FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Protecting 4-Phase Delay-Insensitive Communication Against Transient Faults

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Florian Ferdinand Huemer BSc.
Matrikelnummer 0828465

an der Fakultät für Informatik
der Technischen Universität Wien

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

Wien, 19. Jänner 2017

Florian Ferdinand Huemer	Andreas Steininger

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Protecting 4-Phase Delay-Insensitive Communication Against Transient Faults

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Florian Ferdinand Huemer BSc.
Registration Number 0828465

to the Faculty of Informatics
at the Vienna University of Technology

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

Vienna, 19th January, 2017

Florian Ferdinand Huemer

Andreas Steininger

Erklärung zur Verfassung der Arbeit

Florian Ferdinand Huemer BSc.
Spengergasse 28/21, 1050 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.

Wien, 19. Jänner 2017

Florian Ferdinand Huemer

Acknowledgments

First of all, I want to thank my parents for their great support throughout my studies. I would further like to thank my advisor for the opportunity to work on this very interesting topic. His excellent guidance, the continuous discussions and meaningful remarks were essential in the process of writing this thesis. Special thanks go to Jakob Lechner for his valuable suggestions and feedback.

Kurzfassung

Mit ihrer Robustheit gegen Signallaufzeitschwankungen bieten asynchrone delay-insensitive (DI) Übertragungsstrecken vorteilhafte Eigenschaften im Vergleich zu synchronen Lösungen. Dabei ist allerdings zu beachten, dass dafür spezielle DI Codes notwendig sind. Diese Codes sind aber in der Regel sehr anfällig für transiente Fehler, die während einer Übertragung auftreten können, da bei vielen dieser Codes bereits ein einzelner Fehler (im schlimmsten Fall) eine völlige Änderung des Nachrichteninhaltes zur Folge haben kann. Wenn dem Empfänger einer solchen Nachricht keine zusätzlichen Informationen zur Verfügung gestellt werden, hat dieser keine Möglichkeit, diese Übertragungsfehler zu erkennen, was natürlich schwerwiegende Konsequenzen für ein System an sich sowie dessen Umgebung haben kann.

In dieser Arbeit werden daher Möglichkeiten zur Absicherung von DI Kommunikation untersucht. Darüber hinaus wird ein neuartiges, zweistufiges Kodierungsverfahren vorgestellt, das auf der Kombination von fehlererkennenden und DI Codes basiert. Diese Lösung nützt dabei die inhärente Fehlerwiderstandsfähigkeit von DI Codes aus und erreicht damit eine gute Kodierungseffizienz bei gleichzeitig niedrigem Implementierungsaufwand. Um die Fehleranfälligkeit der Codes zu analysieren und um gültige Lösungen zu ermitteln, kommen Methoden der Graphentheorie zum Einsatz. Im Vergleich zu existierenden Lösungen wird hier sehr genau darauf geachtet, keine Annahmen über das Signallaufzeitverhalten zu treffen. Die vorgeschlagene Lösung ist sehr generisch und kann prinzipiell mit jedem vierphasigen DI Code verwendet werden.

Mittels einer repräsentativen Auswahl von m-aus-n Codes wird gezeigt, wie das Kodierungsverfahren angewendet wird und welche Kodierungseffizienz dabei zu erwarten ist. Zusätzlich wird eine Metrik eingeführt, die es erlaubt geeignete Codes für gegebene Anforderungen an fehlertolerante Übertragungsstrecken zu identifizieren.

Weiters stellen wir eine Reihe von Sender- und Empfängerschaltungen vor, die verwendet werden können, um das neue Kodierungsverfahren zu implementieren. Zwei detaillierte Implementierungsbeispiele auf Gatterebene für Vertreter der m-aus-n Codeklasse demonstrieren dabei die Machbarkeit des Lösungsansatzes und geben einen Einblick in die zu erwartenden Implementierungskosten.

Abstract

Compared to synchronous approaches, asynchronous delay-insensitive (DI) communication links have very interesting and desirable properties with respect to their robustness against timing variations and delay assumptions required to implement them. However, special DI codes have to be used to encode the data being transmitted. These codes are usually prone to transient faults occurring during an ongoing transmission, since, in the worst case, even a single transient fault is sufficient to completely change the contents of a message. Unless further redundant information is provided, the receiver has no means to detect such an erroneous transmission. This can, of course, have severe consequences on a system and the environment depending on it.

In this thesis we therefore investigate existing approaches to secure DI communication against transient faults and propose a novel two-step data encoding scheme that combines DI and error detecting codes. Our solution exploits the inherent fault resilience of DI codes to achieve a low overhead and hence good coding efficiency. We use methods from graph theory to analyze this fault resilience and identify appropriate solutions. In contrast to existing approaches we carefully avoid the introduction of timing assumptions to mask faults. The proposed coding scheme is generic and can, in principle, be used with any 4-phase DI code. We give examples on how to apply it to selected representatives of the important class of m-of-n codes and analyze the resulting coding efficiency. Additionally, we provide a metric that allows to identify which codes are well suited for fault-tolerant communication.

We, furthermore, provide a range of transmitter and receiver circuit variants that implement the presented coding scheme. In particular, we give detailed gate-level implementation examples for two m-of-n codes, that demonstrate the feasibility of our approach and give some insight into the required implementation overhead.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Motivation and Problem Statement

Modern semiconductor technology allows the integration of an ever increasing number of transistors into a single integrated circuit. However, with these huge advances in technology, the traditional synchronous design approach to digital circuits faces new problems and challenges. Over the last decades the feature (i.e. transistor) size shrank from around 10 μm in the 1970s to today's 22 nm [1]. While smaller transistors can operate at a much higher speed (resulting in a lower gate delay), the metal wires connecting them, i.e. the interconnect, suffer from this miniaturization trend. Thinner wires have a higher resistance which results in greater switching delays. This especially affects long distance wires, like communication links connecting functional units of a chip. In the beginning of integrated circuit design, due to relatively slow transistors, the interconnect delay was negligible. However, the continuous miniaturization led to the situation that it now exceeds the gate delay and became the dominant source of delay in integrated circuits. While gate delays are known very early in the design process, the interconnect delay is difficult to estimate and remains unknown until the actual layout process.

Another problem are process variations during production, as a consequence of which different chips from the same product line have different delays. Delays are also affected by environmental conditions such as temperature and supply voltage. To accommodate for such variations, delay assumptions and safety margins have to be chosen pessimistically, which may unnecessarily slow down the device (e.g. lower clock frequency).

The fundamental concept that makes synchronous design so practical to work with, is the common time base, i.e. the clock, that is used to coordinate all operations in a circuit. Today the distribution of the clock signal to all parts of a design accounts for a considerable part of the overall power consumption of a chip [2].

One option to tackle these problems are asynchronous circuits utilizing delay-insensitive (DI) codes [3, 4, 5]. As the name suggests these circuits don't need a global clock source to drive their operation. Additionally, DI codes offer an inherent robustness against delay variations. This is because the receiver of a DI code word can detect the arrival of complete and valid data solely by

checking the received bit pattern for certain properties. Examples for such codes are m-of-n and Berger codes [4]. On the other hand, the use of DI codes comes at the cost of a greater overhead and increased design complexity. Thus, it makes sense to use DI codes to implement global chip or inter-chip communication and use the synchronous or asynchronous bundled data [5] design approach for the actual functional units of a design. In literature this concept is often referred to as GALS (Globally Asynchronous Locally Synchronous) [5].

Another challenge for integrated circuit design are safety-critical applications, like transportation systems (cars, planes, trains etc.), space flight or (nuclear) power plants. For those applications dependable systems are needed because one failure in such a system can have severe consequences on human lives or the environment. Many of these systems also have to operate under harsh environmental conditions, like high radiation levels or ambient temperature fluctuations. Integrated circuits can be affected by permanent and transient faults which can both cause a malfunction of the chip. Permanent faults result in physical damage to the circuit and can not be corrected (although they may be tolerated to some degree). Transient faults only affect the circuit over a limited period of time. However, it is possible that they manifest themselves in a storage element (soft-error), i.e. the state of the circuit, and cause unintended circuit behavior. Dependable systems must be able to cope with faults to a defined extent and still be able to guarantee safe operation. As discussed in [6], transient faults in today's designs are mainly caused by cosmic radiation. The trend to smaller transistors and lower supply voltages further increases the sensitivity to radiation which makes fault-tolerant design also an issue for non-safety-critical applications. While fault-tolerant coding has already been studied intensively, the combination of fault-tolerant and DI coding has not yet been addressed sufficiently.

For these reasons, this work will investigate methods to build and improve fault-tolerant delay-insensitive communication links, which are able to cope with transient faults.

1.2 Structure of the Thesis

First, Chapter 2 presents the theoretical foundation the following chapters build upon. Furthermore, a precise problem definition and a detailed analysis of current fault modeling techniques is provided. Chapter 3 covers related work that also aims at securing DI communication against faults. The main contributions of this thesis are provided in Chapters 4 and 5. While Chapter 4 introduces a novel fault-tolerant delay-insensitive coding scheme from a theoretical point of view, Chapter 5 presents circuits that can be used to actually implement the proposed coding scheme in hardware. Chapter 6 discusses the efficiency of our approach and shows two gate-level implementation examples. Finally, Chapter 7 concludes the thesis and gives a short outlook on what research directions can be explored further.

CHAPTER 2

Theory and Background

This chapter presents a summary of the theory and background information, required to understand the concepts and circuits discussed in the following chapters.

2.1 Coding Theory

In this section we provide a brief introduction into the field of coding theory with a special focus on linear block codes. There is a variety of good references on this subject. We have based this section on the books by Blahut [7] and Roth [8]. Coding theory plays an important role in modern digital communication and storage systems. Digital communication channels, i.e. the physical media that are used to transport information (metal wires, air), are always affected by some form of disturbances (noise, interference, crosstalk, etc.). Likewise, digital storage systems suffer from similar problems (e.g. scratches on an optical disk, change in charge in a flash cell). If data would be transmitted or stored without any encoding, errors would directly change the contents of a message, which can have severe consequences on the overall system. Proper codes allow the receiver to detect and even correct erroneous messages. However, these properties also come at a price. A code always adds a certain amount of redundancy to information, which obviously increases the amount of data that needs to be transmitted. The component that takes the (unencoded) input data, i.e. the information or *data word*, and applies the desired code to create a *code word* is referred to as encoder. A decoder is then used to restore the original data word and apply error detection or correction.

A block code always takes a block of data (i.e. the data word) and converts it to a code word. Every data word is assigned exactly one code word. This means that the encoder does not have an internal state. The encoding of a data word is independent of the history of previously encoded information. This is in contrast to convolutional codes, where the encoding and decoding processes are dependent on the internal state of the encoder and decoder, respectively. The field of convolutional codes gives rise to a more general coding model. However, since this work only uses (linear) block codes, we won't go into further detail on this subject. Linear block codes are

widely used and some well known codes such as Parity, Hamming, Reed-Muller, Reed-Solomon, etc. fall into this category. In the following Section 2.1.1 provides a basic introduction to coding theory in general while Section 2.1.2 explains the special properties of linear codes, relevant to this work.

2.1.1 Introduction

A code C is always defined over some alphabet F of size q. Since our discussion is focused on linear block codes, we restrict the alphabet to be a Galois or finite field, denoted by $GF(q)$ or $\mathbb{F}_q$ [1]. This implies that the size of the alphabet is always a prime power. The simplest finite field is $\mathbb{F}_2 = \{0,1\}$, which contains exactly two elements and gives rise to the well known class of binary codes. We will refer to the elements of F as the symbols of the code. The code C is a (nonempty) subset of the vector space F^n, and its elements are referred to as code words. The size of C is denoted by M, whereas n is referred to as code or block length. In the following we will discuss some fundamental definitions important to coding theory.

Definition 1 (Hamming distance)**.** *The Hamming distance d between two code words* $x = (x_1,..,x_n)$ *and* $y = (y_1,...,y_n)$ *of some code C is the number of positions where* $x_i \neq y_i$*, i.e. the number of positions where x and y differ.*

The Hamming distance is a metric for how much two code words differ. It is always greater or equal to zero, symmetric ($d(x,y) = d(y,x)$) and satisfies the triangle inequality ($d(x,y) \leq (d(x,z) + d(z,y))$.

Definition 2 (Hamming weight)**.** *The Hamming weight of a code word* $x = (x_1,...,x_n) \in C$ *is the number of positions where x is not zero.*

In this context zero means, equal to the zero element of the algebraic structure which makes up the alphabet F of the code. Note that the Hamming distance and weight are related by Equation 2.1.

$$d(x,y) = w(y-x) \tag{2.1}$$

Consider, for example, the code words $x = (0,1,2)$ and $y = (2,1,1)$ of some code $C \subset \mathbb{F}_3^3$ over $\mathbb{F}_3 = \{0,1,2\}$. Their respective Hamming weights are given by $w(x) = 2$ and $w(y) = 3$, while for their Hamming distance we have $d(x,y) = 2$. By calculating the Hamming weight of $y - x = (2,0,2)$, the same value for the Hamming distance of x and y is obtained.

Definition 3 (Minimum distance)**.** *The minimum distance of a code C (*$M > 1$*) is defined as the smallest Hamming distance between any two code word pairs of C.*

$$d_{min}(C) = min_{x,y \in C, x \neq y} d(x,y)$$

[1] Henceforth, we will only use the latter notation

The minimum distance determines a code's error detecting and correcting capabilities. Let f denote the number of errors affecting a code word x of some code C. A single error causes the symbol in one position of x to change into some other (arbitrary) symbol of F. For example, let $C \subset \mathbb{F}_5^3$ be a code over $\mathbb{F}_5 = \{0,1,2,3,4\}$, then the transformation $(0,1,2) \rightarrow (4,1,2)$ would constitute a single error ($f = 1$), while for $(0,1,2) \rightarrow (4,1,3)$ a double error ($f = 2$) would be required. For every code C there exists a decoder that is able to correct up to $\lfloor (d_{\min} - 1)/2 \rfloor$ (symbol) errors. If the decoder is only concerned with detecting errors rather than correcting them, this is possible for up to $d_{\min} - 1$ errors. Intuitively this is easy to understand because we know that there exist (at least) two code words x and y in C with $d(x,y) = d_{\min}$. Hence, $d_{\min}$ symbol flips are necessary to transform x to y. If x is affected by less than $d_{\min}$ errors, no (valid) code word of C is reached and the error must be visible to the decoder. Detailed proofs for these propositions can be found in [8].

2.1.2 Linear Block Codes

In the following we will focus our discussion on *linear* block codes. A block code C over a finite field $\mathbb{F}_q$ is linear if C is a linear subspace of the vector space $\mathbb{F}_q^n$. Since C is a subspace of $\mathbb{F}_q^n$, there exists a set of vectors $g_1, ..., g_k \in C$, called the basis of C. Every code word in C can be written as a linear combination of these base vectors. Formally we can thus write

$$x \in C \Leftrightarrow x = \sum_{i=0}^{k} a_i * g_i, \text{ where } a_0, ..., a_k \in \mathbb{F}_q \,. \tag{2.2}$$

The parameter k is called the dimension of C. Since every code word is generated by a unique linear combination of the base vectors (unique values for the scalars $a_0, ..., a_k$), the code's size M is given by q^k. The parameters k and n are used to calculate the code rate R. This ratio basically measures the amount of actual information contained in a code word with respect to overall length of the code. Hence, it must always be a value smaller than or equal to one.

$$R = \frac{k}{n} \tag{2.3}$$

The value $n - k$ is also referred to as the redundancy of the code.

The base vectors of C can be combined into the so called generator matrix G (Equation 2.4), which comprises k rows and n columns.

$$G = \begin{pmatrix} g_1 \\ \vdots \\ g_k \end{pmatrix} \tag{2.4}$$

Using this matrix the encoding function $f : \mathbb{F}_q^k \mapsto \mathbb{F}_q^n$ can simply be defined by a matrix multiplication. As shown in Equation 2.5 the k-element row vector, representing the unencoded data word, is multiplied by G to generate the corresponding n-element code word (row vector).

$$f(x) = x * G \tag{2.5}$$

A code is completely specified by its generator matrix. If the generator matrix exhibits the form $G = (I_k|A)$, where I_k denotes the $k \times k$ identity matrix then it is called *systematic*. Code words of systematic codes contain their corresponding (unencoded) data words as their first k elements, which can simplify the decoding process. This means that every element in such a code word is either a data or a check symbol, hence these codes are also called separable. Note that the check symbols are generated solely by the $k \times (n-k)$ matrix A. If a linear code is not systematic it is always possible to transform it into an equivalent systematic code, i.e. a (different) code with the same properties.

Another very important matrix, that like the generator matrix, also completely defines a linear code is the parity-check matrix H. As shown in Equation 2.6 multiplying H with a transposed code word c^T yields the all zero vector iff c is a code word of C.

$$c \in C \Leftrightarrow H * c^T = 0 \tag{2.6}$$

This matrix can hence be used to perform error detection. A decoder can simply multiply the received code word with the parity-check matrix and check whether the result is zero. For systematic codes, the $(n-k) \times n$ parity-check matrix can immediately be calculated from the generator matrix G.

$$H = (-A^T|I_{n-k}), \text{ if } G = (I_k|A) \tag{2.7}$$

A commonly used notation for linear block codes over $\mathbb{F}_q$ with length n, dimension k and minimal distance d is $[n,k,d]_q$. To illustrate the presented concepts, consider the the following examples. Equation 2.8 shows two examples for generator matrices for the $[5,4,2]_2$ parity and $[7,4,3]_2$ Hamming code.

$$G_P = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix} \quad G_H = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \tag{2.8}$$

Both codes are systematic and encode 4 bits of information. The Hamming code offers a minimal distance of three, allowing the correction of one error or the detection of up to two errors. The parity code, on the other hand, is only able to detect single bit errors (correction is not possible). Note that the parity code's generator matrix G_P basically works for arbitrary field sizes. The 0 and 1 entries in the matrix simply have to be regarded as the zero and one-element of the particular field. Equation 2.9 shows the corresponding check matrices.

$$H_P = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \end{pmatrix} \quad H_H = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix} \tag{2.9}$$

In the next section another code class will be introduced, the so called unordered codes. These codes have a special property that makes them applicable to asynchronous circuit design.

2.2 Asynchronous Circuits

Storage elements such as flip-flops or latches play a vital role in digital design. Without them it would not be possible to implement any state holding circuits such as state machines or pipelines. The fundamental problem that needs to be solved when constructing digital circuits is how to organize the transfer of data between those elements. To illustrate this problem, consider two storage elements (source and sink) connected back-to-back (i.e. the output of the first one is connected to the input of the second one). Now, for a data transfer to take place, there must be means to answer the following two questions.

- When is the source allowed to apply new data to its output?
- When is the sink allowed to consume the data at its input?

In [9] these two questions are referred to as issue and capture condition. Moreover, it is shown how different design styles approach and solve this problem.

The widely used synchronous design style uses a (global) clock signal that triggers both the issue and capture operation. This means that at the active clock edge every (enabled) storage elements consumes the data at its input (stores it internally) and applies the new data to its output. Because of this common time base, the synchronous design style offers a good abstraction and makes it relatively easy to design circuits. However, this abstraction is based on the assumption that the clock signal reaches all flip-flops of a design simultaneously. The low clock skew required to maintain this assumption is increasingly hard to maintain in modern semiconductor technology. This has led to the situation that the clock distribution network (i.e. the clock tree) of a chip now consumes a high portion (sometimes more than 25%) of the total power [10]. The maximal allowed clock frequency of a synchronous circuit is determined by the static timing analysis. A process that basically localizes the longest path (in terms of delay) between any two registers in a design, which is also referred to as the critical path. Obviously this value poses an upper bound for the maximal clock frequency a circuit can be operated with, because there must be enough time between two clock edges for all signals in a design to reach their destinations. However, to accommodate for PVT variations it is necessary to apply some safety margin to the delay of the critical path. If this value is chosen too pessimistic the circuit suffers a (maybe unnecessarily high) performance penalty. A margin that is too small may, on the other hand, have a negative impact on the yield of the chip. Note that another implication of the critical path is that a processing pipeline can only operate with the speed of the slowest stage.

Asynchronous circuits and design styles use local handshaking signals instead of a global trigger like the clock to coordinate the data transfer between storage elements. Here neighboring stages communicate locally to inform each other when the next data item is available and when the previous data item has been consumed. Because of this locality of control, asynchronous circuits don't suffer from many of the the problems inherent to the synchronous design style. This makes asynchronous design styles a worthwhile alternative in many application areas. Without the (global) clock net they are, for example, an interesting and promising approach for low power devices. The more flexible timing models allow for the design of very robust circuits with respect to PVT variations. It also makes sense to combine asynchronous and synchronous methodologies

in a single design to leverage the advantages of both worlds, which leads to the concept of GALS (Globally Asynchronous Locally Synchronous) circuits.

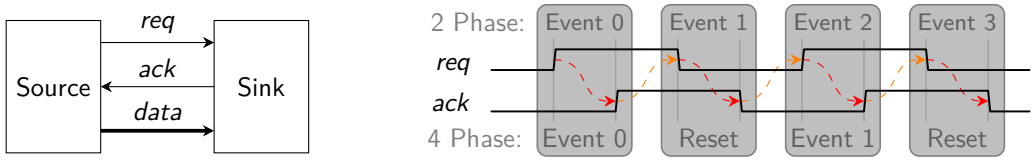
The remainder of this section is mainly based on [5], which provides an excellent introduction to the field of asynchronous circuits.

2.2.1 Handshaking Protocols

All asynchronous circuits use some form of handshaking protocol to transfer data between storage elements. Depending one which component (source or sink) initiates the transfer push and pull channels can be distinguished. In a push channel, such as the one shown in Figure 2.1a, the source uses the request (*req*) signal to notify the sink that new data is available. After the sink has consumed the data it uses the acknowledgment (*ack*) signal to in turn notify the source that it is ready for new data. In a pull channel the sink requests new data from the source, hence the signal directions are reversed. However, since this work only uses push channels, we won't go into further details on this topic.

Generally the handshaking protocols, i.e. the actual sequence of transitions on the *req* and *ack* signals as well as their meaning can be classified as 2-phase or 4-phase. Figure 2.1b illustrates the difference between these two possibilities. In 4-phase protocols only two of the altogether four transitions involved in one handshaking cycle (hence the name 4-phase), carry actual meaning. The other two are only used to reset the handshaking signals to their initial values. The rising edge of the *req* signal is usually used as indicator for when the data is valid. Hence, when the *req* signal reaches the sink the data must already be stable at the sink's input.

The 2-phase protocol, on the other hand, does not incorporate an additional reset phase for the handshaking signals, which can lead to a performance advantage. Here every pair of transitions constitutes a complete handshaking cycle.



(a) Asynchronous (push) channel

(b) Handshaking timing diagram

Figure 2.1: Asynchronous handshaking protocols

As we will see in Section 2.2.4.2 it is not always the case that the request mechanism is implemented by an explicit request signal. The request can also be implicitly encoded in the transmitted data and the receiver has to use a completion detector to decide whether the applied data can be consumed or if further transitions have to be awaited.

2.2.2 Delay Models

Asynchronous circuits can be classified on the basis of the delay assumptions imposed on them. The class of *delay-insensitive* (DI) circuits uses the weakest timing assumptions. The only

restriction on gate and wire delays is that they have to be positive and finite. For the example circuit, shown in Figure 2.2, this means that all gate delays Δ_A to Δ_C and wire delays Δ_1 to Δ_3 may be chosen completely arbitrary and the circuit would still work correctly. However, as shown by Martin [11], the class of circuits which can be constructed with this timing model is very small. This is because the only gates that can be used in DI circuits are inverters and C gates.

To overcome these limitations, isochronic forks [12] are introduced, leading to the class of *quasi-delay-insensitive* (QDI) circuits. With this extension to the DI timing model, arbitrary circuits can be constructed. While there are effectively no restrictions on the delays in DI circuits, QDI circuits require that (some selected) wire forks are isochronic, i.e. both signal paths after the fork must have the same delay. For the circuit in Figure 2.2 this means that $\Delta_2 = \Delta_3$. A detailed discussion of isochronic forks is given in [11].

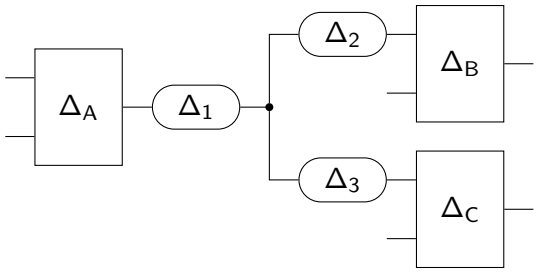


Figure 2.2: Asynchronous circuit model

For the class of *speed-independent* (SI) circuits, we again have arbitrary gate delays, however the wire delays are assumed to be zero ($\Delta_1 = \Delta_2 = \Delta_3 = 0$). Obviously, this timing model is the least realistic one, since interconnect delays play an important (and sometimes even dominating) role in modern chip technology. However, if the forks are regarded as isochronic, it is possible to combine the wire delays with the delay of the gate driving the wire (e.g. add Δ_1 and Δ_2 to Δ_A), which again yields a QDI circuit.

Finally we also want to mention the term *self-timed* (ST) circuits. The other models presented so far are precise (mathematical) concepts and apply to the gate-level of asynchronous circuits. ST circuits is a more general term and refers to circuits that need higher-level timing constraints or assumptions to work as intended. Such an assumption can for example state that the result of a certain (sub-) circuit must be available before another signal reaches its destination. These constraints are often enforced by the use of delay elements.

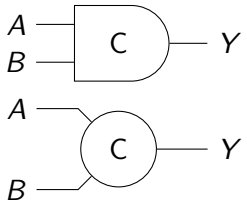
2.2.3 Asynchronous Circuit Primitives

Before it is possible to discuss the implementation level aspects of asynchronous circuits, some basic circuit primitives must be introduced.

2.2.3.1 Muller C-Element

A very basic gate found in nearly every asynchronous circuit is the Muller C-Element (henceforth simply referred to as C gate). Figure 2.3a shows the symbols that are commonly used for this

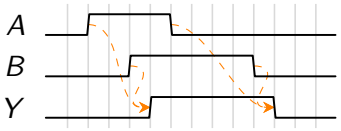
gate. In simple terms its operation can be described as an AND gate with hysteresis. As shown in the truth table in Figure 2.3b, in order to set the output of a C gate to one, both inputs must be set to one (similarly to an AND gate). However, to reset the output back to zero again, both inputs must be set to zero as well. If only one input changes its logical value the output of the gate does not change ("keep" entries in the truth table). Figure 2.3c further illustrates this behavior with a simple timing diagram. To implement this functionality the C gate obviously needs an internal storage to keep track of its current state.



