronous* design needs to be carefully analysed to make sure that all signals comply with the timing constraints (setup/hold constraints) mandated by the clock signal. One reason why this analysis has become more and more difficult with recent technology nodes is the massive imbalance how gate and interconnect delays are affected by scaling. While gate delays generally have been on the decline with every new technology generation, the opposite is true for interconnect delays: Due to longer and thinner wires, as well as smaller wire pitches, interconnect RC delays have deteriorated significantly, as can be seen in [44, 45]. These scaling trends have shifted the earlier dominance of gate delays as the main contributor to total circuit delays towards interconnect delays. In recent technologies relative gate and interconnect delays differ by approximately three orders of magnitude [44]. This has severe impacts on the design flow of integrated circuits. While gate delays are known relatively early in the implementation process, basically right after synthesis, interconnect delays can only be modelled when the final chip layout has been performed, i.e., close to the end of the design flow. If unresolvable timing violations are uncovered at this stage, a complete redesign and reimplementation of the circuit might be unavoidable.

Another side effect of aggressive scaling are timing uncertainties, e.g., caused by process variations. To guarantee correct circuit operation these uncertainties have to be accounted for, usually by adding conservative timing margins (and thereby reducing the performance gains again, which we expect by downscaling feature sizes). Process variations originate from imperfections in chip fabrication, which can be traced back to random processes during the placement of dopants, or limited resolution of lithographic fabrication steps [121]. At nanometre scale these variations have a significant impact on crucial device parameters like oxide thickness, channel length and width, doping profiles, as well as interconnect geometries (wire width/thickness). The gate oxide thickness of transistors, e.g., can be controlled with an astounding accuracy of 1 to 2 atom layers. However, in 30 nm and below the gate oxide only consists of approx. 5 to 15 atom layers [121]. Consequently one or two layers more or less can already make a notable difference. The same is true for the placement of impurities in the channel region. With technology nodes below 90 nm the number of dopant atoms is less than hundred, somewhere in the order of a few tens of atoms [121]. Depending on the specific technology used, process variation-related timing uncertainties can account for up to 30% of the overall timing budget [108].

A final key challenge we want to address here is the design of clock trees [45]. The clock

signal is distributed to all sequential endpoints, which are usually scattered all over the entire chip area. To achieve high circuit performance the skew and jitter of the clock signal need to be rigorously minimised. However, when dealing with large clock trees and high frequencies, layout has become a rather cumbersome task. Growing interconnect delays and process variations, as discussed above, further complicate this design challenge: In large clock trees, sometimes with latencies of several clock cycles, process variations can have a detrimental effect on skew and jitter [13]. Therefore careful routing of the clock net is required, e.g., by using symmetric tree layouts like H-trees. Furthermore an optimal buffer placement and sizing has to be found to evenly distribute clock transitions and drive all loads connected to the clock tree. These clock buffers, however, consume considerable amounts of power since the clock net typically is the most active signal of the circuit. In extreme cases the clock tree's share of total power consumption has been reported to be as high as 40% [24].

Because of the above mentioned issues (and others we did not discuss), globally synchronous circuits with a single clock domain are mostly a thing of the past [109]. Nowadays circuit designers are dealing with complex synchronous systems that incorporate several clock domains, usually running at different clock frequencies. Exchanging data across clock domains requires manual insertion of synchronizer circuits to make sure that incoming data can be safely processed at the receiver side. For complex systems with many clock domain crossings this work can become rather tedious and also error-prone, with severe consequences on the circuit's reliability, if done wrong. The *globally asynchronous locally synchronous design (GALS)* paradigm offers a more robust and systematic approach to integration of separately clocked circuit modules: Communication between these *locally synchronous modules* is performed across asynchronous links, using handshake protocols to enable safe data transfers. This methodology decouples sender and receiver clock domains and eliminates the need for a global clock distribution network. Locally synchronous modules are clocked independently with individual clock frequencies adapted to their specific requirements and capabilities. With local modules that are more compact and smaller in size the design of the local clock tree is simplified [30] and wire lengths of intra-module interconnects can potentially be reduced. Timing problems thus become more manageable and constraints can be easier met. Since even the latest ITRS report mentions asynchronous global signalling as a desirable capability for future technologies [45], it can be expected that GALS architectures will gain in importance.

1.2 Dependable Circuits

Next to performance or power efficiency, dependability can also be an essential requirement for integrated circuits. In safety-critical systems, like airplanes or spacecraft, nuclear power plants, transportation systems etc. high dependability is of utmost importance since a failure can potentially threaten human lives, or cause significant environmental damages. Reliable operation also needs to be specifically addressed for systems that operate in harsh environments, like satellites in space, where increased radiation levels can cause malfunctions of onboard electronics. For other applications, where a failure does not necessarily lead to a severe catastrophe, high availability might still be required because service stops are extremely inconvenient for users or cause notable financial losses.

Since we are on the brink of exciting new developments like commercial space travel or autonomously driving cars, it can be expected that there will be a continuous, if not increasing, demand in dependable integrated circuit architectures, which will potentially have to meet more stringent requirements on computational performance and power-efficiency. However, while large complex designs with high processing speeds can be implemented in modern semiconductor technology, reliability has become a major concern. Due to aggressive technology scaling, leading to reduced feature sizes and reduced supply voltages, integrated circuits have become much more susceptible to faults [10, 94], both of transient and permanent nature. While in the former case, errors vanish after some time or can be corrected, permanent faults are caused by damages in the physical structure of the die and will not disappear. Today transient faults are responsible for most of the circuit failures [114]. They are typically triggered by adverse external influences like radiation, electromagnetic interference or variations of supply voltage and ambient temperature during operation. Radiation effects, typically referred to as single event effects, can produce unwanted voltage pulses in an integrated circuit that might lead to miscomputations and the corruption of the circuit state, which is maintained in storage elements like flip-flops or memory cells. Unaccounted voltage or temperature variations can compromise system timing in rigid synchronous circuits, again resulting in potential miscomputations and erroneous outputs.

To meet the demand of dependably operating circuit designs, fault tolerance has been an active and thriving research area in the past decades. One of the first seminal works on this subject was performed by John von Neumann and published in 1956 in a paper, which describes how reliable systems can be built from unreliable components [117].

1.3 Scope and Methodology

In this thesis we are going to present our research results on robust GALS-based circuit architectures, which tackle current and future circuit design challenges, especially related to timing, and also provide correct operation in the presence of external faults like single event effects or internal faults such as process variations. We were interested to explore how fault tolerance mechanisms can be applied to such circuit architectures and whether there are any benefits in comparison to fault tolerance in classical all-synchronous designs. As described above, GALS circuits consist of a) multiple synchronous modules, which perform computations, and b) asynchronous communication channels between these modules. Therefore, our research efforts were devoted to hardening these two integral circuit parts, i.e., the *computation* and *communication* components of a GALS system. It was a fascinating aspect of our research that this distinctive architecture of GALS systems allowed us to investigate quite different sets of fault tolerance mechanisms, which account for the fundamentally different function and structure of computation and communication systems: While we explored replication-based techniques for GALS modules (modular redundancy), building robust asynchronous communication channels mainly involved protection with error detecting and correcting codes. Even though the presented techniques can cope with permanent faults to some extent and there is a brief discussion on this in Chapter 4, we focus on the mitigation of transient faults, so called single event transients (SETs) and single event upsets (SEUs). Dealing with permanent faults requires quite different counteractive measures, which puts them out of scope for this work.

In order to evaluate the viability of the proposed fault tolerance techniques, the associated circuits were first modelled on register transfer- or gate-level and then synthesised and mapped to a suitable CMOS technology (90 nm in our case, using an industrial-grade standard cell library). Functional verification as well as performance analysis was then performed with the synthesised netlists. We are aware that the accuracy of pre-layout timing annotations is suboptimal, however, for the purpose of comparing different solutions we consider this approach to still yield meaningful results. Verification of the behaviour in the presence of faults was partly done by running exhaustive fault-injection experiments in a simulator, and partly with model checking techniques to formally prove desired liveness and safety properties for smaller subcircuits. Reliability evaluation was performed with Markov chain models to derive formulas for the mean time to failure (MTTF), based on failure and recovery rates as basic parameters.

1.4 Structure of the Thesis

In Chapter 2 we will present a basic taxonomy of dependable computer systems, including concise definitions of relevant terms like *faults*, *errors* and *failures*. Most importantly the concept of reliability is introduced, along with mathematical definitions for MTTF, or reliability and failure rate functions. Furthermore we will address dependability threats in digital integrated circuits in this chapter, first to give an overview on the specific terminology that has been coined in this research area, and secondly to get the reader acquainted with the origins of faults and failure mechanisms in integrated circuits, both transient and permanent. Chapter 3 is intended to provide the necessary circuit-related background for the main part of the thesis. Thus, we give a brief introduction to asynchronous logic and handshaking protocols, present previous work on GALS systems and existing solutions with respect to communication issues between locally synchronous modules. A third section in this chapter then is devoted to fault-tolerant circuit design and the available techniques for leveraging either hardware, time or information redundancy.

The main pillars of this thesis are Chapter 4 and 5. As mentioned above, our research work is partitioned into fault tolerance techniques for locally synchronous modules and asynchronous interconnects. This thesis is structured accordingly. First we will present two approaches for incorporating modular redundancy into GALS circuits. While the system-level fault tolerance concept for both approaches is the same, they use different methods for state recovery, which lead to interesting differences in circuit structure and associated properties. In Chapter 5 we turn to asynchronous interconnects, where we strive for solutions that provide a comprehensive form of robustness, which combines both variation and fault tolerance. Our approach therefore advocates the fusion of delay-insensitive and error detecting/correcting codes. First a theoretical analysis is performed to elaborate what codes can be paired to achieve tolerance against a certain number of faults. Afterwards we present two link architectures along with specific circuits for sender and receiver components, ready to be integrated into I/O wrappers of GALS modules. An overview of the related work will be given at the end of each main chapter, where we elaborate on commonalities and differences in comparison to the methods we propose.

In Chapter 6 we summarise the presented research results and draw our conclusions. The thesis then ends with a short final chapter on possible directions for future work.

CHAPTER 2

Dependable Computer Systems

According to [8] dependability is the ability of a (computer) system to *"avoid service failures that are more frequent and more severe than is acceptable"*. Basically dependability is a measure to what extent a user can trust that a system will deliver a service or a set of services that complies with the specification, or ultimately with the user requirements. In this chapter we will introduce elementary concepts of dependable systems in general and how they relate to digital circuits in particular. We want to present a concise taxonomy of the concepts and the terminology that will be used throughout this thesis.

2.1 Taxonomy of Dependable Systems

The availability of a concise taxonomy for concepts related to dependability along with a well-defined terminology is of vital importance for researchers and engineers working in this field. It allows the development of a clear understanding of the problems and challenges that have to be faced, and most notably provides a common view and vocabulary that enables experts to discuss general concepts and specific techniques, both new and old. Since dependability is a very broad area and a plethora of researchers and engineers have contributed to it in the last decades, the used terminology to describe certain concepts or phenomena is often slightly different or even inconsistent from author to author[1]. Thus, a first effort to develop a unified taxonomy was undertaken in 1980, when a joint committee on "Fundamental Concepts and Terminology" was formed by the *Technical Committee on Fault-Tolerant Computing* of the IEEE Computer Society and an IFIP work group on *Dependable Computing and Fault Tolerance* [8]. Since then many discussions have led to the continuous refinement of dependability concepts and also an integration of security aspects. The most recent publication, giving a comprehensive and elaborated description of the taxonomy dependable and secure computing, is [8]. An outline of this taxonomy is depicted in Figure 2.1. As can be seen, all aspects of dependability and security are partitioned into three fundamental categories: *Attributes*, *Threats* and *Means*.

[1] A wide-spread imprecision, e.g., is the synonymous use of the terms "transient fault"and "soft error".

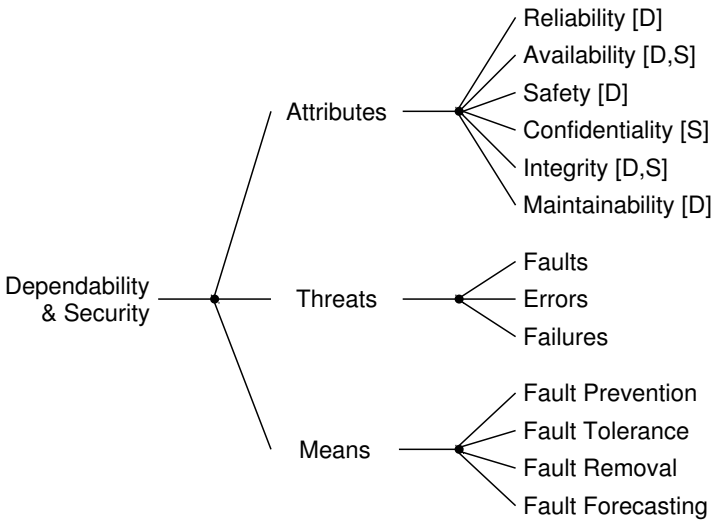


Figure 2.1: Taxonomy of dependability and security concepts.

2.1.1 Threats of Dependability

Threats to the dependability of a system can be explained by three core concepts: *faults*, *errors* and *failures* [8]. Each of these concepts describes a specific class of phenomena, events or system states that occur during the transition of a system from a state of correct operation to a state where it no longer follows its intended function.

Failures

The service of a system can be expressed as a sequence of externally-visible state changes, as perceived by the users of the system. A service or system failure, short failure, therefore denotes the transition from a state where the system delivers a correct service, i.e., operates within its specified function, to a state of incorrect service [8]. The definitions of reliability, availability and safety we will present below all relate to the concept of failure. Failures can be described with a set of *failure modes*, which are characterised from different viewpoints, like the *failure domain*, *consistency*, or the *severity* of a failure's consequences. The failure domain is partitioned into *content failures* (with correct timing) and *timing failures* (with correct content). Obviously, content and timing failures can also occur simultaneously. In the best case a system simply stops its service (*halt failure*), in the worst case the behaviour becomes erratic [8] and wrong data is produced at wrong points in time (babbling idiot [111]). The concept of consistency refers to how a failure is perceived by the system's users. If all user see the same failing behaviour, this is called a *consistent* failure, otherwise the failure is classified as *inconsistent*. The latter failure type is quite difficult to detect and mitigate, since it can trick correct components into an inconsistent behaviour. In literature inconsistent failures often are referred to as *two-faced*, *byzantine*, or *malicious* failures [55].

Errors

An *error* is an incorrect system state, which in a digital computer system is typically manifested by wrong data stored in some kind of memory element, like a flip-flop or an SRAM cell. An error may lead to a failure, when it influences the system's service and its effects become externally visible. Thus, a system failure can also be understood as an error in some external service interface. If the presence of an error is indicated by some form of signal, an error is said to be *detected*. Undetected errors are called *latent errors* [8]. Note that an error does not necessarily need to cause a failure, as it could be overwritten before the erroneous data is processed.

Faults

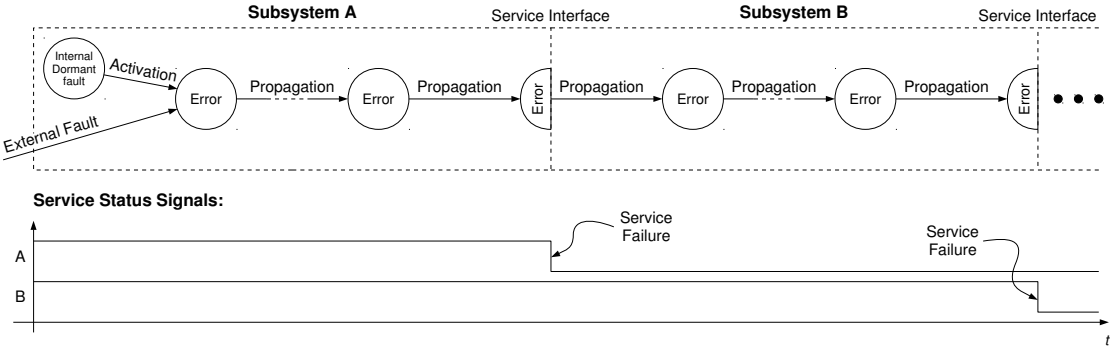
In [78] a *fault* is described as "anomalous physical condition" that affects the system behaviour in some way and can ultimately lead to an error. If a fault causes an error, it is said to be *active*, otherwise the fault is *dormant*. In [8] faults are classified in elementary fault classes, such as:

- Phase of creation: A fault might occur during the *development phase* or during the *use phase*. The development phase includes all activities until a system can be deployed and is ready for use, e.g., initial concept, specification, implementation, verification, fabrication, installation, etc. The use phase begins when system starts its operations and provides its designated service to the respective end user(s).
- System boundaries: An important classification is based on the origin of a fault, which can be either inside or outside the system boundaries. In the former case we refer to *internal faults*, in the latter case to *external faults*. Internal faults might be design flaws, broken hardware components due to manufacturing defects or wearout, etc. External causes among others are incorrect inputs, or harsh environmental conditions such as temperature variations, radiation, or electromagnetic interference [78].
- Persistence: Faults can either be of *transient* or *permanent* nature. Transient faults are bounded in time and vanish without any explicit repair action [55]. Ionising radiation, e.g., is a typical source for transient faults. Permanent faults, like physical defects in a circuit, do not disappear on their own. They persist until the condition has been repaired.
- Phenomenological cause: *Natural faults* vs. *human-made faults*.
- Objective: *Malicious* vs. *non-malicious faults*. Malicious faults are caused by a human with the purpose to harm the system (the term "malicious" in this context should not be confused with "malicious (byzantine) failures").

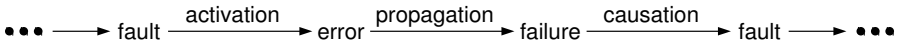
The Relationship between Faults, Errors and Failures

Figure 2.2a illustrates how a fault can lead to an error and consequently to a system failure. The fault can either originate from an internal source or can be caused by some event outside the system boundaries (external fault). The process when a fault turns into an error is called *fault activation*. Once some part of a system is erroneous, the error might be *propagated* to other parts

as a result of the system's computation steps. As soon as an error reaches a service interface, a failure occurs as the (sub)system deviates from its specified behaviour. This failure can then be understood as a fault for another subsystem that uses the services provided by the failed system. Figure 2.2b summarises the relationships between faults, errors and failures.



(a) Error propagation.



(b) Fault chain.

Figure 2.2: Faults, Errors and Failures.

2.1.2 Attributes of Dependability

Attributes refer to properties a dependable and/or secure computer system might need to have to some extent, depending on the specific application and the users' requirements. The primary attributes identified in [8] are: *Reliability*, *Availability*, *Safety*, *Confidentiality*, *Integrity* and *Maintainability*. In Figure 2.1 we have tagged these attributes with capital D and S to highlight their relationship to either dependability or security. As can be seen some attributes are relevant for dependability, some for security, and some relate to both fields. Please note that this classification does *not* say that maintainability, e.g., is only important for dependable, and not at all for secure systems. The given allocation of attributes is only supposed to give an idea where the primary concerns of the respective systems lie. Note that security, even though it is a matter of vital importance, is currently not on our research agenda. Therefore, we will only discuss the mentioned attributes with respect to their significance for dependable systems.

Reliability

The ability of an item to perform a required function, under given environmental and operational conditions and for a stated period of time (ISO8402).

High reliability is crucial for systems where a single deviation from the expected behaviour would lead to catastrophic consequences, e.g., in airplanes or power plants. The attribute is associated with a reliability function $R(t)$, which defines the probability that a system provides

the correct service until time t [92]. $R(t)$ can be defined over the *time to failure* T, i.e., the time it takes from the start of operation until the system fails for the *first* time. T is usually interpreted as a continuous random variable with a probability density function $f(t)$, called *failure density*. The associated distribution function $F(t)$ thus gives the probability that a failure occurs in the interval $(0, t]$ and is defined as

$$F(t) = P(T \leq t) = \int_0^t f(u)\,\mathrm{d}u \qquad \text{for } t > 0. \tag{2.1}$$

Since the reliability function $R(t)$ denotes the probability that *no* failure occurs in interval $(0, t]$ it can therefore simply be derived from $F(t)$ by

$$R(t) = P(T > t) = 1 - F(t) = \int_t^\infty f(u)\,\mathrm{d}u \qquad \text{for } t > 0. \tag{2.2}$$

While the failure density defines the probability that a system will fail within a certain time, very often the probability that a system will fail in some interval $(t, t + \Delta t]$, given that it was fully operational until t, is of interest. This conditional probability can be expressed as

$$P(t < T \leq t + \Delta t \,|\, T > t) = \frac{P(t < T \leq t + \Delta t)}{P(T > t)} = \frac{F(t + \Delta t) - F(t)}{R(t)} \tag{2.3}$$

If this probability is divided by the length of the investigated time window Δt, we receive a probability per unit time. Note that this is no longer a probability but a *rate*, called *failure rate* [53]. Consider, e.g., the conditional failure probability to be 0.4 within a time of 2 days. Then the resulting failure rate would be 0.2 failures per day, or 1.4 failures per week, depending on the unit. A commonly used unit in reliability engineering is *Failures in Time* (FIT), which represents the failure rate in 10^9 hours, i.e., 1 FIT equals a rate of 1 failure in approx. 115000 years. We can now take the limit $\Delta t \to 0$ of Equation 2.3 divided by Δt, which gives us the *failure rate function* $z(t)$, sometimes also referenced as *(instantaneous) hazard rate* [116]:

$$z(t) = \lim_{\Delta t \to 0} \frac{P(t < T \leq t + \Delta t \,|\, T > t)}{\Delta t} = \lim_{\Delta t \to 0} \frac{F(t + \Delta t) - F(t)}{\Delta t} \frac{1}{R(t)} = \frac{f(t)}{R(t)} \tag{2.4}$$

The failure rate function can assume various shapes. For modelling the failure rate over the lifetime of a system the bathtub curve is usually used (see Figure 2.3). Very often it is assumed that the failure rate function has a constant value $z(t) = \lambda$. This means that the probability of failure does not depend on how long ago the system has been put into operation, or in other words, there is no history that influences whether the system is more or less likely to fail at a particular point in time. A constant failure rate is useful for analysing system reliability in case faults are caused by external phenomena, like radiation or electromagnetic interferences, under certain (constant) worst-case conditions (see Section 4.6.3).

The time to failure T can be distributed in many different ways, depending on the specific system and the environment in which it is operating. Examples for possible distributions are: Exponential distribution, Gamma distribution, Normal distribution, Weibull distribution, etc. Most commonly an exponential distribution is chosen in reliability analysis, because it is mathematically very simple and gives a realistic failure model for most systems [92]. It is the only

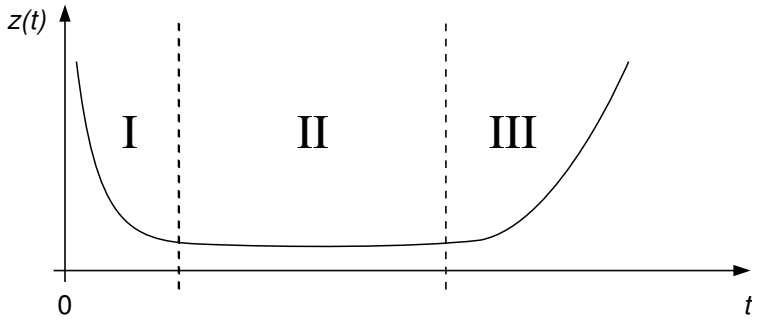


Figure 2.3: Bathtub curve modelling failure rate over a system's lifetime across three typical intervals: I) Burn-in period, II) useful life, and III) wear-out period.

distribution with a constant failure rate and is therefore suitable to model the "useful life" part of a components life cycle (cf. Figure 2.3) [116]. In case of an exponential distribution the failure density $f(t)$ with the parameter λ can be defined as

$$f(t) = \begin{cases} \lambda e^{-\lambda t} & \text{for } \lambda > 0, t > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.5}$$

Consequently, the reliability function $R(t)$ is

$$R(t) = P(T > t) = \int_t^\infty f(u)\,\mathrm{d}u = e^{-\lambda t} \qquad \text{for } t > 0, \tag{2.6}$$

which allows us to derive the failure rate function $z(t)$:

$$z(t) = \frac{f(t)}{R(t)} = \frac{\lambda e^{-\lambda t}}{e^{-\lambda t}} = \lambda. \tag{2.7}$$

A key measure to describe a system's reliability is the *Mean Time to Failure* (MTTF), which is defined as the expected time to the first failure after the start of system operation. Formally the MTTF can therefore be defined as the expected value of the time to failure T:

$$MTTF = E(T) = \int_0^\infty t f(t)\,\mathrm{d}t. \tag{2.8}$$

Since $f(t)$ can be written as

$$f(t) = \frac{\mathrm{d}F(t)}{\mathrm{d}t} = \frac{\mathrm{d}(1 - R(t))}{\mathrm{d}t} = -R'(t), \tag{2.9}$$

it is also possible to express the MTTF in terms of the reliability function:

$$MTTF = -\int_0^\infty tR'(t)\,\mathrm{d}t = -[tR(t)]_0^\infty + \int_0^\infty R(t)\,\mathrm{d}t \tag{2.10}$$

As the reliability function equals 0 when $t \to \infty$, the term $[tR(t)]_0^\infty$ evaluates to 0, and the MTTF can therefore simply be stated as

$$MTTF = \int_0^\infty R(t)\,\mathrm{d}t. \tag{2.11}$$

In case of an exponential failure distribution the MTTF therefore is

$$MTTF = \int_0^\infty e^{-\lambda t}\,\mathrm{d}t = \frac{1}{\lambda}. \tag{2.12}$$

Maintainability

The ability of an item, under stated conditions of use, to be retained in, or restored to, a state in which it can perform its required functions, when maintenance is performed under stated conditions and using prescribed procedures and resources (BS4778).

Maintenance operations can either be *repairs* or *modifications* [8]. The former deals with the removal of faults (*corrective maintenance*) or with the detection and removal of dormant faults (*preventive maintenance*), the latter is concerned with adjustments due to environmental changes or augmentations of the system's function (*adaptive* and *augmentive* maintenance). Note that repair operations are similar to the recovery of erroneous states in fault tolerance mechanisms. However, we will follow the definition in [8] that maintenance involves the activity of an *external agent*, e.g., a repair man who replaces failed units.

Availability

The ability of an item (under combined aspects of its reliability, maintainability and maintenance support) to perform its required function at a stated instant of time or over a stated period of time (BS4778).

Formally the availability $A(t)$ of a system can be expressed as the probability that the system is operational *at* time t. Note that if a system is permanently broken after a failure and cannot be repaired $A(t) = R(t)$, i.e., the probability that the system is operational in the interval $(0, t]$ [25]. Availability as dependability measure, however, only makes sense for systems where service interruptions are tolerable and a repair is possible. To quantify a system's dependability very often the *average availability* is used. This can be expressed as the fraction of time the system is available to provide correct service [55], i.e., as the ratio of *uptimes* to total time (*uptimes* + *downtimes*). The downtimes are typically modelled as the *Mean time to Repair* (MTTR), or alternatively as *repair rate* μ [25]:

$$\mu = \frac{1}{MTTR}. \tag{2.13}$$

The repair rate depends on several factors, like the mean time it takes after a failure until repair is started, i.e., how fast the failure is detected and the responsiveness of maintenance staff. Other issues might be how fast spare parts are available in case of defects, and also the time it takes to fix the system, which is influenced by the maintainability. If failure and repair rates are constant, the average availability can simply be computed as follows [55]:

$$A_{average} = \frac{MTTF}{MTTF + MTTR}. \tag{2.14}$$

Safety

The *safety* attribute is an extension of reliability, where failures are partitioned into two categories: *fail-safe* and *fail-unsafe* [25]. In so-called *safety-critical* systems fail-unsafe failures can have dramatic consequences, like human injuries, loss of life, or environmental disasters. The safety function $S(t)$ thus denotes the probability that a system is fully operational, or in case of failures behaves in a fail-safe manner during the time interval $(0, t]$. Safety-critical systems usually have to be certified by an independent certification agency [55].

Integrity

This attribute is concerned with the avoidance of "improper" system alterations [8]. Improper, in this context, typically means unauthorised.

2.1.3 Means to Attain Dependability

Fault Tolerance

The aim of fault tolerance is to build systems that are able to avoid service failures in the presence of faults [8]. An essential requirement for designing such systems is the precise formulation of a *fault hypothesis* [55]. This hypothesis specifies what kind of faults the system is supposed to mitigate. Faults are therefore either *covered* or *uncovered* by the fault hypothesis. If properly designed and implemented, the system should be able to detect and recover errors that are the result of the activation of covered faults. For uncovered faults, on the other hand, the system behaviour is unspecified and can potentially lead to service failures. Obviously in a reasonable fault hypothesis the latter kind of faults should be highly improbable so that they do not pose a threat for dependable system operation.

To be able to mitigate the effects of faults a fault-tolerant system thus needs to be equipped with mechanisms to detect internal errors and recover the system state before these errors can propagate to externally visible service interfaces. The key concepts involved in fault-tolerant system design are *error detection* and *recovery*. Figure 2.4 shows a classification of the most commonly used fault tolerance techniques. Error detection can either be done concurrently to the system operation or in a preemptive fashion, where the system stops to allow for detection of latent errors or dormant faults. *Preemptive error detection* is often performed at system start-up, but also during operation where, e.g., spare components are checked or memories are analysed as part of scrubbing routines [8].

Once an error is detected a recovery mechanism has to be executed to prevent the propagation of errors. Furthermore, it is essential to bring a system back to an error-free state before another fault is activated, as fault tolerance mechanisms can usually only cope with a limited number of concurrent faults (as specified in the fault hypothesis). Recovery mechanisms can be split into two groups: *error handling* and *fault handling* techniques. Classical error handling techniques

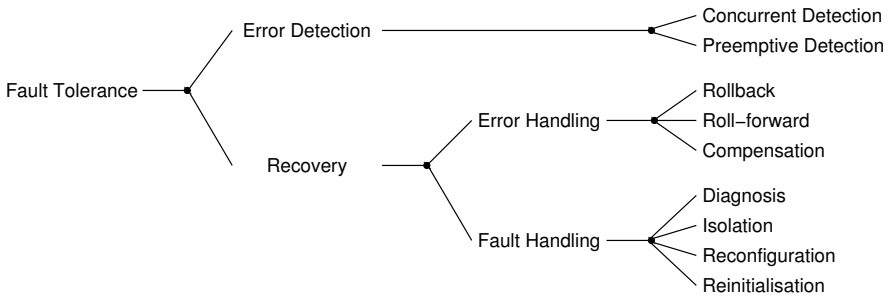


Figure 2.4: Classification of fault tolerance techniques.

include *rollback* and *roll-forward* strategies. The fundamental concept for rollback mechanisms is *checkpointing*. A checkpoint simply is a complete snapshot of the system state at a specific point in time, which is stored in a safe memory for later use [56]. If an error is detected, the system state can be rolled back to the last checkpoint and computations are repeated from there on. Although the basic principle is simple, many difficult questions are entailed, like when and how often a checkpoint should be captured, or how can a checkpoint be generated in a distributed environment with many communicating processes. Furthermore, this approach leads to a performance degradation due to the execution time overhead when taking a snapshot and the recomputation when a rollback is necessary. For applications where such overheads are not acceptable roll-forward techniques can be employed. In these approaches error correction is directly applied on the current state when errors are detected [42]. Clearly some form of redundancy is required to be able to do this. The third error handling technique depicted in Figure 2.4, *compensation*, follows the idea of *masking* errors, again by using redundancy to identify and filter erroneous computation results. Rollback and roll-forward are typically applied *on demand*, whereas compensation can either be performed on demand or *systematically*. In the latter case masking actions are performed unconditionally at predefined points in time without prior detection of an error (cf. voting on output data).

When internal faults cause errors, fault handling can be performed following error detection to prevent that these faults will be activated again [8]. As can be seen in Figure 2.4 the steps involved in fault handling are *diagnosis*, *isolation*, *reconfiguration*, and *reinitialisation*. Diagnosis aims at identifying the location and type of a fault and isolation then tries to separate a faulty component from the system, so the fault becomes dormant. Reconfiguration replaces faulty units with spare components or redistributes tasks among active non-faulty components. Reinitialisation finally updates the state of the new system configuration.

Redundancy plays a key role in the design of fault-tolerant systems. Redundancy can either be *static*, where fault tolerance is structurally built into a system and directly masks fault effects [104], or *dynamic*. For dynamic redundancy active actions are required, like error detection and subsequent error and fault handling activities, as described above. Consequently, static and dynamic redundancy are also referred to as *passive* and *active* redundancy.

Depending on what resource is used to create redundancy, we can distinguish the following four forms: *hardware*, *time*, *information*, and *software redundancy*. Many advanced fault toler-

ance mechanisms often use a bundle of approaches, like static techniques along with dynamic ones, or a mix of hardware and time redundancy. This is called *hybrid redundancy* [104].

In hardware redundancy a computation is executed on multiple hardware components to be able to detect and counteract errors [56]. Typically this is achieved by replication of hardware units, which can be done on different levels, from transistor and gate to system level. The type and number of errors that can be mitigated hereby depends on how many physical copies are available. A well-known example for hardware redundancy are *modular redundant* systems.

Alternatively to executing a computation on multiple hardware components, in time redundancy re-executions are done on a single component or processing node. Thus potentially high hardware overheads can be avoided. The downside, however, can be a substantial performance degradation. Also time redundancy is only effective against transient faults, where it can be assumed that a fault vanishes before re-execution is performed. Time redundancy can be applied systematically, i.e., when every computation is replicated and results of successive execution runs are compared, or, when other error detection mechanisms are available, it can also be implemented on demand (e.g., checkpoint and rollback recovery strategies) [56].

Information redundancy typically uses error detecting and error correcting codes to protect computations or data transfers against faults. These codes extend processed or transmitted data words with additional bits, which are some form of checksum of the encoded data bits. With these redundant bits and with the help of specific decoding algorithms it can be determined whether a data word contains errors or not. If the applied code is strong enough, i.e., if there is sufficient redundant information available, the locations of erroneous bits can be inferred, thus allowing for recovery of the correct data word. Note that information redundancy requires both additional hardware, for encoders and decoders, as well as extra computation time.

In many complex computer systems nowadays an integral part of the computation is performed in software. For high-reliability applications software redundancy is sometimes used as a means to mitigate software faults (bugs). This is achieved with the implementation of multiple functionally equivalent versions of the program code (design diversity), ideally by independent teams of programmers. The hope driving this approach is that it is unlikely for differently implemented program versions to contain the same type of bugs, i.e., fail for the same input [56]. The redundant program versions can be executed in parallel on different processors (hardware redundancy), or in sequence on the same processor (time redundancy).

Fault Prevention

Fault prevention includes all possible means to prevent that faults can occur in the first place [25]. During the development phase, e.g., this can be achieved by using rigorous specification of system requirements [91], structured design methods and careful verification with, e.g., code reviews or formal methods. During the use phase the probability of faults can be reduced by using robust hardware components that are not affected by adverse environmental conditions (electromagnetic interference, radiation, etc.), shielding of sensitive system parts, frequent maintenance operations, etc.

Fault Removal

Removal of faults to enhance dependability can be performed either during the development or during the use phase. In the former case faults have to be identified using verification, diagnosed and finally corrected. During the use phase fault removal is mainly conducted through maintenance, either preventive or corrective.

Fault Forecasting

The term fault forecasting describes various means to evaluate the behaviour of a given system in the presence of faults [8]. This can either be done with a *qualitative* or a *quantitative* evaluation, or sometimes a combination thereof. Qualitative evaluations aim to analyse conditions that could lead to a system failure, identify potential failure modes and classify and rank them with respect to their effect, frequency and severity. A general procedure that can be employed for this kind of evaluation is called *failure mode and effect analysis* [107].

Quantitative evaluation techniques on the other hand use probabilistic models of the system to derive numerical values to what extent a system satisfies the dependability attributes presented above. In this context these attributes are also called *measures of dependability* [8]. A widespread technique in reliability modelling, e.g., is the use of *Markov chains*. An aspect often involved in quantitative evaluation is *testing*, which delivers data on failure and maintenance processes, etc. This data can then be filled into system models to provide probabilistic estimates for dependability measures, based on realistic system or environmental parameters.

2.2 Faults in Integrated Circuits

In this section we will move our discussion from the general concepts in dependable systems to the specific dependability issues we face in digital integrated circuits. A widely-used classification of circuit faults is the distinction of *transient* and *permanent* faults [25][2]. We will follow this classification and briefly address some of the currently predominant causes for both types of faults in this section. Our focus, however, will be on transient faults since circuit faults with permanent effects are only a side issue in this thesis. Furthermore, we want to stress that the fault tolerance mechanisms presented in the Chapters 4 and 5 of this thesis were designed for mitigation of faults that occur during the *use phase* of a circuit. This excludes all sorts of development faults, like specification and implementation faults or defects that are introduced during circuit fabrication. While these are critical issues in circuit design, they are usually tackled with different countermeasures, which are out of scope for our work.

2.2.1 Transient Faults

As described earlier, the duration of a transient fault is assumed to be bounded in time and is usually very short, e.g., below or around a nanosecond. There are various causes for transient faults

[2] Sometimes *intermittent faults* are listed as a third class. However, with respect to the taxonomy presented above, they can also be described as dormant permanent faults that are activated from time to time, or as repeatedly occurring transient faults.

in integrated circuits, which are normally originating from outside the system boundaries (external faults). Examples are radiation effects, electrostatic discharge, electromagnetic interference, drops in supply voltage or temperature variations [25]. In particular, sensitivity of semiconductors to radiation has increased dramatically due to technology scaling over the last decades [10]. Thus, radiation effects have become a key threat of deep submicron circuits, even when they operate at sea level where radiation flux is much smaller than in space or at high altitudes.

Terminology

Since a lot of research has been done in the past decades on radiation effects in semiconductors, a whole body of technical terms has been coined in this area that have been widely adopted by most researchers and engineers working in fault-tolerant circuit design. The *JESD89A standard* [47], which defines procedures for testing and characterising the resilience of integrated circuits in (terrestrial) radiation environments, provides definitions of the most commonly used terms. Generally, radiation effects are referred to as *single event effects* (SEEs), which are defined as a "measurable or observable change in the state or performance" of an integrated circuit, caused by a single energetic particle strike [47]. Hence, all terms we will introduce below describe some kind of single event effect.

Errors that are caused by radiation-induced faults are typically called *soft errors*, often also referred to as *single event upsets* (SEU) [10][3]. The term "soft" reflects on the fact that the circuit is not physically impaired, but a value stored in a memory cell, a latch or flip-flop, e.g., was changed and thus the circuit state is compromised. Obviously, this condition can be reversed or recovered by rewriting the correct value into the respective element. Soft errors can be classified by the way the circuit state is modified: Typically only a single storage cell is affected, which leads to a *single bit upset* (SBU). If more than one bit of information is altered, this is called *multiple-cell upset* (MCU), and in the case that the affected bits belong to the same data word this is described as *multiple-bit upset* (MBU) in the JESD89A standard. An important metric for the reliability assessment of an integrated circuit is the *soft error rate* (SER), which simply describes the rate at which soft errors occur. Clearly, the SER depends on a variety of parameters, like the radiation flux, the circuit topology, supply voltages, transistor sizes, doping profiles, etc. [50]. Characterising the soft error rates for a given circuit is a complex task, which is usually based on various circuit models and simulations along with an experimental validation.

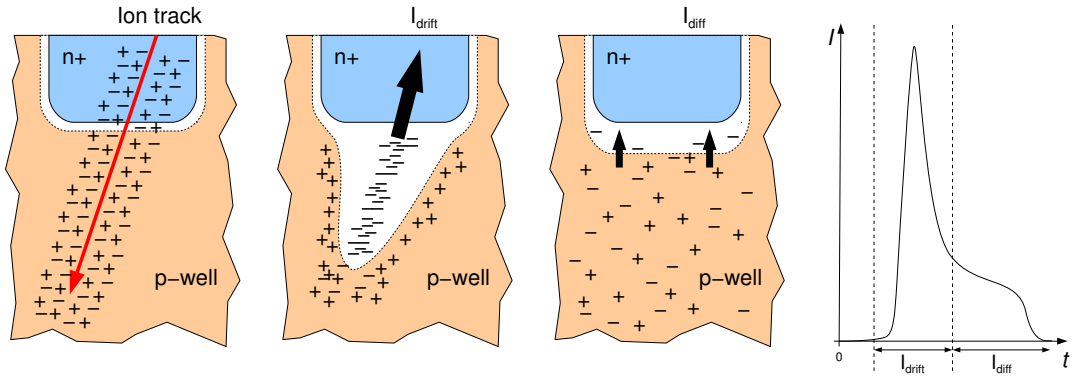
A radiation effect that causes a voltage pulse at a node of a circuit is called *single event transient* (SET). In a sequential element this SET can directly result in a soft error. On the other hand, if the node is part of a combinational circuit, the SET might propagate through logic gates until it is eventually latched by a sequential circuit element, which then ends in a soft error. In the latter case, however, most SETs are never captured since they are filtered by *logical, temporal or electrical masking* effects as they progress on combinational paths [50].

[3]The term soft error first appeared in 1978 [47] in a publication by May and Woods from Intel [69] on alpha particle-induced faults in DRAMs. The name single event upset was then introduced one year later by Guenzer, Wolicki and Allas [39] in a paper that also dealt with radiation faults in DRAMs, but caused by neutrons and protons.

Effects of Radiation on Semiconductor Devices

Radiation-induced faults occur when ionised particles travel through the semiconductor material and lose energy along their track, which results in the deposition of charge in the form of electron hole pairs. In a terrestrial environment there are two major sources for ionised particles: I) alpha particles, which originate from radioactive isotopes in the die or the packaging, and II) secondary particles that are generated by collisions of neutrons with silicon, oxygen or dopant atoms in the die [80]. When the deposited charge is collected at p-n junctions of MOS transistors, a current/voltage pulse is created at that particular node [10]. Whether this pulse actually results in a harmful SET or even in a soft error, depends on the magnitude of the collected charge Q_{coll} and the node's sensitivity, which is influenced by various parameters like node capacitance, operating voltage or the strength of the feedback inverters in the storage loop of sequential elements. This sensitivity can be expressed in terms of a *critical charge* Q_{crit}, which needs to be exceeded by a radiation event to be harmful for the computation of the circuits, i.e., $Q_{coll} > Q_{crit}$.

An SET pulse unfolds in three phases after an ionised particle hits the die, as can be seen in Figure 2.5. In the first phase charge is generated around the track of the particle. This is followed by a rapid charge collection process due to the electric field in the p-n junction, which creates a steep increase of the current at that node. During this process the depletion region forms a funnel-shaped extension into the substrate, which boosts the charge collection. This phase lasts for less than a nanosecond [10], after which the funnel collapses. The collection of the remaining charge can then be attributed to diffusion of electrons into the depletion region, which happens at a much slower rate. The final phase of the SET can therefore last hundreds of nanoseconds [10]. Figure 2.5d shows the junction current over these three phases, which is often approximated with a double-exponential function [119].



(a) Phase I. (b) Phase II. (c) Phase III. (d) Current pulse.

Figure 2.5: SET generation after a single event effect.

2.2.2 Permanent Faults

Physical defects in integrated circuits can be caused by a variety of effects. Most commonly circuit defects are introduced during the fabrication process due to mask misalignment, over- and under-etching, or spot defects, e.g., caused by contamination with dust particles [56]. Permanent faults, however, can also occur during the use phase of a circuit, triggered by electrical, mechanical or thermal stress, or wearout/ageing mechanisms like *electromigration* (EM), *Hot Carrier Injection* (HCI), *Negative Bias Temperature Instability* (NBTI) or gate oxide breakdown [33]. Possible effects of permanent faults are open or short circuits, threshold voltage shifts, increased leakage currents, variability of carrier mobility, etc. [104].

Various fault models have been developed that provide an abstraction from the specific physical fault manifestations. This helps to analyse potential circuit defects and implement procedures for fabrication tests or for built-in self tests, which can be executed on a regular basis during circuit operation. The most popular model is the *single stuck-at fault model* (SSAF), where a single circuit signal is assumed to be permanently tied to a value of 0 or 1, while all other circuit signals are non-faulty. Even though this model is very simple it can detect a high percentage of manufacturing defects. In Chapter 4 we will perform fault-injection experiments based on this model. Other important fault models are: *Multiple stuck-at faults* (MSAF), *bridging faults*, *open* faults, *path-delay* faults and I_{DDQ} faults [36].

CHAPTER 3

Circuits Background

3.1 Asynchronous Circuit Design

A fundamental principle in electrical engineering and computer science is the use of abstractions, which help us to focus on the relevant properties of a system when describing or solving a particular engineering problem. As a matter of fact, the design of many complex systems only becomes tractable due to the use of such powerful abstractions [2]. Most digital circuits, e.g., are based on two fundamental abstractions: a) All signal values or stored bits of information are assumed to be binary, with the respective physical representations denoted as 0 or 1 (*low* or *high*), and b) there is a clock offering a global discrete time base and all circuit state changes occur instantaneously at the ticks of this clock. These two abstractions have simplified the lives of many circuit designers in the previous decades and are the main characteristics of so-called *synchronous circuits*. In this class of circuits a square wave signal with a fixed frequency is used as clock, where rising or falling signal transitions denote the clock tick. All circuit operations like computations, communication among different components or sequencing of events are performed with the temporal granularity of these clock ticks.

Asynchronous circuits, on the other hand, do not depend on the notion of a global, discrete time [102]. Instead they use other mechanisms based on explicit control signalling and handshake protocols, which implement a closed-loop form of control in contrast to the open-loop control fashion of synchronous circuits [108]. Therefore circuit activities are not rigidly tied to predetermined points in time but are triggered by local events, like the availability of data for a particular component or an incoming request from the circuit's environment. Because of this adaptability and in the light of timing trends in recent technologies, asynchronous circuits are a promising alternative to traditional synchronous circuit designs. Potential benefits include [40, 102]:

- *Lower dynamic power consumption* – As asynchronous circuit are event-based and only perform operations when an actual request is active, switching activities can be significantly reduced. Synchronous circuits achieve a somewhat similar behaviour by clock

gating mechanisms. In asynchronous circuits, due to the ubiquitous presence of handshakes, deactivation of unused circuit parts is performed implicitly and potentially at a much finer granularity.

- *Higher operating speeds* – In synchronous circuits clock periods are fixed and have to be adjusted to the worst-case timing and the most critical paths. However, if worst-case propagation of signals, e.g., in arithmetic circuits, only occurs infrequently, a lot of performance is squandered in the average case [21]. Due to their adaptability and localised control, asynchronous circuits, especially delay-insensitive ones, have the potential to deliver an improved average-case performance.

- *Reduction of electromagnetic noise* – Since switching activities in asynchronous circuits are not tightly concentrated on the regular ticks of a global clock signal, but happen rather randomly distributed across time, it has been demonstrated that emissions of electromagnetic noise can be reduced [35, 83].

- *Robustness against timing variations* – Asynchronous circuits measure the delay of combinational paths [108], or use coding techniques to determine the completion of computations. Therefore they can easily adapt to process variations, or to voltage and temperature variations during operation.

- *Better composability & modularity* – Due to the use of handshake-based interfaces asynchronous components can be integrated more easily, irrespective of global timing or synchronization issues. Even if different protocols are used, converters allow a smooth interconnection of asynchronous modules.

3.1.1 Delay Models and Classification of Asynchronous Circuits

Although asynchronous circuits do not operate with the abstraction of a discrete time base, this does not mean that no assumptions on circuit timing are made at all. In literature three basic delay models are defined [21]: a) *Fixed*, b) *bounded*, and c) *unbounded* delays. In the latter case delays are allowed to assume any finite value. When a bounded delay model is used, delays have to be within certain intervals, whereas fixed delays assume a specific constant value. The used timing model has a significant impact on the design of asynchronous circuits and depending on the assumptions for gate and wire delays the following classification can be established [21]:

- For *delay-insensitive (DI) circuits* unbounded delays are assumed for both gates and wires. DI circuits therefore have to be designed in a way that allows them to operate correctly for arbitrary finite gate and wire delays. Unfortunately it has been shown that this class of circuits is rather limited and no reasonable computations can be performed [68]. Nevertheless, the attribute delay-insensitive is often used for circuits that introduce complex components to perform useful computations. While local timing assumptions within such components are required, the circuit remains delay-insensitive on component level [21].

- A less restricted class of circuits is called *quasi-delay-insensitive (QDI)*. Like DI circuits gate and wire delays are arbitrary, but additionally identical delays for certain critical wire

forks are assumed. Forks that must adhere to this assumption are called *isochronic*. Other, more realistic, formulations of this assumption require that the skew of different branches of these forks has a bounded or negligible value. The reasoning behind the introduction of isochronic forks is an important principle of asynchronous circuits, called *indication*. For further information the interested reader is directed to [102].

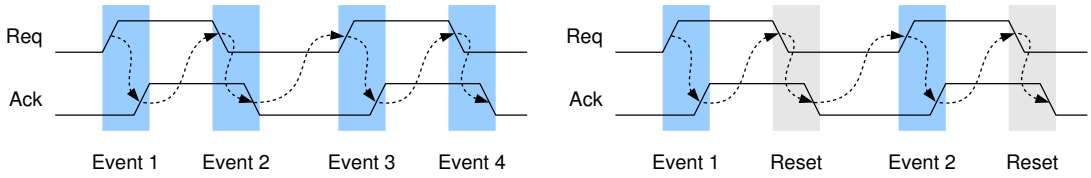
- In case of *speed-independent (SI)* circuits gate delays can be arbitrary, but wire delays are assumed to be ideal, with zero delay. Clearly this is no longer a realistic scenario in today's semiconductor technologies, where the overall circuit delay is actually dominated by wires. However, it is possible to lump wire delays into gate delays when the isochronic fork assumption is applied. There is a well-established theory on the synthesis of speed-independent control circuits, e.g., used in the synthesis tool *Petrify* [19].

- Another popular choice for implementing asynchronous circuits is the use of the bounded delay model, consequently called *bounded delay* or *matched delay* circuits. They employ timing assumptions for gates and wires, typically in the form of relative constraints, e.g., between control signals and associated data path components. This approach is similar to synchronous circuits, however, all timing assumptions only have local effects on the involved sub-circuits and do not impact the timing of a global control signal, like a clock.

3.1.2 Asynchronous Handshake Protocols

Signalling in asynchronous circuits is typically performed with the help of a request and an associated acknowledge signal. The former initiates some action or event, whereas the latter indicates the completion of that action. Request and acknowledge signals form the backbone of asynchronous communication interfaces. When data is transferred, the communicating components are called *sender* and *receiver*. Depending on who initiates the data transfer, the sender or the receiver, we either speak of a *push* or a *pull channel*, respectively [102].

There is a large number of alternatives how asynchronous handshake protocols can be implemented. Most commonly transitions of request and acknowledge signals alternate in a strict sequence and, depending on the interpretation of rising and falling transitions, we distinguish between *2-phase* and *4-phase* protocols. In a 2-phase protocol, also known as *Non-Return-to-Zero (NRZ)*, or *transition signalling* protocol, every transition of the request signal initiates a new action. The subsequent transition of the acknowledge signal then expresses the completion of that action and grants the permission to start a new request. In a 4-phase protocol, on the other hand, every request, typically signalled with a rising transition, is terminated by a reset phase, which returns both interface partners back to the initial state. The term 4-phase relates to the four transitions performed during a single handshake cycle. Due to the reset phase, this type of protocol is also called *Return-to-zero (RTZ)*, or *level signalling* protocol. Figure 3.1 illustrates both forms of handshake protocols. Note that 4-phase protocols in comparison with 2-phase protocols require twice the number of request and acknowledge transitions per initiated action/handshake cycle. Therefore 2-phase protocols potentially yield a better performance and power efficiency. In practical implementations, however, 2-phase circuits turned out to be more complex and do not automatically lead to more efficient solutions [21].

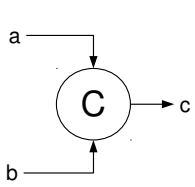


(a) 2-phase (NRZ). (b) 4-phase (RZ).

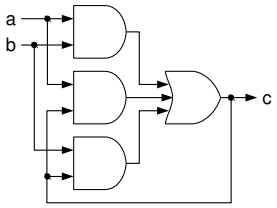
Figure 3.1: Asynchronous signalling protocols.

3.1.3 Control Elements

Two fundamental components of asynchronous control circuits are *Muller C-elements* and *mutual exclusion elements* (*MUTEX* or *ME* elements). A basic C-element has two inputs and one output, and can be described as an AND-gate for transitions. If and only if both inputs observe a rising transition, the output also generates a rising transition. The same principle applies for falling signal transitions. In case of inconsistent input values, the C-element maintains its current output value. C-elements are extensively used in asynchronous circuits for synchronizing two concurrent handshake signals, e.g., when two request signals need to be joined into a single one. Since the output in some states does not only depend on the input values but also on the current output, C-elements are sequential circuits, which need to be able to store data values, similar to RS-latches [102]. Figure 3.2 shows the circuit symbol, the truth table and a standard gate implementation of a two-input C-element. Specialised transistor-level implementations have been developed, e.g., by Sutherland [105], Martin [66] and Van Berkel [113].



a	b	c^+
0	0	0
0	1	c
1	0	c
1	1	1



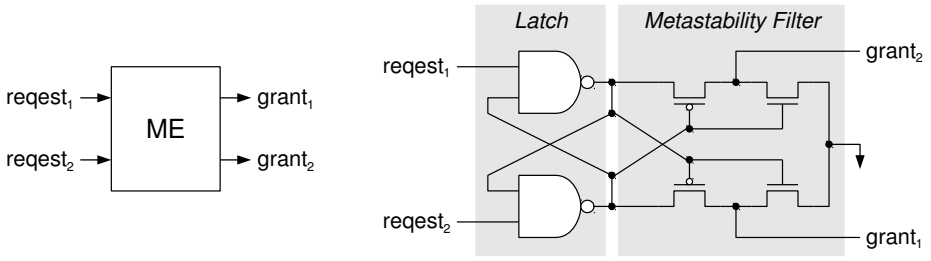
(a) Symbol. (b) Truth table[1]. (c) Standard gate implementation.

