Informatics

Contributions to Efficiency and Robustness of Quasi Delay-Insensitive Circuits

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

eingereicht von

Dipl.-Ing. Florian Huemer, BSc
Matrikelnummer 0828465

an der Fakultät für Informatik

der Technischen Universität Wien

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

Diese Dissertation haben begutachtet:

Prof. Matthias Függer

Prof. Alex Yakovlev

Wien, 17. Mai 2022

Florian Huemer

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Informatics

Contributions to Efficiency and Robustness of Quasi Delay-Insensitive Circuits

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Dipl.-Ing. Florian Huemer, BSc
Registration Number 0828465

to the Faculty of Informatics

at the TU Wien

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

The dissertation has been reviewed by:

Prof. Matthias Függer

Prof. Alex Yakovlev

Vienna, 17th May, 2022

Florian Huemer

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
Bibliothek
Your knowledge hub

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Florian Huemer, BSc

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, 17. Mai 2022

Florian Huemer

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Acknowledgments

First of all, I want to thank Kimberly. Without your loving encouragement, help and devotion, especially during the last, very busy months when I put together this thesis, I would not have been able to achieve my goals with this work. Another huge “thank you” goes to Kiara. Your lighthearted and happy nature always sparks joy and is an incredible enrichment to our life. I also want to thank my parents for the great support through the early years of my studies.

Thanks to my adviser Andreas Steininger for introducing me into the world of asynchronous circuits and for the opportunity to work in this fascinating and multifaceted field. I am very grateful for your guidance throughout the years. Our numerous discussions and your specific view on problems always helped me in finding a path forward in my research.

Another special thank goes to Robert Najvirt for our collaborative work as well as for your insightful and valuable input for this work, which was always highly appreciated.

I also want to thank my current and past colleges at the Institute of Computer Engineering for their joint contribution in providing an excellent working environment. I could not have wished for a better setting for my studies. This includes, among many others, Arman, Giorgio, Heinz, Hugo, Krisztina, Kyrill, Martin, Raghda, Rojo, Roman, Thomas, Traude, Zaheer and of course Ulrich Schmid as the head of our working group. In addition to that extra special thanks go to Jürgen Maier and Florian Kriebel for the great cooperation during our joint teaching duties.

This research was partially supported by the project ENROL (grant I 3485-N31) of the Austrian Science Fund (FWF).

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Auf dem Gebiet der digitalen integrierten Schaltungen sind asynchrone und im speziellen quasi-laufzeitunabhängige (engl. quasi delay-insensitive (QDI)) Designs dafür bekannt, besonders robust gegenüber fertigungsbedingten Variationen bzw. Änderungen der Umgebungstemperatur sowie der Betriebsspannung zu sein (engl. process, voltage and timing (PVT) variations) – ein zunehmend erstrebenswertes Verhalten. Der Hauptgrund für diese Eigenschaften liegt darin, dass im QDI Design nur sehr wenige Signallaufzeitbedingungen notwendig sind, um das korrekte Verhalten einer Schaltung garantieren zu können. Dies steht im starken Kontrast zum starren zeitlichen Ablauf, der dem Funktionsprinzip synchroner Schaltungen zu Grunde liegt. Dieses flexible Zeitverhalten macht derartige QDI Designs prädestiniert für den Einsatz in einigen hoch relevanten Anwendungsgebieten – zwei davon stehen im Fokus dieser Dissertation. Die inhärente Robustheit gegenüber Laufzeitvariationen macht QDI Designs (i) bestens geeignet um laufzeitunabhängige (engl. delay-insensitive (DI)) Kommunikationskanäle für Inter- oder Intra-Chipverbindungen zu implementieren und (ii) eine vielversprechende Wahl für die Realisierung fehlertoleranter Systeme.

Der erste Teil dieser Arbeit wird sich daher mit der Untersuchung von laufzeitunabhängiger Kommunikation beschäftigen, welche die Implementierung hochflexibler Übertragungsstrecken erlaubt. Die einschlägige Fachliteratur zeigt, dass eine große Vielfalt an Möglichkeiten für die Realisierung solcher laufzeitunabhängigen Übertragungsstrecken existiert. Eine besondere Herausforderung, die sich dabei stellt, ist es, eine Balance zwischen verschiedenen, oft entgegengesetzten Optimierungskriterien für eine effiziente Implementierung zu finden. Durch eine umfassende Analyse von bereits bekannten, aber auch neu entwickelten Protokollen, Datenkodierungen und Schaltungen präsentiert diese Arbeit einige effiziente Lösungen und Optimierungsstrategien zur Realisierung von laufzeitunabhängiger Kommunikation. Weiters werden auch Schnittstellenmodule untersucht, die einen effizienten Datenaustausch zwischen synchronen und asynchronen Subsystemen einer Schaltung ermöglichen.

Der zweite Teil dieser Arbeit ist der Analyse der Fehlertoleranzaspekte von QDI Schaltungen gewidmet. Dabei wird der Fokus auf die Untersuchung von transienten Fehlern gelegt, welche sich typischerweise als kurze (wenige hundert Picosekunden dauernde) Spannungs- bzw. Stromspitzen in einer Schaltung manifestieren. Solche Fehler werden hauptsächlich durch kosmische Strahlung verursacht und beeinträchtigen eine Schaltung

nur für einen begrenzten Zeitraum, verursachen also keinen permanenten Schaden. Der anhaltende Trend zu immer kleineren Strukturgrößen und Versorgungsspannungen macht digitale Schaltungen anfälliger für diesen Fehlertyp. Aus diesem Grund sind Maßnahmen zur Steigerung der Resilienz nicht nur für Systeme, die eine besonders hohe Zuverlässigkeit garantieren müssen, relevant, sondern in zunehmendem Maß auch für "normale" Verbraucherelektronik.

Die wissenschaftliche Untersuchung der Fehlertoleranzeigenschaften von QDI Schaltungen ist deshalb notwendig, weil sich deren Verhalten unter der Einwirkung von (transienten) Fehlern fundamental von dem synchroner Schaltungen unterscheidet, welche bereits gut erforscht sind. Ein einziger Fehler kann bei asynchronen Schaltungen beispielsweise zur Selbstblockierung (engl. deadlock) eines gesamten Systems oder zur Erzeugung neuer Datenelemente führen – Effekte, für die es in synchronen Schaltungen keine direkten Entsprechungen gibt. Das hat auch zur Folge, dass Maßnahmen zur Erkennung, Vermeidung und Behandlung von Fehlern, die für synchrone Designmethoden entwickelt wurden, nicht (immer) direkt auf asynchrone Schaltungen anwendbar sind. Deshalb beschäftigt sich diese Dissertation speziell mit der Analyse der Auswirkungen von transienten Fehlern auf QDI Designs und untersucht mögliche Verfahren zur Behandlung der auftretenden Effekte. Dabei werden sowohl bekannte Verfahren aus der Literatur sowie eigene Entwicklungen untersucht. Darüber hinaus wird ein umfassendes Softwarepaket für die Erzeugung, Simulation und Verifikation von asynchronen Schaltungen präsentiert, das auch bei weiteren Forschungsarbeiten zu diesem Thema Verwendung findet.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Abstract

In the field of digital integrated circuits asynchronous and especially quasi delay-insensitive (QDI) designs are known to have a high robustness against process, voltage and temperature variations – an increasingly desired property. This is because for QDI designs only very few timing assumptions and constraints are necessary to guarantee the correct behavior of a circuit, which is in strong contrast to the rigid timing scheme of the traditional synchronous design style. This characteristic key-property opens up many interesting and highly relevant application areas – two of are the focus of this thesis. The inherent robustness against timing variations makes QDI design styles and techniques (i) perfectly suited for constructing delay-insensitive (DI) communication channels for global inter- or intra-chip interconnect and (ii) a promising choice for the design of fault-tolerant systems.

The first part of this work is, hence, devoted to the investigation of efficient ways to transmit information in a DI way, which allows for the construction of highly flexible communication links. Literature shows that there is a large design space for how such DI links can be implemented, with many different and often opposing optimization criteria. We provide a comprehensive analysis of available protocols and data encoding schemes and complement them with our own contributions to the field. Furthermore, we also investigate interface components that enable synchronous parts of a system to efficiently communicate with asynchronous ones.

The second part of this thesis explores the fault-tolerance aspects of QDI design. Here we focus our work on transient faults, i.e., short voltage spikes that affect a circuit over a certain amount of time (usually a few hundred picoseconds), which are primarily caused by cosmic radiation. The ongoing trend to ever smaller supply voltages and feature sizes makes circuits more prone to these types of faults. Thus, countermeasures against such effects are not exclusively relevant for highly dependable systems, but also for everyday end-user electronics.

Research into the fault-tolerance properties of QDI circuits is necessary, because their behavior under faults is fundamentally different from that of circuits constructed using the well understood synchronous paradigm. Here, even single faults can lead to the creation of additional erroneous data elements or complete system deadlocks – effects for which there is no direct counterpart in synchronous designs. This also means that fault-mitigation and hardening strategies from synchronous design do not (always) directly translate over to the asynchronous world and that more specialized approaches are required. Hence, in this thesis, we analyze the effects of transient faults, investigate fault-mitigation strategies from literature and present and evaluate our own techniques. Moreover, we also contribute to the design of a comprehensive tool set to generate and simulate asynchronous circuits to enable further research in the future.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Contents

Kurzfassung **ix**

Abstract **xi**

Contents **xiii**

1 Introduction **1**
1.1 Research Questions . . . 4
1.2 Organization . . . 5

2 Background and Related Work **9**
2.1 Handshaking Protocols . . . 9
2.2 Delay Models . . . 12
2.3 Basic Asynchronous Circuit Elements . . . 15
2.4 Channels and Static Data-Flow Structures . . . 20
2.5 Bundled Data Circuits . . . 24
2.6 QDI Circuits . . . 25

3 Crossing the Boundary to Asynchrony **33**
3.1 Related Work . . . 35
3.2 Sampling Muller Pipelines . . . 36
3.3 Proposed Approach . . . 41
3.4 Desynchronizer . . . 43
3.5 Resynchronizer . . . 46
3.6 Results . . . 49
3.7 Conclusion . . . 52

4 Efficient Delay-Insensitive Communication **55**
4.1 Delay-Insensitive Codes . . . 57
4.2 Partially Systematic Constant-Weight Codes . . . 59
4.3 Hybrid Protocols . . . 66
4.4 Completion Detection . . . 73
4.5 Link Architecture . . . 85
4.6 Results . . . 91

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

4.7 Conclusion 101

5 Asynchronous Circuit Description **103**
5.1 Language Specification 105
5.2 Python Production Rule Package 115
5.3 Dual-Rail Logic Synthesis 124
5.4 Bounded Model Checking 138
5.5 Conclusion 156

6 Fault-Tolerance in QDI Circuits **159**
6.1 Related Work 160
6.2 Fault Effects and Sensitivity Windows 162
6.3 Fault Mitigation Strategies 164
6.4 Case Study: Pipeline 165
6.5 Improved Buffer Designs 170
6.6 Case Study: Multiplier 173
6.7 Conclusion 181

7 Conclusion and Future Work **183**

List of Figures **187**

List of Tables **191**

Acronyms **193**

Glossary **197**

Software and Tools **199**

Bibliography **201**

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 1

Introduction

To organize the data transfer between storage elements in digital circuits (e.g., registers in a pipeline) the designer has basically two options. In the widely used synchronous design style all storage elements are (simultaneously) triggered by a clock signal, while asynchronous designs use local closed-loop handshakes to perform this task [Spa20].

Since the clock signal must be routed to every single flip-flop of a design it is, by its nature, a very high fan-out signal. However, at the same time it must exhibit a minimal amount of skew, because the rigid synchronous timing model demands that all flip-flops in a design are triggered at (virtually) the same time. This leads to the situation that a considerable amount of (electrical) energy and engineering resources have to be put into the clock tree of a synchronous chip to uphold this abstraction [DIBM03, Fri01, LKM10]. The ever-increasing miniaturization of semiconductor circuits and the accompanying rising in chip-complexity and clock rates further exacerbate these problems and challenges.

To determine the maximal possible clock frequency a design can be operated with, a static timing analysis is performed. This analysis essentially searches for the longest delay between any two flip-flops in a design, which is referred to as the critical path. The clock frequency must then be chosen such that the clock period allows for the signal traveling along the critical path to reach its destination before the next clock edge. Because of process, voltage and timing (PVT) variations, some timing margin has to be added to the critical path delay (which thus lowers the maximal clock frequency). This means that the speed of the circuit is determined solely by the slowest path in the design, even if this path may only be relevant in very rare cases. Nevertheless, the synchronous design style is indisputably popular for industrial/commercial designs because it models time as discrete steps, which greatly simplifies circuit design and speeds up the development process. Virtually all commercial electronic design automation (EDA) tools are specifically optimized for and tailored to synchronous design.

However, modern high-performance chips heavily use dynamic (voltage and) frequency scaling and often multiple separate clock domains [RMGW09, SMB+02], which undermines this abstraction to a certain degree. Hence, these designs naturally contain a large number of clock domain crossings which come with their own set of problems (metastability, latency, area overhead, etc.). This situation lead to the concept of Globally Asynchronous Locally Synchronous (GALS) Systems [Cha84, TGL07], which use synchronous "islands" communicating over some form of asynchronous interconnect [KSS+16].

For these reasons it makes sense to also consider an alternative. Since asynchronous design styles don't use a (global) clock signal they don't suffer from many of the discussed problems. However, they come with their own difficulties, challenges and drawbacks. Asynchronous design styles can (among other characteristics) be classified based on the timing assumptions imposed on the circuits.

Bundled-data (BD) designs [SN07, Sut89] are quite similar to synchronous circuits as they use the same combinational logic to process binary data. However, instead of a clock signal they "bundle" the data that is transmitted from one storage element (the source) to the next (the sink) with an additional signal that indicates the validity of the data. This signal is usually referred to as *request* and can be viewed as a local replacement of the clock signal. It is used by the sink to trigger the capturing event of the transmitted data. As soon as the data is stored the sink uses the *acknowledgment* signal to inform the source that the data has been consumed and that new data can be sent[1]. Note that there is an immanent race condition between the request signal and the data that is being transmitted. It must be guaranteed that the request reaches the sink only after the data is stable at its input. In BD designs this timing constraint is usually fulfilled by inserting delay elements in the request paths. Thus, for correct circuit operation, it must be ensured that the delay on the request signal is at least as long as the critical path of the combinational logic between source and sink. Since for different input data, different critical paths through the combinational logic are relevant, it is even possible to dynamically select an appropriate delay using multiple different delay elements, depending on the actual data being processed – a strategy that improves the average-case performance of a circuit [Now96]. This is in contrast to the synchronous style which always has to accommodate the worst case. An important side-effect of the way asynchronous logic operates, is that there is only (switching) activity in the circuit when there is actually something being processed, which can have a positive impact on the power consumption of a chip. For synchronous designs, techniques like clock gating are required to achieve a similar effect. Asynchronous design styles further offer the benefit that circuits with different speeds can simply be connected without the need for clock domain crossing circuits, as would be needed for synchronous designs operating with different clock speeds.

Another possibility for implementing the request in an asynchronous circuit is to implicitly

---

[1] This explanation assumes push channels. In pull channels the meaning of the request and acknowledgment signals are reversed, see [Spa20] for a more detailed discussion.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

encode it in the data being transmitted. It is then 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 the circuit that performs it is called completion detector (CD). It is only possible if the code used to encode the data has certain properties [Ver88]. Possible choices include constant-weight (e.g., dual-rail) and Berger codes. Of course such special encodings cause a certain overhead, especially when considering computational logic. In comparison to synchronous or BD circuits the data processing is very different and induces additional costs in terms of area and delay (and hence power). However, the huge advantage that comes with this design paradigm is that circuits can be implemented in a quasi delay-insensitive (QDI) way, which means that they are basically immune to delay variations[2]. This allows circuits to, e.g., easily cope with (power) optimization techniques like sub-threshold operation.

These properties open up promising application areas for QDI circuits. The robustness against timing variations makes delay-insensitive (DI) communication channels very interesting for global inter- or intra-chip interconnect [BF02, PFT+07, Lin03, MG20], especially in the light of GALS systems. Moreover, asynchronous circuits in general and particularly QDI circuits are well-suited for systems that have to operate over a wide range of environment temperatures and supply voltages [KMM15, BRWG05]. Since their correct behavior doesn't rely on strict timing guarantees inside the circuit they can gracefully degrade performance and simply run slower if, e.g., the supply voltage is low. However, it is also worth noting that QDI circuits can be used to implement specialized high-performance hardware, which has been successfully demonstrated by Fulcrum Mircosystems with their network switches [DLD+14]. On a side note, there is also some indication that QDI circuits have some inherent robustness against power analysis attacks [HPL+16, LHCG17], which is important for security-critical applications.

PVT variations in digital circuits mainly cause delay faults. For a delay fault to manifest itself as an error in the value domain (i.e., a wrong binary value in some storage element) some timing assumption must be violated by it. However, as already discussed, asynchronous and especially QDI design styles have an inherent robustness against these faults. Unfortunately digital circuits are also affected by other environmental influences as well as internal defects which can directly cause value errors.

Basically faults can be classified as transient and permanent. Permanent faults, also referred to as hard-errors, result in physical damage to a circuit (gate oxide/interconnect wear-out, latchup, etc.) and cannot be corrected (although they may be tolerated to some degree) [WH11]. Transient faults, on the other hand, only affect a circuit over a limited period of time (usually a few hundred picoseconds) and cause a voltage spike on some internal node of the circuit, referred to as single-event transient (SET). If such a pulse is captured by a storage element it can manifest itself in the circuit's state, which is then referred to as a soft-error or single-event upset (SEU). In modern designs transient faults

---

[2] As will be explained in more detail in Section 2.2, the only timing constraint in QDI circuits comes in the form of the isochronic fork assumption, which is also the reason why we speak of *quasi* delay-insensitive and not complete delay-insensitive circuits.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

are mainly caused by cosmic radiation, i.e., high energy particles hitting a transistor in a chip [Bau05]. However, also cross-talk effects can play a role [MMK+02]. The ongoing trend to ever smaller supply voltages and feature sizes makes circuits more prone to these effects, because it reduces the critical charge a particle hit has to "overcome" to create an SET. Moreover, because the physical area a single transistor occupies is ever decreasing, the chance that a single particle hit affects multiple transistors is also increased. This altogether makes transient faults an issue not only for dependable, high-critically systems in harsh operation environment (like airplanes or space-probes) but also for everyday, non-safety-critical systems like consumer electronics [DDC05].

1.1 Research Questions

This thesis will focus on two aspects of QDI circuits which correspond to the two application areas identified in the discussion above. We want to (a) investigate efficient ways to transmit information in a DI way and (b) analyze existing QDI design styles for their behavior under transient faults and use this information to further improve their resilience to environmental influences.

1.1.1 Efficient DI Communication

For this research direction we start out by thoroughly investigating the current approaches for DI data transmission with regard to three main aspects:

- DI Codes (encoding/decoding overhead)
  How are the data words (usually in unencoded binary representation) mapped to the code words of some DI code? What are the advantages and disadvantages of the different DI codes?
- Completion detection
  What are the best known CDs for the different codes and are there any limitations to their QDI properties?
- Protocols and protocol converters
  Which protocols exist for DI communication and how do they impact both of the aforementioned points? How can the conversion between QDI and non-QDI asynchronous protocols be implemented most efficiently?

Based on this analysis we will then address the following research questions:

- While the mapping of data words to the code words is predefined for systematic codes like the Berger code, finding a good mapping for the non-systematic constant-weight codes is a non-trivial problem. From this observation, two questions arise: Is there a strategic way to perform this mapping? Can such a mapping simplify and, hence, improve the required encoders and decoders?

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

- Can the known CDs for constant-weight and Berger codes be further optimized with respect to area and delay? Can residual problems in the QDI properties be fixed?
- Are there areas in the design space of DI protocols, previously not explored by literature that have advantages over the state-of-the-art?
- How do the available and proposed design options compare to each other and how do they affect the overall overhead and performance of DI communication links?

Another important research direction that we want to pursue in this thesis is the junction between the synchronous and asynchronous worlds. In particular we want to investigate how an efficient data transmission between a synchronous and an asynchronous timing domain can be implemented. This process of *timing domain crossing* is vital for all asynchronous systems that at some point have to exchange information with a synchronous system or have to provide some synchronous interface. Conversely, synchronous systems that want to leverage the benefits of asynchronous subsystems – like, e.g., in a GALS system – are in need of such conversion circuits. Obviously, these must be fast and energy efficient, otherwise they would undermine the original purpose of the conversion.

1.1.2 Fault Tolerance in QDI circuits

For the second part of this thesis we strive to answer the following questions:

- Is there some inherent fault resilience in QDI circuits (i.e., fault masking)? If so, where exactly does it originate from? Can it be leveraged or even enhanced?
- Are there certain parts or states of a QDI circuit that are specifically prone to faults? Can the system be operated in a way to minimize its fault susceptibility?
- What are the high-level effects of faults (e.g., deadlock, token generation, etc.)?
- Which circuit design strategies exist that can prevent some of the fault effects or make them at least less likely to affect the system?

The main tool to investigate these research questions is simulation-based fault injection. Based on the analysis of the results gathered this way, we then present our own approaches for handling faults.

1.2 Organization

First, Chapter 2 gives a general overview of the field of asynchronous and specifically QDI circuits and discusses some related work. This will lay the foundation the following chapters build upon.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Chapters 3 and 4 tackle the research questions regarding DI communication, outlined in Section 1.1.1. First, Chapter 3 examines the challenges associated with passing data between synchronous and asynchronous timing domains and presents a novel solution to this problem. The results of this chapter have been published in

- Florian Huemer and Andreas Steininger. Timing Domain Crossing using Muller Pipelines. In *26th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 44–53, May 2020

Then, Chapter 4 focuses on the actual DI codes, protocols and communication links and their associated overhead considering all the parameters posed in Section 1.1.1. The three publications that originate from this work appeared in

- Florian Huemer and Andreas Steininger. Partially Systematic Constant-Weight Codes for Delay-Insensitive Communication. In *24th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 17–25, May 2018
- Florian Huemer and Andreas Steininger. Advanced Delay-Insensitive 4-Phase Protocols. In *Austrochip Workshop on Microelectronics (Austrochip)*, pages 50–55, Sep. 2018
- Florian Huemer and Andreas Steininger. Novel Approaches for Efficient Delay-Insensitive Communication. *Journal of Low Power Electronics and Applications*, 9(2), 2019

While the first two entries in this list are conference papers, the last one is an in-depth journal article that builds on top of them. The results of those two chapters are entirely my own work.

The second part of the thesis addresses the research questions outlined in Section 1.1.2, regarding the resilience of QDI circuits against (transient) faults and the tools we created to facilitate the required experiments for this investigation. Hence, Chapter 5 first presents our Python-based asynchronous circuit design tools package, that enables us to generate, verify and simulate the circuits needed in Chapter 6. Two selected parts of this Chapter 5 have been published in

- Florian Huemer and Andreas Steininger. Sorting Network based Full Adders for QDI Circuits. In *Austrochip Workshop on Microelectronics (Austrochip)*, pages 21–28, Oct. 2020
- Florian Huemer, Robert Najvirt, and Andreas Steininger. On SAT-Based Model Checking of Speed-Independent Circuits. In *IEEE 25th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 100–105, April 2022

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

I performed the majority of the research, conceptualization, design and implementation for the tools presented in this chapter. However, as this is a software project that is used by other people in our research group, I also received various useful suggestions for improvements and bugfixes, especially by Robert Najvirt.

Finally, Chapter 6 presents the research that is directly concerned with the topic of fault resilience of QDI circuits. Its results have been published in

- Florian Huemer, Robert Najvirt, and Andreas Steininger. Identification and Confinement of Fault Sensitivity Windows in QDI Logic. In *Austrochip Workshop on Microelectronics (Austrochip)*, pages 29–36, Oct. 2020
- Patrick Behal, Florian Huemer, Robert Najvirt, Andreas Steininger, and Zaheer Tabassam. Towards Explaining the Fault Sensitivity of Different QDI Pipeline Styles. In *27th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 25–33, 2021

As part of a larger research project, those publications are collaborative works. However, I had a key role in the conceptualization and the design of the performed experiments as well as the analysis of the results. The proposed circuit improvements are entirely my own work. For performing the actual large-scale (fault-injection) simulations we relied on tools mainly development by Patrick Behal and Robert Najvirt [BHNS21].

Chapter 7 summarizes the results of my work and concludes the thesis.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

CHAPTER 2

Background and Related Work

This chapter covers some of the basics of asynchronous design as well as common literature, that all followings chapters rely on. For a more thorough introdcution we refer to [Spa20] and [BOF10].

2.1 Handshaking Protocols

In contrast to the rigid time-driven regime of the synchronous design style, asynchronous circuits always use some form of closed-loop handshaking protocol to control the data transfer between storage elements (e.g., the individual buffer stages in a pipeline). As shown in Figure 2.1 this handshake (usually) involves two signals, which are referred to as request (*req*) and acknowledgment (*ack*). The rising edge of the *req* signal is typically used as an indicator by the source to notify the sink that new data is available. The sink then uses the *ack* signal to inform the source that it has received the data and that new data can be transmitted[1].

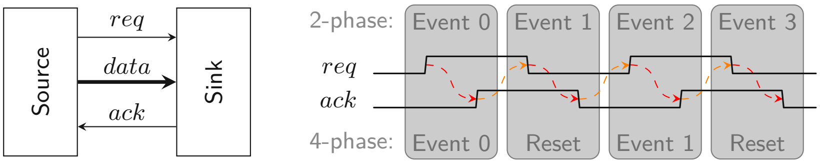


Figure 2.1: Asynchronous handshaking protocols

At this point we have to address the difference between 2-phase and 4-phase protocols, which is also shown in Figure 2.1. In the former case, every transition of *req* and *ack* conveys actual information. Hence, every handshaking cycle (labeled "Event 0-3" in the

---

[1] This explanation assumes *push* channels. For *pull* channels the meaning of the request and acknowledgment signals are reversed, see [Spa20] for a more detailed discussion.

figure) consists of two transitions. 4-phase protocols, on the other side, always entail a reset phase where both signals return to zero again. This is also the reason why we will use the terms return-to-zero (RZ) and non-return-to-zero (NRZ) protocols, respectively.