(a) Circuit symbols

A	B	Y
0	0	0
0	1	keep
1	0	keep
1	1	1

(b) Truth table



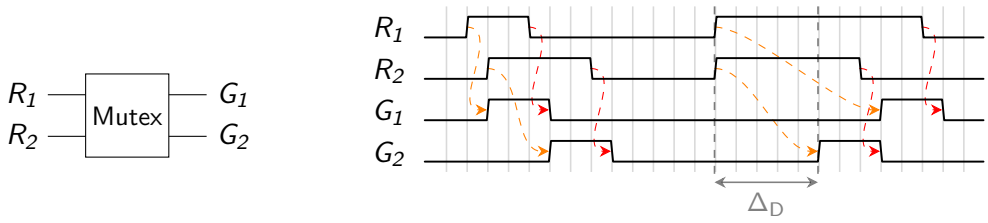
(c) Timing diagram

Figure 2.3: Muller C-Element

There are basically three popular ways to implement the C gate in CMOS logic. Since a detailed discussion on the implementation details would go beyond the scope of this work, we refer to [13], which presents the different CMOS circuits and compares their advantages and disadvantages.

2.2.3.2 Mutex

The other important component, we briefly want to present here, is the Mutex. As shown in Figure 2.4a it has two inputs (R_1 and R_2) and two outputs (G_1 and G_2). A rising edge (i.e. request) on one of the inputs is acknowledged (i.e. granted) by a rising edge on the associated output. The output is kept asserted until the input goes low again. The purpose of this component is to provide mutual exclusion for input requests. Thus at any given time at most one of its outputs can be asserted. This behavior is illustrated in the timing diagram in Figure 2.4b. In the left part of the figure there is a clear difference in the arrival times of the input requests. Hence R_1 will be granted first and R_2 has to wait until the first request is withdrawn. The right part shows what happens when both requests arrive virtually simultaneous. Similarly to the case in synchronous circuits when the clock and data edge arrive too close to each other at the inputs of a flip-flop, also the Mutex can go into a metastable state, where it is unable to decide which input request should be granted first. Like with flip-flops the result of this metastability can be a late transition on the output. However, unlike flip-flops, the outputs of a Mutex will not go into an undefined voltage range but rather both stay low until the metastability has been resolved. Note that this time period (marked Δ_D in the timing diagram) can be arbitrary long.



(a) Circuit symbol

(b) Timing diagram

Figure 2.4: Mutex

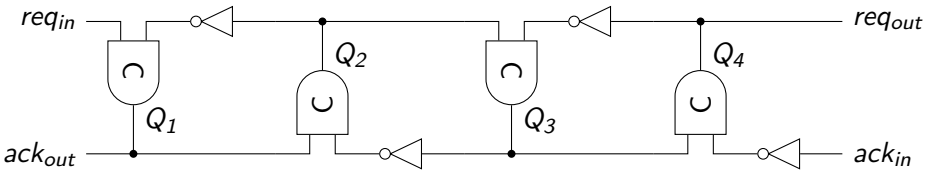
2.2.4 Pipeline Implementations

There are various ways to use the protocols discussed in Section 2.2.1 to implement pipelined circuits. However, before we can discuss the implementation details of the different pipeline styles, we first need to introduce a very basic asynchronous circuit.

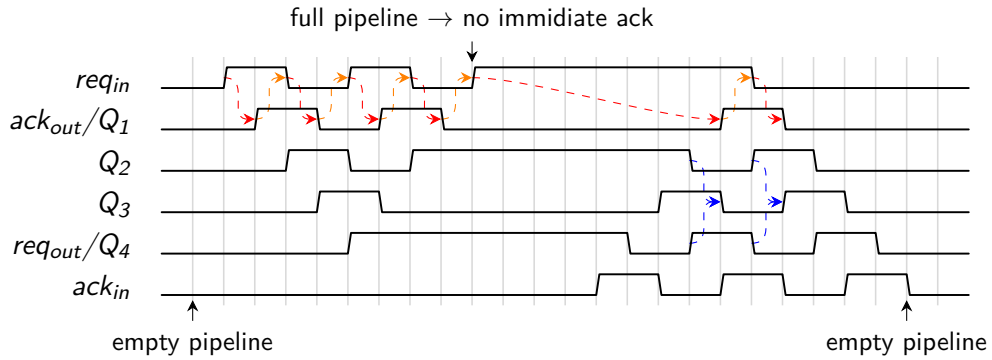
Figure 2.5a shows the Muller pipeline, as presented in [14]. Basically, the purpose of this circuit is to store and transport handshakes from its input (req_{in}, ack_{out}) to its output port (req_{out}, ack_{in}). To understand how it works consider the timing diagram in Figure 2.5b. For this diagram it is assumed that initially all C gates are set to zero and that the C gate and the inverter have a combined delay of one time unit. The circuit operates according to a very simple rule. Every C gate (i.e. stage) changes its output Q_i if Q_{i-1} and Q_{i+1} differ in value. Hence a C gate forwards a one from the preceding stage if the succeeding stage is zero (and vice versa for zeros). This behavior is highlighted for Q_3 by (blue) arrows in the timing diagram. As the pipeline is initially empty the first rising transition on req_{in} can immediately ripple through the whole circuit until it appears at the output req_{out}. The input handshakes, as produced by the environment of the circuit, are marked with arrows. The following transitions on req_{in} fill up the pipeline. A full Muller pipeline can be identified by alternating logical states in each C gate. Notice that the full pipeline does not provide an acknowledgment for the last rising input transition at req_{in}. The input transitions can be viewed as waves traveling through the circuit. In the second half of the diagram the environment toggles the ack_{out} input to "read out" the stored transitions.

The Muller pipeline by itself is not a very useful circuit. However, it is nevertheless very important as it is an often reoccurring structure in asynchronous designs and forms the (basic) control circuit for many pipelines. In the next section, we will see how the Muller pipeline is basically everything that is needed to control the data flow though an asynchronous (bundled data) pipeline.

Finally it is important to stress that this circuit is in principle completely agnostic to the handshaking protocol. As can be seen in the timing diagram it is only a matter of interpretation if the input transitions form a 2-phase or 4-phase handshake. Moreover, notice that since the Muller pipeline is only built from C gates and inverters it is one of the few really DI circuits.



(a) Muller pipeline circuit



(b) Example timing diagram

Figure 2.5: Muller pipeline

2.2.4.1 Bundled Data Pipelines

Figure 2.6 shows an example for an asynchronous 4-phase bundled data (BD) pipeline. Notice the control structure that generates the enable signals for the data latches. Except for the delay elements, it resembles the Muller pipeline from Figure 2.5. The purpose of the delay elements is to ensure that the request signals reach the next stage only after the associated data is stable and valid at the input of the latches. Since the data usually passes some combinational logic the delay element has to be tuned to the critical path of this circuit. This is why these circuits are also referred to as matched delay pipelines. Notice that in contrast to synchronous circuits a slow path between two storage elements only affects the delay (i.e. performance) of the associated stage and does not necessarily have an impact on the overall circuit. It is even possible to make the delay element data dependent (i.e. multiplex between different delays) to optimize the performance for different use cases.

Recall that a full Muller pipeline stores alternating values in its C gates. This means that if the presented pipeline becomes full, only every second latch contains data (the other latches are transparent). Note, however, that there is a similar situation with the master and slave latch of a flip-flop in synchronous pipelines.

There are a lot of different pipeline styles that can be found in literature. The original way to utilize the Muller pipeline as proposed by Sutherland in [14] was to use the 2-phase protocol in combination with special capture/pass registers. Another interesting approach is the MOUSETRAP pipeline proposed by Singh et. al [15], which uses XOR gates instead of C gates to implement the latch controllers. Another approach, aimed at increasing the degree of

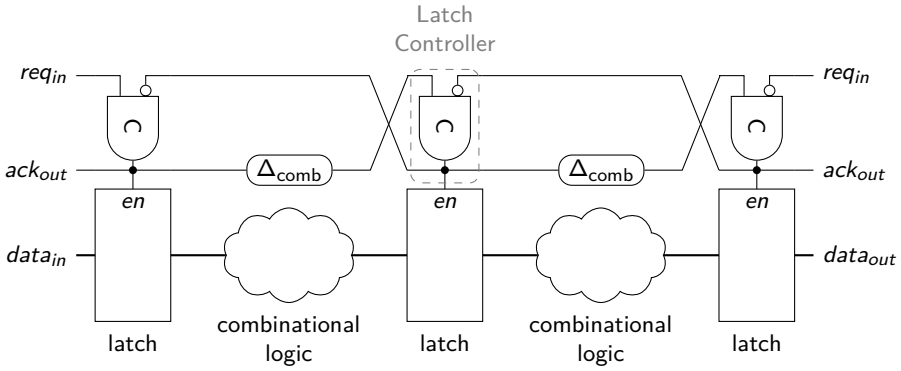


Figure 2.6: 4-Phase bundled data pipeline

decoupling between stages in 4-phase pipelines was proposed by Furber et. al. [16]. This paper uses Signal Transition Graphs (see Section 2.2.5) to model and implement more sophisticated latch controllers (compared to the single C gate in Figure 2.6), that ultimately also allow for a pipeline to fill up all its stages.

A data item that travels through a pipeline is often referred to as data token. An empty place in a pipeline is called a bubble. Data tokens travel in forward direction through a pipeline and replace bubbles, which hence move backwards. This means that a functional asynchronous pipeline always needs at least one bubble for data tokens to move to. A pipeline ring without bubbles is deadlocked.

2.2.4.2 4-Phase Delay-Insensitive Codes

In contrast to the bundled data circuits discussed in the previous section 4-phase (or return-to-zero) DI circuits don't use an explicit request signal. The request is rather implicitly encoded in the transmitted data and it is the responsibility of the receiver to decide when this data is complete (i.e. valid) and can thus be consumed. This process is referred to as completion detection and is only possible if the code that is used to encode the transmitted data has certain properties. Two successive code words (data phase) are always separated by a spacer (zero or null phase), which does not carry any information and is usually encoded by logical zeros on all bus wires. Figure 2.7 shows a timing diagram of the 4-phase protocol. Note that the arrival times of the transitions on

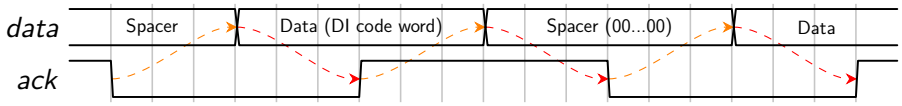


Figure 2.7: 4-Phase DI protocol

the individual bus wires, also referred to as rails, at the receiver are unknown and don't matter for the correct operation of the protocol. The fact that it is not necessary to impose any timing assumption (except for some isochronic forks) on QDI circuits makes them very robust against

(PVT) variations. The completion detector (CD) at the receiver is attached to all data rails and has exactly one output, which will be referred to as the *done* signal throughout this thesis. As soon as the bit pattern on the data rails forms a complete (i.e. valid) code word the *done* output is asserted. The signal is then kept asserted until the CD detects the spacer word.

The simplest DI code is the dual rail (DR) encoding, which is widely used for asynchronous circuit design. As the name suggests the DR code uses two rails to encode one bit of information. In the data phase of the 4-phase protocol exactly one of the two rails makes a transition to one. The CD for a DR bit can thus be implemented by a simple OR gate. The individual wires of a DR bit are also referred to as the true and false rail. Figure 2.8 shows a three stage one bit DR pipeline. Notice that this pipeline can also be viewed as two interlocked Muller pipelines.

Because of its simplicity and the systematic encoding it is also possible to perform logical operations on DR encoded data, although with a lot more overhead than with combinational logic in synchronous or BD designs. The simplest, but also most expensive (in terms of area overhead) is the Delay Insensitive Minterm Synthesis (DIMS) [17]. DIMS uses an array of C gates to exclusively map every possible (valid) input data word to a dedicated signal (one-hot code). In a second stage OR gates map this code to the desired output signals. The actual logical function depends only on these OR gates. Note that, despite the name, circuits obtained by applying this design style are only QDI. The forks in the input rails which are used to connect the inputs to the C gate stage must be isochronic. Further note that the DIMS design style is not restricted to function blocks operating on DR coded data and can in principle be applied to every 4-phase DI code. In particular it can also be used to construct CDs for arbitrary DI codes.

A more efficient design style was proposed by Theseus Logic [18], with the disadvantage of requiring special threshold gates. Another very efficient approach was proposed in [19], which has the big advantage of requiring only standard gates. Note that we don't use the term combinational logic for these types of functions blocks because the 4-phase protocol requires this logic to contain storage elements (i.e. C gates), to keep track of the current protocol phase.

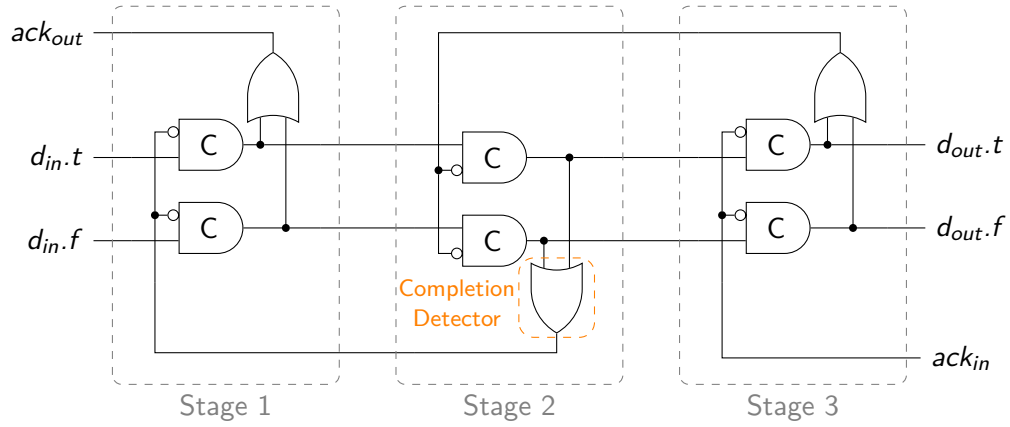


Figure 2.8: Single bit dual rail pipeline (3 stages)

Clearly the DR encoding is not the only DI code. The DR code is basically one representative of the class of constant-weight (m-of-n) codes, which are all DI. Other examples are Berger and

Zero-Sum codes, which will be presented in more detail in Section 3.2.

In the following we will discuss the mathematical properties that make a code DI. Verhoeff [4] shows that a DI code has to be unordered.

Definition 4 (Unordered Bit Vectors)**.** *Let $x = (x_0, ..., x_{n-1})$ and $y = (y_0, ..., y_{n-1})$ be two bit vectors of length n. Further let $S_x = \{i | x_i = 1\}$ and $S_y = \{i | y_i = 1\}$ be two sets that only contain the bit positions where the associated vector is one. The bit vectors x and y are unordered iff $S_x \nsubseteq S_y$ and $S_y \nsubseteq S_x$. If $S_x \subset S_y$ we say that y covers or contains x, denoted by $x \leq y$.*

The sets S_x and S_y in Definition 4 are referred to as the support of x and y [20]. Note that the term unordered bit vectors comes from the relation of these two sets. If there can be established an *order* between the support S_x and S_y (i.e. one is a subset of the other) then the associated bit vectors cannot be unordered.

Definition 5 (Unordered Code)**.** *A code C is unordered iff every pair of code words $x \in C$ and $y \in C$ ($x \neq y$) is unordered (i.e. $x \nleq y$ and $y \nleq x$).*

From this definition the following theorem immediately follows.

Theorem 1. *The minimal Hamming distance (d_{min}) between any two code words in an unordered code is two.*

Proof. Assume that there exist two code words c_1 and c_2 in some unordered code C with a Hamming distance of one. Hence in order to transform c_1 into c_2 one must either change a one in c_1 to a zero or a zero into a one. For the first case this would mean that c_1 is contained in c_2 and vice versa for the latter case. However, now we arrive at a contradiction because if code words are contained in one another C cannot be unordered. □

A very important property of DI codes is that the result of their (bit wise) inversion as well as concatenation is again DI. The size of a concatenated code, i.e. the number of representable symbols, is obtained by multiplying the sizes of the individual codes. Completion detection is then performed separately on the individual (sub-)codes and the results are joined by a C gate.

There are many properties to analyze when assessing the quality of a DI code. If function blocks should operate on the encoded data (e.g. in a 4-phase QDI processor) then the DR code is generally the only viable option. Other DI codes are, with some exceptions [21], not well suited for this task but are rather only used to transmit information in a DI fashion. This is because more sophisticated codes often have some desirable properties with regard to data transmission. When transmitting data a high coding efficiency and a low (dynamic) power consumption are of interest. As already discussed in Section 2.1 the coding efficiency R relates the number of bits encoded by a certain code word to the total number of rails required to transmit it. To minimize dynamic power during a transmission the number of transitions in the data phase should be kept as low as possible. This code property is captured by the power metric P which specifies the number of transitions required to transmit one bit of data. For the DR code $R = 0.5$ bits/rail and $P = 2$ transitions/bit. Note, however, that there is a certain trade-off between these two parameters P and R. The 1-of-16 code has, for example, a rather low coding efficiency ($R = 0.25$), while requiring only two transitions to transmit four bits of information ($P = 0.5$). The 3-of-6 code, on

the other hand, has a far better coding efficiency ($R = 0.66$) but requires six transitions for the same amount of information ($P = 1.5$).

Other very important properties of DI codes are the overhead for completion detection as well as for encoding and decoding to and from e.g. the binary representation of the data. There are several publications proposing and discussing efficient CD designs [22, 23, 3, 24].

2.2.4.3 2-Phase Delay-Insensitive Codes

For the sake of completeness we also want to briefly mention 2-phase DI codes. As shown in Figure 2.9, these protocols don't use a spacer to separate the data phases, which can obviously lead to a performance gain. However, there are of course also some drawbacks associated with these protocols. Generally, the required CDs as well as the encoder and decoder circuits are more complex and have a higher overhead than for 4-phase codes.

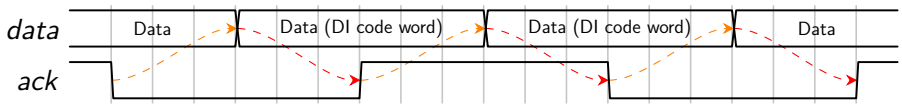


Figure 2.9: 2-Phase DI protocol

Basically, 2-phase protocols can be classified as transitional or level encoded. Using transitional encoding only signal transitions (rising and falling) transmit information. They actual state of the rails does not carry any information. Hence, here the notion of code words is a little different when compared to 4-phase codes, were a code word basically referred to a certain state of the data rails. Notice that a 4-phase code word can also be defined by the subset of rails that need be one. With this definition in mind, a 2-phase transitional code word can be defined as the subset of rails that need to make a transition. Hence, every DI code (in the sense of Definition 5) can be used for a 2-phase transitional protocol. Because of this similarly it also possible to adapt 4-phase CDs for transitional encodings [24].

Level encoded protocols like Level Encoded Dual Rail (LEDR) and Level Encoded Transition Signaling (LETS) [25] use a different approach to separate two successive data phases. Here the set of code words is divided into two groups, where each of these groups is assigned a phase. The protocol then alternates between code words of these two phases. This means that there are always two code words that convey the same logical information.

LEDR or FSL (Four State Logic) use two rails (data and parity) to encode one bit of information. The data rail carries the binary representation of the transmitted information, while the parity is used to indicate the current phase. Figure 2.10 shows a state chart of the encoding scheme. Note that only one rail (data or parity) toggles its logical value per phase. The states on the left encode logical zeros and while the states on the right encode logical ones. Note that the CD for a single LEDR bit can be implemented by an XOR gate.

Further details on 2-phase coding schemes can be found in [25]. Furthermore, [26] provides a brief comparison of different 2-phase and 4-phase coding schemes.

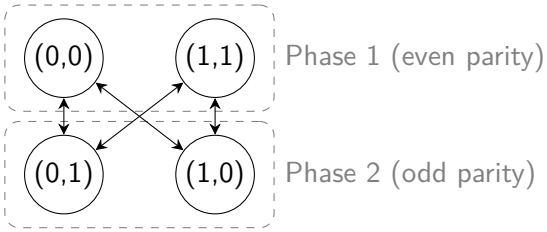
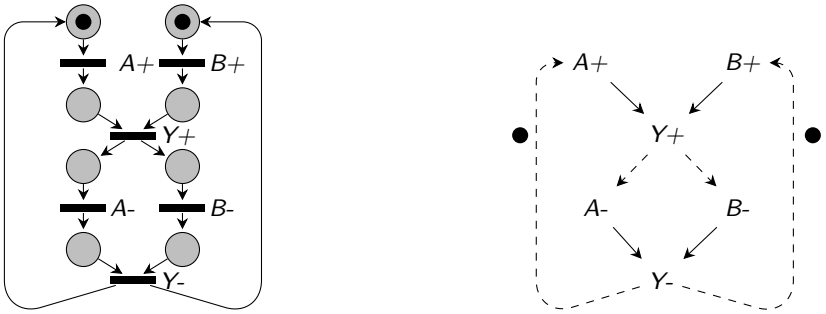


Figure 2.10: LEDR/FSL state chart

2.2.5 Control Circuit Synthesis

Control circuits play an important role in asynchronous circuits design. This section will briefly discuss a process that can be used to automatically create these circuits from a more abstract specification.

Consider the example of a Muller C gate. The timing diagram of Figure 2.3c, describing the behavior of the C gate, can also be formalized in a Petri net (PN), which is shown in Figure 2.11a. Basically a PN is a directed graph that can be used to model concurrent systems. It consists of places (gray circles) and transitions (black bars). The PN is "executed" as tokes flow through it. Tokens (black circles) are stored in places. A transition can fire if there are tokens on all its inputs. If that is the case the tokens are removed from the inputs and placed at the outputs of the transition.



(a) Petri Net

(b) Signal Transition Graph

Figure 2.11: C gate specifications

An STG is a special PN, where some restrictions apply. Here the transitions model real signal transitions and the edges of the graph basically indicate the causal and temporal order of these events. For a PN to be a meaningful STG it must be free from deadlocks, never have more than one token in a place and the signal transitions must strictly alternate between rising (+) and falling (-) transitions. Moreover, it must be guaranteed that once a transition is enabled it must fire, i.e. it may not be disabled again by another signal transition. All these restrictions as well as "input free choice" (not presented here) are discussed in more detail in [5].

For STGs, such as the one in Figure 2.11b, the places are not explicitly drawn. Every edge can be considered to contain a place. The initial state is indicated by the tokens on the appropriate edges.

Note that the STG (as well as the PN) also has to model a (well-behaved) environment of the circuit. In the example at hand the environment simply sets both inputs when the output of the C gate is zero and resets them when the output is one. By convention we draw transitions that must be fulfilled by then environment with dashed lines.

The advantage of STGs over PNs is that they can automatically be converted into (speed-independent) circuits. For this task tools like Petrify [27] or Workcraft [28] can be used. Workcraft is particularly interesting since it provides a graphical front-end to Petrify and also offers STG and circuit editors.

2.3 Fault Tolerance and Delay-Insensitive Codes

This section first introduces important concepts and terminology regarding the field of fault tolerance and dependable computing. After establishing this basis we then take a closer look at how faults affect delay-insensitive communication systems and how to deal with them.

2.3.1 Introduction and Terminology

Avizienis et. al. define dependability in the following way [29]. *Dependability of a system is the ability to avoid service failures that are more frequent and more severe than is acceptable.*

Figure 2.12 shows an overview of the different aspects of dependability as well as security. These concepts can be characterized with respect to the *attributes* a system (or service) must provide, *threats* to these attributes and *means* to uphold (correct) operation and avoid failures despite the presence of these threats. Note that although security and dependability are related concepts their main priorities lie on different aspects (i.e. attributes). This fact is indicated by the letters D and S in Figure 2.12. Detailed definitions for these attributes can be found in [29].

The terms fault, error and failure have a certain, well-defined relation to each other. A failure is the transition of a system that delivers a correct (i.e. intended) service to a state where this service is no longer provided or provided in an incorrect way. Failures are caused by errors. An error refers to an unintended system state, e.g. an erroneous value in the register file of a processor. If the processor uses this value in a subsequent instruction the system may exhibit a failure (e.g. a deadlock). Errors are in turn caused by faults. Faults can be classified as internal or external to a system. However, for an external fault to cause an error there must already be an associated internal vulnerability (i.e. fault) present in the system. Furthermore, we can distinguish between transient and permanent faults. Permanent faults are caused by physical damage to a system (e.g. a stuck-at fault in a digital circuit) and cannot be recovered without physical repair. Transient faults have, as the name suggests, only temporary effects on a system and thus also disappear without active intervention. An error is the manifestation of a fault in a system's state. This event is referred to as fault activation. Note that a failure of a subsystem can cause a fault on higher system level.

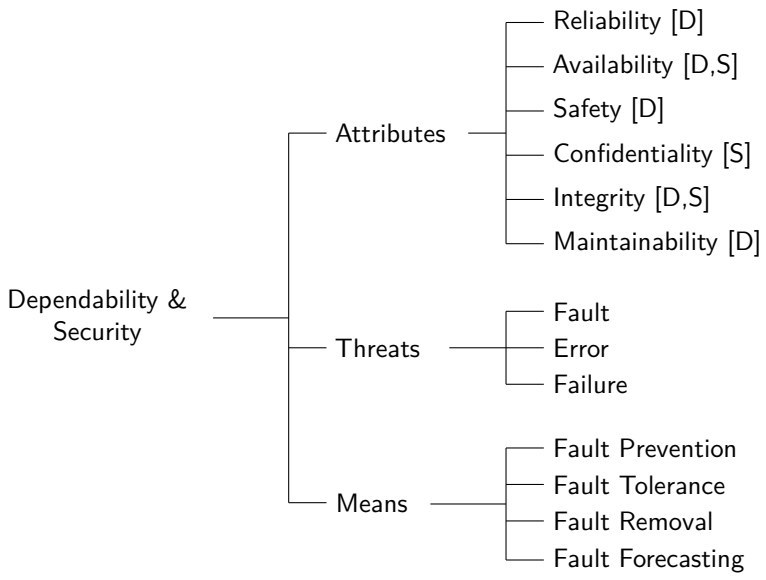


Figure 2.12: Dependability and security tree [29]

Transient faults in integrated circuits, which are the main focus of this thesis, are mainly caused by external sources. These include for example radiation effects, electromagnetic interference, electrostatic discharge, supply voltage drops and temperature changes. Due to the aggressive technology scaling in recent years, semiconductors became especially sensitive to radiation-induced faults [6]. Radiation, i.e. ionized particles hitting a circuit, can cause an unintended (short) voltage pulse on the affected node of the circuit. This pulse is referred to as Single Event Transient (SET). The situation where a SET directly hits a storage element and (directly) changes its state or causes an erroneous value at some other node of a circuit that is then captured by a storage element is referred to as Single Event Upset (SEU) or soft error. However, note that, due to temporal, logical or electrical masking not every SET may necessarily result in a soft error.