Figure 3.2: Muller C-element.

Sometimes sequencing of concurrent asynchronous handshake signals is required, e.g., if several producers want to deliver data to a single consumer, which can only process one request at a time. In this case arbitration is needed to decide in what order requests are handled. This can be done with mutual exclusion elements. As Figure 3.3a shows, a 2-way mutex element has two request inputs and two associated grant outputs. When a request arrives on one of the inputs, the corresponding grant signal is raised. As long as this request remains active, another request arriving at the second input will not be granted. This behaviour is trivial as long as two incoming

[1]The notation c^+ refers to the *next* output value, based on the inputs a and b, and the current output value c.

requests are well separated in time. If both requests, however, arrive almost simultaneously, the mutex element can be driven into a so-called *metastable state*, where it can arbitrarily decide to grant one or the other request. This decision can theoretically take an infinite amount of time. In practice this metastable state resolves eventually in favour of one or the other request. This analogue issue is well-known in synchronous circuits, where data and clock signals need to be sufficiently separated in time to avoid metastability in flip-flops. We will briefly discuss this phenomenon in Section 3.2.1. Figure 3.3b shows the implementation of a mutex element, which basically consists of a latch, made of cross-coupled NAND-gates, and a metastability filter. This filter keeps the the grant outputs low in case the latch enters a metastable state and only produces rising output transitions after metastability has been resolved internally.

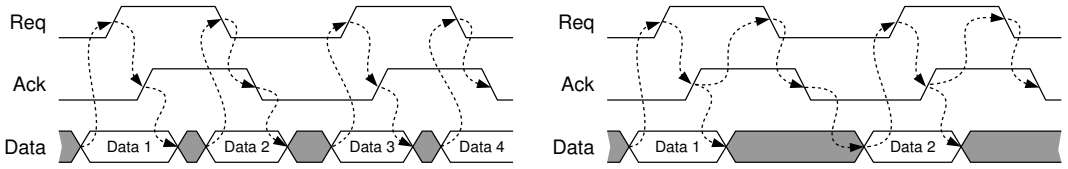


(a) Symbol. (b) Implementation (cf. [102]).

Figure 3.3: Mutual exclusion element.

3.1.4 Datapath Implementation

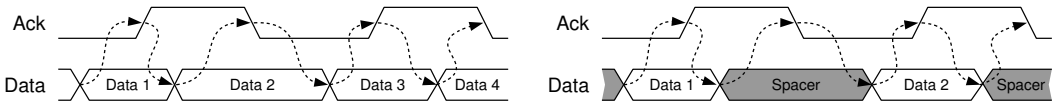
In asynchronous circuits data transfers from one module to another, or from one pipeline stage to another, are typically controlled by handshake protocols. Depending on the specific circuit class, bounded delay or some form of delay-insensitivity, the request signal indicating data validity can either be implemented as a separate wire or can be encoded along with the data signals. In the first case, the delay of the separate request wire needs to be matched with the delays of the bundled data signals. This implementation style therefore is called *matched delay* or *bundled data* and obviously requires knowledge about upper bounds of the data signal delays. Figure 3.4 shows the waveforms of two bundled data channels, using either a 2-phase or a 4-phase protocol.



(a) 2-phase. (b) 4-phase.

Figure 3.4: Bundled data channels.

In case of delay-insensitive circuits, special codes, which support completion detection, are used for encoding data words. The receiver then is able to detect an incoming data request by the completeness of the transmitted codeword. Therefore a separate request signal can be omitted and no timing assumptions for the delays of data signals have to be made. Figure 3.5 shows delay-insensitive channels, again for 2-phase and 4-phase protocols. The data signal is a parallel bus, whose bit width depends on the length of the used codewords. Note that in case of 4-phase protocols a special codeword is transmitted during the reset phase, which typically resets all data signals to zero. This codeword is called *empty*, or *spacer* codeword, since it just serves the completion of the handshake cycle and does not encode any useful data.



(a) 2-phase. (b) 4-phase.

Figure 3.5: Delay-insensitive data channels.

Since we cannot make any timing assumptions on the delay of individual wires of the data bus, the transitions, when changing from one codeword to another, or between codewords and spacers, may appear at any time in any order at the receiver. Completion detection can only work, if no intermediate bit pattern occurring during a transmission can be confused with another codeword. Only when the last bit changes, the received bit pattern forms a valid codeword again. DI codes, which satisfy this property, exist for 4-phase and 2-phase protocols.

4-Phase Delay-Insensitive Codes

The most common 4-phase DI code is a *dual-rail* code, where every data bit is encoded with a 1-of-2 code and transmitted over two wires, a.k.a. rails. One rail is called *false* rail, the other one *true* rail. Formally, $c = (c^t_{n-1}, c^f_{n-1}, ..., c^t_0, c^f_0)$ is a dual-rail encoding of $x = (x_{n-1}, ..., x_0)$, when $c^t_i = x_i$ and $c^f_i = \bar{x_i}$. Consider that the spacer codeword initially resets all rails to zero. Consequently during the data transfer, a single *rising* transition will appear on the data bus for *every* dual-rail pair, either on the false or on the true rail. Thus, for an n-bit wide data word, the receiver will see exactly n rising transitions at its inputs. Completion detection therefore is just a matter of waiting for the correct number of input transitions. Dual-rail codes have been successfully used in a variety of different techniques for building asynchronous datapath circuits, like *Null Convention Logic* (NCL) [34, 54], *self-timed boolean functions* [20], *Delay-Insensitive Minterm Synthesis* (DIMS) [103], or approaches that are based on dynamic logic [67, 100, 120].

The concept of dual-rail codes can be generalised to multi-rail codes, which can encode more than a single bit per codeword. Popular examples are *m-of-n* codes like 1-of-4, 3-of-6, or 2-of-7. Consider, e.g., a 1-of-4 code: $C_{14} = \{0001, 0010, 0100, 1000\}$. With four codewords obviously two data bits can be encoded. Like for the dual-rail code, the overhead wrt. redundant bits is again 100%. However, in case of the 1-of-4 code we only need a single rising transition per transmission instead of two. Thus a 1-of-4 code is more power-efficient. Depending on the specific requirements of the circuit, a designer needs to choose the best code. For good

interconnect area efficiency (fewer wires), e.g., a 3-of-6 code might be beneficial, since four bits can be transferred with just six wires, which is a rather low overhead for a DI code. However, when power is the main optimisation target a 2-of-7 code could be a better trade-off, since it allows the transmission of four data bits with just two rails that have to make a transition.

Dual-rail and general m-of-n codes are constant-weight codes, i.e., all codewords have the same *Hamming weight*. Delay-insensitivity in this case can be easily implemented since the receiver knows the exact number of transitions after which the incoming codeword is complete. However, there are other codes, which are not constant-weight, and still can be used for DI data transfers. One example are Berger codes, which are systematic error correcting codes, i.e., the codewords can be partitioned into a field containing the unmodified transmitted data word, and another field that consists of additional check bits. In the parlance of delay-insensitive codes these check bits sometimes are also called *synchronization* part [115]. Let x denote the vector of data bits and s the synchronization bits. For Berger codes the value assigned to s equals the binary representation of the number of zeros in the encoded data word x. Formally, s is the r-bit binary representation of $w(\bar{x})$, where r is the length of the synchronization field and w denotes the *Hamming weight* of a binary vector. The length r, consequently, needs to be adjusted to the length of the encoded data words, denoted as k, so the maximum number of zeros can be represented: $r = \lceil log_2(k+1) \rceil$. Due to this logarithmic dependence Berger codes are very efficient and the length of codewords scales well with larger k. An example of a Berger code for $k = 3$ with a 2-bit synchronization field is:

$$C = \{000|11, 001|10, 010|10, 011|01, 100|10, 101|01, 110|01, 111|00\}$$

The fact that Berger codes fulfil the prerequisites for delay-insensitivity is not as intuitive as for m-of-n codes. The interested reader is therefore directed to a proof presented in [115].

2-Phase Delay-Insensitive Codes

The *level-encoded dual-rail* scheme (LEDR) [22] is the 2-phase counterpart of the dual-rail code described above. LEDR codewords again encode a single bit of data with two rails, however the encoding scheme is a bit more intricate. In a 2-phase protocol there is no spacer codeword, so one valid codeword follows right after the other. To still support completion detection the sender strictly alternates between two different phases, called *even* and *odd* phases, for successive handshake cycles. In these phases two different sets of encodings for 0 and 1 are used, as can be seen in Table 3.1. It can be easily checked that for a change from any value of any phase to another value in the other phase only one rail transition is needed. Thus, like before in the 4-phase dual-rail code a single transition will occur for every bit of the data word and completion detection can be easily performed by the receiver. Note that encoding and decoding of LEDR values is rather simple since the first digit directly represents the encoded bit, and the second digit, also called parity, is generated by a simple XOR operation of the encoded bit with a constant *zero* in case of an even phase, or a constant *one* in the odd phase.

Again a generalisation for multi-rail 2-phase codes exists, called *level-encoded transition signalling* (LETS). Since we do not use these codes throughout the rest of this thesis, we will

Table 3.1: LEDR encoding scheme.

	0	1
even	00	11
odd	01	10

not discuss them further at this point. The interested reader, however, is directed to [14, 70], where such codes are analysed and generic procedures how to generate them are presented.

3.2 Globally Asynchronous Locally Synchronous Circuits

As we already outlined in the introduction of this thesis, large globally synchronous systems are mostly a thing of the past. On one hand scaling trends and timing uncertainties cause more and more problems when implementing large-scale synchronous circuits, and on the other hand today's complex Systems-on-Chip (SoCs) integrate many different functional modules, which have different timing requirements and therefore need to be clocked with specific frequencies. One of the main challenges when designing GALS systems, or more generally systems that are fragmented into multiple clock domains, is how to exchange data from one domain to another in a reliable and efficient way. The *globally asynchronous locally synchronous* (GALS) design methodology promotes the use of asynchronous data channels and protocols, as presented in the previous section, to solve this challenge. In other words, GALS systems use *locally synchronous* modules for computation, and asynchronous circuits for global communication. This design methodology was first proposed by Chapiro in 1984 [15], and has ever since been the focus of active research. Figure 3.6 shows a high-level illustration of a GALS system, consisting of different modules and various clock sources. As can be seen, a frequently used approach in GALS is to surround locally synchronous modules with an asynchronous wrapper, which contains the necessary circuitry to perform asynchronous communication and takes care of synchronization problems. Potential benefits of GALS include [109]:

- *Simplified timing closure* – Partitioning a circuit into locally synchronous modules allows the designer to tackle timing issues independently for each module. Smaller module sizes also lead to less complex clock trees, shorter intra-modular interconnect wires, easier power distribution and potentially reduce the impact of process variations.

- *Easier system integration & reuse of modules* – Since locally synchronous modules are encapsulated by wrapper circuits and due to the great composability of asynchronous communication interfaces, system integration is simplified. These properties also reduce efforts to reuse existing modules in new designs.

- *Power advantages* – In conventional synchronous designs the entire circuit is clocked with the *maximum* frequency, needed for the subcomponents with the highest throughput requirements. In GALS systems individual modules can be clocked with the *lowest* possible clock frequency to attain the necessary performance of each module. Lowering the frequencies of some circuit parts directly translates into power savings. GALS designs

are also well-suited for dynamic voltage and frequency scaling (DVFS) to implement advanced power reduction strategies [11, 122].

- *Less electromagnetic noise* – In traditional synchronous systems the concentration of all switching activities at clock edges and the associated rise and fall of supply currents on the circuit's power rails, can cause large undesirable electromagnetic emissions. In GALS circuits locally synchronous modules can be operated with unrelated clocks and signal switching is thereby spread over time, which reduces supply currents. Furthermore, it has been shown that methods like clock phase and frequency modulation, or power-balanced partitioning of GALS modules lead to significant noise reductions [31, 32].
- *Standard EDA tools applicable* – Since GALS modules are ordinary synchronous circuits standard EDA tools can be used. This is an advantage over entirely asynchronous circuits, which require non-standard design methods and tools. To efficiently implement GALS circuits, however, standard tools will still have to adopt their design flows and support asynchronous wrapper implementation and GALS modules partitioning.

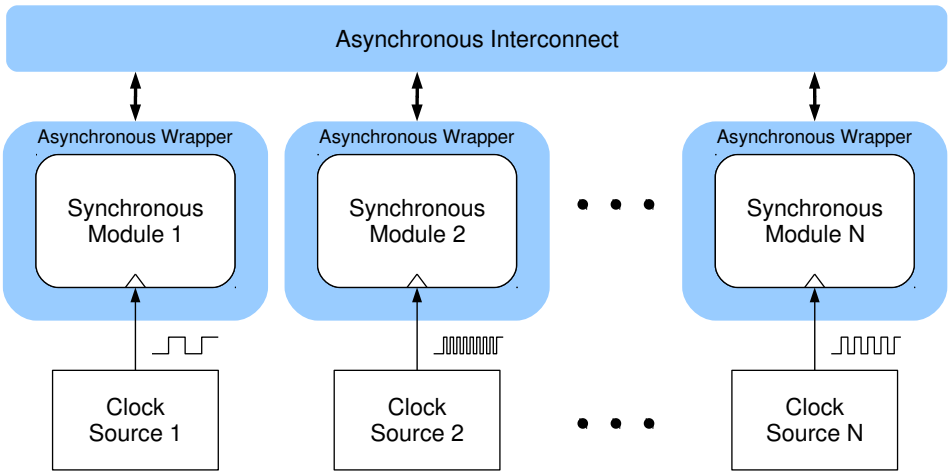
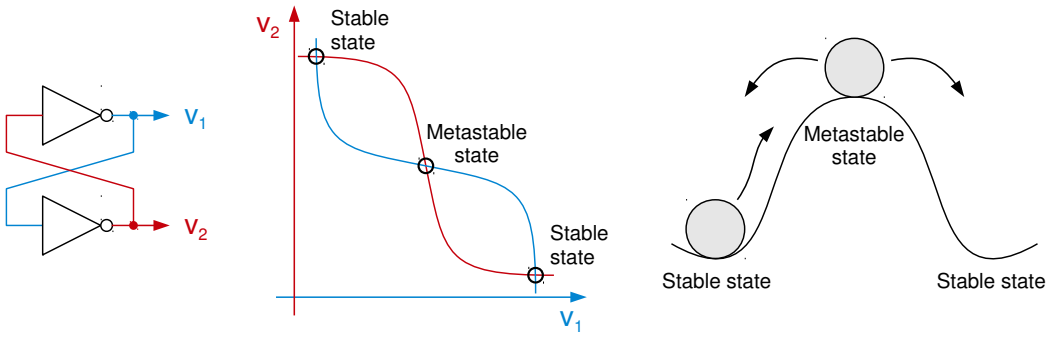


Figure 3.6: High-level view of a GALS system.

3.2.1 Synchronization in Digital Systems

As mentioned above, safe communication between different clock domains is a critical design issue. In synchronous circuits data is stored in flip-flops, which can only operate in a reliable way, if certain timing requirements are met [52]. The data input of a flip-flop needs to be stable during a certain time *before*, and for a certain time *after* the clock edge (*setup/hold* time). A violation of this requirement due to an asynchronous input, originating from a foreign clock domain, can cause a perturbation during the transition of the flip-flop's internal storage loop from the old to the new value. In a regular state the nodes of the storage loop, typically built from two cross-coupled inverters (see Figure 3.7a), assume stable digital voltage values, either high or

low. During the transition from one stable state to another, however, both inverters also assume intermediate non-digital outputs. This behaviour can be seen in Figure 3.7b, which shows the overlapped input-output characteristics of the two cross-coupled inverters. In the top-left and bottom-right corner, where the two curves cross, the flip-flop assumes a stable state, and either stores a *zero* or a *one*. There is, however, a third intersection in the middle. This is a *metastable point*, where the inverters can both assume non-digital voltages in a fragile equilibrium, which can last for an indefinite amount of time. This transition between the two stable states is often illustrated with the analogy of pushing a ball from one side of a hill to the other. If only a marginal amount of energy is supplied to perform the push, the ball might just make it to the top of the hill instead of going over it. This is called a metastable state, because in practical systems the slightest disturbance might either nudge the ball back to where it came from or finally to the other hill side. An illustration of this behaviour can be seen in Figure 3.7c.



(a) Storage loop. (b) Inverter characteristics. (c) Metastability analogy.

Figure 3.7: Metastability in digital circuits.

Metastability is a fundamental issue in any bi-stable element, like flip-flops or the mutual exclusion element, we mentioned above. Even though it only occurs in rare events, it poses a severe threat to the correct function of a circuit: Either a wrong data value might be captured, or a late output transition is generated after the metastable state has resolved, which in turn violates the timing requirements of other elements in the synchronous circuit.

3.2.2 Brute-Force Synchronization

To solve the problem of metastability external signals need to be synchronized to the clock domain of the receiver circuit. A simple synchronizer circuit can be built from one or more flip-flops connected in a chain. This delays the input signal by additional clock cycles, which allow metastable upsets to resolve before the signal is used by the receiver. This does not completely eliminate metastability but significantly reduces its probability after every flip-flop stage.

A basic solution, using two-flop synchronizers, to build a reliable unidirectional communication channel between two GALS modules is shown in Figure 3.8. As can be seen, this solution simply implements a bundled data channel using an asynchronous handshake mechanism for flow control between sender and receiver (cf. [52]). Since request and acknowledge signals

cross between the clock domains of the two communicating GALS modules, they have to be synchronized to the clock domain, where they are processed. Assuming that the data signals are stable, when the request signal has been synchronized to the receiver clock domain, they can be safely latched by the receiver without the need of additional synchronizer circuits.

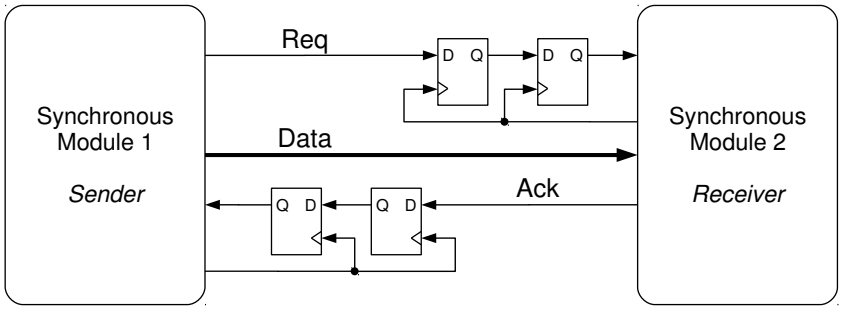


Figure 3.8: Brute-Force synchronization of asynchronous handshake signals.

The drawback of this solution is clearly an increased latency and a lowered throughput for data transfers between the GALS modules. FIFO buffers can be used to improve throughput, since they decouple the write and read processes of the sender and receiver [52]. Also, if clocks are related in some way, specialised synchronizers can be employed: In case of GALS modules with clocks that have the exact same frequency but a constant phase offset (*mesochronous* timing relationship [72]), a synchronizer only needs to perform a phase compensation and then communication can occur with full throughput [37, 109]. Another situation, where efficient synchronization is possible, are rationally related clock frequencies [96].

3.2.3 Pausible Clocking

Instead of minimising the probability of metastability it can be completely avoided by the use of pausible clocking mechanisms. The basic idea, which was already proposed by Chaprio [15] for GALS systems, is to stop the sender and receiver clocks when asynchronous inputs have to be processed. This elegantly bypasses the synchronization problem, since asynchronous input and output controllers are used to handle data transfers, and synchronous operation is only resumed once the input signals have stabilised or were safely stored in some input latch. The next active clock edge is basically delayed to some later, safe point in time. Therefore some authors also describe this mechanism as *clock stretching* or *stoppable clocks*. A high-level schematic of two GALS modules using pausible clocking can be seen in Figure 3.9. The asynchronous wrappers surrounding the locally synchronous modules contain input and output ports, which execute the asynchronous handshake protocol during data transmission. Furthermore each module is equipped with a local clock generator, which can be enabled and disabled by the respective I/O ports. Note that wrappers can of course contain multiple I/O ports, depending on the number communication channels to other modules. Numerous GALS architectures have been developed based on the stoppable or pausible clocking scheme. While they all follow the same fundamental idea, there are many variations with different behaviour and properties. An important character-

istic of all these approaches is the specific strategy that is used in I/O ports to stop and start local clocks during or after asynchronous data transfers.

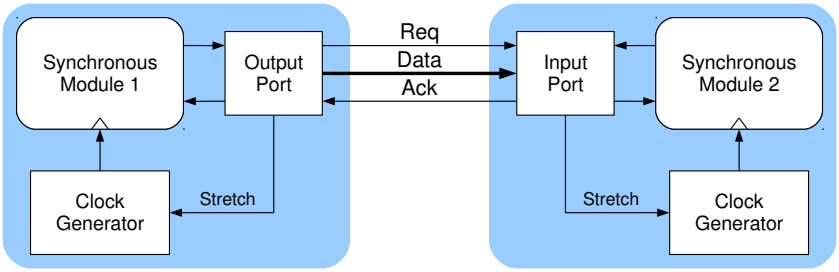


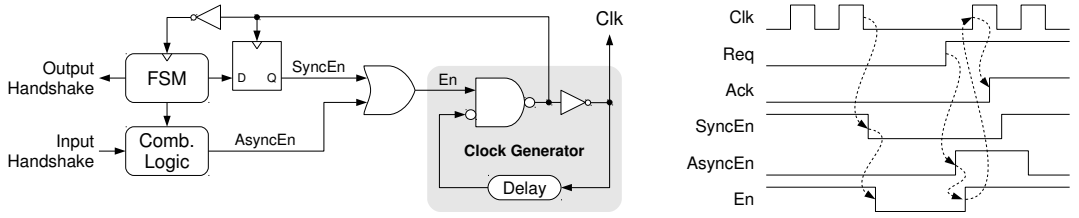
Figure 3.9: Concept of pausible clocking.

In [77] Muttersbach distinguishes between *Demand-type (D-type)* or *Poll-type (P-type)* I/O ports for the implementation styles described above. However, we prefer to use the terms *blocking* and *non-blocking I/O*, since they offer a more direct description of the locally synchronous modules' behaviour during an ongoing data transmission. In case an output port is implemented in a blocking manner, the clock and thereby the operation of the synchronous sender module is immediately stopped at the beginning of the data transmission, and only restarted by the asynchronous wrapper when the handshake procedure has completed. In other words, the clock is stopped synchronously, when the sender module initiates an output transmission, whereas the restart of the clock is performed asynchronously. In case of non-blocking output behaviour, the clock is *not* stopped and the sender module continues its computations during the asynchronous data transfer. The clock generator is only briefly paused when the asynchronous acknowledge input needs to be processed. As soon as the returning acknowledge signal has been safely latched, the synchronous sender module is permitted to issue the next output.

Similarly, solutions for input ports can be distinguished by their behaviour how the clock generator is stopped. Again, designers sometimes use blocking solutions, where the synchronous module decides at which clock cycle new inputs are required for further computations, and instructs the asynchronous wrapper to stop the clock. The wrapper then takes over control and restarts the clock generator when the wanted input data has been safely stored. The other option for implementing input ports is to only pause the clock when a new asynchronous input request arrives. The synchronous module therefore does not stop its operation to wait for new input data at predetermined clock cycles, but new inputs can be delivered at arbitrary points in time.

The first systems with stoppable clocks used blocking behaviour [15, 98]. Figure 3.10a shows a circuit template for the implementation of blocking input and output ports, which was presented in [41]. The heart of the circuit is a stoppable clock generator, which is built from an on-chip ring oscillator. The ring is formed by an inverter to enable oscillations and a delay element for adjusting the frequency of the generated clock signal. The NAND-gate, which is inserted into the ring, can be used to block clock transitions when the *enable* input is deasserted. A synchronous state machine (FSM) is responsible for switching the circuit operation into transmission mode, when data has to be sent or new inputs are required (depending on whether the circuit is used an output or input port). In this case the FSM deasserts the *SyncEn* signal, which

in turn blocks the next rising clock edge. In case of an output port, a request is initiated over the *Output Handshake* signal, and the combinational logic block waits for a corresponding acknowledgement from the receiver module, which is connected to the *Input Handshake*. As soon as the acknowledgement has arrived, the *AsyncEn* signal is raised. This reactivates the clock generator and the next rising clock edge is released. An input port works in a similar way, only with swapped roles for input and output handshake signal. After the clock has been stopped the input port waits for a new request on the input handshake signal. Upon the arrival of this request input data can be captured[2], and the clock generator can be safely re-enabled. The synchronous FSM can acknowledge the reception of the data upon the next clock transition. Figure 3.10b illustrates the clock stretching mechanisms for an input port.



(a) Circuit of I/O port (cf. [41]).

(b) Clock stretching during transmission.

Figure 3.10: Blocking I/O port template.

Yu and Donoehue [125] extended these earlier approaches and introduced a non-blocking GALS architecture. Since the synchronous sender and receiver modules are only stopped when asynchronous request or acknowledge signals need to be processed, they have added mutex elements to the clock generator to arbitrate between the next synchronous computing step and the external input events. Many researchers have developed GALS architectures that are based on this concept [29, 31, 71, 76]. An interesting implementation by Moore et al. [75] is depicted in Figure 3.11. As can be seen, the clock generator is still implemented with a ring oscillator. After the inverter the ring forks into two parallel branches. The lower one includes the delay element for adjusting the clock period, whereas in the upper branch the mutual exclusion element is inserted. The two branches reconverge at a Muller C-element, which only outputs a new clock transition, when both branches consistently deliver a rising or a falling edge. The mutex elements, which arbitrate between incoming asynchronous handshake signals (upper mutex input) and the next rising clock transition (lower input), can therefore pause the clock generation process. Note that the circuit in Figure 3.11 implements a 2-phase bundled data channel between the sender and receiver module. Thus an XOR-gate is used to generate rising transitions for the upper mutex input, in case there is a change on the request or acknowledge signals. If arbitration over the clock generator is won, the asynchronous input event can be safely processed. Metastability can, of course, occur in the mutex elements, if the next rising transition of the clock and the asynchronous input event are only marginally separated in time. This, however, is uncritical since both grant outputs of the mutex remain deasserted while metastability is resolved. During this time, no matter how long, the clock generator is paused and the processing of the

[2]Please note that the data path signals have been omitted from the schematic in Figure 3.10a.

asynchronous request or acknowledge signals is delayed. In other words, the module remains idle until the mutex decides on the next action – either another synchronous computing step or handling the asynchronous input event. Clearly this introduces an element of non-determinism to the system, which can be problematic for fault-tolerant systems with replicated components that are expected to behave consistently. We will further discuss this issue in Chapter 4.

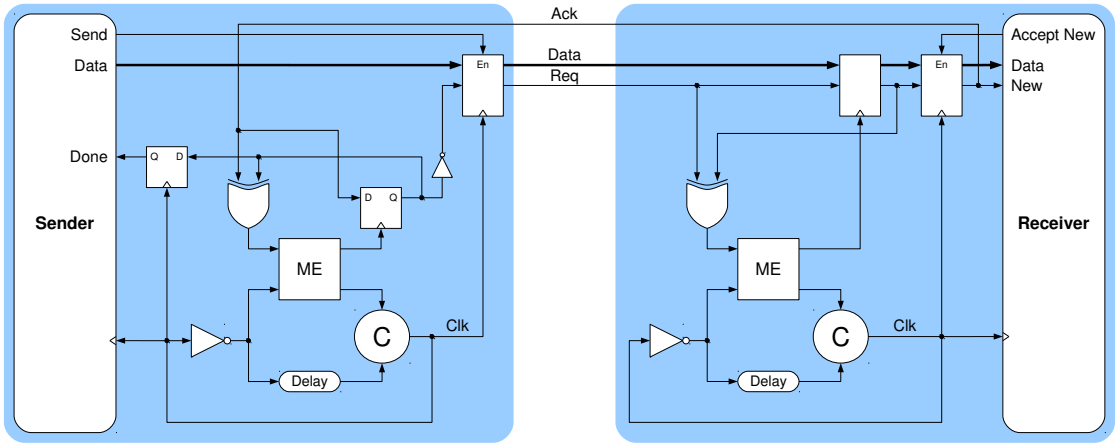


Figure 3.11: Two GALS modules connected over non-blocking I/O ports (cf. [75]).

3.3 Fault Tolerance in Integrated Circuits

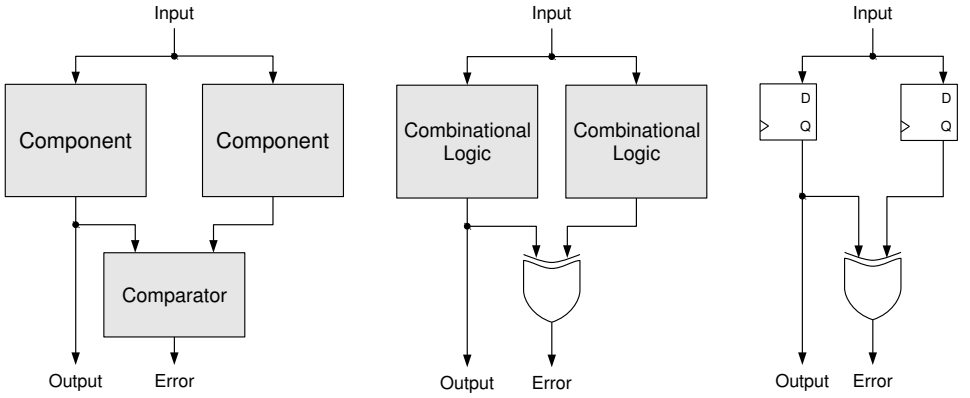
3.3.1 Hardware Redundancy

Duplex Systems

The simplest form of hardware redundancy are duplex systems, which can be used for error detection [56]. The general concept of this approach, also known as *dual modular redundancy (DMR)*, is shown in Figure 3.12a. A critical module is duplicated and a comparator unit is connected to the output signals. If the comparator finds a disagreement between the outputs, it raises an error signal to trigger some higher-level error handling mechanism. Figure 3.12b and 3.12c show two application examples for detecting SETs in combinational and SEUs in sequential circuits [51]. Note that duplex systems can provide protection against transient and permanent faults. However, simple duplication does not allow the system to decide which of the two replicated units is erroneous.

N-Modular Redundant Systems

To be able to detect faulty components, replicated systems can be built from three or more identical units. Such systems, generally known as *N-modular redundant* architectures, are able to mask errors as long as the majority of replicated units provide correct outputs. The most common form of N-modular redundant systems is triple modular redundancy (TMR), where systems are



(a) General concept. (b) Combinational logic. (c) Sequential circuits.

Figure 3.12: Duplex systems.

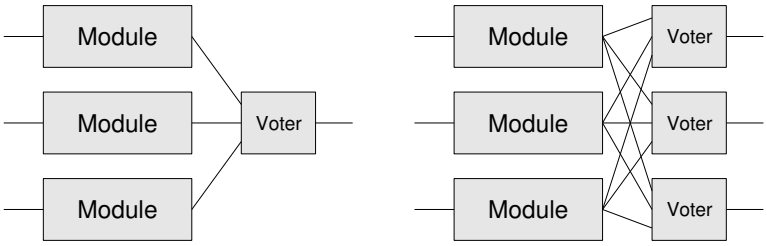
built from three replicated components. TMR was first proposed by John von Neumann [117] as a means to build reliable systems from unreliable components. This concept is therefore called *probabilistic computation*. Von Neumann introduced a basic component, the *majority organ*, which has three inputs and whose function can be described with following boolean equation:

$$m(a, b, c) = ab + ac + bc \tag{3.1}$$

Based on this majority organ von Neumann showed that it is possible to construct automata, which exhibit a lower probability of error than the basic components (organs) the automata are made of. Majority organs are nowadays well-known as voters and in CMOS technology they can be built from standard gates or more efficiently in the form of a complex gate, which implements the boolean equation above. Figure 3.13a shows the schematic of a TMR system, as proposed by von Neumann. As long as no more than one replicated unit is erroneous, the system will deliver correct outputs. However, in this configuration the voter is a single point of failure and as such might dominate the failure probability of the entire system. Although faults are typically less likely to happen in voters in comparison with the replicated components [78], for ultra high-reliable systems the use of redundant voters might be advisable. A schematic of a fully replicated system is illustrated in Figure 3.13b.

Modular redundancy can be applied by replicating the entire system/chip, but it is often reasonable to break the design down into smaller components, which are then replicated and interconnected to form the target system. Adding voters on signals between subcomponents ensures that errors are contained within a single subcomponent. Finding an optimal size (cluster size) and partitioning for subcomponents, is a critical design task, which has a significant effect on the overall system reliability. In [104] an elaborate design methodology is therefore presented that extends traditional circuit synthesis and logic optimisation with reliability specifications and constraints for optimising not only area, power and performance but also reliability. This also includes steps for finding a reliability-optimal partitioning of a synthesised netlist.

Both transient and permanent faults can be mitigated with modular redundant systems. In



(a) Single voter TMR. (b) Replicated voter TMR.

Figure 3.13: Triple modular redundancy.

the latter case, performing maintenance to replace failed components plays a crucial role, since the reliability of a degraded modular redundant system can actually drop below the reliability of a simplex system [56]. For transient faults it was traditionally assumed that they are masked as long as they only affect a minority of replicated components and unlike permanent faults have no long-term effect on the system. Multiple transient faults could therefore be easily tolerated, if their occurrences are well separated in time [117]. However, this is only true as long as replication is applied at the level of basic components. When modular redundant systems are designed at a higher level and the replicated modules contain sequential machines, John F. Wakerly established that even a single transient fault can have a permanent effect [118].

Wakerly therefore investigated active recovery and resynchronization schemes for sequential TMR circuits [118]. A sequential machine can be represented by a 5-tuple $\langle I, Q, Z, \delta, \omega \rangle$, where I is a set of input vectors, Q is a set of states, Z is a set of output vectors, $\delta : Q \times I \rightarrow Q$ is the next state function, and $\omega : Q \rightarrow Z$ is the output function. If a transient fault affects the computation of the next state function, a sequential machine can make a transition into an erroneous state Q_e, which is different from the correct state Q_c. In general a sequence of input vectors i, applied to the states Q_e and Q_c, will not lead to a common state again, i.e., $\delta(Q_e, i) \neq \delta(Q_c, i)$. In other words, once the state of a sequential machine is corrupted future states may be incorrect as well.

Long-term corruption of a replicated module of a modular redundant system obviously is undesirable, as it might impair the system's ability to mask future faults. Consequently, sequential machines have to be built in a way, where for every pair (Q_e, Q_c) an input sequence i_r exists such that $\delta(Q_e, i_r) = \delta(Q_c, i_r)$ does hold. Wakerly calls sequential machines with this property *restorable* and an appropriate i_r is named *restoring sequence*. Note that for different erroneous states different restoring sequences might be needed. A widely-used pattern how to build a restorable circuit can be seen in Figure 3.14, where voters are directly placed after every sequential element, i.e., flip-flop. In this circuit architecture faults are immediately masked and an erroneous state of one replicated module will be overwritten in the next clock cycle. Therefore *any* input sequence of length 1 basically is a restoring sequence. In Chapter 4 we will develop more intricate architectures to build restorable sequential circuits for GALS-based systems.

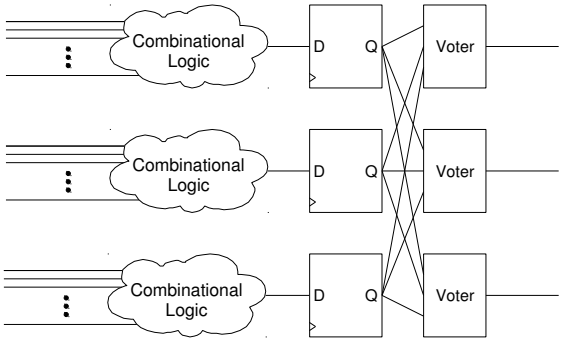
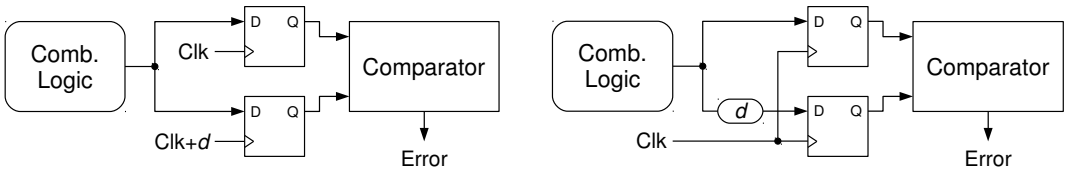


Figure 3.14: Conventional voter architecture for triplicated sequential circuits.

3.3.2 Temporal Redundancy

Fault/Error detection and correction techniques can also be implemented with temporal/time redundancy. Nicolaidis [79], e.g., presented architectures that can tolerate single event transients in combinational logic. The circuits rely on the transient nature of faults and therefore evaluate the output signal at different points in time. Figure 3.15a and Figure 3.15b show two realisations of this concept for detecting SETs. In both cases the output of the combinational logic is captured by two flip-flops and then evaluated by a comparator circuit, which generates an error output in case of an inconsistency. In the first solution the second flip-flop is clocked with a phase offset of d, thereby acquiring a newer sample of the output. With this setup one of the two flip-flops certainly captures the correct output value, if the maximum length of an SET pulse is known to be less than d. The same effect can be achieved with the second circuit implementation in Figure 3.15b. Here the input of one flip-flop is delayed by d. Both flip-flops are supplied with the same clock signal, where the clock period, however, needs to be increased by d to accommodate for the additional delay in the data path. This way the upper flip-flop captures the most recent output value, and the lower one samples an older version.



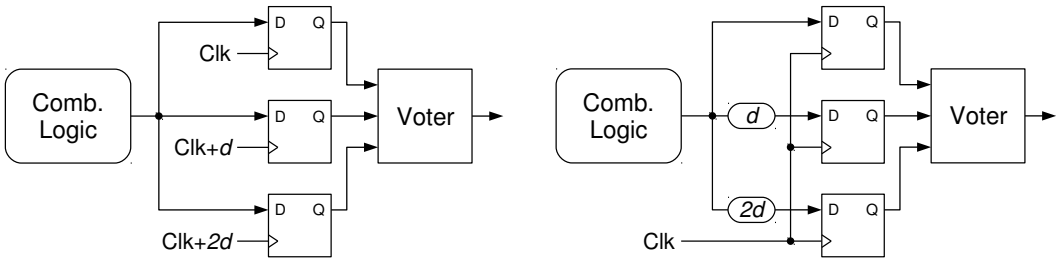
(a) Delay on clock.

(b) Delay on data path.

Figure 3.15: Fault detection in combinational circuits.

If faults are supposed to be masked, like in modular redundant approaches, three or more output samples have to be taken at different points in time. Assuming that the majority of the captured samples represents the correct output value, voting can be performed. Figure 3.16a and Figure 3.16b depict two circuit architectures by Nicolaidis [79] that work similar to a TMR system in the time domain. In both implementations three output values are sampled with offsets

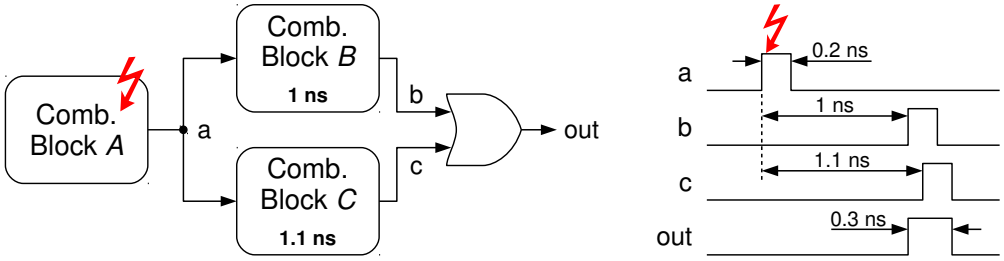
of 0, d and $2d$. Two of out three samples therefore are certainly correct, if the fault duration is shorter than d. A voter can then recover the correct output value.



(a) Delay on clock. (b) Delay on data path.

Figure 3.16: Fault masking in combinational circuits.

As could be seen, the maximum length of a transient fault at the output of the combinational logic is a critical parameter and therefore needs to be known at design time. This length depends on a variety of parameters, including the characteristics of the process that caused the fault (e.g., amount of energy deposited in case of a particle hit), the used technology, or the circuit structure [79]. Reconvergent forks in the combinational circuits, e.g., can lead to broadening of an SET pulse, if branches of the fork have different delays. Consider the example shown in Figure 3.17, where the output of Block A fans out into two other blocks with a delay of $1\,ns$ and $1.1\,ns$. The outputs of these blocks then are joined again by an OR-gate. An SET pulse in Block A can trigger two faulty pulses at the inputs of the OR-gate, which then results in a longer output pulse. Reconvergent paths, therefore, have to be carefully analysed and delays on different branches should be balanced during synthesis to avoid such pulse broadening effects.



(a) Two branches with different delays. (b) Pulse broadening on output.

Figure 3.17: Broadening of SET pulses on reconvergent paths.

3.3.3 Information Redundancy

A wide-spread method for protecting data, e.g., stored in unreliable memories or transmitted over unreliable communication channels, is the use of *error detecting* or *error correcting* codes. Although many pages could be written about the great variety of existing coding schemes and

information theoretical and algebraic foundations fill complete books, we restrict this section to a short introduction of basic principles and present only two codes, which we will use later on in this thesis: *Parity check codes* and *Hamming codes*.

In order to protect data words against errors redundant symbols, called *control* or *parity check symbols*, are added based on a specific rule [87]. This process is called *encoding* and transforms a data word, made of k symbols, into a codeword with n symbols, where $n > k$. The symbols are elements of some predefined alphabet A, and data and codewords can usually be interpreted as vectors of A^k and A^n, respectively. Typically a finite field $GF(q)$ is used as alphabet. The most important case is the field $GF(2)$, which only consists of two elements, 0 and 1. Codes that are defined over this alphabet are consequently known as *binary codes*. Addition and multiplication in $GF(2)$ are performed modulo 2 and therefore correspond to logical XOR and logical AND operations, which can be easily implemented in digital circuits.

A code C is characterised by its encoding function $f_C : A^k \rightarrow A^n$, which uniquely maps each data word to a codeword. If the data word is directly embedded in the generated codeword, i.e., $f_C(a_1a_2 \dots a_k) = (a_1a_2 \dots a_k \; c_{k+1}c_{k+2} \dots c_n)$ for $(a_1a_2 \dots a_k) \in A^k$, the code is called *systematic*. A code can also be described as a set of possible codewords $C = \{f_C(\vec{a}) \mid \vec{a} \in A^k\}$, where $C \subset A^n$ for all practical codes. The most important parameters of a code are [87]:

- Length of the data words: k.
- Length of the codewords: n.
- Information rate: $R = k/n$.
- Minimum distance of codewords: d.

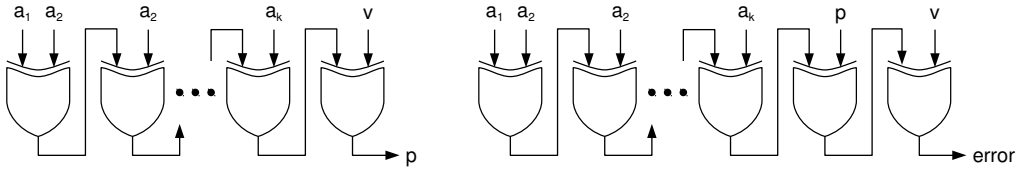
The minimum distance d is defined as $min\{d(\vec{a}, \vec{b}) \mid \vec{a}, \vec{b} \in C, \vec{a} \neq \vec{b}\}$, where $d(\vec{a}, \vec{b})$ is the *Hamming distance* of two codewords. This is a fundamental parameter because it determines the error detection and correction capabilities of a code.

Theorem 1. *A Code C with minimum distance d can detect up to $d - 1$ errors in a codeword, and can correct up to $\lfloor \frac{d-1}{2} \rfloor$ errors. In order words, for detecting t errors the minimum distance needs to be $d \geq t + 1$, and $d \geq 2t + 1$ for correcting t errors [87].*

Proof. If $d \geq t+1$, f symbols of a codeword $\vec{c} \in C$ can be changed, $0 < f \leq t$, and the resulting erroneous vector will not be transformed into another valid codeword. Thus up to t errors can certainly be detected. However, for error correction it is also necessary to identify *what* symbols are erroneous. This can only be guaranteed for t errors, if $d \geq 2t + 1$. Consider the sphere of radius t around a codeword $\vec{c} \in C$, defined as $B_t(\vec{c}) = \{\vec{x} \in A^n \mid d(\vec{x}, \vec{c}) \leq t\}$. If $d \geq 2t + 1$, then the spheres of all codewords are disjoint, i.e., $B_t(\vec{c_1}) \cap B_t(\vec{c_2}) = \emptyset, \forall \vec{c_1}, \vec{c_2} \in C, \vec{c_1} \neq \vec{c_2}$. Consequently, a vector with a maximum of t errors can only be element of a single sphere $B_t(\vec{c})$, and $\vec{c} \in C$ needs to be the correct codeword. □

Parity Check Code

The most simple error detecting code is a parity check code, where just a single symbol or bit, in case of a binary code, is appended to the data word. The value of the parity bit hereby needs to be set to *zero* or *one*, so that the sum of all data bits and the parity bit modulo 2 equals a predefined value v, i.e., $a_1 + a_2 + \cdots + a_k + p \equiv v \mod 2$ [106]. This sum is called a *checksum function* and can be recomputed by the decoder to evaluate the validity of the codeword. If checksum result v is arranged to be *zero*, the number of ones in correct codewords is always even, and the code is therefore called *even parity code*. In the other case, when v equals *one* for a correct codeword, we speak of an *odd parity code*. Parity check codes have a minimum hamming distance of 2 and can therefore detect single bit flips in a codeword. More generally, any odd number of errors can be detected since this changes the result of the checksum function. As mentioned above, adding numbers in $GF(2)$ is performed with a logical XOR-operation. Encoder and decoder circuits therefore are simply built from chains of XOR-gates, as can be seen in Figure 3.18.



(a) Encoder. (b) Decoder.

Figure 3.18: Circuits for parity codes.

Hamming Code

A code $C(n, k, d)$ is called a linear code, if its codewords form a vector space. Consequently, the sum of two codewords, or a codeword multiplied by a scalar is again an element of the code. For linear codes, such as Hamming codes, a basis of this vector space $\{g_1, g_2, \ldots, g_k\}$ can be used to build a *generator matrix* $\mathcal{G}$ of the code. $\mathcal{G}$ is a $k \times n$ matrix, where the rows are assembled from the basis vectors g_i. An encoding function f_C of a linear code C can then be expressed in terms of its generator matrix: $f_c(\vec{a}) = \vec{a} \cdot \mathcal{G},\ \forall \vec{a} \in A^k$.

Example 3.1. *Consider, e.g., the generator matrix of a (7,4) Hamming code:*

$$\mathcal{G} = \left(\begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{array}\right)$$

The encoding of $\vec{a} = (a_1, a_2, a_3, a_4) = (0, 1, 0, 1)$*, e.g., can then be computed as follows:*

$$f_c(\vec{a}) = \vec{a} \cdot \mathcal{G} = (0, 1, 0, 1, 1, 0, 1)$$

Note that the generator matrix in the example above has the canonical form $\mathcal{G} = \left(\begin{array}{c|c} \mathcal{E}_k & \mathcal{M} \end{array} \right)$, where $\mathcal{E}_k$ denotes the $k \times k$ identity matrix and $\mathcal{M}$ is some other $k \times (n-k)$ matrix. In this case the generator matrix then describes a systematic code, where the bits of the data words appear as the first k bits in the codewords. Furthermore, it can be seen that the columns of submatrix $\mathcal{M}$ specify the checksum functions for the computation of the check bits in the codewords. Consider the first column of $\mathcal{M}$ in Example 3.1 with the value $(1, 1, 1, 0)$. This specifies that the first check bit is the sum of the first three bits of a data word, i.e., $p_1 = a_1 + a_2 + a_3$. An encoder circuit can therefore be built by assembling appropriate XOR-chains for every check bit, just like we presented for parity check codes above.

Another way to characterise a code C is to use a matrix $\mathcal{H}$, called *parity check matrix* or *control matrix*. $\mathcal{H}$ is defined so that the product of a vector with the transposed form of $\mathcal{H}$ yields the null vector, if and only if this vector is a correct codeword, i.e., $\vec{c} \cdot \mathcal{H}^T = \vec{0} \Leftrightarrow \vec{c} \in C$. For binary codes the control matrix can be easily derived from the generator matrix. If $\mathcal{G}$ is specified in canonical form, $\mathcal{G} = \left(\begin{array}{c|c} \mathcal{E}_k & \mathcal{M} \end{array} \right)$, the control matrix is constructed from the transposed form of $\mathcal{M}$ appended by an $(n-k) \times (n-k)$ identity matrix, i.e., $\mathcal{H} = \left(\begin{array}{c|c} \mathcal{M}^T & \mathcal{E}_{n-k} \end{array} \right)$,

Example 3.2. *The control matrix of the (7,4) Hamming code presented in Example 3.1 is:*

$$\mathcal{H} = \left(\begin{array}{cccc|ccc} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{array} \right)$$

If we multiply the codeword we computed in the example above $\vec{c} = (0, 1, 0, 1, 1, 0, 1)$ with $\mathcal{H}^T$, it can be easily verified that indeed $\vec{c} \cdot \mathcal{H}^T = \vec{0}$.

Hamming codes are described by their control matrices [87]. To build the control matrix of a Hamming code with r check bits, the columns of $\mathcal{H}$ are filled from all possible vectors of $[GF(2)]^r \setminus \vec{0}$. The control matrix consequently is a $r \times (2^r - 1)$ matrix (cf. Example 3.2, where $r = 3$). The length of the codewords consequently is $n = 2^r - 1$, and the number of data bits that can be encoded equals $k = n - r = 2^r - r - 1$. It can be shown that Hamming codes have a minimum distance of $d = 3$, and can therefore detect double errors, or correct single errors [87].

Error detection for a given vector $\vec{x} \in A^n$ can be performed by multiplication with the transposed control matrix. The result of this multiplication is called *syndrome*, denoted $s_H(\vec{x})$. Due to the definition of the control matrix, the syndrome is zero if $\vec{x} \in C$, and non-zero otherwise. In case of Hamming codes every syndrome vector can uniquely be related to an error vector $\vec{e}$, which identifies the erroneous bit in $\vec{x}$, assuming that only single-bit errors can occur.

Example 3.3. *Let $\vec{x} = (0, 1, 0, 1, 1, 0, 1)$ and $\vec{y} = (1, 1, 0, 0, 0, 0, 1)$ be two valid codewords of a (7,4) Hamming code. Assume a single error, flipping the second bit, occurs during the transmission of $\vec{x}$ and $\vec{y}$ over an unreliable channel. On the receiver side thus $\vec{x'} = (0, \mathbf{0}, 0, 1, 1, 0, 1)$ and $\vec{y'} = (1, \mathbf{0}, 0, 0, 0, 0, 1)$ are delivered. It can be easily checked that both vectors have the same syndrome:*

$$s_H(\vec{x'}) = \vec{x'} \cdot \mathcal{H}^T = s_H(\vec{y'}) = \vec{y'} \cdot \mathcal{H}^T = (1, 1, 0)$$

The syndrome $(1, 1, 0)$ consequently is mapped to the error vector $\vec{e} = (0, 1, 0, 0, 0, 0, 0)$.

With the knowledge of the mapping between syndromes and error vectors, correction of an erroneous codeword can be easily implemented. Figure 3.19 shows the high-level schematic of the implementation of a decoder circuit. First the syndrome is computed from the possibly erroneous input word. Then a lookup table can be used to determine the error vector, which is finally added to the input word. This addition, according to arithmetics of vectors over $GF(2)$, is performed bitwise with XOR-gates, and simply inverts erroneous bits in the input vector to retrieve the correct codeword again.

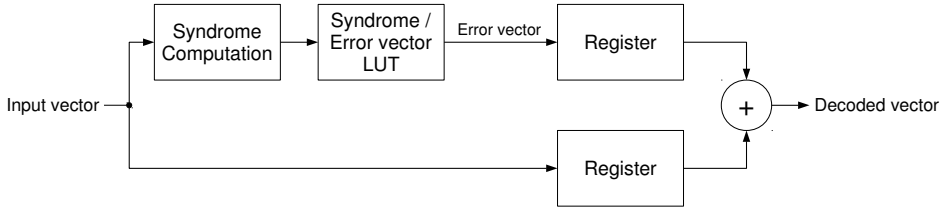


Figure 3.19: Decoder circuit for Hamming codes.

Extended Hamming Codes

Let $\mathcal{H}$ be the control matrix of a regular Hamming code, as defined above. The control matrix of an extended Hamming code can then be formed as follows:

$$\mathcal{H}_{ext} = \begin{pmatrix} & & & & 0 \\ & & \mathcal{H} & & 0 \\ & & & & \vdots \\ & & & & 0 \\ 1 & 1 & 1 & \cdots & 1 \end{pmatrix}$$

The last row of $\mathcal{H}_{ext}$ defines the checksum function of an additional check bit p_{r+1}, which is introduced for extended Hamming codes. Given a codeword $\vec{c}$ of a regular Hamming code, $\vec{c} = (a_1, a_2, \ldots, a_k, p_1, p_2, \ldots, p_r)$, the new check bit's value is simply the sum of all data bits and the regular check bits, i.e., $p_{r+1} = \sum_{i=1}^{k} a_i + \sum_{i=1}^{r} p_i$.

This extension improves the minimum distance d from 3 to 4. A regular Hamming code with $d = 3$ can perform safe error correction, if only *single* errors occur. Error correction fails in case of *double* errors, since two erroneous bits in some codeword $\vec{c_1} \in C$ would be confused with a single-bit error of some other codeword $\vec{c_2} \in C, \vec{c_1} \neq \vec{c_2}$. With $d = 4$ an extended Hamming code, however, can safely distinguish single and double-bit errors. Single errors then can be corrected, while double errors are flagged as uncorrectable errors. Codes with these capabilities are called *single-error correcting, double-error detecting* (*SECDED* codes).