In the following subsections we will discuss the two main classes of asynchronous protocols, namely BD and DI protocols. Note that there is also the option to let the source and sink communicate over just a single wire, which has been explored by Sutherland and Fairbanks in [SF01]. However, since this approach is not used in this work, we won't go into further detail on that.

2.1.1 Bundled Data Protocols

Note that there is an immanent race condition between the request signal and the data that is being transmitted. It must be guaranteed that the request reaches the sink only after the data is stable at its input. In the so called BD approach this is usually accomplished with a delay element in the request path. This requirement is not dissimilar from the setup-constraint in synchronous design and it has the same drawback, namely the need to know a bound for the propagation delay of the data path.

The advantage of this approach is that the transmitted data itself, does not need to be handled or encoded in any special way. In fact the data path of a BD circuit can be implemented exactly the same way as for a synchronous design. Moreover, when an RZ protocol is used typically only the handshaking signals (i.e., *req* and *ack*) go through the four phases (see Figure 2.1). The data can directly transition from one data word to the next.

The term BD stems from the fact that the data is *bundled* with the request and acknowledgment signals.

2.1.2 Delay-Insensitive Protocols

The request mechanism does not need to be implemented as a dedicated signal. Another possibility is to implicitly encode the request into the transmitted data. It is then 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 the component that performs this task is called a CD. It is only possible if the code used to encode the data has certain properties. Of course this encoding and the requirement for completion detection cause a certain overhead. However, it has the advantage that the communication is DI, i.e., the transitions on the individual wires (also referred to as rails) of a DI link may arrive in any order and there is no race condition between data and request (as with the BD approach).

DI communication can also be implemented in a 2- or 4-phase scheme. In RZ protocols 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 represented by logic zeros on all rails. Possible choices for the data encoding for 4-phase DI protocols are, e.g.,

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

constant-weight (m-of-n) or Berger codes [Ver88]. Further details on these codes will be presented in Section 4.1. In this section we will only use the dual-rail code, which technically also falls into the constant-weight category. The dual-rail code encodes each bit using two rails (i.e., wires) which we refer to as the true and false rail. Throughout this thesis we will use the point notation to denote the individual rails of a dual-rail signal. Hence, given the dual-rail bit d, $d.T$ refers to its true rail, while $d.F$ refers to its false rails, respectively.

The dual-rail encoding is arguably the most simple DI code. Still, as will be explained in more detail in Section 2.6.2, combinational circuits (adhering to the QDI timing model, see Section 2.2) that operate on dual-rail encoded data easily entail more than double the hardware overhead, compared to a "classic" single-rail implementation of the same function. Circuits operating on more complex codes (with higher information density) that are often not even systematic (in contrast to the dual-rail code) would involve even higher hardware costs. Consequently, the dual-rail code is the only one of practical relevance for circuits that actually process data (in contrast to just transmitting it)[2].

In order to decide whether dual-rail data is complete, the receiver only needs to check if there has been a transition on either the true or false rail of each received dual-rail signal pair. A task that can simply be implemented by an OR or NOR gate. However, more sophisticated codes often need much more complicated CDs, which will be thoroughly discussed in Section 4.4.

Figure 2.2 shows an example transmission of two dual-rail bits d_1 and d_0. The order of the individual rails for the labels of the DI data trace in the figure is $(d_1.T, d_1.F), (d_0.T, d_0.F)$. The transmission starts out with a spacer, i.e., all data rails are zero. Then, the DI data bus transitions to the code word $\mathbf{c}_n$, which encodes a logic zero for both data bits (data phase). This condition is detected by the CD of the receiver and a rising transition on the *ack* signal is issued. After the acknowledgment the data rails return to the spacer again, which is in turn acknowledged by a falling transition on the *ack* signal. Then, another code word ($\mathbf{c}_{n+1}$) is transmitted, which encodes the logic value $d_1 d_0 = 01$.

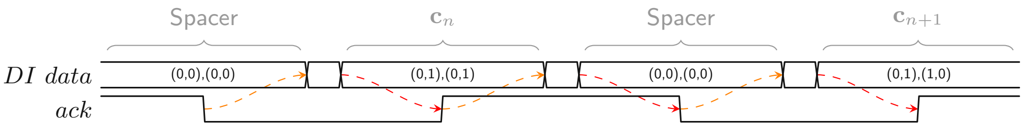


Figure 2.2: RZ (4-phase) DI protocol

Note that a 4-phase protocol can also be implemented using a return-to-one scheme, where the spacer is implemented with logic ones on all wires. This approach is used by [MOPC13] to reduce energy consumption. To improve security by increasing a circuit's robustness against power analysis attacks, [SMBY05] proposes to dynamically switch between the all-one and all-zero spacer (either strictly alternating or randomly). The

---

[2] Despite that, there is some research that also investigates other codes, e.g., [LG01] where a 1-of-4 code has been used.

authors further show that using different spacers for different circuit parts in a fine-grained manner, can also allow for area and throughput optimizations. To this end, [MBSC18] follows a similar strategy, although with a different gate-level logic style. However, strictly speaking, all of these techniques still use a 4-phase protocol.

For NRZ protocols, level or transition encoding can be used. With level-encoded protocols the currently transmitted value can directly be derived from the state of the DI bus. The Level-Endcoded Dual-Rail (LEDR) [DWD91] and Level-Encoded Transition Signaling (LETS) [MAMN08] protocols are examples for such a strategy.

For transition encoding every 4-phase DI code can be used. However, here the information is only contained in wire transition events (no matter the direction), the actual DI bus state is only meaningful when compared to the previous state. Hence, the actual transmitted code word can only be obtained by performing a bit-wise XOR between the current bit pattern on the bus and the previous one. Figure 2.3 visualizes this approach. Again two dual-rail bits with the binary values d_1d_0 of 00, 01 and 10 are transmitted. The DI data bus starts with logic zero on all rails. Since the figure does not show the bus state prior to this value, we don't know which code word is being transmitted by it. After that the three data values are sent. Notice that there are no spacer phases where the data rails and the *ack* signal have to return to a known ground state. This has the obvious benefit of needing fewer bus transitions to transmit the same information when compared to 4-phase protocols. However, as will be shown in Chapter 4 there is significant area overhead associated with actual hardware implementations of this protocol. For example, the CD for the individual dual-rail signal pairs can no longer be implemented with an OR gate – now an XOR gate is required.

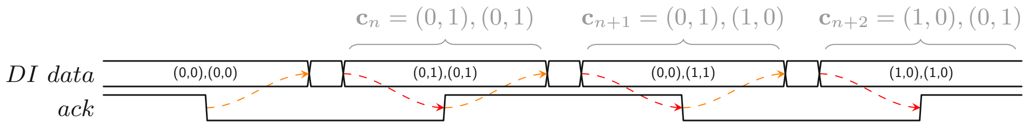

$\mathbf{c}_n = (0,1),(0,1)$
$\mathbf{c}_{n+1} = (0,1),(1,0)$
$\mathbf{c}_{n+2} = (1,0),(0,1)$
DI data
ack
(0,0),(0,0)
(0,1),(0,1)
(0,0),(1,1)
(1,0),(1,0)

Figure 2.3: NRZ (2-phase) transition signaling DI protocol

In general it can be summarized that, for circuits that actually process data, practically only the RZ dual-rail protocol is relevant. All other codes and protocols find their application areas mainly in the transmission of data as will be discussed in Chapter 4.

2.2 Delay Models

As we will see in the following sections the design space for asynchronous circuits is quite large and diverse. One important classification aspect of asynchronous circuits or design styles are the imposed delay assumptions.

In this categorization the class of DI circuits uses the weakest timing assumptions. Here gate and wire delays can be completely arbitrary, the only restriction is that they have to be positive and finite. Consider the example circuit snippet shown in Figure 2.4.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Following the DI model, all gate delays Δ_A to Δ_C as well as the wire delays δ_1 to δ_3 may assume any positive value and the circuit would still be guaranteed to work correctly. However, as shown by Martin [Mar90], this class of circuits is very limited, as the selection of gates to construct such circuits is restricted to inverters and C gates (see Section 2.3.1).

To overcome these limitations, *isochronic forks* [vB92] are introduced, yielding the class of QDI circuits. While this delay model still does not impose any restrictions on gate delays, it demands that (some selected) wire forks must be *isochronic*, i.e., both signal paths after the fork must have the same delay. For the wire fork in Figure 2.4 this means that $\delta_2 = \delta_3$ must hold. With this modification of the DI timing model, arbitrary circuits can be constructed. More detailed discussions of isochronic forks can be found in [Mar90, MM15]. However, as will be discussed in more detail below, in practice strict isochronicity is often not really required.

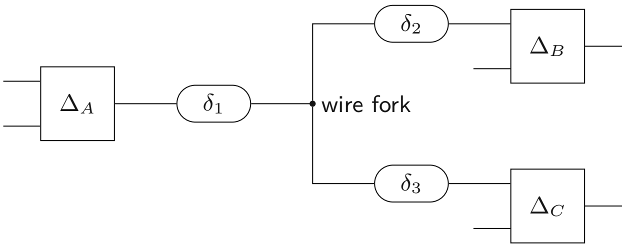


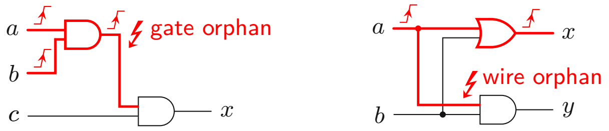
Figure 2.4: Asynchronous circuit model

For the class of speed-independent (SI) circuits [Mul59], 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 for modern technology nodes. However, if all wire forks in a circuit are regarded as isochronic, it is possible to merge the wire delays with the gate delays . For our example in Figure 2.4 this would add δ_1 and δ_2 to Δ_A. Hence, an SI circuit is simply a QDI circuit where *all* forks are isochronic. There is also an interesting relation to DI circuits. A circuit is DI if a transformed circuit, where all wires have been replaced with buffers (after the fork points), is SI.

The models presented so far are precise (mathematical) concepts and apply to the gate-level of asynchronous circuits. Self-timed (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 ensure 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.

Note that an asynchronous design may apply different delay models for different parts of the design. As we will see in Section 2.5, the control logic for BD circuits is often SI or even DI, while the data path relies on timing assumptions enforced by delay elements making it ST.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



(a) Gate orphan (b) Wire orphan

Figure 2.5: Examples for gate and wire orphans

Finally, we need to address the issue of gate and wire *orphans* and its relation to the presented delay models. From the definitions above we know that a circuit is considered DI, QDI or SI iff it operates correctly, despite of arbitrary gate (and wire) delays. Since under this condition, it is not possible to make any assumptions about the time it takes for a circuit (falling into one of these three categories) to process a set of input values, it must be possible to determine that the circuit is done processing by just observing its primary outputs. If the environment of a circuit is not able to unambiguously discern this situation, the next set of input values applied to the circuit may interfere with the previous one leading to incorrect behavior. An orphan transition is a transition that happens on some node (gate or wire) inside a circuit for some input pattern without having *any* influence on the primary outputs of the circuit (or sub-circuit). In other words it is an unobserved internal circuit transition. The unbounded delay model for gates (and wires) makes it impossible to make any assumptions about such events. Since the environment of a circuit eventually creates new input transitions in response to the observed output transitions, it is possible that an orphan interferes with actions inside a circuit caused by the new input transitions at unpredictable points in time. This can cause the circuit to produce invalid results or even deadlock.

We say that such a circuit contains gate or wire orphans, depending on where this transition can appear. Figure 2.5 demonstrates the difference between gate and wire orphans using two simple examples.

A gate orphan, as shown in Figure 2.5a, is produced by a gate G that changes its output value for some particular input pattern (i.e., $a = b = 1$, $c = 0$), but the output of G does not have any impact on the primary (observable) outputs of the circuit (i.e., x), because it is masked by some other gate. If a circuit is operated according to its specification and still contains gate orphans it cannot be considered DI, QDI or even SI.

A wire orphan is an unobserved transition at some wire originating at a fork, as illustrated in Figure 2.5b. To show how a non-isochronic fork may lead to undesirable circuit behavior, let's examine the this example circuit a little more closely. For that purpose, consider the scenario shown in Figure 2.6. The figure shows how the state of the circuit from Figure 2.5b evolves over time, if we assume that the fork at the input a is non-isochronic. This is indicated by the delay element in the signal path going to the input of the AND gate. Initially (State 1) all inputs and output as well as all internal nodes of the circuit are low. Then, an input transition is applied to input a, which will set the output x driven by

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

the OR gate (State 2). However, because of the long delay on the path to the AND gate, the transition will still be on its way to its destination when the environment already deasserted the input a again (State 3), constituting a wire orphan. The deassertion of a is correctly reflected by the deassertion of the output x, which in turn causes the environment to assert input b (State 4). However, because of the non-isochronicity of the fork the falling transition on input a still has not reached the AND gates, which means that the single input transition at b erroneously sets *both* outputs x and y.

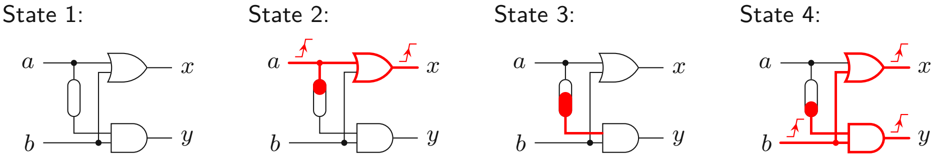


Figure 2.6: Possible effect of a non-isochronic wire fork

This example shows how wire orphans can lead to problems in circuits if the delays on wire forks are not considered correctly. However, it also shows that there is not always the need for strict isochronicity. To fix the issue in the scenario in Figure 2.6, it is sufficient to demand that the delay (i.e., the skew between both paths after the fork) must be less than the time it takes the environment to switch the inputs from $(a, b) = (1, 0)$ to $(0, 0)$ and then to $(0, 1)$ on top of the input-to-output delay caused by the OR gate. This condition is far easier to satisfy with an appropriate circuit layout than a “real” isochronic fork would be.

By definition QDI and SI circuits cannot contain wire orphans, since the isochronic fork assumption prevents this. However, for the DI model, wire orphans cannot be ruled out, and must be accounted for in the design.

We can conclude, that the presence for the potential of an orphan transition (gate or wire) automatically invalidates the DI, QDI or SI property of a circuit, as such a circuit would need additional timing assumptions to work correctly in every case.

2.3 Basic Asynchronous Circuit Elements

This section discusses some basic asynchronous circuits, circuit elements and specification methods used throughout the thesis.

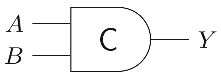
2.3.1 Muller C Gate

An essential gate found in nearly every asynchronous circuit is the (Muller) C gate[3] [Mul59]. Figure 2.7a shows the circuit symbol that will be used throughout this thesis. Here a 2-input version is depicted, but conceptionally C gates can have any number of inputs. There are multiple ways to define the functionality of a C gate. In an informal

---

[3] In literature also the term (Muller) C-Element is used.

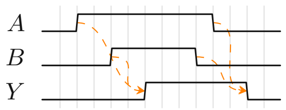
Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.



(a) Circuit symbol

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

(b) Truth table



(c) Timing diagram

Figure 2.7: 2-input Muller C gate

way it might be described as an AND gate with hysteresis or an AND gate for transitions. This fact is also reflected by the circuit symbol.

For a more formal definition, consider Equation (2.1) describing the output of a 2-input C gate and the resulting truth table in Figure 2.7b. Notice that the output Y depends on the inputs A and B as well as the current state of the output itself, which we denoted with Y'.

$$Y = (A \wedge B) \vee (Y' \wedge (A \vee B)) \tag{2.1}$$

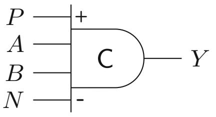
Thus, to set the output of a C gate to one, *all* inputs must be set to one (similarly to an AND gate). However, to reset the output back to zero again, *all* inputs must be set to zero as well. If only one input changes its logical value the output of the gate does not change (“keep” entries in the truth table). Figure 2.7c further illustrates this behavior using a timing diagram.

From this definition it is obvious that in order to implement a C gate some form of internal storage is required to keep track of its current state. There are multiple ways how this can be implemented in Complementary Metal-Oxide-Semiconductor (CMOS) logic [SEE98, MMC14]. The three main variants are referred to as the Martin, Sutherland and Van Berkel style C gate, which each have their own advantages and disadvantages [MOMC12]. The implementation costs for a 2-input C gate range form 8 to 12 transistors, which shows that they can be significantly more expensive than simple AND gates. Although, extending the CMOS C gate implementations to more than two inputs is possible, the resulting circuits quickly become quite large with several transistors in series in p- and n-stack. Hence, C gates with a high number of inputs are often broken up into (multi-level) C gate trees.

The C gate concept can also be extended to derive so called asymmetric C gates. Those gates have additional inputs marked with a plus or minus symbol (see Figure 2.8a). A positive input, i.e., one marked with a plus symbol, is only relevant for rising output transitions. This means that in order for an asymmetric C gate with positive inputs to switch its output to one, all normal as well as the positive inputs must be asserted. To switch back to zero only the normal inputs must be deasserted, the positive input may be kept asserted. Negative inputs have a similar effect for falling output transitions, i.e., an asymmetric C gate can only switch its output to zero if all normal and negative inputs are deasserted.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.



(a) Circuit symbol

A	B	P	N	Y
0	0	don't care	0	0
1	1	1	don't care	1
all remaining cases				keep

(b) Truth table

Figure 2.8: Asymmetric C gate with one positive, one negative two normal inputs

Asymmetric C gates can have any combination of positive, negative and normal inputs. Consider the example in Figure 2.8, showing an asymmetric C gate with one positive input P and one negative input N alongside its two normal inputs A and B. To set its output to one both of the normal inputs as well as the positive asymmetric input P must be asserted. To reset it A, B and N must be deasserted. The behavior can also be expressed using a Boolean formula in Equation (2.2).

$$Y = (A \wedge B \wedge P) \vee (Y' \wedge (A \vee B \vee N)) \tag{2.2}$$

2.3.2 Muller Pipeline

A Muller pipeline (MPL), such as the one shown in Figure 2.9, is a fundamental asynchronous circuit composed of C gates and inverters. Its purpose is to store and transport handshakes from its input (req_{in}, ack_{out}) to its output port (req_{out}, ack_{in}), where each port consists of a request and an acknowledgment signal. In this sense it is also often described as a first in, first out (FIFO) buffer for transitions.

The MPL is completely agnostic to the handshaking protocol (2-phase vs. 4-phase). It is rather just a matter of interpretation, whether the stored transitions constitute 2-phase or 4-phase handshakes. Input handshakes can also be viewed as *tokens* that are fed into and travel through (or get stored in) the pipeline. In Figure 2.9 these tokens travel from left to right.

The operation principle of an MPL is quite straightforward. If the successor and predecessor of some node (i.e., C gate) x_i ($1 \leq i \leq n$) in the pipeline differ in their logic values, x_i (eventually) takes on the value of its predecessor. The predecessor of x_1 is x_0, i.e., the input request req_{in}, while the successor of x_n is x_{n+1}, i.e., the input acknowledgment ack_{in}.

An MPL is considered empty when all its C gates store the same logic value, which must also be matched by input signals x_0 and x_{n+1} ($\forall_{0 \leq i \leq n} x_i = x_{i+1}$). A full pipeline is identified by a strictly alternating pattern on the circuit nodes x_0 to x_{n+1} ($\forall_{0 \leq i \leq n} x_i = \neg x_{i+1}$). Section 3.2 goes into further detail on the possible states of MPLs and also presents more formal definitions for them.

One interesting fact about the MPL is that it is a *completely* DI circuit, which means that it works correctly with arbitrary gate and wire delays. However, the MPL by itself

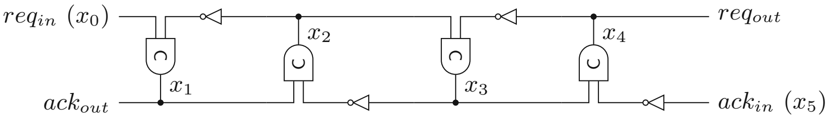


Figure 2.9: 4-stage Muller pipeline

is not yet a very useful circuit. It has nevertheless great significance as it is an often reoccurring structure in asynchronous designs and forms the (basic) control circuit for many pipelines, which will become clear in Section 2.5 and Section 2.6.

2.3.3 Asynchronous State Machines

Asynchronous state machines (also referred to as sequential or control circuits) play an important role in the design of asynchronous circuits. This section only examines SI circuits – so called Huffman circuits [Huf54], which rely on much stricter timing constraints, are not covered here.

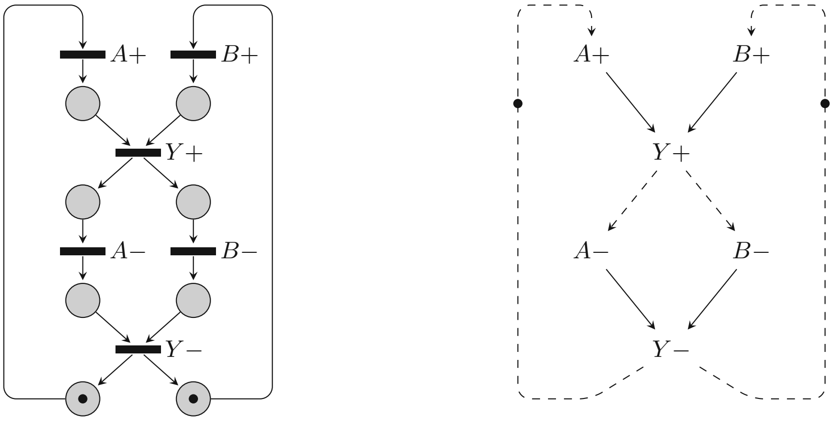
A convenient way to specify such circuits on an abstract level are Petri nets (PNs) [Pet62, Pet77] or more specifically Signal Transition Graphs (STGs) [RY85, Chu87]. These specifications can then be translated into gate-level SI circuits using fully automated software tools. For the STGs presented in this thesis Workcraft is used [PKY09].

STGs are a special type of PNs, a widely used model for distributed and concurrent systems. For the following explanations, consider the example of a PN specification of a Muller C gate. A PN is a directed graph that consists of two types of nodes: places (gray circles) and transitions (black bars). The predecessor nodes of a transition are always place nodes and vice versa, i.e., arcs always connect places and transitions. The PN is "executed" as tokens flow through it. Tokens (black circles) are stored in places. A transition can fire if there is a sufficient number of tokens at all of its inputs. If that is the case the tokens are removed from the input places and placed at the outputs of the transition. The exact number of tokens that are removed from the input places and put into the output places depends on the PN (and is usually specified by a label on the arc). If nothing is specified, as is the case with the C gate example at hand, exactly one token is removed from the input and placed into the output places.

Notice that the timing diagram of the C gate (shown in Figure 2.7c), can directly be translated into the PN.

As already mentioned STGs represent a special subclass of PNs. Here the transitions model real signal transitions and the edges of the graph indicate the causal and temporal order of these events. Signals can either be inputs to the STG or outputs that have to be generated by the STG. Internal signals are also allowed. For a PN to be a valid STG it must be free from deadlocks and the signal transitions for each variable must strictly alternate between rising (+) and falling (-) transitions for every possible execution path. For the purpose of circuit synthesis, it is usually also demanded that an STG has to be

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.



(a) Petri Net

(b) Signal Transition Graph

Figure 2.10: C gate specifications

safe (or one-bounded), which means that there may never be more than one token in a place (all STGs presented in this work are safe) [KKY98]. It must further be guaranteed that once a transition is enabled it must fire, i.e., it may not be disabled again by another signal transition. Another important property is the so called "input free choice". This property demands that, if there are mutually exclusive paths through the graph then the selection which path is actually taken must be handled via mutually exclusive inputs to the STG. An immediate consequence of this constraint is that it is generally not possible to specify a mutex (i.e., a mutual exclusion element) with an STG. However, recently a slight extension has been proposed, which allows such conditions by introducing special mutex states in the STG [SKYL18].

For STGs, such as the one for the C gate shown in Figure 2.10b, places are not drawn if there is only a single incoming and outgoing edge, i.e., every edge can be considered to contain an implicit place. Hence, only for STGs containing choice, i.e., mutually exclusive paths, explicit places are needed. The initial state is indicated by the tokens on the appropriate edges. For further details please refer to [Spa20].

Note that STGs (as well as PNs) also have 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 the environment using dashed lines.

A slightly more complex example for an STG is shown in Figure 2.11. This STG specifies a 3-stage MPL with the inputs req_{in} and ack_{in} and the outputs req_{out} and ack_{out} as well as the three internal signals x_1 to x_3. Using this visualization the operation rule of MPLs discussed in the previous section is clearly visible. Consider, for example the transition x_2+. The graph shows that in order for this transition to happen, the predecessor x_1 must exhibit a rising transition while the successor x_3 must transition to low.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

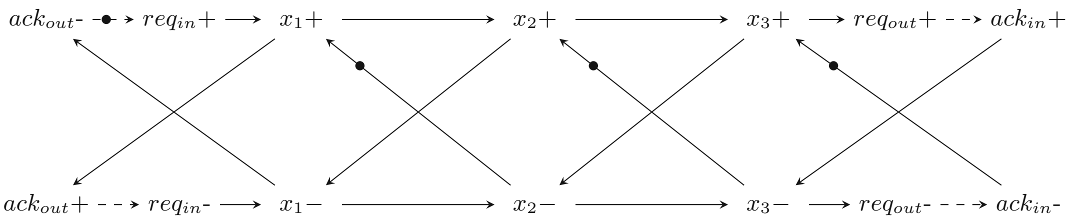


Figure 2.11: STG describing a 3-stage MPL

2.4 Channels and Static Data-Flow Structures

Before we discuss the actual implementation details of the various asynchronous design styles, we first introduce the concepts of channels and Static Data-Flow Stuctures (SDFSs). An asynchronous *channel* refers to a set of signals that are associated with each other and which are jointly used to convey (i.e., transmit) information. This includes data signals as well as their accompanying control signals (like *req* and *ack*). An asynchronous circuit can be viewed as a set of operations performed on such channels. This gives rise to the notion of SDFSs, which can be defined as directed graphs, where the edges are channels and the nodes are operations (such as combinational data transformations, storage/buffers or flow control elements). This representation corresponds to the register-transfer level (RTL) in synchronous circuits and facilitates abstract circuit design, that disregards the actual handshaking protocol (i.e., BD or DI) and the data encoding.