Figure 2.12 also lists the means that can be applied to mitigate faults or the effects thereof. In this thesis the concept of fault tolerance is used. However, this does not mean that arbitrary faults can be tolerated. The *fault hypothesis* exactly defines which types of faults a system can cope with and how many are to be expected within certain amount of time. Detailed discussion of the other concepts can again be found in [29].

2.3.2 Transient Faults and Delay-Insensitive Communication

As already discussed in Section 2.2.4.2, there are no assumptions on signal delays in (4-phase) DI communication schemes. Transitions of the individual rails of a DI bus may arrive at the receiver in any order and a CD is used to decide whether the input bit pattern is a valid (i.e. complete) code word or if further transitions have to be awaited.

The fact that a single transition, if considered as the last one, completes a transmission, makes DI codes specially prone to (single) faults. To illustrate this problem, consider the example transmission of a 3-of-6 code word shown in Figure 2.13.

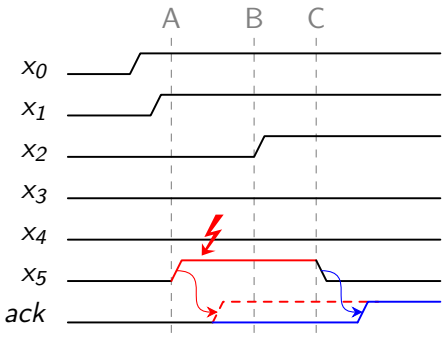


Figure 2.13: Transmission faults on a DI communication link

First two valid transitions happen on the rails x_0 and x_1. Hence, only one transition is missing to complete a 3-of-6 code word. At A a transient fault strikes, which causes rail x_5 to go high and completes the code word. Without further information the receiver is not able to identify this error and may erroneously and prematurely assert the *ack* signal (dashed line). After the last valid transition happens on rail x_2 (B) the code word becomes invalid and the receiver would (theoretically) be able detect that something went wrong. At C the transient fault vanishes and the correct code word emerges at the receiver's input. Hence, in our proposed approach (presented in Chapter 4) the receiver will be provided with just enough redundant information to detect that a transmission was affected by transient faults and to delay the *ack* signal to the point in time where all faults have vanished.

Another challenge that arises with DI codes is how the encoded data is affected by faults. Assume that the fault-free code word 111000 is mapped to the data word 0000 and the erroneous code word 110001 is mapped to 1111, then a single transient fault caused a quadruple bit error in the encoded data. This example demonstrates that the mapping of code words to data words is critical and that special care must be taken, when constructing fault-tolerant DI communication systems.

To characterize and reason about the fault resilience of DI codes the next section presents a graph theoretical approach to model their, rather unique, fault behavior.

2.3.3 Safe Overlap Graph

In [30] Lechner et. al. present a comprehensive analysis on how transient faults can affect the transmission of 4-phase DI code words. In the following we discuss these concepts in extended detail, because they are essential to the proposed coding scheme.

The Safe Overlap Graph (SOG) offers a compact way to characterize the fault resilience of a DI code. However, before the SOG can be defined formally, the notion of overlapping code words must be introduced (Definition 6).

Definition 6 (Overlapping Code Words)**.** *Two code words* $x = (x_{n-1}, x_{n-2}, \ldots, x_0)$ *and* $y = (y_{n-1}, y_{n-2}, \ldots, y_0)$ *overlap iff there exists a bit position i, where* $x_i = 1$ *and* $y_i = 1$*. This relation is reflexive, symmetric but not transitive. We denote the number of overlapping bit positions 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:* $u(x,y) = w(x) - c(x,y)$*. Analogously,* $u(y,x) = w(y) - c(x,y)$ *denotes the number of bit positions of y that do not overlap with x.*

Note that since $u(x,y)$ counts the bit positions where x is one and y is zero and $u(y,x)$ counts the bit positions where y is one and x is zero adding these two values yields the Hamming distance of the two code words ($d(x,y) = u(x,y) + u(y,x)$).

Given two code words x and y the function $u(x,y)$ returns the minimum number of faults necessary to transform an incomplete bit pattern of y into (the complete pattern of) x. To visualize this property consider the example shown in Figure 2.14. The code words x and y belong to a four bit Berger code. The bit positions (i.e. rails) are labeled with $r_0, ..., r_6$. The Hamming weight

	r_6	r_5	r_4	r_3	r_2	r_1	r_0
x	0	0	0	1	0	1	1
y	0	1	1	1	0	0	1

Figure 2.14: Example of overlapping code words

of x is three and it does not overlap with y in exactly one position (r_1). Hence, calculating $u(x,y)$ yields $3-2=1$. This means that a single transient fault (affecting r_1) is enough to transform the intermediate pattern 0001001 of y into x.

The above example also demonstrates the asymmetric nature of the function u ($u(x,y) = 1$ while $u(y,x) = 2$). While it only takes a single fault to confuse y with x, at least two faults (at the positions r_4 and r_5) are required to transform (an intermediate pattern of) x into y. In general we can state that a code word y cannot be confused with code word x iff $u(x,y) > f$, where f is the number of faults in the fault assumption.

By calculating the function u for every code word pair in a particular DI code C the SOG can be constructed.

Definition 7 (Safe Overlap Graph)**.** *Let C be an unordered code and f be the number of transient faults that can affect the transmission of a code word of C, then the undirected graph* $G = (V,E)$ *is the Safe Overlap Graph of C (under the fault assumption f), where*

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

Two code words are connected in the SOG if it is not possible that they can be confused with one another under the fault assumption of up to f faults. Some examples for SOGs can be found in Figure 2.15.

In the course of the work done for this thesis, we identified further properties of the SOG. For the sake of clarity and completeness we also included these findings into this section.

Theorem 2. *The minimal Hamming distance between two code words which are connected in the SOG is* $2f+2$.

Proof. We know that for two nodes represented by the code words x and y to be connected in the SOG $u(x,y) > f \wedge u(y,x) > f$ must hold. This can be rewritten as $u(x,y) \geq f+1 \wedge u(y,x) \geq f+1$. By inserting these inequalities into $d(x,y) = u(x,y)+u(y,x)$, we arrive at $d(x,y) \geq (f+1)+(f+1) \rightarrow d(x,y) \geq 2f+2$. □

Note that the converse is not true in general. However, for the class of m-of-n codes the implications are valid for both directions. Since the code words of an m-of-n code have constant Hamming weight u is symmetric for this code class. In this case $u(x,y)$ is simply given by $d(x,y)/2$. Hence, the definition of the edge set of the SOG can be rewritten as $E = \{(x,y)|x,y \in C, d(x,y) > 2f\}$. Because of the symmetry of the Hamming distance the resulting graphs are regular (see Theorem 3).

Theorem 3. *A SOG constructed from an m-of-n code is k-regular (the vertex degree is constant), where k is given by*

$$k(m,n,f) = \sum_{j=0}^{m-f-1} \binom{m}{j} \binom{n-m}{m-j}.$$

Proof. Let c be an arbitrary code word of an m-of-n code. We know that there are exactly m positions in c which are one. To calculate the number of code words which do not overlap with c in more than f positions (i.e. safe code words) we can now use the following iterative process. First consider the set A_j of code words which overlap with c in $j = m-f-1$ positions. Note that j is the minimum distance to c a code word must have in order to be considered safe by the definition of the SOG (see Theorem 2). Hence, j is also the maximum number of overlapping bit positions a safe code word is allowed to have with c. The number of code words in A_j is given by $\binom{m}{j}$ multiplied by $\binom{n-m}{m-j}$. The first term calculates the number of combinations how j ones can be arranged in the m bit positions where c is one. These j ones constitute the overlapping bit positions. The second term calculates the number of combinations how the remaining $m-j$ ones, which are needed to form a valid m-of-n code word, can be arranged in the $n-m$ bit positions where c is zero (non-overlapping bit positions). This process can be repeated for all A_{j-x}, where $x > 0$ and $j-x \geq 0$, until all cases for the number of overlapping bit positions have been covered. The overall number of safe code words with respect to c is then obtained by summing up the sizes of the sets $A_0, ..., A_j$. Since we did not make assumptions about c, this holds for every code word and therefore every node of the SOG. □

Tables 2.1a and 2.1b show the vertex degrees of SOGs constructed from m-of-n codes for all $n \leq 12$ and fault assumptions $f = 1$ and $f = 2$, respectively. Note that the first row ($m = 1$) in Table 2.1a and the first two rows ($m \leq 2$) in Table 2.1b are zero, i.e. the corresponding SOGs don't have edges. This is the case, because in a 1-of-n code a single fault can change the spacer (all zero pattern) to every possible valid code word. The same is true for 2-of-n codes under the fault assumption $f = 2$. Hence, all code words can be confused with one another and no safe code words pairs exist.

m \ n	3	4	5	6	7	8	9	10	11	12
6				0	0	15	65	185	431	887
5			0	0	10	40	105	226	431	756
4		0	0	6	22	53	105	185	301	462
3	0	0	3	10	22	40	65	98	140	192
2	0	1	3	6	10	15	21	28	36	45
1	0	0	0	0	0	0	0	0	0	0

(a) $f = 1$

m \ n	3	4	5	6	7	8	9	10	11	12
6				0	0	0	20	95	281	662
5			0	0	0	10	45	126	281	546
4		0	0	0	4	17	45	95	175	294
3	0	0	0	1	4	10	20	35	56	84
2	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0

(b) $f = 2$

Table 2.1: Vertex degrees in m-of-n SOGs

m \ n	4	5	6	7	8	9	10
6				1	4	12	30
5			1	3	8	18	36
4		1	3	7	14	18	30
3	1	2	4	7	8	12	13
2	2	2	3	3	4	4	5
1	1	1	1	1	1	1	1

(a) $f = 1$

m \ n	4	5	6	7	8	9	10
6				1	1	1	5
5			1	1	2	3	6
4		1	1	2	2	3	5
3	1	1	2	2	2	3	3
2	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1

(b) $f = 2$

Table 2.2: Maximal clique size in m-of-n SOGs

Another point worth noting is that, although some SOGs have the same vertex degrees, this does not imply that there exists a (subgraph) isomorphism (e.g. 2-of-7, 3-of-6) between them. The following subsections present two applications of the SOG proposed in [30], to create fault-tolerant DI coding schemes. Note that both methods aim to mitigate the effects of transient faults only. Furthermore, it is assumed that the DI communication channel does not contain storage elements (i.e. pipeline stages).

2.3.3.1 Fault-Tolerant Subcodes

For the first approach the SOG is used to find a fault-tolerant subcode to the original DI code. This process generally results in a code with fewer symbols.

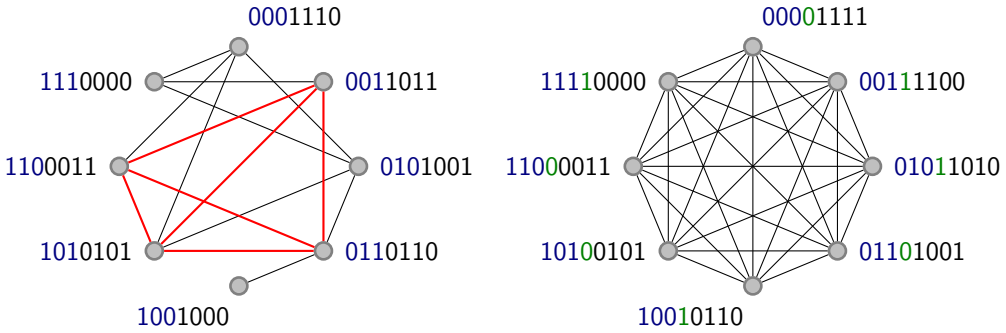
The problem of generating the subcode can be reduced to the well-known graph theoretical problem of finding the largest clique in a graph. A clique is a fully connected subgraph and represents a (safe) group of code words that cannot be confused with each other in the presence of f faults. Hence a DI link that only uses code words of such a clique is able to tolerate up to f transient faults. This obviously implies that the CD used in the receiver must not react to the unused code words (i.e. code words that are not element of the safe group).

Tables 2.2a and 2.2b show the maximal clique sizes for m-of-n codes of different sizes for the case of $f = 1$ and $f = 2$, respectively.

Figure 2.15a shows the SOG of the 3-bit Zero-Sum code (see Section 3.2) with fault assumption $f = 1$. Note that, since this code is systematic, the data part of each code word has

been highlighted. The largest clique in this SOG comprises four nodes. This means, however, that this Zero-Sum code is (without reduction of the code size) not able to tolerate one arbitrary transient fault. Consider for example the code words $c_1 = 1100011$ and $c_2 = 1110000$, which are not connected in the SOG. When c_1 is being transmitted, a single fault striking on the fifth position, i.e. rail, is able to transform the intermediate pattern 1100000 into the valid code word c_2. Without additional information the receiver would not be able to detect this incidence and might erroneously process the wrong code word.

Notice, that a DI code that is by itself fault tolerant can be easily identified by the means of its SOG. Figure 2.15b shows the SOG of a Dual Rail Parity Code (see Section 3.1). Since this code is systematic as well, we again highlighted the data part and the associated parity bit of each code word. It can be seen that the SOG resembles a complete graph with 8 nodes. Hence, trivially the largest clique also has 8 nodes and we can conclude that this code is tolerant against one fault.



(a) 3 bit Zero-Sum code

(b) 3 bit Dual Rail Parity Code

Figure 2.15: Example SOGs ($f = 1$)

The maximal clique problem is NP-complete, which makes it intractable for large SOGs and their associated codes. Note, however, that in the case of m-of-n codes, it is not necessary to use the whole graph, when searching for a maximal clique. It is sufficient to pick one code word $c \in C$ as "fixed" member of the maximal clique and remove all nodes from the SOG that are not direct neighbors to c. The maximal clique that can be found in this sub graph is also a maximal clique of the SOG. This simplification follows from the symmetric nature of SOGs of the m-of-n code class.

Theorem 4. *For every code word $c \in C$ in an m-of-n code there exists a maximal clique that contains c.*

Proof. Let K denote a maximal clique in the SOG of C and c some code word of K. Since every code word in C has the same Hamming weight, it is possible to transform c into any other code word c' by applying a certain permutation to the individual bit positions. By applying the same transformation to the other code words in K, a new maximal clique K' containing c' is constructed. Note that by permuting the bit positions the overlapping property (Definition 6) is preserved. □

Theorem 4 allows the reduction of the size of the input graph for the maximal clique problem from $\binom{n}{m}$ to $k(n,m,f)+1$ nodes, where k is defined in Theorem 3.

2.3.3.2 Combining Error Detecting and Delay-Insensitive Coding

The other application utilizing the SOG is the combination of error detecting and DI coding. Here the data is secured by an error detecting code before the actual DI code is applied. The SOG is used to find an optimal mapping of data words to code words of a particular DI code that minimizes the required strength of the error detecting code that provides the fault tolerance. As shown in Section 2.3.2 the mapping of data words to code words is crucial, because a single fault in the code word can, in theory, lead to arbitrarily many bit errors in the decoded data word.

Figure 2.16 shows an overview of the proposed coding scheme. In the first step a (binary) error detecting code is used to calculate the check bits. Then the resulting bit vector (data and check bits) is partitioned into equally sized blocks and encoded with the desired DI code.

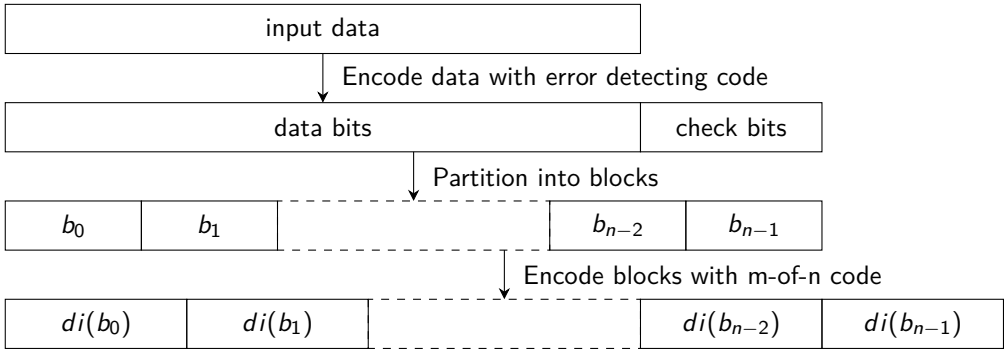


Figure 2.16: Fault-tolerant coding scheme [30]

The question that must be answered now is how strong the error detecting code must be chosen, such that all bit errors that can be produced in the encoded data by transient faults affecting the DI code word can be detected. The code's strength is of course measured in terms of number of detectable bit errors. As mentioned before this number depends on the chosen mapping of data words to DI code words.

To find this mapping an iterative process is applied. In the core of this process again lies a graph theoretical problem namely the task of finding a subgraph isomorphism between two graphs.

First, a candidate error detecting code is selected, starting with a simple parity code. Then the error detection capability of this code is modeled in the graph ED. The vertices of this graph are the data words encoded by the selected DI code. Edges are added between two data words d_1 and d_2, if the error detecting code would not be able to detect if an error changed the contents of a DI block from d_1 to d_2 or vice versa. Hence, the ED graph basically models the shortcoming of the selected code. In the second step the SOG of the DI code is generated. To find a suitable mapping of code words to data words, we must now try to establish a subgraph isomorphism between the SOG and the ED graph. If such an isomorphism exists, then every problematic data word pair

can be mapped to a code word pair that cannot be confused under the fault assumption. In the case that no subgraph isomorphism exists, the error detecting code was chosen too weak and the process must be repeated with a stronger code (higher error detection capability).

Note that the coding scheme proposed in Chapter 4 is inspired by these techniques and uses similar concepts.

CHAPTER 3

Related Work

DI codes and communication architectures have been widely studied in literature [3, 31, 25, 5]. This section covers those techniques and approaches that are conceptionally related to ours in that they aim to increase the fault resilience of such communication schemes.

3.1 Cheng & Ho

The first paper that investigates the effects of transient as well as permanent faults on 4-phase DI communication channels was presented by Cheng & Ho [32]. In their work they define three error models and propose possible solutions to deal with a certain number f of faults. While the two asymmetric models only assume unidirectional (i.e. either $0 \rightarrow 1$ or $1 \rightarrow 0$) faults, the general model allows both fault types to occur. For the asymmetric $1 \rightarrow 0$ model not all transitions always arrive at the receiver. To solve this problem the authors propose to use an m-of-n code and a CD that prematurely triggers when only $m - f$ transitions have arrived. Hence the used code must provide an error-correcting capability of $2f$ bits. To provide the required fault tolerance in the case of the asymmetric $0 \rightarrow 1$ model a normal CD can be used. However, the code must still be able to correct $2f$ faults, since in DI coding one fault can lead to two bit errors in the code word. For the reset phase a timeout mechanism is used. If the spacer does not emerge on the DI bus after a certain amount of time, the acknowledgment signal is deasserted anyway. This behavior is required to be able to deal with permanent (stuck-at-one) faults. The paper further shows that for the general fault model, the used DI (m-of-n) code must be able to correct $3f$ faults. Again a prematurely triggering CD must be used.

Finally the so called Dual Rail Parity Code (DRPC) is proposed, which can be employed when either one of the asymmetric error models with the fault assumption of $f = 1$ is assumed. The encoding process for this code works by taking the binary input data and appending a parity bit. The resulting vector is then dual-rail encoded.

3.2 Agyekum & Nowick

In [33] and [34] Agyekum & Nowick propose Zero-Sum codes, a new class of unordered codes that are conceptionally related to Berger codes[35] and can be viewed as a generalization thereof. Berger codes are systematic unordered codes. Hence their code words can be separated into a data part d (containing the unencoded binary representation of the data) and a synchronization (or check) part $s = sync(d)$, which makes the code unordered. The Berger code uses the binary representation of the number of zeros in the data part as synchronization bits.

Zero-Sum codes additionally associate a weight to every (bit) position in the data part. The synchronization part is then calculated by summing up the weights of the bit positions, which are zero in the data part (hence the name). Figure 3.1 shows an example of the encoding process for a 5-bit Zero-Sum code.

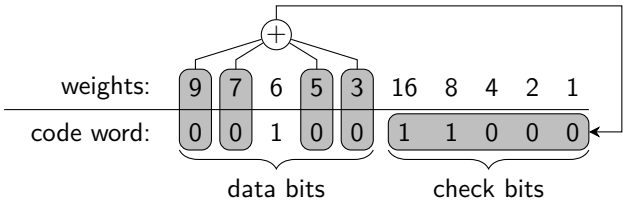


Figure 3.1: 5-bit Zero-Sum encoding example

The data part sum of this code word is 24, yielding 11000 for the synchronization bits. The required number of digits for the synchronization part is given by binary logarithm of the sum over all weights rounded up to the next integer. In this example the maximal possible sum value is $3+5+6+7+9 = 30$, thus five check bits are required.

It is easy to show that every possible weight assignment for the data part guarantees that the resulting code is DI. However, the class of Zero-Sum codes explicitly demands ascending non power-of-two values. Using this weight assignment a code with minimal Hamming distance of $d_{min} = 3$ is generated, which allows for the correction of one or the detection of up to two errors. Moreover, [34] also introduces the class of Zero-Sum$^+$ and Zero-Sum* codes. Using an additional parity bit the Zero-Sum$^+$ code offers three bit error detection as well as detection for all odd numbers of errors. The Zero-Sum* code employs a slightly different weight assignment and can thereby support multi-bit error correction.

However, as shown by Lechner et. al. [30] none of the proposed Zero-Sum codes offers sufficient error-detecting capabilities, when used in a purely DI communication scheme. Since it can be shown that there are always some unconnected nodes (i.e. code words) in the SOGs generated from these codes, there is always a possibility that some code words may be confused with each other. Such a confusion can not be detected solely based on the check information provided by the different Zero-Sum codes.

3.3 Pontes, Calazans & Vivet

Pontes et. al. propose two different approaches to add fault tolerance to DI codes. In [36], the Temporal Redundancy Delay Insensitive Code (TRDIC) is introduced. This coding scheme can only be applied to DI data that is already encoded with a 1-of-n code. The data (i.e. the 1-of-n code words) that should be transmitted is combined with the data sent in the previous transmission cycle and transformed into a 2-of-(n+1) code. This transformation is achieved by a bit-wise OR operation between the two successive (1-of-n) code words d^i and d^{i-1}. If both code words are equal, i.e. the same data is transmitted in two successive transmission cycles, the last bit of TRDIC code word is set to one (see Equation 3.1).

$$c_{TRDIC} = (d_0^i \vee d_0^{i+1}, ..., d_{n-1}^i \vee d_{n-1}^{i+1}, d^i = d^{i-1}) \tag{3.1}$$

Obviously this construction always leads to a 2-of-(n+1) code word. The paper argues that a DI channel using the proposed code is less prone to faults because single faults can no longer produce new data tokens, as would be the case using a 1-of-n code. Furthermore, since the receiver has knowledge about the previous transmitted (1-of-n) code word (d^{i-1}) it is able to predict one bit position in c_{TRDIC} that must be one. If the received data does not have the expected shape, a transmission fault occurred.

The other approach presented in [37] can be applied to arbitrary m-of-n codes. The data to be transmitted is first divided into equally sized blocks. Each of these blocks is then individually encoded with an m-of-n code. In a second step n parity bits are calculated, one for each bit position in the m-of-n code words. These parity bits are in turn divided into blocks, encoded with the same m-of-n code and transmitted alongside the actual data blocks. Based on the provided redundant information the receiver is able to detect and correct a single fault. As we will see in Chapter 4, this approach has some similarities to the coding scheme proposed in this thesis. Both schemes use a combination of DI and error detecting codes. However, here the error detecting code is applied to the DI data whereas the new approach calculates the check information before DI encoding is performed.

3.4 Lechner et. al.

Besides the theoretical results on DI codes from [30] (already discussed in Section 2.3.3) Lechner et. al also propose a generic fault-tolerant DI communication link architecture [38, 39]. This architecture is basically applicable to arbitrary DI codes. However, an implementation example is only given for the dual rail (1-of-2) encoding. In this approach the data is first extended by a parity bit and then encoded by the dual rail code, similarly to the DRPC introduced in [32]. The receiver captures the transmitted data word in an input register and uses the redundant information of the dual rail encoding and the parity bit to either perform forward error correction or issue a re-capture of the DI code word, if correction is not possible. Interestingly, [38] is the only work that also considers the possibility of metastability. We will use the idea of re-capturing erroneous data for the solutions provided in this thesis. In particular the proposed circuits will serve as a basis for the implementation of our link architecture discussed in Chapter 5.

CHAPTER 4

A New Fault-Tolerant Coding Scheme

As the main contribution of this thesis, this chapter covers the new fault-tolerant delay-insensitive coding scheme. Parts of this work have already been published in [40]. The proposed coding scheme is based on a combination of error detecting and delay-insensitive coding. While similar approaches have already been proposed (see Section 2.3.3.2), the solution at hand generally offers a better coding efficiency while retaining a relatively small implementation overhead.

This chapter is structured in the following way. First Section 4.1 defines the requirements and assumptions we impose on the hardware implementing the proposed coding scheme and formulates the fault hypothesis. Section 4.2 presents an extension to the SOG fault model, which also includes invalid code words. The actual encoding and decoding processes are discussed in Sections 4.3 and 4.4, respectively. Finally, Section 4.5 briefly shows how the proposed coding scheme can also be used to perform forward error correction.

4.1 Hardware Model and Fault Hypothesis

Figure 4.1 shows an outline of the type of DI link the coding scheme proposed in this chapter can be applied to. The sender and receiver both have bundled data interfaces to receive data from preceding components and to transmit it to succeeding ones. For these interfaces the 4-phase protocol is used.

As indicated in Figure 4.1 this work is focused on mitigating the effects of *transient* faults affecting the data rails of the DI bus. We do not address faults interfering with the internal components of the transmitter or receiver circuits. This issue could, for example, be handled by a TMR solution [41].