CHAPTER 4

Fault-tolerant Computation in Synchronous Modules

In Section 3.2 we explained that computation in GALS systems is performed in locally synchronous modules. This chapter is devoted to their protection against transient faults and to some extent to the mitigation of permanent faults. Since the core parts of GALS modules are conventional synchronous circuits, many of the existing fault tolerance techniques, which we have described in Section 3.3, are eligible candidates for application in a GALS setting. For this thesis we decided to pick modular redundancy for detailed investigation, because it is well-known and widely used in industry. It should, however, be made clear that by no means we want to advocate modular redundancy as the only possible, or even the best solution for building hardened GALS modules. As fault tolerance always incurs some form of additional costs for the overall system, the choice of the employed fault tolerance scheme very much depends on the specific application and its requirements and constraints.

4.1 Modular Redundancy in GALS

As we will see in this Chapter, a crucial design decision for implementing modular redundant GALS systems is the chosen replication scheme for individual modules. This decision entails tremendous consequences for the overall system architecture, the fashion in which error recovery is performed and for system attributes like area, performance and reliability. A simple strategy is to replicate all circuit elements within the boundaries of the module's *core logic*, i.e., the synchronous parts of the GALS module, which are encapsulated within an asynchronous wrapper. In case of a TMR system[1] all combinational and sequential parts of the core logic are triplicated and voters added, as is illustrated in Figure 4.1. This replication strategy closely resembles the

[1] Please note that most of the time in this chapter we will refer to triple modular redundant systems and that all the circuits presented assume a TMR configuration. The results can, however, be easily applied to general modular redundant systems having an arbitrary (odd) number of replicated components.

way modular redundant circuits are typically implemented at gate level (cf. Section 3.3.1). Aside from the fact that we use a stoppable clocking scheme, for reasons that will become apparent shorty, Figure 4.1 does not expose many details about the asynchronous wrapper surrounding the core logic. Wrapper implementation, especially with respect to replication of asynchronous I/O components and ports, will be addressed at a later point in Section 4.4 of this thesis. The clock generator, however, needs immediate attention since it directly affects the replication strategy of the core logic. As can be seen in Figure 4.1, there is only a single clock generator driving a single clock tree, which connects to all flip-flop endpoints of the triplicated module core. Since voting is performed synchronously on every clock cycle in this replication scheme, maintaining one common clock domain and therefore a single clock generator is unavoidable. Obviously this creates a *single point of failure* in the redundant system and a single transient fault in the clock generator or the clock distribution network could affect several replicated units during the same clock cycle, thereby compromising the overall circuit state beyond recovery. A spurious clock transition could, e.g., cause setup/hold time violations and consequently metastable upsets, or simply lead to data inconsistencies when flip-flops sample at a wrong point in time.

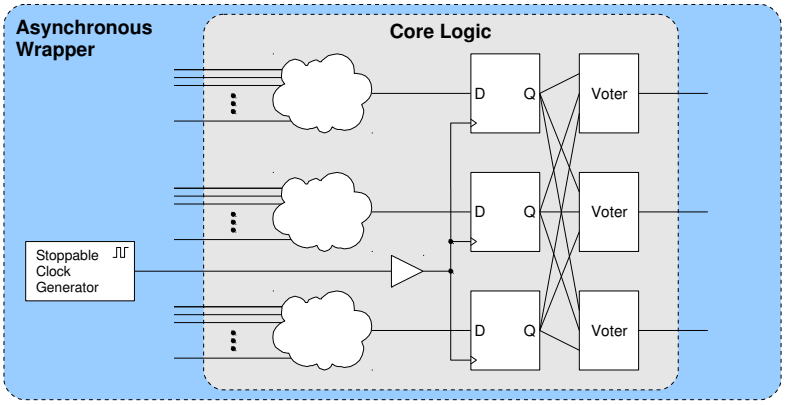


Figure 4.1: Triplicated GALS module.

As the clock generator and clock tree consist of standard gates, which can be assumed to be susceptible to faults just like the rest of the circuit, we believe this single point of failure poses a major drawback of this replication strategy. Further disadvantages are:

- In a modular redundant system with n replicated units, the overall area of the GALS module is increased more than n-fold (including overheads from voters). It can be assumed that the dimensions of the clock distribution network also grow a in similar fashion. In case of tight timing constraints, achieving timing closure for the module might therefore be much more challenging. Recall that the one of the basic principles of GALS is to break down a complex globally synchronous circuit into several small locally clocked modules in order to alleviate timing issues. A replication strategy that significantly increases the module size again, might therefore reduce the effectiveness of the GALS design approach.

- The voter components introduce additional delay in the critical path of the circuit and contribute to area and power overheads.
- Using majority voters requires a fully connected network between the replicas to exchange redundant state information. If a circuit with m flip-flops is fully replicated n-fold, following the scheme shown in Figure 4.1, $m \cdot n^2$ connections are required. This significantly increases the complexity of the interconnect and thereby has direct implications on the physical implementation of the modular redundant circuit. Limited routing capacities and strict timing constraints therefore enforce a compact circuit layout, where replicated components have to be placed in close proximity. This might reduce the reliability of the resulting system, especially if multiple-bit upsets should become more frequent with future technologies. In general, physical separation is a desirable property in fault-tolerant systems since this reduces the possibility of *spatial proximity faults* [55].

In this chapter we will therefore present the results of our research to find alternative replication strategies and state restoration mechanisms to avoid the above mentioned issues. Two solutions, we have published in [60] and [64], will be discussed in detail. In both approaches we use stoppable ring oscillators to generate the clock for GALS modules, and take advantage of the fact that these provide a simple mechanism to freeze the circuit so that some form of recovery routine can be conveniently performed.

4.2 Approach I: Parallel Recovery

A main concern, as mentioned above, is to avoid critical single points of failure in a fault-tolerant design. Thus, an early design decision was that the clock generator and the clock generation network should be replicated as well. While this removes the clock as single point of failure, this concept also requires a modification of the voting mechanism. For three independent systems voting can no longer be performed for each and every clock cycle. Even if the clock generators are set to the same frequency and start with the same phase offset, frequency jitter and phase drift will eventually lead to incorrect voting results or metastable upsets. In Figure 4.2 we therefore suggest an alternative architecture, in which the three replicas of the synchronous module are all operated with their own local clock and the voting is executed at safe *checkpoints*, where normal circuit operation is interrupted. Consequently, we distinguish between two operation modes: *computation* and *recovery*. During computation mode every replicated module runs independently at its own pace and progresses with its computational task. When a checkpoint is reached, the modules change to recovery mode and wait until all replicas have stopped their operation, i.e., have stopped their clock generators. Then a common *recovery cycle* is inserted into the execution in order to correct recent soft-errors, which might have compromised the state of one replica. Since all clock generators have been stopped, the voting can be safely performed and the voter outputs are fed back to the inputs of the corresponding flip-flops. As can be seen in Figure 4.2, a multiplexer controlled by a *recover* signal (*rec*) switches between the normal data signal of the combinational logic and the feedback signal of the voters. The recovery is done in a single cycle in parallel for all the flip-flops, hence the name *parallel recovery*. Note that this is a rollforward error handling approach. In contrast to rollback recovery, where checkpoints denote

snapshots of the system state, here we use the term *checkpoint* to refer to a point in time, where the system state is *checked* and state restoration is performed. A similar scheme for conventional synchronous TMR systems was presented in [123] (see related work in Section 4.7).

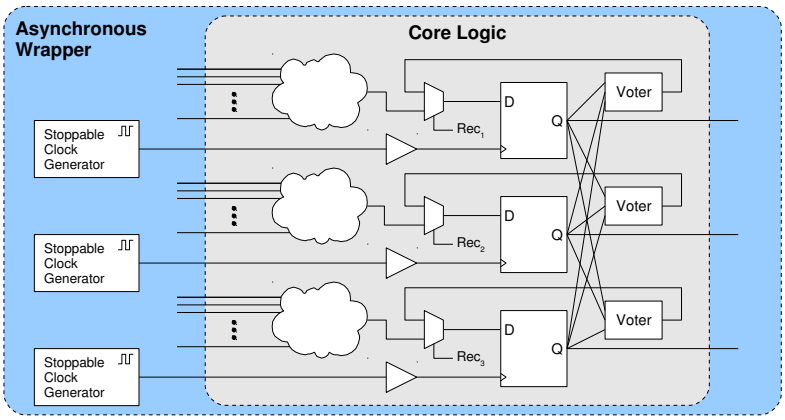


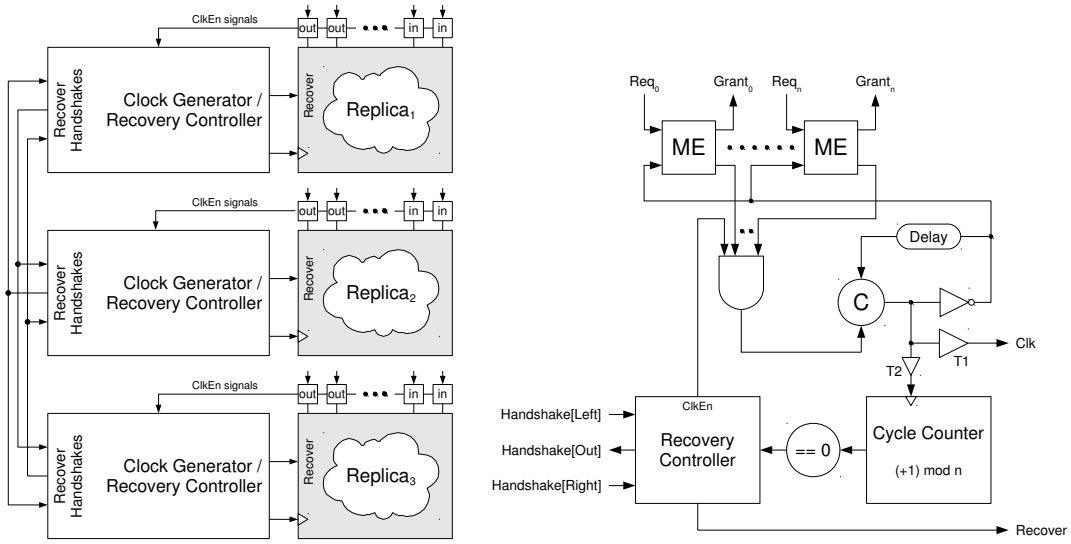
Figure 4.2: Adapted voting mechanism.

4.2.1 Recovery Controller

Figure 4.3a shows a schematic of the overall TMR system with redundant clocking. In order to coordinate the recovery process, the local clock generators of the three replicas are augmented by a recovery controller. These controllers are then interconnected by a network of asynchronous handshake signals, which can be interpreted as request signals indicating when a replica is ready to start the recovery process. The controllers also generate the necessary *recover* signal, which switches the multiplexers of the replicated units (see Figure 4.2).

As can be seen in Figure 4.3b, the extensions to the clock generator include three main parts: A cycle counter, a comparator component and, most importantly, the recovery controller, which is basically an asynchronous state machine. The purpose of the cycle counter is very simple. All it does is to keep track how many cycles have passed since the last recovery process. After a predefined number of cycles n, the counter wraps back to zero, which in turn triggers the comparator to raise its output signal. This indicates that the next cycle will be a recovery round. Note that this is a very basic mechanism to introduce regular checkpoints in the execution. We will discuss details about more sophisticated strategies, like dynamically scheduled checkpoints, in Section 4.5. The overall recovery mechanism, however, stays the same, independent from the points in time when the recovery cycle is inserted.

Due to the rising transition of the comparator the recovery controller is activated and immediately deasserts the *clock enable* signal. Consequently, the ring oscillator is stopped and the next rising clock edge is delayed until the *enable* signal is asserted again. Furthermore, the recovery controller activates the recover request over the signal *Handshake[Out]* to the other two controllers and waits for them to likewise raise their recover requests. Once these requests have been received, all three replicas have stopped their operation and it is safe to perform majority voting on the circuit state. A single clock edge then is issued and the voter results are latched



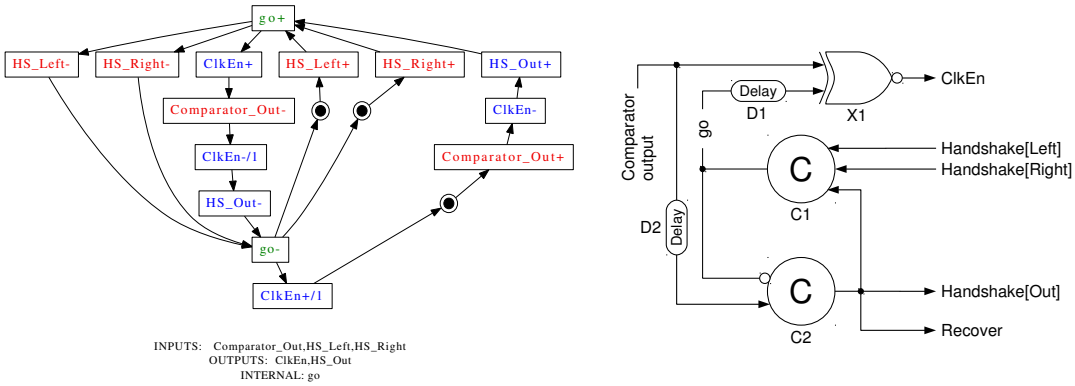
(a) Top-level view.

(b) Clock generator and recovery circuitry.

Figure 4.3: GALS TMR system.

into the respective flip-flops. Furthermore, the cycle counter is incremented to one and as a consequence the comparator output is deasserted again. Now the recovery controller releases the request signal *Handshake[Out]*, which indicates that the recovery has been completed. When the other two controllers also release their request signals, the local controller knows that all three replicas have successfully completed the recovery. At this point the value of all replicated flip-flops is consistent again, even in case a soft-error had occurred before. Finally every controller is allowed to set the *clock enable* signal to one again and normal operation is resumed.

The implementation of the recovery controller is a central part of the presented fault-tolerant architecture. Since the recovery process demands to switch the clock generator off and on again, the state machine of the recovery controller clearly cannot be clocked by the clock generator itself. Thus, it needs to be implemented as an asynchronous control circuit. Figure 4.4a shows a signal transition graph (STG), which models the behaviour of the controller as described above. STGs are a specific kind of petri-net, which can be synthesised into asynchronous control circuits. We used the tool *Petrify* [19] to perform this task. The resulting circuit, after performing some minor manual optimisations, can be seen in Figure 4.4b. It only consists of two Muller C-elements, a delay element and an XNOR gate. The main part of the state machine is implemented by the two C-gates, which store the current state and also compute the next state. The gate *C2* is responsible for detecting the start of a new recovery cycle, as indicated by the comparator, and raises the request output in order to notify the other two replicas. *C1* then performs the join operation, which waits for all replicas to be stopped and subsequently triggers the voting mechanism by asserting the *go* signal. This join exactly matches the transition *go+* at the top node of the STG, which is only enabled when all of the three handshake signals are asserted. The XNOR gate enables and disables the clock generator appropriately during all stages of the recovery process. Note that the *clock enable* signal is not only deasserted when waiting for the



(a) STG describing the behaviour.

(b) Synthesised circuit.

Figure 4.4: Implementing the recovery controller.

beginning of the recovery but is also disabled directly after the recovery cycle. This ensures that a fast replica is stalled until the other replicas have confirmed that they are done with the voting as well and have safely captured the voter outputs. In this case all three handshake signals are reset to zero, the C-gate *C1* deasserts the *go* signal and in turn re-enables the clock generator.

4.2.2 Timing Constraints

Circuits synthesised by *Petrify* are speed-independent. Nevertheless, these circuits still demand a specific behaviour from the environment for safe operation. The environment is only allowed to change its input when the circuit has produced the outputs related with the previous input changes. This mode of operation is called input-output mode [102]. Furthermore, there are some timing constraints which arise from the interaction of the asynchronous control logic with the synchronous replicas and the clock generator. The following circuit paths need to be considered:

- Clock to clock enable: When the clock generator issues the final clock edge before the recovery starts, i.e., the clock edge upon which the cycle counter resets to zero, it must be guaranteed that the clock generator is disabled before another clock edge can be issued. Consider the path that starts at the C-element output of the clock generator, includes the cycle counter, the comparator, the recovery controller and the AND-gate, and finally ends at the lower input of the C-element. Clearly, the overall delay of this path must be shorter than the clock period to disable the clock generator in time. While the total sum of the clock-to-output delay of the cycle counter and the combinational delays of the comparator, the recovery controller and the AND-gate might be well below the clock period, this path also includes the clock tree latency, which needs to be considered carefully. If a single clock tree is used for the entire GALS module and the cycle counter is just another endpoint of this tree, this latency might be easily longer than a clock period. Therefore, we propose to implement two clock trees, as can be seen in Figure 4.3b. The large clock tree *T1* fans out to the the regular GALS module, whereas *T2* exclusively clocks the cycle

counter. Since the cycle counter just contains a few flip-flops the latency of *T2* will be minimal, possibly being just the delay of a single clock buffer.

However, this creates a mesochronous system, where the GALS module lags behind the recovery circuitry by the latency difference of the two clock trees. Consequently, the recovery operation must not start immediately when the clock generator is disabled but only after the all clock transitions have propagated through the module's clock tree *T1*. This can be achieved by delaying the comparator output signal for the lower input of C-element *C2* in the recovery controller (see delay element *D2* in Figure 4.4b). If *D2* is matched to the latency difference of the two clock trees, the recovery starts right in time after all computation cycles have been executed by the GALS module. A similar method where the delay of asynchronous control signals is matched to clock tree latencies to ensure safe data transfers in GALS SoCs has been proposed in [23].

- Setup time of data path: Obviously, the voting results need to be stable at the flip-flop inputs before the clock generator is enabled again and the next clock edge is issued. In order to ensure this setup requirement holds, a delay element on the *go* signal can be included in the recovery controller (see Figure 4.4b). This delays the assertion of the *clock enable* signal and gives voters and the multiplexers time to stabilise.

- *C2* output to local *C1* output: This path should be shorter than the external control loop starting from the outgoing handshake signal, crossing the remote recovery controller of the other replicas and returning with the incoming handshake signals. Otherwise the other two replicas might withdraw their handshake requests again before the C-gate *C1* of the local recovery controller is even able to react on the asserted handshakes. This timing relationship can be easily met since a local path is usually faster than the external paths.

4.2.3 Robustness of the Recovery Circuitry

There are two different types of erroneous behaviour that can occur in a recovery circuitry: 1) Benign transient faults which only cause errors in the local replica. Although they can spoil the state of the local replica, this can be tolerated as long as the other two replicas are able to continue their operation without problems. When the transient fault disappears, the faulty replica continues to work and its state will be corrected on the next correct recovery cycle. Consider, e.g., an upset in the cycle counter of the replica R_3. This just corrupts the state of R_3, which might then either start the recovery process a certain number of clock cycles too early or too late, depending on the counter value after the upset. Nevertheless, the counter will eventually reset to zero and the recovery mode will be initiated. The other two fault-free replicas will start the recovery as planned in the correct clock cycle. At this point the registers of R_3 will likely store different values than the other replicas, since the computation was stopped at the wrong cycle. However, after the restoration process is performed this inconsistency will be purged and all three replicas maintain the same circuit state again and can continue their computations.

2) The other type of erroneous behaviour are faults that drive the three recovery controllers into an inconsistent global state, which could lead to a deadlock of the whole system. As explained above, the recovery controllers are basically three instances of an asynchronous state

machine, which are interconnected and control the recovery procedure in a distributed manner. Table 4.1 shows all possible deadlock states. An analysis of these states reveals that two conditions need to be fulfilled for a deadlock to occur: I) The three handshake signals need to be inconsistent, i.e., the recover request is asserted for at least one recovery controller and deasserted for at least one other controller. II) All the local clock generators need to be disabled. This is the case when the *go* signal and the comparator output have different signal values and thereby deassert the output of the XNOR gate, which drives the clock enable signal (see Figure 4.4b).

Table 4.1: Deadlock states.

$Compare_1$	HS_1	go_1	$Compare_2$	HS_2	go_2	$Compare_3$	HS_3	go_3
0	0	1	0	0	1	1	1	0
0	0	1	1	1	0	0	0	1
0	0	1	1	1	0	1	1	0
1	1	0	0	0	1	0	0	1
1	1	0	0	0	1	1	1	0
1	1	0	1	1	0	0	0	1

An example for a fault causing a deadlock can be seen in Figure 4.5. At the beginning the handshake signal of the local replica and the handshake signals of the other two replicas are zero, i.e., all three replicas operate independently in normal mode. Assume that the local copy is slower than the other two instances, which pull up their handshake signals first (*step 1*). Since the local handshake output is still zero, the C-gate *C1* holds its current output value, i.e., it remains zero. Now assume this C-gate is affected by a single event effect at this point in time and flips the stored value to one. Then the *go* signal is erroneously activated (*step 2*), which in turn deactivates the *clock enable* signal (*step 3*). Now the clock generator is stopped and the comparator output will be stuck at zero. As a consequence the local handshake output will also remain zero and block the operation of the other two replicas, which results in a deadlock.

In order to prevent such a scenario, it is necessary to protect the state of the recovery controller from upsets. As can be seen in Figure 4.4b, this state is contained in two C-elements. These can be disturbed by erroneous signal transitions either on their inputs or on the internal feedback signal of the storage loop. An option for increasing the resilience of C-elements against SETs is the use of robust C-elements, such as the Van Berkel implementation [113] or dual-rail C-elements. Although this decreases the likelihood of upsets, each of the C-elements still would be a single point of failure. Hence, we suggest a solution based on replacing the C-elements by duplicated double-checking gates (DD gates). This fault tolerance technique for asynchronous QDI circuits was presented in [46]. The modified circuit schematic is shown in Figure 4.6.

In DD gates the original gate is duplicated and then connected to two double-checking C-elements. In Figure 4.6 the duplicated gates, which perform the actual function of the circuit, are shown in white, and the double-checking C-gates, which prevent errors from propagating, in grey colour. This circuit structure is able to withstand and correct SEUs in any of the C-elements. Assume, e.g., that one of the white C-elements is perturbated leading to a corruption of the stored value. Nevertheless, the fault will not propagate because the other C-element in the duplicated set still stores the correct value and the double-checking C-gates will not change

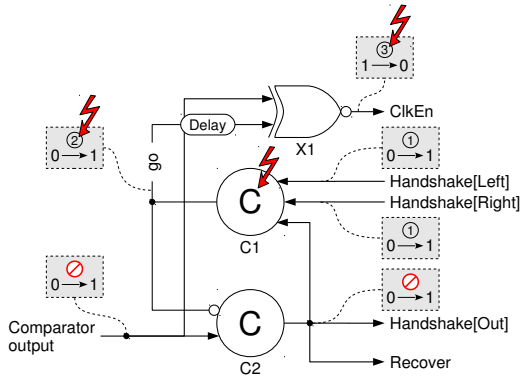


Figure 4.5: Execution leading to a deadlock.

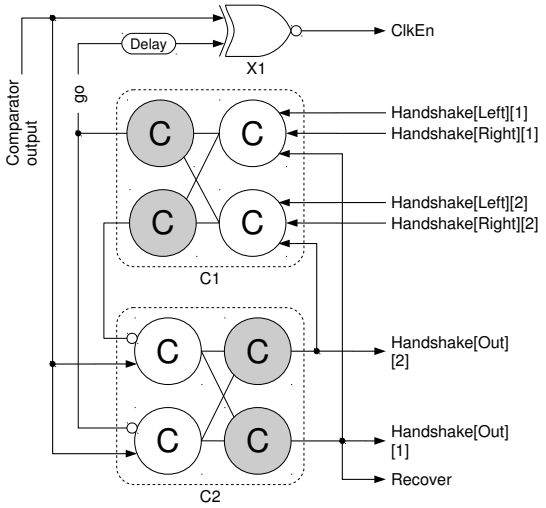


Figure 4.6: Robust implementation.

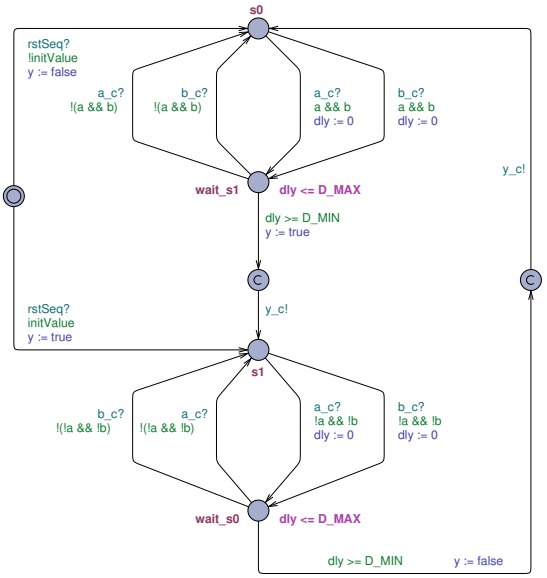
their output value, if they see inconsistent inputs. The correct value will be restored immediately when the transient error disappears and the inputs of the C-element still have the same value. If the inputs are no longer equal, the incorrect value will simply be overwritten with the next correct value, as soon as all inputs have finished their transition.

For the second case, assume that one of the double-checking C-gates is hit. Since the duplicated gates, which drive the inputs of the perturbated gate, store the same value, the corruption will be fixed almost immediately in most cases. Only when the duplicated gates perform a transition they may get inconsistent for a short period of time and therefore the recovery of the corrupted gate will be delayed. In this case it is possible that the fault disturbs the succeeding gate. This again only affects one gate within a duplicated set, which can be restored after the SEU in the preceding double-checking gate is removed. A full proof for the ability of DD gates to withstand SEUs and even some kind of multiple upsets can be found in [46].

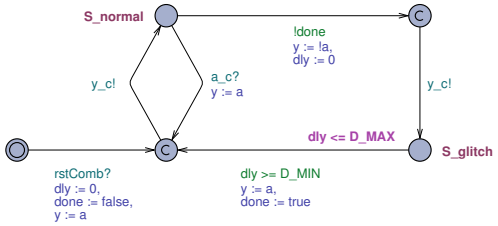
4.2.4 Formal Verification of the Recovery Controller

Unfortunately, there is still a single point of failure which may disrupt the described double-checking mechanism: As can be seen in Figure 4.6, the lower inputs of the duplicated *C2* C-gates are both connected to a single signal, i.e., the output signal of the clock cycle comparator. If long enough, glitches will overthrow the stored value of both duplicated C-gates and hereupon the values of the attached double-checking C-elements. In order to verify whether such faults can cause deadlocks or mess up the recovery scheme, we analysed the recovery controller with the UPPAAL model checker [59]. UPPAAL is typically used for verification of real-time systems, as its models are based on networks of timed automata. The query language for checking specific properties of the modelled system is a subset of TCTL (timed computation tree logic [6]).

The basic idea is to provide a gate-level model of the circuit and assign lower and upper bounds for the switching delays of the gates. This way the behaviour can be verified in the



(a) UPPAAL model of a 2-input C-gate.



(b) UPPAAL model of an SET saboteur.

1	$A\Box$ not deadlock
2	$A\Diamond(r1.hs$ or $r2.hs$ or $r3.hs)$
3	$(r1.hs$ or $r2.hs$ or $r3.hs)$ $-->$ $(r1.hs$ and $r2.hs$ and $r3.hs)$
4	$(r1.hs$ and $r2.hs$ and $r3.hs)$ $-->$ $(!r1.hs$ and $!r2.hs$ and $!r3.hs$ and $r1.clkEn$ and $r2.clkEn$ and $r3.clkEn)$

(c) Verified properties.

Figure 4.7: Models and properties for verification.

presence of glitches. In a first step, we developed suitable UPPAAL models for every gate in the circuit, i.e., the 2-input and 3-input C-gates, the XNOR and the inverter gates. Timing properties can be reflected in UPPAAL with the help of clock objects, which can be used to specify state invariants and/or guards on state transitions. The model of a 2-input C-gate, e.g., is shown in Figure 4.7a. It consists of four main states: Two stable states *s0* and *s1*, where the gate output is either zero or one and the two transition states *wait_s1* and *wait_s0*. The latter states model the switching time of the gate. The state *wait_s1*, e.g., is annotated with the invariant $dly \leq D_{max}$. Since the clock variable dly is set to zero on all input transitions, this means that the automaton must not remain in this state for more than the maximum switching delay of D_{max}. A lower bound for the switching delay is modelled with a clock guard on the output transition: $dly \geq D_{min}$. After this minimum delay the output transition is allowed to fire and the gate output y is set to one. Note that a wait state can only be entered when the gate inputs change and the guards, reflecting the logic function of the gate, are satisfied. In case of the 2-input C-gate, the guards on the input transitions for the state $wait\_s1$ consequently are a && b. Furthermore, there are also transitions leading from a wait state back to the stable state. These transitions can be fired, if an input is changed again during the switching time of the gate. Then the automaton drops back to the previous stable state. This mechanism models the behaviour of input glitches, which may or may not cause a transition at the output of the gate.

The models of all other gates have the same basic structure as the 2-input C-gate model presented in Figure 4.7a. Only the transition guards need to be adapted to the respective gate function. Using these gate models, it is possible to build a complete model of the recovery controller. This can then be triplicated and the interconnection of the handshake signals forms a model of the entire system, which we want to investigate. Unfortunately UPPAAL does not

support hierarchical models at this point in time. Thus, it was not possible to combine gates into submodels. All gates of the system had to be manually instantiated at the top-level model. In order to reduce the complexity of the this model, we decided to perform the verification on the non-duplicated version of the recovery controller, as can seen in Figure 4.4b. Note that the catastrophic scenario for the duplicated recovery controller is a glitch on the comparator output affecting the duplicated gates in the same way. Therefore, a proof that a glitch cannot compromise the non-redundant version, also proves the correctness of the redundant implementation.

Table 4.7c lists the TCTL expressions we used for checking the desired properties. The first formula verifies that there are no deadlocks on all possible execution paths. For this purpose UPPAAL conveniently provides the *deadlock* attribute. If this property holds, the three recovery controllers never get stuck during the whole operation of the circuit. Yet, this property does not guarantee that the recovery process is always performed in a consistent way. Therefore, more detailed checks are required to verify the correct execution of the handshake protocol: The second query in Table 4.7c verifies that on all execution paths eventually at least one of the recovery controllers starts a new recovery cycle by asserting its handshake signal. Subsequently, the third formula can be used to check that once one controller has started a recovery cycle, the other two controllers will eventually join the recovery process. In UPPAAL this property can be expressed by the statement ϕ $-->$ ψ, which is a short-hand notation for the TCTL expression $A\Box(\phi \rightarrow A\Diamond\psi)$. The last behaviour which remains to be verified is the property that a started recovery cycle is eventually completed by all three replicas, i.e., all handshake signals are deasserted and all clock generators are enabled again. The verification of this property is done with the last expression listed in Table 4.7c.

In a first verification step we applied the four queries to a fault-free model, where the comparator output is not disturbed. In this case all properties held as expected. For investigation of the behaviour when a fault occurs at the comparator output, we added a saboteur unit for this signal into the model. This is quite similar to the use of saboteur devices for fault injection experiments. Since an SET pulse is only a temporary inversion of the current signal value, this fault can be easily modelled by a simple state machine. As can be seen in Figure 4.7b, the saboteur unit has a single input a and a single output y. During normal operation the state machine just propagates input changes instantaneously to the output. However, when a glitch is supposed to occur, the right-hand output transition of the state *S_normal* is taken, which inverts the current input value and writes it to the output. Note that this transition is allowed to fire at any time. There are no timing constraints given. Thus, a glitch may occur at any possible system state and the verified properties apply throughout the whole circuit operation. In contrast to the time of occurrence, the length of the glitch is constrained to same bounds used for the switching time of the other gates. This restricts the number of glitch variations to a reasonable amount. Since we only want to verify the circuit for the single-fault assumption, the saboteur model is only allowed to generate a glitch for a single time. This is controlled with the variable *done*, which is set to true once the first glitch has been generated.

After adding the saboteur unit to the comparator output signal of one replica, we verified the properties again. While the first three properties could be satisfied, the forth property failed. An analysis of the trace of a counter-example, for which the verification failed, revealed that problems can arise when the cycle counter does not hold the correct value after the recovery

process is completed. Fortunately, a simple solution to elude this issue is to make sure that the cycle counter is overwritten with the correct value during the recovery cycle. This can be achieved by adding an appropriate reset circuitry to the counter register, which is controlled by the recovery signal. After including this improvement into the UPPAAL model, the verification of all TCTL queries terminated successfully.

The last error scenario, which needs to be considered is the possibility of metastability. Very short pulses on the comparator output can put the C-gates into a metastable state, which may break the double-checking mechanism. A solution to this problem is to incorporate a metastability filter in the duplicated C-gates. This will delay any output transition as long as the C-gates have not resolved from a metastable upset [88].

4.2.5 Area & Performance

In comparison to a conventional TMR circuit architecture, two new circuit structures are introduced in our approach: The first is the addition of multiplexers to the replicated target application, as can be seen in Figure 4.2. This requires one extra 2-to-1 multiplexer for every replicated flip-flop. Clearly, the area overhead of these multiplexers and the caused performance degradation depends on the specific circuit. We have therefore applied our technique to a real-life circuit, an embedded 32-bit processor named SCARTS [65]. Detailed area and performance results of this showcase will be presented in Section 4.6.2.

The second extension are the recovery controllers, which are added to the stoppable clock generators of the GALS modules. While the area is constant for most parts of the controller, the required resources of the cycle counter and the delay element in the recovery controller depend on the specific implementation. In Table 4.2 we give an area overview in terms of needed transistors, assuming a 4-bit wide cycle counter and a delay element, which is built with an inverter chain of 20 inverters. For the flip-flops we assume a master-slave implementation with 26 transistors. For the C-gates the weak-feedback implementation by Alain Martin [66] with 8 transistors is considered. In total this adds up to 292 transistors for the robust version of the recovery controller. This is a very decent footprint, even though the size of the recovery controller increased quite significantly due to the duplication, which was needed to meet the high robustness requirements.

Concerning the performance of the proposed method, we have to analyse fault-free executions and scenarios, when a soft-error occurs. In the first case, the execution time is simply extended by the periodic recovery cycles. Depending on the chosen frequency of these recovery cycles, the performance hit can be more or less significant. Considering the above example of a 4-bit wide cycle counter, a recovery cycle is executed every 16th cycle. The performance degradation in this case is less than 10%. If a soft-error occurs, the execution behaviour depends on the hit location. While a hit within the triplicated target application causes no additional delay, an SET within the recovery controller can prolong the recovery process. If one of the duplicated C-gates is hit, e.g., the double-checking C-gates will stop the execution until the transient fault has vanished. Since transient faults typically last for a very short time, the inflicted delay does not constitute a notable performance loss. A longer stall of multiple clock cycles can, however, be caused if the cycle counter is perturbated. In the worst-case a faulty replica can skip an upcoming recovery cycle when the cycle counter jumps over the zero value. Then the two correct

Table 4.2: Area in terms of transistors.

Block	Gates	Count	Total transistors
Cycle counter	Flip-flop	4	104
	XOR2	3	36
	OR2	2	12
	MUX21	1	12
Comparator	NOR4	1	8
Recovery controller	CGATE2	6	48
	GGATE3	2	20
	XNOR2	1	12
	Delay (inverters)	20	40
			292

replicas will be blocked for a whole computation period until the faulty cycle counter reaches the value of zero again. Fortunately, the probability of upsets in a 4-bit wide counter, e.g., is very low and will therefore not have a significant impact on the execution time of the circuit.

4.2.6 Proof of Concept

To test a full implementation of our approach we have designed a small GALS module, containing just an auto-incrementing counter. We then triplicated all combinational and sequential circuit structures, extended them with voters and multiplexers for recovery, and added the recovery controllers. This is a simple yet adequate proof of concept that allowed us to verify the implementation of the clock generator and the behaviour of the recovery mechanism in the presence of faults. Figure 4.8 shows a gate-level simulation of the complete system. On top the three clock signals can be seen. We deliberately chose three different clock frequencies to demonstrate that the replicas work independently from each other. For a real-world application the clock generators would obviously be adjusted to the same frequency. The next three signals displayed on the waveform are the outputs of the cycle counters. It can be seen that every counter is incremented by one at the active edge of the corresponding clock signal. When the cycle counter reaches the value zero, the clock generator is stopped and the local *recover* signal is raised. As can be seen in Figure 4.8, the third replica with the shortest clock period is the first to stop and waits for the other two copies to catch up. For the demonstration of the recovery mechanism we injected a transient fault into an internal signal of the third replica to provoke an SEU in the counter register. As a consequence the counter output jumps to the value 133 instead of 5 at the beginning of the sixth clock cycle. For this demo configuration we chose a recovery action to be scheduled every eighth clock cycle. When all replicas have stopped and the recovery controllers have raised the *recover* signals, the result of the majority voters is written back to each of the registers and the faulty value is corrected, as can be seen in the waveform. At the end of the recovery cycle the counters of all three replicas store the same value again. Finally all clock generators are re-enabled and the replicas continue their computations independently for the next 7 clock cycles.

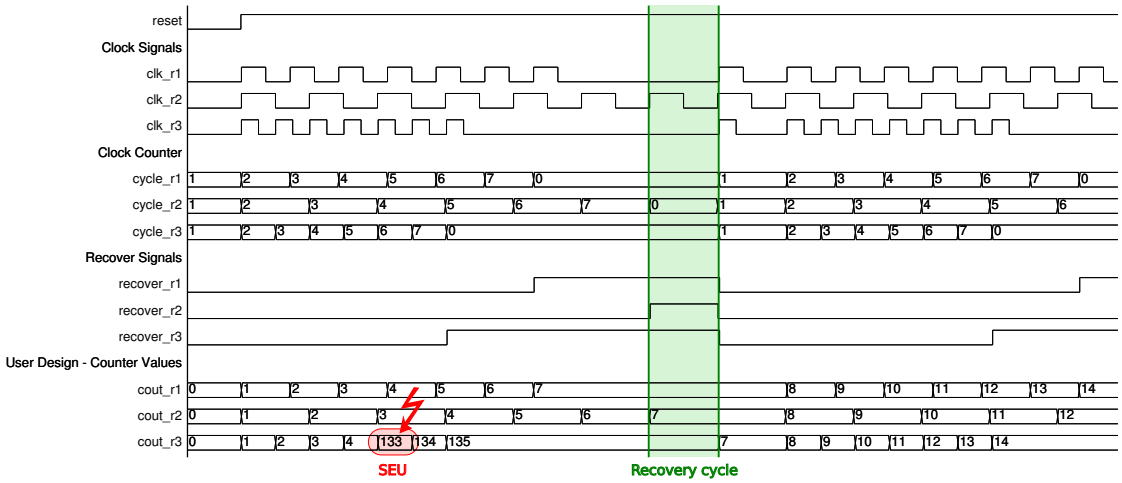


Figure 4.8: Simulation: Recovery from an SEU.

4.3 Approach II: Serial Recovery

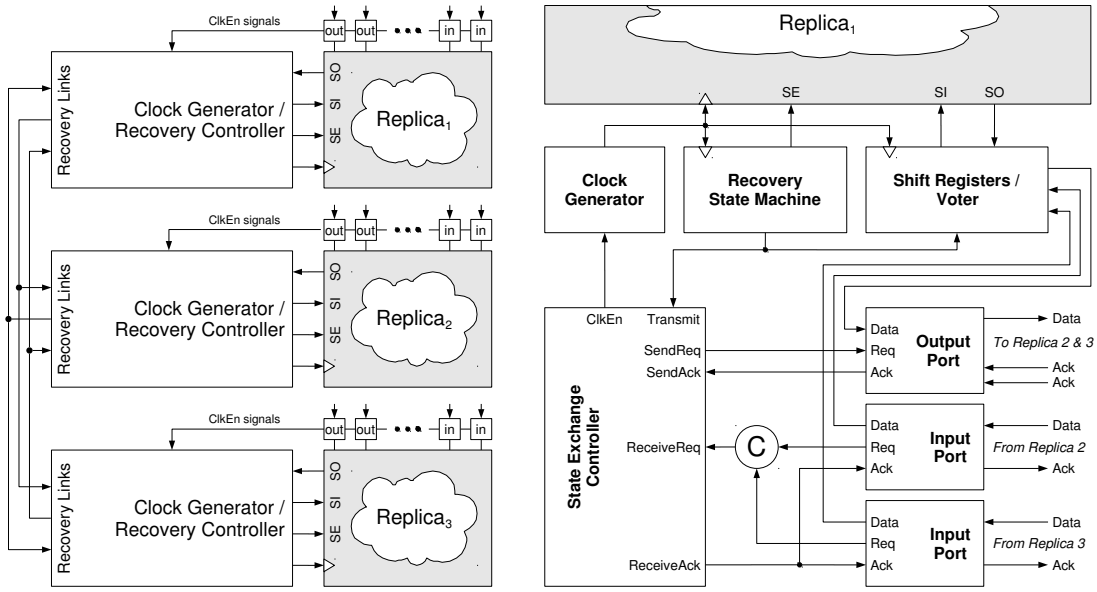
In the last section we demonstrated a simple mechanism for implementing modular redundant GALS modules in a way that clock domains are decoupled and voting and recovery can still be performed at certain checkpoints. The proposed solution cautiously avoids any single point of failure by replicating clock generators and clock distribution networks.

However, the parallel recovery approach does not solve the strong physical dependency between replicated circuit structures, which we criticised in Section 4.1. The interconnect network required for voting, still limits the degree of freedom for placement and layout of module copies. In order to avoid these restrictions and reduce physical coupling to its minimum, we will therefore introduce another solution to perform state restoration. Let us first investigate the three basic ingredients any recovery mechanism in a modular redundant system needs to have:

1. Read/write access on the state-holding elements of all replicated modules.
2. A communication mechanism to exchange state information among the replicas.
3. Majority voting circuits to mask faults in the exchanged state.

In the previous approach read/write access is gained directly by tapping the outputs of flip-flops and inserting a feedback signal to their inputs. The communication mechanism is implemented by wires for sharing the tapped flip-flop outputs among all replicas. These wires connect to voter components, which are added for every flip-flop. Thus, the three basic parts of the recovery mechanism are directly embedded into the replicated circuit.

Another method to get read/write access on the state of a circuit is to use register scan chains. In synchronous circuits scan chains are a well-known and widely-used method for implementing design for test features, which are a necessity for fabrication tests. Re-using scan chains for state recovery in a modular redundant system is therefore a very elegant and efficient approach since



(a) Top-level view.

(b) Clock generator and recovery controller.

Figure 4.9: GALS TMR system.

it helps to avoid new overheads for sake of fault-tolerance alone. The use of scan chains in a TMR system has been proposed by Ebrahimi et al. [26, 27].

Based on this technique it is now possible to build a recovery mechanism for GALS modules that is non-intrusive and preserves physical independence of replicated components. The basic concept is to read out the internal state from the circuit's scan chain, distribute this data among all replicas, perform voting and shift the data back into the scan chain to scrub erroneous bits. As the recovery is done in several cycles bit for bit, we refer to this approach as *serial recovery*. Figure 4.9a again shows the top-level structure of a triplicated GALS module. As can be seen, the basic arrangement looks a lot like in the approach presented in the last section. However, instead of a single recovery signal that controls many internal multiplexers, the recovery controller is connected to the ports of the module's scan chain, *scan enable*, *scan input* and *scan output*. Note that the recovery controller could be interfaced with an arbitrary number of scan chains. To keep the circuit presentations simple, however, we will assume that the module just implements a single scan chain, which contains all the internal registers. On the left-hand side of the controller a network of communication channels, we call them *recovery links*, can be seen. In contrast to the parallel approach, where the recovery controllers just communicated over a set of control signals, the recovery links are fully-fledged data channels, dedicated for the exchange of the state information, which is read out from the scan chains. With respect to the basic concept it does not really matter how these recovery links are implemented. Nevertheless, we propose the use of asynchronous, preferably delay-insensitive, channels. Thus, global timing assumptions are avoided and full timing independence among the replicated module copies is preserved.

4.3.1 Recovery Controller

A closer view on the internal structure of the recovery controller can be seen in Figure 4.9b. The controller basically consists of four core components: 1) the *recovery state machine*, which controls the whole recovery process 2) a set of shift registers together with the voting circuitry, 3) the *state exchange controller*, responsible for exchanging state data, 4) input and output ports, which are connected to the asynchronous recovery links between the replicas. The first two components are synchronous circuits, whereas the state exchange controller and the I/O ports are implemented as asynchronous components. The reason for this partitioning will become evident when we take a closer look on the execution of the recovery process, which alternates between *scan* and *communication* phases. As the state of a typical GALS module would most certainly be too large to be transferred to the other module copies in one I/O operation, it needs to be scanned out and transmitted in several blocks, one after another. Let us assume that the replicated modules are located on three different chips for maximum independence in the fault-tolerant system. The bus width of the off-chip recovery links is therefore limited, which obviously restricts the number of bits that can be transferred at once.

Figure 4.10 shows the state chart of the (synchronous) recovery state machine. As can be seen, the state machine only consists of three states: *Compute*, *Scan* and *Exchange*. The compute state is the idle state, where the state machine rests when no recovery is performed. In this state the only action is to increment a cycle counter with every local clock tick. Note that we use the same counter-based mechanism to schedule periodic checkpoints like in the parallel approach (cf. Section 4.2.1). Thus, the compute state is left when the cycle counter reaches a predetermined value and switches to the scan state. This state transition marks the beginning of the recovery process. In scan state the *scan enable* signal is activated and the read-out of the circuit state begins. Figure 4.11 shows three shift registers that are connected to the output of the scan chain. The width of these registers needs to match exactly to the bus width W of the asynchronous recovery links. The state machine thus stays in the scan state for W cycles to completely fill the shift registers and then changes to the third state, i.e., the exchange state. This is a very interesting state transition as now the control over the recovery process is handed over from the synchronous state machine to the asynchronous state exchange controller. As can be seen in the state chart, the *transmit* signal is activated, which notifies the state exchange controller to begin with its operation. At this point in time the controller (see STG specification in Figure 4.12a) disables the clock generator and thus all synchronous components of the GALS module and the synchronous parts of the recovery controller come to a full stop.

Now the state exchange phase begins and the controller raises the *send request* signal to trigger the output port, as can be seen in Figure 4.9b. The output port then reads the data block currently stored in the shift registers and encodes it with a delay-insensitive (DI) code. As mentioned before, we use a DI transmission scheme to avoid timing assumptions for the recovery links. For our prototype system we employed a 3-of-6 code with a 4-phase handshake protocol, because the resulting links are very resource-efficient. In case low power consumption is a criterion, one might rather choose a 2-phase 2-of-7 code, like in the *SpiNNaker* project [99].

In parallel with the outgoing transmission the other two module copies, having switched to recovery mode as well, will transmit the data they have extracted from their scan chains. Two local input ports (cf. Figure 4.9b) will receive the incoming data blocks and inform the transmit

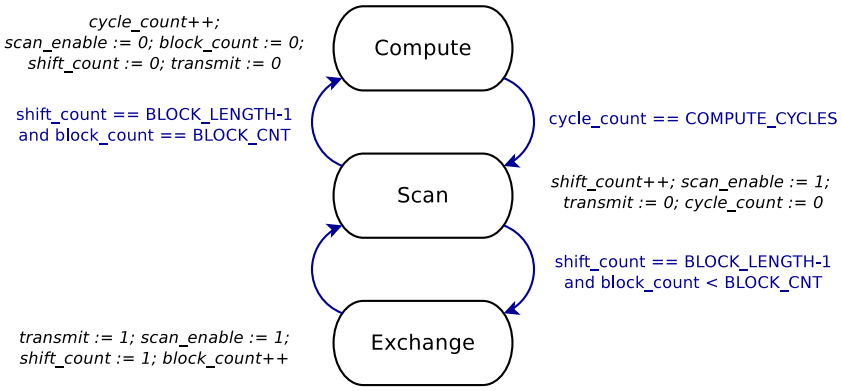


Figure 4.10: Recovery state machine.

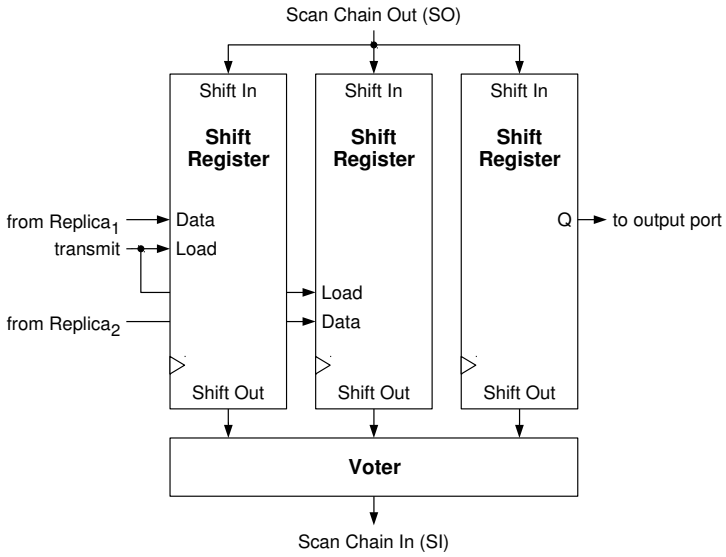


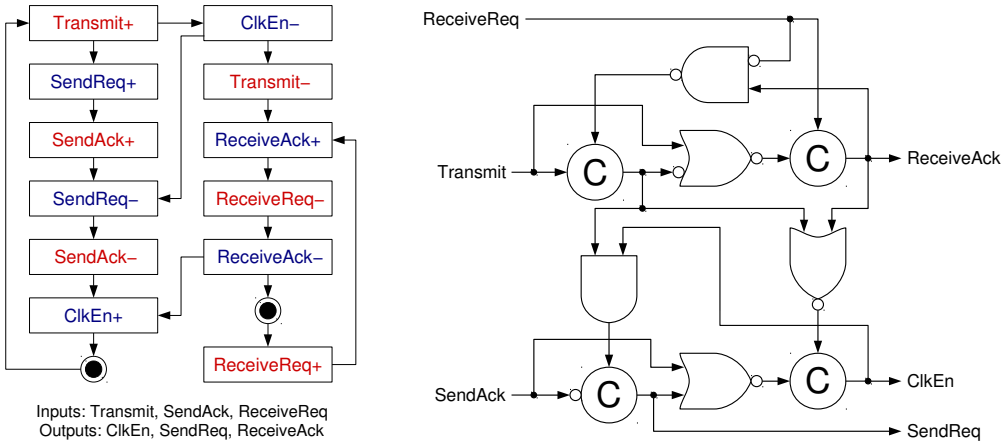
Figure 4.11: Transmit/receive shift registers with voter.

controller once these have been captured and fully decoded. When output and input operations have been completed, the state exchange controller re-enables the clock generator and returns control to the synchronous recovery state machine.

The data received from the two module copies is then loaded into the shift registers (parallel load, see Figure 4.11) and the recovery state machine changes back to scan the state[2]. At this point in time one of the three shift registers contains the data block extracted from the local scan chain and the other two registers have stored the received data. Bit for bit majority voting can now be performed for the contents of the shift registers and the resulting (correct) value is shifted

[2]Note that the arc in the state chart of the transition from transmit to scan state is not annotated with a condition. Thus, the transition is immediately taken with the first clock tick, once the clock generator gets re-enabled.

back into the module's scan chain. At the same time the next data block is moved from scan chain output into the shift registers. After W cycles the state machine enters the exchange state again. This routine is repeated until the all bits of the scan chain have been read out and have been overwritten with voted values. Then the recovery state machine jumps back to the compute state and the recovery process is finished.



(a) Signal transition graph.

(b) Synthesised circuit.

Figure 4.12: Specification and implementation of the State Exchange Controller.

The state exchange controller was again specified with an STG and synthesised with *Petrify*, as can be seen in Figure 4.12b. The left column of the STG shows the 4-phase handshake protocol performed with the output port (transmitter), the right column incorporates the receive part with input ports (receiver)[3]. The STG specification is quite straightforward and follows the descriptions of the recovery process above. There are, nevertheless, two details that require closer examination: 1) The *Transmit* signal, which is issued by the synchronous recovery state machine to trigger the state exchange process, is expected to be reset again as soon as the clock generator is disabled (*ClkEn-*). In our implementation this behaviour is simply ensured by connecting the *ClkEn* signal to the active-low asynchronous clear input of the flip-flop that drives the *Transmit* signal. At this point we also want to stress that the *Transmit* signal should be registered in order to filter potential glitches during state transitions. 2) The sequence of events for the receiver handshake is a bit more intricate than can be seen in the STG since the received state information needs to be loaded into the shift registers before the clock can be re-enabled. Depending on the implementation of these shift registers, a parallel load can either be performed synchronously or asynchronously. In our implementation we chose the former approach and generate an extra clock pulse just for loading the shift registers. This clock pulse can be directly derived from the *ReceiveAck* signal. As can be seen in the STG, this signal produces a positive pulse as part of the

[3]This STG and consequently the synthesised circuit are different from what we published in [64]. In the published paper we only considered transient faults, whereas for this thesis we also investigated long fault durations, i.e., permanent defects (see next subsection). For these defects, the controller implementation had to be revised as the original STG included some timing assumptions that are violated when certain signals are permanently stuck.

4-phase handshake cycle at the right point in time, i.e., *after* the state words have been received by the input ports and *before* the clock is re-activated. When data is transferred like this from asynchronous circuit parts into a synchronous module, clearly some timing constraints have to be satisfied.

4.3.2 Timing Constraints

- Shift registers – parallel load: As explained above, data from other replicas are loaded synchronously into the shift registers during state exchange. Since the corresponding clock edge is derived from the *ReceiveAck* signal, it is necessary to delay the rising edge of this signal until the received data are stable at the inputs of the shift registers. Furthermore, it needs to be ensured that the shift registers have changed from shift mode to parallel load.

  A more subtle timing constraint is caused by the two clock sources of the shift registers: i) the clock generator, and ii) the *ReceiveAck* signal. Both sources are combined by an OR-gate to produce a single clock signal for the shift registers. Consequently, it is necessary to ensure that the rising transition of the *ReceiveAck* signal occurs at least one nominal clock period after the last regular rising clock edge of the clock generator.

- Shift registers – shift operation: When the recovery process changes back to scan operation after the state exchange, the clock generator must not be re-enabled too fast. The first clock edge should occur with a minimum delay of one clock period after the last rising clock edge, which was generated by the *ReceiveAck* signal. Furthermore, it needs to be ensured that the shift registers have changed back from parallel load to shift mode.