This section only focuses on 4-phase SDFSs (operating on 4-phase channels) since this thesis is mainly concerned with those types of circuits. Although Chapters 3 and 4 also deal with 2-phase channels and interfaces, only linear pipelines without any processing logic are used (hence, no need for SDFSs). However, with some modifications SDFSs can also be used to describe of 2-phase circuits. For more details we refer to [Spa20].

2.4.1 Structure

As already mentioned SDFSs consist of different types of operation nodes that are shown in Figure 2.12. The notation introduced with this figure will be used throughout this thesis and is in accordance with [Spa20].

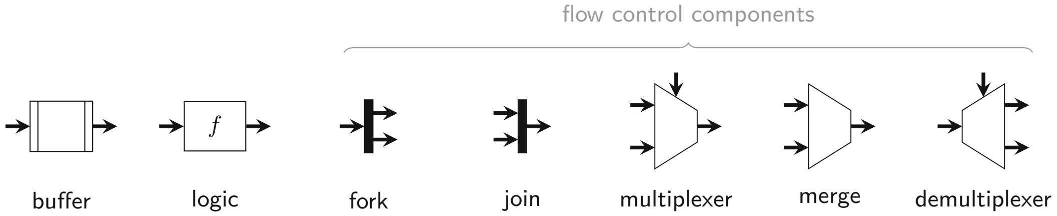


Figure 2.12: SDFS operations

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

The buffer (or latch) has a single input and output channel and stores data items, like registers in synchronous circuits do. These data items (conveyed by the channels) are referred to as tokens and will be explained in more detail in Section 2.4.2. A function or logic block applies a (Boolean) operation to the token on its (single) input channel and relays it to its (single) output channel. Using these two components it is already possible to design linear pipelines. However, for many practical applications feedback paths are required. Hence, we also need flow control components like fork, join, multiplexers, merge and demultiplexers[4].

A fork takes a single input channel and splits it up into multiple output channels, each carrying either all the data of the input channel or only parts of it. The join performs the opposite operation, taking multiple input channels and combining them into a single output channel. The multiplexer also has multiple input channels, but only relays the token of one of them to its single output channel, depending on the token received on the control channel (top). A similar operation is performed by the merge component. However, here it must be guaranteed that there is only one token on one of the input channels (making the control input unnecessary). Finally, the demultiplexer takes an input token and relays it to a specific output channel, selected by the control input.

2.4.2 Token Flow

An SDFS models how data is processed by an asynchronous circuit. For that purpose, data items are represented as *tokens*, which flow through the system in accordance with a set of rules, called *token game semantic*. Note that multiple different rule sets exist, which may lead to different results when applied to the same graph [SPY07]. However, for the circuits discussed in this thesis there is no ambiguity, even under different token game semantics. Hence, we stick to an intuitive (informal) definition (refer to [SPY07] for strictly formal definitions of various token game semantics).

Since the behavior of 4-phase circuits is modeled, two types of tokens must be distinguished, namely data and empty tokens. Tokens can only be stored in buffers. In order for a token to enter and get stored in a buffer, the buffer must not already contain another token, i.e., it must be empty. To represent this absence of tokens the concept of *bubbles* is used. Whenever a token leaves a buffer it leaves behind such a bubble. Hence, we again have to distinguish between data and empty bubbles. Notice that those four types of flow elements (i.e., data tokens, empty tokens, data bubbles and empty bubbles) directly correspond to the four phases of the handshaking protocol. Asserting the request on a channel creates a data token, which becomes a data bubble when the associated acknowledgment is asserted. Following that event the request is deasserted again, creating an empty token that is eventually acknowledged to become an empty bubble. Thus, a bubble can be viewed as an old (already acknowledged and thus consumed) copy of the respective token.

---

[4] [Spa20] additionally defines the mutex and the arbiter component. However, since those elements are not used in the work, we don't cover them here.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

A data token can only move forward to a buffer that contains an empty bubble, while an empty token can only enter a buffer that contains a data bubble. Hence, as tokens move forward through a circuit bubbles move backwards. Figure 2.13 shows how a single data token moves in a pipeline ring consisting of three buffers[5].

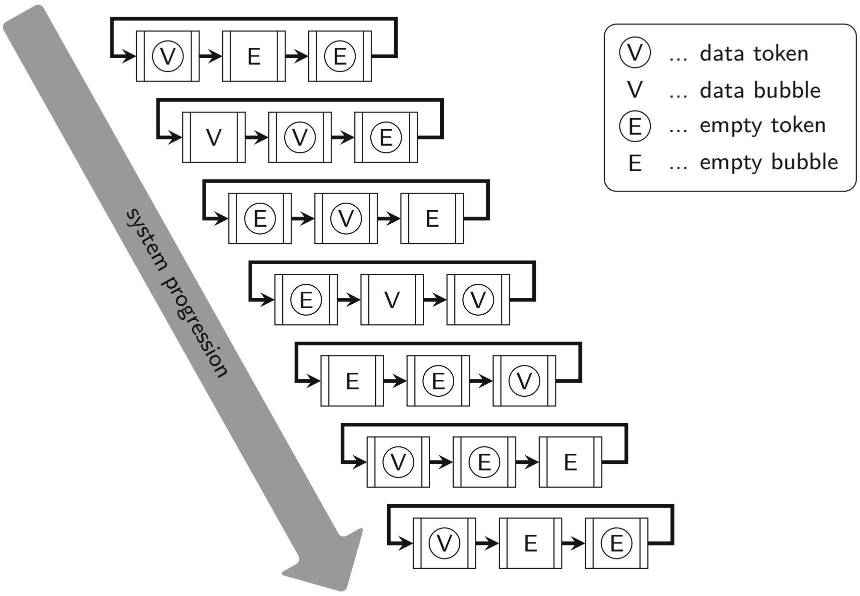


Figure 2.13: Token progression in a 4-phase 3-stage pipeline ring SDFS

There are a few things to note about this example. It demonstrates that the minimal number of buffers for ring structures in (4-phase) SDFSs is three. Otherwise no token progression would be possible and the system would deadlock. Furthermore, each ring must contain at least one bubble, because again otherwise no token movement is possible. In fact, the performance of asynchronous circuits heavily depends on the number of bubbles, or more specifically the ratio of tokens and bubbles in the circuit. This means that sometimes it can be beneficial for the circuit performance to add additional buffers to facilitate a more efficient token flow. Also note that in a 4-phase pipeline that is completely full, only every second stage contains data tokens, since the other stages must hold empty tokens.

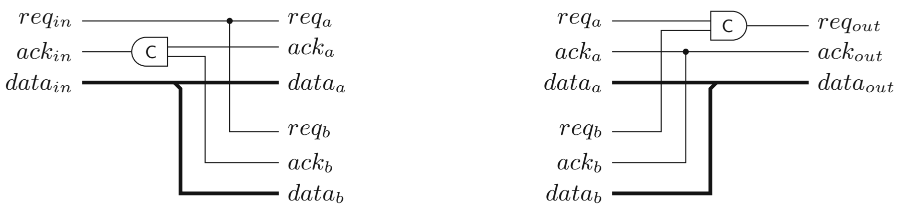
In contrast to buffers, flow control components don't store tokens but simply relay them from their input to their output channels. Presented with a token, a fork creates multiple output tokens. To implement this in hardware for a 4-phase BD channel the request signal is forked (see Figure 2.14a). The acknowledgment signals need to be joined using a C gate, since the fork must only generate an acknowledgment on its input channel when it has received acknowledgments on all output channels. For a join to generate an output token, tokens on all input channels must be present. Hence, a C gate is used to

---

[5] We use the letter V to denote data tokens/bubbles, in order to be consistent with the notation used in [Spa20], where data tokens (bubbles) are also referred to as "valid" tokens (bubbles).

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

join the individual request signals to generate the request for the output channel. The acknowledgment on the other hand can simply be relayed to all input channels.



(a) 4-phase fork (b) 4-phase join

Figure 2.14: Fork and join implementations for 4-phase BD channels

The fork and join circuits shown in Figure 2.14 can also be used for DI channels. The only modification that is required is the removal of the request signals, since the request is encoded in the data anyway.

The multiplexer and the demultiplexer operate in a slightly different way. For the multiplexer to produce an output data token, there must be a data token on the control input channel as well as a data token on the input data channel selected by this control token. Data tokens that might be pending on input data channels not selected by the control token are simply not relayed further or interacted with by the circuit. Such a token keeps sitting at the input data channel until an appropriate control token arrives. After the data token passed the multiplexer, an empty token must be provided by the same data input as well as the control input to complete the handshaking cycle on both of these channels. For the demultiplexer, the situation is similar. The data token at the input data channel is relayed to the output channel selected by the control channel token. After that, the following empty token on the data input channel will be relayed to the same output channel. Circuit implementations for these components and some special variations of them can be found in [Spa20].

Sections 2.5 and 2.6 will show how to implement buffers and pipelines with BD and QDI design styles. For the latter, also the implementation of function blocks will be discussed, since due to the DI encoding of the data it is not possible to simply use conventional combinational logic as is the case with BD circuits.

2.4.3 Iterative Multiplier Example

Figure 2.15 shows a slightly larger example for an SDFS specifying a multiplier circuit that uses an iterative approach to process the inputs a and b and produces the result z. The figure shows the basic flow control elements (fork, multiplexer and demultiplexer) as well as combinational blocks and buffers.

The circuit basically consists of two ring structures formed by the buffers L_3, L_2, L_1 and L_3, L_5, L_4. The lower ring contains the actual data path, while the upper one contains

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

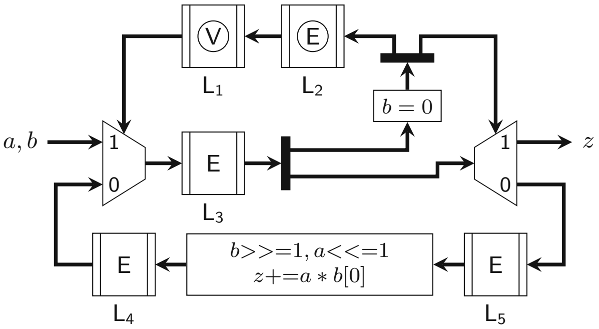


Figure 2.15: SDFS of an iterative multiplier circuit

the control logic for the multiplexer and demultiplexer. New tokens enter the circuit via the multiplexer at the input and then "rotate" in the circuit until the end condition $b = 0$ is reached. This then leads to the result token in L_3 being relayed to the output channel via the demultiplexer and enables a new token to enter the circuit. In [Spa20] this circuit would be identified as a typical application of the "while" template. Additional basic circuit templates for "for" and "if" constructs can also be found there.

2.5 Bundled Data Circuits

Over the years many different asynchronous design styles have been proposed. One of the most important and influential circuits in this context is certainly the MPL, which has already been covered in Section 2.3.2. The MPL can be utilized as control logic for the storage elements in BD circuits. Figure 2.16 demonstrates how this approach can be applied to construct a pipeline. The actual storage elements used in this circuit are D latches. The outputs of the C gates (comprising the MPL) are used to control whether the latches in a pipeline stage are opaque or transparent. In this configuration a 4-phase protocol has to be used to operate the pipeline. The figure shows two delay elements δ_A and δ_B that delay the request signal going from one pipeline stage to the next. These delays must be matched to the respective data path, i.e., the combinational logic in between the pipeline latches. For performance reasons the combinational delays of the stages should be balanced, because the slowest stage limits the throughput of the whole pipeline.

Note that, as already discussed in Section 2.4, in a full pipeline only every other stage contains actual data tokens. We can also see that here, when we look at the alternating pattern stored in the C gates of a full MPL. The pattern leads to only every other latch in the pipeline being opaque, i.e., storing a data token, the rest of the latches are transparent.

To operate an MPL-controlled BD circuit using a 2-phase handshaking protocol, different storage elements have to be used. For that purpose, Sutherland [Sut89] proposed so

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

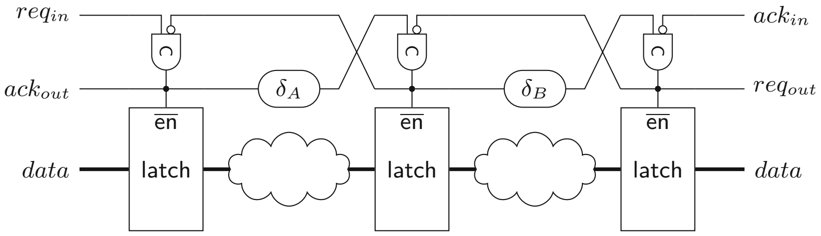


Figure 2.16: 4-phase BD pipeline with MPL as handshake control circuit

called capture/pass latches. Another possibility are special double-edge triggered D flip-flops [YBA96].

The C gates in the MPL can be viewed as the simplest form of a 4-phase latch controller circuit which allow for little parallelism between the handshakes on the input and the output of a stage. Moreover, using it prevents the full utilization of the storage elements in a pipeline. To address this issue [FD96] describes advanced 4-phase latch controller circuits with different degrees of independence between input and output channels. A similar approach is also presented in [VSB10].

Another important and influential BD style, proposed by Singh and Nowick, is the MOUSETRAP pipeline [SN07]. This approach does not rely on C gates but instead uses a single XNOR gate and a D latch to control the pipeline's storage elements (D latches). MOUSETRAP operates using a 2-phase protocol and yields high-performance circuits.

Regarding 2-phase circuits, another notable design approach are so called Click elements [PtBdWM10]. Click-element-based pipelines use D flip-flops as storage elements. The pipeline stage controllers only use standard gates and a D flip-flop to store the current protocol phase, C gates are not required. This facilitates the integration of scan-chains, which is not so easy for, e.g., MPL- or MOUSETRAP-based circuits.

2.6 QDI Circuits

This section covers 4-phase QDI pipeline and logic styles. Since practically 2-phase circuits are not really used for building data processing circuits, they are not addressed here. However, Section 4.5 will show how to construct pipelines capable of transporting data using 2-phase DI protocols.

The QDI circuits used and covered by the work in this thesis, are mainly based on the Weak-Conditioned Half Buffer (WCHB) (Section 2.6.1) and the logic styles covered in Section 2.6.2. However, for the sake of completeness Section 2.6.3 briefly addresses asynchronous QDI design templates based on dual-rail domino logic.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

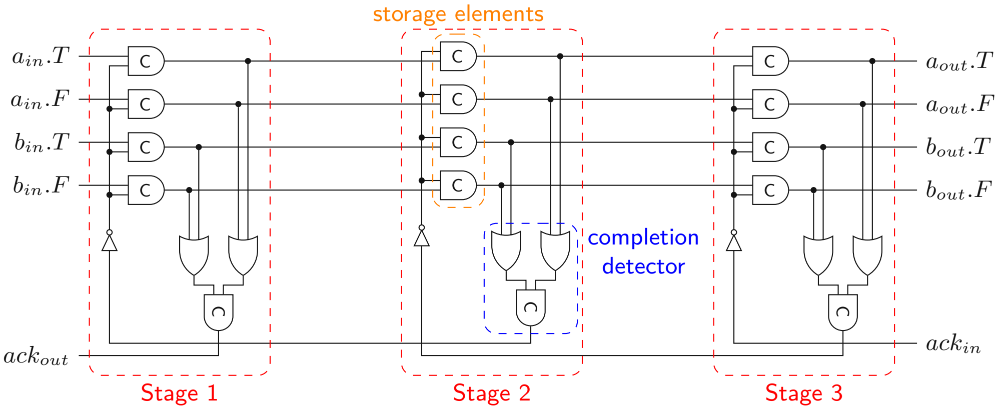


Figure 2.17: 3-stage WCHB pipeline

2.6.1 Weak-Conditioned Half Buffer

The arguably most basic way to construct QDI pipelines is using the WCHB, as demonstrated in Figure 2.17. The figure shows a 3-stage pipeline with a data width of two dual-rail bits (i.e., comprising four data rails). Each buffer consists of two main parts:

- The C gates used as storage elements for the incoming data, which are controlled by the acknowledgment signal of the succeeding stage.
- A CD that monitors the output data signals and informs the preceding stage that a data or spacer token has been received.

The circuit operates in the following way: Assume that initially all storage elements as well as all input signals ($a_{in}.T$, $a_{in}.F$, $b_{in}.T$, $b_{in}.F$, ack_{in}) are zero. This in turn leads to all outputs and internal acknowledgment signals being set to zero as well. Due to the inverters on the acknowledgment paths the buffer C gates (in all three stages) are armed for rising transitions on their input data rails. After input data arrives at a_{in} and b_{in}, *some* of the C gates in the first stage will be switched to one (i.e., exactly one for each dual-rail pair). The CD detects this condition and eventually generates a rising transition at the acknowledgment signal ack_{out} at the input side of the pipeline, indicating to the circuit's environment that the spacer can be applied (i.e., all data rails can be reset to zero again). At the same time the input data transitions also travel through the rest of the pipeline, setting the respective C gates in each stage along the way, until the data appears at the output (a_{out} and b_{out}). When the data fully propagated through the CD in the second stage the acknowledgment to the first stage will be asserted, arming the C gates for falling input transitions, i.e., the spacer. Note that it does not matter whether the falling input transitions on the data rails arrive before or after the C gates are armed.

The example circuit from Figure 2.17 can also be viewed as four MPLs, whose acknowledgment wires are merged and interlocked by the CDs. This perspective visualizes an

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

important point about QDI circuits in general: While there is a clear separation between data and control paths in BD circuits, they are strongly intertwined in QDI circuits, since every data wire is, essentially, also a request signal. From this point of view it is now also easily possible to identify the isochronic forks in this circuit. Interlocking the MPLs introduces a fork in the acknowledgment wire, which now has to fan out to all the buffer C gates of a particular stage. Consider the situation when the buffer C gates of a stage are armed for a data token (i.e., rising input transitions) by deasserting the acknowledgment input. Due to the dual-rail encoding only half of the C gates will actually eventually switch to one. Hence, the signal transitions that arm the C gates that don't switch to one, constitute wire orphans. Introducing an arbitrary delay on one of these signal paths, could lead to a scenario, where the associated C gate switches at a point in time that brings the circuit into an erroneous state. Hence, to guarantee correct behavior of the WCHB, the wire fork in front of the buffer C gates must be considered isochronic. However, in practice it is sufficient to keep the skew between the different paths of this fork within certain bounds in relation to the cycle time of the circuits attached to its input and output channel.

The CD in the example pipeline checks each dual-rail bit for completion separately using an OR gate and then combines the results using a C gate. This approach can be extended to n dual-rail bits using an n-input C gate, which in practice can be implemented as a C gate tree. However, as will be discussed in more detail in Chapter 4 the general structure of the WCHB can be used for arbitrary DI codes. The only thing that needs to changed is the CD.

In literature there exist many variations of the basic WCHB design. One example for such a modification is the relocation of the stages' CDs to in front of the buffer C gates, which offers some performance improvements by introducing a higher amount of parallelism, but compromises on the strict QDI paradigm by introducing additional timing constraints [Smi02, PS13]. Chapter 6 will further present and analyze WCHB variants with improvements regarding their fault tolerance.

2.6.2 Logic Styles

Combinational logic or function blocks for QDI circuits must of course be able to operate on the DI code used for the data encoding. Since practically only the dual-rail code is of relevance for the design of (complex) QDI circuits, this section will only focus on this code.

A QDI logic style must ensure the preservation of the circuit's QDI properties. In particular this means that the last (dual-rail) output of the combinational logic block may only transition to the data phase after *all* inputs are in the data phase. Conversely, the last output must only transition to the spacer phase if all inputs have switched to the spacer. For this protocol to work properly, each rail is allowed to make only one transition (if any) per phase, which naturally forbids the occurrence of glitches anywhere in the circuit. Furthermore, it must be guaranteed that the circuit is free from gate

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

orphans (see Section 2.2). Hence, for combinational QDI circuits it must be possible to determine that the circuit has finished its computation by *only* observing its outputs.

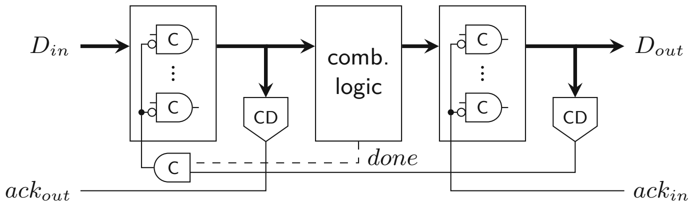
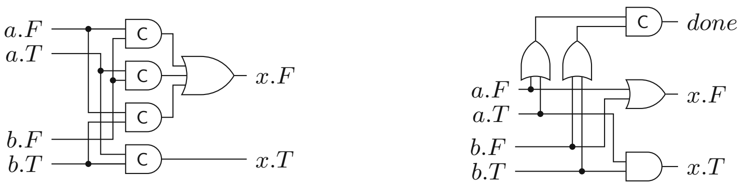


Figure 2.18: WCHB pipeline with combinational logic

If this requirement cannot be satisfied for a circuit, an additional control signal (*done*) may be introduced. The purpose of this output signal is to "collect" all orphan transitions from "hidden" internal circuit nodes, essentially resembling the output of an internal CD. This *done* signal is combined with the acknowledgment signal, such that the circuit can only proceed when the combinational logic has stabilized, i.e., all gates that needed to transition for the particular input data (or the spacer), actually finished transitioning (see Figure 2.18).

One possibility to implement QDI function blocks is the Delay-Insensitive Minterm Synthesis (DIMS) design style [SS93]. DIMS uses an array of C gates to exclusively map every possible (valid) dual-rail input data word to a dedicated internal 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. Figure 2.19a demonstrates this approach for a 2-input dual-rail AND gate. Notice that circuits produced with the DIMS design style don't contain gate orphans, which means that there is no need for an internal CD.



(a) DIMS AND gate

(b) AND gate with input CD

Figure 2.19: QDI dual-rail AND gates

It is apparent that this design style does not scale well with the number of inputs. Thus, complex circuits have to be broken down into a collection of simpler gates with fewer inputs each. Moreover, because of the high usage of C gates, DIMS-based circuits have a relatively high area overhead.

To tackle some of these disadvantages, a more efficient design style was proposed, which is based on special so called threshold gates [Fan05, DS09]. This design style is referred

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

to as Null Convention Logic (NCL). Threshold gates can be viewed as a generalization of C gates. Typically the notation THmn is used to describe a threshold gate with n inputs and a threshold of m. Hence, in order to set the output of such a gate at least m of the n inputs must be asserted. To reset it, *all* inputs must be deasserted again, which shows that there is a similar hysteresis behavior as with C gates. Using this naming convention a THnn threshold gate would refer to a conventional n-input C gate, while TH1n refers to an n-input OR gate. Figure 2.20 shows how an AND gate is implemented using this design style. Note that the lower threshold gate (i.e., TH34W22) has a weight of two associated with two of its inputs, as indicated by the suffix W22 in its name. Therefore, to reach the threshold of this gate (3) it is sufficient that one of the weighted inputs and one of the normal inputs are asserted.

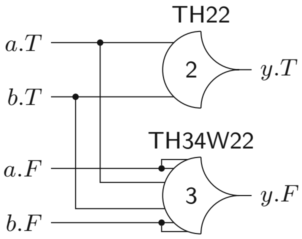


Figure 2.20: NCL AND gates

Compared to the DIMS design style, NCL yields more efficient circuits. However, looking at their CMOS implementations, threshold gates have a quite considerable footprint due to their large stacks and the required storage loop for the hysteresis behavior [SF98]. Hence, as with DIMS, large, potentially slow gates are used directly in the data path.

A different implementation template for combinational circuits that does not rely on C gates (or other state-holding gates) to implement the actual Boolean function was proposed in [DGY92] and further improved in [KL02], where it is called NCLX (NCL with explicit completion detection). However, we will present it here in a slightly different way in order to better fit our explanations and narrative. Figure 2.19b shows an AND gate implemented in this style. As can be seen the output signals $x.T$ and $x.F$ of the gate are generated using an AND and an OR gate. Without further precautions a dual-rail AND gate only comprised of these two gates would violate the QDI properties presented in Section 2.2. In a data phase, where one dual-rail input is false (i.e., its false rail is asserted), the circuit immediately produces a valid output signal (with the OR gate asserting $x.F$) without waiting for the second input to become valid. A similar situation arises in the reset phase where the output may transition to the spacer before both inputs did so. For this reason the circuit in Figure 2.19b also includes the input CD which generates the *done* signal. Here C gates are still required in order to merge the completion signals for the individual dual-rail bits. However, as shown in [KL02] in practice this CD can often be shared with the CD of the preceding pipeline buffer.

Now let's investigate how a circuit consisting of multiple cascaded gates would be implemented. For that purpose, consider the function $y = (a \wedge b) \vee c$. Using the DIMS

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

design style this is quite straightforward to implement. The DIMS AND gate from Figure 2.19a can simply be connected to a DIMS OR gate, no special care must be taken to avoid orphans. Note that in QDI design signal inversions are achieved by swapping the true and false rail of a dual-rail bit, which essentially makes them “free” in terms of area and delay. Hence, by De Morgan’s rule a dual-rail OR gate is equivalent to a dual-rail AND gate with all input and output rails swapped. The given function could also be implemented using a 3-input DIMS gate. However, such a circuit requires eight 3-input C gates, which depending on the target library may not be available and produce a circuit with higher area requirements anyway[6].