Another problem, that may arise, are faults that erroneously trigger the acknowledgment signal *ack*. Such faults can of course have severe consequences on the handshaking protocol. By wrongly notifying the transmitter that the current protocol phase has completed, data may be corrupted or the whole link can get dead locked. However, the protection of a single control signal is a rather simple task and could for example be solved by triplicating the acknowledgment signal for transmission and rejoining the replicas at the receiver by a C gate.

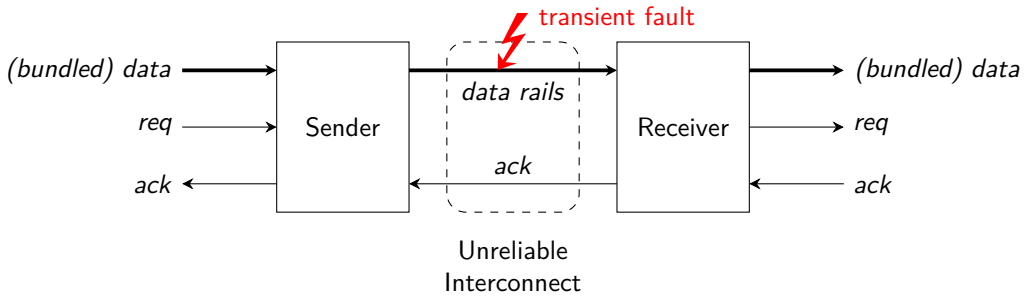


Figure 4.1: Delay-insensitive link

Similar to the approach described in [39] and [30] there may not be any storage elements, i.e. pipeline stages, along the DI bus. With this constraint it is possible to safely assume that transient faults will eventually vanish after a certain amount of time and cannot manifest themselves in storage elements (along the link). This is a very fundamental requirement, because a large part of the concepts and techniques discussed in this work do not work otherwise. Figure 4.2 shows a general model of the receiver circuit.

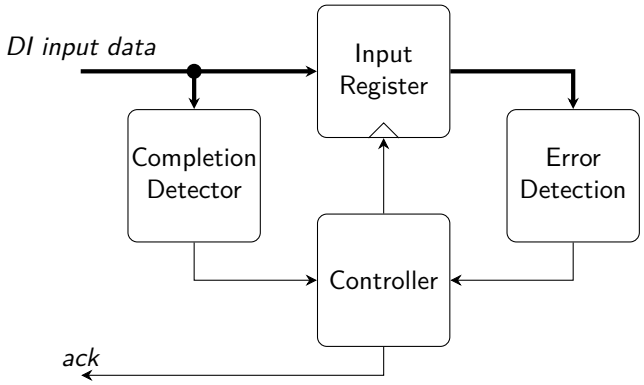


Figure 4.2: Fault-tolerant receiver model

The DI input data is captured into the input register when the CD notifies the controller that it is complete. Next the receiver performs error detection on this snapshot of the input data. If it detects that data has been corrupted (i.e. a transient fault occurred), the controller will recapture the input data and the whole process starts over. This way the receiver can simply wait for all transient faults to disappear until it finally acknowledges the reception of the data by asserting the *ack* signal.

Although the proposed approach is focused on error detection and resampling of data, we will nevertheless also briefly investigate how the presented scheme can be extended and modified in order to allow for forward error correction.

Throughout the chapter f will denote the number of (single bit) faults possible under the fault

assumption. Note that a sufficiently large amount of faults happening simultaneously could even produce new data tokens. However, to rule out this possibility, we assume that the total number of transitions in the transmitted data is greater than f.

4.2 Fault Model

As we have seen in the discussion in Section 2.3.2, it is easily possible that a fault changes one valid code word of a DI code into another. It has further been shown how the SOG can be used to capture this behavior and identify code words that can be confused with each other and others that don't. However, the example transmission of Figure 2.13 also demonstrated that the point in time, when the receiver actually captures the state of the DI bus into its input register is crucial. Due to a delay between completion detection and the actual latching event, the code word that triggered the CD is not necessarily the one that is finally latched. This holds for late valid transitions as well as late faults that may supersede a previous different trigger. Hence, the first contribution of this chapter is a fault model describing which invalid code words are to be expected at the receiver's input and which valid code words may have been the original, i.e. fault free, versions of them. This model is complementary to the SOG and provides vital information to prove that the proposed coding scheme operates correctly under all circumstances (covered by the fault hypothesis). In our model we consider both $0 \rightarrow 1$ as well as $1 \rightarrow 0$ faults.

For the following sections let $C \subset \mathbb{F}_{2^{k'}}$ be a DI code. Note that the representation of code words as elements of a finite field with a size of a power of two is equivalent to a bit vector representation. The elements of $\mathbb{F}_{2^{k'}}$ can also be represented as polynomials of degree less than k' and coefficients from $\mathbb{F}_2$ (i.e. $\{0,1\}$). These coefficients exactly resemble the bit vector.

A code word is regarded as element of C if the CD at the receiver is triggered by it. Consequently $C' \subseteq C$ denotes the set of actually used code words, i.e. code words with data words mapped to them. Further let U denote the set of unused code words and I the set of invalid code words reachable over f faults from the set of actually used code words C'.

A code word is unused if there is no data word mapped to it although the CD at the receiver reacts to it, hence $U = C \setminus C'$. This means that a correctly working transmitter will never apply such a code word to the DI data bus. However if, due to a transmission fault, the code words appears at the receiver's input the CD is triggered and it can be captured in its input register. While for Berger and Zero-Sum codes, the sets C and C' are equal (i.e. all valid code words are used, hence $U = \{\varnothing\}$), the size of constant weight codes is generally not a power of two and only a subset of the available code words can actually be used. For these codes the CD implementation is decisive. Consider for example a 2-of-5 code, which is able to encode ten symbols. If a sorting network based CD [22] is used, all ten code words trigger the CD. Hence U encompasses two code words. If, on the other side, the CD is constructed from the DIMS approach, two unused code words can simply be left out from membership test, resulting in $U = \{\varnothing\}$.

The definition of the set of invalid code words is a little bit more involved. A simple approach would be to declare all $\mathbb{F}_{2^{k'}} \setminus C'$ vectors as invalid code words. However, depending on the fault assumption not all of these vectors can actually appear at the decoder's input, which makes this approach too restrictive. In order to calculate the set I, we apply the following procedure. For every valid code word c sent by the transmitter, we use the function $N_I(c)$ to calculate the set of

"neighboring" invalid code words, i.e. vectors to which c can be transformed in the presence of f faults. For this task we need the following auxiliary functions (Equations 4.1-4.4).

$$N_H(x,f) = \{z \in \mathbb{F}_{2^{k'}} | h(x,z) \leq f\} \tag{4.1}$$

$$N_V^+(x,f) = \{z \in C | u(z,x) \leq f\} \tag{4.2}$$

$$N_V^-(x,f) = \{z \in C' | u(x,z) \leq f\} \tag{4.3}$$

$$T(x,y) = \{z \in \mathbb{F}_{2^{k'}} | z \vee (x \leftrightarrow y) = x \vee y\} \tag{4.4}$$

The function $N_H(x,f)$ calculates the set of code words (valid or invalid) reachable from x over (up to) f bit flips, i.e. the set of vectors with a Hamming distance smaller or equal to f (note that $x \in N_H(x,f)$). $N_V^+(x,f)$ computes the set of valid code words (i.e. code words that can trigger the CD), reachable from an intermediate pattern of the code word x over f faults ($N_V^+(x,0) = \{x\}$). For the sake of completeness we also define the function N_V^-. Given a code word x (that triggered the CD), it returns the set of code words that could have been sent by the transmitter, if up to f faults affected the transmission. It basically answers the complementary question to N_V^+. Note, however, that in contrast to N_V^+, here only code words of C' are considered, because an unused code word can never be a candidate for the originally transmitted code word. Finally the function $T(x,y)$ returns the set of vectors which lie "between" x and y. This is the case for all vectors that constitute a possible intermediate pattern on a bus that switches from x to y. Note that x and y themselves are element of $T(x,y)$. The symbols $\vee$ and $\leftrightarrow$ in the definition of $T(x,y)$ denote bitwise OR and COR operations. Now we have all that we need to specify $N_I(c)$.

$$N_I(c) = \left(\bigcup_{v=0}^{f} \bigcup_{x \in N_V^+(c,v)} \bigcup_{y \in T(c,x)} N_H(y, f-v) \right) \setminus C \tag{4.5}$$

To understand the intuition behind Equation 4.5 consider a DI communication link such as the one shown in Figure 4.1 in its null phase (all data rails are zero). Now the transmitter applies the code word c to the DI bus. Remember that due to the DI timing model the transitions can arrive at the receiver in any order. At this point we make a case distinction, which corresponds to the first union operator in 4.5. Assume that the transmission of c is affected by v transient faults, where $0 \leq v \leq f$. This changes the code word c to another code word $x \in N_V^+(c,v)$ (second union operation), which then triggers the CD of the receiver. The output of the CD ultimately causes the receiver to capture the current data on the DI bus into its input register. However in the time window, spanning from the point in time where there is a valid code word (x) at the input of the CD and the actual triggering of the input register, the DI bus can still change its value. The transient fault(s) that led to the erroneous code word x can disappear. Simultaneously missing transitions of the code word c can reach the receiver. Thus if c has originally been applied to the DI bus, but x triggered the CD every code word $y \in T(c,x)$ can end up at the receiver's input register. However, to make matters worse, further $f - v$ faults can additionally happen in the

same time window. Hence to cover all cases we also have to consider all vectors that lie within a Hamming distance of $f - v$ of all possible values for y, given by $N_H(y, f - v)$ (third union operation). Finally, all valid code words that where reached by the described enumeration process are removed from the result. By calculating N_I for all code words in C' the set I of invalid code words that can appear at the receiver's input is obtained.

$$I = \bigcup_{c \in C'} N_I(c) \tag{4.6}$$

Using N_I we can further define the function Θ, which basically answers the opposite question to N_I. If an invalid code word x is received, $\Theta(x)$ returns the set of valid code words that could have been sent by the transmitter. Thus we know that the code word c originally transmitted is an element of the set $\Theta(x)$.

$$\Theta(x) = \{z \in C' | x \in N_I(z)\} \tag{4.7}$$

4.3 Encoding

Figure 4.3 shows an overview of the steps involved in the encoding process of the proposed fault-tolerant DI coding scheme. In the first step the binary input data is partitioned into equally sized blocks of k bits. The result of this step are n data blocks $d_0, ..., d_{n-1} \in \mathbb{F}_{2^k}$. These blocks are then encoded with the function $di_d : \mathbb{F}_{2^k} \mapsto \mathbb{F}_{2^{k'}}$ to generate the DI data blocks $D_0, ..., D_{n-1}$, which are ready to be transmitted over a DI communication channel. The approach basically works with every DI code. Note, however, that the selection of the code as well as the mapping of data words to code words affect others parts of the encoding process (especially f_c, which is defined below).

The actual check information for the transmitted data is generated in a second processing path. For each data block $d_0, ..., d_{n-1}$, the function $f_c : \mathbb{F}_{2^k} \mapsto \mathbb{F}_{2^j}$ calculates the so called Per Block Check Bits (PBCB). The purpose of this function is to reduce the number of bits in the data blocks in such a way that every fault possible under the fault assumption is still "visible" in the PBCB after DI decoding. Since the error detecting code ($ed : (\mathbb{F}_{2^j})^n \mapsto (\mathbb{F}_{2^j})^m$), which creates the check blocks $c_0, ..., c_{m-1}$, is calculated based on the results of this step, a smaller bit width will result in a simpler encoder as well as smaller check blocks.

Finally the function $di_c : \mathbb{F}_{2^j} \mapsto \mathbb{F}_{2^{j'}}$ is used to generate the DI check blocks $C_0, ..., C_{m-1}$, which are then transmitted alongside the DI data blocks. Unlike to di_d, for di_c the mapping from code words to data words is uncritical, with respect to error detection, as we will see in the next sections.

In the following the functions f_c and ed are discussed in more detail. Both processing steps are of course dependent on the number f of faults defined by the fault assumption. Note that the variables as well as functions introduced so far, will be used throughout the next (sub)sections, e.g. k always denotes the number of bits in a data block.

4.3.1 Per Block Check Bits (f_c)

Basically f_c depends on the mapping of code words to data words of the selected DI code as defined by di_d. In Section 2.3.3 it was shown how DI codes can be analyzed for their fault

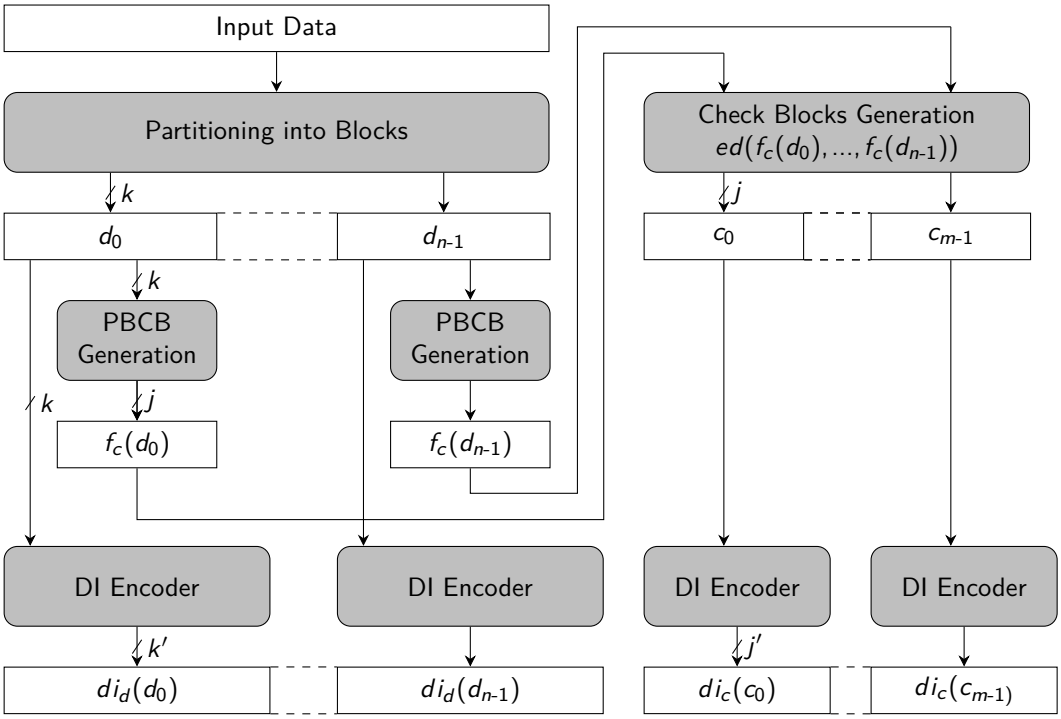


Figure 4.3: Encoding overview

resilience using the SOG. Recall that the SOG provides means to identify code word pairs that can be confused under a certain fault assumption (f) and others that can not (i.e. safe pairs). The basic idea is now to define f_c in such a way that data words, whose associated code words can be confused, are mapped to different check bit patterns. Code words that can not be confused can have the same check bits. Hence the receiver will be able to detect if an incomplete intermediate bit pattern in a DI data block has been transformed into a complete but erroneous code word. In other words, by exploiting the fact that not all DI code words can be confused with each other, the required amount of check information is reduced.

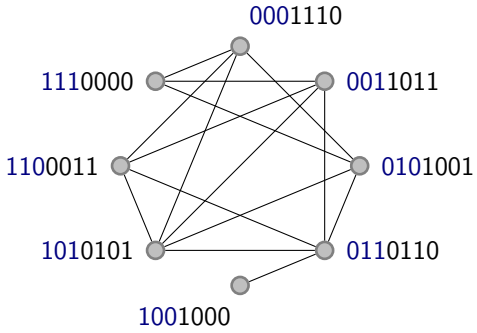
We have identified two strategies which can be applied to define f_c. In both cases the SOG is used to find a suitable definition.

$\mathbf{di_d \rightarrow f_c}$: Sometimes the mapping of data words to code words is fixed, e.g. if there already exist efficient hardware implementations for encoder and decoder circuits or if the used DI code is systematic, like in the case of Berger or Zero-Sum codes (see Section 3.2). In these cases f_c needs to be defined based on a predefined di_d. To visualize the steps involved in this procedure we use the example of a 3 bit Zero-Sum code and a fault assumption of $f = 1$. Figure 4.4a shows how this code maps the data words d to its code words. Since the used code is systematic, observe that each data word is contained in the three MSBs of its corresponding code word.

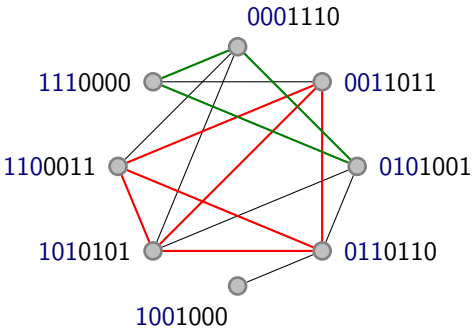
To define f_c, first the fault resilience of the used DI code is modeled using the SOG (see

d	$di_{ZS3}(d)$
000	0001110
001	0011011
010	0101001
011	0110110
100	1001000
101	1010101
110	1100011
111	1110000

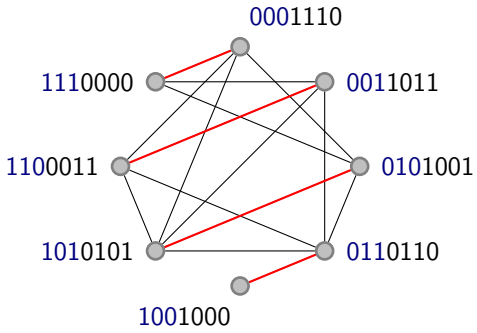
(a) DI encoding function



(b) SOG ($f = 1$)



(c) Partitioning: K_4, K_3, K_1



(d) Partitioning: $4 \times K_2$

Figure 4.4: Example: 3 bit Zero-Sum code

Figure 4.4b). Now the graph is partitioned into a set of non-overlapping, i.e. vertex disjoint cliques. Figures 4.4c and 4.4d show two possibilities how this can be done for our example code. Notice that these cliques don't need to have the same size. Every clique (or partition) represents a group of code words, which cannot be confused with each other under the fault assumption. Therefore every code word in such a group can be assigned the same check bit pattern. If a transmission fault transforms an incomplete intermediate pattern into a valid but incorrect code word, we can be sure that this code word will have different PBCB. The total number of cliques yields the bit width of the PBCB, which is denoted by j. In our example, two check bits would be required, to uniquely identify every clique. If this number is smaller than the data block width, a reduction has been achieved.

Note, however, that this is not possible for every DI code. For instance, SOGs derived from 1-of-N codes, don't have any edges. Hence a partitioning into cliques always results in N K_1 graphs[1], which in turn requires $k = \log_2(N)$ PBCB. This observation can also be generalized in the following way. SOGs derived from m-of-n codes where $f \geq m$ don't have edges and hence don't allow for $j < k$ PBCB.

[1] In graph theory K_n denotes a complete graph with n nodes, i.e. a graph where every vertex is connected the every other vertex.

$\mathbf{f_c} \rightarrow \mathbf{di_d}$: For this approach we fix f_c and try to find a suitable definition for di_d, i.e. a mapping from data words to code words. Note that this is not always possible. If f_c is chosen too weak then there does not exist a corresponding di_d. The example code that will be used in this section is a 2-of-5 code, which is able to encode three bits of data. The fault assumption is $f = 1$.

With a potential hardware implementation in mind the function f_c should be defined as simple as possible. It should only perform very simple or no computations at all. Figure 4.5a shows the selected definition for f_c for the example code. Note that here f_c is simply a mapping to the left-most two bits of every data word, hence no actual computation is required. Now the error detecting capability of f_c is modeled in the graph ED. The vertices of this graph are the data words that are encoded with the selected DI code. Edges between two data words are added if they have the same PBCB. This means that the ED graph specifies which data words could not be distinguished based on their PBCB. The formal definition is given in Equation 4.8. Note that, this definition implies that ED is a disconnected graph, where the connected components are complete graphs.

$$ED = (V,E), V = \mathbb{F}_{2^k}, E = \{(x,y) | f_c(x) = f_c(y)\} \tag{4.8}$$

In the second step the SOG of the selected DI code is constructed (Figure 4.5c). Now the task of finding a suitable definition for di_d can again be reduced to a graph theoretical problem. Similar to the approach discussed in [30], we try to find a subgraph isomorphism between ED and the SOG, i.e. find a subgraph in the SOG which is isomorphic to ED. If such an isomorphism exists, a mapping of data words to code words can be derived. As shown in Figure 4.5d this is indeed possible for our example. The bit pattern below the horizontal line represents the data word which is mapped to the respective code word.

Now the question arises whether in this case it would have also been possible to use PBCB with just a single bit in width. Note that no matter how f_c would defined in this case, the graph ED would always be composed of two disconnected complete subgraphs K_x and K_y, where $x + y = 8$. It is easy to see that the biggest clique contained in the SOG is only a K_2, hence there cannot exist a subgraph isomorphism.

A simple algorithm to calculate the minimum number of PBCB required for a particular DI code, can thus be implemented by a linear search over all values for $1 \leq j \leq k$.

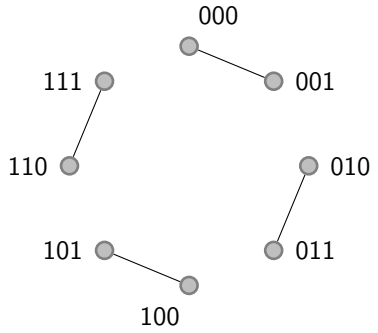
Note that both of the approaches discussed in this and the previous section, basically perform the same operation. The SOG is partitioned into a set of cliques Π, where each element $P \in \Pi$ represents a set of code words which cannot be confused under the fault assumption and which are thus assigned the same PBCB. In the following subsections we will refer to Π as the partitioning of the SOG.

4.3.2 Check Blocks (*ed*)

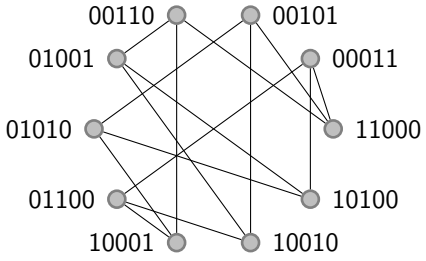
After all PBCB have been generated, the function *ed* is used to calculate the actual check blocks $c_0, ..., c_{m-1} \in \mathbb{F}_{2^j}$. As shown in the previous section the construction of f_c ensures that every transmission fault possible under the fault assumption is "visible" in the PBCB. Note, however, that this reasoning is only valid, if we assume that the decoder at the receiver only has to deal with valid code words. Thus, for now we restrict the discussion to faults that change an incomplete

d	$f_c(d)$
000	00
001	00
010	01
011	01
100	10
101	10
110	11
111	11

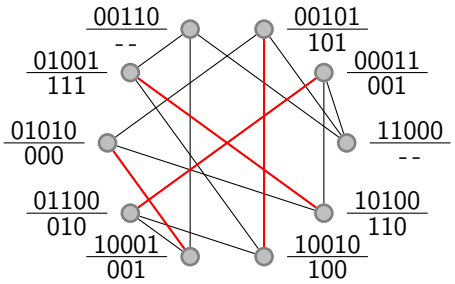
(a) $f_c : \mathbb{F}_8 \mapsto \mathbb{F}_4$



(b) ED Graph



(c) 2-of-5 SOG ($f = 1$)



(d) Isomorphic subgraph

Figure 4.5: Example: 2-of-5 code

intermediate bit pattern into a valid but incorrect code word and omit the case where invalid code words are presented to the decoder. For now we may simply assume that the receiver is able to detect invalid code words.

Let us first consider the case where $f = 1$. In this scenario a fault affects exactly one DI data or check block. If a DI check block is changed obviously also the encoded data changes. If the fault hits a DI data block, we can be sure that the PBCB which are calculated from the decoded DI data block will be different from the ones originally generated for this block. Hence, to be able to detect a transmission fault, the used error detecting code must be able to detect exactly one block error. A single block error can be detected by a simple j-ary parity code over the vector $(f_c(d_0),...,f_c(d_{n-1}))$, which will generate exactly one check block $c_0 \in \mathbb{F}_{2^j}$. We can conclude that for the single fault assumption one check block is sufficient to secure the transmission, regardless of the number of data blocks.

In the general case, where $f \geq 1$, at most f (data or check) blocks can be affected by a transmission fault. Consequently a j-ary linear systematic error detecting code with an error detecting capability of at least f symbols (i.e. blocks) has to be used. As discussed in Section 2.1.2 linear codes can always be specified in form of a generator matrix. Equation 4.9 shows the general form of the $n+m \times n$ generator matrix used to calculate the check blocks. Since the code is also systematic, the matrix is decomposeable into an identity matrix I_n and a part which actually generates the check blocks, denoted by A.

$$M_{ed} = \begin{pmatrix} 1 & 0 & \dots & 0 & a_{0,0} & \dots & a_{0,m-1} \\ 0 & 1 & & 0 & a_{1,0} & \dots & a_{1,m-1} \\ \vdots & & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 & a_{n-1,0} & \dots & a_{n-1,m-1} \end{pmatrix} = (I_n|\mathbf{A}) \tag{4.9}$$

Hence *ed* can now be defined as

$$ed(\mathbf{v}) = \mathbf{v} * \mathbf{A} \,. \tag{4.10}$$

By inserting the PBCB vector $(f_c(d_0), ..., f_c(d_{n-1}))$ into Equation 4.10, the check blocks are finally given by

$$(c_0, ..., c_{m-1}) = (f_c(d_0), ..., f_c(d_{n-1})) * \mathbf{A} \,. \tag{4.11}$$

4.4 Decoding

Figure 4.6 shows an overview of the decoding process. $D_0, ..., D_{n-1}$ and $C_0, ..., C_{m-1}$ denote the DI data and check blocks as received over the DI communication link (i.e. the data captured by the receiver's input register). First DI decoding is performed on these blocks using the functions $di_d^{-1} : \mathbb{F}_{2^{k'}} \mapsto \mathbb{F}_{2^k}$ and $di_c^{-1} : \mathbb{F}_{2^{j'}} \mapsto \mathbb{F}_{2^j}$ respectively. As result of this step, we again obtain the binary representation of the transmitted data blocks $d'_0, ..., d'_{n-1}$ and check blocks $c'_0, ..., c'_{m-1}$. However, due to transmission faults one or more of these blocks may be incorrect. Based on the decoded data blocks, all check information is now recalculated by applying the functions f_c and *ed*, resulting in the check blocks $c''_0, ..., c''_{m-1}$. Finally, these check blocks are compared to the ones received over the DI bus. If they match, no fault occurred during the transmission. Hence the received data is correct and the acknowledgment signal can be asserted. If, however, a mismatch is detected, the data has been corrupted and the receiver has to resample the DI data and start the whole decoding process over.