- Clock to clock enable: When the recovery state machine changes from scan state to exchange state, the state exchange controller needs to disable the clock generator before another clock edge can be issued. As explained for the parallel recovery approach, this is a critical timing path, since it involves the clock tree, which can have a significant latency for more complex GALS modules. A solution to bypass this issue might be to decouple the disabling of the clock generator and the enabling of the transmission process. Then the recovery state machine could activate the control signal for disabling the clock generator a certain number of clock cycles early before the end of the scan phase and thereby compensate for the latency of the clock tree. The transmission process, on the other hand, is only started after the last bit of state information has been retrieved from the scan chain.

4.3.3 Robustness of the Recovery Controller

The recovery controllers of such a replicated system obviously are very critical components, as a transient fault in one of these controllers cannot only cause the recovery of a local component to fail but might also drive the system into an inconsistent state or a deadlock. Assume, e.g., that a message sent over a recovery link is lost due to a soft-error. This clearly leads to a deadlock since the receiver waits for the message to arrive and the sender waits for the acknowledgement of the receiver. Another scenario, leading to an inconsistency or a deadlock, is the corruption of the recovery state machine's state vector, either during computation or recovery phase.

In many known approaches with active redundancy and roll-forward recovery (see [27,123]), the recovery controller therefore is required to be fault-tolerant itself. In case of our approach the control logic, which can be found in the recovery state machine, the transmit controller and the I/O ports, needs to be protected against faults. This could be done with classical replication-based fault tolerance mechanisms at gate or circuit level. Since the control logic of our recovery controller is quite small (see Section 4.3.6), the costs of replicating these circuits might still be acceptable. We, however, want to present a different approach. Instead of preventing faults at hardware level, we devised a mechanism that guarantees correct behaviour on system level, despite of local faults in a single recovery controller. The basic concept is to design the recovery controllers such that a faulty controller can be re-integrated by the other two correct controllers when the next recovery action is performed. The recovery thus serves not only for restoration of the state of a faulty module, but also for readjustment of the respective recovery controller.

The key observation for the design of such a mechanism is that an inconsistent, faulty controller will freeze the recovery process, either because it does not change to recovery mode when the other correct controllers do, or because it fails to send all required messages during recovery. Recall that the clock generator is disabled when communication is performed over the asynchronous recovery links. Thus, either the module copy with the faulty controller, the two correct and consistent copies, or all three copies will be stalled. To resolve this situation we have extended our recovery controller implementation with two watchdog timers – a fast and a slow one. The precise meaning of fast and slow will be discussed later. Right now, just assume that the slow watchdog has a significantly longer timeout than the fast one. Basically both timers check for inactivity during the recovery process and their timeout signals are connected to the recovery state machine, which has been extended to deal with situations when the recovery fails.

The slow watchdog counter is activated when the local module copy decides to broadcast state information, i.e., during the transmit phase of the recovery process, when the local clock generator is disabled by the state exchange controller. Now let us assume that the handshake-based communication with the other module copies cannot be finished before the timeout expires. Then the slow watchdog asserts its timeout signal and the following "emergency protocol" is executed: 1) The local clock generator is re-enabled, overriding the decision of the state exchange controller to stop the clock. 2) The recovery state machine awakes and aborts the recovery process as a reaction to the raised watchdog timeout. The data that has been read out of the module's scan chain is shifted back, without any voting and any further attempts to exchange state information with the other two replicas.

To explain why a second watchdog timer is needed let us explore what could happen, if the recovery controller was only equipped with the slow watchdog mechanism. Assume we are dealing with a TMR system, where a recovery is periodically executed with p clock cycles of computation time between two successive recovery processes. For a reasonably fast deadlock resolution the watchdog has to be configured to a timeout t that is much smaller than this computation time, i.e., $t \ll p$. Thus, let us assume that $t < p/2$. Figure 4.13 shows a problematic execution, where a fault resets the state of the recovery controller of replica R_1, exactly $p/2$ clock cycles after a recovery has been successfully performed. The recovery state machine therefore returns to the old state it had right after the end of the last recovery ($E(CP_1)$). From there on it lags half the number of compute cycles behind the non-faulty controllers of R_2 and

R_3, which therefore reach the next checkpoint CP_2 much earlier. Since $t < p/2$, R_2 and R_3 run into the timeout and abort the recovery *before* R_1 is able to catch up. The same situation is repeated for every future checkpoint, hence the system cannot recover from the fault.

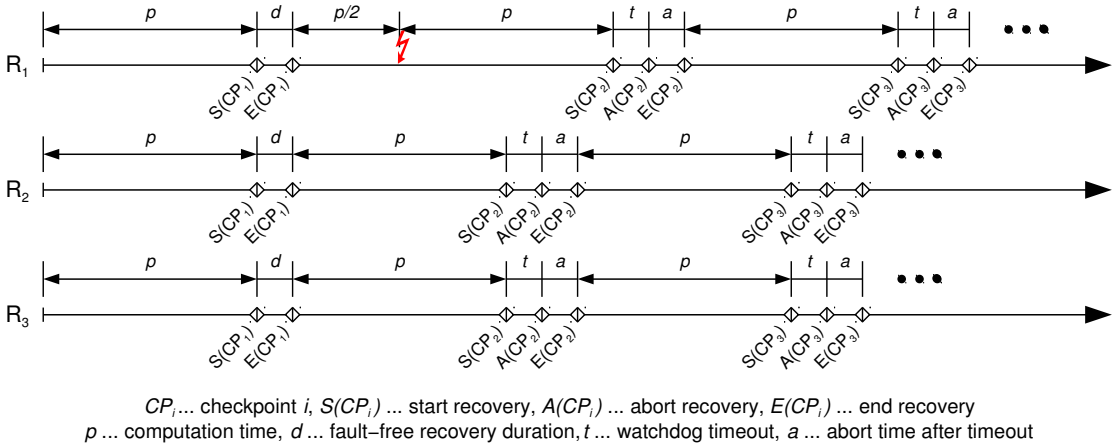


Figure 4.13: Recovery failure with single watchdog.

To resolve this issue, a second shorter timeout is needed that is activated when a module is in *computation mode* and there are active requests on a majority of recovery links. This condition can be easily evaluated by monitoring the outgoing and incoming recovery link signals. In case of TMR, if there are active requests on both incoming links and the outgoing link is idle, the fast watchdog counter should be activated. For sake of simplicity we used two separate watchdog modules in our implementation – one preset to a long, the other one to a short timeout. However, since the two counters never operate at the same time, resource sharing is possible and only one watchdog module, dynamically set to the appropriate timeout value, would be sufficient. Another important implementation detail we like to share is that the circuit for enabling the slow watchdog module should be purely combinational and must not depend on any (possibly corrupted) memory element of the local recovery controller. Otherwise a faulty internal state might tamper with the correct detection of the recovery link state and thereby prevent proper activation of the watchdog. Figure 4.14 shows a simplified version of the extended recovery state machine, which includes an additional state for finishing the scan in case an abort is necessary.

Following this mechanism it can be guaranteed that, 1) correct modules remain correct and consistent, if the recovery fails and has to be aborted, and 2) a faulty component is re-integrated when the next recovery action is executed. An essential requirement for the design of the watchdog module, however, is that the timeout values are carefully chosen. Therefore we need to define some upper and lower bounds for the slow and the fast timeout values. Assume we have a modular redundant system with n copies of a specific GALS module. Let s_i be the duration of the slow timeout, for module copy $i, i = 1, \ldots, n$, and f_i be the value for the fast watchdog. Due to considerable variations in modern technologies we need to investigate and constrain the timing in different *PVT* corners. Variables with superscript bc therefore denote a timing value in the best-case corner, whereas superscript wc identifies worst-case values. Now let us assume that

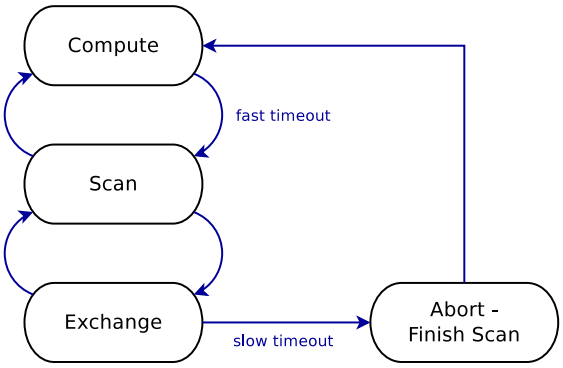


Figure 4.14: Extended recovery state machine.

the module copies after the execution of a recovery process are almost perfectly synchronized, i.e., the copies, when switching from recovery mode to computation mode, initially perform all operations almost simultaneously, with negligible time difference. Over time this tight synchrony will be lost due to PVT variations, which affect the clock generator or the timing of the asynchronous circuits when I/O operations are performed with other modules of the overall GALS system. Thus, the time when the module copies reach the next recovery checkpoint will not be exactly the same, and consequently these differences need to be considered in the timing constraints of the watchdog modules. Now let p_i denote the time that will pass for module copy i between the end of the last and the start of the next recovery process, and let p_i^{wc} and p_i^{bc} be worst-case and best-case values, respectively. Then the following three timing relationships for the slow and fast watchdog timeouts can be determined:

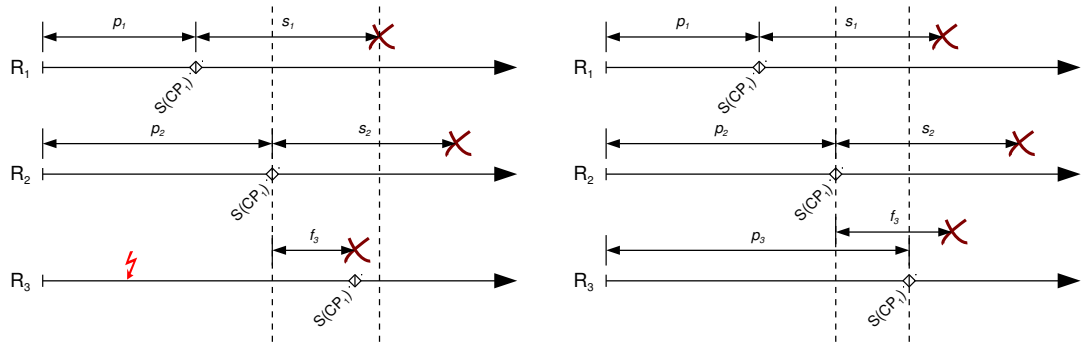
$$s_i^{bc} > \max_{j \neq i}(p_j^{wc}) - p_i^{bc} \tag{4.1}$$

$$f_i^{wc} < \min_{j \neq k \neq i, j \neq i}(p_k^{bc} + s_k^{bc} - p_j^{wc}) \tag{4.2}$$

$$f_i^{bc} > p_i^{wc} - \max_{j \neq i}(p_j^{bc}) \tag{4.3}$$

These constraints need to be satisfied by the watchdogs of every module copy i. Equation 4.1 defines a lower bound for the slow timeout. This bound ensures that the timeout is long enough so that the slowest module copy can finish its regular computations and is able to join the recovery process before the faster copies abort the recovery. The second constraint is an upper bound on the fast watchdog timeout that makes sure that a faulty copy joins the recovery process fast enough, i.e., *before* the slow timeout of first non-faulty module copy runs out. This situation is illustrated in Figure 4.15a, where the non-faulty components R_1 and R_2 are stalled when they start the recovery because R_3 does not participate (as its recovery controller was affected by a fault). According to the procedure described above the fast watchdog is activated in R_3, when an incoming recovery request is pending from both R_1 and R_2 (first dashed line). Now the watchdog needs to force the faulty module R_3 into recovery mode, before the slow watchdog of R_1

causes an abort (second dashed line). Finally, Equation 4.3 describes a lower bound for the fast timeout, which is needed to guarantee that the slowest but non-faulty module copy can finish its computations before joining the recovery process. Figure 4.15b shows that the fast watchdog of R_3 is activated when R_1 and R_2 start the recovery (first dashed line). Consequently, f_3 needs the be long enough so that R_3 can finish its current computation round of length p_3 (second dashed line). Note that Equation 4.3 is only valid for TMR systems. The fast timeout counter is activated when a *majority* of module copies have started the recovery process. In TMR this is equal to the situation where the penultimate copy enters recovery mode (as depicted in Figure 4.15b). In general modular redundant systems with $n > 3$, the majority can be reached before the penultimate component. To fix Equation 4.3 it is therefore advisable to replace the $\max$ with a $\min$ function. Even though the resulting bound is not tight, as it assumes that the fast watchdog starts counting with the *first* module that enters recovery mode, it is still a valid bound and a reasonable approximation. Similarly, the Equation 4.2 is an overly conservative upper bound in the case of general modular redundancy. The bound is computed as if the penultimate module copy that enters the recovery activates the fast watchdog counter (note that the term on the right hand-side is minimised when p_j^{wc} is maximised). Nevertheless, the bound is valid even in the general case with more than 3 copies.



(a) Upper bound.

(b) Lower bound.

Figure 4.15: Fast timeout constraints.

An implementation of the watchdog counter is shown in Figure 4.16. Since the counter needs to be operational when the local clock generator is turned off, it is equipped with an own ring oscillator. When the *enable* signal is activated, the oscillator generates clock ticks for a counter unit, which can be efficiently built with a *linear feedback shift register* (LFSR) [5]. A comparator unit evaluates the counter outputs, and raises the timeout signal, when the maximum value of the count sequence is reached.

4.3.4 A Short Note on Long Faults (Permanent Defects)

So far we have only discussed how transient faults and soft-errors can be mitigated in a replicated GALS module (core logic and recovery controller). In this section we also want to show that this mechanism, with some small modifications, is able to provide protection against per-

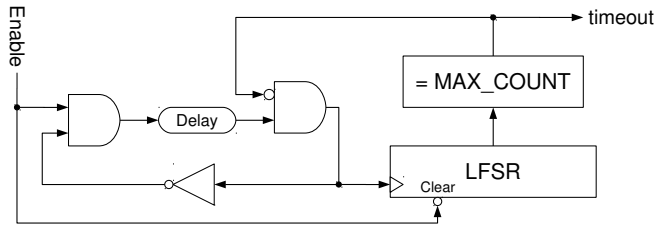


Figure 4.16: Watchdog module.

manent defects. Clearly, modular redundant systems can mask permanent defects in a minority of components, as long the other replicas remain functional and produce correct outputs. It is, however, important that the replicated circuits are truly independent and that a defect in one copy cannot affect the other healthy copies. In case of our replication mechanism and recovery scheme for GALS modules, we therefore we need to look into two cases, depending on where a defect occurs: 1) Defects in core logic of the module (combinational or sequential parts). In this case the state of the affected module copy might be corrupted beyond recovery, or might be compromised again immediately after a performed recovery. Consequently, all outputs produced in the presence of this defect might be erroneous. Nevertheless, assuming that the remaining module copies are not affected by faults, correct system outputs could be produced by subjecting the outputs of all replicated modules to a majority vote[4]. 2) A defect can occur in the asynchronous wrapper, either in the clock generator or in the recovery controller. In both cases this might prevent a module copy from participating correctly in the recovery process, which leads to a deadlock that stops the entire system. This situation is similar to what we have discussed for soft-errors in the previous section. Hence, the watchdog timers and the abort mechanism again can be used to ensure that a broken replica cannot stall the entire modular redundant system. The only difference, however, is that a permanent defect, unlike soft-errors, cannot be recovered.

Assume, e.g., a permanent defect in the output port of one recovery controller in a TMR system, which hinders the affected module copy to successfully broadcast its state information to the other replicas. Consequently, the recovery process will be aborted every time it is started. In this situation the recovery therefore becomes useless and only reduces system throughput. It is therefore advisable to cancel all future checkpoints until the defective module copy has been replaced by maintenance. Figure 4.17 shows an adapted recovery state machine to deal with such permanent failure scenarios. The basic concept is to change to a *degraded configuration*, if a permanent defect has been detected. For this detection the state machine counts the number of successive recovery processes that have failed. If this count surpasses a certain value, the state machine will not return to the standard *compute* state but but to a *degraded compute* state. The clock generator is re-activated and the module then continues with its ordinary computations without being interrupted for recovery checkpoints anymore. This system state prolongs until the defective unit is replaced by maintenance. Upon some external trigger, indicating that repair has been carried out, the state machines in all module copies make a transition back to the

[4]Note that an output of a defective module might also get lost, if the asynchronous output handshake is not initiated. We will deal with voting in this situation in Section 4.4.3.

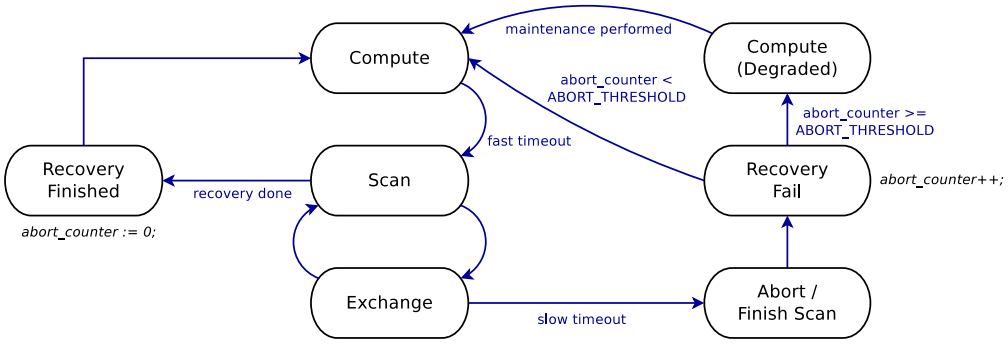


Figure 4.17: Recovery state machine dealing with defects.

ordinary *compute* state. Thereby recovery checkpoints can be executed again, and the state of newly replaced component will be synchronized with the current state of the other correct replicas. Thus our recovery mechanism does not only help to mitigate soft-errors but also allows re-integration of hot-swapped units after permanent defects. This feature is of utmost importance for systems with high availability requirements. Consider, e.g., control circuits of a power plant that need to remain operational even during maintenance procedures.

Note that the condition of the state machine for entering the degraded state only covers defects in the recovery controllers. Obviously, a mechanism to detect permanent faults in the rest of the replicated GALS modules is necessary as well. A simple approach would be to monitor the values produced at output ports. If outputs of one copy repeatedly deviate from the correct results of the other replicas, the system should also change into the degraded computation state.

Another detail we should mention about the presented extension of the recovery state machine is that the degraded state needs to be implemented very carefully. As we have explained, no recovery processes are performed in this state and the only way to end it is an externally triggered input by maintenance staff. Clearly, one does not want that the state machine gets stuck in this state as a result of a *single transient* fault. This problem can be avoided by three counter-measures: 1) Choosing a state encoding where the state vector of the degraded state has a Hamming distance greater than or equal to 2 with respect to all other state vectors. This prevents that a single upset in the state register itself steers the state machine into the degraded state. 2) Make sure that the next-state logic cannot produce the vector of the degraded state in case of an *SET*. 3) Protect state registers that are evaluated by the next-state logic to transition into the degraded state. In the state machine presented in Figure 4.17 the abort counter, if changed to the threshold value by an upset, would lead to a transition into the degraded state. For our prototype we have therefore chosen to encode the counter value in a unary code, also known as thermometer code, where the number of bits set to one represents the value. This way a single upset in the counter register becomes uncritical.

4.3.5 Verification

For functional verification of our approach we have built a simple prototype system consisting of two GALS modules, M_1 and M_2, both triplicated and extended with the recovery controller

presented above. In order to test the recovery mechanism, the functionality of the modules can be kept very simple. Both modules therefore only implement a 9-bit wide counter, which is incremented every clock cycle until a predetermined final value is reached. At a certain point in the count sequence, the counter values are exchanged among the modules to be able to test and verify inter-module data transfers over asynchronous channels.

Regarding the recovery process, we have integrated the counter registers into a scan chain, which consists of 32 scan flip-flops in total. The recovery controllers of both modules are configured to trigger a recovery action every 64 cycles. Note that is a very short recovery interval, which we have deliberately selected for the purpose of functional verification and with the aim to keep simulation times as small as possible. For the recovery links we have chosen a bus width of 8 bit. Consequently, it takes 4 transmission rounds to exchange all the contents of the scan chains among the module copies during the recovery process.

We have synthesised this GALS system with a standard cell library for a 90 nm process. Based on the results of a (pre-layout) static timing analysis, we adjusted the clock generators of M_1 and M_2 to a frequency of 1 GHz and 782 MHz, respectively. The motivation for using different clock periods was to examine communication across two independent timing regions. Based on the synthesised netlist and (pre-layout) timing annotations, we then simulated numerous test scenarios to check that the complete design works as intended.

Furthermore, we also wanted to systematically test the resilience of all parts of the design against transient and permanent faults, and therefore exhaustive fault injection (FI) experiments have been conducted. The experimental framework builds on a simulation-based fault injection approach [127], where faults can be injected on nets and cells of the simulated netlist with the help of simulator commands. The target of our fault injections was one copy of module M_1, which included the entire core logic, the I/O ports, all components of the recovery controller and even the clock generator. The resulting target set consisted of 950 nets, 192 flip-flops and latches, and 27 C-elements. The transient faults we injected had a length of 1 ns, uniformly distributed in steps of 0.5 ns over one full computation round, i.e., 64 clock cycles, and the following recovery action. Both positive and negative pulses were injected. At the end of the simulation the state of M_1 and M_2, including all copies, was compared with the expected values of a fault-free simulation. The exact same set of experiments was then performed with permanent defects, the only difference being that the fault effect was not revoked after one 1 ns but the signal value remained stuck at a constant value until the end of the simulation. Furthermore in the evaluation of the simulation runs we excluded the defective module copy and only checked that the state was consistent and correct for the other replicas.

In total we executed approx. 1.9 million FI campaigns (one half for transient, the other half for permanent faults, one fault per simulation run). Due to this massive number of simulations we distributed the workload on nine workstations (equipped with quad-core Intel® Xeon™ processors clocked at 3 GHz, 8 GB memory), each of them running two simulations in parallel. With this configuration we were able to carry out all simulations in 16 hours and 20 minutes.

These experiments gave us important feedback on the resilience of the tested circuits and the first runs actually uncovered several issues we had not thought of during the design and the implementation of the recovery mechanism. Most things were minor implementation bugs or problems with the proper state encoding of the recovery FSM. One particular set of injected

(permanent) faults, however, revealed a critical issue with the communication over recovery links. It turned out that a component with *Byzantine* fault behaviour is able to bring down the recovery mechanism. Recall that a Byzantine fault in one component is observed differently by the other components of the system. In other words non-faulty units have an inconsistent view upon the faulty component [55]. In a modular redundant system this can ultimately drive the non-faulty components into an inconsistent state, a potentially leading to system failure since now a majority of replicas is inconsistent or faulty.

Figure 4.18 shows exactly such a Byzantine fault scenario. The waveform displays data and acknowledge signals of the recovery links, as well as state vectors and timeout signals from the recovery controllers. As can be seen, the system is in recovery mode and state information, retrieved from the modules' scan chains, is exchanged. Now assume that R_3 is faulty and produces a Byzantine output behaviour: On the last communication cycle, which is performed for the current recovery action, the replica R_3 properly executes the full handshake protocol with R_2, but fails to reset the acknowledge signal for R_1 (because of some error in the circuitry of R_3, see flash sign and red line in Figure 4.18). From the viewpoint of R_2 all communication activities have been successfully completed. Thus, it ends the recovery process and changes back to computation mode. For R_3 we assume the same behaviour. Only R_1, which still waits for a falling transition of the acknowledge signal from R_3, is left behind in recovery mode. After some time the watchdog mechanism wakes up R_1 and it finally carries on in computation mode. At this point in time we have the faulty component R_3 in the system, whose state might be completely corrupted, and two correct replicas, R_1 and R_2, which perform the correct computations. However, due to the incurred timeout of R_1, the correct replicas have been set apart in time. The next recovery process will thus be started by R_2 and R_3 and due to the short timeout mechanism, the computation of R_1 is interrupted, as it seems to be the outlier of the system. Thus, a correct component is prematurely forced into the recovery process, which transforms the temporal inconsistency of the two correct replicas into an inconsistency in the value domain. In this situation the attempt to perform recovery with majority voting is likely to fail.

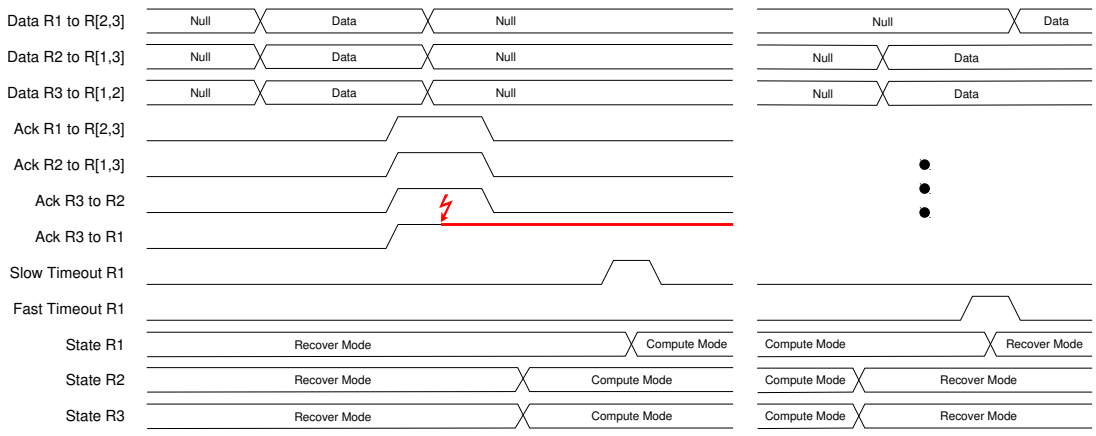


Figure 4.18: Byzantine error on recovery link.

Therefore a solution where non-faulty units can find *consensus* despite of this malicious behaviour would be required. This problem was first formalised as the *Byzantine Generals Problem* [58], where a group of generals of the Byzantine army need to agree on a common plan of action, communicating only over messengers. However, there are a certain number of traitors (faulty units), which try to confuse the loyal generals (non-faulty units) by sending inconsistent messages. In this situation the authors of [58] show that the loyal generals are only guaranteed to come to a common agreement, if *more* than two-thirds of the generals are loyal. In other words, if there are f traitors, the total number of generals has to be at least $3f + 1$. This leads to considerably more expensive system architectures compared with typical modular redundant systems, where a simple majority is sufficient to mask faulty computations by voting.

Instead of increasing redundancy tremendously to be able to perform consensus in the rare event of Byzantine faults, we therefore propose a more practical fault avoidance strategy to minimise the risk that a malicious fault can occur. This strategy can be implemented with some minor changes to the architecture of our recovery links. Figure 4.19 shows the result of this redesign. The essential modification, as can be seen, is that each replica now has only two output signals for the recovery links: the data signal driven by the output port, and a *single* acknowledge signal for the input ports. Note that a Muller C-element has been added to combine individual ack outputs of the input ports into a single acknowledge signal. Due to this single output strategy for the communication signals any fault in a replicated unit will be observed by all other units in the same way.

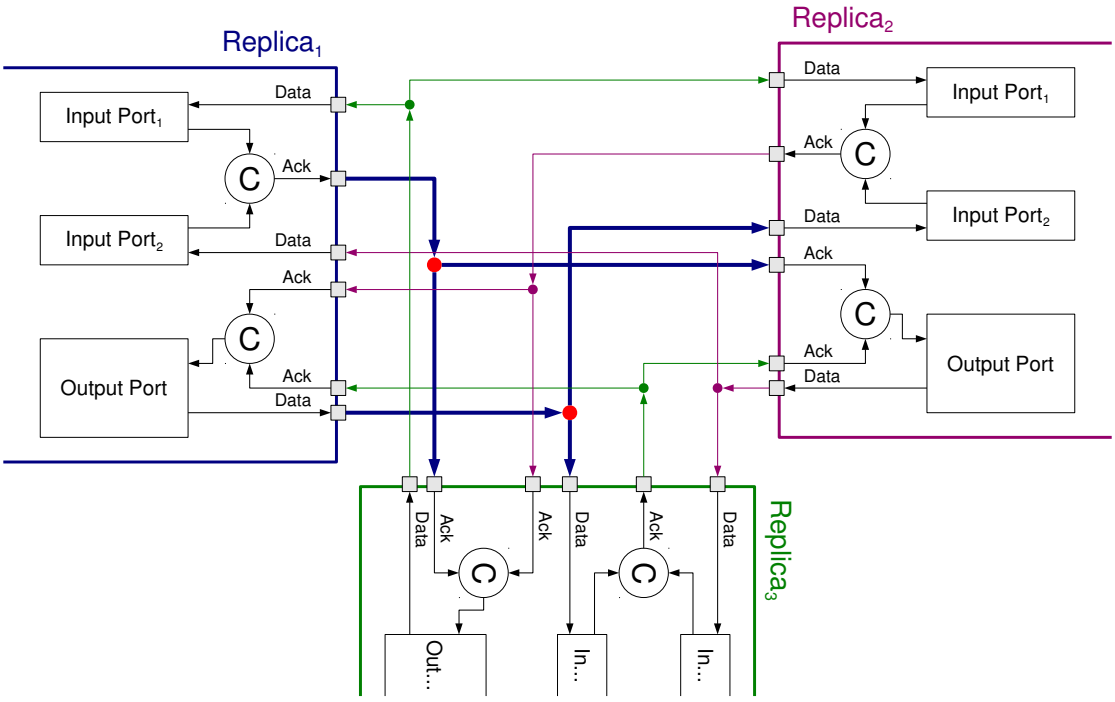


Figure 4.19: Redesign of Recovery Links to reduce risk of Byzantine faults.

In case of permanent faults, however, Byzantine behaviour might still occur. One scenario is a driver defect that results in an intermediate voltage level on the output signals and is then perceived differently by the other replicas. Another critical point are wire forks in the interconnect network, which are now required due to our modification. Consider, e.g., the data and acknowledge signals of Replica_1 (thick, blue wires). As can be seen in Figure 4.19, these signals need to fork into two branches for connecting the other two replicas (this is once again a simple TMR scenario). Clearly, these forks must not exhibit a Byzantine behaviour. Even though such a behaviour in a simple wire fork seems extremely unlikely, considering that a fork is just a passive metal connection, it is not completely impossible. Consider, e.g., a two-sided crack within the fork as illustrated in Figure 4.20. Due to, say temperature changes, the cracks open an close over time so that there is either a connection with the outgoing branch A or with branch B. Such a defective fork might be able to first fool the communication with the unit on one branch, while the handshake protocol can be successfully completed for the second branch. Then the fork changes its behaviour, and the defective branches are swapped. This could lead to a situation where the recovery controllers of all replicated components are forced into different states, and any further attempt to perform recovery results in undefined and therefore potentially harmful behaviour.

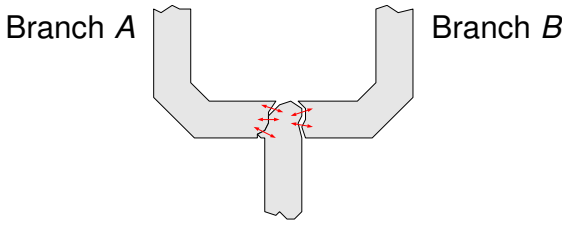


Figure 4.20: Malicious fork.

As said before, such an intricate faulty behaviour is very unlikely to occur. More realistic are faults that affect one specific branch of the fork (e.g., a permanent defect due to an open circuit, or an SET in a buffer caused by radiation). However, one can consider the branches *after* the wire forks to be part of the recipient module. With this abstraction, a fault in one of the branches is no different from an internal fault in the associated module, i.e., it is the faulty component *itself* that observes a faulty input. Note that this situation is different from the Byzantine behaviour described above, where the *faulty* units sends *faulty* messages to the healthy components. In contrast, a defect or transient in a single branch cannot lead to a catastrophic inconsistency among the fault-free module copies.

4.3.6 Area & Performance

The presented approach is very area efficient in comparison to a conventional TMR design. Replicated modules can remain unchanged since no internal voters are required and scan chains can be re-used. Furthermore, the presented design of the recovery controller is very lightweight with a total cell area of 3945.08 μm^2 (an equivalent of 1257 two-input drive-strength-one NAND gates), as can be seen in Table 4.3. Note that the area numbers are taken from the prototype im-

plementation used for the fault injection experiments. The largest components are the recovery state machine and the I/O ports for the recovery links, which contain encoder, decoder and completion detection units for the delay-insensitive 3-of-6 code. Consequently, a key parameter influencing the area complexity of the recovery controller is the bus width of the recovery links.

Table 4.3: Area evaluation – Recovery controller (area units in μm^2).

Component name	Comb. area	Seq. area	Total area	Gate equivalent
Recovery state machine	573.10	373.97	947.07	301.62
Shift registers / voter	48.61	413.95	462.56	147.31
Transmit controller	31.36	21.95	53.31	16.98
Input Port I	450.02	192.86	642.88	204.74
Input Port II	450.02	192.86	642.88	204.74
Output Port	322.22	174.83	497.06	158.30
Watchdog Timer Fast	109.76	62.72	172.48	54.93
Watchdog Timer Slow	307.33	62.72	370.05	117.85
Misc. circuits	141.12	15.68	156.80	49.94
Total	2433.53	1511.54	3945.08	1256.39

Regarding performance, replicated modules can be operated with the same maximum frequencies like in a non-TMR system. There is, however, a performance overhead imposed for the overall system since the computation phases are interrupted for executing the recovery process. Clearly, the size of this overhead depends on the frequency and the duration of the recovery process. The approx. recovery duration can be computed by adding the length of required scan cycles with the overall communication latency of the recovery links:

$$d_{recovery} \approx T_{clk} \cdot L_{sc} + L_{link} \cdot N_{blocks}, \tag{4.4}$$

where T_{clk} is the clock period, L_{sc} is the length of the scan chain, L_{link} is the full communication latency of one transmission over the recovery links, and N_{blocks} is the number of transmitted blocks, i.e., L_{sc} divided by the bus width of the recovery links. Assuming, e.g., a module where 2000 registers have to be recovered with a clock period of 1 ns, and 8-bit wide recovery links with a latency of 5 ns, the duration of the recovery process would be 3.25 μs.

Like for the parallel modular redundancy approach we have also applied our scan chain-based state restoration scheme to a real-life circuit design, namely the SCARTS processor. Detailed area and performance results for this showcase design, as well as a comparison between different TMR architectures will be presented in Section 4.6.2.

4.4 System Architecture

So far we have only discussed the replication and recovery mechanism of individual GALS modules, without considering the architecture of an entire GALS system, which typically contains several modules. Figure 4.21 shows a mock-up model that illustrates several key system-level design issues. The figure shows three different chip designs: 1) D_A, the big yellow box on the

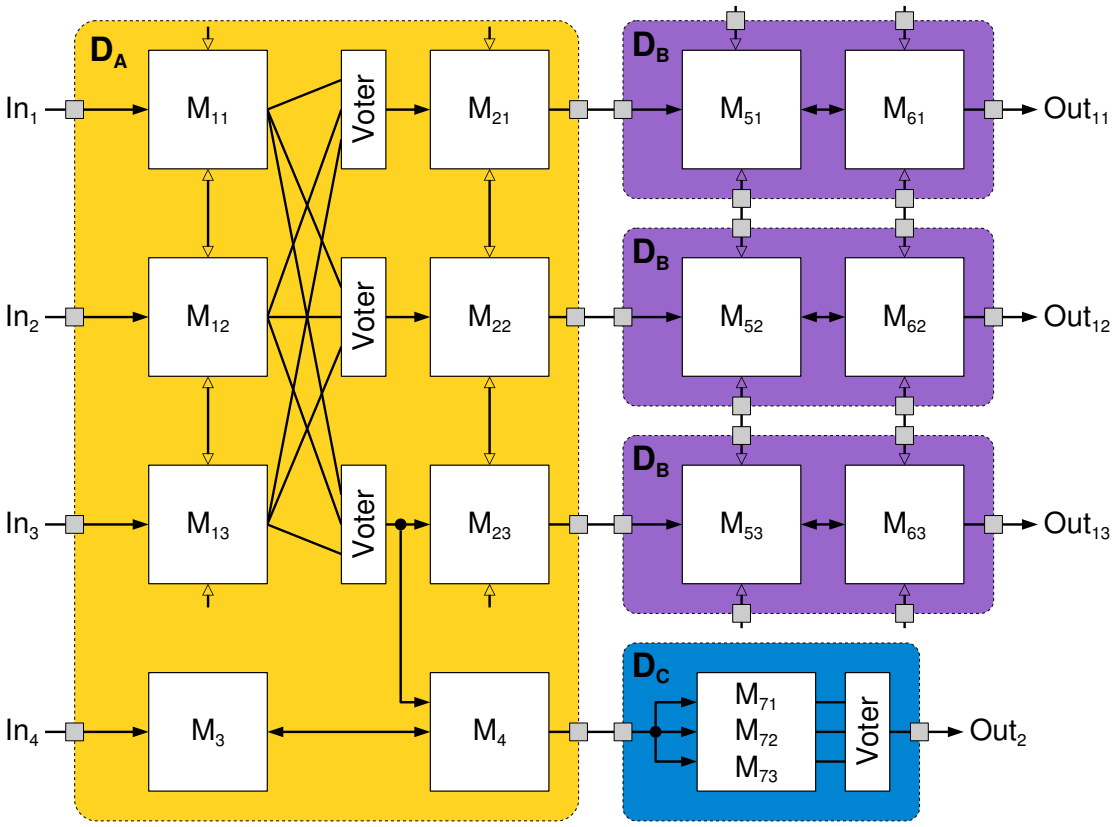


Figure 4.21: Illustrative example of system architecture.

left-hand side, 2) three instances of D_B (violet boxes on the right), and D_C, the blue box in the bottom-right corner. Hence, the system consists of a total of five chips. Input and output pads are depicted as little grey boxes, which sit on the dashed outlines. Within each chip GALS modules can be seen, named either M_i or M_{ij}. Single index names represent non-replicated modules, double-indexed names belong to replicated modules, where index i denotes the module number and index j the number of the associated copy. In case of the replicated modules, both recovery approaches, parallel and serial, can be seen. The modules $M_{1j}, j = 1, \ldots, 3$, e.g., are an example of the latter. The replicated modules are drawn as separate boxes connected over dedicated recovery links (vertical links with unfilled triangular arrows at their ends). In case of the parallel replication strategy the module copies are depicted as single box, as can be seen for the modules M_{7j} in the design D_C. This reflects the strong interdependence of the module copies due to the interconnect networks required for all registers where voting is performed.

4.4.1 Selective Hardening of GALS Modules

Figure 4.21 illustrates that replicated and non-replicated modules, such as M_{1j} and M_4, can naturally co-exist in a GALS system. This is a relevant feature since not all parts of a chip

design necessarily have to be critical for reliable system operation. Consider, e.g., a module that produces output images for a screen. Brief transient faults that change a pixel for a fraction of a second might have a negligible effect on the quality of the output for a human viewer. On the other hand even single-bit upsets in many control circuits, e.g., in a processor have the potential to cause severe malfunctions. With rising integration capabilities we are more likely to see complex SoCs that include components of mixed criticality. In general, such systems might greatly benefit from the GALS design paradigm due to its inherent modularity, which simplifies separation of critical and non-critical components.

4.4.2 Replica Partitioning

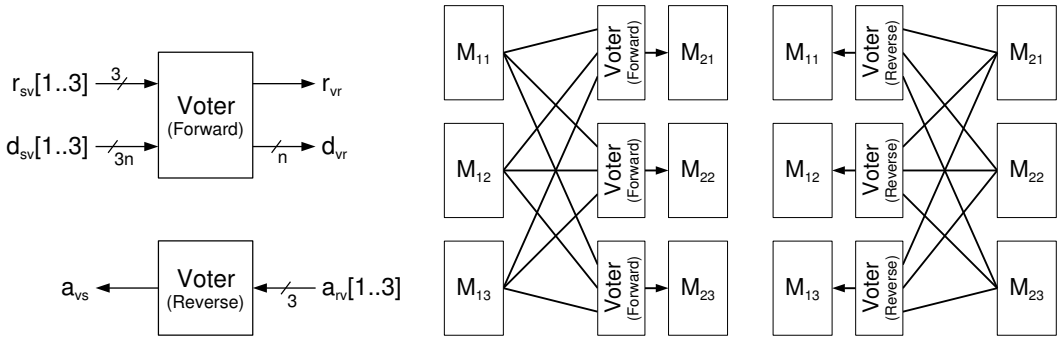
Another design decision is the partitioning of replicated components on different dies. This is a viable option, if the serial recovery approach is used since the interconnect signals needed for recovery only include the dedicated recovery links. Complexity is therefore low enough to route recovery signals off-die or off-chip, as can be seen for the modules M_{5j} and M_{6j} in Figure 4.21. Hence, a modular redundant system can be formed out of identical dies. Depending on the desired spatial separation between each of the dies, a system designer has the choice to put these dies into the same chip package, into different packages on the same board, or even on different boards. Using an asynchronous delay-insensitive handshake protocol for the recovery links, any of these scenarios is automatically supported and the designer does not need to worry about signal delays on on-chip or off-chip wires.

The partitioning of replicated modules on separate dies, however, has substantial consequences for the ability to perform voting on module outputs. From a reliability point of view, the best solution is to place voters between output and input ports of two communicating modules, as illustrated in Figure 4.21 for modules M_{1j} and M_{2j}. This prevents that errors in one module can propagate to another module, e.g., by a data transfer from M_{11} to M_{21}. If errors can propagate over inter-module I/O channels, these modules can no longer be assumed to fail independently, a fact that clearly has to be considered in the reliability analysis of the entire system. Furthermore this situation also influences the design and scheduling of the modules' recovery processes. To purge errors in all potentially "infected" modules, the recovery processes are required to be coordinated, as we will show in Section 4.5.4.

However, sanitising I/O data with voting comes at a cost, both in performance and area complexity. Especially the latter might be prohibitively large when module replicas are located on separate dies since the number of pads is usually too limited to route a bigger number of inter-module signals to other dies/chips. In Figure 4.21 we have therefore deliberately avoided voters between the modules M_{5j} and M_{6j}. The only off-chip signals are recovery links and regular primary input and output signals.

4.4.3 Voting on Output Data

Designing the voter unit for modular redundant inter-module GALS communication channels, as they can be seen in Figure 4.21, itself is already a notable task since it also involves voting on asynchronous control signals, like request and acknowledge. In fact, Figure 4.21 oversimplifies matters a little bit since two sets of voters are needed, one for the *forward* and one for the



(a) Voter interfaces. (b) Request & data nets. (c) Acknowledge nets.

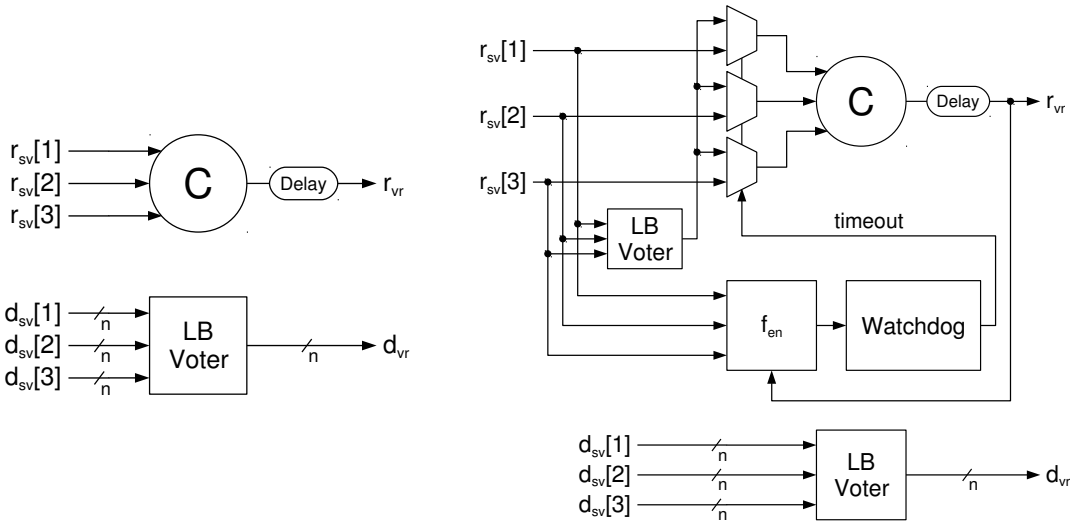
Figure 4.22: Voting on inter-module I/O links.

reverse paths of the handshake signals. The former are part of the receiving modules and vote on incoming request and data signals, the latter are associated with sender modules and process the returning acknowledge signals. Let us assume we have to design a voter for a TMR system, which handles three unidirectional bundled data links that transfer data between two sets of module copies. Thus, the modules' I/O channels can be broken down into the following signals:

- Sender modules to voter (forward path): request signals $r_{sv}[i]$, data vectors $d_{sv}[i]$,
- Voter to receiver module (forward path): request signal r_{vr}, data vector d_{vr},
- Receiver modules to voter (reverse path): acknowledge signal $a_{rv}[i]$,
- Voter to sender module (reverse path): acknowledge signal a_{vs},

where $i \in \{1, 2, 3\}$ in case of a TMR system. Figure 4.22a shows a schematic representation of the voters' ports, and Figures 4.22b and 4.22c illustrate the insertion of the voter components into the forward and reverse paths of the inter-module links.

Let us investigate the design of the forward path voter first (the reverse path then follows easily): Faults on the data signals $d_{sv}[i]$ can simply be masked with conventional level-based majority voters (*LB voters*). Majority voting on the request signals, however, can *not* be done just by inspecting the signal levels and forwarding the majority level. Doing this could break the handshake protocol for a slow non-faulty sender, which simply performs its request transition at a later point in time. For describing the desired voter behaviour for request signals let us consider the following two cases: 1) All request signals are fault-free, and 2) one request signal diverts from the correct behaviour. We assume that this diversion is caused by a fault in the sender driving the affected request signal. In the fault-free request scenario the behaviour and structure of the forward voter are straightforward, as can be seen in Figure 4.23a. The incoming handshake channels are joined to a single output request r_{vr}, which is generated by a 3-input Muller C-element. This ensures that a request transition is only propagated when *all* of the



(a) Initial implementation.

(b) Extension with watchdog mechanism.

Figure 4.23: Forward voter implementation.

incoming requests $r_{sv}[i]$ have made a transition. Also note that transitions on the output request r_{vr} should be delayed to match the propagation delays introduced by majority voting on the data vectors (bundled data approach, see delay element in Figure 4.23a).

With regard to Case 2, when one sender module is faulty, the essential responsibility of the voter is to ensure the successful execution of the data transfer between the other two fault-free senders and the (fault-free) receiver module. In other words, it must be guaranteed that handshakes started by the fault-free senders will eventually complete. The only acceptable interference of a faulty sender is to delay the communication until the fault has faded or has been detected by the voter. Two types of faulty behaviour of the request signals can be distinguished: 1) *Early transitions*. In this case the faulty sender f issues a request transition without providing stable and correct data over the respective $d_{sv}[f]$. However, such an early transition only becomes effective, if the other two senders have already issued their (correct) request signals. Since the data vectors $d_{sv}[i], i \neq f$ in this case can be assumed to be correct and stable, they will win the majority voting and consequently the data output d_{vr} will be correct. 2) *(Infinitely) late transitions*. This could, e.g., be the result of an erroneous state that hinders a sender to initiate an intended output handshake. In the voter design described above the *wait-for-all* semantics of the C-element clearly prevents the propagation of the rightful request transitions of the fault-free senders to the receiver. To resolve this situation a watchdog mechanism can be employed.

Figure 4.23b shows the necessary extensions to the voter implementation. A watchdog module, like we introduced for the recovery controllers in Section 4.3.3, controls a set of multiplexers that have been inserted after the incoming request signals $r_{sv}[i]$. The second input of these multiplexers is connected to a level-based voter, which determines the majority value of the request inputs. Thus, the C-element is pushed to make a state change, as soon as the *timeout* output of the watchdog is asserted. Note that if all of the request signals make correct transitions, the

watchdog will never run into a timeout and the voter behaves exactly like the initial implementation in Figure 4.23a. The watchdog timer only needs to be enabled when two out of the three incoming request signals have made a transition and the C-element has not yet changed its state. This condition is evaluated by an *enable function* f_{en} by examination of the request inputs and the C-element output. The truth table of this function block is shown in Table 4.4.

Table 4.4: Watchdog enable.

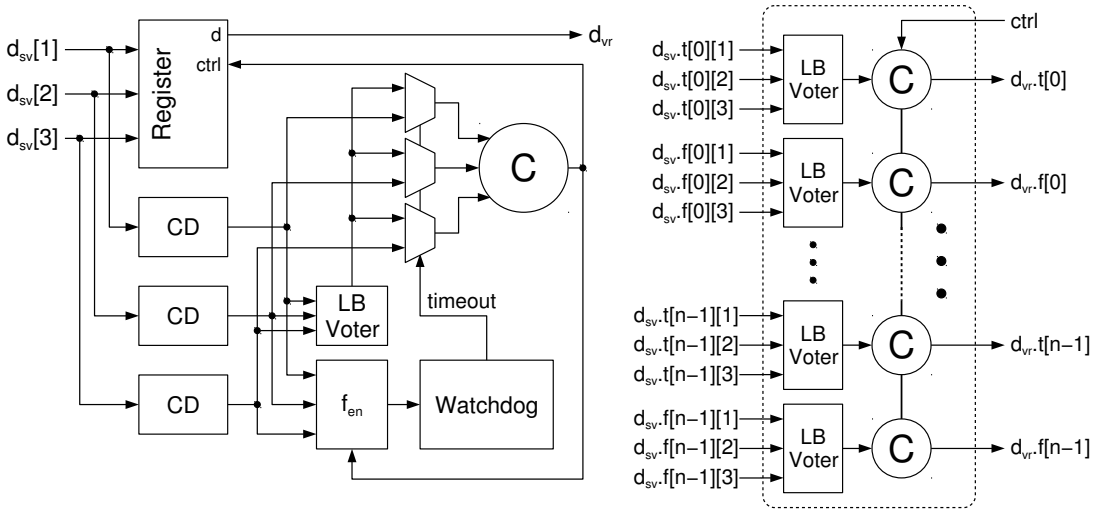
$r_{sv}[1]$	$r_{sv}[2]$	$r_{sv}[3]$	r_{vr}	f_{en}
0	0	0	0	0
0	0	0	1	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

It is essential that the timeout duration is chosen carefully for correct operation of this mechanism. Obviously this late transition detection must not trigger too early, otherwise the handshake routine of a fully functional sender module might be broken. Consequently, it is mandatory that the maximum execution time skew of the fastest and the slowest sender module is known. Let us assume that the sender modules, directly after the completion of a data transfer with the receiver modules, are almost perfectly synchronised, i.e., have a negligible skew. Let p_i denote the time it takes for sender i until the next output request transition is issued. Then the lower bound for the timeout value t of the forward voter can be defined as:

$$t > \max(p_i^{wc}) - \min(p_j^{bc}), \quad i \neq j \tag{4.5}$$

Since an increase of the communication latency only occurs in the (rare) case of an erroneous sender, the timeout value can be conservatively chosen with a big safety margin. This will not degrade overall system performance. It is, however, quite probable that an erroneous sender module, which fails to deliver output data on time once, will also fail to participate in subsequent data transfers. This would then result into a systematic performance degradation – until the inconsistent state of the erroneous sender is recovered. It is therefore advisable to trigger a recovery of the involved modules if a voter runs into a timeout several times in a row. We will discuss an appropriate recovery strategy in the next section.

As mentioned before, the design of the reverse voter for acknowledgement signals directly follows from the forward voter. Obviously a receiver module can be erroneous as well and consequently might fail to properly participate in the asynchronous handshake during a data



(a) Full voter with watchdog mechanism.

(b) Delay-insensitive voter register.

Figure 4.24: Delay-insensitive (dual-rail) voter implementation.

transfer. Again an intended transition of a broken receiver's acknowledge signal might occur (infinitely) late, thus hindering the successful completion of the handshake of the non-faulty sender with the other two non-faulty receivers. Hence, the same mechanism we have developed for the forward voter can be reused. The only difference is that the level-based data voters can be omitted, if there is no data channel bundled with the acknowledge signal.

Finally, we want to discuss a voter implementation for delay-insensitive data channels. The basic structure is quite similar to the bundled data version presented above. The main difference in case of the DI protocol is that request and data signals are combined into a single signal $d_{sv}[i], \forall i \in \{1, 2, 3\}$. As can be seen in Figure 4.24a, these data signals are processed by a register and completion detection (CD) units, one unit per redundant data channel. When *all* channels become complete ($timeout$ signal deasserted), a C-element, which is connected to the CD units, changes its value and triggers the register's control port to capture the new data word (or the spacer codeword in case of a 4-phase protocol). In case of a faulty input transition, the timeout mechanism works exactly like in the bundled data voter. The f_{en} block observes the outputs of the completion detection units and enables the watchdog, if a majority of CD outputs disagree with the value stored by C-element. If the watchdog then runs into a timeout, it raises its output signal, which again controls multiplexers for the C-element inputs. The multiplexers switch from the regular CD outputs to a voted version of these signals and thereby triggers an output change of the C-element, which in consequence enables the input register.

Figure 4.24b shows an input register for a 4-phase dual-rail code. For every rail of the resulting data vector d_{vr} the register stores the current value in a C-element. The top inputs of these C-elements are connected to the register's control input, and the second input is driven by a level-based voter, which evaluates the respective data rails from the modular redundant senders. This voter masks a faulty rail and ensures that the correct value is written into the register.

4.5 Recovery Strategy

For implementing the recovery process in a reliable and efficient manner it has to be decided when and how often a recovery should be initiated and what needs to be recovered. The strategies discussed in this section apply to both recovery approaches, parallel as well as serial.

4.5.1 Recovery Period

The time interval between scheduled recovery actions is dependent on the soft error rates that have to be expected in the environment during circuit operation and the specific circuit's reliability requirements. The basic assumption for any recovery/voting mechanism in a modular redundant system is that only a minority of replicated components are erroneous. Thus, when the state of replicated components is compromised, recovery actions need to be frequent enough to ensure with a sufficiently high probability that errors are repaired before a majority of module copies can become faulty. Consequently, a higher soft error rate clearly demands for faster recoveries. In order to find a viable recovery frequency for a given error rate a probabilistic analysis has to be performed. To this end we will present Markov chain models in Section 4.6.3.