Now consider the circuit shown in Figure 2.21. Similar to the CD at the inputs of the AND gate in Figure 2.19b all three inputs of this circuit must be equipped with CDs as well. Since the output of the dual-rail AND gate is not used as a primary output of the overall circuit, a CD is required for this intermediate dual-rail signal.

Note that collecting all orphans to produce the *done* signal usually involves a large tree or cascade of C gates (or a large multi-input C gate, which is not practically feasible). However, this happens concurrently to the actual computation of the output value, which, compared to the DIMS approach, involves fewer gates, resulting in faster circuits. Furthermore, even with the internal CDs the area overhead can still be significantly lower as we will show for some example circuits in Section 5.3. Also, notice that the *done* signal actually propagates upstream, i.e., its destination is the previous pipeline buffer, where it joins with the acknowledgment signal of the succeeding buffer. However, for this acknowledgment signal to transition the data signals have to traverse the buffer C gates as well as the CD of the associated stage. This is obviously not required for the *done* signal, which further reduces the performance criticality of this signal path.

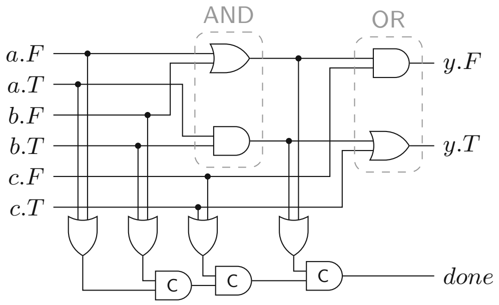


Figure 2.21: NCLX AND-OR structure with 3 inputs and input/internal CDs

Since this is only a basic introduction to QDI dual-rail logic styles, (advanced) optimization techniques [SMBY05, JN07, JN08, DS09, CC13] are not covered here. However, Section 5.3 will demonstrate how to automate the synthesis process of QDI dual-rail circuits using standard EDA tools.

---

[6] Eight 2-input C gates and two 3-input OR gates vs. eight 3-input C gates, one 3-input OR gate and one 5-input OR gate.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Finally, we have to introduce an important concept in QDI design. Multi-output QDI function blocks can be classified as strongly or weakly indicating [Spa20]. A strongly indicating function block only starts switching (any of) its outputs to the data (null) phase when *all* inputs switched to the data (null) phase. Weakly indicating circuits may already change some output data with only a subset of the inputs in a suitable phase. However, in both cases it is guaranteed that the last output switches to the data or null phase only after all inputs switched to the respective phase (since only at this point the CD triggers the change to the next phase at the input).

2.6.3 Dynamic Circuit Templates

Dual-rail domino circuits [WH11] are very fitting for implementing 4-phase QDI circuits as the precharge and evaluate phases of their operation cycle nicely map to the spacer and data phases of the DI RZ protocol.

Figure 2.22 shows a single Prechared Half Buffer (PCHB) (pipeline) stage [BOF10]. The transistor-level circuit on the right shows how the individual domino buffers, each storing a single dual-rail bit, are implemented. To avoid unnecessary additional inverters in the figure the acknowledgment signals in this circuit are low-active (in contrast to the WCHB circuit discussed above).

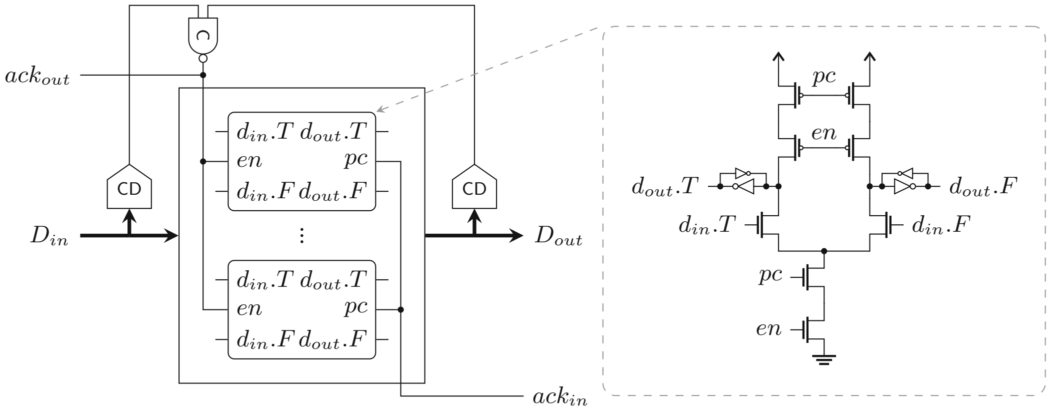


Figure 2.22: PCHB pipeline stage

Assume that all input data rails are zero and that all data outputs of the domino buffers are zero as well. The acknowledgment signals ack_{in} and ack_{out} are one (low-active). Hence, neither the p-stack nor the n-stack of the individual dual-rail buffers are active, i.e., the domino gates are precharged. This means that the (weak-feedback) keepers are responsible to hold the outputs at a stable level. However, both footer transistors (connected to *pc* and *en*) are switched on, priming the gates for the arrival of input data at D_{in}. Eventually one input transition arrives for each dual-rail bit in D_{in}, which cause one of the keepers in each dual-rail buffer to switch its state (evaluation phase). This condition is detected by the input and output CDs (also referred to as left and right CDs), which eventually cause the (controller) C gate to deassert ack_{out}. This event also

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

deasserts the *en* signal, effectively decoupling the domino gates from the input D_{in}, which may now switch back to spacer phase. When the output data at D_{out} is acknowledged by the succeeding logic via the desassertion of ack_{in}, the buffers are precharged again (i.e., the p-stack is now active), which will switch D_{out} to the spacer. The output CD indicates when precharging is complete by deasserting its output. As soon as D_{in} also switched to the spacer the output of the input CD will be deasserted as well, which again causes the assertion of ack_{out} and *en*. Meanwhile the handshake on the output side is completed (assertion of ack_{in}), which brings the whole circuit back into the state from the beginning of this explanation and the whole process can start over.

Notice that, unlike to the WCHB, this circuit needs an additional CD at the input side. Without this CD the circuit would not be able to detect when the input switched to the spacer phase. Since input transitions can only affect the storage loops of the buffers in one direction the output CD is not sufficient to detect the spacer at the input.

Besides the PCHB presented here there also exist other variants such as the Prechared Full Buffer (PCFB), that offer more parallelism between in the input and output channels or reduced hardware overhead. We refer to [BOF10] for more details. To implement combinational logic for this type of asynchronous circuits dual-rail domino gates (with signal keepers) can be used.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

TU WIEN Bibliothek Your knowledge hub

CHAPTER 3

Crossing the Boundary to Asynchrony

This chapter represents a bridge between the synchronous and asynchronous worlds. Modern system on a chip (SoC) designs comprise multiple clock domains and/or a mixture of synchronous and asynchronous styles, i.e., multiple *timing domains*. Hence, by the nature of this design approach, data must be able to cross the boundaries of these domains, such that, different parts of a system can communicate with each other. The interfaces required for such clock and timing domain crossings require specific care in their conception and design, as metastability [KC87] becomes a relevant issue there, and they also tend to constitute significant performance bottlenecks.

Consider Figure 3.1 that shows a very basic way to equip a synchronous system with an asynchronous (BD) output channel. The request signal can simply be generated synchronously by some synchronous state machine. However, when reading the acknowledgment signal (ack) care must be taken. Because of the asynchronous nature of the interface, this signal may transition at any point and is, hence, not synchronized to the clock the synchronous system is operated with (clk). Sampling it at an inauspicious point in time may lead to metastability being introduced into the synchronous system. This can have unwanted effects inside the circuit and can ultimately cause the whole system to fail. Hence, a synchronizer must be employed to reduce the risk of metastability propagating to the system. The downside of incorporating a synchronizer at this point in the circuit is that it introduces additional latency and reduces the overall throughput of the interface, since a single data transfer is always prolonged by the synchronizer latency.

For a synchronous system that has an asynchronous input channel a similar situation arises with the request signal, where a synchronizer is necessary as well.

Hence, for performance critical interfaces a more elaborate data transfer strategy must be found, which (i) avoids metastability by utilizing suitable synchronizers and (ii) is

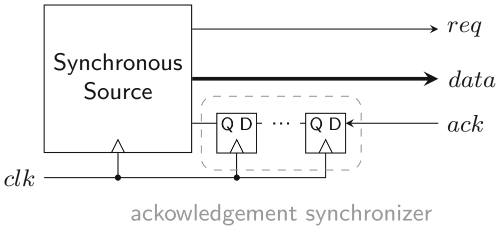


Figure 3.1: Synchronous system with asynchronous output channel

able to hide the latency of said synchronizers such that they have no or very little impact on the overall data throughput. To satisfy those constraints a natural choice is the FIFO buffer (or short just FIFO). These buffers are able to sustain throughput in spite of synchronization delays and facilitate efficient data transfer even in the face of fluctuations in the relative transfer speeds of sender and receiver.

In this chapter we present rationale, design and analysis of two building blocks that are fundamental for timing domain crossing: a synchronous/asynchronous interface that allows to control data transfer from a synchronous domain into an asynchronous one, and an asynchronous/synchronous interface for the reverse direction. Figure 3.2 shows an overview of the respective interfaces considered in this chapter. Besides the actual data lines, the synchronous write interface consists of two control-flow signals. The *write* signal must be asserted (for one clock cycle) to push one data item into the asynchronous domain. The *full* signal indicates that it is not able to process more data items. Thus, if *full* is active, *write* must not be asserted. Similarly, the *empty* signal of the synchronous read port indicates whether data can be read from the interface. Hence, the *read* signal must only be activated if *empty* is deasserted. For each cycle in which *read* is asserted the interface outputs one data item in the next clock cycle. The *full* signal basically implements a back-pressure mechanism, while the *empty* signal constitutes a synchronous request signal. Generating these two flow-control signals typically represents the most challenging function in a timing domain crossing. Our key idea is to employ an MPL (see Section 2.3.2) for transferring the information relevant for flow-control across the domain border.

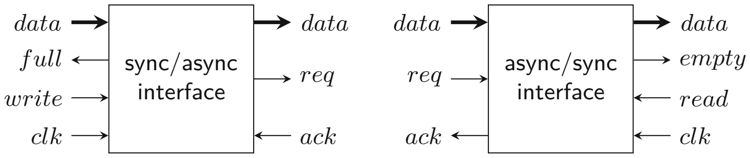


Figure 3.2: FIFO interfaces

The asynchronous interfaces are basically BD channels (we only consider 2-phase push channels in this chapter) where flow-control is given by the asynchronous handshaking protocol. For the remainder of this work we refer to FIFO architectures that implement synchronous to asynchronous interfaces (and vice versa) as A/S FIFOs.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

The chapter is structured as follows: Section 3.1 briefly reviews some existing work on the topic. Next, Section 3.2 demonstrates how to reliably obtain the fill level of an MPL through sampling its internal states. Following an outline of our basic approach given in Section 3.3, the key components of our solution, namely the Desynchronizer and the Resynchronizer, are elaborated in Sections 3.4 and 3.5, respectively, along with an analysis on timing and metastability issues. An experimental evaluation of the design is given in Section 3.6, before Section 3.7 concludes the chapter.

3.1 Related Work

Later on in this chapter we will combine our modules to form a bisynchronous FIFO. While this serves as a compact illustration of our approach, it is not its primary application. There are many other bisynchronous FIFO applications around [AG17], and we do not claim that ours is better than those, albeit it may be beneficial in specific settings. We present the related work about bisynchronous FIFOs here, since they face similar problems as we have for our A/S FIFOs, namely synchronizing information about the read and write pointers for the FIFO memory across clock domains, such that the flow-control signals ($full$ and *empty*) can be generated. If a parallel synchronizer is used to directly move these pointers, a special encoding is required such that the respective other side receives a consistent image of the actual value of the pointer. For this purpose Gray code pointers [AYM^{+}07] or one-hot (unary) pointers [PG07] can be employed. Another quite elegant way is the even/odd synchronizer [FMRM10]. Here a phase prediction between the read and write clocks is used to avoid long synchronizer chains, which contribute to the overall latency of the FIFO.

In the approach proposed by Keller et al. [KFK15] the FIFO uses pausible clocks and mutexes to only transmit pointer increment events to the other clock domain, i.e., the actual value of the read/write pointers is never transferred explicitly. This approach is actually somewhat related to the techniques we present in this chapter, because we also only transfer pointer increment information across the timing domain boundaries – in our approach in the form of read tokens. A token-based approach is presented in [CD03]. Our approach differs from that in not requiring mutex elements, and not assuming a ripple FIFO for the data path.

As a matter of fact pausible clocks make the problem of timing domain crossing a little easier because these circuits are by their nature much more related to asynchronous circuits, in that metastability can in principle be converted to additional wait times through the use of mutexes (time-safe rather than value-safe [Spa20, Cha84]). This property is for example utilized in the A/S FIFOs proposed in [MTMR02] and [TBV09].

In contrast to that the A/S FIFO proposed in [BV06] uses Gray-encoded read and write pointers. The FIFO architecture presented in [CN04] can be used for bisynchronous and A/S FIFOs. However, this design relies on special, custom-design FIFO cells, that are selected for read and write operation through a token-ring-based structure.

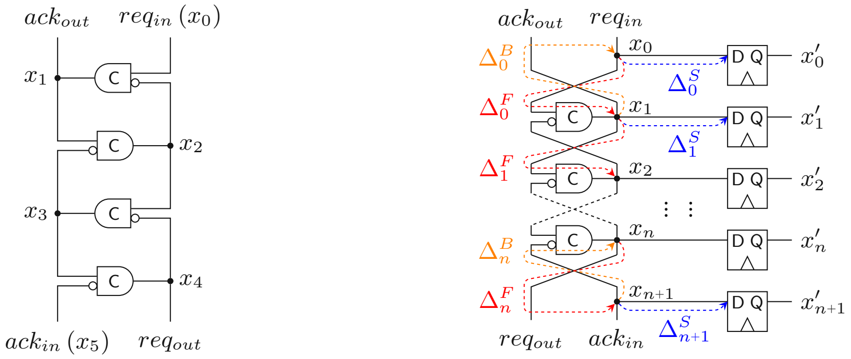
Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

3.2 Sampling Muller Pipelines

This section first clearly defines the state of an MPL and then analyzes how (consistent) state information can be extracted from it in a synchronous timing domain. Although the MPL is agnostic to the handshaking protocol, we only consider the 2-phase protocol here, since this is the protocol used by the circuits proposed in this chapter. Furthermore, we assume that the environment of the MPL will operate the input and output port strictly in accordance with the protocol.

3.2.1 Pipeline States

The state of an n-stage MPL is defined by the $n+2$-element vector $\mathbf{x} = x_0, \ldots, x_{n+1}$. The entries $x_1, \ldots, x_n$ hold the (output) states of all C gates, while input variables x_0 and x_{n+1} hold the logic value of the request at the input port (req_{in}) and the acknowledgment at the output port (ack_{in}). The pipeline outputs ack_{out} and req_{out} are equal to the nodes x_1 and x_n, respectively. Figure 3.3a shows an example pipeline consisting of four C gates, which is, consequently, described by a 6-element state vector $x_0, \ldots, x_5$.



(a) 4-stage MPL

(b) State sampling circuitry

Figure 3.3: MPL circuits

Recall that, MPLs operate according to a simple rule: If the successor and predecessor of some node x_i ($1 \leq i \leq n$) differ in their logic values, x_i (eventually) takes on the value of its predecessor. Note that the nodes x_0 and x_{n+1} are inputs to the pipeline. Thus, their values are controlled by the environment. We define an MPL to be in a *steady state*, when its defining state vector $\mathbf{x}$ does not allow for changes to $x_1, \ldots, x_n$ when applying the pipeline operation rule formulated above. Thus, the only way to get a pipeline out of a steady state is by toggling one of its inputs (x_0 or x_{n+1}). A pipeline that is not in a *steady state* is *transitioning*. In the following we will define and analyze what it takes for a pipeline in a steady state to be regarded as full or empty.

An n-stage pipeline is regarded *empty* iff all elements of its state vector $\mathbf{x}$ have the same logic value, i.e., $x_i = x_{i+1}$. Note that this also includes the inputs: A mismatch between

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

ack_{in} (x_{n+1}) and req_{out} (x_n) indicates that there is still one transition in the pipeline that has not been acknowledged. A mismatch on the input side (i.e., $x_0 = \neg x_1$) indicates a new request that causes a transition to propagate towards the output. More formally, this violates the stability condition from above for x_1 and hence our presumption of a stable state. The only possibility to get an empty pipeline out of this steady state is to externally toggle the req_{in} (x_0) input. Without any read being performed at the output, n complete handshakes can be performed on the input side of an empty n-stage pipeline. After that the input side can once more toggle the input request (x_0), but won't get an acknowledgment unless the output x_{n+1} is toggled.

An n-stage pipeline is *full* iff there is a strictly alternating 0/1 pattern on the state vector $\mathbf{x}$, i.e., $x_i = \neg x_{i+1}$. Again the interfaces are included here: Unless there is an active input request (i.e., $x_0 = \neg x_1$), there is still one "free" position in the pipeline. Likewise, $x_n = x_{n+1}$ would indicate an acknowledged output request, which makes all transitions contained in the pipeline move by one position. This is a dynamic state, as witnessed by the violation of the stability condition for x_n. Assuming no further write access, on a full n-stage MPL $n + 1$ 2-phase read handshakes can be performed before the pipeline becomes empty.

3.2.2 Obtaining State Information

Since an MPL is an event driven circuit, care must be taken when extracting (synchronous) state information from it. Figure 3.3b shows how flip-flops can be used to sample the pipeline state vector $\mathbf{x}$, effectively transferring its information into the synchronous timing domain. To model the state of the pipeline nodes at the synchronous clock events (i.e., when they are sampled), we use the set $\mathbb{B}_T = \{0, 1, \uparrow, \downarrow\}$. Consequently, $\mathbf{x}$ is described as a vector of length $n + 2$ over $\mathbb{B}_T$. The values 0 and 1 indicate the normal logic states for the nodes x_i that were stable during the setup/hold (S/H) window of the associated sampling flip-flop. The value $\uparrow$ ($\downarrow$) indicates that the respective node was transitioning from 0 to 1 (1 to 0) during the S/H window. If a flip-flop samples a transition, it may become metastable. Hence, we use a different set of values $\mathbb{B}_M = \{0, 1, M\}$ to model the outputs $\mathbf{x}'$ of the sampling flip-flops. The values 0 and 1 again have the usual meaning of logic low and high. The value M denotes a metastable state, and can basically be considered as a wild card for arbitrary behavior of a signal, including late transitions or glitches. This meaning is consistent with the notion of metastability used in metastability containing circuits [FFL18]. Using $\mathbb{B}_M$, Boolean equations can be evaluated normally with slightly extended truth tables for the logic operators[1].

The flip-flops in the figure essentially perform the task of a synchronizer, in that they (try to) map the asynchronous signal at their inputs to a stable output value; hence, they must be appropriately extended in practice (see later). The function s (Equation (3.1)) models this operation[2]. It maps a value in $\mathbb{B}_T$ to a set of possible values in $\mathbb{B}_M$, to

---

[1] $\neg M = M$, $0 \wedge M = 0$, $1 \wedge M = M$, $M \wedge M = M$, all other operations can be reduced to the given rules using De Morgan's laws.

[2] In Equations (3.1) and (3.2) $\mathcal{P}(A)$ denotes the power set of set A.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

which the synchronizer might resolve (or fail to resolve, in the case of M). Note that sampling a transition can basically cause an arbitrary output value of the synchronizer. The synchronizer can either settle on the old or the new value of the input (0 or 1), or become metastable itself (M). Which of the values from $\mathbb{B}_M$ the output actually assumes cannot be predicted for a single instance.

$$s : \mathbb{B}_T \mapsto \mathcal{P}(\mathbb{B}_M),\ s(x) = \begin{cases} \{0, 1, M\} & \text{if } x \in \{\uparrow, \downarrow\} \\ \{x\} & \text{otherwise} \end{cases} \tag{3.1}$$

Similarly s_v yields the set of possible values for the a whole vector of length m.

$$s_v : \mathbb{B}_T^m \mapsto \mathcal{P}(\mathbb{B}_M^m),\ s_v(\mathbf{x}) = \{\mathbf{y} \in \mathbb{B}_M^m | y_i \in s(x_i)\} \tag{3.2}$$

Ultimately, our goal when evaluating the pipeline state is to retrieve the number T of tokens present. For the stable pipeline we have argued that T equals the number of state changes between consecutive elements along the state vector. So we need to be able to find all instances of 01 or 10 in $\mathbf{x}'$. Considering that the propagation delays Δ_i^S from the individual pipeline nodes to the respective sampling flip-flops (see Figure 3.3b) may vary, one might experience an inconsistent result when taking a sample while several transitions are "in flight". To safely prevent that, we require that all Δ_i^S $(0 \leq i \leq n+1)$ are equal, i.e., there is no (or negligible) skew between the wires connecting the nodes x_i to the flip-flops.

In the general case, where no timing constraints are imposed on the MPL, every possible vector in $\mathbb{B}_T^{n+2}$ can appear at the input of the sampling flip-flops. Under our sampling model this would make it impossible to infer reliable state information about the pipeline, let alone obtaining an exact token count. Consider, for example, the state vector $\uparrow\uparrow 0$, representing a single token traveling through an initially empty pipeline. Sampling this vector (and assuming metastability is resolved) may result in 000, 010, 100 or 110 for $\mathbf{x}'$, which obviously correspond to vastly different pipeline states and token counts.

To avoid such problems, we constrain the forward and backward delay of the pipeline stages. Let T_S and T_H denote the setup and hold time of the synchronizer/sampling flip-flops as specified in their datasheet. Moreover, Δ_{F_i} and Δ_{B_i} denote the forward and backward delay of stage i in the MPL. For the sake of simplicity we assume equal delays for rising and falling transitions (namely the shorter of both). As indicated in Figure 3.3b we always measure the delays Δ_{F_i} and Δ_{B_i} from and to the nodes x_i. They, hence, incorporate the delay of the C gate as well as the interconnect and the inverter.

$$\forall i, 0 \leq i \leq n, \Delta_i^F > T_S + T_H \wedge \Delta_i^B > T_S + T_H \tag{3.3}$$

The constraint in Equation (3.3) guarantees that it can never be the case that two neighboring flip-flops in the synchronizer sample transitions at the same time. This in turn means that neighboring flip-flops cannot both evaluate to random values (or become metastable) at the same time, i.e., if a node x_i experiences a transition, its

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

neighbors x_{i-1} and x_{i+1} must be stable. Moreover, because of the pipeline operation rule, these two stable nodes must have opposite values, i.e., $x_i = \downarrow \Rightarrow x_{i-1} = 0 \wedge x_{i+1} = 1$ and $x_i = \uparrow \Rightarrow x_{i-1} = 1 \wedge x_{i+1} = 0$ for $1 \leq i \leq n$. For the pipeline inputs node x_0 and x_{n+1} (which obviously don't have a predecessor or successor node, respectively) these rules can be simplified to $x_0 = \downarrow \Rightarrow x_1 = 1$, $x_0 = \uparrow \Rightarrow x_1 = 0$, $x_{n+1} = \downarrow \Rightarrow x_n = 0$ and $x_{n+1} = \uparrow \Rightarrow x_n = 1$. Figure 3.4 illustrates this behavior. Note that any variance of Δ_i^S between different nodes x_i, basically "enlarges" the S/H window and must be added on the right side of the constraint inequality. Clock skew between the sampling flip-flops has the same effect.

In summary, if we assume that metastability is resolved, every token, i.e., every 01 or 10 transition in the state vector $\mathbf{x}'$ can be identified reliably. For the token detection it does not matter to which value metastable flip-flops resolve, since the total number of state changes stays the same in both cases (recall that transitions must be enframed by two opposite stable values). The only exception to this are transitions on the pipeline inputs x_0 and x_{n+1} (e.g., ↑00 can be sampled as 000).

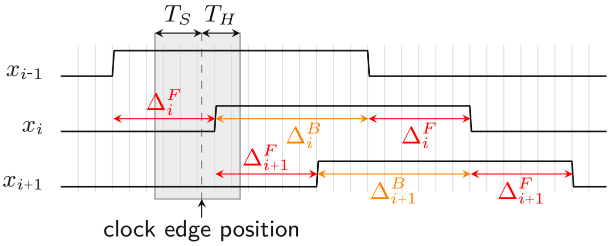


Figure 3.4: Setup/hold window

Now let's reason about the possible states an n-stage pipeline can have under our sampling model. Let $X^m \subset \mathbb{B}_T^m$, $m = n+2$ denote the set of possible values the vector $\mathbf{x}$ can take under the timing constraints formulated above. Starting with the set $\mathbb{B}_T^m$ we can simply remove all states that violate one of the rules formulated above to, hence, obtain X^m.