The scheme described above works fine if the receiver is confronted with valid, but possibly incorrect, code words. However, we still need to analyze how invalid and unused (IaU) code words are handled. Without proper handling of these cases, transmission faults might slip through the error detection mechanism with severe consequences on the overall system.

To illustrate this potential problem consider a DI communication link implementing the discussed coding scheme under the fault assumption of $f = 1$. Further assume that the data blocks are encoded using a 3-of-6 code. This code comprises 20 symbols and is hence able to carry four bits of information, which in turn means that four symbols are unused. It can be shown that the SOG of the 3-of-6 code can be partitioned into $4 \times K_4$. Hence the PBCB need to be two bits wide.

Assume that, due to a transmission fault, one of the four unused code words c_u is received. Now the question arises to which data word this (erroneous) code word should be mapped such that the transmission fault is visible in the PBCB $f_c(di_d^{-1}(c_u))$ generated from it. To answer this question we first need to investigate what a problematic mapping looks like, such that it can be

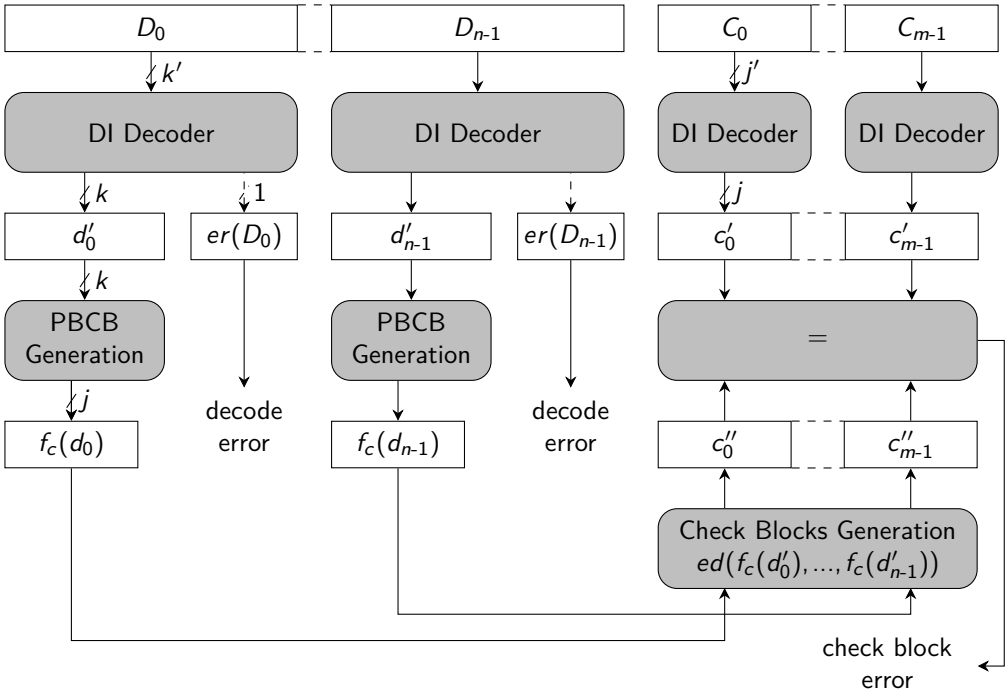


Figure 4.6: Decoding overview

avoided. Recall that each clique in the SOG is assigned a unique PBCB pattern. Hence code words (and ultimately data words), which are part of the same clique are indistinguishable based on their PBCB. However, since (based on the fault assumption) we know that such code words cannot be confused anyway, this property does not pose a problem. If c_u can be confused with more than one code word of a particular clique P, it may not be assigned the same PBCB as P, i.e. it may not be mapped to one of the data words to which the other code words in P are mapped to. Unfortunately, it can be shown, that such a mapping does not exist for the 3-of-6 code. Hence, in this case some other means are necessary to deal with this situation.

A similar problem arises if the receiver has to process invalid code words. For our example this would be vectors with two or four bits set to one. As shown by this example the special treatment of IaU code words is crucial for the proposed coding scheme to work correctly in all cases. We have thus developed strategies, that allow to cope with these situations.

4.4.1 Omit PBCB Generation

The simplest way to deal with the problem of invalid and unused code words is to not perform the step of calculating the PBCB. In this case $f_c(d) = d$ is simply given by the identity function. The error detecting code is then calculated solely over the data blocks themselves. Hence a k-ary error detecting code is required ($j = k$). This means that every transmission fault directly alters a symbol which is protected by the error detecting code. Note that the fact that a single transmission

fault can, in the worst case, cause a k-bit error in the encoded data is not important here, because this situation is still considered as a single symbol (i.e. block) error. The DI decoder can now map invalid and unused code words to arbitrary data words. If the invalid or unused code word is decoded into a data word which is different from the data word that the transmitter intended to send, the error detecting code will reveal the transmission fault. If, on the other hand, the decoder by chance outputs the "right" data word, the receiver won't even notice the error.

The obvious advantage of this approach is that the mapping of data words to code words is completely uncritical. This may be beneficial for the design of the encoder and decoder circuits, since it allows to select a mapping, which is well suited for a hardware implementation (small area or delay). However, since we don't try to leverage the inherent fault resilience of the used DI code, this approach leads to a more complex *ed* function which also increases the number of check rails required.

Notice that the that Dual Rail Parity Code [32], is a special case of this encoding scheme.

4.4.2 Error Detection

This approach is based on the fact that the receiver exactly knows which code words have to be expected at its input. Hence, if an invalid or unused code word arrives, the DI decoder is able to detect it and can assert an error signal. This approach is also depicted in Figure 4.6. Besides the actual binary data extracted from the DI data block (di_d^{-1}), the decoder now also generates an error signal given by the function $er : \mathbb{F}_{2^{k'}} \mapsto \mathbb{F}_2$. This function basically maps all IaU code words, which can, based on the fault assumption, be expected to appear on the decoder's input to 1. Formally er can be defined as follows.

$$er_{decode}(x) = \begin{cases} 1 & \text{if } x \in Z \\ 0 & \text{if } x \in C' \\ \perp & \text{otherwise} \end{cases} \tag{4.12}$$

Here the set Z is given by $U \cup I$. Note, however that it is generally not necessary to generate the error signal for all IaU code words. Hence, in practice the size of Z can be reduced drastically. This is discussed in more details in the next section. For vectors which neither lie in Z nor in C' the behavior of the function is undefined (indicated by $\perp$). Since, due to the fault assumption, these vectors can never appear at the decoder's input, this information can be used to optimize the decoder circuit in a real hardware implementation (don't care output).

For the case of m-of-n codes with a single fault assumption ($f = 1$) invalid code words can easily be identified by a parity function. Since, in this case, all valid code words have the same constant Hamming weight (and thus the same parity) an additional or missing one always changes the parity of the code word.

4.4.3 Decoder Analysis

This approach tries to map IaU code words to suitable data words. For each of these code words the DI decoder at the receiver basically "guesses" a possible data word, which might have been sent by the transmitter. However, special care must be taken, because if the guess is not correct,

it must be guaranteed that the error introduced by it is visible to the error detection mechanism. As we have seen with the example of the 3-of-6 code this is not always possible. In such cases a different approach has to be applied.

Let $G = (C', E)$ denote the SOG constructed from the code C'. By applying the procedures from Section 4.3.1 a partitioning $\Pi = \{P_0, ..., P_x\}$ of G into a set of cliques is obtained ($\cup_{P \in \Pi} P = C'$). Hence the members of Π are sets of code words. Recall that every code word in such a set is assigned the same PBCB.

In order to find a suitable mapping for all IaU code words, we propose the following procedure: For every IaU code word x construct a set Π'_x by intersecting every element $P \in \Pi$ with the set $\Theta(x)$ or $N_V(x, f)$ depending on whether x lies in I or U. Formally this is defined by Equation 4.13.

$$\Pi'_x = \{P \cap N(x) | P \in \Pi\}, \text{ where } N(x) = \begin{cases} \Theta(x) & \text{if } x \in I \\ N_V^-(x, f) & \text{if } x \in C \end{cases} \tag{4.13}$$

Note that $N(x)$ returns the set of candidate code words that could have been sent by the receiver if x was received. Based on the size of the members $P' = P \cap N(x)$ of Π'_x the following case distinction is possible.

i. $|P'| = 0$:
In this case, none of the code words $N(x)$ which can be confused with x are part of the clique represented by P. This means that, if x is mapped to any of the data words where members of P are mapped to, the PBCB generated from $di_D^{-1}(x)$ will certainly be different from the PBCB generated by the transmitter from the original, i.e. fault-free, version of x. Thus the fault will be detected.

ii. $|P'| = 1$:
Here there is exactly one code word $y \in P'$ that is a candidate for a code word originally sent by the transmitter. If we now map x to the same data word as y, there are basically two cases that need to be considered. If y indeed was the code word transmitted, then the performed mapping to a data word is correct and we actually performed some kind of forward error correction. However if some other code word $z \in N(x) \setminus \{y\}$ is the fault-free version of x, the error detection still works correctly, since the PBCB generated from $di_D^{-1}(x) = di_D^{-1}(y)$ will again be different from the ones originally generated by the transmitter.

iii. $|P'| > 1$:
In the last case a safe mapping to one of the elements of P is not possible, because there exist at least two code words y and z with the same PBCB ($f_c(di_D^{-1}(y)) = f_c(di_D^{-1}(z))$). This means that if x would for example be mapped to the same data word as y a fault that actually changed z to x would not be detectable.

Obviously the goal must be to find at least one member of Π'_x which falls into the first two categories, since this guarantees that a fault detection (and sometimes correction) is always possible.

If such a mapping can be found for all elements of $U \cup I$, then a DI decoder that safely deals with all possible code words (valid and invalid) can be constructed. Of course it is also possible

to use the presented procedure to analyze a particular hardware implementation of a decoder circuit. In this scenario, the mapping of IaU code words is already fixed and the analysis is simply used to find the problematic cases where the mapping is ambiguous and error detection would fail. Hence, we now have a method to precisely determine the elements of the set Z used in the definition of the decode error detection function of Equation 4.12. In contrast to the general IaU detection logic discussed in the previous section this circuit only needs to cope with a subset of all IaU code words and might hence be cheaper to implement.

To visualize the presented process consider the following example. Figure 4.7 shows a possible partitioning of the 3-of-6 code. For the sake of clarity the code words are labeled $c_0,...,c_{15}$. The code words in the middle are three representative examples of IaU code words that need to be analyzed for a fault assumption of $f = 1$. The topmost code word (011010) is one of the four unused code words, while the other two (100001 and 110011) are invalid ones. The edges indicate whether a valid (and used) code word is member of the set $N(x)$, i.e. a possible candidate for the originally sent code word if x was received. Using this information the sets Π'_x can be constructed, which are shown in Equations 4.14-4.16.

$$\Pi'_{011010} = \{\{c_1, c_2\}, \{c_4, c_6\}, \{c_8, c_{11}\}, \{c_{13}, c_{14}, c_{15}\}\} \tag{4.14}$$
$$\Pi'_{100001} = \{\{\varnothing\}, \{c_5\}, \{\varnothing\}, \{c_{13}\}\} \tag{4.15}$$
$$\Pi'_{110011} = \{\{c_2\}, \{c_4\}, \{c_8\}, \{c_{13}\}\} \tag{4.16}$$

For the two invalid code words safe mappings to data words can easily be found. The code word 100001 can, for example, be mapped to the corresponding data words of every element of the partitions P_0 and P_2 as well as the code words c_5 and c_{13}. However, for the unused code word $c_u = 011010$ a safe mapping is not possible, since every element of the set Π'_{011010} contains more than one element. If c_u would, for example, be mapped to the same data word as c_1, the receiver could not detect a fault that changed (an intermidiate pattern of) c_2 to c_u, since the PBCB generated from c_1 and c_2 are the same.

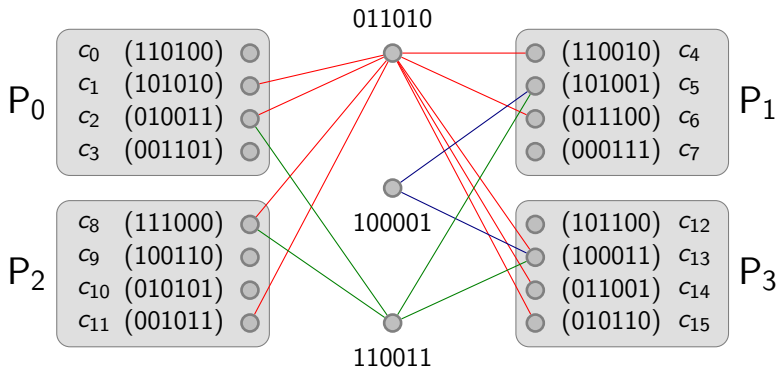


Figure 4.7: Code word analysis example

The question that remains to be answered is what are the benefits of this solution to the previous one discussed in Section 4.4.2. A slight advantage over error detection is that in the case

of a transient fault there is a chance that the assigned data word is the right one and no re-sampling is required. However, this is only a very marginal benefit and should thus not be assigned too much weight. If we compare both approaches based on their implementation (hardware) costs, it strongly depends on the actual case, whether the enhanced decoder is cheaper to implement than the error detection circuit. However, if the decoder is constructed using a lookup table (which may possibly be the case in an FPGA implementation) the extension of the decoder has zero overhead and is therefore the preferable solution because no additional logic for error detection is needed.

4.4.4 Forward Error Correction

This approach is actually very similar to the previous one. Here, we assume that the used DI code, has by itself certain error correcting capabilities. This is, for example, the case for Zero-Sum codes. For this code the decoder can automatically correct invalid code words. However, it must be ensured that if an invalid code word has more than one potential candidate for its fault-free version, that these candidates are from different partitions of the SOG. If they originate from the same partition, they are also assigned the same PBCB which makes them indistinguishable for the error detection logic. Note that this basically corresponds to case $|P'| > 1$ of the previous section, which means that a safe mapping to either of these code words is not possible.

4.5 Decoding with Error Correction

This section describes how the proposed coding scheme can be modified to support forward error correction (FEC). A receiver using FEC can immediately acknowledge the reception of a DI code word to the transmitter, even if it contains errors. This can obviously speed up the transmission, because there is no need to stall the receiver until the faults vanish.

Lets assume that we have a transmitter that implements a variant of the proposed encoding scheme. In this scenario, the PBCB are generated by the identity function ($f_c(x) = x$), which means that the check and data blocks will have the same size ($j = k$). For the function *ed* a k-ary Hamming code is used. If normal error detection is performed by the receiver, it is thus able to detect up to two erroneous (data or check) blocks. Hence we are able to detect and tolerate $f = 2$ faults.

However, as discussed in Section 2.1, a Hamming code is not only able to detect up to two symbol errors but can also correct one. If we thus restrict the fault assumption to $f = 1$ the receiver would be able to correct one erroneous symbol (i.e. block). Note that this property is independent of the used DI code.

This observation is very trivial. A k-ary Hamming code is used to correct one symbol of a k-ary block code. The interesting question is, if this still works for other non-trivial functions f_c. Or in other words, is it possible to exploit the fault resilience of the DI code in order to reduce the amount of check information transmitted, but be still able to perform FEC.

A very basic constraint that can immediately be established is that for FEC to work in the presence of f faults the function *ed* must be able to correct f symbols. However, if f_c is different from the identify function ($j < k$) we are only able to reconstruct the correct PBCB of each data

block. This information is generally not enough to assign a data word to an erroneous code word c_e, because there are cases in which more than one data words are assigned the same PBCB. However, with the information about the corrected PBCB z it is possible to cut down the set of candidates for the fault free version of c_e to the elements of the partition identified by z. In the following we will refer to this partition as the source partition of a code word. Consequently, to be able to perform error correction, we also have to take c_e into account. Hence it must be ensured that the combination of z and erroneous code word c_e is unique and can therefore be unambiguously mapped to a data word using some appropriate function ec.

We will now examine the criterion which must be fulfilled such that FEC is possible for a particular code and SOG partitioning Π and how the function ec can be defined. Basically we perform the same procedure as described in Section 4.4.3 with the slight difference that now all code words (valid and invalid), i.e. not only the IaU ones, must be analyzed.

For every code word x in $C \cup I$ (recall that $C = C' \cup U$) the set $N(x) \subset C'$, i.e. the set of code words which constitute candidates for the fault free version of x is calculated. Then the set $\Pi'(x)$ (Equation 4.13) are constructed and its members are examined. Again there are three different cases of how the elements $P' \in \Pi'$ (which are themselves sets) can look like. For the following case distinction P denotes the partition used to generate P' ($P' = P \cap N(x)$).

i. $|P'| = 0$:
   In this case there is no code word in P that can be confused with x, hence P' is empty. It is not possibly that a fault changes a code word in P to x, hence no error correction is required in this case.

ii. $|P'| = 1$:
   Here, the set P contains exactly one code word, which may have been the fault free version of x. Observe, that since we can identify the source partition of x based on the corrected PBCB there are two cases that need to be considered. If P indeed was the source partition of x, we can conclude that the fault free version of x is the only element left in P'. If on the other hand, P is not the not the source partition then all code words in P can be excluded from the set of candidates for x.

iii. $|P'| > 1$:
   In this case error correction is not possible, because there are more than one elements in P, which may have been the fault free version of x. If the corrected PBCB identify P as the source partition of x then there is no way to decide which element of P' is the correct one.

Hence, we conclude that in order for FEC to be possible, the last case where $|P'| > 1$ may never happen.

$$\forall x \in C \cup I \forall P' \in \Pi'(x) : |P'| \leq 1 \Rightarrow \text{FEC possible} \tag{4.17}$$

Furthermore, we can now define the function ec as shown in Equation 4.18.

$$ec(x,z) = N(x) \cap \{c \in C' | f_c(c) = z\} \tag{4.18}$$

The function basically calculates the intersection between the set of possible candidates for the fault free version of x and the set of code words contained in the partition identified by the corrected PBCB z. The result of this operation is again a set. However, for all possible values for x and z this set must only contain a single element, the fault free version of x. If there is one case, where $|ec(x,z)| > 1$ FEC is not possible. Hence, an alternative form of the condition shown in Equation 4.17 can be written as follows.

$$\forall x \in C \cup I \forall z \in \mathbb{F}_{2^j} |ec(x,z)| \leq 1 \Rightarrow \text{FEC possible} \tag{4.19}$$

CHAPTER 5

Link Architecture

This chapter presents circuits that can be used to implement the coding scheme proposed in Chapter 4 and discusses the advantages and drawbacks associated with the different approaches. The different transmitter and receiver circuits discussed in Sections 5.1 and 5.2 can be used in arbitrary combinations. To close the chapter, Section 5.3 presents a brief analysis of metastability hazards in the proposed circuits. Metastability mainly poses a thread to the receiver since it could cause a malfunction of the circuit that is not detectable by the coding scheme.

5.1 Transmitter

The transmitter circuits mainly differ in the approach that is used to generate the null phase of the 4-phase protocol. Note that there are essentially two ways to accomplish this. Either an array of AND gates, or an output register, composed of resetable D flip-flops or D latches, can be used to force all output rails to zero in the null phase.

5.1.1 AND-Masking Transmitter

Figure 5.1 shows a high-level overview of the AND gate based transmitter circuit. The binary input data of the BD interface is fed into the check bits generator, which basically applies the functions f_c and *ed* as described in Section 4.3. The output of this block, the check bits, is then encoded alongside the actual input data with some suitable DI code. Note that it is not required for the DI encoder or the check bits generator to be QDI circuits, since they both operate in the BD domain. This is also true for every other circuit presented in this section. However, it must be guaranteed that the delay Δ_{req} is long enough such that the outputs of the encoder are stable and valid when the *req* signal reaches the controller. The spacer insertion block consists of an array of 2-input AND gates, one for each output rail of the encoder. The other input of each AND gate is driven by the *en* output of the controller. This allows the controller to force all output rails of the transmitter to zero in order to generate the null phase. The spacer insertion block can be viewed as the border between the BD and the DI protocol. In the beginning the *en* output of the controller

is zero, thus the AND gates are masking the output of the encoder. The encoder can now make arbitrary transitions until it finally stabilizes. As soon as the controller receives a rising edge on the *req* input it asserts the *en* output to activate the AND gates. The DI data emerges at the output of the transmitter and eventually causes the receiver to assert the ack_{in} signal. The controller now asserts the ack_{out} signal, while it simultaneously deasserts the *en* output to switch the AND gates back into the masking state again, effectively generating the null phase on the output. The delay element Δ_{ack} in the acknowledgment path must ensure that all AND gates are definitely in the masking state, before the acknowledgment causes the preceding logic to change, i.e. invalidate, the input data to the transmitter. Eventually, the null phase is acknowledged by the succeeding logic stage (the receiver) by deassertion of the ack_{in} signal. Meanwhile, the handshaking protocol on the input side of the transmitter is completed (deassertion of *req* and ack_{out}) and the circuit is finally ready to process the next input request.

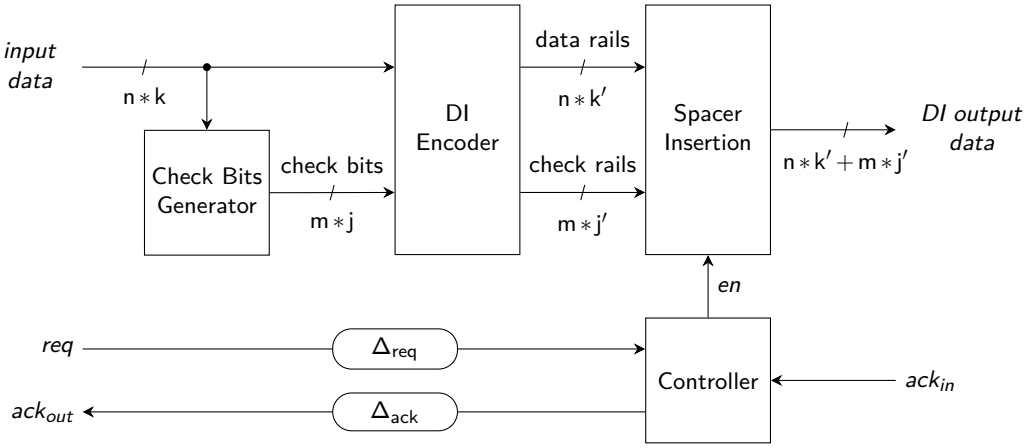
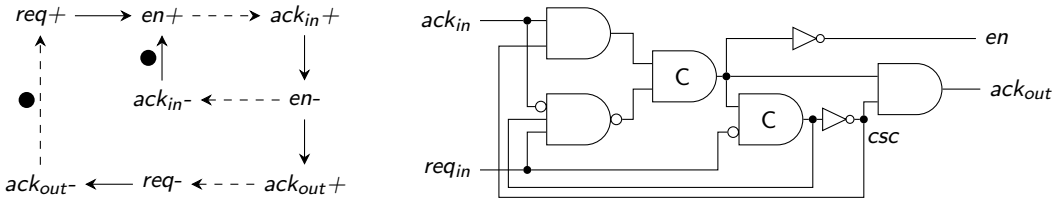


Figure 5.1: AND-masking transmitter

The behavior of the controller circuit is described by the STG in Figure 5.2a. Using the tool Petrify a (speed-independent) circuit can be generated from this specification. The result of this process is shown in Figure 5.2b. The initial, i.e. reset, values for both C gates is one. Note that this STG contains a Complete State Coding conflict which is automatically resolved by Petrify by introducing a new state internal variable *csc*. For most of the controller circuits in the following sections we will only show the STG specifications, since the process of generating circuits out of them is completely automated.

A slight optimization to the transmitter's operation speed can be achieved by using asymmetric delay elements for Δ_{req} and Δ_{ack}. For both signals it is sufficient to only delay the rising edges. The falling edges do not need to be delayed because they do not indicate the validity of data. These transitions are only required by the 4-phase protocol, in order to reset all handshaking signals to their initial value (zero). Note that for a two phase implementation of the BD interface, this optimization would not be required, because there the handshaking signals don't return to zero after each data transmission.



(a) Controller STG

(b) Controller circuit

Figure 5.2: AND-masking transmitter controller

5.1.2 D Flip-flop Transmitter

This version of the transmitter circuit is based on the approach presented in [39], which could (in its original form) also be used with the presented coding scheme. However, we implemented slight optimizations that allow for considerable hardware savings and faster operation speed. In the following, we will now briefly present the original approach and then discuss its drawbacks and show how to overcome them.

Figure 5.3 shows the adaptation of the transmitter circuit of [39] for the proposed coding scheme. The delay element Δ_{req} ensures that the DI data (or the spacer) is stable and valid at the input of the output register before the *req* signal reaches the controller. The gate-level implementation of the controller is shown in Figure 5.4. As soon as it receives a rising edge on the *req* signal the output of the C gate is set to one. This transition generates a pulse at the trg output which is used to clock the output register and capture the DI data. The output of the C gate is also used as ack_{out} signal to notify the preceding logic that the input data has been consumed. The acknowledgment eventually causes the *req* input to be deasserted. This event is propagated to the DI encoder through its *phase* input, which reacts with resetting all its outputs to zero. If *req* is zero and ack_{in} is one (acknowledgment from the DI channel) the controller generates another pulse on the trg signal to capture the spacer into the output register. The spacer is eventually acknowledged by a falling transition on the ack_{in} signal, which makes the controller ready to receive the next input request.

The observation that led to the improved circuit was made by investigating the start up/reset behavior of the original circuit. To ensure correct operation after start up, it must be ensured that the output register is reset to zero. A different initial value would violate the 4-phase protocol and could potentially lead to a malfunction of the circuit. Thus the D flip-flops forming the output register must be equipped with (asynchronous) reset inputs. This raises the question if this reset logic can also be used to generate the null phase of the 4-phase protocol. For this purpose we propose the circuit shown in Figure 5.5.