Picking the right recovery frequency obviously poses a trade-off between system reliability and system performance. In case of the parallel recovery approach this trade-off is rather uncritical since the recovery can be performed in a single clock cycle. Therefore, a recovery can be scheduled very frequently to achieve ultra-high reliability without sacrificing performance. In case of the serial approach, where recovery might take hundreds or thousands of clock cycles, depending on the scan chain length, choosing a too high recovery frequency might prohibitively degrade the system performance. More sophisticated recovery strategies than a simple periodic recovery process might therefore be needed.

One way to maintain high reliability but keep the number of recovery actions and hence the performance loss low is to employ a twofold strategy: 1) An *event-triggered* recovery process could be implemented that is started immediately when an error is detected, e.g., by voter components at the primary outputs of the GALS system or on inter-module links. Since most of the state corruptions in a sequential circuit can be expected to have an effect on the outputs, the mean time a replicated component is erroneous can significantly be reduced. Some errors, however, can be latent and might be propagated at a later point in time, possibly causing an unrecoverable failure. Thus, the implementation of a 2) *time-triggered* recovery process is advisable, which is periodically started, without a previously detected error. Since latent errors are rare and are most likely to be overwritten or detected within short time, the frequency of the time-triggered recovery process can be low and will therefore not compromise the system performance.

4.5.2 Minimising the Recovery State

Aside of the frequency of the recovery processes, the specific points in time when a recovery is conducted have to be carefully selected. Some instants might be more advantageous for executing a recovery than others since the chosen time has a significant impact on *what* needs to be recovered. In general, the recovery process needs to include state-holding elements of a circuit, i.e., the register cells in a typical synchronous GALS module. However, it does not necessarily

need to be the case that *all* registers of the circuits have to be recovered. In [123] three classes of registers are distinguished: I) registers that contain output data, II) internal registers that will be overwritten after the checkpoint, and III) internal registers that contain data that will be further processed after the checkpoint. Clearly type III registers have to be included in the recovery and type II registers can be ignored as the stored value will no longer be used. Regarding output registers, the decision depends on whether out-going data values are subjected to voting (see system-level architecture considerations in Section 4.4). If this is the case, a faulty value will be masked by the voter and will vanish on its own when the next output value is written. In contrast, if an output register drives a communication link between two modules and no voting is performed, the corrupted data value is not masked and could still proliferate after the checkpoint. In this case output registers have to be included in the recovery process.

Let S_{rec} be the set of registers that need to be part of the recovery process, i.e., type I and type III registers. For maximum efficiency of the recovery process it is beneficial to insert checkpoints when the size of this set is minimal. This is typically the case when one computation has ended and the next one is yet to start. At this point in time there might be just a few internal registers that contain information needed for the next computational task, and maybe no output registers at all that need to be recovered since all outgoing data values have been transmitted. In [55] a state when no active task is executed is called *ground state*. Obviously, such a state does not necessarily exist in every circuit.

Minimising the circuit state that has to be recovered is both beneficial for the parallel and the serial recovery approach. In the former case fewer registers involved in the recovery means better area efficiency, i.e., fewer voters, multiplexers and interconnect signals. For the serial approach minimising S_{rec} directly reduces the execution time of the recovery process and thereby improves the system performance. Obviously, scan chains have to be organised appropriately to leverage this benefit, e.g., by gathering all registers to be recovered in a separate scan chain[5].

4.5.3 Replica Determinism

When active replication with a voting mechanism is used to build fault-tolerant systems, fault-free components need to behave *replica deterministic*, i.e., they are required to produce "*identical outputs in an identical order within a specified time interval*" [86]. Otherwise voting cannot be safely performed. For our recovery approaches replica determinism is required for register set S_{rec}, i.e., all registers $r \in S_{rec}$ in all (non-faulty) module copies need to have identical values when a checkpoint is scheduled.

For replicated GALS modules with identical module copies (with respect to their gate-level structure) we therefore propose a design style that enforces consistent values of *all* registers after *every* clock cycle[6]. This strict form of replica determinism can be established if three

[5]Note that the area overhead of separate scan chains is negligible since it does not increase the number of scan flip-flops but just adds an additional set of scan chain ports (se, si, so).

[6]This design style is actually more strict than necessary for successful recovery actions. As explained above consistency is only required for registers $r \in S_{rec}$ and only when a checkpoint is scheduled. Somewhen in the middle of two successive checkpoints the sequence of register values does not need to be identical for different module copies. In this thesis we only consider replicated components with the same structure, but it might also be viable to use very different implementations for the "replicas" of a modular redundant GALS module. As long as the

conditions are met: 1) all modules perform internal state changes in local synchrony and only use deterministic circuits for computation of the next state, 2) all copies receive the same inputs at the same local clock cycle, and 3) execute the same number of clock cycles between checkpoints.

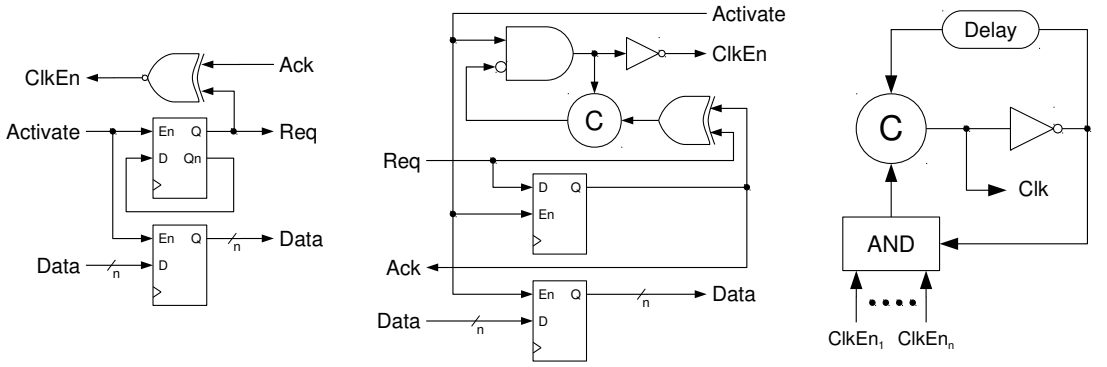
The first condition is usually satisfied for common synchronous GALS modules, and the third condition can be controlled by the design of the recovery controller, e.g., like we did with cycle counter, which had a fixed period. The second prerequisite, however, is more intricate. At first sight it seems to be in conflict with the GALS design style, where modules are only locally synchronized and communication is done over asynchronous channels. Fortunately, the desired behaviour can be achieved when a model of communication is used, where the (internally deterministic) modules themselves get to decide when an I/O operation is performed: A sending module, which performs a deterministic sequence of computation steps, will produce an output request after a specific, self-determined number of cycles. Similarly, a receiver module runs its computations and once it requires new inputs it will actively request the start of an input operation. The local computations of both the sender and the receiver are stopped until the end of an I/O operation. Recall that in GALS-systems with stoppable clocks generators such I/O ports are known as *demand-type* ports (we also use the term *blocking I/O*, cf. Section 3.2). The key property of this communication scheme is that *both* input and output operations are actively initiated. Obviously, this enforces that I/O activities have to be matched. For every input operation, initiated by a receiving module, there needs to be an output operation by the respective sending module, and vice verse. This model of communication was used, e.g., in [49], where a GALS-based FPGA architecture is introduced.

Figure 4.25 shows the design of I/O ports and a clock generator, similar to those presented in [49]. For both output and input ports an *activate* signal can be seen, which a GALS module can set to one for initiating an I/O operation. As a consequence, the *clock enable* signal of the respective port goes low, which then stops the clock generator. Note that the clock generator does not include mutual exclusion elements to perform arbitration of the next clock transition and asynchronous input signals. In this port design they are not needed as the clock is always stopped synchronously by the local module[7].

Of course, a real-life GALS system will typically be connected to external inputs as well, which are not part of a handshake protocol or a well-defined sequence of I/O operations. If such inputs are directly sampled by replicated modules, even if this is deterministically done at the same local clock cycle, they will capture the input value at slightly different points in time due to the non-synchronized nature of the system. Therefore, it cannot be guaranteed that they will read a consistent value. Note that input consistency is a general problem for any replicated system [85] and the same problem also exists for replicated circuits with a single clock domain. Simply consider three flip-flops in a TMR system linked to separate input pads, which then connect to the same external signal. Due to clock skew and different input delays, it is not possible to sample the input signal three times at the exact same point in time. Furthermore, asynchronous signals have to be synchronized to avoid metastability. This is a non-deterministic

relevant internal state at the selected checkpoints is identical, this does not contradict with the recovery mechanisms we propose. Design diversity might actually be a powerful means to increase the robustness of GALS system.

[7] In fact, mutex elements would be highly undesirable in a replica-deterministic design since the arbitration process obviously is a source of non-determinism.



(a) Output port. (b) Input port. (c) Stoppable clock generator.

Figure 4.25: I/O ports and clock generator for replica deterministic GALS modules.

process, which can lead to inconsistent results.

One solution to guarantee deterministic behaviour for the GALS system is to use a non-replicated input component, which can be requested by the replicated modules over demand-type ports. The component then samples input signal once and returns this value to the waiting modules. Since this component would be a single point of failure a careful reliability assessment is required or the application of non-replication-based fault-tolerance mechanisms.

Alternatively, input components could be replicated as well. However, in this case the components need to implement a fault-tolerant consensus protocol [84] to agree on the same value, which can then be forwarded to the replicated GALS modules.

4.5.4 System-Level Considerations

Until now we have just discussed the recovery process for single replicated modules. Since a complete GALS system can contain multiple (replicated and non-replicated) modules, an important question is whether a checkpoint can be chosen individually for every replicated module or if a coordinated recovery is necessary.

Note that a replicated module in a GALS system can be stopped at any given time to perform a recovery without interfering with the regular operation of other system's modules. These would only be stopped, if they needed to perform an I/O operation with the recovering module. Stopping and starting local modules while other parts of the circuit remain completely operational is a very powerful feature. GALS circuits naturally support this concept by providing the stoppable clocks and elastic interfaces between modules based on asynchronous handshakes.

Intuitively one might therefore think that recovery actions could also be executed completely independent for different modules. However, care has to be taken with the timing of recovery processes in case of communicating modules. An effective recovery process needs to ensure that a single fault occurring in one module copy and the subsequent corruptions can only proliferate until the module reaches its next checkpoint. This requirement always holds, if the module does not communicate with other modules. However, if there are I/O operations among modules

between two successive recovery actions and *no* voting is performed on incoming data word, the checkpoints cannot be scheduled arbitrarily.

To reason about this situation we need to consider the lifetime of an erroneous state within a module. Figure 4.26 shows a problematic trace with two communicating module copies, M_1 and M_2. Each module performs two recovery operations, the respective checkpoints are named CP_{ij}. These checkpoints divide the execution of the system into three computation rounds, where round i is terminated when *all* modules have reached the checkpoint at the end of their i-th computation cycle and have executed the recovery action. In the first round module M_1 is corrupted by a fault. Since M_1 sends no messages after the corruption, the faulty state is contained in M_1. The corrupted state is then recovered at checkpoint CP_{11}. The second computation round thus starts with a consistent and fault-free system state – until a second transient fault again causes a corruption of M_1. This time a message, m_3, is sent to module M_2. Assuming the message is erroneous due to the second transient fault, the state of M_2 is impaired as well (recall that M_2 does not perform voting on incoming messages). After some clock cycles M_1 runs into checkpoint CP_{12} and recovers its state with the help of its correct module copies. Unfortunately M_2, still corrupted because of m_3, sends back a faulty message, m_5, before it reaches its upcoming checkpoint CP_{22}. Thus, the error that has already been recovered by M_1 is reintroduced. To prevent such a scenario, messages must not be exchanged across different computation rounds. In the above example message m_5 is sent in round II and received in round III. If both checkpoints, CP_{12} and CP_{22}, had been scheduled either before or after the transmission *and* reception of m_5, the errors in both modules would have been properly recovered.

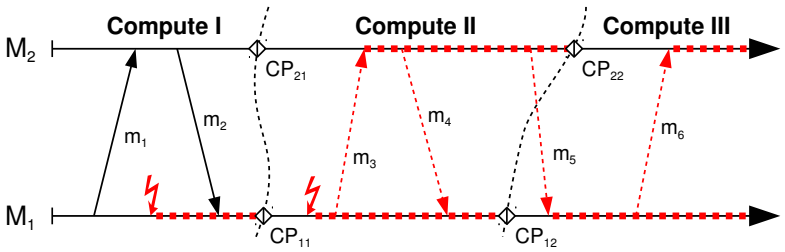


Figure 4.26: Problematic scheduling of checkpoints.

4.6 System Evaluation

In this section we will evaluate area, performance and reliability of the proposed TMR approaches are compare them with a conventional TMR architecture. Since specific results can only be stated for a specific circuit, we have implemented a representative real-life showcase design, which has a reasonable size to get a good understanding of the effects of the applied fault tolerance methods. The baseline design is a synchronous embedded 32-bit processor with a RISC architecture. The processor is named SCARTS[8] [65] and was developed in our research group, the *Embedded Computing Systems* group of the *Vienna University of Technology*.

[8]Formerly also known as Spear2.

4.6.1 Design Automation

Since a full processor is much too complex to build a TMR system manually, we have developed a tool for automating the design process. The tool performs a gate-level transformation of a given non-redundant circuit and is capable of generating a triplicated netlist for conventional TMR designs with voters directly inserted into the data path, as well as netlists for our parallel GALS TMR configuration, where feedback paths with voters and multiplexers need to be introduced for the recovery mechanism. For building designs following our serial TMR approach we do not need a customised tool, since scan insertion obviously can be done with conventional synthesis programs and triplication is effortlessly achieved by manually instantiating the top-level component of the synthesised netlist three times.

The first step of the devised design flow is a conventional synthesis of the non-redundant circuit's RTL specification, using a technology-independent target library. Then our tool parses the generated netlist and performs at series of transformation steps to produce a new netlist with the required triplicated circuit structures. The changes applied to the netlist are:

- Adaption of the interface of the module and its subcomponents: Triplication of all input and output ports, including data ports as well as the clock port. Introduction of the three additional *recover* input ports at every level of hierarchy (in case of the parallel approach).
- Triplication of all gates and signal nets within the module and its subcomponents, both combinational and sequential. Flip-flop cells are replaced with specialised TMR flip-flop components, which simply contain three flip-flops and the voting and recovery circuits, as required by the chosen TMR architecture. The TMR flip-flop components have triplicated input and output signals (*reset*, *clk*, *d*, *q*) and can therefore be easily integrated into the triplicated netlist. In case of the proposed parallel TMR structure, the components also provide three input ports for connecting the *recover* signals.

Since no changes to the basic circuit structure have to be performed, the algorithm remains fairly simple and runs with linear time complexity, in terms of module ports and internal gate count. Currently only VHDL netlists are supported. However, the transformation could be easily extended to process Verilog netlists as well.

The triplicated netlist generated by our tool can finally be mapped to standard cells of the desired target technology, using a conventional synthesis tool again. The remaining design tasks until sign-off then follow the same steps as performed in an ordinary ASIC design flow.

4.6.2 Area & Performance

For our evaluations we have implemented and synthesised four designs of the SCARTS processor with a standard cells library for a UMC90 process: 1) the unchanged non-redundant processor version (referred to as *simplex baseline*), 2) a version following the conventional TMR architecture, and finally our two GALS-based solutions with a 3) parallel and 4) serial state restoration scheme. The simplex design contains 1288 flip-flops and for sake of simplicity we assumed that all of them are critical and have to be triplicated and included in the recovery mechanism. Since design for test structures are necessary for any real-life circuit nowadays, a single

scan chain was inserted in all four processor versions. In case of the serial TMR approach, this scan chain was then reused for state restoration.

Figure 4.27 shows the area comparison of the resulting circuits normalised to the area of the simplex baseline. The figure illustrates the individual contributions of the SCARTS processor and recovery controllers to the total area, whereby combinational and sequential parts are listed separately. In case voters or multiplexers have been added, the combinational part of the SCARTS processor is further subdivided to show their contributions.

As can be seen, the sequential area of our TMR designs is more or less three times the respective area of the baseline processor. This is not surprising since all we did was to triplicate the flip-flops. However, the combinational areas in case of the conventional TMR design and our parallel recovery approach have notably increased. This clearly shows the non-negligible effect of inserting (triplicated) voters and multiplexers into the processor design. In total the conventional TMR version is approx. 3.7 as big as the baseline, whereas the area in case of the parallel approach is quadrupled, due the additional multiplexers. The processor area of the serial TMR approach, on the other hand, is exactly three times the simplex processor, since it was simply triplicated at top level without any internal changes.

With respect to the recovery controllers, it we can observed that in both GALS TMR designs they only have a minor contribution to the overall area. In case of the parallel approach it is less than 1%, for the serial approach it amounts to approx. 5% of the system's area. Area-wise this makes the serial recovery mechanism the clear winner of all TMR solutions, since its area totals just slightly above three times the non-redundant processor implementation. A detailed break down of the area results for all four designs can be found in Table 4.5.

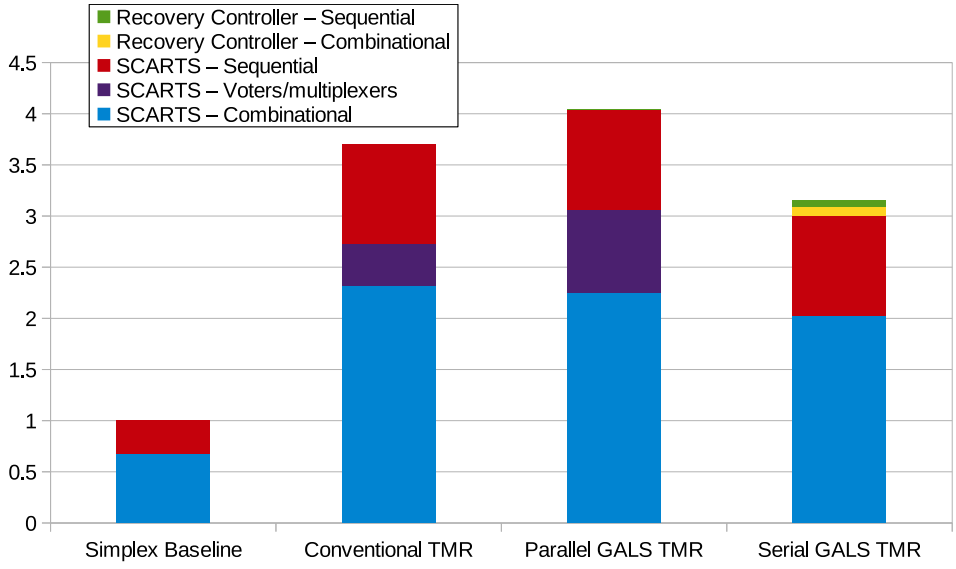


Figure 4.27: Area comparison.

[9]Combinational area excluding voters and multiplexers used for recovery.

Table 4.5: Detailed area results (units in μm^2).

		Simplex Baseline	Conventional TMR	Parallel GALS TMR	Serial GALS TMR
SCARTS	combinational[9]	51936.86	177982.90	173320.45	155810.59
	voters/multiplexers	0	31461.14	62432.272	0
	sequential	24932.77	74834.37	74679.92	74798.30
	total	76869.63	284278.40	310432.64	230608.90
Recovery Controller	combinational	0.00	0.00	237.55	7300.60
	sequential	0.00	0.00	446.88	4534.63
	total	0.00	0.00	684.43	11835.23
Entire design	combinational	51936.86	209444.03	235990.27	163111.19
	sequential	24932.77	74834.37	75126.80	79332.94
	total	76869.63	284278.40	311117.07	242444.12

An important aspect for the evaluation of the system performance is the degradation of the maximum operating frequency due to voters or multiplexers, which are added to the original circuits in order to implement the state restoration mechanism. We have therefore performed a static timing analysis on the three synthesised TMR designs and compared their maximum clock frequencies with the simplex baseline processor. The results of this analysis are shown in Table 4.6[10]. While the serial TMR design obviously does not suffer from a reduced operating frequency, in case of the other two TMR systems a small but noticeable performance drop needs to be accepted. This drop is a bit more pronounced for the conventional TMR architecture since the voters are in the critical path. In case of the parallel approach voters are not in the critical path, however, multiplexers have been added, which also account for a reduction of the clock frequency. Note that the voters nevertheless have a minor impact on the circuit timing during normal operation since they introduce additional load to the flip-flop outputs.

Table 4.6: Performance comparison.

	Simplex Baseline	Conventional TMR	Parallel GALS TMR	Serial GALS TMR
Min. clock period (ns)	2.24	2.59	2.50	2.24
Max. clock frequency (MHz)	446.43	386.85	400.80	446.43
Performance degradation (%)	N/A	13.35	10.22	0

For the parallel and serial TMR approaches an additional performance degradation is caused by the service interruption due to the execution of recovery processes. As discussed in Section 4.5, the amount of this degradation depends on the frequency and the duration of these recovery processes. The duration in case of the parallel approach is simply one clock cycle, for the serial approach it depends on the frequency of the clock to shift data out of and back into the scan chain, the length of the scan chain, the bus width of the recovery links and their commu-

[10]Please note that all figures presented here are based on a pre-layout timing analysis.

nication latency. In our SCARTS showcase design, the clock period is $2.24\,ns$, the number of flip-flops is 1288 and the recovery links have a bus width of 8. Based on a timing simulation we have determined the communication latency for one full handshake cycle on the recovery links to be $2.43\,ns$. Please note that in our simulation model all three replicas are assumed to be on the same die. If data and acknowledge signals between the recovery controllers need to cross chip-boundaries, a longer communication latency has to be expected. In total, our simulation shows that a full recovery process of the SCARTS processor takes $3.28\,\mu s$, i.e., approx. 1463 clock cycles. For the serial TMR approach the key to achieve a reasonable overall performance therefore is to find the minimum recovery frequency, which is required to achieve the desired MTTF. To this end a reliability analysis of the given circuit needs to be performed.

4.6.3 Reliability

The overall reliability of a modular redundant GALS module clearly depends on the failure rates that have to be expected for the module's circuit parts, the chosen replication and recovery strategies including the designated restoration rates and durations. In order to evaluate the reliability of the fault tolerance approaches presented in this thesis, we have developed mathematical models based on Markov chains. The basic input parameters, which will be common to all models in this section, are the overall soft error rate of the original non-redundant GALS module λ_m, and the individual SER contributions of the combinational/sequential logic and the clock tree:

$$\rho_{comb} = \frac{\lambda_{comb}}{\lambda_m}, \quad \rho_{seq} = \frac{\lambda_{seq}}{\lambda_m}, \quad \rho_{clk} = \frac{\lambda_{clk}}{\lambda_m} \tag{4.6}$$

Let us first investigate the reliability of conventional TMR circuits, where both combinational and sequential parts are triplicated and a voter is inserted after every flip-flop. Erroneous states are therefore restored every clock cycle (cf. Section 3.3.1). As we have seen in the previous subsection, the introduced voters increase the area of the combinational circuit parts and therefore contribute to the failure rate of the module copies in the TMR system. Let a_{voters} denote the increase in combinational area in one copy due to the new voters. We can then define the failure rate λ_r of a replicated module, which accounts for soft errors in combinational and sequential circuits parts:

$$\lambda_r = \lambda_{seq} + \lambda_{comb}(1 + a_{voters}) = \lambda_m \cdot \rho_{seq} + \lambda_m \cdot \rho_{comb}(1 + a_{voters}) \tag{4.7}$$

A critical (and often overlooked) parameter for the reliability analysis of a conventional TMR circuit is the failure rate of the clock tree. Let us assume that the clock tree is not replicated and therefore a single transient pulse could affect multiple circuit copies at once, corrupting the system state beyond recovery. This fact needs to be reflected in our reliability model. Assuming that the clock tree area grows linearly with the number of endpoints, the TMR system's clock tree would be three times as large as the corresponding tree of the non-redundant circuit, and hence have a threefold increased failure rate. We denote this failure rate with $\lambda_c = 3 \cdot \rho_{clk} \cdot \lambda_m$.

As a final step, before we can build the desired Markov chain, we need to make sure that the failure rates defined above are specified in a useful unit for our model. Since state restoration

is performed at every clock cycle, we will be using discrete-time Markov chains, with time units in clock cycles. Component failure rates, on the other hand, are typically specified in *failures/hour*. We therefore need to divide λ_r and λ_c by a scaling factor $s = 60 \cdot 60/p_{clk}$, where p_{clk} denotes the clock period of the system in seconds. Let λ_{rs} and λ_{cs} be the failures rates scaled to *failures/clock cycle*. Based on these parameters we can now devise a very simple Markov chain, which models the failure behaviour of a conventional TMR circuit.

As can be seen in Figure 4.28, this Markov chain consists of two states: In State 1 the system delivers a correct service, whereas State 2 represents the occurrence of a system failure. State transitions are performed in discrete time steps with every clock cycle, based on the transition probabilities specified on the arcs. The system goes into the failed state, if one of the following events occurs during a clock period: i) Two out of three copies become faulty. In terms of λ_{rs} the probability of this event can be expressed as $\binom{3}{2}\lambda_{rs}^2(1-\lambda_{rs})$. ii) All three copies get corrupted due to three independent soft errors with probability λ_{rs}^3. iii) A single transient fault in the clock tree overthrows the state in all copies. The probability of this event equals λ_{cs}.

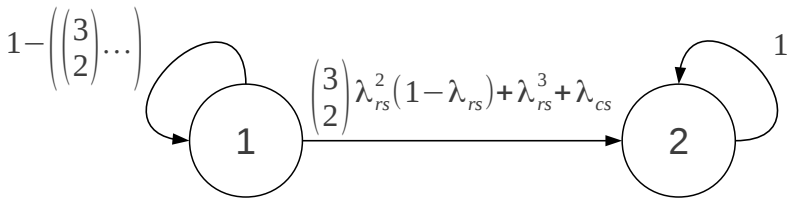


Figure 4.28: Markov chain of conventional TMR circuit.

On the other hand, if no error occurs during a clock cycle or just a single copy becomes faulty, the system stays in State 1. The probability of this event can be expressed as 1 minus the probability of making a transition to State 2. Note that State 2 is never left again. The probability to stay in this state consequently is 1. Such a state is called *absorbing*, and a Markov chain that has at least one absorbing state therefore is called *absorbing Markov chain* [38]. All other states, which are not absorbing, are called *transient*. The transition matrix associated with the Markov chain shown in Figure 4.28 is

$$T = \begin{pmatrix} 1 - \left(\binom{3}{2}\lambda_{rs}^2(1-\lambda_{rs}) + \lambda_{rs}^3 + \lambda_{cs}\right) & \binom{3}{2}\lambda_{rs}^2(1-\lambda_{rs}) + \lambda_{rs}^3 + \lambda_{cs} \\ 0 & 1 \end{pmatrix} \tag{4.8}$$

If the states are ordered so that transient states have the lowest numbers, as it is the case for our simple Markov chain, the transition matrix has the following *canonical* form:

$$T = \left(\begin{array}{c|c} Q & R \\ \hline 0 & \mathcal{E} \end{array}\right) \tag{4.9}$$

Assuming there are t transient states and r absorbing states, Q is a $t \times t$ matrix, which describes the probabilities to move from one transient state to another. R is a $t \times r$ matrix, which contains the probabilities to make a transition from a transient to an absorbing state. The zero submatrix at the bottom left of T represents the zero probability to leave an absorbing state, whereas the identity matrix $\mathcal{E}$ denotes the probability of 1 to stay in an absorbing state.

If the transition matrix is given in canonical form, the so-called *fundamental matrix* N can be easily calculated. An element n_{ij} of N equals the expected number of visits to a state j before an absorbing state is entered (absorption), given that the initial state was i. The fundamental matrix is defined as follows [38]:

$$N = \sum_{k=0}^{\infty} Q^k = (\mathcal{E}_t - Q)^{-1} \tag{4.10}$$

Based on the fundamental matrix, a vector t can be computed, whose i-th entry equals the expected number of state transitions made before absorption, given that the initial state was i:

$$t = Ne, \tag{4.11}$$

where e is a column vector with all entries set to 1. For our reliability evaluation, where absorbing states represent failed states, this is a very convenient methodology to compute the MTTF of the described TMR system. Applying the above equations to the transition matrix of our Markov chain, an evaluation of the t-vector's first entry yields the following system MTTF[11]:

$$MTTF = \frac{1}{-2\lambda_{rs}^3 + 3\lambda_{rs}^2 + \lambda_{cs}} \approx \frac{1}{3\lambda_{rs}^2 + \lambda_{cs}} \tag{4.12}$$

To illustrate the effectiveness of the conventional TMR approach we have applied the above MTTF formula to the SCARTS processor system. Based on the performance and area results presented in the previous subsection, we assume a value of $2.59\,ns$ for the clock period p_{clk} and the combinational area overhead due to the inserted voters can be set to $a_{voters} = 0.14$. Figure 4.29 plots the results of our reliability evaluation. As can be seen, the plot profiles the redundant system's MTTF as a function of different MTTF values for the corresponding simplex system. Obviously, the soft error rates of a circuit depend on the specific technology used and the environment the chip operates in. To show the impact of different failure rates, we have assumed a simplex MTTF ranging from 100 to 1000 hours, i.e., $\lambda_m = \{1/100, \ldots, 1/1000\}$.

Furthermore, we need an assignment of the parameters ρ_{comb}, ρ_{seq} and ρ_{clk} to determine the contributions of combinational, sequential and clock tree subcircuits to the total soft error rate. In [73] it has been reported that in typical designs, such as microprocessors or network processors, 11% for the soft errors can be attributed to static combinational logic, whereas sequential elements and unprotected SRAM have a share of 49% and 40%, respectively. Since we did not consider circuits with SRAM cells for our TMR approaches, we will disregard their share and assume that failures in combinational and sequential elements as well as the clock tree add up to 100% of the total failures. Regarding the susceptibility of the clock tree we want to refer to the analyses performed in [17, 18], which strongly suggest that its contribution should not be neglected. In [97] it is even estimated that the clock SER of a flip-flop based design accounts for up to 9% of the total soft error rate. To comprehend the impact of transient faults in the clock network, we have therefore computed and plotted the MTTF of the TMR system for different values of ρ_{clk}, ranging from 1% to 10^{-6}% of the total soft error rate. As can be seen

[11] Note that this result gives the MTTF in units of clock cycles. To get more meaningful numbers a conversion to failures/hours can by done by dividing with the scaling factor s.

in Figure 4.29 this parameter has a dramatic effect on the resulting system reliability, spanning multiple orders of magnitude from the best-case to the worst-case scenario. This result illustrates the importance of using separate clock trees for replicated circuit parts.

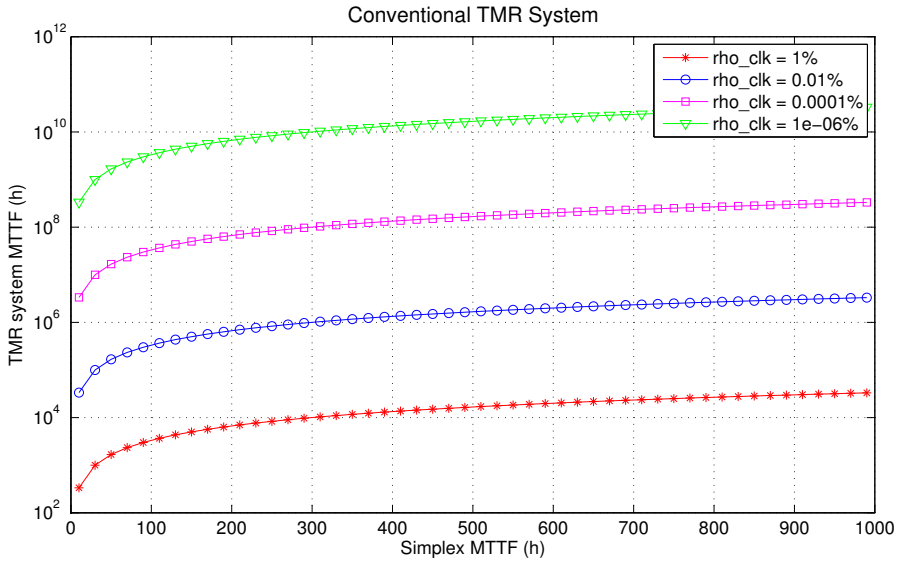


Figure 4.29: Plot of $MTTF$ for a conventional TMR system.

The Markov chain for the parallel and serial TMR approaches is a bit more comlex than the model presented above since we need to incorporate the recovery mechanism, which interrupts the regular computation at specific points in time. For sake of simplicity we will assume a periodic schedule for recovery processes. In this case, let us denote the durations of computation and recovery phases with the parameters d_{comp} and d_{rec}, respectively (in units of clock cycles). Furthermore we need to model the effect of the area overheads caused by circuit elements related to the recovery mechanism, i.e., voters, multiplexers, recovery controllers, on the failure rate of the replicated modules. Let λ_m be again the failure rate of the non-redudant circuit design. The failure rate of a replicated module, including all extensions, can then be written as:

$$\lambda_r = \lambda_m \cdot (\rho_{seq} \cdot (1 + a_{recSeq}) + \rho_{comb} \cdot (1 + a_{recComb}) + \rho_{clk} \cdot (1 + a_{recClk})), \tag{4.13}$$

where a_{recSeq}, $a_{recComb}$ and a_{recClk} denote the area overheads of all recovery-related combinational, sequential and clock tree circuits with respect to the non-redundant implementation. Dividing λ_r by s again gives us the failure rate per clock cycle λ_{rs}. To get the failure rate of the computation and recovery phases we can now simply multiply λ_{rs} with their respective duration:

$$\lambda_{comp} = \lambda_{rs} \cdot d_{comp} \tag{4.14}$$

$$\lambda_{rec} = \lambda_{rs} \cdot d_{rec} \tag{4.15}$$

Based on these two parameters we have then devised the Markov chain shown in Figure 4.30. Note that this model can be used for both our parallel and serial TMR approach, since they follow the same fundamental concept and only differ in the specific assignment of the parameters defined above (e.g., $d_{rec} = 1$ for the parallel approach, and $d_{rec} > 1$ in case of the serial approach). As can be seen in Figure 4.30, the Markov chain consists of five states. States 1 and 2 represent system states during regular computation phases, whereas States 3 and 4 model an active recovery process. State 5 is the only absorbing state, which is assumed in case of a system failure. Initially the model starts in State 1, where all replicated modules operate in fault-free condition. After a computation phase the model makes a transition to State 3, 4, or 5, depending on the number of failed replicated modules. A failure of two or more modules results in a system failure and State 5 will be entered. State 3 respresents the execution of a recovery process following a computation phase, which suffered from one failing replica. In this state any further module failure again leads to a system failure and therefore to a transition to State 5. If the recovery is successful, on the other hand, the model returns back to State 1.

The transition from State 1 to State 4 is taken, if no soft error occured during the a computation phase. In this case the failure of a single module can still be tolerated during the subsequent recovery process. If such a failure occurs, State 2 is entered after the recovery process. This state denotes that the next computation phase is executed with one module being corrupted right from the start of the phase. This is a critical configuration since another soft error in one of the two remaining modules immediately causes a system failure, which is modeled by the transition from State 2 to State 5. However, if the system is able to survive this critical phase, State 3 is entered when the next recovery process is started, which might then lead back to an error-free configuration in State 1. Using the transition matrix of this Markov chain we can again derive an MTTF formula, which applies for the parallel and the serial TMR approach:

$$MTTF_{steps} = \frac{2}{\lambda_{rs}^2(d_{comp} + d_{rec})(3 \cdot d_{comp} + 9 \cdot d_{rec})} \tag{4.16}$$

Note that the above formula expresses the system MTTF in terms of the step counts until absorption. Due our modeling approach of this Markov chain, a step is not a single clock cycle, but covers either a full computation phase or a recovery phase. This step count, however, can be easily transformed into clock cycles. Since computation and recovery phases alternate, we can infer that two successive steps have a duration of $d_{comp} + d_{rec}$. Consequently, the final MTTF formula in units of failures per hour can be expressed as follows:

$$MTTF = \frac{MTTF_{steps}}{2 \cdot s}(d_{comp} + d_{rec}) = \frac{1}{s \cdot \lambda_{rs}^2(3 \cdot d_{comp} + 9 \cdot d_{rec})} \tag{4.17}$$

We can now perform a reliability evaluation of the parallel and serial TMR versions of the SCARTS processor design. For the parameters ρ_{comb} and ρ_{seq} we assume the same base values as above, for the contribution of the clock tree to the total soft error rate we set ρ_{clk} to a fixed value of 1%. The MTTF of the simplex implementation λ_m ranges again from 100 to 1000 hours. Based on the area results of Section 4.6.2, we can determine area overheads associated with the recovery circuits. In case of the parallel approach we need to account for additional voter and multiplexer area. The recovery controller increases both combinational and sequential circuit

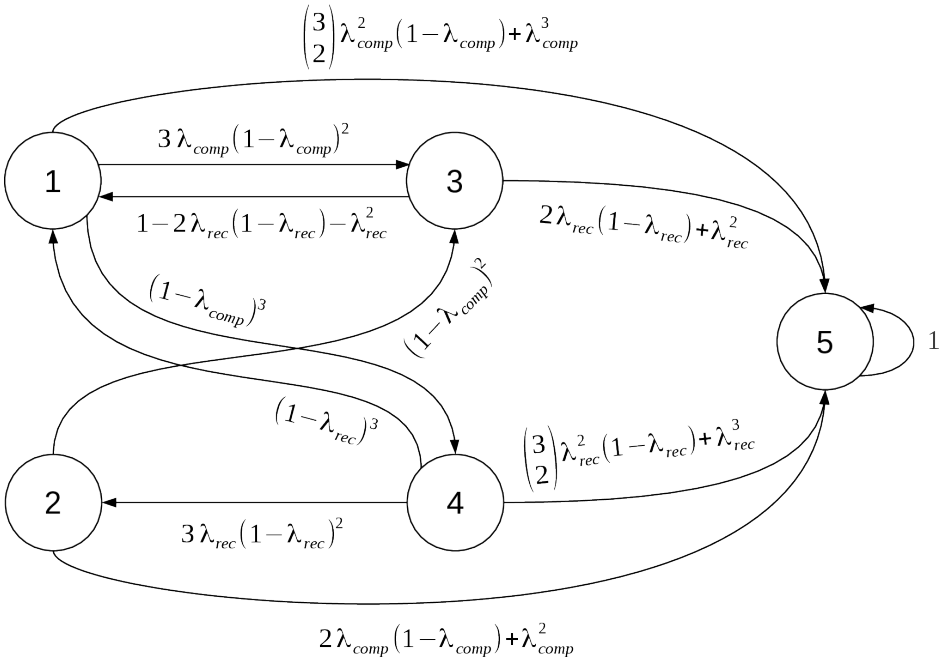


Figure 4.30: Markov chain of TMR approaches with periodically executed recovery processes.

area. For sake of simplicity we include the full recovery controller into the computed overhead, even though the parts of the asynchronous control cicuits have been designed to tolerate single event transients. In any case, the area contribution of the recovery controller is much too small to have a significant effect on the reliability results. The clock tree is increased by a few endpoints due to the counter implemented in the recovery controller. Assuming an 8-bit counter, the area overhead in comparison to the SCARTS clock tree with 1288 endpoints is close to zero.

In case of the serial TMR approach, the processor module itself remains unchanged and we only have to account for overheads caused by the recovery controller. The specific overhead values for both TMR approaches are listed in Table 4.7.

Table 4.7: Area overheads.

	$a_{recComb}$	a_{recSeq}	a_{recClk}
Parallel approach	0.515	0.004	0.006
Serial approach	0.047	0.061	0.039

With respect to performance we use the clock periods, which were determined for the parallel and serial TMR solutions in the previous subsection. Furthermore, the recovery duration d_{rec} is set to 1 in case of the parallel approach and to 1463 clock cycles for the serial recovery solution. As we discussed before, the frequency of the recovery processes has a crucial influence on the resulting system performance as well as the reliability. In order to analyse this relationship we evaluate the resulting system reliability for different performance overheads of 0.1%, 1%,

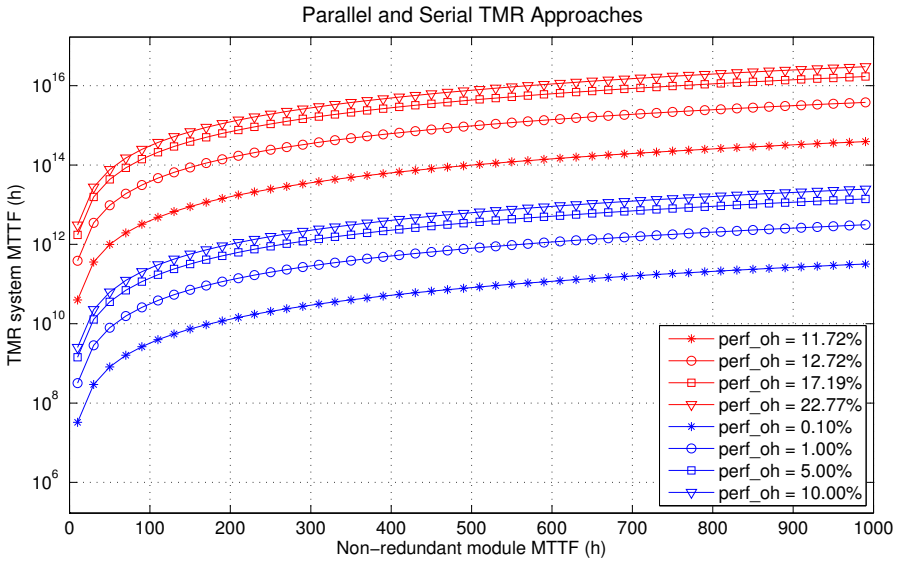


Figure 4.31: MTTF of TMR systems using periodic recovery for different performance overheads (parallel approach in red, serial approach in blue).

5% and 10%. The length of the computation phase d_{comp} is assigned accordingly. Note that these are recovery-related overheads, which are on top of the possible performance degradation due to the increased clock period (with respect to the simplex implementation). Figure 4.31 shows the resulting MTTF for the parallel and the serial approach. In the former case, the overall performance overhead is always greater than approx. 11%, which is due to the reduced clock frequency. On the other hand, for the serial approach, which does not suffer from a degradation of the clock frequency, the performance overhead can be made arbitrarily small, simply by reducing the recovery frequency. Performance can therefore be traded with reliability. Nevertheless, our results show that the SCARTS system with serial recovery can deliver ultra-high reliability even for very small performance overheads. As can be seen, with a degradation of 0.1% the TMR system stills provides an MTTF of more than 10^9 hours, given that the simplex module MTTF is greater than 100 hours. Obviously, the parallel approach can provide an even higher reliablity, due to the single clock cycle recovery latency. Nevertheless, when looking at all results with respect to area, performance and reliability we can conclude that the serial approach certainly provides the most efficient solution, which outperforms both conventional TMR systems and the parallel recovery scheme we have presented.

4.7 Related Work

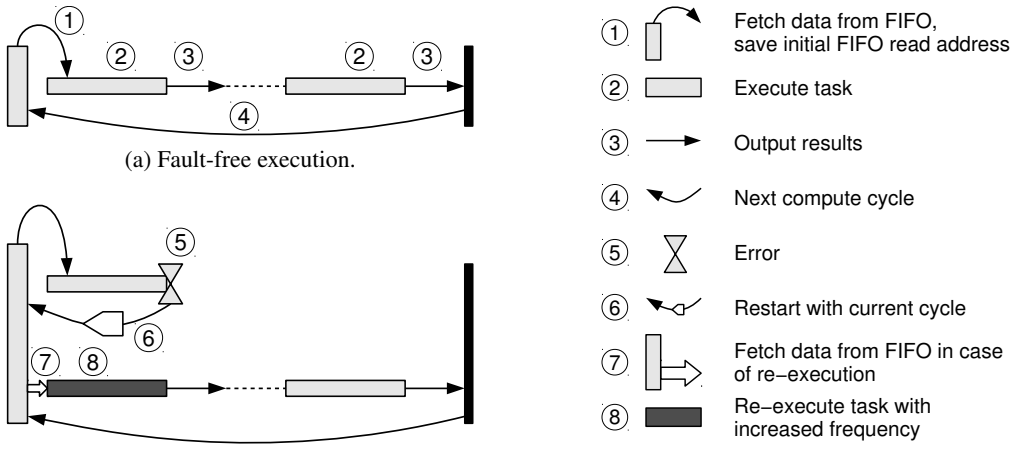
To the best of our knowledge we are the first to explore the application of modular redundancy in GALS systems. In general, fault-tolerant computing in GALS has not been covered in the scientific community. The only exception we know of are Yu, Shi and Zeng [124], who describe a fault-tolerant system of GALS multiprocessors. However, there exist some works on similar

fault tolerance techniques and concepts not related to GALS, which we have referenced in the relevant sections of this Chapter. We want to introduce them here in greater detail and highlight similarities and differences to our work.

Yu, Shi & Zeng

In [124] a GALS-based computing architecture is introduced, consisting of a 2-D array of processors as locally synchronous modules. The system is targeted at stream DSP applications, which can be broken down into parallel tasks and then distributed among the available processors. Data exchange between the processors is performed over dual-clock FIFOs, which serve as data buffers and provide reliable communication across clock domains, in case the processors are operated with different clock frequencies. The used programming model for stream DSP applications is quite simple. Data processing is executed periodically by reading data from the input FIFOs and then running the appointed task, which might be interleaved by writing results to the outputs. Once the task is finished the next chunk of data can be read from the input FIFOs.

Resilience against transient faults is achieved with a duplication and comparison approach and checkpointing with rollback recovery. Tasks are executed on two neighbouring processors, which exchange all computed results and compare them before they are passed to the next processor. A checkpoint is taken whenever new input data is read from a FIFO. The saved status information basically includes the local processor's program counter and the current read address of the FIFO. In case an error is detected in the results, the involved two processors can easily perform a rollback by resetting the FIFO read address and the program counter to the last checkpoint and then re-execute the program from there on with the same inputs, still available in the FIFO buffers. Fault-free outputs, which have already been generated in the first run, are suppressed during the re-execution. This is simply done with a counter, which keeps track of successfully produced outputs during an execution cycle. A graphical illustration of this recovery approach can be seen in Figure 4.32.



(a) Fault-free execution.

(b) Re-execution due to error occurrence.

(c) Symbol descriptions.

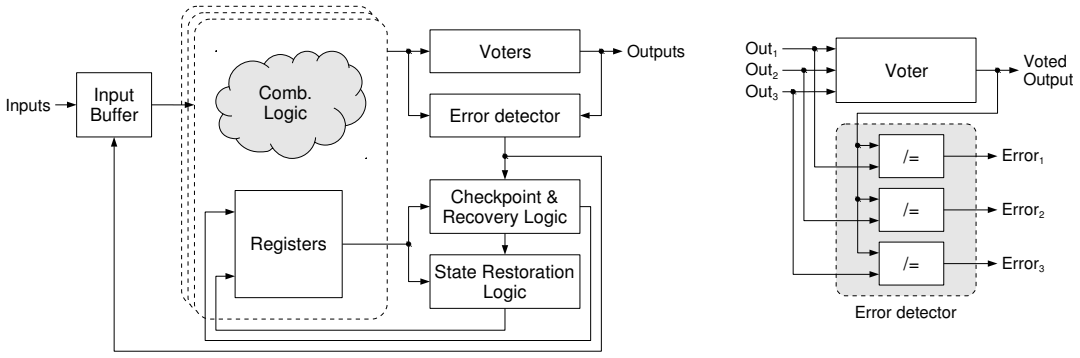
Figure 4.32: Fault tolerance scheme for stream DSP applications (cf. [124]).

Obviously re-execution results in a performance penalty. Yu et al. therefore propose to use voltage and frequency scaling to boost the performance of the recovery process. This can be most efficiently done in a GALS system, where processors are independently clocked. In such a system only the processors that actually perform a recovery are temporarily operated with a higher frequency. This approach is much more energy-efficient compared to increasing the clock frequency in a globally synchronous system, where the full system would run at higher speed.

Since this approach only requires duplication of processing resources, it is clearly more area and power efficient than a TMR system, and still achieves comparable performance. Due to frequency scaling during re-executions and the fast checkpointing process, traditional performance penalties of rollback recovery strategies can be compensated. This efficiency, however, can only be achieved due to restriction to a simple model of computation for data-driven DSP applications. A weak point in the proposed architecture, mentioned in [124], are the local clock generators of the GALS system. Unlike in our own work, clock generators are not replicated, i.e., the clock for redundant processors, running the same task, is supplied from a single source. A transient fault in the shared clock generator can therefore still cause a system failure.

Yu & McCluskey

The roll-forward recovery scheme we used for protecting modular redundant GALS modules against transient faults was greatly influenced by a similar approach introduced by Yu & McCluskey [123]. A basic block diagram of their TMR system can be seen in Figure 4.33a. Like in our solution, they propose to trigger the state restoration process at pre-scheduled checkpoints. These checkpoints can be introduced into the execution as dedicated states in circuit's state machine, or using a cycle counter with a predefined period. Consequently, the *checkpoint recovery logic* is connected to the registers of the module in order to evaluate the current state. In contrast to our approach, the recovery logic only starts state restoration, if an output error is reported by an *error detector*. Clearly, latent errors in one of the modules will not be recovered with this strategy. The implementation of the error detector is shown in Figure 4.33b. It compares the outputs of module copies against the voted output value to identify a faulty copy.

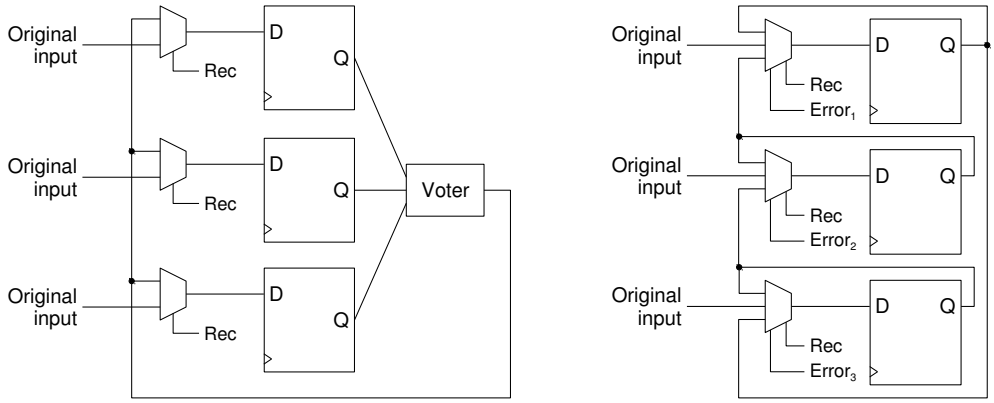


(a) Concept block diagram.

(b) Voter with error detector.

Figure 4.33: Roll-forward recovery scheme for TMR systems (cf. [123]).

For the *state restoration* block two implementation schemes are proposed. The first one, which we also used in our parallel recovery approach, employs a voter and a feedback signal to restore erroneous values in registers (see Figure 4.34a). The second scheme, named *direct-load scheme* avoids voters and directly restores the state of a faulty module from another copy, which is known to be correct. This can be done because the error detector is able to identify a faulty module and thereby controls the multiplexers at the input side of the circuit's flip-flops (see Figure 4.34b). If a module is diagnosed to be fault-free, the local flip-flop values are maintained using feedback signals, otherwise the multiplexers switch to the state signals of another (fault-free) module copy. Clearly, this restoration process only works, if errors are confined to a single module copy. The voting scheme, on the other hand, is capable to recover multiple errors in more than one module copy as long as different flip-flops are affected. Nevertheless, Yu & McCluskey argue that the direct-load scheme is a bit more area efficient, due to the saved voters.



(a) Restoration with voters. (b) Direct-load scheme.

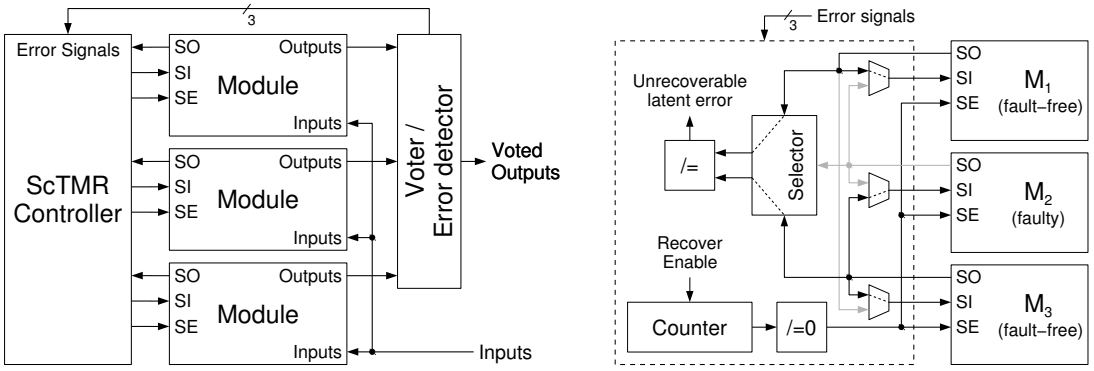
Figure 4.34: Two state restoration schemes (cf. [123]).

Even though the basic recovery concept is quite similar to the approaches we have presented in this chapter, the key element of our work is to employ these restoration schemes in a GALS-based circuit architecture. Temporal independence, achieved by asynchronous design techniques, allows redundant module copies to run even with non-related clock signals, whereas conventional voting or direct-load restoration schemes strictly depend on tight global synchronization. Another important asset of our concept is the systematic avoidance of single points of failures. In contrast, for Yu & McCluskey's approach critical resources like voters, the error detector, as well as checkpoint and restoration logic are not replicated. In [123] they point out that faults in these shared circuit parts cannot be tolerated, unless they are protected using other fault tolerance mechanisms.

Ebrahimi et al.