To construct X^m, for some given m, in a recursive way the following approach can be used: Given X^{m-1} the set X^m is generated by appending a new element to the front of the vectors in X^{m-1} and considering the possible values this new element can take based on the exclusion rules formulated above. From a circuit view this means the original input request x_0 is now driven by a newly added C gate whose inverted input is connected to x_1. The new input request of the extended pipeline is then given by this C gate's non-inverted input. For all $\mathbf{x} \in X^{m-1}$ where we have $x_0 = \uparrow$ ($\downarrow$) a new vector $\mathbf{z}$ is added to X^m, where $z_0 = 1$ (0) and $z_i = x_{i-1}$ for $1 \leq i \leq n+1$. All other symbols for z_0 are forbidden by the exclusion rules. If $\mathbf{x} \in X^{m-1}$ starts with a constant value, three new vectors are added to X^m. The only value that is forbidden for z_0 now is ↑ if $x_0 = 1$ and ↓ if $x_0 = 0$. Equation (3.4) formalizes this construction.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

$$\begin{aligned} X^m = &\{1\mathbf{x} \mid \mathbf{x} \in X^{m\text{-}1} \wedge x_0 = \uparrow\} \cup \{0\mathbf{x} \mid \mathbf{x} \in X^{m\text{-}1} \wedge x_0 = \downarrow\} \cup \\ &\bigcup_{y \in \{0,1,\uparrow\}} \{y\mathbf{x} \mid \mathbf{x} \in X^{m\text{-}1} \wedge x_0 = 0\} \cup \bigcup_{y \in \{0,1,\downarrow\}} \{y\mathbf{x} \mid \mathbf{x} \in X^{m\text{-}1} \wedge x_0 = 1\} \end{aligned} \tag{3.4}$$

From this recursive construction it is now possible to derive a suitable recurrence relation that yields $v_m = |X^m|$, i.e., the number of possible state vectors for a pipeline with n stages (i.e., C gates). Let v_m^C and v_m^T, denote the number of state vectors (for a given vector length m) that start with a constant (i.e., 0 or 1) and transitional (i.e., ↑ or ↓) value at x_0, respectively. Therefore, we have:

$$v_m = v_m^C + v_m^T \tag{3.5}$$

Using the recursive construction of X^m, Equation (3.6) can be established.

$$v_m = 3v_{m\text{-}1}^C + v_{m\text{-}1}^T \tag{3.6}$$

From the discussion above it is also clear that Equation (3.7) must hold, because for all vectors in X^m starting with a constant value in x_0 a single vector starting with either ↑ or ↓ is added to X^{m+1}.

$$v_{m+1}^T = v_m^C \tag{3.7}$$

After some simple transformations (Equation (3.8)), we arrive at the recurrence relation shown in Equation (3.9).

$$\begin{aligned} v_m &= 3v_{m\text{-}1}^C + v_{m\text{-}1}^T && \text{(apply Equation (3.5) to } v_{m\text{-}1}^C\text{)} \\ &= 3(v_{m\text{-}1} - v_{m\text{-}1}^T) + v_{m\text{-}1}^T && \text{(resolve parenthesis)} \\ &= 3v_{m\text{-}1} - 2v_{m\text{-}1}^T && \text{(split off } v_{m\text{-}1}\text{)} \\ &= 2v_{m\text{-}1} + v_{m\text{-}1} - 2v_{m\text{-}1}^T && \text{(apply Equation (3.6) to } v_{m\text{-}1}\text{)} \\ &= 2v_{m\text{-}1} + 3v_{m\text{-}2}^C + v_{m\text{-}2}^T - 2v_{m\text{-}1}^T && \text{(split off } v_{m\text{-}2}^C\text{)} \\ &= 2v_{m\text{-}1} + v_{m\text{-}2}^C + v_{m\text{-}2}^T + 2v_{m\text{-}2}^C - 2v_{m\text{-}1}^T && \text{(apply Equation (3.5) to } v_{m\text{-}2}^C + v_{m\text{-}2}^T\text{)} \\ &= 2v_{m\text{-}1} + v_{m\text{-}2} + 2v_{m\text{-}2}^C - 2v_{m\text{-}1}^T && \text{(apply Equation (3.7) to } v_{m\text{-}2}^C\text{)} \\ &= 2v_{m\text{-}1} + v_{m\text{-}2} + 2v_{m\text{-}1}^T - 2v_{m\text{-}1}^T && \text{(cancel right-most terms)} \\ &= 2v_{m\text{-}1} + v_{m\text{-}2} \end{aligned} \tag{3.8}$$

$$v_m = 2v_{m\text{-}1} + v_{m\text{-}2} \tag{3.9}$$

Now it remains to establish suitable initial values for v_1 and v_0. Since $m = n + 2$, at first glance v_1 and v_0 don't make much sense in the context of an actual pipeline circuit. However, setting $v_0 = 0$ and $v_1 = 4$ yields the value 8 for v_2, which is the expected

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

number of states for a pair of handshake signals (i.e., a “zero-stage” MPL). Similarly, v_3 yields 20, which again matches the expectation for a “pipeline” consisting of a single C gate. For that matter $v_1 = 4$ can be viewed as the number of states a single wire can have.

Solving this linear recurrence relation under the established initial conditions yields Equation (3.10).

$$v_m = \sqrt{2}\left((1+\sqrt{2})^m - (1-\sqrt{2})^m\right) \tag{3.10}$$

This result (Equation (3.10)) is not really needed for the explanations that follow. However, as an interesting corollary of the discussion in this section, it is included for the sake of completeness.

3.3 Proposed Approach

In essence, we propose two basic modules, a synchronous/asynchronous interface which we will call *Desynchronizer* in the following, and an asynchronous/synchronous interface called *Resynchronizer*. These two components form the cornerstones for our A/S FIFO architecture and can, if connected together, also be used to implement a bisynchronous FIFO. We will leverage the knowledge from Section 3.2 about the possible pipeline behaviors under the imposed restrictions for those implementations.

3.3.1 Problems and Requirements

The fundamental problem with timing domain crossing interfaces is to compensate for the lack of a handshake on the synchronous side, by means of $full$ or $empty$ status flags (recall Figure 3.2). The generation of these flags necessarily involves combining status information from both timing domains, which creates the demand for synchronization. A well known instance of this problem is the comparison of read and write pointers in bisynchronous FIFOs outlined in Section 3.1. While the use of synchronizers can easily solve this, synchronizers, in the general case, do create a delay that makes the validity of the status flags questionable: Knowing that the asynchronous domain *was* ready to receive data, e.g., two clock cycles ago, does not imply that this is still the case when this information is received (i.e., after the synchronizer delay). This seems to be a fundamental issue that cannot be eliminated, but rather requires appropriate consideration. In our attempt to provide an efficient approach for the latter, let us pin down the requirements more precisely:

(R1) Validity of the $full$ flag: If the $full$ flag is inactive at any clock cycle, then the asynchronous domain must indeed be able to accept a data item at the next active clock edge.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

(R2) Validity of the *empty* flag: If the *empty* flag is inactive at any clock cycle, then the asynchronous domain must indeed be able to deliver a valid data item at the next active clock edge.

(R3) Maximum Throughput: It shall be possible to move a data stream over the interface at a sustained rate of one data item per clock cycle.

(R1) is essential to prevent losing data through overflows while (R2) prevents reading stale data from an empty FIFO.

While (R1) and (R2) are mandatory for any reasonable timing domain interface to ensure correct data transfer, they do not prevent overly conservative activation of the $full$ and *empty* flags. (R3) is introduced to counterbalance this by also considering performance. Note that our definition of (R3) only applies to the data *throughput*, it does not restrict the *latency*. We chose to do so, since having FIFO applications in mind, the efficient transfer of single data items may not be the primary concern anyway.

3.3.2 Key Solution Principle

The core of our proposed FIFO architectures is an MPL. For each new data item written to the write port of the FIFO, a token is placed into this pipeline. For each of these tokens the read port may perform exactly one read operation on the FIFO, removing the token from the pipeline. In this sense, the MPL is used as an asynchronous up/down counter, representing the fill-level of the FIFO. The tokens themselves don't carry any information, only their total number stored in the pipeline is relevant. The actual data must be stored in some shared memory, and each port of the FIFO must maintain its own memory pointer to it. Since fundamentally the MPL is an asynchronous structure, reading from and writing to it from the asynchronous domain is natural and straightforward; and with $full$ or *empty* flags in place on the synchronous sides for establishing flow-control, maintaining a correct token count remains feasible.

Unlike conventional approaches where the read and write pointer must be related in some way to obtain information on the fill level, the token count in our up/down counter directly reflects the fill level. Consequently, the pointers for the shared memory on the read and write port of the FIFO can be maintained independently and never need to be mapped into the respective other timing domain, as no comparison is required – this is a significant advantage of the up/down counting underlying our approach. For every write operation, one token is added to the MPL and the write pointer is incremented, and upon every read one token is removed and the read pointer incremented. Of course, the pipeline depth must be adjusted to the number of locations in the actual FIFO memory.

Now the remaining challenge is to generate the $full$ and *empty* flag the synchronous sides rely on to comply with (R1) and (R2). To this end, we have to overcome three problems:

First, reading the fill status of an immanently event-driven MPL is non-trivial. We have already elaborated on that in Section 3.2.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Second, the status must be moved across the timing domain boundary from asynchronous (where the MPL is located) to synchronous (where the flag is needed). In principle, the *full* or *empty* flag can be generated on the asynchronous side, in which case only the flag as such needs to cross the timing domain. While this seems conceptually very attractive, we have decided to move the flag generation to the synchronous domain, as this simplifies the implementation as well as eases the reasoning for why the proposed circuits work correctly. Even though, as a consequence, a number of pipeline status signals now need to cross the timing domain, only one of them is actually critical for metastability, as will become clear later on.

Third, as already mentioned, synchronizing an asynchronous event to a fixed clock causes a delay. This delay in obtaining the pipeline state information needs to be appropriately accounted for.

To handle these issues we designed the De- and Resynchronizer circuits. These two components basically evaluate a synchronized state of a certain number of pipeline stages. For the Desynchronizer the sub-pipeline comprises the *first* stages after the circuitry with which the synchronous side places a read token[3] into the pipeline. The obtained information is then used to generate the *full* flag for the synchronous domain. Consequently, the Resynchronizer must look at the *last* few stages to generate the *empty* flag. As a consequence of (R3) the number of pipeline stages that are synchronized must be appropriately matched to the synchronizer delay, which is usually chosen with a specific mean-time between upsets (MTBU) requirement in mind.

Although we consider FIFOs an important application and use one to illustrate our approach, the focus of this chapter will be on the two interface blocks, the Desynchronizer and the Resynchronizer. With these components in place, the remaining parts of a FIFO implementation (memory cells and memory pointers) can each operate in a single timing domain and are, hence, relatively easy to implement. We will give a brief implementation example in Section 3.6.

3.4 Desynchronizer

An overview of the Desynchronizer circuit is shown in Figure 3.5. It can be seen that the single flip-flops from Figure 3.3b have been replaced by n-flip-flop synchronizers, where n can be chosen according to the reliability demands of the circuit, i.e., the desired MTBU. This synchronizer array reads the state vector $\mathbf{x} = x_0, ..., x_{k+1}$ of the first k stages of the MPL used to buffer the read tokens and produces the synchronized state vector $\mathbf{x}'$. We will show that in order to compensate for the synchronizer delay of n cycles, we need to sample exactly n stages of the MPL, i.e., k must be equal to n, which is already indicated by the figure. The bottom-most C gate in the figure is not part of this n-stage (sub-) pipeline; it is, however, relevant for the pipeline timing constraints. Since this C gate

---

[3] We refer to them as *read tokens*, because for each of these tokens one read operation may be performed on the FIFO.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

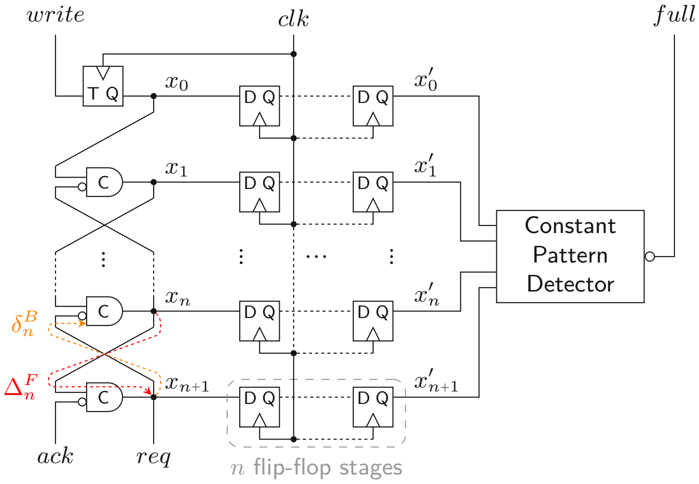


Figure 3.5: Desynchronizer circuit

produces the input acknowledgment x_{n+1} to the n-stage (sub-) pipeline, it determines Δ_n^F as indicated in the figure. It also represents the first C gate in the subsequent pipeline portion that is used to buffer read tokens. Recall that the total pipeline depth (in terms of token capacity) must be equal to the number of memory locations in the FIFO (which may be significantly larger than n).

The synchronous write port uses a toggle flip-flop to place tokens (i.e., 2-phase handshakes on *req* and *ack*) into the pipeline[4]. At every active clock edge where *write* is high the input request to the pipeline (x_0) is inverted, hence, generating a new token. As it is not possible to directly observe the output acknowledgment (x_1) of the pipeline, because of metastability concerns, the input port must assume that (i) whenever *full* is low the pipeline can accept a token and that (ii) this write operation completes within one clock cycle. In order for this design to work properly, we have to impose timing constraints (Equation (3.11)) on the minimum clock period T_c (i.e., the maximum frequency) the synchronous port can be operated with.

$$T_c > t_{CO} + \sum_{0 \le i \le n} \Delta_i^F + \Delta^S + T_S \bigwedge T_c > \max_{0 \le i \le n} \delta_i^B \tag{3.11}$$

Here, t_{co} denotes the clock-to-output delay of the toggle flip-flop and T_S again denotes the setup time of the sample flip-flops. Moreover, δ_i^B denotes the delay from node x_i back to the input of the previous C gate (including the inverter). This is illustrated in the figure for δ_n^B. The constraint ensures, that given an empty pipeline a token that is generated during a write operation must have left the (n-stage) pipeline within one clock cycle (of course assuming that the subsequent logic at the asynchronous side is ready to receive this token). This means that the synchronizer will again sample a constant

---

[4] The proposed principle would also work with 4-phase handshakes. However, this would mean, among other things, that the MPL would need to be twice as long.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

pattern and the $full$ flag will stay deasserted. This is required to sustain a continuous data stream, as demanded by (R3).

Hence, for every clock cycle in which a token is written to the pipeline (and the pipeline does not start to fill up) the state of *all* nodes x_i must toggle and the new C gate states must propagate to the respective sampling flip-flops (first inequality). Furthermore, there must be enough time such that all C gates in the pipeline are again armed for the next (opposite) transition, i.e., the next token (second inequality). If the pipeline starts to fill up, because tokens are not removed fast enough at the asynchronous side, the state vector $\mathbf{x}$ sampled by the synchronizer will be different from the all-zero/all-one vector, which indicate an empty pipeline. In particular the first token that is not able to leave the pipeline within one clock cycle results in a state vector where the value of x_n and x_{n+1} will be different.

After n cycles the sampled status vector $\mathbf{x}$ will have propagated through the synchronizer and appear as $\mathbf{x}'$ at its output. Because of this latency the MPL must be able to absorb another n read tokens without overflowing after the first token that was not able to leave the pipeline within one clock cycle. As initially claimed, we see that $n = k$ must hold for the Desynchronizer to work correctly.

As a consequence $full$ is activated as soon as the pipeline is recognized not being completely empty, i.e., already when it holds a single token. Therefore, it is possible that the $full$ signal is asserted in some cycle c although there were no write accesses for the last $n-1$ cycles. Obviously, this is pessimistic and does not fully utilize the pipeline in every case, but it is safe with respect to (R2) and allows for a very simple, fully combinational implementation of the logic to generate the $full$ flag. Moreover, n will typically be as low as 3, so the sub-pipeline is not very deep. This also means that timing constraint for T_c in Equation (3.11) should not be too difficult to fulfill.

As shown in Figure 3.5 the $full$ signal is driven by the inverted output of the constant pattern detector ($full = \neg cp$), whose implementation is defined by Equation (3.12).

$$cp(\mathbf{x}') = \left(\bigwedge_{0 \le i \le n+1} x'_i \right) \vee \left(\bigwedge_{0 \le i \le n+1} \neg x'_i \right) \tag{3.12}$$

Using this approach it is guaranteed that whenever $full$ is asserted it stays asserted until the pipeline is empty, i.e., there are no tokens left in the circuit that could lead to further transitions on $\mathbf{x}$.

Finally, the risk for metastability on the $full$ signal must be evaluated. To this end we leverage the findings from Section 3.2 and examine all possible pipeline states given by the set X^{n+2}. Note that writes to the MPL are only performed synchronously. Thus, the flip-flop chain at x_0 will never sample a transition, and hence cannot become metastable. This means that for our analysis we don't have to consider those vectors $\mathbf{x} \in X^{n+2}$ where $x_0 \in \{\uparrow, \downarrow\}$. This only leaves two cases for $\mathbf{x}$, that, when sampled and converted to $\mathbf{x}' = s_v(\mathbf{x})$, can lead to an arbitrary value at the $full$ signal (i.e., 0, 1 or M). In particular

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

these are constant patterns with a transition at the last node x_{n+1} ($x_i|_{i\leq n} = 1$, $x_{n+1} =\uparrow$ or $x_i|_{i\leq n} = 0$, $x_{n+1} =\downarrow$). These situations occur when a token is just about to leave the pipeline but didn't quite make it. Hence, in our application it is irrelevant to which stable value the synchronizer resolves the transition. Either the pipeline appears non-empty for another cycle, which is a safe state since writes are not allowed, or it is considered empty, which is also fine because the token was just about to leave the pipeline anyway. Consequently, M is the only critical output, and the overall MTBU of the circuit is, hence, determined solely by the synchronizer on x_{n+1}. Metastability on all the other nodes is always masked. This indicates that it is beneficial to use flip-flops with good metastability resolving capabilities on this signal to minimize the risk of an upset – the remaining "synchronizer" chains are just needed for timing alignment of the paths. They do not at all contribute to the MTBU, and cheaper implementations may be used. This is in contrast to other schemes, e.g., ones using Gray-encoded pointers, where all paths need to have good metastability resolution, even though only one specific path is relevant at a time (i.e., for a given count value).

3.5 Resynchronizer

An overview of the Resynchronizer circuit is shown in Figure 3.6. It can be seen that its general structure is closely related to that of the Desynchronizer. However, now the *last* n stages of the token buffer pipeline are sampled by n-flop synchronizers to transfer the pipeline status across the timing domain boundary and to ultimately deduce the FIFO fill level. Again, we need to sample n stages, in order to compensate for the synchronizer latency. Similar to the Desynchronizer here the top-most C gate is not considered part of this pipeline. It is included in the figure as it determines Δ_0^B. The toggle flip-flop is now used to control the input acknowledgment, i.e., to remove tokens from the pipeline.

For the minimum clock period T_c a similar constraint applies as for the Desynchronizer (Equation (3.13)).

$$T_c > T_{CO} + \sum_{0\leq i\leq n} \Delta_i^B + \Delta^S + T_S \bigwedge T_c > \max_{0\leq i\leq n} \delta_i^F \tag{3.13}$$

The variable δ_i^F denotes the delay from the node x_i to the non-inverting input of the next C gate. This constraint ensures that after a read from a full pipeline, the pipeline is again indicated as full for the next clock cycle, if there was already a new token pending on the input port ($req \neq ack$). To accomplish this all values x_i must invert their logic state within a clock cycle, which means that the bubble (free space) created by removing the token through the read must ripple up the whole pipeline from x_{n+1} to x_0. In addition, this constraint is more than sufficient to ensure that a read operation completes within one clock cycle.

Since it must be guaranteed that the *read* signal is only asserted if there is at least one token in the pipeline, care must be taken when generating the *empty* signal. As can be

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

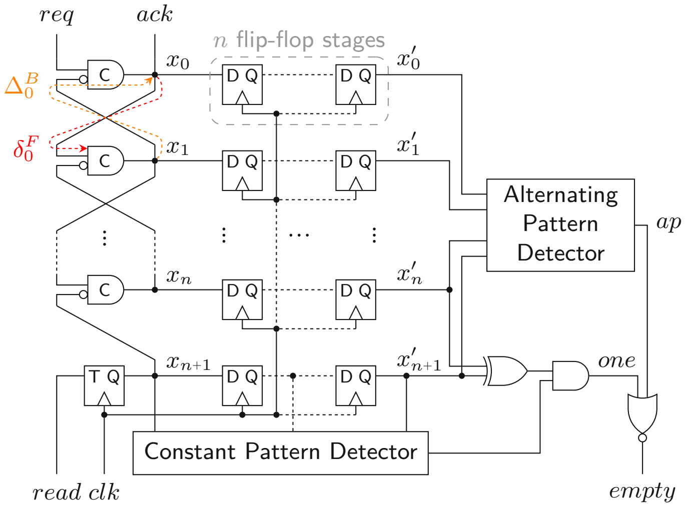


Figure 3.6: Resynchronizer circuit

seen from Figure 3.6, there are two conditions for the deassertion of *empty*, either (i) *ap* or (ii) *one* must be asserted.

Let's first consider the former condition. Whenever there is an alternating pattern detected on the vector $\mathbf{x}'$, we know that the pipeline was full n cycles ago. This means that, even if we assume the worst case, namely that n read operations have been performed in the last n cycles, there must still be one token left to read in the pipeline. Hence, whenever there is an alternating pattern on $\mathbf{x}'$, *empty* can be safely deasserted, allowing for a read operation in the next cycle. As long as there is a constant stream of new tokens arriving at the input side of the pipeline (*req* and *ack*), the (n-stage) pipeline will always stay full and the synchronous side can perform continuous reads (requirement (R3)). This behavior is illustrated by the timing diagram in Figure 3.7 for $n = 2$. For the sake of clarity we added x_{n+1} as an individual signal trace, as it represents the synchronously generated input acknowledgment to the pipeline. The signal $\mathbf{x_s}$ holds the intermediate value of the flip-flop (synchronizer) chain. The pipeline starts out in a full state where $\mathbf{x} = \mathbf{x}' = 0101$, which would allow for three read operations. After the first read another token immediately enters the pipeline in the same clock cycle, which leaves the pipeline in the state $\mathbf{x} = 1010$ (red state label). Note that, if no new token would enter, the state of the pipeline after the read would be 0010. Because of the new token entering the pipeline, in total four consecutive read operations are possible.

If the design would just use the *ap* signal to generate the *empty* signal, the Desynchronizer would already work fine for streaming applications. However, this way the situation may arise that there are tokens left in the pipeline (i.e., data left in the FIFO), that cannot be read because their number is too small to trigger the alternating pattern detector. Consider, for example, the case where only a single token arrives. This token would not be detected, and could hence not be read until further tokens arrive to fill up the pipeline.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

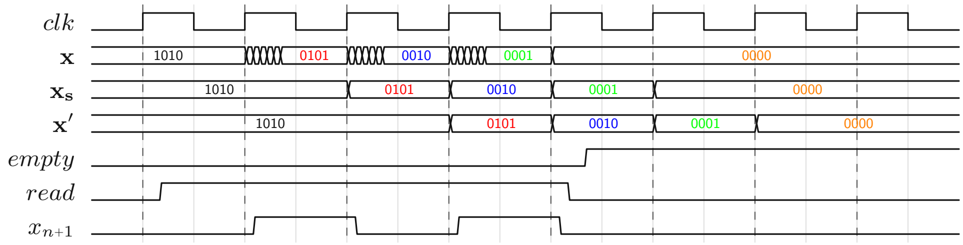


Figure 3.7: Example timing diagram ($n = 2$)

To eliminate this issue, we introduced condition (ii), i.e., the *one* signal. The *one* signal basically indicates whether there is an active, i.e., unacknowledged, output request ($x_{n+1} = \neg x_n$), which means that there is at least one token left in the pipeline. For that matter it can be viewed as a full indicator for the 0-stage sub-pipeline formed by the output request and input acknowledgment signals. Since the state vector x' only corresponds to the state of the pipeline n cycles ago, it is only safe to interpret the output of the XOR gate if there have not been any reads during this time. For this purpose the constant pattern detector checks if the input acknowledgment to the pipeline has changed during the last n cycles.

Now it only remains to analyze the effects of metastability on the pipeline state information signals *one* and *ap* and therefore the *empty* signal; specifically which scenarios can lead to a violation of (R2) or metastability propagation.

Similar to x_0 in the Desynchronizer, the flip-flop at x_{n+1} can never sample a transition, and thus cannot become metastable. Hence, knowing that $x'_{n+1} \in \{0, 1\}$, the remaining condition for the *one* signal to produce a stable output is that $x'_n \in \{0, 1\}$ must hold. Unfortunately, for the case where $x_n \in \{\uparrow, \downarrow\}$ we may still get an arbitrary output (i.e., 0, 1 or M) at x'_n, and therefore at *one*. Let's analyze the consequences of that:

- In the case of an M at x'_n metastability propagation to the *empty* output can ultimately cause the circuit to fail. This situation may arise, when a transition is traveling through the MPL, but did not quite make it to the bottom-most C gate. In such a case we must rely on the synchronizer to minimize the probability that this value actually appears at its output.

- If the synchronizer internally resolves metastability at x'_n to the same logic value as held by x'_{n+1}, *one* stays low, which is a safe outcome because read operations are forbidden.

- If, on the other hand, x'_n takes the opposite value of x'_{n+1}, *one* will be asserted. However, since the transition of the C gate at x_n that caused this value occurred n cycles ago, we can be certain that it has settled in the meantime and a token can successfully be read from the pipeline.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

For generating the *ap* signal, we have the alternating pattern detector in place whose basic function is represented by the left product term in Equation (3.14) (the symbol $\oplus$ denotes an exclusive or operation): The pipeline is full when all neighboring nodes have opposite logic states.

$$ap(\mathbf{x}') = \left(\bigwedge_{0 \leq i \leq n} x'_i \oplus x'_{i+1} \right) \wedge \left(\bigwedge_{0 \leq i \leq n\text{-}1} \neg(x'_i \oplus x'_{i+2}) \right) \tag{3.14}$$

Unfortunately, this function alone is not sufficiently resilient to metastability at any x'_i. To improve that, we have added the second (right) term in Equation (3.14), which states that every node must have the same logic state as the nodes next to its direct neighbors. While this term is logically redundant, it allows for metastability masking on all x'_i for $1 \leq i \leq n$ for the following reason: From the discussion about the possible values for $\mathbf{x}$, we know that transitions (i.e., $\uparrow$ or $\downarrow$) at x_i must always be framed by two opposite stable values (i.e., $x_{i\text{-}1} = \neg x_{i+1}$). This means that even if one (or more) of the signals x'_i for $1 \leq i \leq n$ evaluate to M because the synchronizer was not able to resolve metastability, M would not be able to propagate to the *ap* signal, since, due to the second term, its stable neighbors will make *ap* evaluate to a stable 0 in these cases. Hence, regarding the *ap* signal metastability is only an issue for node x_0. The overhead for the metastability containing alternating pattern detector can be estimated by a factor of two, when compared to a non-metastability containing version.