The big advantage that comes with this modified circuit is that the DI encoder no longer needs the *phase* input, because the null phase is now generated by the register reset signal *rst*. This saves a considerable amount of hardware, because in order to enable the encoder to force all of its output signals to zero (regardless of the input), a mechanism similar to the spacer insertion block discussed in the previous section would be required. Apart from the hardware savings, the depth

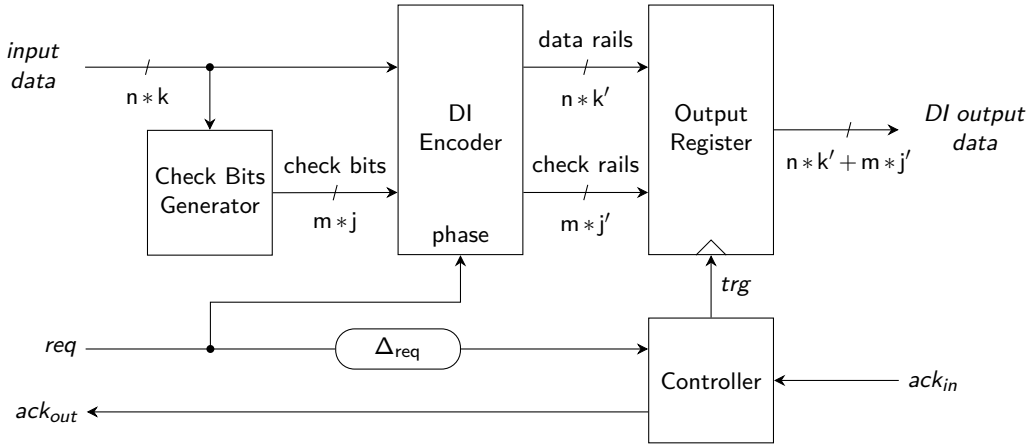


Figure 5.3: D flip-flop based transmitter [39]

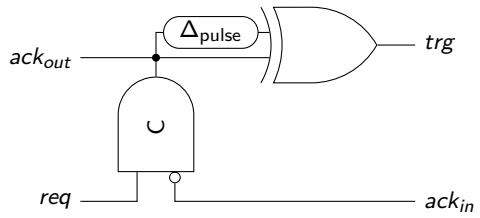


Figure 5.4: D flip-flop based transmitter controller circuit [39]

of the decoder circuit is also reduced, which has a positive effect on the circuit's delay and thus the operation speed. Like with the previous approach the delay element Δ_{req} can be implemented asymmetrically, to reduce the time required for resetting the handshaking signals.

Depending on how much parallelism is allowed between the execution of the two handshaking protocols on the BD input and the DI output channel, the controller circuit must be implemented accordingly. In the following we will present two possible versions.

Simple Controller (Low Parallelism) The simple version of the controller circuit is shown in Figure 5.6. Consisting of just a single C gate and a pulse generator, it is very similar to the one proposed by [39]. It can be divided into two main blocks, the acknowledgment and the reset generator. The former generates the ack_{out} signal for the BD channel, whose rising edge is also used to trigger the D flip-flops. The reset generator is triggered by the falling edge at the output of the C gate and produces a pulse, which resets the output register to generate the null phase. The delay element Δ_{rstpls} must be chosen long enough to fulfill the requirements of the output register, but must also be short enough such that the reset is completed before the next rising edge at the output of the C gate (i.e. ack_{out}). The main drawback of this circuit is that the two handshaking protocols on the BD input and DI output channel are interlocked. This issue becomes apparent, if the STG describing the behavior of the acknowledgment generator is investigated. As can be

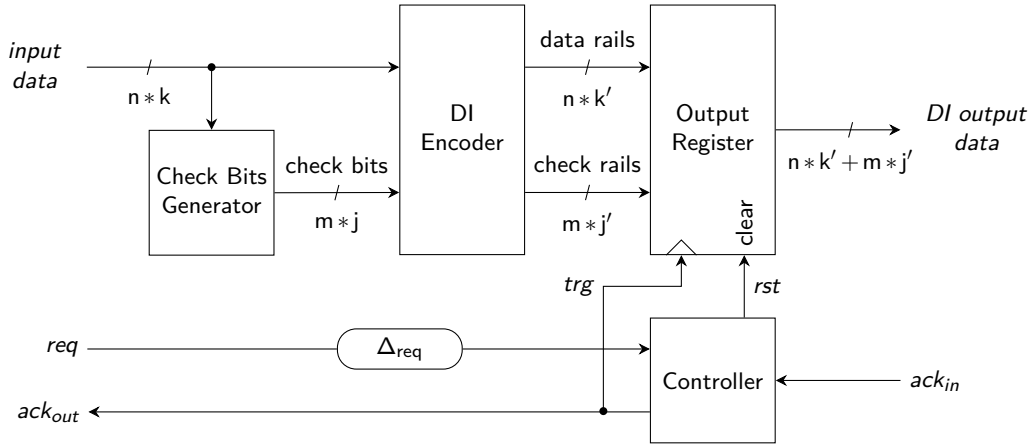
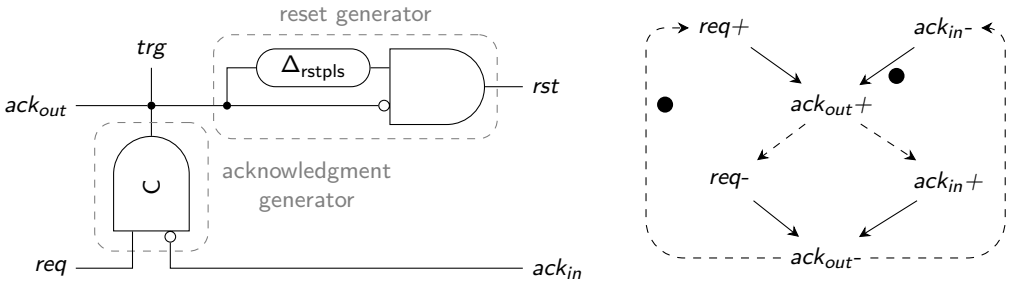


Figure 5.5: Transmitter using D flip-flops with asynchronous reset



(a) Controller circuit

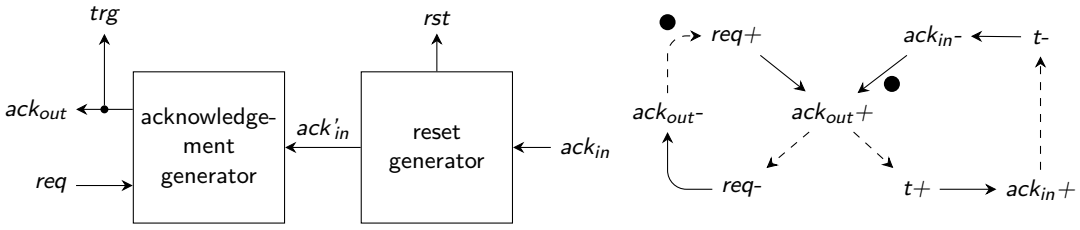
(b) Acknowledgment generator STG

Figure 5.6: Simple controller circuit

seen in Figure 5.6b the $ack_{out}-$ transition is beside $req-$ also dependent on $ack_{in}+$. Furthermore $ack_{in}-$ is dependent on $ack_{out}-$. Obviously these preconditions are necessary for the controller to work correctly. However, they are actually a result of the fact that the *rst* signal is generated from ack_{out}. If it would be possible to generate the *rst* signal solely based on ack_{in} these dependencies and the tight coupling between the handshaking protocols could be eliminated.

Advanced Controller (High Parallelism) The advanced controller, as shown in Figure 5.7a, is able to overcome the tight coupling disadvantage of the simple controller. It again consists of an acknowledgment and a reset generator. However, in this circuit the *rst* signal is generated from ack_{in}. The behavior of the acknowledgment generator is described by the STG in Figure 5.7b. As can be seen there is no coupling between the handshaking protocols. The only time where the protocols are "synchronized" is when the rising edge of the ack_{out} signal is generated. Since this edge is used to capture new data into the output register, it must be ensured that both the input and the output channel are ready for this event.

The actual gate-level circuit implementation of the acknowledgment generator can again be generated by the tool Petrify. However, in order for Petrify to process the STG, an additional (internal temporary) state variable t must be introduced. Notice that the rising edge on the ack_{in} signal essentially generates the null phase which in turn leads to a falling edge on ack_{in}. However, the sequence $ack'_{in}- \rightarrow ack'_{in}+$ would be problematic because there are two subsequent transitions in the STG that must be fulfilled by the environment and not by the actual controller that should be described. This causes a conflict that cannot be automatically resolved by the tool. Hence, the state variable t is inserted into the affected part of the STG, changing the dependency relation to $ack'_{in}+ \rightarrow t+ \rightarrow ack'_{in}- \rightarrow t-$. The internal variable essentially indicates whether the (rising) acknowledgment transition has already happened.



(a) Controller circuit (b) Acknowledgment generator STG

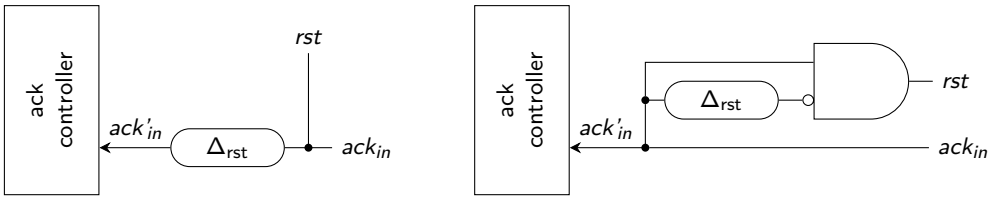
Figure 5.7: Advanced controller circuit

As shown in Figure 5.8 there are two possibilities of how to implement the reset generator. Version A simply uses the acknowledgment signal itself as reset signal for the output register. Hence, the output register is kept in the reset state until ack_{in} goes low again. The delay element Δ_{rst} ensures that there is enough time between the end of the reset phase (deassertion of ack_{in}) and the next rising edge of ack_{out} caused by a (potentially) pending input request.

Version B only reacts to the rising edges of the acknowledgment signal and uses a pulse generator to generate the reset signal. This behavior makes it very similar to the reset generator of the simple controller. Hence, the same constraints on the delay element Δ_{rstpls}, i.e. the reset pulse width, apply here. Regarding performance version B has a slight advantage over the alternative because in version A the cycle time is extended by Δ_{rst} for cases where new data is already pending on the input side of the transmitter.

5.1.3 D-Latch Transmitter

This version of the transmitter is very similar to the previous one, but uses pulsed D latches [1] instead of D flip-flops to implement the output register. For this to work, obviously the register trigger signal trg generated by the controller circuits must be converted to a pulse. This modification has the advantage that the hardware costs for the output register can be reduced significantly. On the downside, the complexity of the controller is increased (additional pulse generator) and a few new timing constraints are introduced.



(a) Version A (b) Version B

Figure 5.8: Reset generator circuit variants

5.2 Receiver

For the receiver circuits, we only consider the error detecting and resampling scheme discussed in Sections 4.1 and 4.4, i.e. we won't present circuits that perform error correction as introduced in Section 4.5. First Sections 5.2.1 and 5.2.2 introduce the base version of the receiver circuit. Then Sections 5.2.3 and 5.2.4 discuss some modifications to this base version that potentially offer advantages in some use cases.

5.2.1 Base Receiver

The base version of the receiver circuit is shown in Figure 5.9. It is essentially an adaptation of the circuit presented in [39]. However, we modularized and extended the controller circuit to allow for higher parallelism between the DI input and BD output channels.

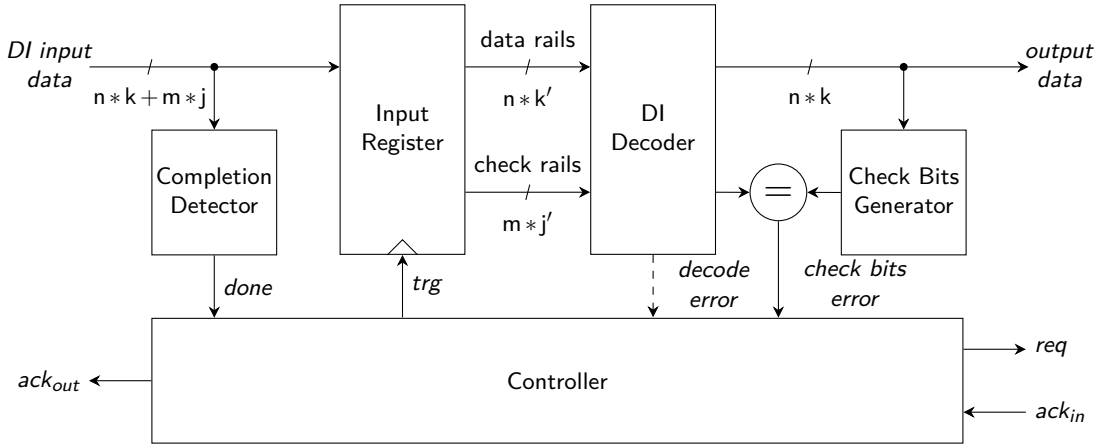


Figure 5.9: Base receiver

For this purpose, we have divided the controller circuit into two sub components (see Figure 5.10). The sampler generates the trigger signal *trg*, whose rising edge causes the input register

to capture the DI input data. It is further responsible to resample the input data until the error detection circuit confirms a fault free transmission. The protocol controller contains the actual logic that handles the handshaking protocols on the input and output channels.

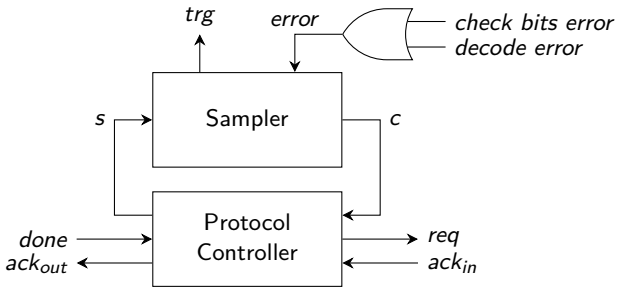
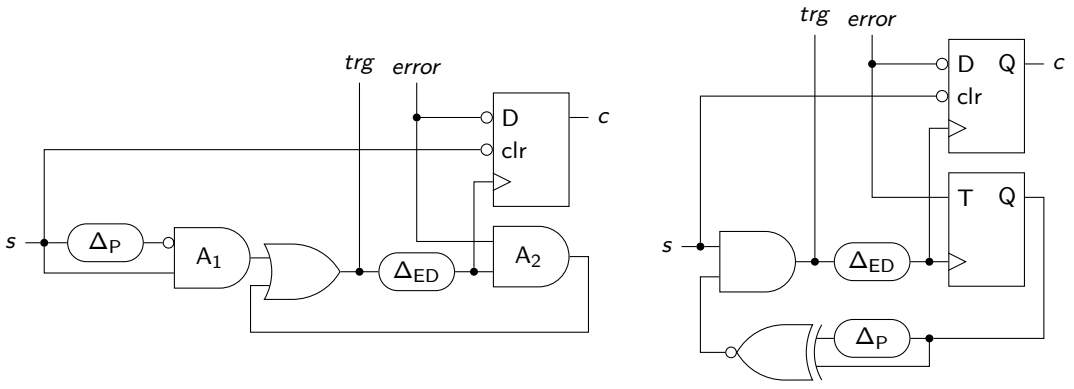


Figure 5.10: Base receiver controller

Sampler Figure 5.11 shows two possibilities how the sampler can be implemented. Both versions use the same interface. A logical one on the *s* input starts the sampling process. The circuit immediately produces a rising edge on the *trg* output to capture the DI input data. After the time period Δ_{ED} the decoding and error detecting circuit is ready and the *error* signal becomes valid. Depending on whether a transmission fault has been detected the circuit either resamples the input data (by producing another rising edge on the *trg* output) or in the case of a fault-free transmission asserts the *c* output to indicate completion. Deasserting the *s* input resets the circuit (i.e. clears the output *c*).



(a) Version A [39]

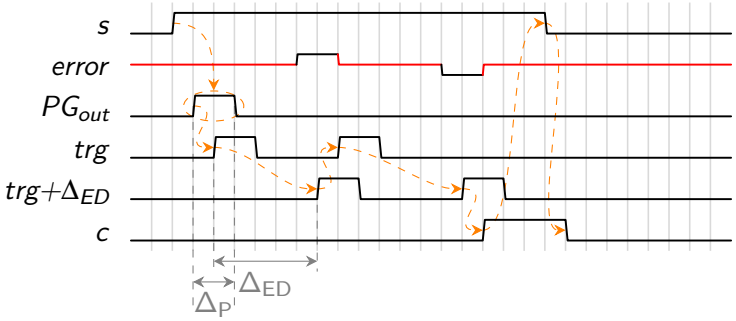
(b) Version B

Figure 5.11: Sampler circuit variants

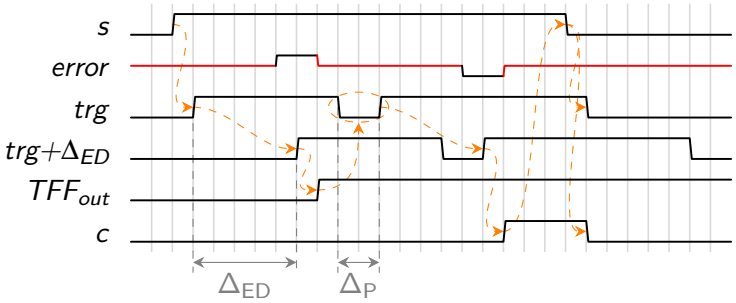
Version A of the sampler, shown in Figure 5.11a, is directly adopted from [39]. It works by initially generating a pulse of length Δ_P using a delay element and gate A_1. This pulse is then

used to produce the rising edge at the *trg* output. If an error is detected, the pulse is fed back into the circuit using the AND gate A_2 to generate another rising edge at the *trg* output.

Version B is an alternative implementation proposed by this work. It has the slight advantage that there is only a single gate between the *s* input and the *trg* output. Hence, the delay between the assertion of the *s* input and the initial rising edge on *trg* is slightly smaller than with version A. In case of an error, the feedback path of the circuit generates a pulse that is used by the AND gate to force the *trg* output to low (for the time period Δ_P) generating another rising edge on *trg*. Note that the difference in the behavior of the circuits is that in the idle phase, i.e. the time period where the circuits "wait" for the error detection logic to finish (Δ_{ED}) the *trg* output stays high in version B while it goes low in version A. Hence, in the case of a transmission fault, the performance of version B is a little worse, since the *trg* signal must first be reset to zero for the period Δ_P, before a new rising edge is generated. Figure 5.12 illustrates the difference in the signal trace of *trg*. The (red) lines between the high and low levels in the traces of the error signals indicate the period in which this signal is invalid and must not be sampled.



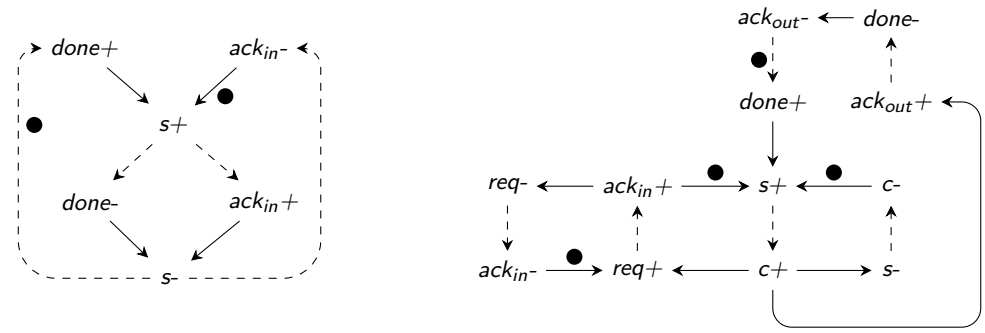
(a) Version A



(b) Version B

Figure 5.12: Sampler timing diagrams

Protocol controller Similar to the controller variants of the D flip-flop based transmitter, the protocol controller can also be implemented with varying degrees of parallelism between the handshaking protocols on the input and output channels.



(a) Simple STG ($c = req = ack_{out}$)

(b) Advanced STG

Figure 5.13: Protocol controller STGs

Figure 5.13 shows two possible STGs describing the controller's behavior. The low parallelism controller basically boils down to a simple C gate (see Figure 2.11b) with the inputs *done* and (not) ack_{in}. The output signals *req* and ack_{out} are directly connected to the *c* output of the sampler. Note that this is exactly the circuit proposed in [39]. It is easy to see that in this version the handshaking protocols are interlocked. There are situations where the DI protocol on the input side cannot proceed until the BD output has reached a certain state and vice versa.

To overcome these limitations, the advanced controller represented by the STG in Figure 5.13b has been developed. As can be seen in the Figure, there are no dependencies between the handshaking protocols except for the $s+$ transition which triggers the input register and captures new input data. This is the only point in time where all parallel operations of the circuit need to be "synchronized".

5.2.2 Base Receiver with D Latches

It is also possible to implement the receiver with D latches instead of D flip-flops for the input register. Since a D flip-flop is essentially composed of two latches this would effectively halve the implementation overhead for the input register. The *trg* output of the controller is hence connected to the enable input of the D latches instead of the clock input of the D flip-flops. However, depending on which sampler version of Figure 5.11 is used for the circuit, there are some things to consider.

Sampler A: Figure 5.12a shows that version A of the sampler generates pulses on the *trg* output that are Δ_P long. This signal can directly be used to switch the input latches to transparent mode and hence capture the current state of the DI input data bus. This means that the input flip-flops are basically replaced by pulsed latches[1] and no other modifications to the circuit are necessary.

Sampler B: Notice that this sampler keeps the trg output asserted (Figure 5.12b) during the sampling process. Only if resampling is required, a negative pulse is generated on the trg output causing it to go low for the duration Δ_P. This means that it is not possible to simply use the trg output of the sampler as enable signal for the latches. Doing so would inhibit the generation of a consistent snapshot of the DI input data, which is required for the error detection logic to work correctly.

There are essentially two possibilities to work around this problem. The first approach uses an additional pulse generator to transform the rising edge of the trg signal to a short pulse that then triggers the input latches.

For the other approach the trg signal is inverted, which means that the latch only holds its value when trg is asserted and is transparent otherwise. Hence resampling makes the latch transparent for Δ_P to capture the new state of the DI input data. However, for this to work it must be ensured that the s input of the sampler is not deasserted before the BD channel on the output of the receiver has acknowledged the reception of the transmitted data. Note that the deassertion of s in turn causes trg to go low which makes the input latches transparent again. Hence a premature deassertion of s could lead to the output data being invalidated before the succeeding logic had a chance to capture it. Notice that the simple controller (Figure 5.13a) guarantees this behavior. However, to use the advanced controller one additional edge from $ack_{in}+$ to $s-$ has to introduced.

5.2.3 Parallel Completion Detection and DI Decoding

Notice that the circuits discussed in the previous sections first capture the DI input data into their input registers and then perform the error detection process. This is consistent with the fault tolerant receiver model shown in Figure 4.2. The circuit shown in Figure 5.14 places the decoder in front of the input register. This arrangement has the advantage that while completion detection takes place the DI data can already be decoded, which can obviously increase the performance of the receiver. Moreover, since the DI data always needs more rails (i.e. bits) than the binary representation, the size of the input register can also be reduced. However, this increased level of parallelism also has some drawbacks.

Consider the fork in the DI data bus, that provides the input to both the decoder as well as the CD. For the base version it was only necessary to ensure that the path from the fork to the register inputs is faster than the delay added by the CD (i.e. when the CD asserts the *done* signal the data must be stable and valid at the input register). This timing requirement is not very difficult to fulfill, since the path to the registers does not contain any logic. However, if the placement of the input register and the decoder is reversed this constraint may become harder to satisfy.

In Section 4.4 we have seen that there are basically two methods of dealing with IaU code words in the proposed coding scheme. Either the decoder guarantees that these code words are never mapped to problematic data words or it has to provide an additional decode error signal for cases where this is not possible. For the normal (i.e. fault-free) operation of the circuit this means that it must never be the case that the error signal switches to zero (indicating validity of the output data) while the output is still invalid. By invalid we mean that the output of the decoder constitutes a problematic mapping of a code word to a data word as discussed in Section 4.4.3. However as soon as a fault strikes the error signal must be asserted before the output of the decoder can become invalid. This asymmetric behavior of the error signal for rising and falling

edges and the fact that every single possible case of a fault affecting the transmission would have to be analyzed makes a decoder circuit that fulfills this requirement very hard or even impossible to implement. Hence, we see that there is an inherent race condition to this circuit that is not present in the base version of the receiver.

A similar problem arises with the data outputs of the decoder. Here it must be guaranteed that during the transition from one output data word to the other (which was caused by a fault), the intermediate bit patterns never pose a problematic code word to data word mapping, which can be captured into the input register.

For these reasons this version of the receiver is only practical if no PBCB are used (Section 4.4.1). If the "full" check information is transmitted the actual effects of faults on encoded data blocks are irrelevant, because the receiver is able to detect errors at the block level. This is because multiple bit errors in a single block are only seen as a single error by the error detecting code. Moreover, it would even be possible to detect and tolerate faults that happen inside the decoder itself.

The controller can basically be implemented the same way as for the base version. Furthermore it is obviously also possible to use D latches for the input register.

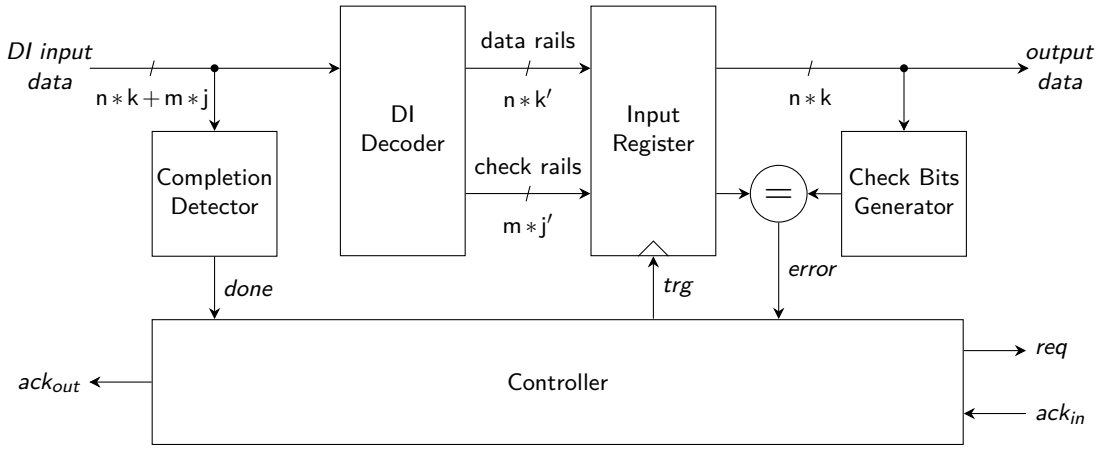


Figure 5.14: Receiver with parallel completion detection and DI decoding

5.2.4 Dual-Use Completion Detector

In this receiver variant the CD is placed after the input latches. Notice, that in contrast to the previous circuits, here it is necessary to use latches because flip-flops can not be switched into a transparent mode. This is however necessary, since the CD must be connected directly to the DI input rails.