Another recovery technique for TMR systems, called *ScTMR*, is presented in [27]. It is very similar to the method by Yu & McCluskey since state restoration is again performed with the

direct-load scheme, and a voter/error detector is used to mask faulty module outputs and identify the faulty module. The new idea of ScTMR is to employ scan chains to copy the state of fault-free modules to the faulty one. Figure 4.35a shows a block diagram of the TMR system, consisting of three module copies, the voter/error detector and a recovery logic block, named *ScTMR controller*. A close-up of the ScTMR controller can be seen in Figure 4.35b. It shows a particular situation during recovery operation, where the middle module M_2 is faulty, while the top and bottom modules M_1 and M_2 are fault-free. As can be seen, the multiplexers in the ScTMR controller, controlled by the output signals of the error detector, are configured so that the scan chain inputs of the two fault-free modules are directly driven by the respective module's scan chain output. Only the scan chain input of the faulty module M_2 is fed with data from another module, namely M_3. A counter, which is preset to the length of the scan chains at the beginning of the recovery operation, is decremented with every clock cycle and thereby controls the duration of the recovery mode.



(a) Architecture. (b) Recovery Controller[12].

Figure 4.35: ScTMR recovery technique.

Since state restoration is only performed when a faulty module output is detected, latent errors might survive unnoticed for extended periods of time. A second error in a different module, which triggers a recovery operation at a later point in time, can then lead to the propagation of the latent error to another module, and ultimately to a system failure. As discussed in the above section about Yu & McCluskey's approach, the direct-load scheme is not capable of correcting errors in multiple modules, even if they occur in different flip-flops. Ebrahimi et al. therefore included some logic in the ScTMR controller to detect latent errors. A selector component forwards the scan chain output of the two modules that are believed to be fault-free to a comparator (see Figure 4.35b). If the comparator finds a disagreement of the scan chain data during the recovery operation, a latent error has been detected and a global error signal is raised.

In our serial recovery approach we also employed a restoration scheme based on scan chains. Due to the limitations of the direct-load scheme, however, we have decided to apply a majority vote on every data bit, which is read from scan chains. Thus, multiple errors in different module copies can be resolved in our approach, as long as different flip-flops are affected. Like in Yu

[12]Connections of error signals, which control selector and multiplexer components, are omitted in the figure.

& McCluskey's technique, an ScTMR system is limited to synchronized module copies, which are operated in the same clock domain. Furthermore, common system parts, like the ScTMR controller, are not replicated and therefore remain a single point of failure.

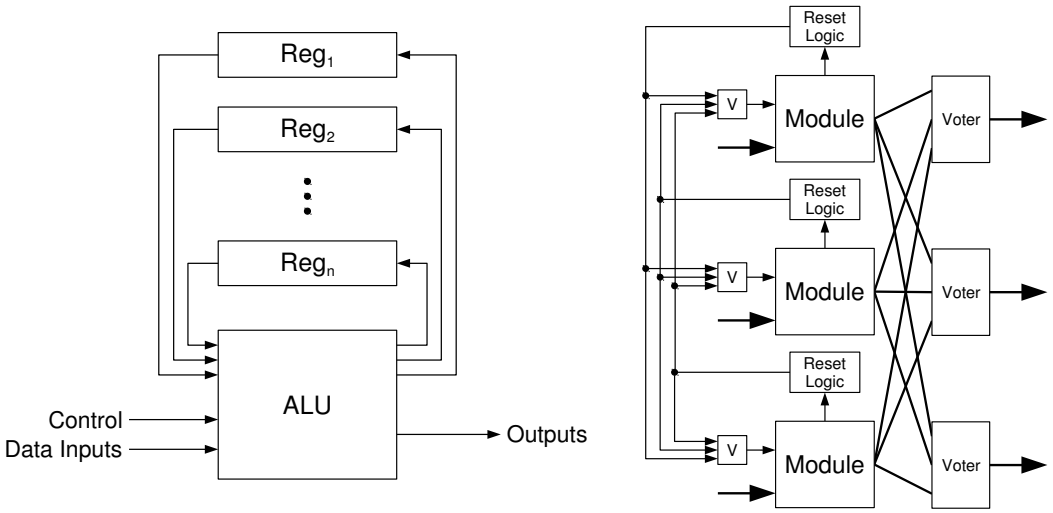
In a recently published journal paper [28], Ebrahimi et al. present an extension, called *SMERTMR*, which addresses the issue of latent errors and the accumulation of multiple errors in different module copies until the next recovery is started. To fight latent errors they introduced checkpoints to start a recovery operation in addition to the event-triggered method when errors are detected at the modules' outputs. At the beginning of the recovery a new comparison step is proposed, which reads out state information from all scan chains, and compares data from different module copies bit by bit. For every pair of two module copies a counter is implemented, which is increased when a mismatch is found between the two corresponding modules. By analysing the counter values after the comparison step, faulty and fault-free module copies can be identified. In case of multiple faulty copies, however, it has to be assumed that different flip-flops are affected. Otherwise the identification algorithm cannot reliably distinguish between faulty and non-faulty modules. During the comparison step multiplexers for scan chain signals (cf. Figure 4.35b) are configured such that a scan chain input of a module is directly connected to the SO signal of the same copy. After faulty/non-faulty copies have been identified state restoration is performed, and the multiplexer connections are rearranged to connect the scan chain inputs of *all* copies to the scan chain output of one fault-free module copy.

Since SMERTMR and our serial recovery approach have quite similar error correction capabilities, we need to make a detailed comparison for different error scenarios. *Case 1*: If errors are confined to a single module copy, both techniques are able to perform a successful recovery. *Case 2*: If errors occur in multiple copies, we need to distinguish three sub cases, depending on the exact location of errors. Assume a TMR system were the state of two replicas M_i and M_j is compromised. Let EFF_i and EFF_j be the sets of erroneous flip-flops of M_i and M_j, respectively. *Case 2.1*: $EFF_i \cap EFF_j = \emptyset$. Both techniques are able to perform a successful recovery. In case of SMERTMR M_i and M_j are identified as faulty copies, and the state is restored from the remaining fault-free replica. Voting in our approach is also successful, since for every state bit there are at least two flip-flips with the correct value. *Case 2.2*: $EFF_i \cap EFF_j \neq \emptyset \wedge EFF_i \neq EFF_j$. SMERTMR detects an unrecoverable condition as it cannot unambiguously identify the faulty module copies. The recovery operation is therefore aborted. In case of our approach, voting will lead to wrong results for commonly erroneous flip-flips in $EFF_i \cap EFF_j$. This can potentially result in a system failure. *Case 2.3*: $EFF_i \cap EFF_j \neq \emptyset \wedge EFF_i = EFF_j$. In case of SMERTMR copies M_i and M_j are incorrectly identified to be the fault-free replicas. For our approach this case is not different from Case 2.2. Hence, both solutions incorrectly perform state restoration, potentially causing a system failure.

While latent errors are certainly an important issue to address, we believe it is a more sensible strategy to avoid the occurrence of multiple errors between successive recovery operations. To reduce the probability of such a scenario to a negligible level, the checkpoint frequency needs to be adjusted appropriately. Even though SMERTMR is an interesting technique, we prefer our solution since it is simpler and faster (scan chains only need to be read once). Again, SMERTMR can only be applied to module copies synchronized to the same clock. This limitation, which we were able to overcome in our GALS-based designs, is briefly addressed by Ebrahimi et al. [28].

Wakerly

In Section 3.3.1 we already introduced Wakely's concepts on restorable sequential machines for modular redundant systems. In [118] he discussed two other interesting state restoration schemes. A high-level recovery approach is discussed on the example of a register and arithmetic logic unit (RALU), as can be seen in Figure 4.36a. This RALU can perform arithmetic and logical operations on its inputs and internal registers. Clearly, if such a module was to be replicated, upsets in the internal registers need to be recovered. A simple way to do this is to use the *load* instructions provided by such an RALU to overwrite erroneous values stored in the registers. Assume, e.g., an RALU is used in a processor, which runs some kind of operating system. The operating system could be written such that frequently executed routines include sequences that periodically load all registers with some predefined value.



(a) Register ALU.

(b) State recovery with reset logic.

Figure 4.36: Examples for restorable sequential machines.

Furthermore, Wakerly presents a circuit-level technique based on reset inputs, usually only used for initialisation purposes, to return a circuit to back to a well-defined correct state. Assume that a certain state Q_r is visited on a regular basis during the runtime of a sequential circuit. Then a combinational function can be designed, which identifies the predecessors of this state, and enables a reset signal, once the circuit is supposed to enter Q_r. If majority voting is performed on these reset signals, as can be seen in Figure 4.36b, correct units of the modular redundant system will force a faulty member into a reset, thereby scrubbing a potentially erroneous state.

CHAPTER 5

Robust Asynchronous Inter-Module Communication Channels

After presenting techniques for building fault-tolerant GALS modules in the last chapter, we will now focus on reliability issues of asynchronous data channels for inter-module communication in a GALS system. Since communication channels and computational blocks have a fundamentally different structure and function, the selected mechanisms for protection against faults are usually different as well. For communication interfaces probably the most popular choice is the use of information redundancy. More or less sophisticated error detecting and correcting codes can achieve a much better efficiency compared to mere hardware redundancy. Consider, e.g., the TMR mechanisms we implemented in the last chapter. If this technique is applied to a k-bit wide communication bus, the total number of wires required is $3k$, i.e., an increase by $2k$ wires. This solution is able to detect two faulty bits and correct/mask one faulty bit. The same error detection and correction capability can also be achieved with a Hamming code, where redundant check bits scale logarithmically with data bits, i.e., an increase by only $log_2(k)$ wires.

In this chapter we will therefore investigate the strengths of error detecting (ED) codes to secure GALS communication. The interesting research question we pursued in this context is how these codes can be efficiently combined with delay-insensitive codes. Delay insensitivity can be a very desirable property for global inter-module communication links, especially when considering modern ultra-deep submicron process technologies, which suffer from significant timing uncertainties due to process variability. For GALS modules that are linked over long global interconnect routes signal delays can vary significantly. Usually such interconnects are not simply a metal wire but include vias to cross between different layers for routability, and frequently buffers or pipeline stages [110] are added to decrease wire delays and/or increase the throughput of the interconnect link. Even more complex interconnects can be found in reconfigurable architectures like FPGAs, where connections are routed over switch boxes. These switch boxes can be configured by the programmer in order to connect different circuit parts as required by the implemented circuit design. While buffers, pipeline stages and switch boxes help to increase performance and flexibility, they also make interconnects susceptible to radiation-induced

faults and contribute to delay variations. Using delay-insensitive communication mechanisms, global timing constraints between GALS modules can be completely avoided.

While we considered permanent defects in the last chapter to some extent, our work on communication channels solely applies for transient faults. Consequently, all proposed fault tolerance mechanisms rely on the fact that faults vanish after a certain amount of time and that delay-insensitive interfaces are elastic, i.e., an incoming data transmission can be deferred until the fault has been resolved. In Section 5.1 we will first formally analyse delay-insensitive codes and determine how they can be efficiently reinforced with ED codes to make transmission faults detectable. Based on these fundamental findings we will then present two hardware implementations of sender and receiver components for fault-tolerant delay-insensitive channels. The first approach is a specific solution that works for 4-phase dual-rail codes, whereas in Section 5.3 a generic mechanism is presented that is suited for both 4-phase and 2-phase protocols and matching combinations of DI and ED codes. Both link architectures have been published in conference papers, the first in [62] and the second in [63].

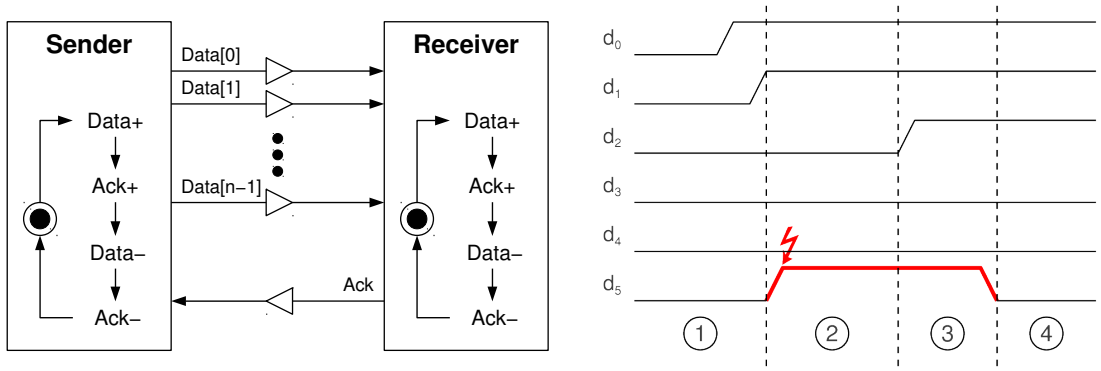
5.1 Delay-Insensitive Fault-Tolerant Codes

5.1.1 Problem Description: Transmission Faults

Since delay-insensitive communication mechanisms entail the freedom of timing assumptions for signal delays, transitions on individual wires of a bus signal, transmitting a multi-bit data word, may appear at any time in any order at the receiver. As discussed in Section 3.1.4, special codes have to be used to support delay-insensitivity. These codes allow completion detection, i.e., enable the receiver to tell when the *last* wire has made a transition. The property that a single transition, if considered as the last transition, completes the data transfer is what makes delay-insensitive codes prone to faults. Consider the scenario where a fault changes the signal value of a wrong wire so that a complete yet erroneous input word is presented to the receiver.

Figure 5.1a shows the basic model of the communication links we consider in this section. As can be seen from the STGs depicted within the sender and receiver components, we restrict our discussions to 4-phase asynchronous protocols. Even though some of the techniques we present later on in this section could be extended to 2-phase protocols, the omission of a reset phase still adds significant complexity and therefore 2-phase protocols are out of scope for this work. The interconnect signals comprise data wires driven by the sender and an acknowledge signal, which is controlled by the receiver. On all the wires we have inserted buffer cells as an example for interconnect components that are susceptible to external faults. Note that throughout the whole chapter we only consider faults that occur in one of these interconnect components. The sender and receiver components are assumed to operate error-free, in particular it is assumed that they execute the designated interface protocols flawlessly. This restriction allows us to focus our work purely on communication faults and the codes as well as the encoder and decoder components that are required for their mitigation.

To reason about transmission faults we need to consider the *location* and the *time* of their occurrence. The location can either be on any of the data signals or the acknowledge signal. For the remainder of this chapter we will focus on data signals since this involves interesting



(a) Unreliable DI link.

(b) Faulty completion of data transmission.

Figure 5.1: Transmission faults on a delay-insensitive communication link.

research problems regarding DI and ED codes. Faulty transitions on the acknowledge wire obviously can break the handshake protocol by prematurely causing the sender component to advance its protocol state machine. However, mitigation of faults on a single control signal that is routed over unreliable interconnect resources is a rather trivial problem. Without resorting to any timing assumptions a simple solution is to duplicate the acknowledge signal and let the sender wait for proper transitions on both of the acknowledge wires, e.g., using a C-element. With this approach a transient fault in one of the duplicated acknowledge signals can be masked.

Whether a fault on the data signals will be masked or can actually cause an error at the receiver, depends on the specific timing of the fault. Figure 5.1b illustrates a problematic fault occurence. The waveform shows the transmission of the codeword 111000 of a 3-of-6 code. The transitions on rails d_0 and d_1 are correct, however, before the final correct transition on rail d_2 arrives, a faults strikes on rail d_5. At this point a valid 3-of-6 codeword is formed: 110001. Obviously, this is not what the sender wants to transmit, but without redundant information the receiver cannot be aware of this. The receiver's completion detection circuits will be triggered and an erroneous input word will be processed. In the waveform in Figure 5.1b we have highlighted four interesting sections: Section 1 shows the beginning of the data phase, where the transmitted codeword is still incomplete. In Section 2 the faulty transition completes the codeword. This is then followed by Section 3, where the missing correct transition appears and the faulty signal state of rail d_5 is still in place[1]. At this point in time the receiver can actually tell that something went wrong since an input pattern with four active rails is clearly a non-codeword in a 3-of-6 code. Finally the fault vanishes in Section 4 and the correct codeword appears. In conclusion we can see that Section 2 is critical and we need to add some degree of redundancy that allows the receiver to discard this intermediate codeword.

[1]Note that a short fault might also vanish before the final correct transition is received. In this case the receiver would observe an incomplete codeword again, like in Section 1.

5.1.2 Formal Prerequisites

Since transitions on data wires can occur in any order at the receiver, a DI code must guarantee that no intermediate bit pattern can be mistaken for a valid codeword. As said above, only when the last bit changes, i.e., the last transition occurs, the received bit pattern forms a valid codeword. This is the case, if no codeword *covers* another valid codeword.

Definition 1. *Covering codewords: A codeword* $x = (x_{n-1}, x_{n-2}, \ldots, x_0)$ *covers another codeword* $y = (y_{n-1}, y_{n-2}, \ldots, y_0)$ *iff for all* i, $y_i = 1$ *implies* $x_i = 1$. *This covering relation is denoted as* $y \leq x$.

Codes, which are built of codewords that do not cover any other codeword, are called *unordered* [12, 115]. All codes used for delay-insensitive data transmission, e.g., *m-of-n* codes, *Berger and Sperner* codes, are unordered. Formally, unordered codes are defined as follows:

Definition 2. *Unordered codewords/codes: Two codewords* x *and* y *are called unordered, iff* $x \not\leq y$ *and* $y \not\leq x$. *Two sets of codewords* X *and* Y *are called unordered, iff for each* $x \in X$ *and* $y \in Y$: $x \not\leq y$ *and* $y \not\leq x$.

To be able to reason about the resilience of delay-insensitive codes against faulty transitions we propose a generalisation of the concept of covering codewords:

Definition 3. *Overlapping codewords: A codeword* $x = (x_{n-1}, x_{n-2}, \ldots, x_0)$ *overlaps with another codeword* $y = (y_{n-1}, y_{n-2}, \ldots, y_0)$ *iff there exists a bit position* i, *where* $y_i = 1$ *implies* $x_i = 1$. *This relation is reflexive, symmetric but not necessarily transitive. We denote the number of overlapping bit positions* i, *i.e., where* $y_i = 1$ *and* $x_i = 1$, *as* $c(x, y)$. *The number of bit positions of* x *that do not overlap with* y *can then be calculated by subtracting* $c(x, y)$ *from the Hamming weight of* x. *We write:* $u(x, y) = weight(x) - c(x, y)$.

Based on this definition, we now can analyse the minimum number of faults that are required for the receiver to confuse two codewords during a data transmission. Let x, y be two codewords of an unordered code C, and let us assume that y is transmitted to the receiver. Following Definition 3, $u(x, y)$ faults are sufficient to turn an intermediate transmission pattern of y into x. Recall the example of the 3-of-6 code from the previous section. Let x be the codeword 110001 and y be 111000. The number of bit positions of x that do not overlap with y is one, i.e., $u(x, y) = 1$. As we have seen in the previous section, a single fault is sufficient to turn the intermediate pattern 110000 of codeword y into codeword x.

5.1.3 Building Subcodes

In order to avoid the possibility of confusing codewords in the presence of faults, one solution is to make sure that $u(x, y) > f$ and $u(y, x) > f$ for each $x, y \in C$. If this property holds, then an incomplete codeword will always be invalid even in case of f faults. Starting with an arbitrary delay-insensitive code, it is possible to construct a fault-tolerant subcode by removing codeword pairs that violate the above condition. Note that a subcode of a DI code is again a DI code [115]. Consider, e.g., the non-overlapping bits of codeword pairs of a 2-of-4 code

(Table 5.1). If we assume $f = 1$, i.e., single-bit faults, only pairs of codewords with $u(x, y) > 1$ and $u(y, x) > 1$ can be chosen. A maximally-sized set of codewords satisfying this requirement is, e.g., $\{0011, 1100\}$. Thus, the number of codewords is reduced from six to two, if single-bit faults have to be tolerated.

Table 5.1: Count of non-overlapping bits of a 2-of-4 code.

$u(x, y)$	0011	0110	1100	0101	1010	1001
0011	0	1	2	1	1	1
0110	1	0	1	1	1	2
1100	2	1	0	1	1	1
0101	1	1	1	0	2	1
1010	1	1	1	2	0	1
1001	1	2	1	1	1	0

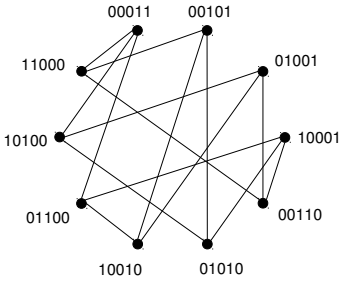


Figure 5.2: Safe overlap graph.

In order to get the highest possible coding efficiency, the aim is to find subcodes with maximum size, i.e., codes that consist of a maximally-sized set of codewords that can not be confused in the presence of faults. This can be done by modelling the overlap relationships between the codewords of the DI code in an undirected graph, which we call the *safe overlap graph*. The vertices of this graph represent the codewords, and edges are added between every pair of codewords that cannot be confused even in case of f faults. Figure 5.2 shows the graph for a 2-of-5 code and for $f = 1$. The safe overlap graph $G = (V, E)$ is formally defined as follows:

$$V := C; E = \{(x, y) | x, y \in C, u(x, y) > f, u(y, x) > f\}$$

A subcode of maximum size can now be derived by searching the largest subgraph S of G that is fully connected, i.e., where there is an edge between every pair of nodes. Fully connected subgraphs are called *cliques* and the problem of finding a *maximum clique* is well-known in graph theory. Since S is fully connected, there is no pair of codewords in S where the number of non-overlapping bits is too small to tolerate f faults. The delay-insensitive subcode $C_{sub} = V(S)$ therefore is able to mitigate up to f faults, which occur during the transmission.

Tables 5.2 and 5.3 show fault-tolerant subcodes we have derived for m-of-n codes and Berger codes, two well-known types of DI codes. For each code we have generated two safe overlap graphs: one for single faults ($f = 1$) and one for double faults ($f = 2$). For finding maximum cliques in these graphs we used the tool *Cliquer* from the University of Helsinki [81]. Note that the maximum clique problem is NP-hard and finding exact solutions can take a significant amount of time in case of large graphs. As can be seen in Tables 5.2 and 5.3, for some subcodes the maximum clique only contains one codeword. Obviously these subcodes cannot be used to transmit data. An analysis of the resulting subcodes is presented in Tables 5.5a and 5.5b. Comparing the numbers in the second column (original code size) with those in the fifth column (subcode size), it can be seen that the number of codewords is drastically reduced when single faults or even double faults should be tolerable. The reduction of codewords obviously results in a smaller number of data bits that can be encoded. The last column of the tables shows the information rate for each subcode, which is defined as the ratio k/n, where $k = \lfloor log_2(\text{subcode size}) \rfloor$

is the number of encoded data bits and n is the bit length of the codewords. As can be seen, the reduction of codewords unfortunately leads to very poor information rates.

Table 5.2: Fault-tolerant subcodes of m-of-n codes.

Code	Faults	Codewords
1-of-2	1	10
	2	10
1-of-4	1	1000
	2	1000
2-of-4	1	1001, 0110
	2	1100
2-of-5	1	10010, 01100
	2	11000
3-of-6	1	001101, 110001, 010110, 101010
	2	110001, 001110
2-of-7	1	0001001, 1000100, 0110000
	2	1100000

Table 5.3: Fault-tolerant subcodes of Berger codes.

n	Faults	Codewords
2	1	1100
	2	1100
3	1	01101, 10010
	2	11100
4	1	1000011, 1111000
	2	1111000
5	1	00011011, 01110010, 10100011, 11001010
	2	01110010, 10001011
6	1	000111011, 011001011, 011110010, 101011010, 101100011, 110010011, 110101010
	2	101110010, 110001011
7	1	0001111011, 0010010101, 0011001100, 0100001101, 0101010100, 0110111010, 0111100011, 1000011100, 1001000101, 1010101011, 1011110010, 1100110011, 1101101010, 1110000100
	2	1101110010, 1110001011
8	1	001010100101, 001101110011, 001111000100, 010011110011, 010100110100, 010110000101, 011000010101, 011001100100, 100001010101, 100011100100, 100110110011, 101000110100, 101100000101, 110000100101, 110010010100, 111011000011, 111101010010, 111110100010
	2	010000110101, 011110000100, 100101100100, 101011010011

5.1.4 Combining DI and ED Codes

Another option is to accept that certain codewords of a delay-insensitive code can be confused due to faults and to try to detect these confusions rather than preventing them by reducing the

Table 5.4: Evaluation of fault-tolerant subcodes.

Code	Original code size	#Wires/codeword	#Faults	Subcode size	#Databits/codeword	Information rate
1-of-2	2	2	1	1	0	0
			2	1	0	0
1-of-4	4	4	1	1	0	0
			2	1	0	0
2-of-4	6	4	1	2	1	0.25
			2	1	0	0
2-of-5	10	5	1	2	1	0.2
			2	1	0	0
3-of-6	20	6	1	4	2	0.333
			2	2	1	0.167
2-of-7	21	7	1	3	1	0.143
			2	1	0	0

(a) Subcodes of m-of-n codes.

Original data field size	Original code size	#Wires/codeword	#Faults	Subcode size	#Databits/codeword	Information rate
2	4	4	1	1	0	0
			2	1	0	0
3	8	5	1	2	1	0.2
			2	1	0	0
4	16	7	1	2	1	0.143
			2	1	0	0
5	32	8	1	4	2	0.25
			2	2	1	0.125
6	64	9	1	7	2	0.222
			2	2	1	0.111
7	128	10	1	14	3	0.3
			2	2	1	0.1
8	256	12	1	18	4	0.333
			2	4	2	0.167

(b) Subcodes of Berger codes.

code set. To be able to perform this detection, redundant information needs to be added to the transmitted data. Therefore, we propose that the sender encodes a data word in two steps: First an error detecting code is applied, which adds the required redundancy. The result of this encoding step is called *ED codeword* in this section. Then, in a second step, the ED codeword is encoded with a delay-insensitive code to guarantee for correct transmission even in case of varying signal delays. We will use the term *DI codeword* to refer to the output of this step.

The receiver then works as follows: In a fault-free transmission the receiver will eventually observe a complete DI codeword, which can be decoded. After checking the absence of errors in the retrieved ED codeword and decoding the correct data word, its reception can be acknowledged to the sender. On the other hand, if faults occur during the transmission, the behaviour of the receiver becomes more complex. Note that from now on we will call faults that change bits in DI codewords *transmission faults*. Two cases can be distinguished:

1. Transmission faults can transform the incoming DI codeword into a non-codeword (with respect to the used DI code). Obviously, such faults will prevent completion detection and stall the receiver. However, if faults are assumed to be transient, a complete and correct DI codeword will eventually be observed and the receiver can then continue its operation like in a fault-free transmission.

2. As we have discussed above, faults can also transform an incomplete intermediate DI codeword x into a complete DI codeword y, $y \neq x$. In this case the completion detection

will indicate the availability of a new DI codeword, which will then be decoded. Due to the transmission fault(s), however, the received ED codeword will be different from the original one generated by the sender: One or more bits will be erroneous. We will call these bit-flips *decode errors*. If the number of these errors is smaller than or equal to the detection capabilities of the used ED code, the receiver can tell that a transmission fault has occurred and can wait for the transient fault to disappear. Eventually the correct DI codeword will be perceived and the transmission can then be completed.

Hence, the receiver performs both completion detection *and* error detection before a new data word is processed and acknowledged. This is the key element of the proposed approach. The principal question, which now remains to be answered, is how much redundancy needs to be included in the ED codeword in order to be able to detect all possible decode errors. Assuming a certain maximum number of transmission faults f we obviously want to minimise the required overheads and efficiently use the available redundancy of both the delay-insensitive code and the error detecting code. We have therefore systematically analysed m-of-n and Berger codes again to find out what minimum capabilities an error detecting code needs to provide to tolerate a certain number of transmission faults.

m-of-n Codes

Figure 5.3 shows the encoding steps for m-of-n codes. First the error detecting code is applied to the original data word. Subsequently the resulting codeword is partitioned into equally-sized blocks, which then are translated into codewords of the used m-of-n code. The block length k depends on the number of bits that can be encoded by this m-of-n code.

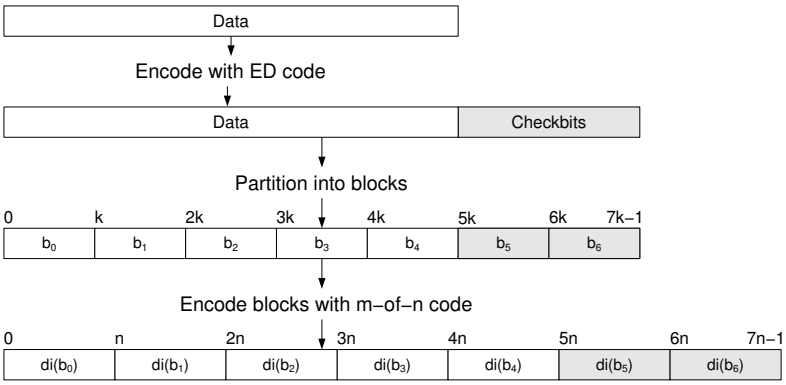


Figure 5.3: Encoding steps for m-of-n code.

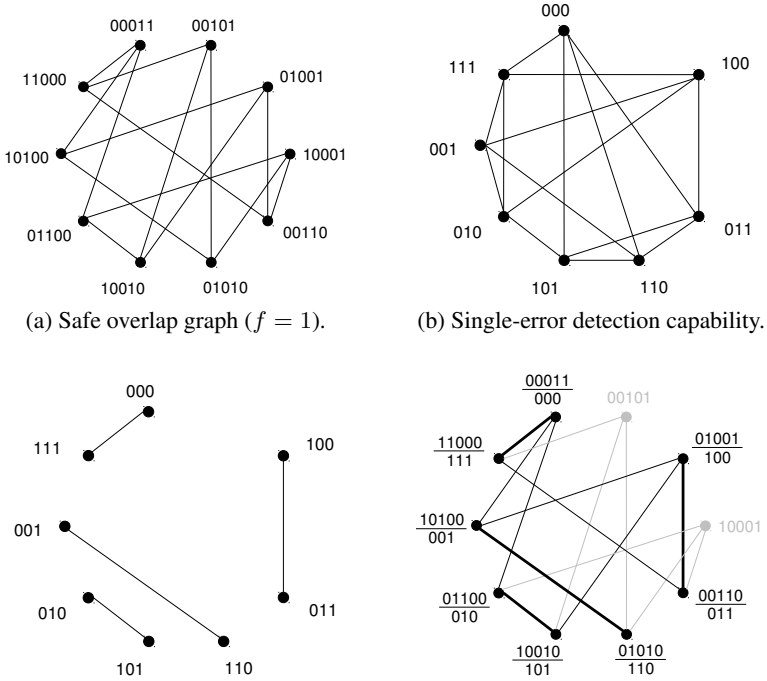
An encode function is required to define how data blocks are transformed into m-of-n codewords (cf. function *di* in Figure 5.3). In general, m-of-n codes are not systematic, i.e., the input bit vector is not embedded as a part of the codeword. Thus, with respect to data transmission, every function that assigns bit vectors to codewords is a suitable encode function. With respect to fault tolerance, however, the choice of this function is crucial for the required redundancy of the error detecting code and therefore for the efficiency of the overall data encoding. This

can be easily illustrated with the following example: Consider again the codewords 111000 and 110001 of a 3-of-6 DI code. As discussed above, a single fault can lead to the confusion of these two codewords. Now let us assume that 111000 encodes the bit vector 0000 and 110001 is the DI encoding of 1111. In this case a single transmission fault in the DI codeword can lead to a quadruple error in the ED codeword. Therefore, the used error detecting code would need to be able to detect four erroneous bits. If we want to minimise the overhead imposed by the error detecting code, we need to find an optimal mapping, where the maximum possible number of decode errors is minimal. More specifically, for all DI codewords that can be confused in case of a certain number of transmission faults, the Hamming distance of the encoded bit vectors should be minimal. For DI codewords that can *not* be confused in the given fault hypothesis, the mapping is uncritical and will not affect the choice of the error detecting code.

The problem of finding an optimal mapping for a specific *m-of-n* code and a specific number of transmission faults, can again be described as a problem in graph theory. First, we characterise the DI code under investigation with its safe overlap graph. Furthermore, we need to model the properties of a candidate ED code as graph (we call this the *ED graph*). This is done by building a graph whose vertices represent all bit vectors from the set $\{0,1\}^k$, i.e., the set of words that can be encoded in a single block by the given DI code. Edges then are drawn between those words where the candidate ED code would *not* be able to identify an error, if the two bit patterns were exchanged in a block of the ED codeword. Having built these two graphs, we now can try to find a subgraph in the safe overlap graph that is isomorphic to the ED graph. In graph theory this is called the *subgraph isomorphism problem* [112]. If such a subgraph exists, then a mapping from ED blocks to DI codewords can be derived, where all possible decode errors are detectable. Recall that in the safe overlap graph there is an edge between DI codewords that cannot be confused by transmission faults. Finding an isomorphic subgraph means that we can assign all problematic pairs of bit patterns to DI codewords that will not be confused anyway.

Figure 5.4 illustrates our approach on the example of a 2-of-5 code for single transmission faults ($f = 1$). The left-most figure shows the safe overlap graph. Figure 5.4b models problematic bit vectors in case a code with single error-detecting capabilities is used, i.e., word pairs with a Hamming distance greater than or equal to 2. Figure 5.4b shows the same graph for an ED code that can detect double errors, but can *not* detect triple errors. Thus, word pairs with a Hamming distance of 3 are connected with an edge. You can now easily see that it is not possible to find a subgraph in the the safe overlap graph that is isomorphic to the ED graph in Figure 5.4b. Thus, a single-error detecting code is not sufficient to secure the delay-insensitive data transfer with a 2-of-5 code against single transmission faults. For a double-error detecting code, on the other hand, the desired isomorphic subgraph can be found (see Figure 5.4d). The respective mapping of bit vectors to DI codewords can then be derived by simply assigning the labels of the ED graph in Figure 5.4c to the labels of the found subgraph.

We repeated this evaluation for different m-of-n codes, as can be seen in Table 5.5. After generating the safe overlap graphs for $f = 1$ and $f = 2$ as well as all ED graphs, we used the tool *LAD* [101] to find the desired isomorphic subgraphs. Note that for double faults two cases need to be considered: 1) Both faults occur in the same DI codeword. 2) The faults occur in different codewords. The strength of the used ED code must then be adjusted to the maximum number of decode errors that can occur in these two cases. Table 5.5 lists the most efficient



(a) Safe overlap graph ($f = 1$).

(b) Single-error detection capability.

(c) Double-error detection capability.

(d) Possible solution.

Figure 5.4: Finding the optimal error-detecting code in order to secure a 2-of-5 code.

Table 5.5: Finding optimal error-detecting codes for m-of-n codes.

	$f = 1$			$f = 2$		
	General	Odd error count	AUED	General	Odd error count	AUED
1-of-2	1-bit ED	✓	✓	2-bit ED	✓	✓
1-of-4	2-bit ED	-	-	4-bit ED	-	-
2-of-4	1-bit ED	✓	✓	2-bit ED	-	-
2-of-5	2-bit ED	-	-	4-bit ED	-	-
3-of-6	2-bit ED	-	-	4-bit ED	-	-
2-of-7	3-bit ED	-	-	6-bit ED	-	-

classes of ED codes we have proven to be strong enough for the mitigation of single and double transmission faults. We have grouped ED codes into three classes: 1) Codes that can detect a certain number of bit errors (single bit errors, double bit errors, etc.). The column named "General" lists the minimum number of bit errors that a code needs to be able to detect, if the most optimal DI encode function is used. 2) The second class of ED codes we tested contains codes that can detect odd numbers of bit errors. 3) Furthermore we have analysed codes capable of detecting unidirectional errors (AUED codes). For the latter two classes a check mark in Table 5.5 indicates that a code suitable to be paired with the respective DI code.

Based on our results on the required error detection capabilities, we analysed the efficiency

of the combination of these DI codes with specific ED codes for different data word sizes. We investigated the use of single parity bits, which is the most efficient protection against single errors, as well as *Hamming codes* and *extended Hamming codes* for providing double and triple error detection capabilities, respectively. Table 5.6 shows the results of our analysis for single transmission faults ($f = 1$). In the last three columns the total number of wires that are required for transmitting the data word, the resulting information rate and the required signal transitions per bit can be seen. While the number of wires and the information rate are important indicators for the area complexity of the analysed codes, the number of signal transitions provides valuable insights on their power efficiency. It can be seen that 2-of-5 and 3-of-6 have the best scores regarding area, while 1-of-4 and 2-of-7 would be the best choices for highly energy-efficient data transmissions. It can also be seen that all numbers improve for all codes with rising word sizes. This can be easily explained with the overheads of the used ED codes, which are either constant (single parity) or scale logarithmically (Hamming codes) with the word size.

Berger Codes

In case of Berger codes the mapping of bit vectors to DI codewords is defined by the construction of the code itself. Recall that Berger codes simply extend the input word with a synchronization field, which represents the number of zeros contained in this word. Let us consider a Berger code C_k with a data field length of k. To find the most efficient error detecting code that can be combined with C_k for mitigating a certain number of transmission faults f, the maximum number of decode errors t needs to be computed. This can simply be done by iterating over all DI codewords that can be confused in the presence of faults and by finding the maximum Hamming distance of the encoded bit vectors. Let $x, y \in \{0,1\}^k$ be two bit vectors of length k and $x_s, y_s \in C$ denote the respective DI codewords. Formally, t can be defined as follows:

$$t = max\{d(x,y) \mid x \neq y, u(x_s, y_s) \leq f \vee u(y_s, x_s) \leq f\} \tag{5.1}$$

We have analysed Berger codes with a data field, whose length ranges from 2 to 8 bits. Table 5.7 shows the results for single and double transmission faults. As can be seen, even for small data word lengths, a high number of decode errors can occur, which is in some cases even equal to the number of data bits.

Theorem 2. *The number of decode errors t in Berger codes is monotonically increasing with the data word size n.*

Proof. From Table 5.7 we can infer that the theorem holds for small word sizes. Assume that the transmission of codewords of a Berger code C_k for k-bit data words can result in a maximum of t decode errors in case of f transmission faults. Let x and y be two codewords of C_k whose data fields have a Hamming distance of t and which can be confused due to transmission faults, i.e., $u(x,y) \leq f \vee u(y,x) \leq f$. We can then investigate a Berger code with a word size of $k+1$. This code includes the codewords $x' = 1x$, $y' = 1y$, whose data fields still have a Hamming distance of t. Since $u(x',y') = u(x,y)$ and $u(y',x') = u(y,x)$, x' and y' can again be confused in the presence of f transmission faults, which then leads to a t-bit decode error. □

Table 5.6: Analysis of m-of-n codes extended with suitable ED codes ($f = 1$).

Data word size	Code	#Wires/codeword	#Databits/codeword	#Transitions/codeword	Required detection capabilities	ED code	#Checkbits	Block count	Total number of wires	Information rate	#Wire transitions/bit
2	1-of-2	2	1	2	1-bit	Single Parity	1	3	6	0.333	3
	1-of-4	4	2	2	2-bit	Hamming	3	3	12	0.167	3
	2-of-4	4	2	4	1-bit	Single Parity	1	2	8	0.250	4
	2-of-5	5	3	4	2-bit	Hamming	3	2	10	0.200	4
	3-of-6	6	4	6	2-bit	Hamming	3	2	12	0.167	6
	2-of-7	7	4	4	3-bit	Extended Hamming	4	2	14	0.143	4
4	1-of-2	2	1	2	1-bit	Single Parity	1	5	10	0.400	2.5
	1-of-4	4	2	2	2-bit	Hamming	3	4	16	0.250	2
	2-of-4	4	2	4	1-bit	Single Parity	1	3	12	0.333	3
	2-of-5	5	3	4	2-bit	Hamming	3	3	15	0.267	3
	3-of-6	6	4	6	2-bit	Hamming	3	2	12	0.333	3
	2-of-7	7	4	4	3-bit	Extended Hamming	4	2	14	0.286	2
8	1-of-2	2	1	2	1-bit	Single Parity	1	9	18	0.444	2.25
	1-of-4	4	2	2	2-bit	Hamming	4	6	24	0.333	1.5
	2-of-4	4	2	4	1-bit	Single Parity	1	5	20	0.400	2.5
	2-of-5	5	3	4	2-bit	Hamming	4	4	20	0.400	2
	3-of-6	6	4	6	2-bit	Hamming	4	3	18	0.444	2.25
	2-of-7	7	4	4	3-bit	Extended Hamming	5	4	28	0.286	2
16	1-of-2	2	1	2	1-bit	Single Parity	1	17	34	0.471	2.125
	1-of-4	4	2	2	2-bit	Hamming	5	11	44	0.364	1.375
	2-of-4	4	2	4	1-bit	Single Parity	1	9	36	0.444	2.25
	2-of-5	5	3	4	2-bit	Hamming	5	7	35	0.457	1.75
	3-of-6	6	4	6	2-bit	Hamming	5	6	36	0.444	2.25
	2-of-7	7	4	4	3-bit	Extended Hamming	6	6	42	0.381	1.5
32	1-of-2	2	1	2	1-bit	Single Parity	1	33	66	0.485	2.0625
	1-of-4	4	2	2	2-bit	Hamming	6	19	76	0.421	1.1875
	2-of-4	4	2	4	1-bit	Single Parity	1	17	68	0.471	2.125
	2-of-5	5	3	4	2-bit	Hamming	6	13	65	0.492	1.625
	3-of-6	6	4	6	2-bit	Hamming	6	10	60	0.533	1.875
	2-of-7	7	4	4	3-bit	Extended Hamming	7	10	70	0.457	1.25
64	1-of-2	2	1	2	1-bit	Single Parity	1	65	130	0.492	2.03125
	1-of-4	4	2	2	2-bit	Hamming	7	36	144	0.444	1.125
	2-of-4	4	2	4	1-bit	Single Parity	1	33	132	0.485	2.0625
	2-of-5	5	3	4	2-bit	Hamming	7	24	120	0.533	1.5
	3-of-6	6	4	6	2-bit	Hamming	7	18	108	0.593	1.6875
	2-of-7	7	4	4	3-bit	Extended Hamming	8	18	126	0.508	1.125

Table 5.7: Finding optimal error-detecting codes for Berger codes.

Data word size	$f = 1$	$f = 2$
2	2-bit ED	2-bit ED
3	2-bit ED	3-bit ED
4	4-bit ED	4-bit ED
5	4-bit ED	5-bit ED
6	4-bit ED	6-bit ED
7	4-bit ED	6-bit ED
8	8-bit ED	8-bit ED

Table 5.8: 3-bit Berger codewords combined with Hamming codes ($f = 1$).

Data word size	Parity bits	Number of 3-bit Berger codewords[2]	Total number of wires	Information rate
4	3	3	15	0.267
8	4	4	20	0.4
16	5	7	35	0.457
32	6	13	65	0.492
64	7	24	120	0.534

As can be seen in Table 5.7, only for data word sizes of two and three, the required detection capabilities are below four, which means that Hamming codes can be used. Data words with four or more bits already require stronger ED codes, which have much more complex encoder and decoder circuits. Assume we use 3-bit Berger codes and apply a Hamming code to detect decode errors. Following the encoding scheme presented above for m-of-n codes, data words of arbitrary length have first to be extended by parity bits from the Hamming code and are then cut into 3-bit chunks to be encoded with the selected Berger code. Table 5.8 shows the required interconnect wires and the resulting information rate of this encoding scheme for different word sizes, assuming a targeted tolerance against single faults ($f = 1$). Compared to our results for m-of-n codes, it can be seen that this particular Berger/Hamming code combination can offer similar information rates as 2-of-5 or 3-of-6 codes. Encoding and decoding of the systematic Berger code, however, might be a bit more efficient.

5.2 Approach I: Robust 4-phase Dual-rail Channels

In this section we will present the implementation of a robust asynchronous link, which is specifically targeted at and optimised for the use of a 4-phase dual-rail code to achieve delay-insensitivity. As good portion of this section is devoted to the description of input and output port components that are ready to be used in asynchronous communication wrappers of GALS modules. Figure 5.5 shows the general structure of the proposed robust asynchronous communication interface. The sender consists of a *sender module* and an associated *output port*. The

[2]For a 3-bit data field two synchronization bits are needed. The codewords therefore have a total length of five.

receiver is partitioned into a *receiver module* and an *input port*. Input and output ports are assumed to be connected over a global interconnect for on-chip as well as off-chip communication, which is susceptible to transient faults and where signal delays are affected by PVT variations.

As described in the previous section, the output port encodes data in two steps to achieve both fault tolerance and delay-insensitivity: First, the data word is extended by parity bits or some kind of checksum, and then all data bits as wells as the additional parity/checksum bits are encoded with the DI code. Based on this encoding, the input port of the receiver can perform completion detection *and* error detection. An on-going data transmission is only acknowledged, if the received data word is *complete and correct*, or if a detected error can be *corrected* by the error correction unit. This is the key principle of the presented link implementation.

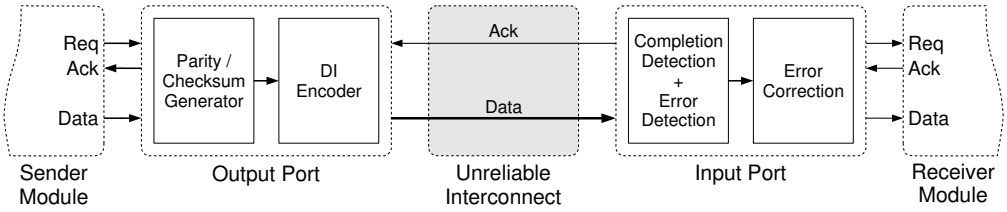


Figure 5.5: Overview of fault-tolerant link.

Since we target GALS approaches where pausible clock generators are employed, input and output ports need to operate without a working clock signal and hence have to be implemented asynchronously. The local interface between a GALS module and the associated input and output ports consequently is handshake-based, as can be seen in Figure 5.5. Since the data exchanged over this interface is in the form of single-rail data words, a bundled data protocol is used and the ports are consequently implemented as matched delay circuits. Note that a 4-phase protocol is assumed for the circuits discussed here.

5.2.1 Fault Model

All assertions about fault tolerance made for the presented approach rely on the following assumptions about faults occurring in the system:

1. All faults are of transient nature. Consequently, two types of erroneous signal pulses can be distinguished: 1) $0 \rightarrow 1 \rightarrow 0$ and 2) $1 \rightarrow 0 \rightarrow 1$. A key factor, when designing a fault-tolerant communication link, is the structure of the transmission channel, i.e., the interconnect network. A major concern is the question whether the interconnect contains storage elements, e.g., in case it is pipelined. In [62], we therefore introduced the following classification of interconnect links:

   I) Interconnects without state-holding elements

   II) Interconnects with state-holding elements

   For interconnect type I, it is possible to assume that the receiver will eventually perceive a complete and correct data word at its input. This assumption clearly does not hold

for the second type of interconnects. If there is an upset in some storage element of a pipelined interconnect and no error correction is performed by the pipeline stage, the receiver will never obtain a correct data word. To make matters worse, the receiver cannot even assume that eventually a complete data word will arrive, thus leading to a deadlock of the handshake protocol. As demonstrated in [16] and [61] solutions that are able to cope with such a scenario lead to rather expensive and complex implementations, if delay insensitivity is supposed to be preserved. Therefore we assume type I interconnects for the two link architectures presented in this and the following section.

2. Faults only occur in the interconnect. As we explained above, this assumption is used for simplifying the presentation of this approach. In a fully fault-tolerant system, the sender and receiver components would of course be protected by some kind of fault tolerance mechanism, such as triple modular redundancy.

3. No more than one fault occurs during one communication cycle (single fault assumption).

5.2.2 Proposed Implementation

As stated above, we have decided to use a 4-phase dual-rail code for this specific link architecture, partly because of its simplicity and its wide-spread use in the asynchronous circuits community. From our analysis of m-of-n codes in Section 5.1, we know that such a dual-rail code can be protected with an ED code that has single-bit error detection capabilities, assuming single transmission faults (cf. Table 5.5). Single-bit error detection can easily be achieved with an inexpensive parity bit that is added to the transmitted data word.

Figure 5.6 shows the basic structure of the output port. First the parity bit is generated for the input data word, which is provided by the sender module. Then the *DI encoder* transforms the parity-extended data word into a delay-insensitive dual-rail codeword, which is finally stored in an output register. A simple Muller C-element performs the local handshake with the sender module, processes the acknowledge signal from the remote receiver and triggers the output register when the next data word or the next spacer word has to be issued. Since the communication between the sender module and the output port is performed with a bundled data transfer, a delay element has to be included on the local request signal. This delay is matched to the aggregated delays of the parity generator and the DI encoder.

The corresponding input port at the receiver side is depicted in Figure 5.7. The general concept is very simple: The receiver waits for a complete and fault-free data word, captures it in its input register and then returns an acknowledgement to the sender. Consequently, the input register is transparent at the beginning of the communication cycle. The incoming data word propagates through the input register and is continuously evaluated by the completion detector and the error detection circuitry. If completion detection and error detection indicate that the data word is *complete and correct*, the handshake controller freezes the input register. Subsequently, the local request signal is raised, indicating the availability of new data to the receiver module. The acknowledgement, generated for the sender, then completes the activities of the input port.

In most cases the input register now stores a complete and correct data word, which can be consumed by the receiver module. However, there is a small chance that a fault occurs exactly

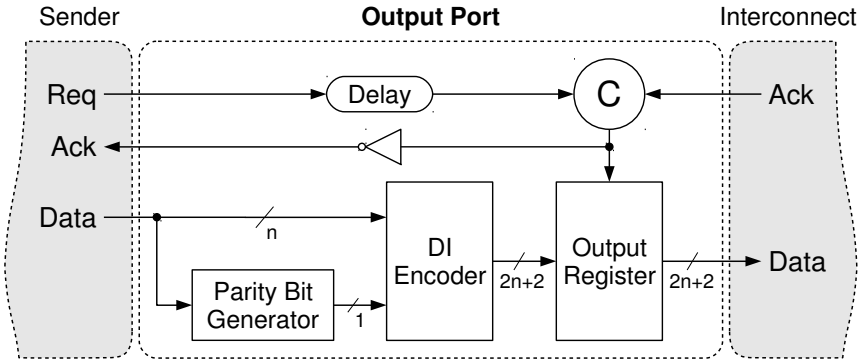


Figure 5.6: Output port.

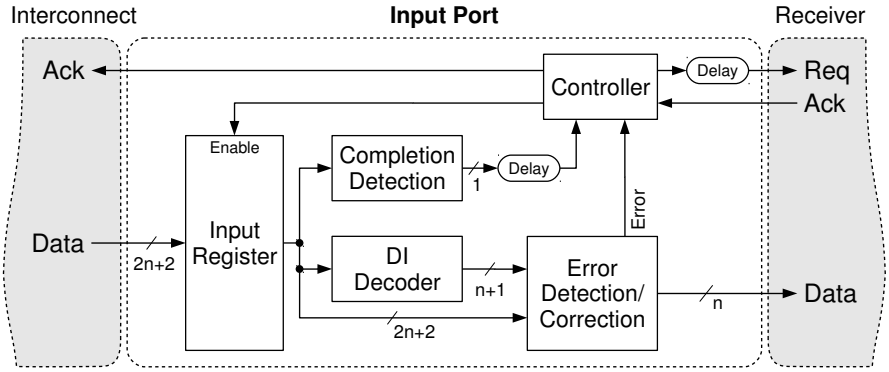


Figure 5.7: Output port.

during the short time interval between the detection of a correct data word and the freezing of the input register. Then the register still might have latched a data word with a single bit-flip. Typically, the available parity bit would not be sufficient to perform forward error correction at the receiver. Our solution, however, uses the combined information redundancy of the delay-insensitive code and the parity bit. In order to show that this composition is sufficient for not just detecting but also correcting single-bit errors, we need to determine the Hamming distance of the resulting codewords:

Theorem 3. *Let C be a code with a minimum Hamming distance of d. If the bits of the codewords of C are encoded with a 1-of-2 code, the new code C' has a hamming distance of $2d$.*

Proof. Let x' and y' be two codewords of C', $x' \neq y'$ and $x, y \in C$. Since C is a code with a minimum Hamming distance of d, at least d bits have to be changed to transform x into y. x' and y' are derived from x and y by encoding each bit of x and y with two bits, using a 1-of-2 code, which has a Hamming distance of 2. Therefore 2 bits need to be changed in x' in order to change a single bit in x. Consequently, at least $2d$ bits have to be changed to transform x' into y', i.e., the minimum hamming distance of C' is $2d$. □

Based on Theorem 3, the Hamming distance of the transmitted data words can be easily calculated: Data words, which are protected by a single parity bit, have a Hamming distance of $d = 2$. Encoding the parity-extended data words with a 1-of-2 code results in a total Hamming distance of $2d = 4$. Recall that for correcting $f = 1$ erroneous bits the minimum Hamming distance between two codewords is required to be $2f + 1 = 3$. Thus, the composition of a parity bit and a 1-of-2 code is sufficient for single-bit error correction.

The implementation of the correction mechanism is very simple: In case a late faulty signal transition slipped into the input register, a previously correct dual-rail pair will either be changed to 00 or 11. Both values are invalid 1-of-2 codewords and the faulty dual-rail pair can be easily detected. The correction can then be performed as follows: The DI decoder delivers the decoded data word including the received parity bit to the error detection circuit. Then the parity bit is recomputed and compared to the received parity bit. If they are not equal, the decoded data word contains an error. In this case, the error correction unit analyses the dual-rail pairs stored in the input register in order to find the invalid pair. For this purpose, the error correction unit has a separate input, which is directly connected to the input register (cf. Figure 5.7). The data bit that corresponds to the invalid dual-rail pair then is simply inverted to produce a correct data word.

The receiver module, of course, must not process the new data word before the correction process has been completed. Therefore, a delay element is included in the local request signal. The length of this delay needs to be matched to the combined delay of the DI decoder and the error detection/correction circuit. Another delay element is required at the output of the completion detector. Note that the error detector continuously checks the incoming data word and thus might identify an incomplete data word to be correct. Therefore, it is necessary to make sure that the error detector can finish its computations before the handshake controller reacts on the completion and error detection signals. The delay element at the output of the completion detection thus compensates the delay difference between error and completion detection circuits.