An analysis of the set of possible vectors for $\mathbf{x}$ reveals that there are only two cases that can lead to an ambiguous value at *ap* (i.e., vectors $\mathbf{x}$ where $\{ap(\mathbf{y}) | \mathbf{y} \in s_v(\mathbf{x})\} = \{0, 1, M\}$). These cases are alternating patterns starting with a rising or falling transition on x_0 instead of 1 or 0, respectively, i.e., $x_0, x_1 = \uparrow 0$ and $x_0, x_1 = \downarrow 1$, whereas $x_i = \neg x_{i\text{-}1}$ for $i > 1$. However, for the same reasoning as presented for the *one* signal, both of these cases are unproblematic, as long as the synchronizer manages to resolve metastability.

3.6 Results

This section presents a prototype implementation of the presented circuits to demonstrate the viability and practicality of our approach. Furthermore, we briefly analyze the impact of metastability on the MTBU of the circuits.

3.6.1 Prototype

For the prototype implementation we combined the Re- and Desynchronizer circuits to implement a bisynchronous FIFO in a field-programmable gate array (FPGA). Figure 3.8 shows an overview of this design. We used $n = 3$ synchronizer flip-flops and a FIFO depth of $d = 16$ elements.

The Altera/Intel Cyclone IV [Alt16] FPGA, we used for our prototype, features logic cells containing a 4-input lookup table (LUT) and a flip-flop. The interconnect of this

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

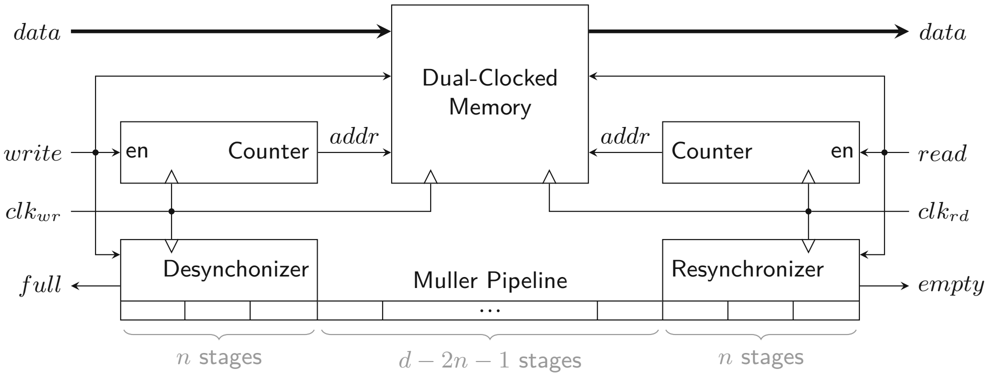


Figure 3.8: Bisynchronous FIFO prototype

device is capable of using the LUT's and flip-flop's output simultaneously. Hence, the 2-input C gates constituting the MPL are implemented in a single LUT, using one input for the feedback path and one as a reset input to ensure the correct start-up of the circuit. Unfortunately, there is no dedicated path in a logic cell from the output of the LUT back to its inputs. Hence, the feedback path is implemented using local interconnect, i.e., the same routing resource that also implements the connections between the C gate in the MPL. We assume that the delay through this local interconnect within the MPL (i.e., from the output of a LUT to the input of another or the same LUT) is always approximately Δ_{LI}. From the operation principle of the MPL, we can derive the following: When a C gate changes its output value, it takes at least $2\Delta_{LI} + \Delta_{LUT}$ before the input of C gate can change again, where Δ_{LUT} denotes the delay of a C gate in the MPL. This should give the feedback path plenty of time to settle before the inputs of the C gate change again. Thus, we don't believe this feedback path delay is critical for correct circuit operation.

The output of the LUT is directly connected to the flip-flop in the same logic cell, which results in a small and more importantly uniform delay for this connection across the whole pipeline (Δ^S in Figure 3.3b). Besides the explicit placements of the MPL and the sampling flip-flops, no further manual interventions were necessary. Actual values for the S/H times are not available in the datasheet of our target FPGA. Thus, we initially assumed that our circuit components and layout don't violate Equation (3.3) and then verified this assumption using extensive hardware tests. The MPL connecting the Re- and Desynchronizer must accommodate for the size of the (dual-clocked) memory, which means that we need $d-1$ stages in total. Recall that a full n-stage MPL contains $n+1$ tokens (Section 3.2).

The bisynchronous FIFO is embedded in a hardware testbench that generates pseudo-random input data using a linear feedback shift register (LFSR) and measures the data throughput. Furthermore, the *read* and *write* signals can also be controlled using LFSRs to simulate sporadic access to the FIFO (of course considering the *full* and *empty* signals). Both FIFO sides maintain a counter that records the total number of data items

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

that passed through the FIFO. Matching these two counters reveals if data has been lost or falsely generated (requirements (R1) and (R2)). The read side also automatically checks whether the read data matches the expected value.

The read and write clocks were generated using uncorrelated external clock sources. We tested a wide range of different read/write clock speed combinations and operation modes (maximal throughput and sporadic access). Each of these test scenarios ran over several days without any errors, which verified that the FIFO indeed works as expected and led us to believe that our initial assumption about the constraint in Equation (3.3) was correct. Furthermore, the experiments showed the throughput of a data stream is only limited by the slower one of the read and write clocks, fulfilling requirement (R3). We were able to measure maximum operation frequencies of up to 275 MHz for both the read and the write port. To put this number into context: The maximum performance for block RAMs (i.e., M9K blocks) in the used FPGA (speed grade C7) is 274 MHz [Alt16]. This means that our FIFO allows to utilize the full performance of the used memory block and that the timing constraints imposed on the MPL don't affect the maximum operation frequency in this case.

To further investigate the operation speed of just the Re- and Desynchronizer we devised a simple experiment. For the Desynchronizer a single token is inserted into an empty pipeline by asserting the *write* signal for exactly one clock cycle. Since the pipeline is empty, this event will cause all nodes x_i for $0 \leq i \leq n+1$ to change their logic value. In the next clock cycle the first level of the synchronizer flip-flops (i.e., those that sample the vector $\mathbf{x}$) is disabled. This means that they will only sample the state of pipeline in the clock cycle immediately following the token insertion and then keep this value. A debug interface then allows to read out the state of the vector $\mathbf{x}'$. If $\mathbf{x}'$ never attains a value different from the all-zero/all-one vector, then we can deduce that the timing constraint for the minimum clock period T_c is not violated. If it does, the token was not able to clear the pipeline within one clock cycle. Moreover, it is possible to gradually increase the frequency (i.e., decrease T_c) to examine the circuit's boundaries. For the Resynchronizer a similar experiment is performed, where a single token is removed from a full pipeline. In this case $\mathbf{x}'$ must always hold a strictly alternating bit pattern.

These experiments showed that the Re- and Desynchronizer could even handle higher frequencies. This is not unexpected, as an MPL stage can operate very fast, and for our timing constraints (Equations (3.11) and (3.13)) only a few stages become effective, namely as many as we have synchronizer stages. Note that this holds even if the FIFO is much deeper.

The external clock generators also allowed us to test the circuit under the influence of significant clock jitters (in the range of tens of MHz). Unsurprisingly the circuit proved to be very robust against such disturbances, because the only thing that must be ensured is that the minimum clock period T_c is still sufficiently long.

The example implementation also shows that our design uses fairly standard components and even maps quite nicely to FPGAs, even though the C gate is somewhat handcrafted.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

However, if a different pipeline style would be chosen, that does not use any special gates (e.g., MOUSETRAP [SN07]) we could even get rid of those.

3.6.2 Metastability Considerations

The prototype implementation is not intended to analyze the metastability behavior of the presented circuits, because the related properties are easy to assess analytically with the usual approach for n flip-flop synchronizers [Gin11]. The only parameter that must be determined for that is the actual rate of transitions at the input of the (relevant) flip-flop chains. Clearly, this data rate strongly depends on the actual use case of the circuit and its environment. However, as a pessimistic limit one can use the maximum data (token) rate.

For the Desynchronizer metastability can only occur if the pipeline starts to fill up because tokens cannot be removed (i.e., read) fast enough on the asynchronous side. Hence, the data rate for the MTBU estimation is basically the rate with which this happens.

For the Resynchronizer, we have a similar situation. As long as the pipeline is kept full, metastability is not an issue because the synchronizer flip-flops always sample stable input signals. Thus, metastability can only occur during the time the pipeline fills up and empties.

So normally one will come along with a very low n (like, e.g., 3), which keeps the latency of the circuit within bounds and also yields a low minimum requirement for the FIFO depth.

3.7 Conclusion

In this chapter we have presented a synchronous to asynchronous and an asynchronous to synchronous interface that are both based on the use of an MPL, intended for implementing A/S FIFOs. More specifically, we employ the MPL to convey tokens across the timing domain boundary. These tokens provide both, sender and receiver with all necessary information for their *local* memory pointer management and flow-control. Reducing the information exchange to tokens in an MPL allows for a systematic and efficient metastability handling at the boundary. Although still synchronizers are mandatory to cope with metastability, which is in general inevitable at timing domain boundaries, we have confined the critical components to a single synchronizer path per direction. In addition our proposed concept pipelines data transfers and thus handles the synchronizer latency in the transmission of data streams without negatively impacting the throughput, resulting in a sustained data rate of one data item per clock cycle.

Moreover, the proposed circuits also allow for a configuration as a bisynchronous FIFO, which we have used to experimentally verify our approach and confirm its robust operation, high throughput and complete flexibility with respect to the frequencies on both synchronous sides (as long as the lower limit for T_c is satisfied).

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Porting the proposed designs to some application-specific integrated circuit (ASIC) technology should be fairly straightforward. Such design flows and target technologies allow for more flexibility regarding routing delays and timing constraints compared to what is possible with an FPGA-based prototype. However, this, of course, assumes that the target technology in question features C gates, which might not be the case. Hence, one potential improvement to our design would be an implementation based on other pipeline styles that can be realized with standard CMOS library components (i.e., without C gates).

Another concern for future developments is related to the performance of non-blocked data streams.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

TU WIEN Bibliothek Your knowledge hub

CHAPTER 4

Efficient Delay-Insensitive Communication

As already explained in Chapter 2, compared to synchronous approaches, asynchronous DI communication links have desirable properties with respect to their robustness against timing variations and delay assumptions required to implement them. This makes them especially interesting as a form of system-level intra-chip or inter-chip connection, particularly in the context of GALS systems [Cha84]. Hence, in this chapter we seek to explore the design space of how such links can be implemented and provide new insights into key components and communication protocols involved.

In many contemporary applications, energy-efficiency of semi-conductor devices is a major concern. It is well understood, that communication links between function blocks (within an SoC, or on a printed circuit board (PCB)) are a significant contributor to the overall power consumption of a system, due to the relatively high capacitances involved. In this context, synchronous communication has some disadvantages due to the high transition rate of the clock line. Moreover, delay mismatch (skew) among the different wires of the communication link is problematic. This also holds true for those asynchronous approaches that employ some explicit request signals such as the BD approach. With ever-increasing PVT variations these issues steadily gather more relevance. DI communication elegantly overcomes these problems by employing a special data encoding (and protocol) that enable the receiver of a transmission to recognize when a code word is complete (i.e., all wires made their final transitions) without the need for an accompanying clock or request signal, and even in the presence of arbitrary skew on the transmission link. Such links have been successfully employed in many applications, like Spinnaker [NFG+13], [SFGP09], Chain [BF02] or [MG20].

A fundamental challenge in the design of DI interconnect is to find the right balance between efficiency of the DI code and protocol on the one hand, and the implementation complexity on the other hand (e.g., the area overhead for encoders, decoders and CDs). In this context efficiency refers to the number of data bits a code word of a given length can hold as well as to the number of bus transitions it requires for transmission. Generally, complex codes and protocols have a better efficiency but are more costly to implement.

To facilitate a fair and meaningful comparison between DI codes and protocols, this chapter considers many design aspects of (pipelined) DI links, such as the one shown in Figure 4.1. This includes the implementation of the transmitter, the receiver and the intermediate pipeline stages as well as their sub-components (encoders, decoders, CDs).

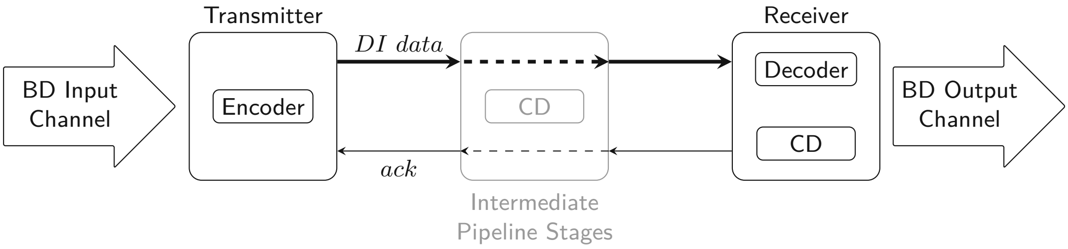


Figure 4.1: Delay-insensitive link overview

One drawback of DI codes is that they are generally not well-suited for data processing. As already explained in Section 2.6, even circuits operating on the arguably most simple DI encoding, i.e., the dual-rail code, entail a significant hardware overhead when compared to an equivalent circuit just processing binary data. Hence, for our analysis we assume that the transmitter and receiver operate on binary coded data, in particular we consider asynchronous BD channels. Consequently, we will also discuss the circuits that convert binary coded data (i.e., a data word) to a DI code word, which we refer to as encoders, as well as circuits that perform the reverse operation, called decoders. Thus, this chapter can be viewed as a continuation to the previous one, where the conversion between synchronous and BD interfaces was investigated.

The chapter starts off by clearly defining and comparing the classes of constant-weight (i.e., m-of-n) and Berger codes in Section 4.1. Furthermore, some of their basic properties are explored and important notation is introduced, which will be used throughout the chapter. In general, Berger codes excel because of their simple encoding and the complete absence of a decoder, while, unfortunately, their CDs tend to become complex and difficult to realize in a QDI way (i.e., without timing assumptions). Constant-weight codes, on the other hand, often provide higher coding efficiency and facilitate completion detection with significantly lower efforts, but incur a higher penalty for encoding and decoding. The reason for the high overhead is that constant-weight codes are not systematic, i.e., the mapping between data words and code words is not predetermined by the code itself (in contrast to Berger codes). However, this mapping strongly impacts the implementation overhead, and even optimizing the implementation for a given mapping is non-trivial as was already tackled in [BTEF03].

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Consequently, the first contribution of this chapter is a code word mapping scheme for constant-weight codes, which divides the code words into a systematic and a non-systematic part (see Section 4.2). We refer to this approach as Partially-Systematic Constant-Weight Codes (PSCWCs). Our presumption is, that the systematic part will simplify the encoding and decoding process. We show that this approach indeed yields very regular mappings with reoccurring sub-codes for the non-systematic part, which allows for efficient encoder and decoder circuits. Although the method is not fully generalized, we carefully explore the design space relevant for DI communication links.

The second contribution of this chapter, presented in Section 4.3, is a new class of DI protocols, which bridge the two "classic" asynchronous approaches – that is the RZ and the NRZ protocol. With these *hybrid protocols*, we are able to show that there is a whole spectrum of DI communication schemes, each with different use cases, complexity, advantages and disadvantages.

Furthermore, we provide a novel CD design approach for the m-of-n and Berger code classes in Section 4.4 that works with the RZ as well as the new hybrid protocols. The approach is mainly based on prior work by Piestrak [Pie98] as well as [HSS15, CJN10] and represents to the most general and optimized approach for CDs yet. In our construction approach, we carefully avoid gate orphans, which compromise the underlying QDI timing model for CDs and which are not fully avoided by current state-of-the-art solutions.

Section 4.5 provides example link implementations for all protocols discussed in Section 4.3. Those circuits will be used by an extensive case study in Section 4.6 where we systematically analyze all presented techniques. We not only investigate the area overhead for encoders, decoders and CDs for all the discussed codes and protocols but also consider the overall implementation costs of complete DI communication links for the model-architectures we use in this context. Regarding the protocols, we examine the classic RZ, the presented hybrid protocols (if applicable to a particular code) as well as the NRZ *transition signaling* protocol. We only consider transition signaling as it works with the same DI codes as the RZ and hybrid protocols. In addition, we perform a systematic analysis of the performance implications of the different approaches. This analysis provides useful insights into the advantages and disadvantages of the individual approaches for different use cases.

Finally, Section 4.7 concludes the chapter.

4.1 Delay-Insensitive Codes

Since there are no assumptions on signal delays in DI communication schemes, transitions of the individual rails of a DI bus may arrive at the receiver in any order. Let $\mathbb{F}_{2^n} = \{0,1\}^n$ denote the set of all possible n bit vectors. Further, if $\mathbf{v} \in \mathbb{F}_{2^n}$ denotes a bit vector then v_0 to v_{n-1} refer to the individual bits. We define a code C with code word length n as a subset of $\mathbb{F}_{2^n}$. Verhoeff shows that a (4-phase) code is DI iff it is *unordered* [Ver88]. This means that there must not exist a code word that is contained in another code word,

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

i.e., the positions of the ones in a code word may not be a subset of the positions of the ones in another code word. Consider the following example, let $\mathbf{c}_1 = 001$ and $\mathbf{c}_2 = 011$ be two elements of some set $C \subseteq \mathbb{F}_{2^3}$. Since $\mathbf{c}_1$ is contained in $\mathbf{c}_2$, i.e., $\mathbf{c}_1 \sqsubset \mathbf{c}_2$, C cannot be a DI code. Hence, formally we can state that a code C is delay-insensitive iff for all $\mathbf{c}_1, \mathbf{c}_2 \in C$ $(\mathbf{c}_1 \neq \mathbf{c}_2)$ we have that $\mathbf{c}_1 \not\sqsubset \mathbf{c}_2$. In this chapter we focus on constant-weight (m-of-n) and Berger codes which both meet this requirement. In the following we will introduce some notations and definitions that will be used throughout the next sections.

A constant-weight or balanced code $C^{cw}_{m,n} \subset \mathbb{F}_{2^n}$ is defined by Equation (4.1):

$$C^{cw}_{m,n} = \{\mathbf{c} \in \mathbb{F}_{2^n} \mid h(\mathbf{c}) = m\}, \tag{4.1}$$

where $h(\mathbf{c})$ denotes the Hamming weight of the bit vector $\mathbf{c}$. The size (i.e., the number of symbols or *code words*) of an m-of-n code is given by the binomial coefficient ($\left|C^{cw}_{m,n}\right| = \binom{n}{m}$). However, when transmitting binary data, only a subset of these code words is actually used, usually the nearest power of two. With the exception of the dual-rail code, m-of-n codes are non-systematic. This means that there does not exist a subset of bit positions in the code that contains the unencoded data (i.e., the *data word*) for all code words. Hence, one is completely free to choose a suitable mapping for a particular purpose. In Section 4.2 we will present one possible mapping strategy.

The Berger code [Ber61], on the other hand, is a systematic code. Hence, every code word can be split into a b-bit data part $\mathbf{d}$ and a k-bit check (parity) part $\mathbf{p}$, where $\mathbf{p}$ carries the binary representation of the number of zeros in the data part. As shown in the formal definition of the Berger code in Equation (4.2), the size of k depends on the size of the data part. Here the colon symbol denotes concatenation, while $[\![\mathbf{p}]\!]$ returns the numerical value of the binary vector $\mathbf{p}$. The size of the Berger code C^B_b is naturally given by 2^b.

$$C^B_b = \bigcup_{\mathbf{d} \in \mathbb{F}_{2^b}} \{\mathbf{d} : \mathbf{p} \mid \mathbf{p} \in \mathbb{F}_{2^k}, [\![\mathbf{p}]\!] + h(\mathbf{d}) = b\}, \text{ where } k = \lceil \log_2(b+1) \rceil \tag{4.2}$$

A Berger code encoder simply uses adders to calculate the check part $\mathbf{p}$ from the data part $\mathbf{d}$ and is, hence, quite straightforward to implement. Since the Berger code is systematic, there is no hardware overhead for the decoding process.

There are a few aspects that define the quality of a DI code. Of course the overhead for encoding and decoding as well as completion detection have to be considered. Besides that it is also important how many bits of information can be encoded by a given code and how many bus transitions it takes to transmit it. The coding efficiency R specifies how many bits can be encoded per rail and always yields a value $0 < R < 1$ (larger values are better). The power metric P on the other hand measures how many transitions are required to transmit a single bit (smaller values are better).

Equations (4.3) and (4.4) show the coding efficiency and power metric for constant-weight codes using the RZ protocol. The binomial coefficient in these equations calculates the

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

number of code words in an m-of-n code. Since this number is generally not a power of two we need the floor operation.

$$R^{cw|RZ}_{m,n} = \frac{\lfloor \log_2 \binom{n}{m} \rfloor}{n} \tag{4.3}$$

$$P^{cw|RZ}_{m,n} = \frac{2m}{\lfloor \log_2 \binom{n}{m} \rfloor} \tag{4.4}$$

The coding efficiency of the RZ Berger code protocol is quite straightforward to calculate (Equation (4.5)). The variable k again denotes the number of parity bits as defined in Equation (4.2). However, since the code words of the Berger code have different Hamming weights the determination of power metric is a little bit more involved. For that we assume that every code word is equally likely to occur. Equation (4.6) basically goes through all possible values p for the parity part $\mathbf{p}$, calculates the Hamming weight of the whole code word $((h(\langle p \rangle) + b - p)$ depending on p and multiplies it with the number of code words $(\binom{b}{b-p})$ that have this Hamming weight. Note that the operator $\langle p \rangle$ returns a binary vector with the numerical value of p such that we can apply the Hamming weight function[1]. The sum of these products is then divided by the total number of symbols (2^b) and the number of bits (b). Notice that Berger codes are most efficient (in terms of both R and P) if $b = 2^x - 1$, because then all available symbols in the parity part $\mathbf{p}$ are actually used in some code word.

$$R^{B|RZ}_{b} = \frac{b}{b+k} \tag{4.5}$$

$$P^{B|RZ}_{b} = 2 * \frac{\sum_{0 \le p \le b} \left(h(\langle p \rangle) + b - p\right) * \binom{b}{b-p}}{2^b * b} \tag{4.6}$$

Notice that since NRZ protocols lack the null phase, the power metric is improved by a factor of two (i.e., $P^{RZ} = 2P^{NRZ}$), the coding efficiency, however, stays the same.

4.2 Partially Systematic Constant-Weight Codes

This section covers the PSCWC, a semi-generic mapping scheme we use to find efficient encoder and decoder circuits for the constant-weight codes used in the case study in Section 4.5. We first give a formal definition of the approach and then show how it can be used to create efficient encoder and decoder circuits.

---

[1] Formally we can define the operator as $\langle p \rangle = \mathbf{p} | \mathbf{p} \in \mathbb{F}_{2^{\lceil log_2(p+1) \rceil}} \wedge [\![\mathbf{p}]\!] = p$

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.2.1 Formal Definition

Given a j-of-k constant-weight code, where $j < \frac{k}{2}$, Equation (4.7) defines the partially systematic $(j{+}s)$-of-$(k{+}s)$ code.

$$\begin{gathered} C^{ps}_{j,k,s} = \bigcup_{\mathbf{d} \in \mathbb{F}_{2^s}} \{\mathbf{d} : \mathbf{c} \mid \mathbf{c} \in C_{h(\mathbf{d})}\}, \\ \text{where } s \leq k - 2j, e \leq \lfloor \log_2 \tbinom{k}{j} \rfloor, C_h \subseteq C^{cw}_{j+s\text{-}h,k} \text{ s.t. } \ |C_h| = 2^e \end{gathered} \tag{4.7}$$

This definition ensures that every code word is composed of a systematic part $\mathbf{d}$ containing s bits of the data word and a non-systematic part $\mathbf{c}$ containing the remaining e bits in some encoded form. Since the Hamming weight of the whole code word must be constant, the Hamming weight $\mathbf{c}$ is dictated by the Hamming weight of $\mathbf{d}$, with its minimum being j (if $h(\mathbf{d}) = s$). This minimum determines the number of bits e encodeable in the non-systematic part $\mathbf{c}$. Also note the restriction on the size of s imposed by Equation (4.7). If $h(\mathbf{d}) = 0$, then the symbols for $\mathbf{c}$ are supplied by the $(j{+}s)$-of-k code C_0. Under the assumption of the number of systematic bits s being maximal (i.e., $s = k - 2j$, as also constrained by Equation (4.7)), we have $j + s = k - j$ and $C_0 \subseteq C^{cw}_{k\text{-}j,k}$. Because of a basic property of the binomial coefficient, stated in Equation (4.8), it is guaranteed that there are enough symbols in this code to encode the required e bits. This holds for all values of $h(\mathbf{d})$ in between 0 and s.

$$\binom{n}{m} \leq \binom{n}{x}, \text{where } m \leq x \leq n - m \tag{4.8}$$

The resulting code $C^{ps}_{j,k,s}$ is a subset of $C^{cw}_{j+s,k+s}$. However, with its size of 2^{s+e} it may encode a smaller number of bits.