The advantage that is gained from this arrangement is that parts of the CD can be reused for the tasks of decoding and, if necessary, detecting IaU code words. Sorting network based CDs, as discussed in [22], can for example easily be extended to detect invalid code words. Note, however,

that since (parts of) the CD are now also used in the BD domain, it can no longer be treated as a purely QDI circuit. This means that the static timing analysis that ultimately determines the delay Δ_{ED} must now also take paths through the CD into consideration.

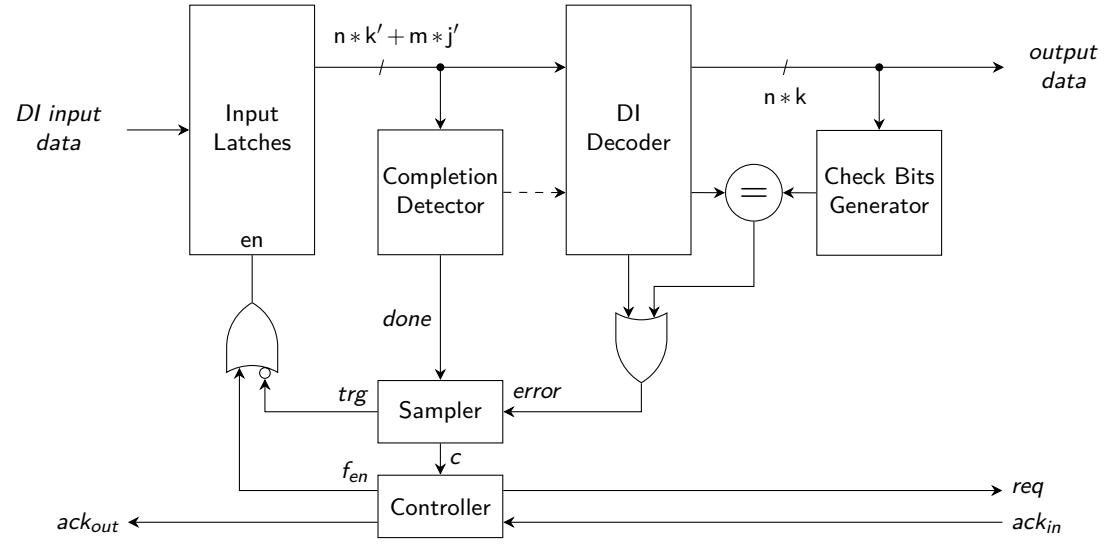


Figure 5.15: Dual-use completion detector receiver

Figure 5.16 shows the STG specifying the behavior of the controller for this receiver version. In the beginning all input rails as well as ack_{in} are zero. Hence the *done* output of the CD, which is connected to the *s* input of the sampler is deasserted. The *trg* output of the sampler as well as the f_{en} output of the controller are zero. This leads to a logical one at the *en* input of the input latch causing it to be transparent.

If a DI input data is applied to the CD the *done* output is triggered and the sampling process is started. It is important to note that here only version B of the sampler can be used. While the error detection process takes place, the sampler asserts the *trg* output to make the input latch opaque. If an error is detected a short (negative) pulse (Δ_P) on *trg* is used to recapture the input data. After successful completion the sampler asserts the *c* signal, which causes the controller to generate the request for the BD output channel and the acknowledgment for the DI input channel. As soon as the output channel acknowledges the transmitted data, the controller asserts f_{en} which forces the input latches to become transparent again. Eventually the spacer word appears at the CD, resetting the *done* signal to zero. Meanwhile the BD handshake can be completed. The falling edge of the *done* signal resets the *c* output of the sampler which in turn signals the controller that f_{en} can be deasserted, since now the logical zero at *trg* ensures that the latch stays transparent. Finally ack_{out} is deasserted as well to complete the handshake on the DI input channel.

To avoid glitches at the *en* input of the latches during the phase where *trg* and f_{en} are subsequently reset to zero, it must be guaranteed that the falling edge on *trg* reaches the OR gate before the one on f_{en}.

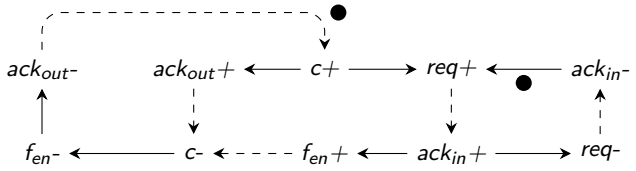


Figure 5.16: Controller STG

5.3 Metastability Concerns

Since we cannot make any assumptions about when a transient fault occurs, it is possible that the erroneous edge it produces violates a setup and hold window of a storage element [42].

This problem only affects the receiver, because we assumed that only the actual data rails can be affected by transient faults. The usual formula to calculate the Mean Time Between Upset (MTBU) for a single stage synchronizer (Figure 5.17) is shown in Equation 5.1.

$$MTBU = \frac{1}{f_{clk} \cdot \lambda_{dat} \cdot T_0} \cdot e^{\frac{t_r}{\tau}} \tag{5.1}$$

The parameters T_0 and τ are technology dependent and essentially characterize the used flip-flops. The time t_r specifies the resolution time, i.e. the time FF_{sync} has to resolve a possible metastable state and provide a correct input value to FF_{sys} without violating its setup and hold window. Hence t_r is simply given by the clock period T_{clk} minus the setup time of FF_{sys}, the (combinational) path delay Δ_p between both flip-flops and the nominal clock-to-output delay t_{CO} of FF_{sync} (Equation 5.2).

$$t_r = T_{clk} - t_{SU} - \Delta_p - t_{CO} \tag{5.2}$$

The frequency of the clock signal is denoted by f_{clk} while λ_{dat} refers to the rate of input data changes at $async_{in}$ (i.e. the average number of edges per second).

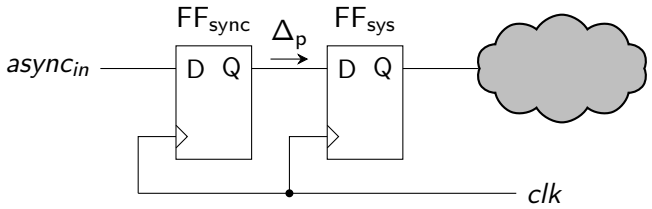


Figure 5.17: Single stage synchronization

In the receiver circuit FF_{sync} corresponds to the input register, while FF_{sys} would be the flip-flop sampling the error signal (inside the sampler). This is shown in Figure 5.18. Recall that

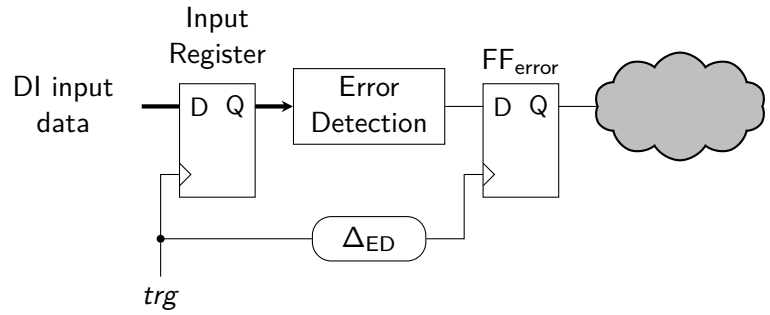


Figure 5.18: Problematic metastability path

the error signal is used to decide whether the input data is correct or if it has to be resampled. Hence, an upset on this signal may lead to incorrect data being forwarded to the BD interface.

However, since the circuits discussed in this chapter obviously don't operate in a synchronous way, Equation 5.1 can not be used directly, but needs to be adapted for this special purpose. First of all asynchronous circuits naturally don't have a clock signal. Hence, we replace f_{clk} by λ_{tx}, which denotes the average transmission rate, measured in handshaking cycles per second. This rate can basically be interpreted as the frequency with which the input register of the receiver is triggered. Since, an upset can only occur in the case of a fault, λ_{dat} can simply be replaced by λ_f, which denotes the average fault rate. Equation 5.3 shows the resulting expression.

$$MTBU_{rx} = \frac{1}{\lambda_{tx} \cdot \lambda_f \cdot T_0} \cdot e^{\frac{t_r}{\tau}} \tag{5.3}$$

The last question that needs to be answered is what value to use for the resolution time t_r. On the path between the input register and the error flip-flop resides the whole error detection logic, which essentially has the combinational delay Δ_{ED}. After this time (plus some safety margin) an edge is generated that captures the error signal. Hence, in our case, the resolution time is essentially zero, which simplifies the exponential term of Equation 5.3 to one. However, to improve the resilience against metastability the delay Δ_{ED} can be increased. On the downside this unfortunately also leads to a performance penalty on the receiver circuit.

Note that the possibility for a failure of the receiver due to metastability is the exception of the already rare event of a transient fault happening. Hence, the residual risk for the receiver circuit to fail can be classified as rather low. To substantiate this claim consider the following estimation. We assume a reasonable average transmission rate of $\lambda_{tx} = 10^8$ transmissions per second. For modern technologies the flip-flop parmeters T_0 and τ usually lie in the lower picosecond range [43]. For this example we assume $T_0 = 10ps$. The resolution time t_r will be specified by multiples of τ. In Equation 5.4 we calculate the residual metastability risk factor R. Multiplying the fault rate λ_f by this factor yields the MTBU of the receiver. Thus R basically specifies rate with which faults lead to upsets on the error signal due to metastability.

$$R(n) = \lambda_{tx} \cdot T_0 \cdot e^{-t} = 10^{-3} \cdot e^{-n} \tag{5.4}$$

Notice the exponential dependency of R on the resolution time. This shows that by degrading the performance of the receiver by approximately 7τ, which basically sums up to value less than 100 picoseconds, the residual risk can easily be improved by another factor 10^3.

CHAPTER 6

Results

This chapter shows examples on how the proposed coding scheme can be applied to m-of-n codes. First, Section 6.1 discusses theoretical results concerning the fault resilience of the analyzed codes. Section 6.2 presents two circuit level implementation examples and shows how they perform in terms of coding efficiency and area complexity. Finally, Section 6.3 shows a behavioral simulation to give a better understanding on how the circuits introduced in Chapter 5 operate.

6.1 Theoretical Results

Table 6.1 shows how the SOGs of the analyzed m-of-n codes can be partitioned into cliques under the fault assumption of $f = 1$. The presented partitionings are optimal in a sense that there does not exist a partitioning that will result in fewer PBCB. Consequently a selection of possible check codes is shown in the last column of the table. Recall that for the fault assumption $f = 1$ a single check (i.e. parity) block is sufficient. To characterize the quality of a code with respect to fault tolerance, we define the *inherent fault resilience* as shown in Equation 6.1.

$$F_f = 1 - \frac{j}{k} \tag{6.1}$$

This parameter relates the number k of bits a code word can contain to the number j of PBCB required to secure a transmission against f faults. It basically specifies how much the information in a data block can be reduced by the function f_c. Hence, F_f is always greater than or equal to zero and less than or equal to one. A zero value, like in the case of 1-of-N codes, indicates no fault resilience. If F_f is one, the code is by itself able to tolerate f faults and no check blocks are required. Consider for example the 3-of-6 and 2-of-7 codes. While both codes allow the encoding of four bits of information, the 3-o-6 code has a better inherent fault resilience, because it only requires two PBCB.

Table 6.2 shows the same information for the case of $f = 2$. Note that here we have only included entries for which F_2 is greater than zero. As already mentioned before, for the fault assumption $f = 2$ the number of check blocks depends on the actual number of data blocks.

Table 6.1: SOG partitionings for m-of-n codes ($f = 1$)

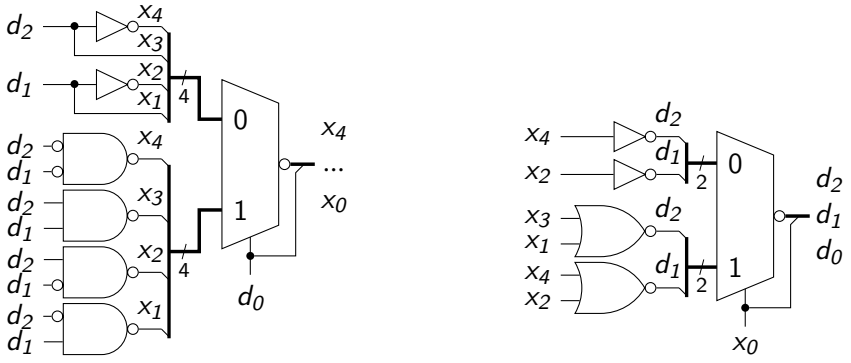
Code	Encoded Databits (k)	Partitioning	PBCB Width (j)	F_1	(possible) Check Block Code
1-of-4	2	$4 \times K_1$	2	0	1-of-4/1-of-2
2-of-5	3	$4 \times K_2$	2	0.33	1-of-4/1-of-2
2-of-7	4	$8 \times K_2$	3	0.25	2-of-5
3-of-6	4	$4 \times K_4$	2	0.5	1-of-4/1-of-2
3-of-7	5	$8 \times K_4$	3	0.4	2-of-5
4-of-8	6	$8 \times K_8$	3	0.5	2-of-5

Table 6.2: SOG partitionings for m-of-n codes ($f = 2$)

Code	Encoded Databits (k)	Partitioning	PBCB Width (j)	F_2	(possible) Check Block Code
3-of-6	4	$8 \times K_2$	3	0.25	2-of-5
3-of-7	5	$16 \times K_2$	4	0.2	2-of-7/3-of-6
4-of-8	6	$32 \times K_2$	5	0.17	3-of-7

6.2 Implementation Examples

For the implementation examples we have selected the 2-of-5 and 3-of-6 codes. According to Table 6.1 both codes require two PBCB, which will be encoded with the 1-of-4 code. Figure 6.1 shows the encoder and decoder circuits that we have developed for the 2-of-5 code. These circuits perform the mapping introduced in Figure 4.5. Notice that the bit d_0 is directly mapped to the rail x_0 and vice versa. This allows for a very efficient circuit design. A similar mapping has



(a) Encoder (b) Decoder

Figure 6.1: 2-of-5 circuits ($f = 1$)

been chosen for the 3-of-6 code. Figure 6.2 shows the corresponding SOG partitioning and code word mapping (the unused code words were omitted from the figure). Here the two right-most

bits in every code word and data word (i.e. (x_1,x_0) and (d_1,d_0)) coincide. These bits are used to control the encoding and decoding process. If (d_1,d_0) is $(1,1)$ or $(0,0)$, the remaining bits in the code word $(x_5,...,x_2)$ form a one-hot or one-cold code, respectively. For the cases $(0,1)$ and $(1,0)$ an "almost" systematic mapping can be achieved for the bits (x_4,x_3). The resulting circuits are shown in Figure 6.3. As already discussed in Section 4.4, there does not exist a decoder for the 3-of-6 code that is able to safely handle the unused code words. Hence the decoder circuit must be extended with a special error detection circuit that detects the unused code words. In particular this is necessary for the code words 001101, 110010, 010101 and 101010. This is of course only necessary, because the used CD is also triggered by these code words. However, the simpler design of the CD offsets the costs for the additional error detection logic.

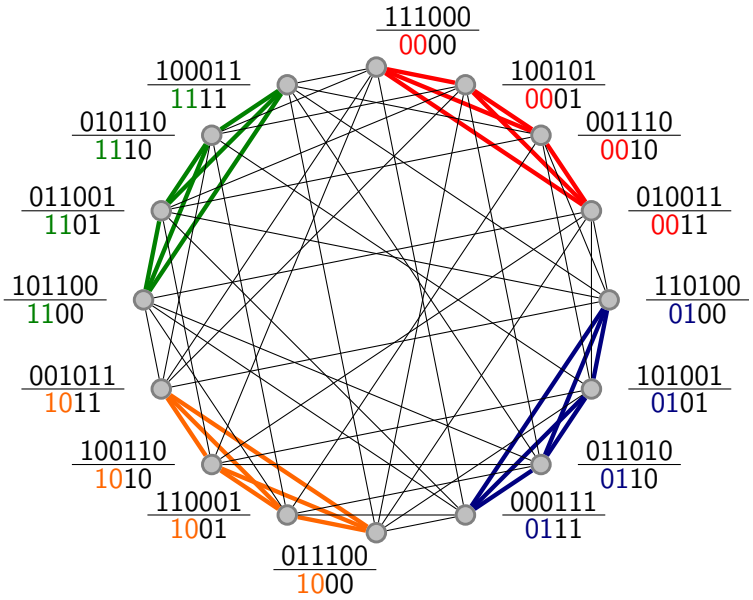


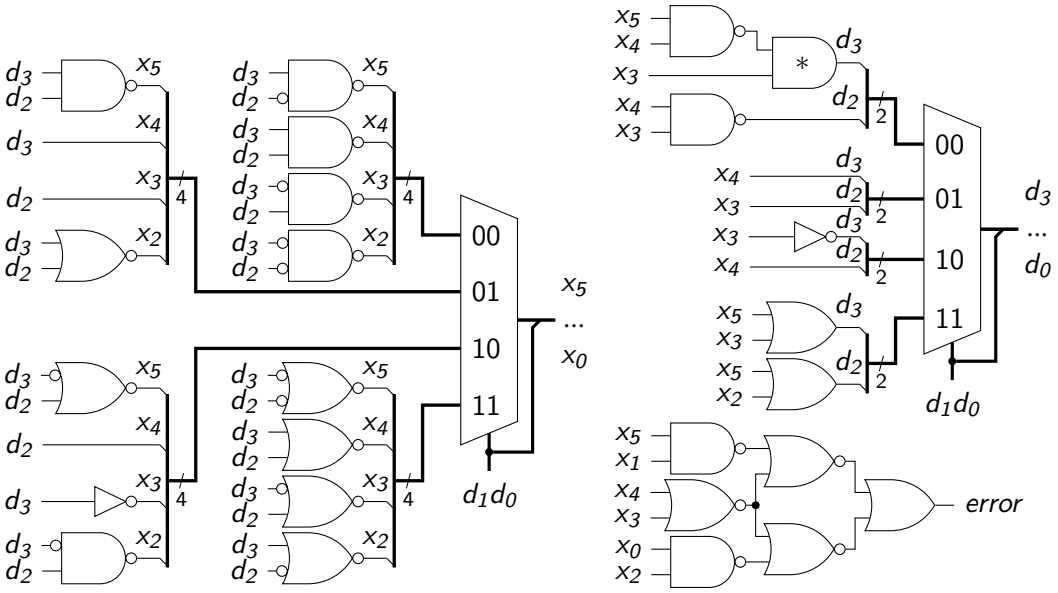
Figure 6.2: 3-of-6 code partitioning and code word mapping

Figure 6.4 shows the CDs for both codes. The design for the 2-of-5 code is inspired by the CD of the incomplete 2-of-7 code proposed in [3]. For the 3-of-6 code a sorting network based approach was chosen [22, 23].

The invalid code words are handled safely by both decoder circuits. Notice, however, the AND gate labeled with the $*$ symbol in the 3-of-6 decoder depicted in Figure 6.3b. For the normal circuit operation on valid code words this gate is redundant. It is, however, required to safely map the invalid code word 010100 to something different from 1100, which would be an ambiguous mapping.

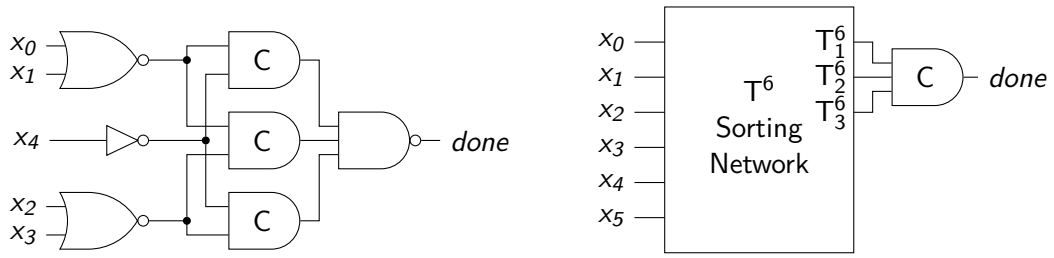
As can be seen in Equations 6.2 and 6.3 the PBCB for both codes are simply a subset of the data bits, hence no calculations are involved.

$$f_c^{\text{2-of-5}}(d_2,d_1,d_0) = (d_2,d_1) \tag{6.2}$$



(a) Encoder (b) Decoder

Figure 6.3: 3-of-6 circuits ($f = 1$)



(a) 2-of-5 (b) 3-of-6

Figure 6.4: Completion detector circuits ($f = 1$)

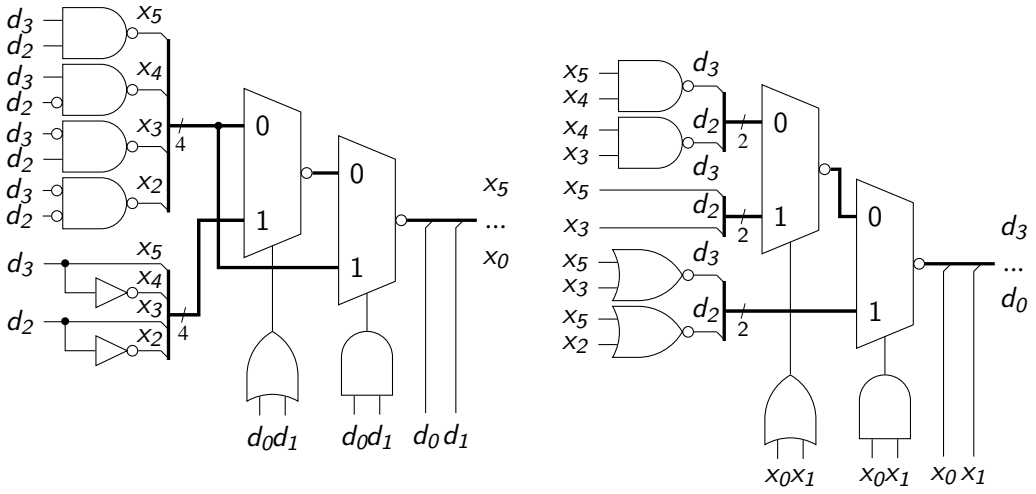
$$f_c^{\text{3-of-6}}(d_3, d_2, d_1, d_0) = (d_3, d_2) \tag{6.3}$$

Table 6.3 summarizes the implementation costs for the required circuits in number of transistors. To calculate the transistor counts in this section we used two and three input NAND/NOR gates (4 and 6 Transistors), Inverters (2 Transistors), XOR/XNOR gates (12 Transistors) and inverting two to one multiplexors (10 Transistors). Additionally two and three input C gates are used (8 and 10 Transistors). To make these values comparable, we have also implemented non fault-tolerant versions of these circuits. However, this only led to a reduction for the 3-of-6 code. The simplified, i.e. non-fault-tolerant, encoder and decoder circuits for the 3-of-6 code are shown in Figure 6.5. Here a different, more regular mapping was used to achieve less area complexity.

Table 6.4 shows the results of the proposed coding scheme for different input data widths.

Table 6.3: Implementation costs m-of-n codes ($f = 1$)

		$f = 0$		$f = 1$	
Code	CD	Encoder	Decoder	Encoder	Decoder
1-of-4	12	24	12	–	–
2-of-5	40	62	34	62	34
3-of-6	110	112	68	164	92+26



(a) Encoder (b) Decoder

Figure 6.5: 3-of-6 circuits (no fault tolerance)

Since the 2-of-5 code can hold three data bits, the last data block can never be fully used in the case of a power-of-2 bus width. For efficiency reasons we therefore use a 1-of-2 or 1-of-4 code for the last block. The columns labeled with $f = 0$ show the results for a non fault tolerant DI communication link. The parameter R denotes the coding efficiency (measured in bits/rails), while P specifies the number of transitions required to transmit one data bit, which gives some insight into the dynamic power consumption of the DI bus. The decoding costs also include the costs for the required CDs. The next set of columns show the corresponding values for a fault tolerant version of the same link. As can be seen the encoding overhead lies between 42 and 80%, while decoding has 33 to 59% higher implementation costs. When comparing both codes, it turns out that the 2-of-5 circuits are significantly cheaper, but also require more rails (worse coding efficiency). This is mainly due to the much less expensive encoding and decoding logic.

Table 6.5 shows R and P for a similar coding scheme proposed in [30] and briefly discussed in Section 2.3.3.2. In contrast to this scheme, which has logarithmic overhead (in terms of check information), our solution requires only constant overhead for the $f = 1$ case. This is because our scheme is specifically tailored for the special problems inherent to DI coding and leverages the knowledge that all effects of a single fault remain confined to the affected DI block. Hence we get better results for all cases.

Table 6.4: Implementation costs for different bus widths

bus size	Code	$f=0$ #rails	$f=0$ R	$f=0$ P	$f=0$ enc. costs	$f=0$ dec. costs	$f=1$ #rails	$f=1$ R	$f=1$ P	$f=1$ enc. costs	$f=1$ dec. costs	$f=1$ enc. overhead	$f=1$ dec. overhead
8	2-of-5	14	0.57	1.25	148	182	18	0.44	1.50	248	290	1.68	1.59
	3-of-6	12	0.67	1.50	224	364	16	0.50	1.75	404	552	1.80	1.52
16	2-of-5	27	0.59	1.38	312	404	31	0.52	1.50	472	578	1.51	1.43
	3-of-6	24	0.67	1.50	448	730	28	0.57	1.63	780	1072	1.74	1.47
32	2-of-5	54	0.59	1.31	644	814	58	0.55	1.50	936	1114	1.45	1.37
	3-of-6	48	0.67	1.50	896	1462	52	0.62	1.56	1532	2116	1.71	1.45
64	2-of-5	107	0.60	1.34	1304	1668	111	0.58	1.38	1848	2226	1.42	1.33
	3-of-6	96	0.67	1.50	1792	2926	100	0.64	1.53	3036	4204	1.69	1.44

Table 6.5: Efficiency comparison to existing solution [30]

bus size	Code	Results of [30] #rails	Results of [30] R	Results of [30] P	Improvement R	Improvement P
8	2-of-5	20	0.40	2.00	1.11	1.33
	3-of-6	18	0.44	2.25	1.13	1.29
16	2-of-5	35	0.46	1.75	1.13	1.17
	3-of-6	36	0.44	2.25	1.29	1.38
32	2-of-5	65	0.49	1.63	1.12	1.08
	3-of-6	60	0.53	1.88	1.15	1.20
64	2-of-5	120	0.53	1.50	1.08	1.09
	3-of-6	108	0.59	1.69	1.08	1.10

6.3 Behavioral Simulation

For the simulation we have selected the AND-masking transmitter and the base receiver with version B of the sampler. The DI link is eight bits wide and uses the 3-of-6 code for its data blocks. To order to keep the simulation simple we use the fault assumption $f = 1$. This means that the encoder and decoder circuits, discussed in the previous section, can be used and that one check block is required. This (single) check block will be encoded using the 1-of-4 code. Figure 6.6 shows an overview of the simulated DI link.