5.2.3 Metastability-Tolerant Implementation

Unfortunately, the design of the input port presented above is prone to metastability. When the input register changes from transparent to hold, an erroneous input pulse can cause a setup/hold-time violation and the register might become metastable. Of course, the metastability will resolve eventually and even if the resolved value is incorrect, this can be compensated by the error correction. However, at this point in time the local request signal for the receiver module might already have been issued. As the metastability resolution time is unconstrained, it cannot be guaranteed that the data signals remain stable when the receiver module processes the incoming request. Even though it seems quite improbable that metastability caused by a fault results in a transmission error, we want to present a metastability-tolerant receiver implementation. Maybe it can prove useful for highly critical systems or extremely harsh operating conditions.

Recall that metastability can never be prevented, if there are bi-stable storage elements in a circuit. Nevertheless, it is possible to change the implementation such that the local request is only issued after a metastable upset has resolved and the persistency of the data word can be guaranteed (assuming that no faults occur in the input port itself, according to our fault model). This requires an alternative realisation of the input register, which must only produce stable and monotonic output transitions. This property is crucial for making the decision when the local

request signal can be safely issued. Instead of a latch-based implementation, a 4-phase dual-rail register can also be built from Muller C-elements (cf. [102]). As you can see in Figure 5.8, one input of the C-elements is connected to the data signals of the interconnect link, whereas the second input is connected to the local acknowledge signal of the receiver module. Depending on the value of the acknowledge signal, the C-element register either accepts the next data word or the next spacer codeword. In case the acknowledge signal is 1, the C-elements are transparent for the $0 \rightarrow 1$ transitions of an incoming data word. Once a 1 has been latched into the C-element, it is preserved as long as the acknowledge signal does not change, even in case of a possible $1 \rightarrow 0$ fault on the data input. Hence, the switching behaviour of the C-elements is monotonic. Of course, a faulty $0 \rightarrow 1$ transition can now penetrate the input register at any time and will immediately be captured. But this drawback is acceptable as we can deal with single-bit errors by means of error correction.

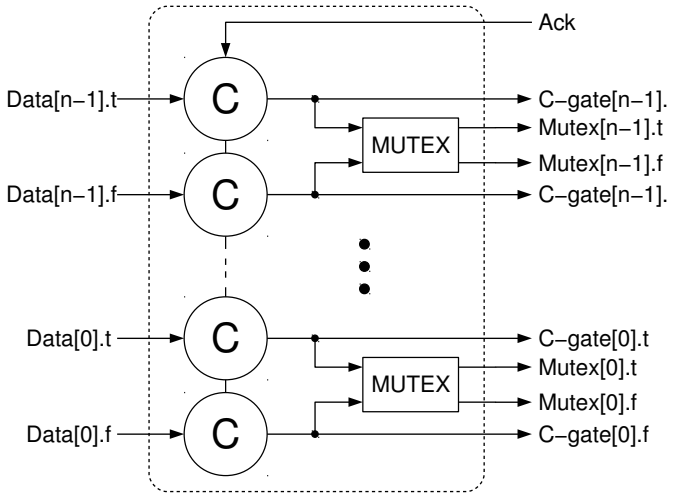


Figure 5.8: Monotonic input register.

However, a C-element is a bi-stable element and can also be affected by metastability. Upon a $0 \rightarrow 1$ input transition the internal storage loop has to change from one stable state to the other, from storing a 0 to storing a 1. This, of course, imposes a hold-time requirement for the data input. A short faulty $0 \rightarrow 1 \rightarrow 0$ pulse might violate the hold-time and the C-element ends up in a metastable state. In this situation an ordinary C-element might produce a pulse at its output, which contradicts the demanded monotonic behaviour. However, it has been shown in [88] that a C-element can be built such that an output transition is only produced after the metastable state has resolved. To this end, the presented C-element implements its output stage with a Schmitt-Trigger. The key observation is that the internal nodes of the element are at an intermediate voltage level, as long as metastability has not been resolved. By selecting appropriate threshold values (one above and one below the metastability voltage), the Schmitt-Trigger ensures that the output of the gate only changes once the voltages of the internal nodes have settled to a proper digital value and thus metastability has been resolved.

Using the proposed C-element input register, completion detection can be safely performed:

A complete data word will remain complete even in case of late errors and therefore the output of the completion detector is monotonic. The same, however, is not true for the correctness of the stored data word: A late erroneous $0 \rightarrow 1$ transition can still change a correct 01 or 10 dual-rail pair into an invalid 11 pair. Furthermore, an incorrect 01 or 10 dual-rail pair, caused by an early faulty $0 \rightarrow 1$ transition, can be changed into an invalid 11 pair when the correct input transition arrives. In these cases, the output of an error detection circuit, directly evaluating the data word stored in the C-element register, would not be monotonic. This problem can be solved by using *mutex* components for arbitration between rising transitions on the *true* and *false* rails of every dual-rail pair. The first rising transition on any of the two mutex inputs will be forwarded to the associated grant output. If a later transition occurs on the second input, it will be blocked. In case of simultaneous transitions on both inputs, the mutex component will arbitrarily issue a grant for exactly one of them. This makes transitions on the *true* and *false* rails mutually exclusive.

Once a dual-rail data word, provided by the grant outputs of the mutexes, is complete, the mutex outputs will not change anymore for the whole data phase of the communication cycle. Error detection can therefore be safely performed: When the data word is correct, it will remain correct. When the data word is incorrect, it will remain incorrect. In the first case, the local request signal can be issued immediately. In the latter case, error correction still needs to be performed. As can be seen in Figure 5.9, the dual-rail data word provided by the mutex outputs is decoded and forwarded to the error detection/correction circuit. If an error is detected, the outputs of the C-elements are evaluated in order to locate the invalid dual-rail pair. As mentioned above, an incorrect 01 or 10 dual-rail pair, stored in the C-element register, will eventually turn into an invalid 11 pair. Then the faulty data bit can be inverted and the correct data word is recovered. After a successful data correction, the error signal is disabled and the local request is raised by the C-element in the top-right corner of Figure 5.9. A detailed presentation of the error detection/correction circuit will be given in the next section. Note that the error signal is only evaluated for rising transitions of the request signal (data phase). For the reset phase of the handshake, is it sufficient to wait until the completion detection goes low, when the all-zero empty codeword has been received. Then the local request signal is reset to zero, which completes the handshake cycle.

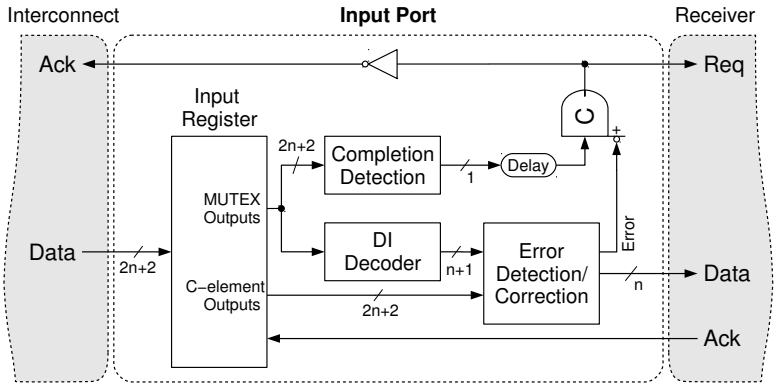


Figure 5.9: Metastability-tolerant input port.

5.2.4 Implementation Details

Input/Output Registers

In case of the standard implementation (no metastability-tolerance) the input register can simply be built with D latches, which are controlled by an *enable* signal (cf. Figure 5.7). The input register for the metastability-tolerant version of the receiver uses C-elements and mutex components, as we have already explained in Section 5.2.3.

The output register of the sender can also be built with C-elements. One input is connected to the control signal, the other to a true-rail or false-rail provided by the DI encoder (cf. Figure 5.6). When the control input is 1, the C-elements capture the rising transitions of the next data word. In order to implement the return-to-zero part of the handshake cycle, where the output register needs to generate the spacer codeword, we have in fact used asymmetric C-elements with a reset function that only depends on the control input. Thus, a falling transition on the control input immediately resets the C-elements, irrespective of the data input value.

Error Detection/Correction

The error detection/correction unit of the input port of the metastability-tolerant solution is depicted in Figure 5.10. It consists of two error detectors and one error corrector. The decoded data word is analysed by the first error detector. If it is correct, the internal error signal is disabled and the data word passes the correction unit without modification. Consequently, the second error detector will not detect a fault and the error output remains low, which triggers the local request signal. In case the first detector identifies a fault (internal error signal set high), correction needs to be performed. As we explained in Section 5.2.3, this is done by looking at the dual-rail pairs stored in the C-elements of the input register. Once the correct rail has arrived, this value will be 11. As can be seen in Figure 5.10, an AND-gate detects this situation and enables the correction of the corrupted data bit. Afterwards the second error detector will set the error output to 0, which in turn triggers the local request signal.

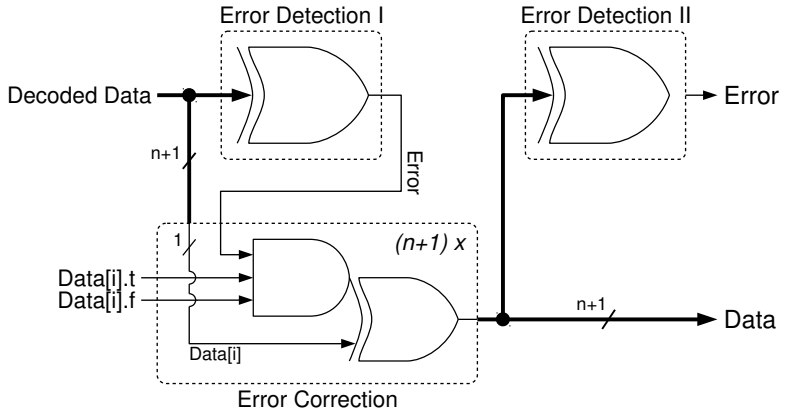


Figure 5.10: Error detection/correction unit.

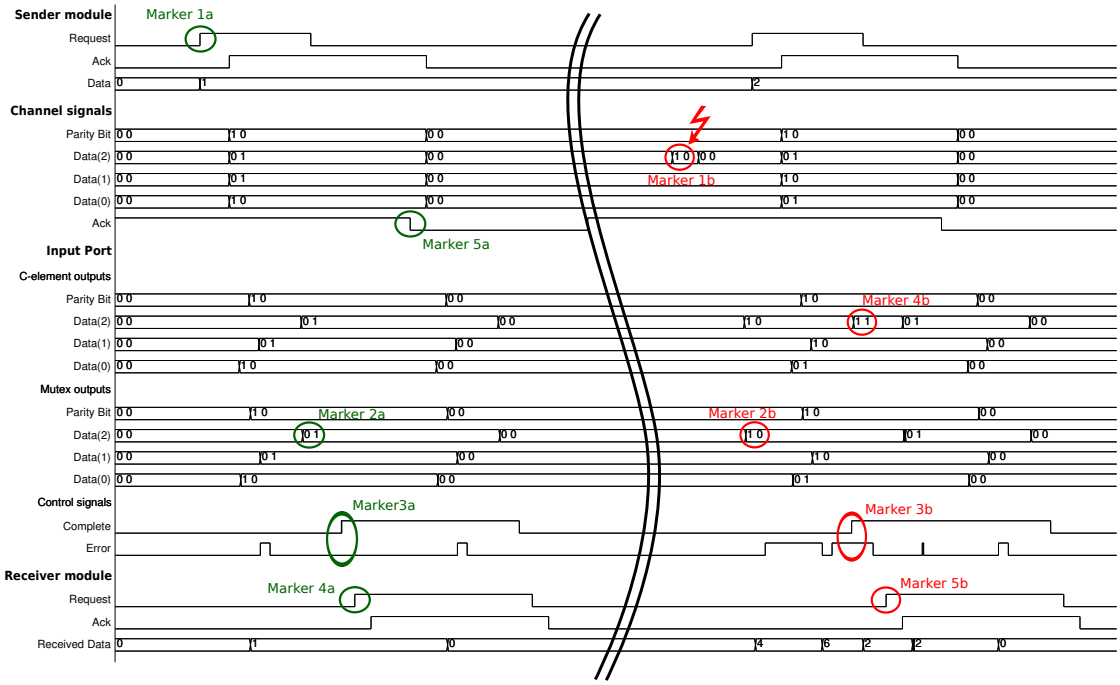


Figure 5.11: Simulation results: Correct transmission on the left, faulty transmission (and error correction) on the right side.

5.2.5 Evaluation

Functional Verification

To be able to perform simulation experiments and analyse the behaviour of the interface components, we have implemented a gate-level model of the described input and output ports. Analog effects, like metastable upsets, can obviously not be investigated directly with a gate-level simulation. Fortunately, the used metastability-tolerant C-elements only produce digital output transitions [88]. Metastable upsets can therefore nicely be covered with delayed output transitions.

Figure 5.11 shows a waveform from a simulation of two data transmissions. The first transmission depicts a fault-free execution. The sender module delivers a new data word to the output port and issues a request to initiate the data transmission (Figure 5.11, Marker 1a). After a short delay the dual-rail encoded data word appears on the communication channel. The data word propagates over the interconnect signals and is captured by the C-elements of the input register at the receiver side. In the middle rows of the waveform the state transitions of the C-elements and the connected mutexes can be seen. We have deliberately added random delays for all interconnect signals to show how completion detection works. In this specific test case, $Data(2)$ is the last dual-rail pair that arrives at the receiver (Marker 2a). Subsequently the completion detector fires and asserts its output signal (Marker 3a). Note that the error signal has produced a glitch during the time when the data word was incomplete. Apparently, the parity check failed for some intermediate bit pattern of the incoming data word. This glitch, however, is discarded

as the error signal is only processed when the data word is complete and the output of the error detection circuit has settled. As can be seen in the waveform, the error signal delivers a stable 0, when the complete signal makes the rising transition (Marker 3a).

After the completeness and correctness of the incoming data word have been verified, the local request signal is raised to notify the receiver module about the availability of new data (Marker 4a). Concurrently, the acknowledge signal of the interconnect link makes a transition (Marker 5a), which notifies the sender about the reception of the data word. The sender then responds with the empty codeword to initiate the return-to-zero part of the handshake cycle.

In the second transmission we have injected an erroneous pulse on the $Data(2)$ interconnect signal (Marker 1b). In this specific test case the error occurs some time before the actual transmission of the data word is started. This nicely allows for an illustration of the error correction capabilities of our solution. As can be seen in Figure 5.11, the error is latched by the corresponding C-element of the receiver and consequently the connected mutex component changes its outputs (Marker 2b). At this point in time, this seems to be the valid start of the next data transmission. The receiver again waits for all other dual-rail pairs to become complete and then checks the result of the error detection. Now the error signal is set, when the completion signal makes the rising transition (Marker 3b). Thus, the data word provided by the outputs of the mutexes contains an error, which needs to be corrected. As can be seen in the waveform, the original data word issued by the sender has a value of 2, whereas the received data word (bottom line of the waveform) is 6. Until correction has been performed, the local request signal therefore remains low. Recall that $Data(2)$ is transmitted on the slowest interconnect wires. Therefore, the last input transition, which arrives, is the correct transition on the false rail of $Data(2)$. This is captured by the corresponding C-element (Marker 4b). Now the error correction, which analyses the C-element outputs, can identify and invert the erroneous data bit. The correct value of 2 appears at the data output and the receiver is notified with the local request signal (Marker 5b).

Area Analysis

The area overheads in terms of interconnect resources are very moderate. Only two additional wires are needed to transmit the extra dual-rail encoded parity bit. Clearly, this overhead is constant with respect to the bit width of the transmitted data word. The overall efficiency, however, is less optimal because of the duplication of wires due to the dual-rail code. In Section 5.1 we have seen that with other DI codes, e.g., with a 3-of-6 code, better information rates and therefore a more frugal use of interconnect resources can be achieved.

Nevertheless, 4-phase dual-rail codes can offer extremely low overheads for encoding and decoding, which reduces the complexity of the input and output ports. For encoding n data bits and one parity bit only $n + 1$ inverters are needed. Decoding needs no gates at all. In general, the input and output port implementations are very light-weight and scalable. As can be seen in Tables 5.10a and 5.10b, the gate count of all components is either constant or increases linearly with the numbers of inputs n. Based on the transistor counts for the specified gates, the final row of the tables gives a formula to compute the total number of transistors for the output and inputs ports, respectively. Note that we did not list the number of gates for building the required delay elements, since this depends on the timing of a specific implementation. It is, however, safe to say that delay lines will only contribute a small fraction to the overall area consumption.

Table 5.9: Area Requirements.

Block	Gate	Transistors/gate	Gate count
Parity Generation	XOR2	12	$n-1$
Dual-rail Encoder	INV	2	$n+1$
Output Register	Asymm. CGATE2	10	$2(n+1)$
Handshake Control	CGATE2	12	1
	INV	2	1
		Total transistors:	$34n+24$

(a) Output Port.

Block	Gate	Transistors per gate	Gate count	
			Standard port	MS-tolerant port
Input Register	D-Latch	18	$2(n+1)$	N/A
	CGATE2[3]	18	N/A	$2(n+1)$
	MUTEX	12	N/A	$n+1$
Completion Detector	OR2	6	$(n+1)+n$	$(n+1)+n$
	AND2	6	n	n
	CGATE2	12	1	1
Error Detection I	XOR2	12	n	n
Error Detection II	XOR2	12	N/A	n
Error Correction	AND3	8	$n+1$	$n+1$
	XOR2	12	$n+1$	$n+1$
Handshake Control	CGATE2	12	1	1
	INV	2	1	1
		Total transistors:	$86n+88$	$110n+100$

(b) Input Port – Standard vs. metastability-tolerant implementation.

In case of the input port, Table 5.10b also compares the standard and the metastability-tolerant port versions. Note the different input register implementations and the additional error detection circuit needed in case of the metastability-tolerant version. In terms of transistor count, the area overhead of metastability-tolerant receiver is defined by $a_{ov}(n) = \frac{110n+100}{86n+88} - 1$. The lower bound of this overhead can be obtained for $n = 1$ and has a value of $a_{ov}(1) = 20.7\%$, whereas the upper bound, as n approaches infinity, equals $lim_{n\to\infty} a_{ov}(n) = 27.9\%$.

Even in comparison with a single-rail synchronous circuit implementation the overheads are reasonable. Consider, e.g., the input and output registers. At first sight these seem to grow significantly as twice the number of storage elements is required. Looking at the transistor count per data bit, however, changes this impression. An asymmetric Muller C-element can be built of 10 transistors (based on Sutherland's C-element implementation). This makes 20 transistors per bit for the output register. In comparison, an edge-triggered master-slave D flip-flop requires ap-

[3]C-element with Schmitt-Trigger as presented in [88].

prox. 30 transistors, depending on the specific implementation. Even for the input register of the metastability-tolerant version, which requires an additional mutex component per bit, the area overhead with respect to a regular flip-flop is acceptable: The metastability-tolerant C-element including a Schmitt-trigger needs 18 transistors [88], and the mutex component, consisting of 2 NAND gates and two inverters, needs 12 transistors. Thus, $2 \cdot 18 + 12 = 48$ transistors are required for the input register per bit.

Of course, the proposed input port needs a completion detector and a second error detection circuit, which are not required for a synchronous design. This price, however, seems acceptable considering the gains provided by delay-insensitivity and tolerance against metastability. Note that the gate counts given for the completion detection, the parity generation and the error detection assume implementations with balanced trees of 2-input gates. In case of n inputs these trees can be built of $n - 1$ gates.

5.3 Approach II: A Generic Sender/Receiver Implementation

In the previous section we have presented a link implementation specifically targeted at 4-phase dual-rail circuits. In this section we want to propose an architecture that is completely generic with respect to the type of handshake protocol (2-phase/4-phase) and the used delay-insensitive and error detecting codes. We will discuss a general framework for implementing asynchronous input and output ports where arbitrary encoder, decoder and completion detection units can be instantiated, depending on the specific choice of the codes that shall be used. To be able to keep this link architecture generic the implementation of the input and output ports had to be adapted. Especially the input port has been subjected to a bigger redesign, as we will show below.

The assumptions made in the fault model of the previous approach are still in place, with the following exceptions: 1) the new approach is not restricted to single transmission faults. Depending on the strength of the selected error detecting code a specific implementation can be derived that is also capable of mitigating multiple transient faults. 2) Due to the different mode of operation the input port can no longer systematically deal with metastable upsets of the input register caused by transient faults during the setup/hold window.

5.3.1 Output Port

The structure of the output port is similar to the port implementation of the previous approach (cf. Figure 5.6). Again an ED encoder unit first augments a new data word arriving from the sender with the required redundancy to enable error detection. Subsequently, the produced data word is encoded with a DI code, resulting in a codeword ready to be transmitted. The control logic and the output register have changed a little bit to support both 2-phase and 4-phase protocols. The control logic now produces a clock pulse to trigger a flip-flop based output register, when newly encoded data is ready to be transferred to the receiver across the interconnect signals. Two conditions need to be satisfied before such a clock pulse is produced: a) New data needs to be ready as indicated by the request signal from the sender module, b) the last transmitted codeword needs to be acknowledged by the input port of the receiver. Since we are again using a bundled data implementation, a delay element is inserted on the request signal and needs to be matched

to the maximal propagation time of the data path, i.e., the ED and DI encoders. This ensures that a new codeword safely stabilises at the register input before the clock pulse is generated. Figure 5.12 shows the schematic of this output port.

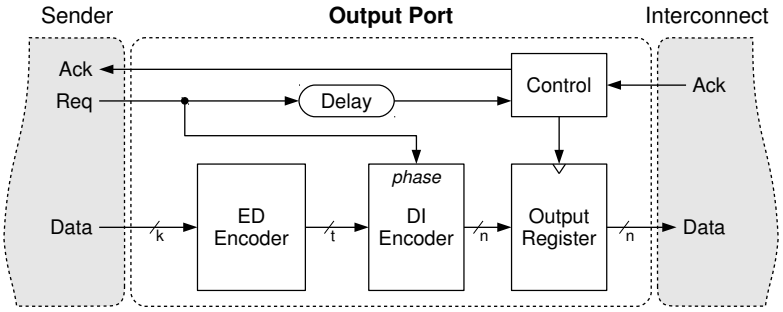


Figure 5.12: Output Port.

5.3.2 Input Port

As said above, the generic input port works a bit differently than the port we presented for the previous approach. Rather than using an input register that is transparent all the time until an incoming transition is complete and correct, we use a flip-flop based input register that samples data once the arriving DI codeword is complete. Error detection is then performed in a second step. If the sampled data word is error-free, it can be propagated to the receiver module, otherwise the incoming data word is sampled again and once more checked for errors. This process is repeated until all transient faults have vanished and a correct data word is stored in the input register. Clearly, this will eventually happen since we only assume transient faults in our fault model. Note that in this approach only error *detection* is performed, error *correction* capabilities of the used codes are not required. Therefore, the number of redundant bits can be reduced, which optimises the use of interconnect resources. Furthermore, this approach streamlines the size of the input port and positively affects the performance of fault-free transmissions.

As can be seen in Figure 5.13, the completion detector (CD) directly connects to the incoming interconnect wires. In case of a 2-phase protocol every complete data word that arrives will toggle the output of the CD unit between one and zero. In a 4-phase protocol the output of the completion detector also changes between one and zero, but only rising transitions indicate the availability of new data. Once a new data word is available, the control unit, connected to the output of the completion detector, will issue a clock pulse, which latches the data word into the input register. Note that the DI decoder is placed before the register. This allows for concurrent operation with the completion detection and also reduces the size of the input register. After the new data word has been stored, error detection can be performed. The control unit waits until error detection is finished and then checks the *error* output of the ED decoder (see Figure 5.13). If this signal is zero, a request transition is produced for the receiver and an acknowledge is returned back to the sender's output port. Otherwise, the control unit simply issues another clock pulse and the data word is sampled again and the process repeats.

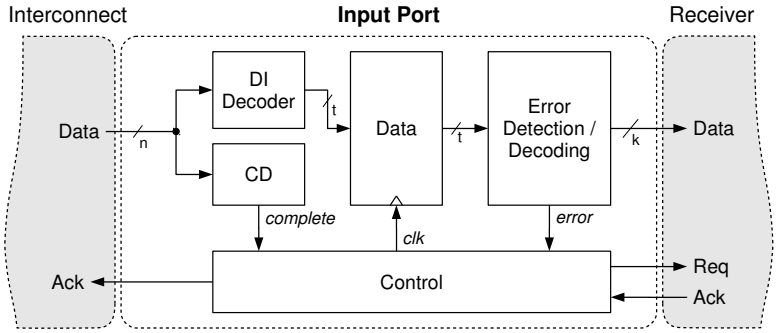


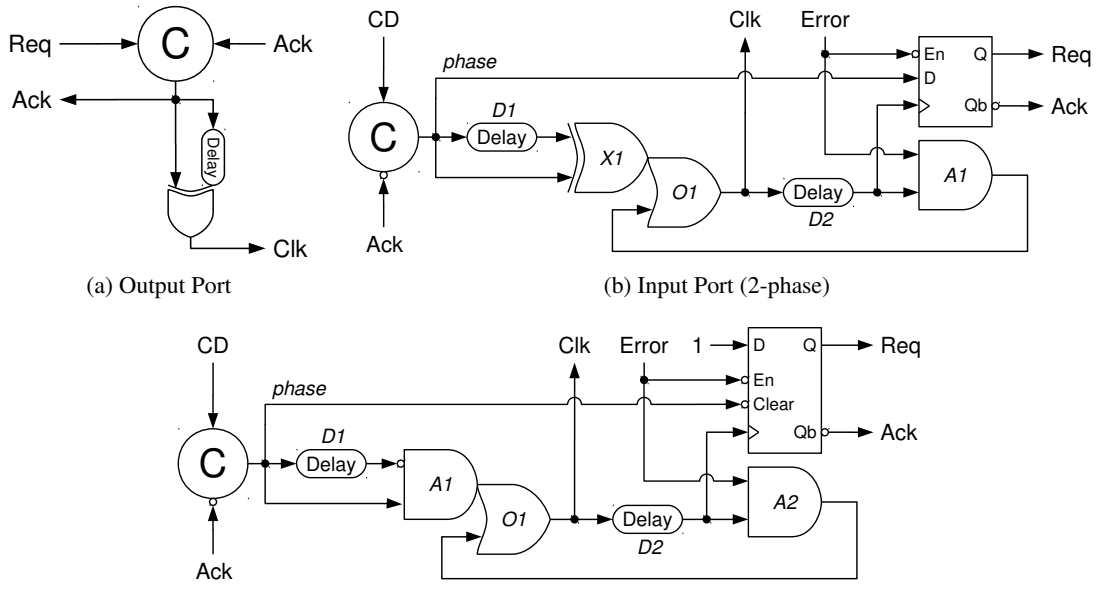
Figure 5.13: Input Port.

5.3.3 Control Circuits

The control circuits we have implemented for the proposed link architecture are completely independent from the employed delay-insensitive and error detecting codes. The controller for the output port is depicted in Figure 5.14a. It can be used both for 4-phase and 2-phase protocols. A C-gate simply waits for a new request from the sender module and an acknowledge from the receiver. If both signals are equal, new data can be transmitted. The output of the C-gate then makes a transition, either falling or rising – depending on the protocol phase. This transition forks into two wire branches that are connected to the same 2-input XOR-gate. As one branch is delayed, the inputs of the XOR-gate will be unequal for a short time and a $0 \rightarrow 1 \rightarrow 0$ pulse is produced at the output of the XOR-gate. This pulse clocks the data register of the output port and a new data word/spacer word is issued. Concurrently the output transition of the C-gate is returned to the sender as acknowledge signal.

The input port controllers are a little more intricate. This time, different circuits are needed for 2-phase and 4-phase protocols. Let us first consider the 2-phase version (Figure 5.14b). The left-hand side of the circuit is the same as the output port controller: A C-gate waits for a transition on the signal from the completion detector and forwards it, if the receiver is ready (see ack signal connected to the lower C-gate input). Due to XOR-gate *X1* an up/down pulse is produced, which is then propagated to the clock output. As explained in the previous section, this clock pulse triggers the sampling of the input data, and thereby initiates error detection. If no error was found, the transition of the completion detection needs to be forwarded to the request output to notify the receiver. Furthermore, the acknowledge signal needs to be toggled to inform the sender about the successful data reception. This task is performed by a flip-flop on the right-hand side of the circuit. Note that this flip-flop is clocked by a delayed version of the initially generated clock pulse (see delay *D2* in Figure 5.14b). Since the flip-flop must only be enabled when the error signal is zero, the rising clock transition needs to be delayed long enough so that the error detection has completed and the error/enable signal is stable. Thus, delay element *D2* needs to be matched to the data path of the input port (i.e., clock-to-output delay of the input register plus propagation time of the error detector plus setup-time of the flip-flop).

In case a fault was detected ($error = 1$), the flip-flop will not change the request/acknowledge signals. Instead the delayed clock pulse is re-used to issue another clock tick for sampling



(a) Output Port

(b) Input Port (2-phase)

(c) Input Port (4-phase)

Figure 5.14: Control Circuits.

the input data once again. Since the error signal is high, a feedback loop is established by AND-gate *A1*. The clock pulse can propagate on the feedback signal and then re-appears on the clock output. Clock pulses will be generated this way until the transient fault disappears. In theory this could go on for an infinite number of cycles. In practice, however, the oscillation will eventually decay due to pulse broadening or shortening. Nevertheless, if the period of the oscillation is chosen long enough (controlled by delay *D2*) and the pulse width is approximately half this period, i.e., $D1 = D2/2$, the transient error will vanish long before the oscillation stops.

The 4-phase controller, which can be seen in Figure 5.14c, looks very similar. However, there are some small modifications. In a 4-phase protocol a clock pulse only needs to be generated for the data phase. Since the communication with the receiver module is performed over a bundled data interface, the spacer codeword does not need to be propagated. Therefore, a clock pulse is only generated for *rising* output transitions of the C-gate. As before, the output of the C-gate forks into two branches, but this time it is connected to an AND-gate (*A1*) instead of an XOR-gate. Note that the delayed branch is inverted. It can be easily seen that rising transitions will produce a pulse on the AND-gate output, whereas falling transitions will be masked.

However, a falling transition at the C-gate's output during the reset phase has another effect. As can be seen in Figure 5.14c, the output of the C-gate is connected to the active-low clear input of the flip-flop, which controls the request/acknowledge signals. Thus, the flip-flop will be immediately reset to zero, which completes the current handshake cycle. Since there is no need to sample input data or perform error detection, the reset phase of the communication cycle is significantly shorter than the data phase. This obviously increases the throughput of the presented link implementation.

5.3.4 Evaluation

For evaluation of the link architecture presented above, we have implemented input and output ports that again use dual-rail codes for delay-insensitive communication across the interconnect. In case of a 4-phase protocol a 1-of-2 code is used once more, for the 2-phase protocol we implemented the LEDR encoding scheme.

Regarding the ED codes, the ports either use a single parity bit or alternatively Hamming codes for fault tolerance. From Section 5.1 we know that these two codes provide protection against *single* and *double* transmission faults in 1-of-2 encoded data transmissions. Even though we did not analyse 2-phase DI codes, we can easily derive that the same level of protection can also be achieved with a 2-phase LEDR encoding. Note that LEDR is a dual-rail code, where every dual-rail pair encodes exactly one data bit. In other words, the value of any LEDR rail only relates to a single bit in the encoded data. Consequently, f faults in a LEDR-encoded data word can cause at most f decode errors. The error detection capabilities of the used ED code therefore directly determine the number of transmission faults that can be mitigated. Pairing the selected DI and ED codes yields the following four combinations, for which we have implemented respective input and output port components:

- 2-phase dual-rail (LEDR encoding) with single parity bit
- 2-phase dual-rail (LEDR encoding) with Hamming code
- 4-phase dual-rail (1-of-2 code) with single parity bit
- 4-phase dual-rail (1-of-2 code) with Hamming code

After modelling all four link versions in VHDL, we performed logic synthesis and mapped the circuits to a 90 nm standard cell library. A timing simulation of the synthesised netlist for 2-phase ports with Hamming code protection can be seen in Figure 5.15. The simulation waveform shows two transmissions. In the first one no faults interfere during the communication. A single clock pulse is generated and the incoming data word is forwarded to the receiver. Note that the short pulse on the error signal is just a glitch that occurs while the error detection circuit performs its computations. During the second transmission we have injected a double fault on the interconnect signal (two transitions in the red circle marked with the flash symbol). As can be seen, the error signal is raised after the input data are latched for the first time. Thus, instead of a request for the sender, a second clock pulse is generated. Since the faults have vanished at this point in time, the error signal is reset to zero and the transmission can be completed.

Table 5.10 shows area and performance characteristics. As can be seen, we have benchmarked different data widths for the first two port designs. Regarding the area complexity the table nicely shows that the circuits scale linearly with increasing number of data bits. The coding efficiency, i.e., the ratio of transmitted data bits to required interconnect wires, is shown in the last column. Since we are using dual-rail codes, the efficiency can of course never be above 0.5.

The performance results are derived from simulations, where we assumed ideal conditions to attain the maximum throughput and minimum latencies. Thus, we did not introduce additional delays on the interconnect signals and used simulation models for ideal sender and receiver components that produce and consume handshake requests in zero time. Latency values denote the

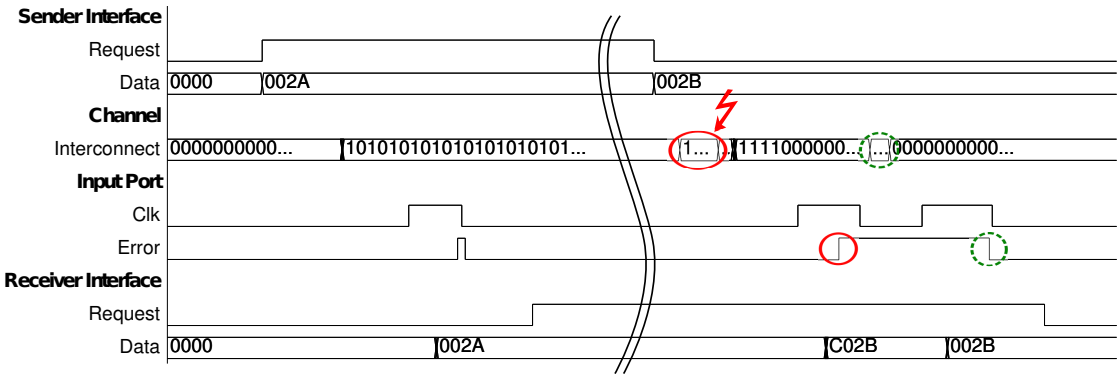


Figure 5.15: Simulation of two transmissions, the first without and the second with a fault.

Table 5.10: Performance & Area Evaluation.

Protocol/ED Code	Data Width (bit)	Throughput (MHz)	Latency (ns)		Area (μm^2)	Coding Efficiency
			no fault	fault		
2-phase/Single Parity	16	855	1.27	1.78	1767	0.47
	32	789	1.43	2.02	3278	0.48
	64	723	1.57	2.23	6523	0.49
4-phase/Single Parity	16	559	1.37	1.92	1642	0.47
	32	524	1.50	2.11	3132	0.48
	64	467	1.73	2.41	6206	0.49
2-phase/Hamming	16	766	1.42	2.08	2340	0.38
4-phase/Hamming	16	513	1.54	2.21	2262	0.38

time it takes for a request transition of the sender to propagate across the I/O ports to the receiver. We have measured latencies both for fault-free and faulty transmissions. In the latter case we only injected faults that vanish before the second sample is taken by the receiver. As can be seen in Table 5.10, the 2-phase links significantly outperform their 4-phase counterparts regarding throughput and latency. This is possible because of the non-existing reset phase. Hence, 2-phase protocols clearly are the better choice for asynchronous communication channels.

5.4 Related Work

In contrast to fault tolerance in GALS modules, which has so far not been addressed by other researchers, there is a good body of existing methods for robust or fault-tolerant asynchronous data transmissions. In this section we want to give a brief presentation of these techniques, with focus on fault tolerance in delay-insensitive communication schemes.

Cheng & Ho

The first paper that addressed transient and permanent faults in 4-phase DI codes has been presented by Cheng and Ho [16]. During the data transmission phase, an error correction mech-

anism is proposed to retrieve the originally sent codeword from a faulty or incomplete input word. In the reset phase a bounded delay approach is assumed since permanent faults might hinder interconnect wires to be reset to zero. If the reset phase does not complete after a certain delay, the receiver still resets the acknowledge signal to finish the handshake cycle. Due to the required delay assumption the authors speak of *semi-delay-insensitive* data communication.

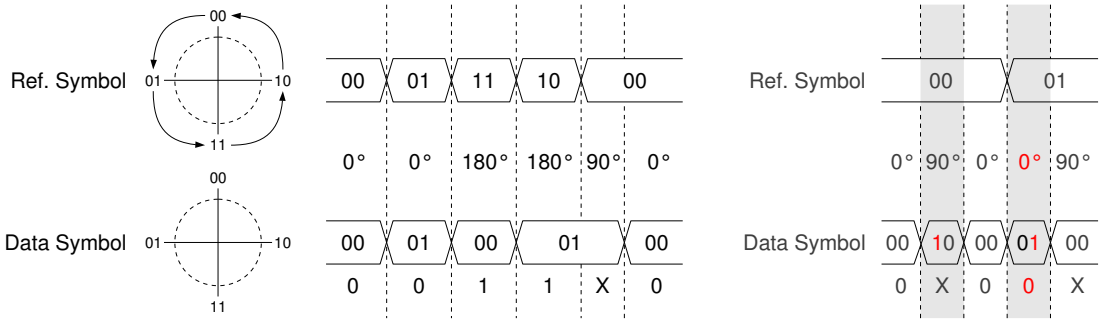
Cheng and Ho distinguish three fault models, depending on what type of transmission faults can occur: In two models only asymmetric (unidirectional) faults are assumed, either $1 \rightarrow 0$, or $0 \rightarrow 1$ faults, whereas the third model describes the general case, where faulty signal transitions can have any polarity. Based on these fault models, the authors then discuss the necessary error correction capabilities for t transmission faults during the data phase. If, e.g., only $1 \rightarrow 0$ faults are assumed, t ones might be missing from the data word (and will never appear in case of permanent faults). Cheng and Ho therefore propose a receiver whose completion detector prematurely triggers when $n-t$ ones are received, and then employs a t-error correcting code to identify the missing bit positions. In the second fault model, which restricts the channel to $0 \rightarrow 1$ faults, t spurious rising transitions could replace t correct transitions, leading to the sampling of a codeword with a maximum of $2t$ erroneous bits. Consequently, the use of a $2t$-error correction code is required. Finally, when arbitrary transmissions faults are allowed, up to $3t$ bits can be erroneous in a codeword. On one hand, the premature processing after the reception of $n - t$ ones, due to possible $1 \rightarrow 0$ faults, contributes to t erroneous bits. On the other hand, it is possible that of these $n - t$ ones, t ones are actually the result of $0 \rightarrow 1$ transmission faults. As argued above, this could lead to additional $2t$ two erroneous bit positions in the sampled incomplete codeword. Therefore, a $3t$-error correction code is mandatory in the general case.

The significant overhead of this solution motivated us to constrain our own research to the mitigation of transient faults. This greatly simplifies the problem since the receiver then can wait until all faults have vanished before the new input data is processed. Consequently, only error *detection* capabilities are needed instead of the more costly support for error *correction*.

Ogg, Al-Hashimi & Yakovlev

In [82] a delay-insensitive link for NoC systems is proposed, which is targeted to be resilient against transient faults. The approach transmits a series of data symbols, one symbol per bit, together with a reference symbol. Since both the reference symbol and the data symbols are transmitted over two rails, $2(n + 1)$ interconnect wires are needed to transmit n bits. The encoding of these symbols is illustrated in Figure 5.16a. The reference symbol changes its value in a circle with a predefined sequence of $00 \rightarrow 01 \rightarrow 11 \rightarrow 10$ and then starts over with the first value. The data symbols can follow the steps of the reference symbol or make a step in the opposite direction. If a data symbol assumes the same value as the reference, it is said to be in-phase and thereby encodes a zero bit. If it assumes a value opposite to the reference symbol's position on the circle, the data symbol is 180° out of phase. This encodes a bit value of one. Note that for every move in the coding circle only one rail has to changed. This scheme therefore forms a 2-phase delay-insensitive code, like LEDR, and the receiver can easily perform completion detection by waiting on a single signal transition for every data symbol and the reference symbol. Figure 5.16a shows five fault-free data transmissions. The transmitted bit value of the data symbol is shown at the bottom row. In the final transmission the reference

symbol switches first, whereas the value change of the data symbol is delayed. During this time the receiver sees a 90° out of phase relationship between the data and reference.



(a) Fault-free transmission (cf. [82]). (b) Potential decode error.

Figure 5.16: DI data transmission with reference symbol for completion detection.

The fact that such 90° phase relationship can also occur when a single fault affects one rail is the foundation for the scheme's resilience against transients. This can be seen in the left grey area in Figure 5.16b, where a faulty one appears on the first rail of the data symbol. In this case the receiver notices the 90° phase relationship, and waits for the transient fault to vanish. This scheme, however, is not bulletproof. Although Ogg, Al-Hashimi & Yakovlev analyse that the receiver is prone to metastability when a faulty signal transition affects the setup/hold window of the input flip-flops, they do not discuss possible scenarios, where a fault can cause a symbol to have a proper 0° or 180° phase relationship, but still encodes an incorrect value. This case is illustrated in the second grey area in Figure 5.16b. The reference symbol changes to the new value 01 and before the correct rail of the data symbol makes a transition a fault affects the other rail. This leads to a correct phase relationship of 0°, however, a wrong value is decoded.

Agyekum & Nowick

In [3, 4] *Zero-Sum codes* are introduced, which can be used both as delay-insensitive and error-correcting codes. To achieve delay-insensitivity Zero-Sum codes are employed in 4-phase handshakes and work in a similar fashion like Berger codes, where check bits encode the number of zeros of the data word. Zero-Sum codes are systematic and codewords therefore consist of a data and a check field. The special trait of a Zero-Sum code is the use of indices, assigned for every bit position, both in the data and in the check field. This can be seen in Table 5.11, which shows the code for 3-bit wide data words. The check bit indices are assigned powers of two in increasing order, starting with one for the right-most check bit. Likewise data bit indices are given increasing values, which are not powers of two, starting from three for the right-most data bit. The value of the check field in a codeword then is set to the sum of index values, where bits of the data field are zero. Consider, e.g., the data word 010 in Table 5.11, which has two zeros at bit positions with the index 3 and 6. Consequently the check field value equals the binary representation of the number $3 + 6 = 9$.

Table 5.11: 3-bit Zero-Sum code.

	Data field			Check field			
Indices	6	5	3	8	4	2	1
Codewords	0	0	0	1	1	1	0
	0	0	1	1	0	1	1
	0	1	0	1	0	0	1
	1	0	0	1	0	0	0
	0	1	1	0	1	1	0
	1	0	1	0	1	0	1
	1	1	0	0	0	1	1
	1	1	1	0	0	0	0

In [3] Agyekum & Nowick prove that Zero-Sum codes are unordered and can therefore be used for delay-insensitive communication. Furthermore, they show that the code is able to correct single-bit errors and can detect all double-bit errors. Error correction is based on the computation of a syndrome, like in Hamming codes. This syndrome simply is the absolute difference of the check field C', recomputed by the receiver, and the check field C, which was appended by the sender: $S = |C' - C|$. In case of a fault-free transmission the syndrome obviously is zero. If a single-bit flip occurred, the syndrome value equals the index of the erroneous bit position, due to the construction of the code. This allows for simple error correction.

However, we did a careful analysis of the error correction mechanism in the context of delay-insensitive data transmission and uncovered that a single fault still can trick the receiver into decoding erroneous codewords. Like in our example for m-of-n codes in Section 5.1.1, a single faulty signal transition can turn an intermediate transmission pattern into a complete codeword, which is different from the sender's codeword. Let us consider the codewords $a = 001\,1011$ and $b = 010\,1001$ of the 3-bit Zero-Sum presented in Table 5.11. Suppose the interconnect wires have been reset with the spacer codeword, and the sender wants to transmit a. Since no assumptions about the wire delays are made in a DI scheme, rising transitions may appear in any order. Table 5.12 shows a sequence, where a fault on bit d_1 turns the incomplete codeword a into the complete codeword b. If this state persists long enough, the receiver's completion detector will fire and codeword b will be processed and acknowledged instead of a.

Table 5.12: Transmission fault on bit d_1.

Transmission Pattern	Data field			Check field				Description
	d_2	d_1	d_0	c_3	c_2	c_1	c_0	
1	0	0	0	0	0	0	0	Spacer codeword
2	0	0	0	1	0	0	1	Correct signal transitions on fast wires
3	0	1^*	0	1	0	0	1	Faulty signal transition on d_1
4	0	1	1	1	0	1	1	Correct signal transitions on slow wires

In Section 5.1.3 we stipulated that the number of non-overlapping bit positions between any two codewords of a fault-tolerant DI code C needs to be greater than the number of faults f,

which should be mitigated: $u(x, y) > f$ and $u(y, x) > f,\ \forall x, y \in C$. In case of the codewords a and b, it can be seen that $u(b, a) = 1$, and therefore a single fault is already critical.

In [4] Agyekum & Nowick extend their previous work and present new codes called Zero-Sum$^+$ and Zero-Sum*, which have improved error correction/detection capabilities. Nevertheless, these codes still suffer from the same deficiency as presented above, and cannot guarantee error-free DI communication in the presence of single transmission faults.

Zhang et al.

An interesting solution to build QDI interconnects that are resilient against transient faults was proposed in [126]. The presented scheme is called *Delay-insensitive Redundant Check* code (*DIRC*) and can be applied to 1-of-n codes. In contrast to our approach the sender does not add check bits to the single-rail data word, but computes a checksum over a group of delay-insensitive codewords, before they are transmitted over an unreliable interconnect. Such a group consists of two or more 1-of-n codewords, $x_0, x_1, \ldots, x_{CN-1}, CN \geq 2$, and the check word c is provided by an additional 1-of-n codeword, which simply is the sum of all codewords x_i in the group:

$$c = \sum_{i=0}^{CN-1} x_i. \tag{5.2}$$

To compute this checksum Zhang et al. define some basic arithmetic rules for 1-of-n codewords. Formally, a codeword is denoted as $D^n(i)$, where n equals the codeword length, and the parameter i, $0 \leq i < n$, identifies which bit is set to one ($i = 0$ denotes the right-most bit in the codeword, and $i = n - 1$ the left-most bit). In a 1-of-4 code the codeword 0010, e.g., is represented by $D^4(1)$. Based on this notation the arithmetical rules are formulated as follows:

$$D^n(a) = D^n(a \bmod n) \tag{5.3}$$

$$-D^n(a) = D^n(-a) = D^n(-a \bmod n) = D^n(n - a) \tag{5.4}$$

$$D^n(a) + D^n(b) = D^n((a + b) \bmod n) \tag{5.5}$$

Example 5.1. *Assume a group consists of the two codewords $D^4(1) = 0010$ and $D^4(3) = 1000$ ($CN = 2$). Then the check word equals $D^4(1) + D^4(3) = D^4(0) = 0001$.*

Faults can turn a correct 1-of-n codeword into an erroneous m-of-n word, with $m > 1$. To perform error correction at the receiver it is therefore necessary to define the above arithmetic rules also for the general case where multiple bits are one in a codeword. The index vector $A^m = (a_0, a_1, \ldots a_{m-1})$ is used to identify the bits that are set to one, and an m-of-n codeword can consequently be represented as $D^n(A^m) = D^n(a_0, a_1, \ldots a_{m-1})$. m-of-n codewords can also be understood as the union of 1-of-n codewords:

$$D^n(A^m) = D^n(a_0, a_1, \ldots a_{m-1}) = \bigcup_{i=0}^{m-1} D^n(a_i) \tag{5.6}$$

The basic arithmetic rules for 1-of-n codewords can therefore be extended as follows:

$$-D^n(A^m) = -\bigcup_{i=0}^{m-1} D^n(a_i) = \bigcup_{i=0}^{m-1} [-D^n(a_i)] = \bigcup_{i=0}^{m-1} D^n(-a_i) \tag{5.7}$$

$$D^n(A^m) + D^n(B^{m'}) = \bigcup_{i=0}^{m-1} D^n(a_i) + \bigcup_{j=0}^{m'-1} D^n(b_j) = \bigcup_{i=0}^{m-1} \bigcup_{j=0}^{m'-1} D^n(a_i + b_j) \tag{5.8}$$

For the error correction process, the receiver can recompute the value of all transmitted codewords x_i from the other codewords of the group, $x_j, 0 \leq j < CN, j \neq i$, and received check word c. These recovered versions of the codewords are denoted x'_i:

$$x'_i = c - \sum_{j=0, j\neq i}^{CN-1} x_j \tag{5.9}$$

If there was no fault, the received codeword x_i and the recomputed x'_i will be identical. On the other hand, if a fault is located in x_i, only x'_i will provide the correct value of the codeword. In case a fault occurred in some other codeword or in the check word, clearly the recomputed x'_i will be erroneous, but then in turn the received x_i will be correct. In other words, a fault either affects x_i or x'_i, but never both. Therefore, error-free codewords x''_i can be easily generated at the receiver by applying a bitwise C-element operation on x_i and x'_i. This allows correct signal transitions to propagate, whereas faults will be filtered. A DIRC code is able to mitigate all single transient faults and all kinds of multiple transients, as long as they are confined to a single codeword x_i or only occur in the check word.

Example 5.2. *Consider the two codewords* $x_0 = 0010 = D^4(1)$ *and* $x_1 = 1000 = D^4(3)$. *Hence, the check word* c *equals* $D^4(1) + D^4(3) = D^4(0)$. *Due to a transient fault,* x_0 *is transformed into* $1010 = D^4(3, 1)$. *Based on this input the receiver computes* x'_0 *and* x'_1*:*

$$x'_0 = c - x1 = D^4(0) - D^4(3) = D^4(1) = 0010$$

$$x'_1 = c - x0 = D^4(0) - D^4(3, 1) = D^4(3, 1) = 1010$$

Finally the correct codewords can be recovered ("$\odot$*" is a bitwise C-element operation):*

$$x''_0 = x_0 \odot x'_0 = 1010 \odot 0010 = 0010$$

$$x''_1 = x_1 \odot x'_1 = 1000 \odot 1010 = 1000$$

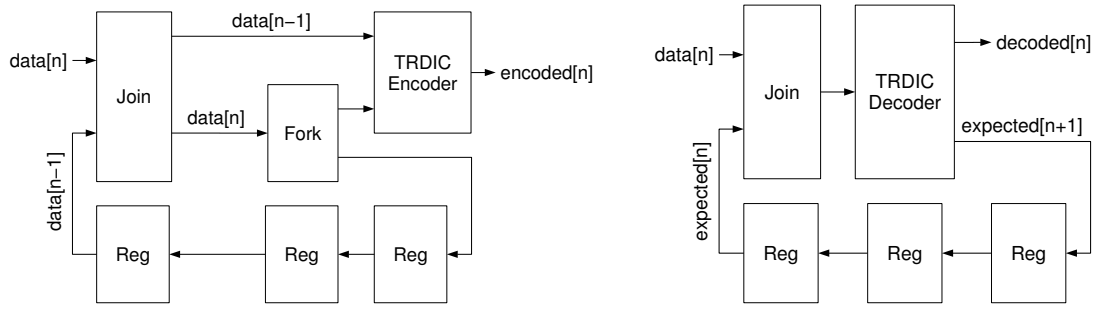
While the concept of DIRC codes is very elegant since error correction is directly performed on delay-insensitive codewords, the overall coding efficiency is rather low. In the paper Zhang et al. present and analyse circuit implementations using 1-of-2 and 1-of-4 codes and group size CN of 2. For this specific scenario the coding efficiency is 0.33, i.e., for transmitting 4 bits of data there are 12 wires required. Coding efficiency can be improved with larger codeword groups, i.e., when $CN > 2$. This, however, increases the complexity of encoder and decoder circuits due to the higher number of adders required for computing sums of 1-of-n codewords.

Pontes, Calazans & Vivet

Another interesting approach, presented in [89] by Pontes, Calazans & Vivet, uses temporal redundancy to mitigate single event effects in delay-insensitive links. Hence, the authors speak of a *Temporally Redundant Delay Insensitive Code* (*TRDIC*). The technique adapts 1-of-n codes, which have little resilience as a single transient fault is enough to generate a complete codeword, by transforming them into 2-of-(n+1) codes. The sender, which would originally send 1-of-n codewords, combines the current 1-of-n codeword and the one previously sent into a 2-of-(n+1) codeword, which is then transmitted to the receiver. This can simply be done with a bitwise OR-operation between the current and the previous codeword. The extra bit to produce a codeword of length $n + 1$ is used to indicate if the current and previously sent codeword are identical.

Example 5.3. *Let us consider a 1-of-4 code. Assume that the current and the previously sent codeword are* $d_i = 0100$ *and* $d_{i-1} = 0001$*, respectively. The output of the TRDIC encoder then is* $0 \,\&\, (0100 \vee 0001) = 00101$*, where "&" denotes concatenation of two bit strings. In case* $d_i = d_{i-1} = 0100$*, the resulting 2-of-5 codeword is* $1 \,\&\, 0100 = 10100$*.*

With this encoding scheme, in fact, two 1-of-n codewords are transferred per transmission, and every codeword is transferred twice in successive transmissions. This way the receiver can buffer a codeword upon the first transmission and compare it with the replay version in the subsequent transmission. Figure 5.17 shows TRDIC encoder and decoder circuits.



(a) Encoder. (b) Decoder.

Figure 5.17: Encoder and decoder circuits for TRDIC links.

In the failure model assumed by Pontes et al. the receiver only needs to decode codewords, where all correct *ones* are available and faults only lead to additional erroneous *ones*. Conse-

quently, decode errors can easily be filtered with a bitwise double checking approach between consecutive versions of the same codeword using C-elements. When faults only lead to codewords with superfluous *ones*, this is a much more benign scenario compared to the general case considered in our work, where the receiver also has to deal with incorrect input data, encoded by correct m-of-n codewords. In [89] these two situations are named *Invalid Corrupted Data* (*ICD*) and *Valid Corrupted Data* (*VCD*) errors, respectively.