To better illustrate this concept, consider the example of the $C^{ps}_{1,4,1}$ code. Here a single systematic bit (i.e., $s = 1$) is appended to the 1-of-4 code (i.e., $j = 1$, $k = 4$, $e = 2$) resulting in the partially systematic 2-of-5 code. Notice that since $k - 2j = 2$, s fulfills the constraint imposed on it by Equation (4.7). Equation (4.9) shows the resulting definitions for this concrete example.

$$\begin{aligned} C^{ps}_{1,4,1} &= \{0 : \mathbf{c} \mid \mathbf{c} \in C_0\} \cup \{1 : \mathbf{c} \mid \mathbf{c} \in C_1\} \subseteq C^{cw}_{2,5} \\ C_0 &= \{0101, 0110, 1001, 1010\} \subseteq C^{cw}_{2,4} \\ C_1 &= \{1000, 0100, 0010, 0001\} \subseteq C^{cw}_{1,4} \end{aligned} \tag{4.9}$$

Notice how the Hamming weight of the systematic part (i.e., the single systematic bit) determines the code for the non-systematic part. The combined Hamming weight of the systematic and non-systematic part is always two, though. So we obtain a subset of the 2-of-5 code comprising only eight symbols (while $\binom{5}{2} = 10$). Hence, we can still encode three bits of data but encoding and decoding may potentially be simplified because of the systematically mapped bit.

This illustrates the basic concept: Use the freedom to (a) select a suitable subset of the full code set and (b) choose a suitable mapping from data words to code words, to

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

make at least part of the bits within the code word systematic, thus simplifying the encoder/decoder implementation. Concerning (b), Equation (4.9) illustrates how fixing the first bit to be systematic restricts the choice in the encoding of the remaining bits. Still, the mapping of elements within, e.g., C_0 to data words starting with 0 can be freely permuted, which leaves further room for optimization in the implementation (which we perform in a heuristic fashion later in Section 4.2.2). Also, there would have been other choices for the four elements within C_0.

However, since we are interested in maximizing the coding efficiency, we want to take a slightly different construction approach. By starting out with an m-of-$2m$ code, which offers the best coding efficiency with regard to the length of its code words ($2m$), we try to map as many bits systematically as possible, without compromising on the total number of bits that can be encoded. This approach is outlined by Equation (4.10). Again s denotes the number of systematic bits in each code word and e the number of bits encoded in the non-systematic part. However, now s is restricted to be the largest number x, such that the code used for the non-systematic part is still able to encode $\lfloor \log_2 \binom{2m}{m} \rfloor - x$ bits. Since the capacity (in number of encoded bits) of the non-systematic part is bounded by the capacity of the m-of-$(2m-x)$ code, it is given by $\lfloor \log_2 \binom{2m-x}{m} \rfloor$.

$$
\begin{aligned}
C_m^{ps} &= \bigcup_{\mathbf{d} \in \mathbb{F}_{2^s}} \{\mathbf{d} : \mathbf{c} \mid \mathbf{c} \in C_{h(\mathbf{d})}\}, \\
&\quad \text{where } s = \max(S), \\
S &= \{x \mid x \in \mathbb{N}, x \leq m, \lfloor \log_2 \tbinom{2m}{m} \rfloor - x = \lfloor \log_2 \tbinom{2m-x}{m} \rfloor\}, \\
e &= \lfloor \log_2 \tbinom{2m}{m} \rfloor - s, C_h \subseteq C^{cw}_{m+s\text{-}h,2m\text{-}s} \text{ s.t. } |C_h| = 2^e
\end{aligned}
\tag{4.10}
$$

To demonstrate this construction with the help of an example, let's take a more in-depth look at the partially systematic 3-of-6 code C_3^{ps}, which is able to encode four bits of data. First, s needs to be calculated. It is not too difficult to verify that the set S only contains the values $\{0, 1, 2\}$, hence $s = 2$ and $e = 2$. With this information, the sets $C_0, ..., C_2$ can be defined, which are in turn used to finally specify C_3^{ps}

$$
\begin{aligned}
C_3^{ps} = \{00 : \mathbf{c} \mid \mathbf{c} \in C_0\} &\cup \{01 : \mathbf{c} \mid \mathbf{c} \in C_1\} \cup \\
\{10 : \mathbf{c} \mid \mathbf{c} \in C_1\} &\cup \{11 : \mathbf{c} \mid \mathbf{c} \in C_2\} \subseteq C^{cw}_{3,6} \\
C_0 &= \{1110, 1101, 1011, 0111\} \subseteq C^{cw}_{3,4} \\
C_1 &= \{0101, 0110, 1001, 1010\} \subseteq C^{cw}_{2,4} \\
C_2 &= \{0001, 0010, 0100, 1000\} \subseteq C^{cw}_{1,4}
\end{aligned}
\tag{4.11}
$$

Since there are three unique values the Hamming weight of the two systematic bits can take, three different codes are required to supply the symbols for the non-systematic part, such that the Hamming weight of the combined code words is always three.

An important question is how many systematic bits can be encoded in a given m-of-$2m$ code. It is quite straightforward to verify by enumeration that for relevant values of m

Table 4.1: Examples for Partially Systematic Codes

Code	# systematic bits	# non-systematic bits
3-of-6	2	2
4-of-8	1	5
5-of-10	3	4
6-of-12	3	6

$(m \leq 20)$, s is always smaller than 4. Table 4.1 shows the partitionings of codes with $m \leq 6$. We will use these codes for the comparison in Section 4.6.

At this point, we want to emphasize the difference to Knuth's coding scheme [Knu86] and related approaches like [IW10]. These schemes use a strict separation between data and parity bits. To encode a data word in Knuth's approach, the first g data bits are inverted, such that the whole data part always has the same Hamming weight. This number g is then encoded with some constant-weight code to get the parity bits of the code word. For decoding, first the number g has to be extracted from the parity bits and then the data has to be inverted accordingly. This approach is very generic and works for arbitrary data word lengths. It can easily be applied to data words several tens or hundreds of bits long. However, as a result of this strict separation the code does not use the full capacity of the underlying constant-weight codes.

In our proposed approach, there is no clear distinction between data and parity bits. Moreover, it is mainly targeted for short length code words and provides optimal coding efficiency for these cases.

4.2.2 Encoding and Decoding

When compared to the quite simple encoders and decoders for the Berger code, the circuits for the partially-systematic (PS) m-of-n codes are more involved. Unfortunately we are not aware of a complete procedure that directly yields efficient circuits. Figure 4.2 shows the general structure of an encoder for a PSCWC $C^{ps}_{j,k,s}$. We use d_i to denote the individual bits of the data words (d_0 is the least significant bit (LSB) and c_i to denote the rails of the code words. The systematic part of the code words $(c_{s+k\text{-}1}...c_k)$ is, hence, always given by the vector $(d_{e+s\text{-}1}...d_e)$. Since the encoding of the non-systematic part changes based on the Hamming weight of the systematic part, an x-of-k multi-encoder is employed, with x being controlled by a sorting-network- or adder-based structure that computes $h(d_{e+s\text{-}1}...d_e)$. This encoder must be able to produce code words of all x-of-k codes $(j \leq x \leq j + s)$ required for the non-systematic part.

Figure 4.3a shows an example implementation of an encoder for the PS 3-of-6 code (as defined by Equation (4.11)). Its control logic consists of an AND and an XOR gate,

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

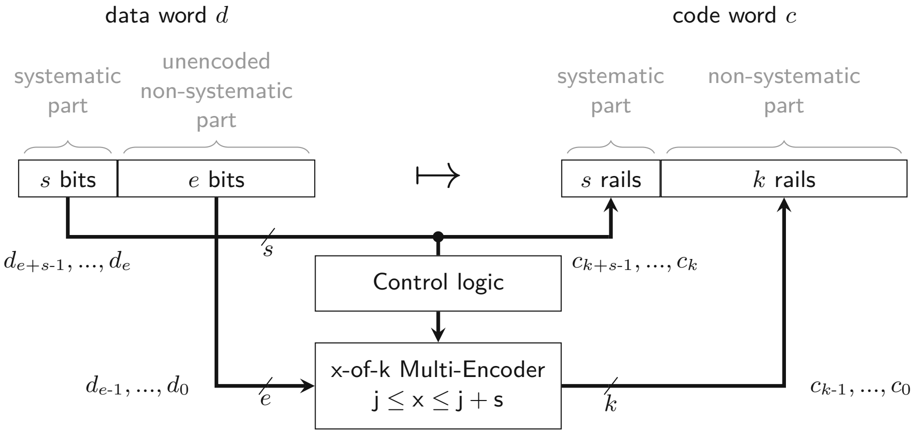
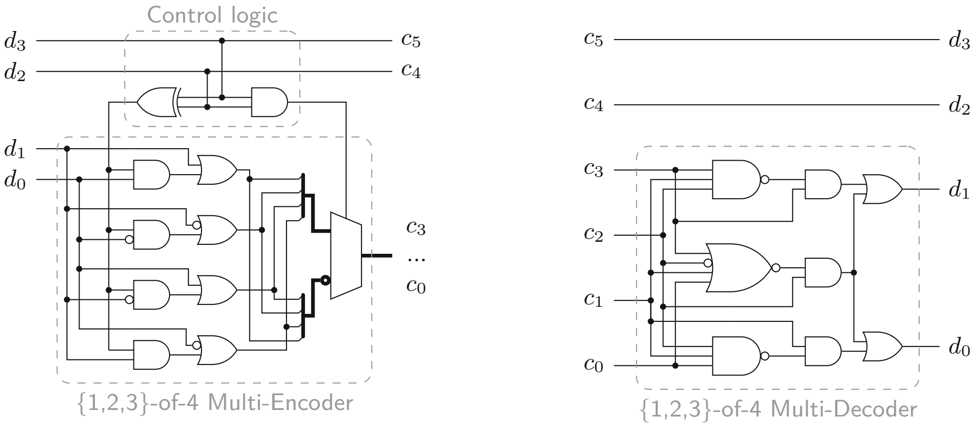


Figure 4.2: PSCWC encoder for $C^{ps}_{j,k,s}$

which essentially form a half adder (HA). This HA is fed by the systematic bits $d_3 d_2$ and generates two control signals. These control signals can take one of three possible values (00, 01 and 10) and select which code must be used by the $\{1, 2, 3\}$-of-4 multi-encoder.



(a) Encoder (b) Decoder

Figure 4.3: Circuits for the PS 3-of-6 code

The decoder circuits for the PSCWCs are built in a similar way. Again the systematic part can be used to generate control signals for an appropriate multi-decoder. However, often this is not really necessary, as the non-systematic part obviously carries the information about the respective value of x. Therefore, in contrast to the multi-encoder, the multi-decoder has all required information to generate the binary output. So in principle, no additional control signals generated from the systematic part are necessary, albeit such a design approach can yield more efficient circuits. Figure 4.3b shows the decoder circuit for the PS 3-of-6 code. Here it can be seen that no additional control logic is required that depends on the Hamming weight of the systematic part. The {1,2,3}-of-4 multi-decoder is by itself able to decode all 1-of-4, 2-of-4 (i.e., dual-rail) and 3-of-4 code words.

Obviously, the multi-encoders and decoders have a large impact on the total hardware overhead of the encoder and decoder circuits. Thus, it is very important to find mappings of data words to the respective code words of the non-systematic part, that allow for an efficient implementation of encoder and decoder. To give a more general approach for dealing with this problem, we draw some ideas from the incomplete m-of-n codes proposed in [BTEF03]. Here larger DI codes are assembled by a concatenation of simpler sub-codes according to certain construction rules. A simple example for this approach is the incomplete 2-of-7 code, where the code words fall in one of two categories: Either the first three bits are zero and concatenated with two dual-rail bits, or the first three bits constitute a 1-of-3 code word followed by a 1-of-4 code word in the next four bits. The term incomplete refers to the fact that some code words, like 1100000, are not part of the code, although they would be valid 2-of-7 code words. However, they are excluded because they don't follow the construction rule of the code. The incomplete 2-of-7 encoding is also shown in the first row of Table 4.4. The notation used in this table as well as Tables 4.2, 4.3 and 4.5 is as follows: The functions m-of-$n(\mathbf{v})$ express the encoding of the binary vector $\mathbf{v}$ to an m-of-n code word. Consequently, $DR(\mathbf{v})$ is used to denote the dual-rail encoding. Note that, since there are only three symbols in the 1-of-3 and 2-of-3 codes, one vector cannot be encoded by these functions. In our implementation this is the data word 00.

The usage of incomplete codes simplifies the implementation of the encoder (and decoder) circuits, because it allows to distribute the task of encoding a (complex) code word to simpler sub-encoders. Hence, for the example of the incomplete 2-of-7 code, a $\{0, 1\}$-of-3 and a $\{1, 2\}$-of-4 multi-encoder are required. The price is a reduction in the number of available code words, but as long as all data words can still be encoded, this is unproblematic.

Tables 4.2 to 4.5 show the mappings performed by the multi-encoders for the PS 3-of-6, 4-of-8, 5-of-10 and 6-of-12 codes, respectively. Note that every line in these tables defines an incomplete m-of-n code. The condition column specifies when a certain code word structure has to be used. The 3-of-7 and 4-of-7 as well as the 2-of-7 and 5-of-7 codes used by 6-of-12 code are exactly the same ones as those listed in the tables for the PS 4-of-8 and 5-of-10 codes.

It can be seen that the construction rules for all x-of-j sub-codes of a particular PS code are very similar. For a specific section of a code word there is only a certain number of possible encodings (i.e., sub-codes). For example, the section $c_3 ... c_0$ of the PS 5-of-10 code alwyas either contains a 1-of-4, a dual-rail or a 3-of-4 code word. This property holds across all codes supported by a particular multi-encoder, which allows for efficient hardware reuse when designing these circuits.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

Table 4.2: {1,2,3}-of-4 multi-encoder for the PS 3-of-6 code C_3^{ps}

$h(c_5c_4)$	C_h	Condition	$c_1...c_0$
2	1-of-4	-	1-of-4(d_1d_0)
1	2-of-4	-	$DR(d_1d_0)$
0	3-of-4	-	3-of-4(d_1d_0)

Table 4.3: {3,4}-of-7 multi-encoder for PS 4-of-8 code C_4^{ps}

$h(c_7)$	C_h	Condition		$c_6c_5c_4$	$c_3...c_0$
1	3-of-7	$d_4 = 0$	$d_3d_2 = 00$	000	3-of-4(d_1d_0)
			$d_3d_2 \neq 00$	2-of-3(d_3d_2)	1-of-4(d_1d_0)
		$d_4 = 1$	$d_3d_2 = 00$	1-of-3$(d_1\overline{d_1})$	$d_0d_0\overline{d_0d_0}$
			$d_3d_2 \neq 00$	1-of-3(d_3d_2)	$DR(d_1d_0)$
0	4-of-7	$d_4 = 0$	$d_3d_2 = 00$	111	1-of-4(d_1d_0)
			$d_3d_2 \neq 00$	1-of-3(d_3d_2)	3-of-4(d_1d_0)
		$d_4 = 1$	$d_3d_2 = 00$	2-of-3$(d_1\overline{d_1})$	$d_0d_0\overline{d_0d_0}$
			$d_3d_2 \neq 00$	2-of-3(d_3d_2)	$DR(d_1d_0)$

Table 4.4: {2,3,4,5}-of-7 multi-encoder for the PS 5-of-10 code C_5^{ps}

$h(c_9c_8c_7)$	C_h	Condition	$c_6c_5c_4$	$c_3...c_0$
3	2-of-7	$d_3d_2 = 00$	000	$DR(d_1d_0)$
		$d_3d_2 \neq 00$	$1of3(d_3d_2)$	1-of-4(d_1d_0)
2	3-of-7	$d_3d_2 = 00$	000	3-of-4(d_1d_0)
		$d_3d_2 \neq 00$	1-of-3(d_3d_2)	$DR(d_1d_0)$
1	4-of-7	$d_3d_2 = 00$	111	1-of-4(d_1d_0)
		$d_3d_2 \neq 00$	2-of-3(d_3d_2)	$DR(d_1d_0)$
0	5-of-7	$d_3d_2 = 00$	111	$DR(d_1d_0)$
		$d_3d_2 \neq 00$	2-of-3(d_3d_2)	3-of-4(d_1d_0)

Table 4.5: {3,4,5,6}-of-9 multi-encoder for the PS 6-of-12 code C_6^{ps}

$h(c_{11}c_{10}c_9)$	C_h	Condition	c_8c_7	$c_6...c_0$
3	3-of-9	$d_5 = 0$	00	3-of-7$(d_4...d_0)$
		$d_5 = 1$	$DR(d_4)$	2-of-7$(d_3...d_0)$
2	4-of-9	-	$DR(d_5)$	3-of-7$(d_4...d_0)$
1	5-of-9	-	$DR(d_5)$	4-of-7$(d_4...d_0)$
0	6-of-9	$d_5 = 0$	11	4-of-7$(d_4...d_0)$
		$d_5 = 1$	$DR(d_4)$	5-of-7$(d_3...d_0)$

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

4.3 Hybrid Protocols

This section proposes four novel 2-phase/4-phase hybrid DI communication protocols that both rely on allowing more than a single spacer. All these protocols use one *default* spacer (the all-zero pattern) and a set of other *special* spacers (for one protocol this set only contains one code word). Hence, one transmission cycle of the new hybrid protocols consists of the data phase and one of two possible spacer phases (default or special).

Recall that in Section 2.1.2 we introduced the notion of the spacer for the RZ protocol and stated that it is usually encoded by the all-zero pattern on every rail of the DI bus. We can generalize that to the statement that the spacer must simply be a single distinct bit pattern. For each bit of the spacer pattern that is zero (one) we can now define that the corresponding rail of the DI bus must only perform

(i) rising (falling) transitions when the bus switches from the spacer to the data phase and

(ii) falling (rising) transitions when the bus switches from the data to the spacer phase.

The code words of the DI code must then be unordered with respect to this chosen spacer pattern $\mathbf{s}$. This means that the set of bit vectors that is obtained by taking the bit-wise XOR of $\mathbf{s}$ and every bit pattern that should constitute a valid DI code word, must be unordered. If we again look at the case of the RZ protocol with the all-zero spacer, *only* rising (falling) transitions are allowed when switching from the data (spacer) phase to the spacer (data) phase. Notice that since there are no spacers in NRZ protocols every rail is always allowed to make a transition when switching from one data phase to the next.

With the hybrid protocols we can relax the two constraints for the switching behavior of RZ protocols formulated above to a certain degree, without allowing the “complete” freedom of the NRZ protocol. We do this by allowing more than a single spacer, and applying a new set of rules depending on the current state the protocol is in. When the protocol is in the default spacer phase again only rising transitions can occur. However, in the data phase one of two things can happen. Either all rails return to zero again (default spacer) or additional ones appear at the DI bus until a special spacer is reached. In the special spacer phase again only falling transitions back to the next data phase (i.e., next valid code word) are allowed.

Although it would again be possible to use an arbitrary bit pattern for the default spacer of the hybrid protocols, we don’t consider this in our explanations for the sake of simplicity. Note that the *ack* signal still makes two transitions for each complete bus transaction (i.e., the transmission of one code word and one spacer).

4.3.1 Data Spacer Protocol

The Data Spacer (DS) protocol uses the spacer to transmit one additional bit of information in the spacer phase and works with m-of-n as well as Berger codes. After each data

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

phase, the transmitter checks this bit b_s and decides whether to go to the all-zero or the all-one spacer (see Figure 4.4). This is possible because every code word of a DI code can be reached from either of these two spacers without any potential for misinterpretation (unorderedness property). Note that, when applied to a single dual-rail bit, a special case of this approach is the LEDR protocol [MAMN08]. So in a sense, the DS protocol represents the smallest step from a 4-phase protocol with its single spacer (that only carries control information but no data) to a 2-phase protocol (in which all protocol phases carry data, and the control information is embedded in the set of code words used to encode these data). While in a conventional level-encoded 2-phase DI code like LEDR the two code sets have equal size, the DS protocol is a very unbalanced 2-phase protocol – which is likely to yield different properties that we are interested to explore.

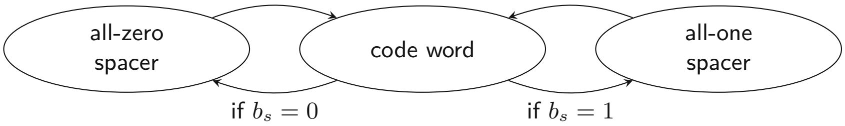


Figure 4.4: DS protocol state diagram

Through the addition of the single extra bit transmitted by the spacer, this approach obviously has improved coding efficiency with respect to a single-spacer (i.e., the RZ) protocol (Equation (4.12)).

$$R^{cw|DS}_{m,n} = \frac{\lfloor \log_2 \binom{n}{m} \rfloor + 1}{n}, \quad R^{B|DS}_{m,n} = \frac{b+1}{b + \lceil \log_2(b+1) \rceil} \tag{4.12}$$

To calculate the power metric we have to consider four different cases. A transmission starts out in one of the two spacers, transitions to the code word and finally transitions either to the all-zero or all-one spacer. We denote the number of DI bus transitions involved in each of those cases with t^{zz}, t^{zo}, t^{oo} and t^{oz}. For m-of-n codes these values can easily be calculated:

$$t^{zz} = 2m, \quad t^{zo} = n, \quad t^{oo} = 2(n-m), \quad t^{oz} = n \tag{4.13}$$

If we assume uniformly distributed data for b_s the average number of transitions for one transmission is given by the mean of those four values, which immediately yields the power metric:

$$P^{cw|DS}_{m,n} = \frac{n}{\lfloor \log_2 \binom{n}{m} \rfloor + 1} \tag{4.14}$$

Furthermore, Equation (4.14) shows that for some cases (e.g., for the class of m-of-$2m$ codes) the DS Protocol also improves the power metric.

The same approach is used to derive the power metric for Berger codes. The values for t^{zo} and t^{oz} are straightforward to calculate because these cases involve the switching of

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

all $b + k$ rails. The other two values depend on the actual code word structure, i.e., the value of $\mathbf{p}$:

$$t_{\mathbf{p}}^{zz} = 2(h(\mathbf{p}) + b - [\![\mathbf{p}]\!]), \quad t_{\mathbf{p}}^{oo} = 2(k - h(\mathbf{p}) + [\![\mathbf{p}]\!]) \tag{4.15}$$

This could potentially demand for a case distinction based on the different possible values of $\mathbf{p}$. However, when calculating the mean of the four cases it turns out that all terms containing $\mathbf{p}$ cancel out and one is left with $b + k$. Hence, the final power metric for Berger codes using the DS protocol is given by:

$$P_b^{B|DS} = \frac{b + k}{b + 1} \tag{4.16}$$

Recall that Berger codes are most efficient (in terms of both R and P) if $b = 2^x - 1$ (i.e., 3, 7, 15, 31 etc.) data bits. Thus, one additional bit comes in handy to “fill” up the transmitted data to some multiple of a byte.

4.3.2 Short Distance Spacer Protocol (m-of-n Codes)

We observe that a 4-phase m-of-n code requires m transitions to go from a code word back to the spacer, and another m to transmit the next code word. The basic idea behind the Short Distance Spacer (SDS) protocol is to dynamically select a suitable spacer between two m-of-n code words $\mathbf{c}_n$ and $\mathbf{c}_{n+1}$ based on their Hamming distance $D(\mathbf{c}_n, \mathbf{c}_{n+1})$ in such a way that only d transitions are required to get from $\mathbf{c}_n$ to that spacer, and another d to get from there to $\mathbf{c}_{n+1}$, where $d < m$. Note that, unlike with the DS protocol, here the spacer does not carry any extra information (as it cannot be freely chosen), so the SDS protocol is still considered 4-phase.

Figure 4.5 shows a state graph visualizing this principle. Besides the usual all-zero (i.e., 0-of-n) spacer, the protocol also uses another type of spacer. However, this spacer, which we will refer to as short-distance spacer, is not a single distinct bit pattern, but rather one dynamically chosen from a set of $(m + d)$-of-n code words (i.e., the code $C_{m+d,n}^{cw}$). Starting in the left-most state, the code word $\mathbf{c}_n$ is transmitted by applying m transitions. After acknowledgment the transmitter checks the next code word $\mathbf{c}_{n+1}$ that will be sent, to see whether it could be reached via an $(m + d)$-of-n short-distance spacer. If this is the case the number of transitions to reach $\mathbf{c}_{n+1}$ can be reduced to $2d$. Otherwise the system falls back to the regular all-zero spacer, which ultimately results in $2m$ transitions to reach the next code word.

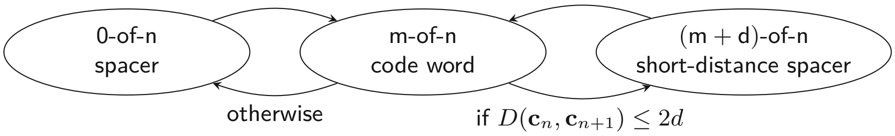


Figure 4.5: SDS protocol state diagram

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Consider the following example, shown in Figure 4.6. Here a DI link using the 3-of-6 code transmits the two code words $\mathbf{c}_n = 000111$ and $\mathbf{c}_{n+1} = 001110$ using the SDS protocol with $d = 1$. Using the normal (single-spacer) RZ protocol this transmission would require nine transitions. However, the SDS protocol is able to leverage the short-distance spacer 001111 to separate the two code words and hence only needs five transitions.