Moreover, the following values are used for delay elements in the transmitter and receiver circuits.

$$\Delta_{req} = 1.5ns, \Delta_{ack} = 0ns, \Delta_{ED} = 1.5ns, \Delta_{P} = 250ps$$

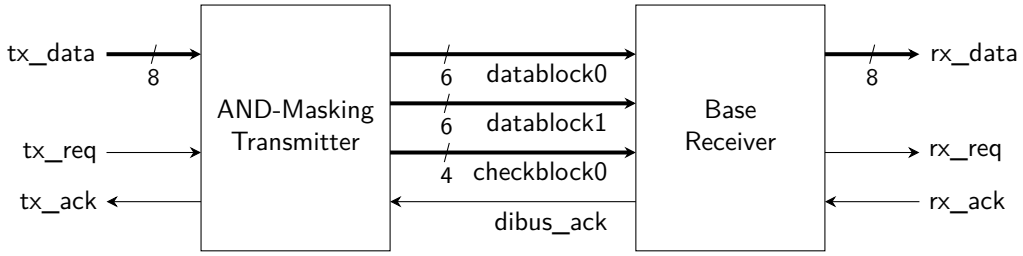


Figure 6.6: Simulated DI Link

The simulated BD input and output channels provide rising and falling transitions for the *tx_req* and *rx_ack* signals $250ps$ after the the associated signal event on *tx_ack* and *rx_req*. The individual rails of the DI bus have been assigned arbitrary delays between 300 and $1000ps$.

As discussed in Chapter 4 a transient fault can transform a valid code word to a bit vector that falls into one of the following categories.

- other valid code word
- unused code word
- invalid code word

We simulated these three scenarios in Modelsim to demonstrate how the different fault types are handled by the receiver. In all simulations the data word $0x00$ is transmitted. Thus in a fault-free transmission both data blocks should transmit the code word $di_D(0000) = 111000$ (notice that $f_c(0000) = 00$). The transient fault, which is introduced into the system by using the "force" command, affects one of the rails of *datablock0*.

Figure 6.7 shows the scenario, where a fault, striking at rail zero of *datablock0*, changes the code word being transmitted to 101001. Notice that the rising edge of the fault completes the 3-of-6 code word which causes the *done* output of the CD and consequently the *trg* signal to go high. Hence the erroneous code word is captured into the receiver's input register (*dibus_in_reg*). Now the error detection process is started. The decoded value for *datablock0* is $di_D^{-1}(101001) = 0101$, which means that its PBCB are given by $f_c(0101) = 01$. This value is obviously different from the PBCB generated for this block by the transmitter ($f_c(0000) = 00$). Thus the error detecting code is able to detect this change and eventually the error signal *checkblock_error* is raised, indicating that the check block calculated from the received data blocks does not match the received check block. The *err* signal, which is given by the logical or between *checkblock_error* and *decode_error*, then triggers the resampling event (negative pulse on *trg*). Since the transient fault disappeared in the meantime the correct data can be captured. After the error detection logic confirms the correct reception the output data can finally be forwarded to the BD output channel.

Figure 6.8 shows how an unused code word is handled by the receiver. Here the transient fault affects rail one and ultimately causes the input register to capture the unused code word 101010. Notice that with this code word as input the decoder produces the data word 0010, which in turn results in $f_c(0010) = 00$ for the PBCB. However, these PBCB match the ones of the original

(i.e. fault free) data block. Hence the error detecting code is unable to detect this change and the decoder must itself provide an appropriate error signal (*decode_error*).

Finally Figure 6.9 demonstrates how a fault can lead to an invalid code word (in this case 111001) being captured by the receiver. The fault strikes at rail zero shortly after the code word presented to the CD is already complete and is hence stored into the input register when the *trg* signal reaches it. However, the PBCB generated from the data word $di_D^{-1}(111001) = 1101$ are $f_c(1101) = 11$ and the error is hence visible to the error detecting code.

To make the timing diagrams easier to understand, we highlighted (letters A-D) four important events that occur in all of the three fault scenarios discussed above.

A) The transient fault strikes and changes the code word transmitted in *datablock0*.

B) The CD recognizes complete code words in all blocks of *dibus_in* and asserts its *done* output, which leads to the current state of the bus being captured into the receiver's input register (*dibus_in_reg*).

C) Since the error detecting logic indicates a transmission error, a negative pulse is produced on the *trg* signal to recapture the DI input data.

D) After error detection confirms a correct reception, the acknowledgment signal of the DI link (*dibus_ack*) is asserted.

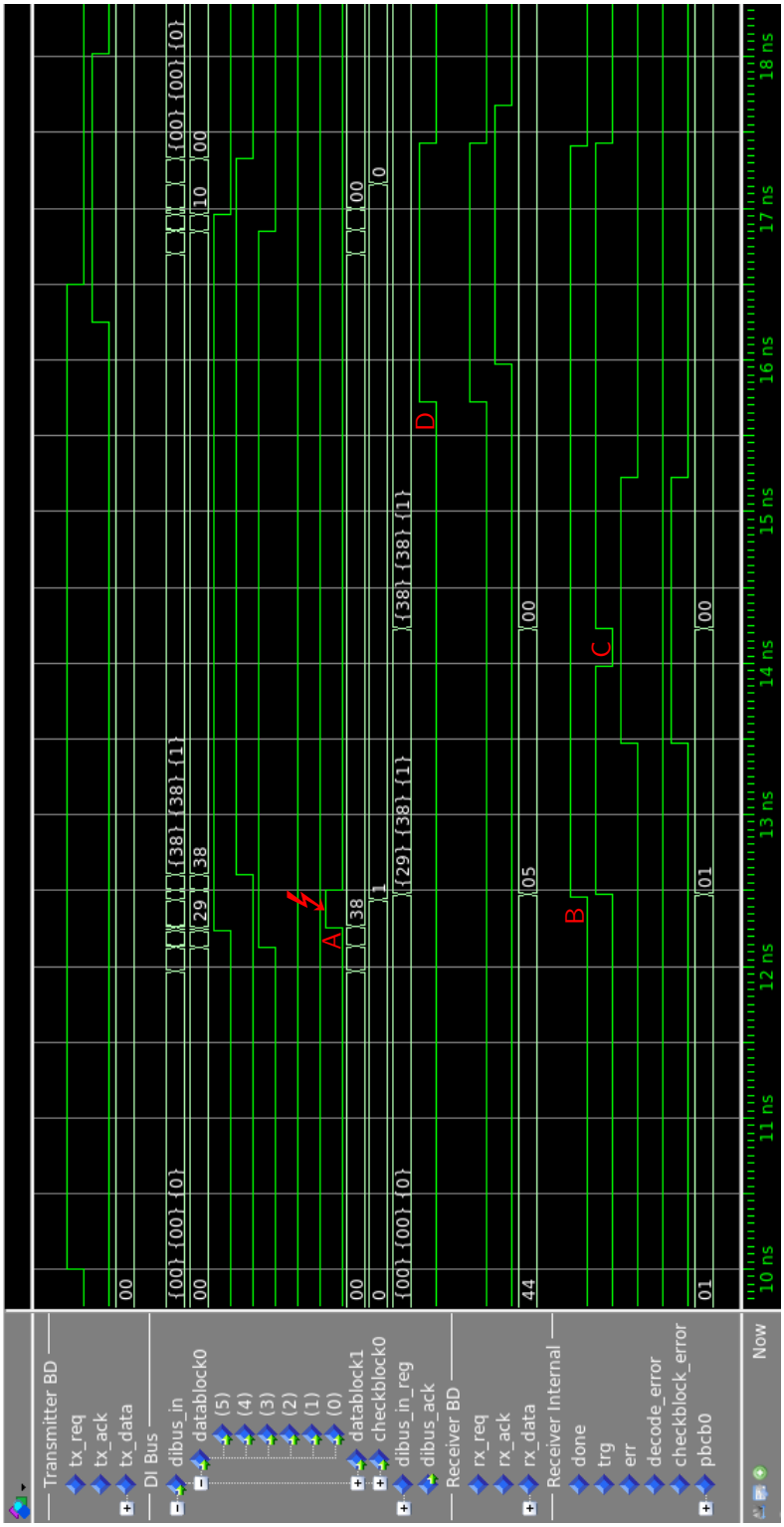


Figure 6.7: Transmission fault generating a different valid code word

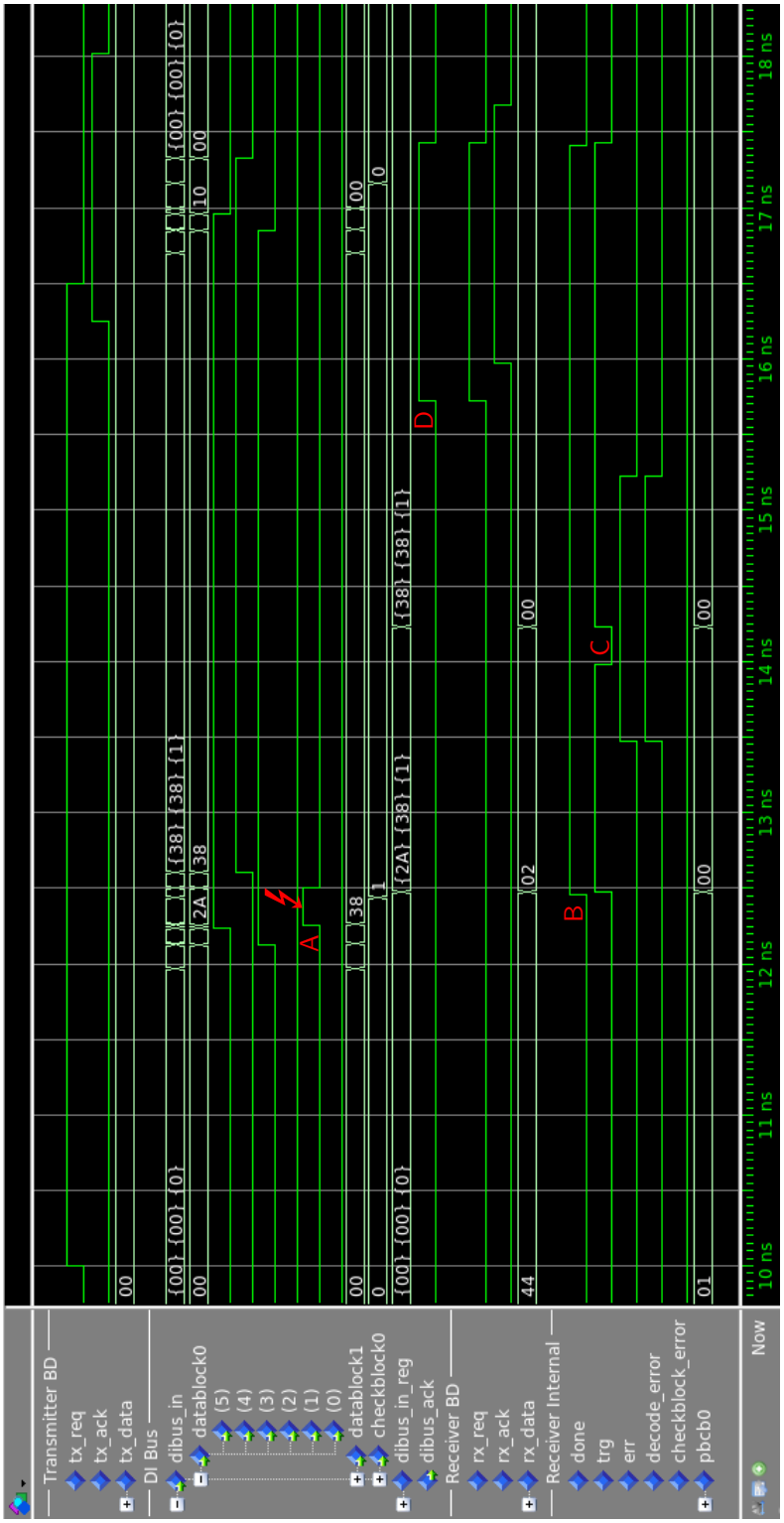


Figure 6.8: Transmission fault generating an unused code word

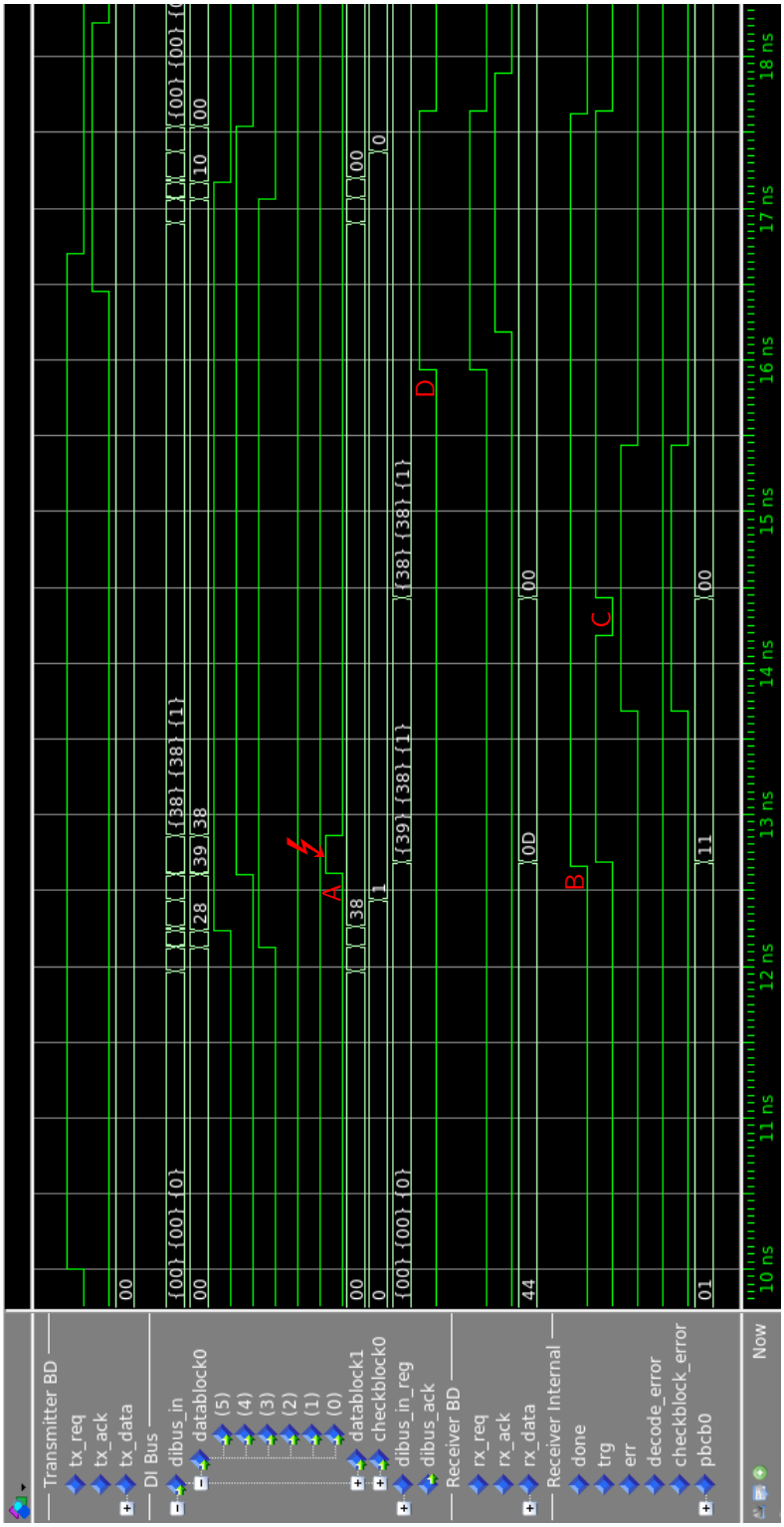


Figure 6.9: Transmission fault generating an invalid code word

CHAPTER 7

Conclusion and Future Work

Due to their robustness against timing variations, delay-insensitive codes present an interesting and very promising approach to implement asynchronous communication links. However, as we have shown, these codes are, without further measures, typically not resilient against transient faults. Moreover, the task of protecting this communication against such faults is nontrivial and requires a good understanding of the fault behavior of DI codes.

Hence, the first contribution of this thesis was an in-depth analysis of the Safe Overlap Graph fault model as well as an extension thereof that also takes invalid code words into account. This extended model allows us to determine every possible fault scenario and check whether a possible solution works in all of these cases. Using this as foundation, we have developed a new fault tolerant DI coding scheme that is based on a two level encoding process comprising an error detecting and DI coding layer. In particular, the payload data is divided into blocks that are then reduced to their so called Per Block Check Bits. The error detecting code, which ultimately yields the check blocks, is then only calculated over these reduced versions of the data blocks which results in less check information and simpler encoding circuits. This reduction is performed using a graph theoretical approach by leveraging the particular properties of the used DI code. In a second step the actual DI encoding is applied to the data and check blocks that can then be transmitted over an appropriate DI channel. The provided redundant information is sufficient to allow the receiver to detect erroneous (even though valid in the DI code space) vectors. In such a case the acknowledgment is deferred until the transient has ceased and the correct data can be read.

Based on existing solutions, we then developed a range of transmitter and receiver circuits that can be used for an actual implementation of the proposed coding scheme. Moreover, we presented two circuit level implementation examples for the 2-of-5 and 3-of-6 code, which show that the approach is feasible and can be implemented with reasonable area overhead.

We have analyzed the coding efficiency and the required number of check bits for representatives of the important class of m-of-n codes. Our results show that the proposed coding scheme scales better than existing approaches and has a considerably better coding efficiency.

Since every thesis has a bounded scope some interesting topics and directions are left unexplored. One interesting point for future research is to expand the proposed approach to permanent faults and pipelined links. These two problems are actually quite closely related in that in both cases faults cannot be recovered by simply waiting for a certain amount of time, because they have manifested themselves somewhere in the circuit (either in a storage element or in the form of a stuck-at fault or broken connection). To find a solution for these cases the forward error correction mechanism, briefly described in Section 4.5 can be a good starting point. However, care must be taken because by allowing a wider range of fault scenarios also the possibility of deadlocks becomes an issue. Another direction worth pursuing are 2-phase codes and communication links, which were not addressed in this thesis. Because of the close relationship between 4-phase and transition encoded 2-phase communication, the proposed fault model and coding scheme could be extended for this case.

Bibliography

[1] N. Weste and D. Harris, *Integrated Circuit Design*. Pearson Education, Limited, 2011.

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

[3] W. Bainbridge, W. B. Toms, D. Edwards, and S. Furber, "Delay-insensitive, point-to-point interconnect using m-of-n codes," in *Ninth International Symposium on Asynchronous Circuits and Systems*, pp. 132–140, 2003.

[4] T. Verhoeff, "Delay-insensitive codes - an overview," 1988.

[5] J. Sparsø, "Asynchronous circuit design - a tutorial," dec 2001.

[6] R. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and Materials Reliability*, vol. 5, pp. 305–316, Sept 2005.

[7] R. E. Blahut, *Algebraic codes for data transmission*. Cambridge, New-York: Cambridge University Press, 2003. Autres tirages : 2004, 2006.

[8] R. Roth, *Introduction to Coding Theory*. New York, NY, USA: Cambridge University Press, 2006.

[9] M. Delvai and A. Steininger, "Solving the fundamental problem of digital design - a systematic review of design methods," in *9th EUROMICRO Conference on Digital System Design (DSD'06)*, pp. 131–138, 2006.

[10] E. G. Friedman, "Clock distribution networks in synchronous digital integrated circuits," *Proceedings of the IEEE*, vol. 89, pp. 665–692, May 2001.

[11] A. J. Martin, "The limitations to delay-insensitivity in asynchronous circuits," in *Proceedings of the Sixth MIT Conference on Advanced Research in VLSI*, AUSCRYPT '90, (Cambridge, MA, USA), pp. 263–278, MIT Press, 1990.

[12] K. van Berkel, "Beware the isochronic fork," *Integr. VLSI J.*, vol. 13, pp. 103–128, June 1992.

[13] M. Shams, J. C. Ebergen, and M. I. Elmasry, "A comparison of cmos implementations of an asynchronous circuits primitive: the c-element," in *International Symposium on Low Power Electronics and Design*, pp. 93–96, Aug 1996.

[14] I. E. Sutherland, "Micropipelines," *Commun. ACM*, vol. 32, pp. 720–738, June 1989.

[15] M. Singh and S. M. Nowick, "Mousetrap: High-speed transition-signaling asynchronous pipelines," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, pp. 684–698, June 2007.

[16] S. Furber and P. Day, "Four-phase micropipeline latch control circuits," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 4, pp. 247–253, June 1996.

[17] J. Sparso and J. Staunstrup, "Design and performance analysis of delay insensitive multi-ring structures," in *Proceeding of the Twenty-Sixth Hawaii International Conference on System Sciences, 1993*, vol. i, pp. 349–358 vol.1, Jan 1993.

[18] G. Sobelman and K. Fant, "Cmos circuit design of threshold gates with hysteresis," in *Proceedings of the 1998 IEEE International Symposium on Circuits and Systems*, vol. 2, pp. 61–64 vol.2, May 1998.

[19] I. David, R. Ginosar, and M. Yoeli, "An efficient implementation of boolean functions as self-timed circuits," *IEEE Transactions on Computers*, vol. 41, no. 1, pp. 2–11, 1992.

[20] M. Blaum and J. Bruck, "Unordered error-correcting codes and their applications," in *Twenty-Second International Symposium on Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers*, pp. 486–493, July 1992.

[21] D. Lloyd and J. Garside, "A practical comparison of asynchronous design styles," in *Seventh International Symposium on Asynchronus Circuits and Systems*, pp. 36–45, 2001.

[22] S. Piestrak, "Membership test logic for delay-insensitive codes," in *Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 194–204, 1998.

[23] F. Huemer, M. Schütz, and A. Steininger, "Revisiting sorting network based completion detection for 4 phase delay insensitive codes," in *Austrian Workshop on Microelectronics (Austrochip)*, pp. 3–8, Sept 2015.

[24] M. Cannizzaro, W. Jiang, and S. Nowick, "Practical completion detection for 2-of-n delay-insensitive codes," in *2010 IEEE International Conference on Computer Design (ICCD)*, pp. 151–158, 2010.

[25] P. McGee, M. Agyekum, M. Mohamed, and S. Nowick, "A level-encoded transition signaling protocol for high-throughput asynchronous global communication," in *14th IEEE International Symposium on Asynchronous Circuits and Systems*, pp. 116–127, 2008.

[26] M. Schütz, F. Huemer, and A. Steininger, "A practical comparison of 2-phase delay insensitve communication protocols," in *Austrian Workshop on Microelectronics (Austrochip)*, pp. 15–20, Sept 2015.

[27] S. Universitat PolitÃÍcnica de Catalunya, Barcelona, "Petrify Website." `http://www.lsi.upc.edu/~jordicf/petrify/`, Dez. 2016.

[28] I. Poliakov, V. Khomenko, and A. Yakovlev, *Workcraft – A Framework for Interpreted Graph Models*, pp. 333–342. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[29] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, pp. 11–33, Jan 2004.

[30] J. Lechner, A. Steininger, and F. Huemer, "Methods for analysing and improving the fault resilience of delay-insensitive codes," in *33rd IEEE International Conference on Computer Design (ICCD)*, pp. 519–526, Oct 2015.

[31] J. Bainbridge and S. Furber, "Chain: a delay-insensitive chip area interconnect," *Micro, IEEE*, vol. 22, pp. 16–23, Sep 2002.

[32] F.-C. Cheng and S.-L. Ho, "Efficient systematic error-correcting codes for semi-delay-insensitive data transmission," in *International Conference on Computer Design, ICCD 2001. Proceedings.*, pp. 24–29, 2001.

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

[34] M. Agyekum and S. Nowick, "Error-correcting unordered codes and hardware support for robust asynchronous global communication," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, pp. 75–88, Jan 2012.

[35] J. Berger, "A note on error detection codes for asymmetric channels," *Information and Control*, vol. 4, no. 1, pp. 68 – 73, 1961.

[36] J. Pontes, N. Calazans, and P. Vivet, "Adding temporal redundancy to delay insensitive codes to mitigate single event effects," in *18th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pp. 142–149, May 2012.

[37] J. Pontes, N. Calazans, and P. Vivet, "Parity check for m-of-n delay insensitive codes," in *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*, pp. 157–162, July 2013.

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

[39] J. Lechner and R. Najvirt, “A generic architecture for robust asynchronous communication links,” in *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation* (J. Ayala, D. Shang, and A. Yakovlev, eds.), vol. 7606 of *Lecture Notes in Computer Science*, pp. 121–130, Springer Berlin Heidelberg, 2013.

[40] F. Huemer, J. Lechner, and A. Steininger, “A new coding scheme for fault tolerant 4-phase delay-insensitive codes,” in *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pp. 392–395, Oct 2016.

[41] E. Dubrova, *Fault-Tolerant Design*. Springer New York, 2013.

[42] L. Kleeman and A. Cantoni, “Metastable behavior in digital systems,” *IEEE Design Test of Computers*, vol. 4, pp. 4–19, Dec 1987.

[43] T. Polzer and A. Steininger, “An approach for efficient metastability characterization of fpgas through the designer,” in *2013 IEEE 19th International Symposium on Asynchronous Circuits and Systems*, pp. 174–182, May 2013.