To enforce that only ICD errors can occur a timing assumption is necessary, which limits the maximum skew between the slowest and the fastest rail of the 2-of-(n+1) codewords. If this skew is smaller than the processing time of the incoming codeword, then it can be guaranteed that all valid input transitions have been sampled. A detailed analysis how faults impact data transmissions with m-of-n codes and a reliability evaluation of the proposed TRDIC scheme can be found in [89]. Even though the number of decode errors is reduced significantly, the authors point out that the used double checking is not capable to mitigate ICD errors in all cases. Resilience could, however, be further improved by a more elaborate three stage decoding.

Another approach by Pontes, Calazans & Vivet, published in [90], employs information redundancy instead of temporal redundancy. The presented solution is targeted for fault-tolerant DI communication in asynchronous *Networks on Chip* (NoCs), where data is transmitted in packets built from a certain number of *flits*. The data transferred in each flit is encoded with a series of m-of-n codewords, which are then organised in a flit matrix, each row being one codeword. To introduce redundancy parity check bits are then computed, one bit for each column of the flit matrix. The values of the check bits are simply assigned the modulo-2 sum of all bits of the respective column, like in a regular parity code. Finally the check bits are encoded with m-of-n codewords and transmitted over the NoC links together with the regular data words.

At the receiver the incoming codewords are again arranged in matrix form. An ICD error can then be easily identified as a row in the matrix that contains more than m *ones*, i.e., a row with invalid m-of-n codeword. The receiver then recomputes the parity bits for each column and compares them with the transmitted parity information. A mismatch between recomputed and transmitted parity bit clearly indicates an erroneous *one* in the respective column. Combining the information of column and row errors, the erroneous bit in the flit matrix can be located and corrected. In contrast for VCD errors the matrix does not contain rows with invalid codewords. However, if *ones* appear in wrong positions and are missing in others, multiple columns will have parity mismatches. Since bit errors in such a scenario can only be detected but not corrected, it is proposed that the receiver requests a retransmission to resolve the situation.

Shi et al.

The SpinNNaker system is a massively parallel computation platform, where processor nodes use delay-insensitive codes for on-chip and off-chip communication. In [99] methods for fault-tolerant inter-chip communication are proposed. In contrast to all approaches we have discussed above, the goal is not to protect the integrity of the transmitted data but the integrity of the communication protocol. The purpose of all fault tolerance extensions introduced in the sender and receiver components is to avoid deadlocks, which would crash a communication link. Detecting and correcting errors in the transferred data stream needs to be performed by other mechanisms.

CHAPTER 6

Summary & Conclusion

In this thesis we proposed and evaluated fault tolerance techniques in the context of GALS circuits. Our main target was to improve resilience against soft errors, which are currently the main contributor to failures of integrated circuits.

In the first part of the thesis we explored the use of modular redundancy to provide reliable computation in locally synchronous modules, the core building blocks in a GALS system. These modules are characterised by a great degree of independence and can be considered as separate design units, both with respect to timing as well as physical implementation. The principal aim of the fault tolerance techniques we introduced in this thesis was to preserve this independence as much as possible. Therefore, we proposed two modular redundant circuit architectures where critical modules are replicated as independent, self-sufficient GALS modules. Consequently, module copies in this architecture have their own stoppable clock generator and clock distribution network, which has far-reaching implications for the implementation of voting and state restoration mechanisms. Since the replicated circuits do not share a single clock domain, voters can no longer be part of the regular data path. In our first attempt to design a suitable recovery circuitry voters are therefore shifted to separate feedback paths, which are only activated for certain checkpoints to overwrite possibly erroneous values in flip-flops. During these checkpoints asynchronous recovery controllers, which can turn on and off the clock generators, are used to synchronize module copies. This mechanism allows state restoration to be safely performed for otherwise independently clocked GALS-modules. Since all flip-flops are recovered in parallel during a single clock cycle, we used the name *parallel approach* throughout the thesis.

Even though this solution has several benefits like a single-cycle recovery latency, lightweight recovery controllers, no single points of failures, etc., the parallel voting scheme requires many interconnect wires that run between replicated circuit parts. This imposes a tight coupling of the modules and the physical implementation, i.e., place & route has to be performed like for a single circuit block. To reach our ultimate goal of real physical independence for module replicas, we consequently looked into an alternative approach to conduct state restoration.

The second solution presented in this thesis uses a serial recovery process, which is based on scan chains to gain read/write access to the internal module registers. Although this scheme

results in significantly longer recovery latencies, the technique offers some compelling benefits in comparison to the previous method and conventional modular redundant circuits. In particular, we want to highlight the *non-intrusiveness* of this approach, which allows for simple and seamless reuse of existing non-redundant circuit implementations. Recall that no internal voters or other recovery-related components need to be added to the original circuit. The only prerequisite is the availability of an adequate scan chain or multiple scan chains, which include all relevant internal flip-flops. The circuit can then be replicated at module level without a single modification, either based on the RTL code, a synthesised netlist or even a fully-placed macro cell. The only required extension are separate recovery controllers, which need to be connected to the scan chain signals and can then take full responsibility for the execution of the recovery process. State exchange during recovery is performed over asynchronous links, like it is the case for the regular data ports of the module. With this architecture replicated modules can finally be viewed like ordinary GALS modules, which are locally clocked and communicate with *all* other modules, including the functionally equivalent replicas, over dedicated asynchronous channels. Replicated circuits can therefore be placed with great flexibility, e.g., in different parts of the die, supplied by independent power and ground rails, on different chips or in extreme cases even on different boards. This kind of flexibility can usually only be provided, if hardware redundancy is introduced at system level, where recovery schemes to deal with failed components rely on specialised protocols like TTP (time-triggered protocol, [55]). In contrast, our solution provides hardware fault tolerance at gate level and is therefore completely transparent to the designer at register transfer level and above.

In the second part of the thesis we addressed fault-tolerant communication across asynchronous delay-insensitive channels. Mitigation of faults in this scenario can be quite tricky and many existing solutions fail to provide full resilience even for single faults (while maintaining full delay-insensitivity). To approach this problem we first pursued a thorough theoretical analysis of the susceptibility of delay-insensitive codes to transmission faults. The main premise of this analysis was to find the number of decode errors that could occur for a certain number of faults. Based on these results we then proposed to associate delay-insensitive codes with classical error detecting codes, which provide the required redundancy to identify decode errors. The use of delay-insensitive and error detecting codes often introduces high overheads for communication links and combining them can be prohibitively expensive for real circuit applications. Consequently, it was essential to find delay-insensitive codes that can already offer a good resilience against faults so additional overheads can be kept at a minimum. It was very interesting to see that this is the case for various m-of-n codes, whereas Berger codes have very little resilience. For encoding four data bits, e.g., either a 3-of-6 code or a Berger code with a 4-bit data field could be used. While a single transmission fault in case of a 3-of-6 code can only cause two decode errors in the worst case, the use of a Berger code can result in three or even four decode errors. A 3-of-6 code can therefore be combined with a regular Hamming code, which results in a reasonable information rate (especially for higher bit widths).

Based on this theoretical analysis, we have developed encoder and decoder components, which can be used as I/O ports in GALS-based circuit architectures. The components are specifically designed for the mitigation of transient faults, and therefore rely on the knowledge that a detected fault will vanish after a short time. The reception of new input data is blocked until

both completeness and correctness have been established. Consequently, the end-to-end communication latency can vary in the presence of transient faults. This, however, is not an issue for a system with delay-insensitive channels, which do not impose any timing constraints.

Like in the first part of the thesis, we have again presented two different solutions, one that is specifically targeted for 4-phase dual-rail codes, and a generic link architecture. While the output port implementations of both approaches are almost identical, the receiver-side components follow a slightly different methodology. In the first solution input words are only sampled once and potential decode errors need to be corrected with the help of the combined redundancy of the used 1-of-2 code and the parity-protection of the transmitted data word. Since a single transmission fault in case of a dual-rail code can only lead to a single-bit decode error, the combination of dual-rail and parity code provides enough redundancy to restore the correct data word. In case of more complex DI codes, however, we have shown that two or more decode errors can occur even for single transmission faults. Since *correction* of multiple errors requires much stronger and more expensive codes, we followed an alternative strategy for the second link architecture. Here error correcting is achieved by re-sampling the input word until the transient fault has vanished on the communication channel. Therefore, the used codes only need to provide enough redundancy to *detect* all possible decode errors instead of having the capability to correct them. This results in better information rates and more efficient I/O port implementations.

CHAPTER 7

Future Work

As it is the case for many research projects, at the end there are often interesting questions remaining, which could not be fully addressed in the given time. The same thing can be said about this PhD thesis as we have gained many new insights and ideas from the results gathered and lessons learnt during the engagement with the topic. In this section we therefore want to give some pointers about possible future research directions, which might be worthwhile to explore.

7.1 Mesochronous Modular Redundant Systems

Stoppable clock generators provide an elegant solution to data synchronization problems. However, the design of robust ring oscillators with low jitter also constitutes a major challenge for practical circuit implementations [109]. Therefore, it might be interesting to apply the replication and recovery strategies, as presented in this thesis, to other types of GALS systems, which do not rely on on-chip stoppable clock generators. Consider when clock frequencies of GALS modules are not chosen arbitrarily but are related in some way. We already mentioned mesochronous systems in Section 3.2.2, where separate components are clocked with the exact same frequency and only the phase offset is unknown. Another example are *ratiochronous* systems with rationally related clocks, or *plesiochronous* systems, where clock frequencies are nearly the same, except for a small mismatch, leading to a slightly varying, non-constant phase relationship. In these cases other synchronization mechanisms can be employed instead of stoppable clocks, as pointed out in [109].

Nevertheless, stopping the clock to freeze the system state during an ongoing recovery process is a vital requirement for our fault tolerance approach. We therefore propose the use of clock gating to attain the same capability. Figure 7.1 shows a slightly modified solution of our TMR-based GALS architecture. As can be seen, all three replicas are clocked by independent external oscillators, which have the same frequency. Assuming that the phase of the clock signals is unknown, the three replicated units form a mesochronous system. Communication between the recovery controllers during a recovery therefore needs to be synchronized. Fortunately, this can

be efficiently done with self-timed FIFOs that compensate for the phase offsets and then communication is possible at full speed every clock cycle (see [37, 109]). Clock gating is simply performed with an AND-gate, which is disabled and enabled by the recovery controller. This can be safely done since the recovery controller is part of the same local clock domain and is therefore able to operate the clock gate synchronously with the clock signal.

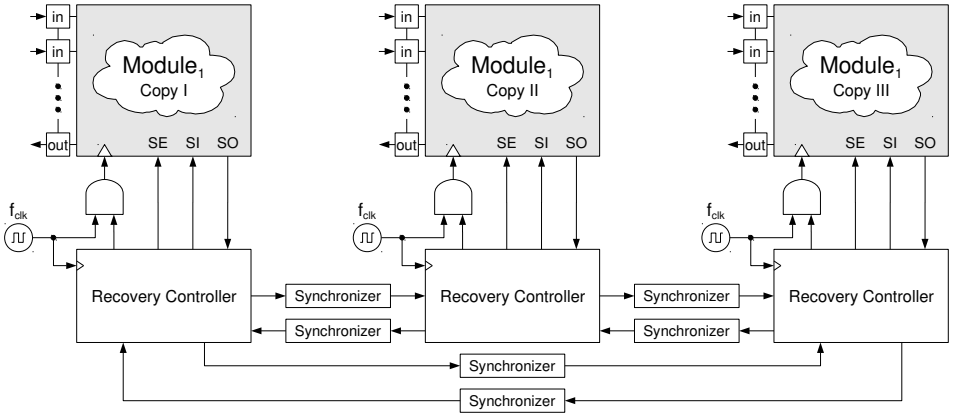


Figure 7.1: Mesochronous replicated GALS modules.

This solution does not sacrifice any of the benefits of the approach presented in this thesis, e.g., minimal physical dependencies between replicated components, and the implementation of the recovery controllers as well as all the general considerations for the recovery process remain the same. With a standard clock gating mechanism instead of on-chip ring oscillators, however, this system architecture might be more readily adopted in industrial projects.

7.2 Reconfigurable GALS Architectures

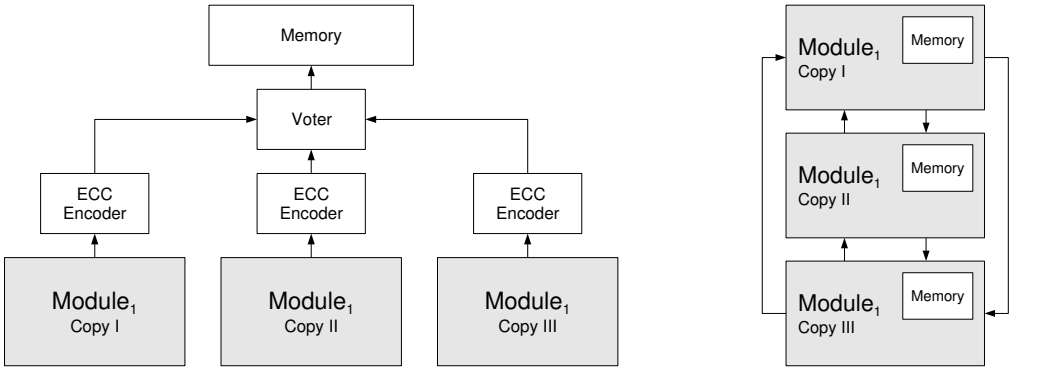
Permanent defects have only been briefly addressed in this thesis. For long-term missions where maintenance is not possible or very expensive, techniques to deal with defects are mandatory. Research in reconfigurable GALS architectures might be a promising follow up project to this thesis to incorporate resilience against permanent faults. Existing FPGA architectures based on GALS, presented in [48, 49, 95], could be a good starting point for this undertaking. These FPGAs are built from a regular array of locally synchronous islands, which are reconfigurable like in conventional FPGAs, and a globally asynchronous interconnect architecture. We propose to extend such an FPGA architecture with *built-in self test* and *dynamic reconfiguration* mechanisms to allow for online reorganisation of islands in case of permanent defects. If a defective island is detected, its configuration can be transferred to a spare island and input and output signals need to be rerouted. We believe this rerouting is significantly simplified in an asynchronous interconnect architecture, where the proper circuit function does not depend of strict timing constraints like in conventional synchronous FPGAs. The crucial question that needs to be investigated is the size of reconfigurable modules and the granularity of reconfiguration in case of permanent defects. To avoid high reconfiguration overheads, it might be necessary to

try local intra-module reorganisations to bypass the fault, before an entire module is relocated. Various approaches to perform, e.g., column- or tile-based reconfiguration have been presented in [1, 7, 43, 57]. Only when local countermeasures fail, spare modules are activated. Another effective recovery mechanism could be *configuration swapping* between two GALS modules with subsequent reconfiguration of the associated interconnect resources. If the defective logic block is used in a different way or not used at all by the swapped configuration, a permanent fault would become dormant and thus would not affect the circuit's functionality.

7.3 Recovery of Memory Cells

In this thesis we have presented two approaches to recover the state of a GALS module stored in registers. However, such modules can also contain memory blocks, e.g., SRAMs, which retain state information. Usually ECC memories are used to protect stored data words from single- or double-bit errors. This is a very powerful solution, but it can, of course, only mitigate corruptions that occur during data retention. If data words are already incorrect when they are written to the memory, encoding with an error correcting code is not effective.

To address this issue we propose to distinguish between *replicated* and *non-replicated* memory architectures, as can be seen in Figure 7.2. In the latter case, which might be employed for bigger memories, where replication might be too costly, there is only one common ECC-protected memory unit. Voting then needs to be performed for every write access to mask erroneous data words, in case a replicated module was corrupted by a soft error. If ECC-protection is applied before voting, ECC encoders and the voter do not even constitute a single point of failure: A transient fault in one of the ECC encoders will be masked by the voter, and a fault in the voter unit can be recovered due to the redundancy introduced by the error correcting code[1].



(a) Non-replicated, with ECC and voting.

(b) Replicated, with recovery links.

Figure 7.2: Memory architectures for replicated GALS modules.

However, voting on output data requires synchronization between replicated modules (see Section 4.4.3), and therefore introduces a performance penalty. For smaller local memory

[1]Obviously, we assume that at most one of the system components is faulty per write operation.

blocks, like fast caches, this penalty might not be acceptable. Thus, a replicated memory architecture could be preferable, where memory blocks are fully replicated as part of the GALS module and read/write operations are only performed locally (see Figure 7.2b). Without voting, however, erroneous computation results can be written into these memories. The recovery process, which is applied to restore erroneous values in registers, therefore needs to be extended to include data words stored in memory blocks. The essential question that remains to be answered is how such a recovery can be performed efficiently. A simple idea to reduce data transfers during recovery is to only include data words that have actually been written since the last recovery. This information can be maintained with dirty bits, like they are used in caches.

7.4 Robust 2-phase Delay-Insensitive Codes

In this thesis we have mainly investigated fault-tolerant data transmissions with 4-phase delay-insensitive codes. For communication links, however, 2-phase protocols are often used because of their superior throughput and power-efficiency. It would therefore be interesting to extend our results and methodologies for DI NRZ codes, like those presented in [14, 70].

7.5 Comprehensive Evaluation of Robust DI Channels

In Section 5.1 we have analysed the resilience of various m-of-n codes and Berger codes against faults and determined required error detection capabilities to mitigate decode errors. The link implementations, introduced in Sections 5.2 and 5.3, however, only focused on dual-rail codes. While these codes offer a decent coding efficiency and moderate complexity for encoder, decoder and completion detection circuits, other DI codes like 3-of-6, 2-of-7 or Berger codes can still provide better solutions in terms of coding or power efficiency. Therefore, it would be worthwhile to perform a comprehensive study of link implementations using the full variety of analysed codes. Evaluation criteria should be information rates, circuit area of input and output ports, data throughput and also power efficiency, which was not addressed at all in this thesis.

In the analysis of m-of-n codes performed in this thesis we have tried to find encoding functions that would minimise the required redundancy of a complementary ED code to perform error detection. The choice of this encoding function, however, also influences the complexity of encoder, decoder and completion detection circuits. In [9], e.g., Bainbridge et al. discuss how to find suitable data-to-DI code mappings to reduce hardware overheads. Based on these results further research work could be done to find encoding functions that are efficient both in terms of code redundancy and in terms of the required hardware resources for DI circuit components.

APPENDIX A

Additional Resources

A.1 Scripts for Fault-Injection Experiments

For our fault injection runs we used the ModelSim® HDL simulator and the *force* command for manipulating signal values. Listings A.1 and A.2 show the Tcl scripts that inject faults in specific nets or flip-flop/latches. While the procedure is quite generic for nets, in case of injecting upsets in flip-flops and latches the code needs to be customised to the specific elements available in the targeted technology library and their simulation models. Note that simulation models of flip-flops/latches in standard cell libraries typically are abstract models, written in VHDL or Verilog. Their purpose is to model the input/output behaviour (function and timing) of the respective cells. They do not contain an accurate representation of the internal cell structure and therefore it is not possible to directly manipulate the stored value in a way that corresponds to the real physical process when an upset occurs (i.e., by forcing the nets of the storage loop to a certain value). Instead we had to fiddle with inputs of the cell models so that the stored value would be changed. For flip-flop and latches this can simply be done in two steps: 1) Tie the cell's data input to the value one wants to inject and 2) force a superfluous pulse on the cell's clock or enable inputs so this value would be latched (cf. Listing A.2).

Listing A.1: Inject transient pulse on net.

```
1 proc upset_net {net_path transient_length upset_value} {
2   set val $upset_value
3
4   # Force net to specified value for certain amount of time
5   force -freeze $net_path $val 0 -cancel $transient_length
6 }
```

Listing A.2: ModelSim/Tcl Code to upset flip-flops and latches.

```
1 proc upset_ff_latch {instance_path setup_time upset_length upset_value} {
2   set val $upset_value
3
4   # force data input signal to upset value
```

```
  force -freeze "$instance_path/d" $val 0 -cancel $upset_length

  # disable synchronous/asynchronous control signals
  if {[find signals "$instance_path/rb"] != ""} {
    force -freeze "$instance_path/rb" 1 0 -cancel $upset_length
  }
  if {[find signals "$instance_path/sb"] != ""} {
    force -freeze "$instance_path/sb" 1 0 -cancel $upset_length
  }
  if {[find signals "$instance_path/ld"] != ""} {
    force -freeze "$instance_path/ld" 1 0 -cancel $upset_length
  }
  ...

  # activate clock signal for flip-flops
  if {[find signals "$instance_path/ck"] != ""} {
    force -freeze "$instance_path/ck" 1 $setup_time -cancel $upset_length
  }
  if {[find signals "$instance_path/ckb"] != ""} {
    force -freeze "$instance_path/ckb" 0 $setup_time -cancel $upset_length
  }

  # activate enable signal in case of latches
  if {[find signals "$instance_path/g"] != ""} {
    force -freeze "$instance_path/g" 1 $setup_time -cancel $upset_length
  }
  if {[find signals "$instance_path/gb"] != ""} {
    force -freeze "$instance_path/gb" 0 $setup_time -cancel $upset_length
  }
}
```

A.2 Reliability Evaluations

Listings A.3 and A.4 show the Matlab scripts we used for our reliability evaluations presented in Section 4.6.3. Note that all constant area and performance values used in the scripts are motivated from the modular redundant SCARTS processor designs we have developed.

Listing A.3: Compute and plot MTTF of conventional TMR systems.

```
% sweep component MTTF = 10h to 1000h
compMTTF = 10:20:1000;
lambda_m = 1./compMTTF;

% set clock period in s
clk = 2.59*10^-9;

% contribution values to total SER
rho_clk = [0.01, 0.0001, 0.000001, 0.00000001];
rho_comb(1:4) = 11/(11+49);
rho_seq = 1 - (rho_clk + rho_comb);

% area overhead
```

```
a_voters = 0.14;

% scaling factor to convert hours to clock periods
s = 60*60/clk;

% compute system MTTF
mttf = zeros(length(lambda_m), length(rho_clk));
for i = 1:length(lambda_m)
    for j = 1:length(rho_clk)
        lambda_r = lambda_m(i)*rho_seq(j) +
            lambda_m(i)*rho_comb(j)*(1+a_voters);
        lambda_rs = lambda_r/s;
        lambda_c = 3*lambda_m(i)*rho_clk(j);
        lambda_cs = lambda_c/s;
        mttf(i,j) = 1/(3*lambda_rs^2 + lambda_cs);
    end
end

% convert results to hours
mttf = mttf./s;

% generate plot
linespec = {'-r*','-bo','-ms','-gv', '-cx'};
legendStrings = cell(length(rho_clk), 1);
for i = 1:length(rho_clk)
  legendStrings{i} = sprintf('rho\\_clk = %g%%', rho_clk(i)*100);
  semilogy(compMTTF, mttfApprox(:,i), linespec{i});
  hold on
end
hold off
grid on
set(gca, 'YMinorGrid', 'off')

legend(legendStrings, 'Location', 'NorthEast');
title('\fontsize{14}Conventional TMR System');
xlabel('Simplex MTTF (h)');
ylabel('TMR system MTTF (h)');
```

Listing A.4: Compute and plot MTTF of TMR systems with periodic recovery.

```
  % sweep component MTTF = 10h to 1000h
  compMTTF = 10:20:1000;
  lambda_m = 1./compMTTF;

  % contribution values to total SER
  rho_clk = 0.01;
  rho_comb = 11/(11+49);
  rho_seq = 1 - (rho_clk + rho_comb);

  % set clock period in s
  clk_par = 2.5*10^-9;
  clk_ser = 2.24*10^-9;

  % area overheads
```

```
a_recComb_par = 0.515;
a_recSeq_par = 0.004;
a_recClk_par = 0.006;
a_recComb_ser = 0.047;
a_recSeq_ser = 0.061;
a_recClk_ser = 0.039;

% recovery latency
d_rec_par = 1;
d_rec_ser = 1463;

% scaling factor to convert hours to clock periods
s_par = 60*60/clk_par;
s_ser = 60*60/clk_ser;

% sweep different values for recovery-related performance overhead
rec_overhead = [0.001, 0.01, 0.05, 0.1];
d_comp_par = d_rec_par./rec_overhead;
d_comp_ser = d_rec_ser./rec_overhead;

% compute system MTTF
mttf_par = zeros(length(lambda_m), length(rec_overhead));
mttf_ser = zeros(length(lambda_m), length(rec_overhead));
for i = 1:length(lambda_m)
    for j = 1:length(rec_overhead)
        lambda_r_par = lambda_m(i)*(rho_seq*(1+a_recSeq_par) +
          rho_comb*(1+a_recComb_par) + rho_clk*(1+a_recClk_par));
        lambda_rs_par = lambda_r_par/s_par;
        lambda_r_ser = lambda_m(i)*(rho_seq*(1+a_recSeq_ser) +
          rho_comb*(1+a_recComb_ser) + rho_clk*(1+a_recClk_ser));
        lambda_rs_ser = lambda_r_ser/s_ser;

        mttf_par(i,j) =
          1/( s_par * lambda_rs_par^2 * (3*d_comp_par(j) + 9*d_rec_par) );
        mttf_ser(i,j) =
          1/( s_ser * lambda_rs_ser^2 * (3*d_comp_ser(j) + 9*d_rec_ser) );
    end
end

% generate plot
linespec_par = {'-r*','-ro','-rs','-rv','-r+'};
linespec_ser = {'-b*','-bo','-bs','-bv', '-b+'};
legendStrings = cell(2*length(rec_overhead), 1);
clk_overhead = clk_par/clk_ser - 1;
for i = 1:length(rec_overhead)
  legendStrings{i} = sprintf('perf\\_oh = %.2f%%',
    ((1+clk_overhead)*(1+rec_overhead(i)) - 1) * 100);
  semilogy(compMTTF, mttf_par(:,i), linespec_par{i});
  hold on
end
for i = 1:length(rec_overhead)
  legendStrings{i+length(rec_overhead)} = sprintf('perf\\_oh = %.2f%%',
    (rec_overhead(i)) * 100);
```

```
68      semilogy(compMTTF, mttf_ser(:,i), linespec_ser{i});
69    end
70    hold off
71    grid on
72    set(gca, 'YMinorGrid', 'off')
73
74
75    legend(legendStrings, 'Location', 'SouthEast');
76    title('\fontsize{14}Parallel␣and␣Serial␣TMR␣Approaches');
77    xlabel('Non-redundant␣module␣MTTF␣(h)');
78    ylabel('TMR␣system␣MTTF␣(h)');
```

Bibliography

[1] M. Abramovici, C.E. Stroud, and J.M. Emmert. Online bist and bist-based diagnosis of fpga logic blocks. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 12(12):1284–1294, 2004.

[2] A. Agarwal and J. Lang. *Foundations of Analog and Digital Electronic Circuits*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2005.

[3] M.Y. Agyekum and S.M. Nowick. An error-correcting unordered code and hardware support for robust asynchronous global communication. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 765 –770, march 2010.

[4] M.Y. Agyekum and S.M. Nowick. Error-correcting unordered codes and hardware support for robust asynchronous global communication. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 31(1):75 –88, jan. 2012.

[5] A. Ajane, P.M. Furth, E.E. Johnson, and R.L. Subramanyam. Comparison of binary and lfsr counters and efficient lfsr decoding algorithm. In *Circuits and Systems (MWSCAS), 2011 IEEE 54th International Midwest Symposium on*, pages 1 –4, aug. 2011.

[6] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on*, pages 414 –425, jun 1990.

[7] A. Antola, V. Piuri, and M. Sami. On-line diagnosis and reconfiguration of fpga systems. In *Electronic Design, Test and Applications, 2002. Proceedings. The First IEEE International Workshop on*, pages 291–296, 2002.

[8] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, 2004.

[9] W.J. Bainbridge, W. B. Toms, D.A. Edwards, and S.B. Furber. Delay-insensitive, point-to-point interconnect using m-of-n codes. In *Asynchronous Circuits and Systems, 2003. Proceedings. Ninth International Symposium on*, pages 132–140, May 2003.

[10] R. C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *Device and Materials Reliability, IEEE Transactions on*, 5(3):305–316, 2005.

[11] E. Beigne, F. Clermidy, S. Miermont, and P. Vivet. Dynamic voltage and frequency scaling architecture for units integration within a gals noc. In *Networks-on-Chip, 2008. NoCS 2008. Second ACM/IEEE International Symposium on*, pages 129–138, April 2008.

[12] M. Blaum and J. Bruck. Unordered error-correcting codes and their applications. In *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, pages 486 –493, jul 1992.

[13] B.H. Calhoun, Yu Cao, Xin Li, Ken Mai, L.T. Pileggi, R.A. Rutenbar, and Kenneth L. Shepard. Digital circuit design challenges and opportunities in the era of nanoscale cmos. *Proceedings of the IEEE*, 96(2):343–365, 2008.

[14] M. Cannizzaro and L. Lavagno. Pid (partial inversion data): An m-of-n level-encoded transition signaling protocol for asynchronous global communication. In *Asynchronous Circuits and Systems (ASYNC), 2012 18th IEEE International Symposium on*, pages 134–141, 2012.

[15] Daniel M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Stanford University, October 1984.

[16] Fu-Chiung Cheng and Shuen-Long Ho. Efficient systematic error-correcting codes for semi-delay-insensitive data transmission. In *Computer Design, 2001. ICCD 2001. Proceedings. 2001 International Conference on*, pages 24 –29, 2001.

[17] R. Chipana, F.L. Kastensmidt, Jorge Tonfat, R. Reis, and M. Guthaus. Set susceptibility analysis in buffered tree clock distribution networks. In *Radiation and Its Effects on Components and Systems (RADECS), 2011 12th European Conference on*, pages 256–261, Sept 2011.

[18] Raul Chipana, E. Chielle, F.L. Kastensmidt, J. Tonfat, and R. Reis. Soft-error probability due to set in clock tree networks. In *VLSI (ISVLSI), 2012 IEEE Computer Society Annual Symposium on*, pages 338–343, Aug 2012.

[19] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, Alex Yakovlev, and Ne Ru England. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, 80:315–325, 1997.

[20] I. David, R. Ginosar, and Michael Yoeli. An efficient implementation of boolean functions as self-timed circuits. *Computers, IEEE Transactions on*, 41(1):2–11, 1992.

[21] Al Davis and Steven M. Nowick. An introduction to asynchronous circuit design. Technical report, THE ENCYCLOPEDIA OF COMPUTER SCIENCE AND TECHNOLOGY, 1997.

[22] Mark E. Dean, Ted E. Williams, and David L. Dill. Efficient self-timing with level-encoded 2-phase dual-rail (ledr). In *Proceedings of the 1991 University of California/Santa Cruz Conference on Advanced Research in VLSI*, pages 55–70, Cambridge, MA, USA, 1991. MIT Press.

[23] R. Dobkin, R. Ginosar, and C.P. Sotiriou. Data synchronization issues in gals socs. In *Asynchronous Circuits and Systems, 2004. Proceedings. 10th International Symposium on*, pages 170–179, April 2004.

[24] M. Donno, A. Ivaldi, L. Benini, and E. Macii. Clock-tree power optimization based on rtl clock-gating. In *Design Automation Conference, 2003. Proceedings*, pages 622–627, 2003.

[25] Elena Dubrova. *Fault-tolerant design*. Springer, 2013.

[26] M. Ebrahimi, S. G. Miremadi, H. Asadi, and M. Fazeli. Low-cost scan-chain-based technique to recover multiple errors in tmr systems. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, PP(99):1, 2012.

[27] M. Ebrahimi, S.G. Miremadi, and H. Asadi. Sctmr: A scan chain-based error recovery technique for tmr systems in safety-critical applications. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1 –4, march 2011.

[28] M. Ebrahimi, S.G. Miremadi, H. Asadi, and M. Fazeli. Low-cost scan-chain-based technique to recover multiple errors in tmr systems. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 21(8):1454–1468, 2013.

[29] Xin Fan, M. Krstić, and E. Grass. Analysis and optimization of pausible clocking based gals design. In *Computer Design, 2009. ICCD 2009. IEEE International Conference on*, pages 358–365, 2009.

[30] Xin Fan, M. Krstić, E. Grass, B. Sanders, and C. Heer. Exploring pausible clocking based gals design for 40-nm system integration. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 1118–1121, 2012.

[31] Xin Fan, M. Krstić, C. Wolf, and E. Grass. A gals fft processor with clock modulation for low-emi applications. In *Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on*, pages 273–278, 2010.

[32] Xin Fan, O. Schrape, M. Marinkovic, P. Dahnert, M. Krstić, and E. Grass. Gals design for spectral peak attenuation of switching current. In *Asynchronous Circuits and Systems (ASYNC), 2013 IEEE 19th International Symposium on*, pages 83–90, 2013.

[33] Jianxin Fang, S. Gupta, S.V. Kumar, S.K. Marella, V. Mishra, Pingqiang Zhou, and S.S. Sapatnekar. Circuit reliability: From physics to architectures: Embedded tutorial paper. In *Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference on*, pages 243–246, Nov 2012.

[34] K.M. Fant and S.A. Brandt. NULL Convention LogicTM: a complete and consistent logic for asynchronous digital circuit synthesis. pages 261 –273, August 1996.

[35] S.B. Furber, J.D. Garside, P. Riocreux, S. Temple, P. Day, Jianwei Liu, and N.C. Paver. Amulet2e: an asynchronous embedded controller. *Proceedings of the IEEE*, 87(2):243–256, Feb 1999.

[36] Rudy Garcia. Rethink fault models for submicron-ic test. *Test & Measurement World*, 21(12):35, 2001.

[37] M.R. Greenstreet. Implementing a stari chip. In *Computer Design: VLSI in Computers and Processors, 1995. ICCD '95. Proceedings., 1995 IEEE International Conference on*, pages 38–43, 1995.

[38] C.M. Grinstead and J.L. Snell. *Introduction to Probability*. 2nd ed. American Mathematical Society, 1997.

[39] C.S. Guenzer, E.A. Wolicki, and R.G. Allas. Single event upset of dynamic rams by neutrons and protons. *Nuclear Science, IEEE Transactions on*, 26(6):5048–5052, 1979.

[40] S. Hauck. Asynchronous design methodologies: an overview. *Proceedings of the IEEE*, 83(1):69–93, 1995.

[41] M.W. Heath, W.P. Burleson, and I.G. Harris. Synchro-tokens: a deterministic gals methodology for chip-level debug and test. *Computers, IEEE Transactions on*, 54(12):1532–1546, 2005.

[42] C. Huang. *Robust Computing with Nano-scale Devices: Progresses and Challenges*. Lecture notes in electrical engineering. Springer, 2010.

[43] Wei-Je Huang and E.J. McCluskey. Column-based precompiled configuration techniques for fpga. In *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*, pages 137–146, 2001.

[44] ITRS. International technology roadmap for semiconductors, 2005.

[45] ITRS. International technology roadmap for semiconductors, 2011.

[46] Wonjin Jang and A.J. Martin. Seu-tolerant qdi circuits [quasi delay-insensitive asynchronous circuits]. In *Asynchronous Circuits and Systems, 2005. ASYNC 2005. Proceedings. 11th IEEE International Symposium on*, pages 156 – 165, march 2005.

[47] JEDEC Standard JESD89A. *Measurement and Reporting of Alpha Particle and Terrestrial Cosmic Ray-Induced Soft Errors in Semiconductor Devices*, October 2006.

[48] Xin Jia and R. Vemuri. The gapla: a globally asynchronous locally synchronous fpga architecture. In *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, pages 291–292, 2005.

[49] Xin Jia and R. Vemuri. A novel asynchronous fpga architecture design and its performance evaluation. In *Field Programmable Logic and Applications, 2005. International Conference on*, pages 287 – 292, aug. 2005.

[50] T. Karnik, P. Hazucha, and J. Patel. Characterization of soft errors caused by single event upsets in cmos processes. *Dependable and Secure Computing, IEEE Transactions on*, 1(2):128–143, April 2004.

[51] Fernanda Lima Kastensmidt, Luigi Carro, and Ricardo Reis. *Fault-Tolerance Techniques for SRAM-Based FPGAs (Frontiers in Electronic Testing)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[52] David J. Kinniment. *Synchronization and Arbitration in Digital Systems*. Wiley Publishing, 2008.

[53] D.G. Kleinbaum and M. Klein. *Survival Analysis: A Self-Learning Text*. Statistics for Biology and Health Series. Springer Science+Business Media, Inc., 2005.

[54] A. Kondratyev and K. Lwin. Design of asynchronous circuits using synchronous cad tools. *Design Test of Computers, IEEE*, 19(4):107–117, 2002.

[55] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer New York Dordrecht Heidelberg London, 2st edition, 2011.

[56] Israel Koren and C. Mani Krishna. *Fault-Tolerant Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[57] J. Lach, W.H. Mangione-Smith, and M. Potkonjak. Enhanced fpga reliability through efficient run-time fault reconfiguration. *Reliability, IEEE Transactions on*, 49(3):296–304, 2000.

[58] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[59] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1:134–152, 1997.

[60] J. Lechner. Designing robust gals circuits with triple modular redundancy. In *Dependable Computing Conference (EDCC), 2012 Ninth European*, pages 227–236, 2012.

[61] J. Lechner and M. Lampacher. Protecting pipelined asynchronous communication channels against single event upsets. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pages 480–481, 2012.

[62] J. Lechner, M. Lampacher, and T. Polzer. A robust asynchronous interfacing scheme with four-phase dual-rail coding. In *Application of Concurrency to System Design (ACSD), 2012 12th International Conference on*, pages 122–131, 2012.

[63] Jakob Lechner and Robert Najvirt. A generic architecture for robust asynchronous communication links. In JoséL. Ayala, Delong Shang, and Alex Yakovlev, editors, *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, volume 7606 of *Lecture Notes in Computer Science*, pages 121–130. Springer Berlin Heidelberg, 2013.

[64] Jakob Lechner and Varadan Savulimedu Veeravalli. Modular redundancy in a gals system using asynchronous recovery links. In *Asynchronous Circuits and Systems (ASYNC), 2013 IEEE 19th International Symposium on*, pages 23–30, 2013.

[65] Walter M. *The SCARTS Hardware/Software Interface*. 2nd ed. OSADL Academic Works, 2011.

[66] A. J. Martin. Formal development programs and proofs. chapter Formal Program Transformations for VLSI Circuit Synthesis, pages 59–80. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[67] A.J. Martin, A. Lines, R. Manohar, M. Nystrom, P. Penzes, R. Southworth, U. Cummings, and Tak Kwan Lee. The design of an asynchronous mips r3000 microprocessor. In *Advanced Research in VLSI, 1997. Proceedings., Seventeenth Conference on*, pages 164–181, 1997.

[68] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In *Proceedings of the Sixth MIT Conference on Advanced Research in VLSI*, AUSCRYPT '90, pages 263–278, Cambridge, MA, USA, 1990. MIT Press.

[69] Timothy C. May and Murray H. Woods. A new physical mechanism for soft errors in dynamic memories. In *Reliability Physics Symposium, 1978. 16th Annual*, pages 33–40, 1978.

[70] P.B. McGee, M.Y. Agyekum, M.A. Mohamed, and S.M. Nowick. A level-encoded transition signaling protocol for high-throughput asynchronous global communication. In *Asynchronous Circuits and Systems, 2008. ASYNC '08. 14th IEEE International Symposium on*, pages 116–127, 2008.

[71] J. Mekie, S. Chakraborty, and D.K. Sharma. Evaluation of pausible clocking for interfacing high speed ip cores in gals framework. In *VLSI Design, 2004. Proceedings. 17th International Conference on*, pages 559–564, 2004.

[72] D.G. Messerschmitt. Synchronization in digital system design. *Selected Areas in Communications, IEEE Journal on*, 8(8):1404–1419, 1990.

[73] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and Kee Sup Kim. Robust system design with built-in soft-error resilience. *Computer*, 38(2):43–52, 2005.

[74] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, April 1965.

[75] S. Moore, G. Taylor, R. Mullins, and P. Robinson. Point to point gals interconnect. In *Asynchronous Circuits and Systems, 2002. Proceedings. Eighth International Symposium on*, pages 69–75, 2002.

[76] J. Muttersbach, T. Villiger, and Wolfgang Fichtner. Practical design of globally-asynchronous locally-synchronous systems. In *Advanced Research in Asynchronous Circuits and Systems, 2000. (ASYNC 2000) Proceedings. Sixth International Symposium on*, pages 52–59, 2000.

[77] Jens Muttersbach. *Globally-Asynchronous Locally-Synchronous Architectures for VLSI Systems*. PhD thesis, ETH, Zürich, 2001.

[78] V.P. Nelson. Fault-tolerant computing: fundamental concepts. *Computer*, 23(7):19–25, 1990.

[79] M. Nicolaidis. Time redundancy based soft-error tolerance to rescue nanometer technologies. In *VLSI Test Symposium, 1999. Proceedings. 17th IEEE*, pages 86–94, 1999.

[80] M. Nicolaidis, editor. *Soft Errors in Modern Electronic Systems*. Frontiers in Electronic Testing. Springer Science+Business Media, LLC, 2011.

[81] Sampo Niskanen and Patric R.J. Östergård. Cliquer user's guide. Research Report T48, Helsinki University of Technology, 2003.

[82] Simon Ogg, Bashir Al-Hashimi, and Alex Yakovlev. Asynchronous transient resilient links for noc. In *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, CODES+ISSS '08, pages 209–214, New York, NY, USA, 2008. ACM.

[83] N.C. Paver, P. Day, C. Farnsworth, D. L. Jackson, W. A. Lien, and J. Liu. A low-power, low noise, configurable self-timed dsp. In *Advanced Research in Asynchronous Circuits and Systems, 1998. Proceedings. 1998 Fourth International Symposium on*, pages 32–42, Mar 1998.

[84] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, April 1980.

[85] Stefan Poledna. *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[86] Stefan Poledna, Alan Burns, Andy Wellings, and Peter Barrett. Replica determinism and flexible scheduling in hard real-time dependable systems. *IEEE Transactions on Computers*, 49:100–111, 2000.

[87] Alain Poli and Llorenç Huguet. *Error Correcting Codes: Theory and Applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[88] Thomas Polzer, Andreas Steininger, and Jakob Lechner. Muller c-element metastability containment. In JoséL. Ayala, Delong Shang, and Alex Yakovlev, editors, *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, volume 7606 of *Lecture Notes in Computer Science*, pages 103–112. Springer Berlin Heidelberg, 2013.

[89] J. Pontes, N. Calazans, and P. Vivet. Adding temporal redundancy to delay insensitive codes to mitigate single event effects. In *Asynchronous Circuits and Systems (ASYNC), 2012 18th IEEE International Symposium on*, pages 142–149, 2012.

[90] J. Pontes, N. Calazans, and P. Vivet. Parity check for m-of-n delay insensitive codes. In *On-Line Testing Symposium (IOLTS), 2013 IEEE 19th International*, pages 157–162, 2013.

[91] L.L. Pullum. *Software Fault Tolerance Techniques and Implementation*. Artech House computing library. Artech House, 2001.

[92] Marvin Rausand and Arnljot Høyland. *System Reliability Theory: Models, Statistical Methods, and Applications, Second Edition*. Wiley-Interscience, 2 edition, Dec 2004.

[93] M. Riordan and Lillian Hoddeson. Crystal fire: the invention, development and impact of the transistor. *Solid-State Circuits Society Newsletter, IEEE*, 12(2):24–29, 2007.

[94] J.A. Rivers, M.S. Gupta, J. Shin, P.N. Kudva, and P. Bose. Error tolerance in server class processors. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(7):945–959, 2011.

[95] Andrew Royal and Peter Y.K. Cheung. Globally asynchronous locally synchronous fpga architectures. In Peter Cheung and GeorgeA. Constantinides, editors, *Field Programmable Logic and Application*, volume 2778 of *Lecture Notes in Computer Science*, pages 355–364. Springer Berlin Heidelberg, 2003.

[96] L.F.G. Sarmenta, G.A. Pratt, and S.A. Ward. Rational clocking [digital systems design]. In *Computer Design: VLSI in Computers and Processors, 1995. ICCD '95. Proceedings., 1995 IEEE International Conference on*, pages 271–278, 1995.

[97] N. Seifert, P. Shipley, M.D. Pant, V. Ambrose, and B. Gill. Radiation-induced clock jitter and race. In *Reliability Physics Symposium, 2005. Proceedings. 43rd Annual. 2005 IEEE International*, pages 215–222, April 2005.

[98] Charles L. Seitz. System timing. In Carver A. Mead and Lynn A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.

[99] Yebin Shi, S.B. Furber, J. Garside, and L.A. Plana. Fault tolerant delay insensitive inter-chip communication. In *Asynchronous Circuits and Systems, 2009. ASYNC '09. 15th IEEE Symposium on*, pages 77 –84, may 2009.

[100] A. Smirnov, A. Taubin, Ming Su, and M. Karpovsky. An automated fine-grain pipelining using domino style asynchronous library. In *Application of Concurrency to System Design, 2005. ACSD 2005. Fifth International Conference on*, pages 68–76, 2005.

[101] Christine Solnon. Alldifferent-based filtering for subgraph isomorphism. *Artif. Intell.*, 174(12-13):850–864, August 2010.

[102] Jens Sparsø and Steve Furber, editors. *Principles of Asynchronous Circuit Design: A Systems Perspective*. Kluwer Academic Publishers, 2001.

[103] Jens Sparsø and Jørgen Staunstrup. Delay-insensitive multi-ring structures. *Integration, the VLSI journal*, 15(3):313–340, 1993.

[104] M. Stanisavljevic, A. Schmid, and Y. Leblebici. *Reliability of Nanoscale Circuits and Systems: Methodologies and Circuit Architectures*. Springer, 2011.

[105] I. E. Sutherland. Micropipelines. *Commun. ACM*, 32(6):720–738, June 1989.

[106] P. Sweeney. *Error control coding: an introduction*. Prentice Hall, 1991.

[107] N.R. Tague. *The Quality Toolbox*. Asq Press, 2005.

[108] Alexander Taubin, Jordi Cortadella, Luciano Lavagno, Alex Kondratyev, and Ad Peeters. Design automation of real-life asynchronous devices and systems. *Found. Trends Electron. Des. Autom.*, 2(1):1–133, January 2007.

[109] P. Teehan, M. Greenstreet, and G. Lemieux. A survey and taxonomy of gals design styles. *Design Test of Computers, IEEE*, 24(5):418–428, 2007.

[110] J. Teifel and R. Manohar. An asynchronous dataflow fpga architecture. *Computers, IEEE Transactions on*, 53(11):1376 –1392, nov. 2004.

[111] C. Temple. Avoiding the babbling-idiot failure in a time-triggered communication system. In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 218–227, 1998.

[112] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, January 1976.

[113] Kees van Berkel. Beware the isochronic fork. *Integr. VLSI J.*, 13:103–128, June 1992.

[114] V.S. Veeravalli, T. Polzer, A. Steininger, and U. Schmid. Architecture and design analysis of a digital single-event transient/upset measurement chip. In *Digital System Design (DSD), 2012 15th Euromicro Conference on*, pages 8–17, 2012.

[115] Tom Verhoeff. Delay-insensitive codes - an overview. *Distributed Computing*, 3(1):1–8, 1988.

[116] A.K. Verma, S. Ajit, and D.R. Karanki. *Reliability and Safety Engineering*. Springer series in reliability engineering. Springer London, 2010.

[117] John von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata Studies*, pages 43–98, 1956.

[118] J.F. Wakerly. Transient failures in triple modular redundancy systems with sequential modules. *IEEE Transactions on Computers*, 24(5):570–573, 1975.

[119] Fan Wang and V.D. Agrawal. Single event upset: An embedded tutorial. In *VLSI Design, 2008. VLSID 2008. 21st International Conference on*, pages 429–434, Jan 2008.

[120] Ted E. Williams. *Self-Timed Rings and their Application to Division*. PhD thesis, Stanford University, June 1991.

[121] Martin Wirnshofer. *Variation-Aware Adaptive Voltage Scaling for Digital CMOS Circuits*. Springer Publishing Company, Incorporated, 2013.

[122] M.K. Yadav, M.R. Casu, and M. Zamboni. Dvfs based on voltage dithering and clock scheduling for gals systems. In *Asynchronous Circuits and Systems (ASYNC), 2012 18th IEEE International Symposium on*, pages 118–125, May 2012.

[123] Shu-Yi Yu and E.J. McCluskey. On-line testing and recovery in tmr systems for real-time applications. In *Test Conference, 2001. Proceedings. International*, pages 240 –249, 2001.

[124] Zhiyi Yu, Zewen Shi, and Xiaoyang Zeng. Fault tolerant computing for stream dsp applications using gals multi-core processors. In *Circuits and Systems (ISCAS), 2011 IEEE International Symposium on*, pages 2305–2308, 2011.

[125] K.Y. Yun and R.P. Donohue. Pausible clocking: a first step toward heterogeneous systems. In *Computer Design: VLSI in Computers and Processors, 1996. ICCD '96. Proceedings., 1996 IEEE International Conference on*, pages 118 –123, oct 1996.

[126] Guangda Zhang, Wei Song, J.D. Garside, J. Navaridas, and Zhiying Wang. Transient fault tolerant qdi interconnects using redundant check code. In *Digital System Design (DSD), 2013 Euromicro Conference on*, pages 3–10, 2013.

[127] Haissam Ziade, Rafic A. Ayoubi, and Raoul Velazco. A survey on fault injection techniques. *Int. Arab J. Inf. Technol.*, 1(2):171–186, 2004.

[128] L. Łukasiak and A. Jakubowski. History of semiconductors. *Journal of Telecommunications and Information Technology*, nr 1:3–9, 2010.

Curriculum Vitae
Jakob Lechner

Personal Details

Date of birth: June 8th, 1984

Citizenship: Austria

Education

Vienna University of Technology — Vienna, Austria
Computer Engineering, PhD — *Mar. 2009 – June 2014*

Vienna University of Technology — Vienna, Austria
Computer Engineering, MSc — *Mar. 2006 – Dec. 2008*

Vienna University of Technology — Vienna, Austria
Software & Information Engineering, BSc — *Oct. 2005 – Apr. 2007*

Vienna University of Technology — Vienna, Austria
Computer Engineering, BSc — *Oct. 2002 – Feb. 2006*

BG/BRG Oberschützen — Oberschützen, Austria
Secondary school (Gymnasium/AHS) — *Sept. 1994 – June 2002*

Experience

RUAG Space GmbH — Vienna, Austria
FPGA/ASIC Design Engineer — *Jan. 2014 to date*

Newcastle University — Newcastle upon Tyne, United Kingdom
Visiting Research Associate — *Mar. 2013 – Sept. 2013*

Vienna University of Technology — Vienna, Austria
Research Assistant — *Feb. 2009 – Feb. 2013*

appl.strudl Software GmbH (Fabasoft Group) — Vienna, Austria
Linux Software Engineer — *Mar. 2008 – Jan. 2009*

Fabalabs Software GmbH (Fabasoft Group) — Vienna/Linz, Austria
Linux Software Engineer — *Aug. 2006 – Feb. 2008*

Vienna University of Technology — Vienna, Austria
Tutor — *2004 – 2006*

Computersysteme Signale GmbH — Pinkafeld, Austria
Windows Software Engineer — *July 2001 – May 2005*

Grants & Fellowships

Marietta Blau-Stipendium: OeAD – GmbH, funded by the Austrian Federal Ministry of Science and Research (BMWF), Vienna, 2013

Own Publications

[1] Marcus Jeitler and Jakob Lechner. Comparing the robustness of synchronous and asynchronous circuits by fault injection. *5th Doctorial Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'09)*, Nov. 2009.

[2] Marcus Jeitler and Jakob Lechner. Speeding up fault injection for asynchronous logic by fpga-based emulation. *International Conference on Reconfigurable Computing and FPGAs (RECONFIG'09)*, pages 65–70, 2009.

[3] Marcus Jeitler and Jakob Lechner. Towards comparing the robustness of synchronous and asynchronous circuits by fault injection. In *Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'09)*, volume 13 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[4] Marcus Jeitler, Jakob Lechner, and Andreas Steininger. Enhancing pipelined processor architectures with fast autonomous recovery of transient faults. In *Design and Diagnostics of Electronic Circuits and Systems (DDECS), 2010 IEEE 13th International Symposium on*, pages 233–236, April 2010.

[5] Marcus Jeitler and Jakob Lechner. Low latency recovery from transient faults for pipelined processor architectures. In *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, pages 219–225, Sept. 2010.

[6] Jakob Lechner. Designing robust gals circuits with triple modular redundancy. In *Dependable Computing Conference (EDCC), 2012 Ninth European*, pages 227 –236, may 2012.

[7] Jakob Lechner, Martin Lampacher, and Thomas Polzer. A robust asynchronous interfacing scheme with four-phase dual-rail coding. In *Application of Concurrency to System Design (ACSD), 2012 12th International Conference on*, pages 122 –131, june 2012.

[8] Jakob Lechner and Martin Lampacher. Protecting pipelined asynchronous communication channels against single event upsets. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pages 480–481, Oct. 2012.

[9] Jakob Lechner and Robert Najvirt. A generic architecture for robust asynchronous communication links. In José L. Ayala, Delong Shang, and Alex Yakovlev, editors, *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, volume 7606 of *Lecture Notes in Computer Science*, pages 121–130. Springer Berlin Heidelberg, 2013.

[10] Jakob Lechner and Varadan Savulimedu Veeravalli. Modular redundancy in a gals system using asynchronous recovery links. In *Asynchronous Circuits and Systems (ASYNC), 2013 19th IEEE International Symposium on*, 2013.

[11] Syed Rameez Naqvi, Andreas Steininger, and Jakob Lechner. An set tolerant tree arbiter cell. In *Asynchronous Circuits and Systems (ASYNC), 2013 19th IEEE International Symposium on*, 2013.

[12] Syed Rameez Naqvi, Jakob Lechner, and Andreas Steininger. Protection of muller-pipelines from transient faults. In *Quality Electronic Design (ISQED), 2014 15th International Symposium on*, pages 123–131, March 2014.