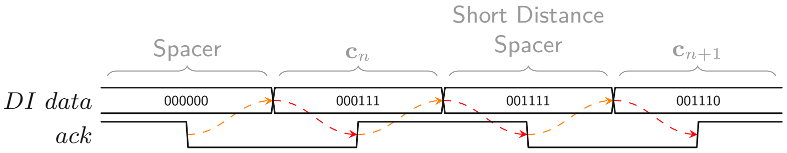


Figure 4.6: SDS protocol example timing diagram

The important question, arising from this concept, is that of the optimal value for d (to achieve the best power metric). Observe that the Hamming distance between two code words in a constant-weight code is always a multiple of two. To calculate the power metric we assume that every code word is equally likely to be transmitted. The number of neighboring code words to any m-of-n code word with a maximum Hamming distance of $2d$ is given by Equation (4.17).

$$N_{m,n,d} = \sum_{x=0}^{d} \binom{m}{x} \binom{n-m}{x} \tag{4.17}$$

This equation has some similarity with Vandermonde's identity. The intuition behind the formula is that the first binomial coefficient provides the number of ways x ones can be selected from the m one-positions in a code word, while the second coefficient yields the number of possibilities how these x ones can be arranged in the remaining $n - m$ zero-positions. Knowing this number we can argue that the percentage p of cases in which the short-distance spacer can be used is given by

$$p_{m,n,d} = \frac{N_{m,n,d}}{\binom{n}{m}}. \tag{4.18}$$

Hence, the power metric $P^{cw|SDS}$ of the SDS protocol is (approximately) given by

$$P^{cw|SDS}_{m,n,d} \approx \frac{2dp_{m,n,d} + 2m(1 - p_{m,n,d})}{\lfloor \log_2 \binom{n}{m} \rfloor} \tag{4.19}$$

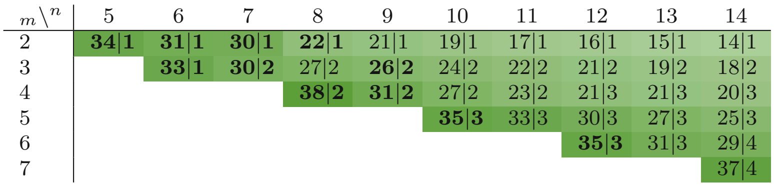
The denominator of Equation (4.19) holds the number encodable bits. Since the binomial coefficient is generally not a power of two only a subset of the actual code words provided by the code is actually used. Note that the selection of this subset obviously has an impact on p, which is disregarded by the equation. A precise way for calculating $P^{cw|SDS}$ is provided by Equation (4.20), where C is the set of used code words. However, for the codes we have examined in this work, the approximation of Equation (4.19) was quite

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

accurate (within a few percent).

$$
\begin{aligned}
P_{C,k}^{cw|SDS} &= \frac{1}{|C|^2} \sum_{\mathbf{c}_1 \in C} \sum_{\mathbf{c}_2 \in C} n(\mathbf{c}_1, \mathbf{c}_2), \\
\text{where } n(\mathbf{c}_1, \mathbf{c}_2) &= \begin{cases} 2d & \text{if } D(\mathbf{c}_1, \mathbf{c}_2) \leq 2d \\ 2H(\mathbf{c}_1) & \text{otherwise} \end{cases}
\end{aligned}
\tag{4.20}
$$

The optimal value for d is given exactly by the number for which P^{SDS} is minimal. Figure 4.7 shows that the improvement for the power metric lies in the range of up to ∼38% for the class of m-of-$2m$ codes. Note that an NRZ protocol leads to an improvement of exactly 50% (disregarding the transitions on the *ack* wire). The bold entries in the figure are exact values for the PSCWCs, or subcodes thereof (as defined in Tables 4.2 to 4.5) discussed in the previous section, the rest are estimates obtained with Equation (4.19). The only exceptions are the 2-of-6 and 2-of-8 codes, which are actually just concatenations of two 1-of-n codes. A 1-of-2 and a 1-of-4 code in the case of former code and two 1-of-4 codes for the latter code.



$m \backslash n$	5	6	7	8	9	10	11	12	13	14
2	**34|1**	**31|1**	**30|1**	**22|1**	21|1	19|1	17|1	16|1	15|1	14|1
3		**33|1**	**30|2**	27|2	**26|2**	24|2	22|2	21|2	19|2	18|2
4				**38|2**	**31|2**	27|2	23|2	21|3	21|3	20|3
5						**35|3**	33|3	30|3	27|3	25|3
6								**35|3**	31|3	29|4
7										37|4

Figure 4.7: Power metric improvement for the SDS protocol (improvement of P [%] | optimal value for d)

It is obvious that this protocol is a little more involved to implement than the RZ, DS or even NRZ protocol. The crucial component in the transmission link is the spacer generator, which basically has two tasks. First, it must determine if a short-distance spacer is applicable to separate the two given code words $\mathbf{c}_n$ and $\mathbf{c}_{n+1}$ or else the system has to fall back on the all-zero spacer. If the short-distance spacer can be used it must then provide an appropriate bit pattern at its output that is element of $C_{m+d,n}^{cw}$. In the simplest case, i.e., if $D(\mathbf{c}_n, \mathbf{c}_{n+1}) = 2d$ the short-distance spacer is obtained by a bit-wise OR operation between the two code words. However, if $D(\mathbf{c}_n, \mathbf{c}_{n+1}) < 2d$, the bit-wise OR produces a bit pattern with a Hamming weight smaller than $m + d$. Hence, there must be some circuitry that allows to set "dummy" zero-positions in this bit pattern in order to get to the required Hamming weight for a valid short-distance spacer. This part of the spacer generator needs a considerable amount of resources, because its hardware overhead is proportional to the maximal number of "dummy" bits, that it must be able to set in a bit pattern. In the worst case (i.e., if $\mathbf{c}_n = \mathbf{c}_{n+1}$) exactly d such dummy positions need to be set.

Hence, one small optimization that can be implemented is not to use the short-distance spacer if the same code word is transmitted twice. This would essentially add the

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

condition $\mathbf{c}_n \neq \mathbf{c}_{n+1}$ to the arc between the code word and the short-distance spacer in the state diagram in Figure 4.5. Assuming uniformly distributed data the exclusion of this case does not have a huge impact on the overall power metric.

4.3.3 Short Distance Dual Spacer Protocol (Berger Codes)

Since there are multiple different values for the Hamming weight of Berger code words, it is also possible to leverage the all-one spacer to reduce the number of bus transitions, instead of transmitting an additional bit of data. Figure 4.8 illustrates this approach, which we refer to as Short Distance Dual Spacer (SDDS) protocol. Whenever the protocol is in the code word (i.e., the middle) state, the Hamming weight of the next code word ($h(\mathbf{c}_{n+1})$) is calculated and compared to the one of the code word that has just been sent ($h(\mathbf{c}_n)$). Based on these values it can then be determined whether it is cheaper (in terms of the number of transitions required) to transition to the next code word through the all-one or all-zero spacer. Note that k again denotes the number of the parity bits (i.e., the width of $\mathbf{p}$).

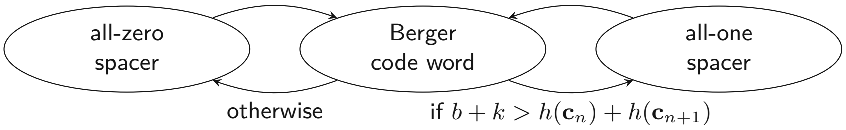


Figure 4.8: SDDS protocol state diagram

Equation (4.21) shows how the power metric of the SDDS protocol is calculated. The equation is quite similar to Equation (4.6). However, here we go through every possible transition with respect to the Hamming weights of the code words involved. The minimum function selects that value, whose corresponding spacer yields the minimum amount of transitions.

$$P_b^{B|SDDS} = \frac{\sum_{0\leq p_1\leq b}\sum_{0\leq p_2\leq b} \min\left(f(p_1,p_2), 2(b+k) - f(p_1,p_2)\right)\binom{b}{b-p_1}\binom{b}{b-p_2}}{2^{b^2} * b}, \quad \text{where } f(p_1,p_2) = h(\langle p_1\rangle) + h(\langle p_2\rangle) + 2b - p_1 - p_2 \tag{4.21}$$

When compared to the RZ protocol, this approach obviously does not affect the coding efficiency. The advantage of this protocol is that it has increased power efficiency and is quite simple to implement, because at least some of the values needed for the spacer-decision (i.e., the Hamming weights of the data parts) already need to be calculated for the encoding process anyway.

4.3.4 Unbalanced Spacer Protocol (Berger Codes)

The Unbalanced Spacer (UBS) can be viewed as the SDS protocol for Berger codes. However, where the spacer for the SDS protocol was basically defined by its Hamming

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub

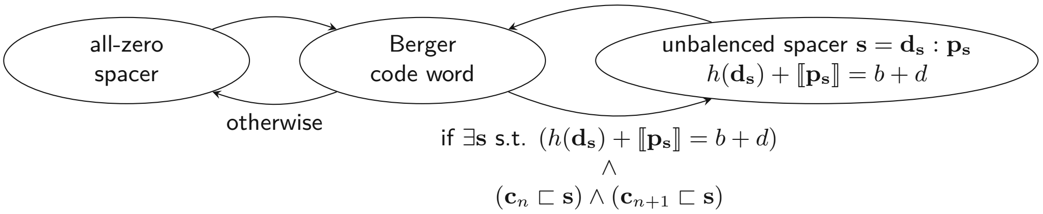

all-zero spacer
Berger code word
unbalenced spacer $\mathbf{s} = \mathbf{d_s} : \mathbf{p_s}$
$h(\mathbf{d_s}) + [\![\mathbf{p_s}]\!] = b + d$
otherwise
if $\exists$s s.t. $(h(\mathbf{d_s}) + [\![\mathbf{p_s}]\!] = b + d)$
$\wedge$
$(\mathbf{c}_n \sqsubset \mathbf{s}) \wedge (\mathbf{c}_{n+1} \sqsubset \mathbf{s})$

Figure 4.9: UBS protocol state diagram

weight, here the spacer definition is a bit more involved. Figure 4.9 shows the state graph of this protocol.

It can be seen that, like the code words themselves, the spacer $\mathbf{s}$ is also divided into a data part $\mathbf{d_s}$ and a parity part $\mathbf{p_s}$. Recall that all code words of a Berger code have a certain balance between the Hamming weight of the data part and the numerical value represented by the parity part (i.e., $h(\mathbf{d}) + [\![\mathbf{p}]\!] = b$, see Equation (4.2)). The spacer $\mathbf{s}$ is now defined as a bit vector for which this balance deviates from the balance of the code words by exactly the value of d (i.e., $h(\mathbf{d_s}) + [\![\mathbf{p_s}]\!] = b + d$). Hence, the name *unbalanced* spacer protocol. The set of all possible spacers for a Berger code with a given b and d is denoted by $S_{b,d}$.

Let's now discuss the condition for when the unbalanced spacer can be used. The first thing a potential transmitter for this protocol has to check is if the balance of the bit pattern obtained by a bit-wise OR of the code words $\mathbf{c}_n$ and $\mathbf{c}_{n+1}$ is less than or equal to $b + d$ (i.e., $h(\mathbf{d}_{\mathbf{c}_n} \vee \mathbf{d}_{\mathbf{c}_{n+1}}) + [\![\mathbf{p}_{\mathbf{c}_n} \vee \mathbf{p}_{\mathbf{c}_{n+1}}]\!] \leq b + d$). Notice that this is a necessary condition that must be fulfilled in order to use an unbalanced spacer. The unbalanced spacer must be a bit vector that contains (in the sense of the unorderedness property) both of the code words $\mathbf{c}_n$ and $\mathbf{c}_{n+1}$, because it must be possible to use only rising transitions to switch from $\mathbf{c}_n$ to $\mathbf{s}$ and then only falling ones to make the switch from $\mathbf{s}$ back to $\mathbf{c}_{n+1}$. Thus, the simplest way to generate such a bit pattern is to use the bit-wise OR of the code words. However, if the balance of this vector is already greater than $b + d$, then there cannot exist a suitable spacer. On the other hand it may be the case that the balance is strictly smaller than $b + d$, which means that some "dummy" bits must be set in order to generate a valid spacer (similar to the spacer generation of the SDS protocol). This is exactly what the condition in Figure 4.9 expresses.

Notice that there are cases where the balance of the bit-wise OR of the code words is smaller than $b + d$, but there still does not exist a suitable spacer. Consider the following example of a Berger code with $b = 4$ (i.e., $k = 3$) and $d = 2$. The bit-wise OR of the code words $\mathbf{c}_1 = 1111 : 000$ and $\mathbf{c}_2 = 1110 : 001$ is $\mathbf{c}_1 \vee \mathbf{c}_2 = 1111 : 001$ (we use the colon to emphasize separation of the data and the parity part). The balance of this bit vector is $b + 1$. Hence, the necessary condition would be fulfilled. However, to get to a spacer we still need to increase this balance by one, which is not possible in this case because the only bits that could be set would increase the balance to $b + 3$ or $b + 5$.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Figure 4.10 shows a comparison between the power metrics of the RZ, DS, SDDS and UBS protocols. The power metric for the UBS protocol has been calculated using a numerical method, which is also the reason why we only have values for $b \leq 20$. For each Berger code with a certain bit width b, the power metric was evaluated for increasing values of d, starting with $d = 1$. The figure shows the first *local* minimum of the power metrics obtained by this process. The corresponding values for are shown in Table 4.6.

Table 4.6: d values used for the power metric evaluation of the UBS protocol

b	3	$4 \leq b \leq 7$	$8 \leq b \leq 9$	$10 \leq b \leq 15$	$16 \leq b \leq 17$	$18 \leq b \leq 19$	20
d	1	2	3	5	6	7	10

Recall that for a single transmission cycle (i.e., a code word and a spacer phase) the DS protocol needs on average $b + k$ transitions. For the SDDS protocol this is the maximum number of transitions required. However, the DS protocol transmits one bit more per transmission cycle. Hence, for values $b < 7$ it is more efficient. The UBS protocol always yields the best results of the four protocols. However, it is still not able to reach the efficiency of the NRZ protocol, and as we will see in Section 4.6 it is also quite expensive to implement, because of its complex encoder (i.e., spacer generator).

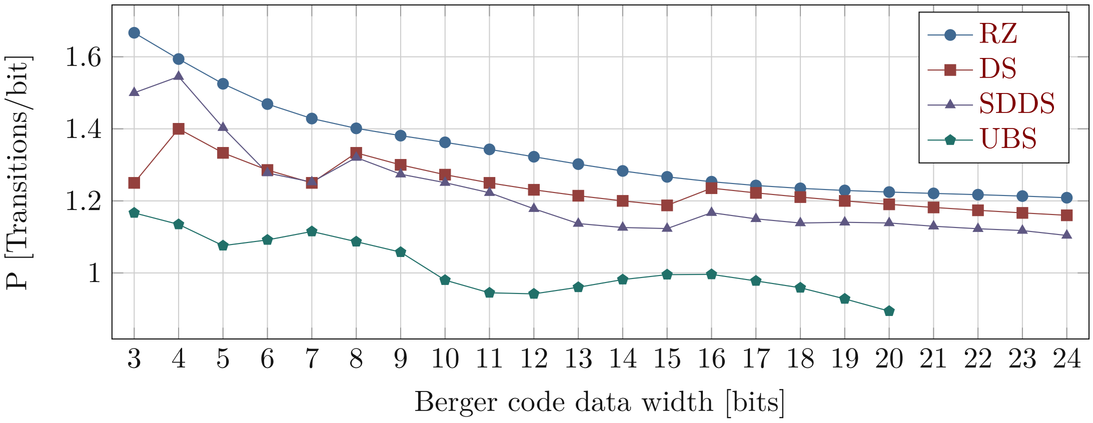


Figure 4.10: Power metric comparison for Berger code protocols (RZ, DS, SDDS and UBS)

4.4 Completion Detection

This section shows how to implement efficient CDs for all codes and protocols discussed in this work. We start out by addressing this problem for the RZ protocol and show how these CDs can also be used for NRZ protocols. Then, we generalize the presented approach to also work with new hybrid protocols.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

The core challenge when implementing CDs is that the resulting circuits must conform to the design rules of the QDI timing model as outlined in Section 2.2. Hence, a paramount concern is that the resulting circuits are free from hazards (i.e., don't produce glitches) and don't contain gate orphans.

A CD for the RZ and the hybrid protocols is a function block that issues a logic one at its (*done*) output, if the bit pattern presented to its input corresponds to a valid code word for some DI code. The CD's output must go to zero when the input constitutes a valid spacer. While the input transitions from the spacer to a valid code word the output must remain at zero. Consequently, it must remain at one during the transition from a code word to the spacer. This implies a hysteresis behavior.

CDs for the NRZ (transition signaling) protocol have a slightly different behavior. Their *done* output must change its state whenever a new set of transitions arrive at their inputs, whose positions constitute a valid DI code word. This value must be kept until the next valid input pattern is detected. With the exception of 1-of-n codes where the NRZ CD is a simple parity function (i.e., cascaded XOR gates)[2], NRZ CDs are usually constructed using 4-phase CDs combined with a 2-phase wrapper circuit [SFGP09, CJN10]. This principle is illustrated in Figure 4.11. For every input rail this wrapper contains one (shadow) latch to store the previous bus state and one XOR gate to detect transitions. Initially the latches are opaque and their output value is equal to the DI bus state $x_0, ..., x_{n-1}$. Input transitions are, hence, converted to rising transitions at the input of the internal 4-phase CD. As soon as the *done* output of the internal CD is asserted the latches are made transparent again, which resets the inputs of the internal CD, effectively issuing a spacer. This again leads to a falling transition on the internal *done* signal prompting the latches to capture the new bus state. The toggle flip-flop generating the actual *done* output changes its state with every *falling* transition on the internal *done* signal. This behavior essentially emulates an RZ protocol for the internal 4-phase CD and artificially introduces the all-zero spacer. Note, however, that this introduces a timing constraint, because it must be guaranteed that the latches are opaque before the next set of transitions arrive at the inputs $x_0, ..., x_{n-1}$.

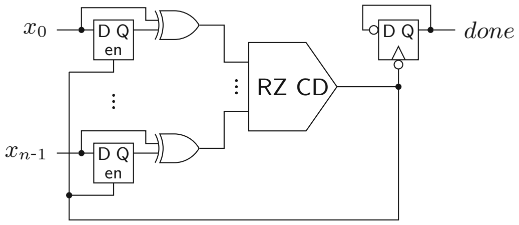


Figure 4.11: NRZ CD constructed from RZ CD with 2-phase wrapper circuit

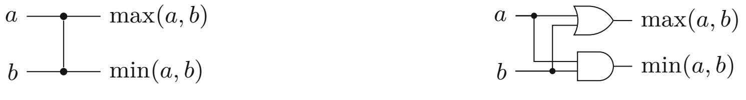
For 4-phase completion detection circuits binary sorting networks (SNs) offer a very generic and efficient design approach [Pie98, HSS15, CJN10]. The idea behind SNs is

---

[2] Cannizzaro et al. also propose a special CD for 2-of-n codes [CJN10]. However, since we don't include these particular codes in our analysis, these circuits are not considered or further addressed.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

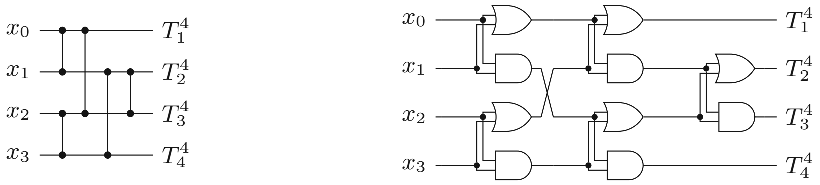
that a set of numbers can be sorted by applying a sequence of predetermined comparison and swap operations to them [Knu98]. This is accomplished by a network of so called comparator cells. A comparator cell, such as the one shown in Figure 4.12a, has two inputs (a and b) and two outputs, where one output generates the maximum of the inputs while the other one generates the minimum. Hence, it basically compares the inputs and swaps them if they are in the wrong order. In the binary case only the (single bit) numbers zero and one have to be distinguished, which is accomplished by an OR and an AND gate (Figure 4.12b).



(a) comparator

(b) binary comparator

Figure 4.12: Comparator cells



(a) Abstract representation

(b) binary implementation

Figure 4.13: T^4 sorting networks

Figure 4.13a shows how these comparators are connected to construct a larger network. We use the notation T^n to denote a SN with n inputs x_0 to $x_{n\text{-}1}$. The outputs are labeled with T_1^n to T_n^n. Figure 4.13a shows the usual abstract representation of a SN, whereas Figure 4.13b shows the gate-level implementation of a binary SN. The output T_k^n of a binary T^n SN is one if at least k inputs are one. The problem of designing optimal SNs for arbitrary number of inputs is still open. However, for a small number of inputs optimal solutions are known. Table 4.7 lists the size (i.e., number of comparators $S(n)$) of the best known SN with minimal depth/delay($D(n)$). For more information on this topic in general, we refer to [Knu98].

Table 4.7: SN implementation costs (minimal depth)

n	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$S(n)$	3	5	9	12	16	19	25	31	35	40	47	52	57	61
$D(n)$	3	3	5	5	6	6	7	7	8	8	9	9	9	9

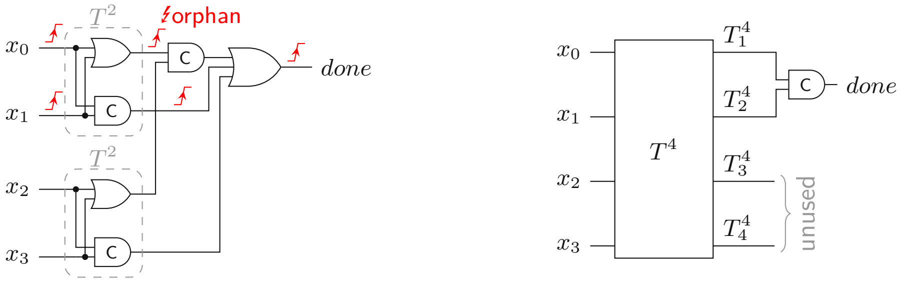
Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.4.1 Constant-Weight Codes (RZ)

The outputs T_1^n to T_n^n of a binary SN can be viewed as the unary encoded Hamming weight of the binary vector presented at its input. This provides exactly the required information to perform completion detection for m-of-n codes. However, a bare binary SN, such as the one shown in Figure 4.13b, is not yet a CD, as it lacks the hysteresis behavior. To construct an m-of-n CD, Piestrak [Pie98] proposes to remove all “unneeded” outputs (i.e., all outputs except T_m^n) of the SN as well as the gates driving them and replace all AND gates with C gates. Alternatively a procedure is provided that directly constructs a CD by using two SNs $T^{\lfloor n/2 \rfloor}$ and $T^{\lceil n/2 \rceil}$ and some appropriate merging logic, which yields similar results. The C gates are required to establish the required hysteresis behavior of the CD. Figure 4.14a shows the resulting CD for a 2-of-4 code.

Unfortunately, this circuit contains orphan transitions. To better understand this issue, consider the case where the input vector 1100 is applied to the circuit. The signals that make transitions to one in this case are marked in the figure. Notice that the top-most OR gate switches to one. However, since no part of the circuit observes (i.e., waits for) this transition before producing an output transition, it constitutes gate orphan. Recall that, as discussed in Section 2.2, gate orphans must generally be avoided in QDI circuits because they conflict with the unbounded delay model.

An alternative approach that does not suffer from this problem is to combine the outputs T_1^n to T_m^n of the T^n SN with an m-input C gate [HSS15]. This has the secondary advantage that the AND gates in the SN don’t have to be replaced by C gates. The hysteresis is solely implemented by the final C gate. The unused outputs T_{m+1}^n to T^n as well as the gates driving them can still be removed from the circuit.



(a) Original CD with orphans

(b) Orphan-free alternative

Figure 4.14: 4-phase 2-of-4 CDs

This is the circuit variant we use as the basis for our proposed solution, that will offer further optimizations. Notice that the T^4 SN in the 2-of-4 CD basically maps every 2-of-4 input code word to the output pattern 1100. However, it is also guaranteed that every 1-of-4 input code word is mapped to 1000. The latter behavior is actually not really required. Hence, the specification of what the SN should do in the CD can be relaxed to: Output the two largest input values at the outputs T_1^4 and T_2^4 in arbitrary order. This is

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

exactly what a *selection network* [Ale69] does. Figure 4.15 shows the general construction for a selection network with $2m$ inputs. The set $\{y_i \mid 0 \leq i \leq m-1\}$ contains the m largest values of the set $\{x_i \mid 0 \leq i \leq 2m-1\}$, while $\{y_i \mid m \leq i \leq 2m-1\}$ contains the m smallest ones. Note that from here on we refer to the characteristic output stage of a selection network as selection network merging logic (SNML). An SNML with $2m$ inputs z_0 to $z_{2m\text{-}1}$ contains m comparators which conditionally swap the inputs z_i and $z_{2m\text{-}i\text{-}1}$ for $0 \leq i < m-1$.

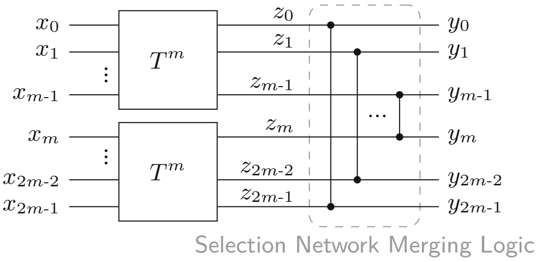


Figure 4.15: Selection network

Using this method we can already construct m-of-$2m$ CDs in a quite efficient way, by connecting an m-input C gate to the outputs y_0 to $y_{m\text{-}1}$. Again the unused outputs can be removed from the circuit (i.e., the AND gate of the SNML driving the outputs y_m to $y_{2m\text{-}1}$). The overhead is similar to the original approach by Piestrak [Pie98], because we also use two T^m SNs for a CD with $2m$ inputs. However, the construction of the merging logic now ensures that there are no orphans in the circuit.

In the following we will generalize this approach for arbitrary m-of-n CDs. Given an m-of-n code, a CD can be constructed by using two SNs T^q and T^r where $q + r = n$, some appropriate merging logic and a single m-input C gate, which will be referred to as the *output C gate*. The inputs to the CD (x_0 to x_{n-1}) are connected to the inputs of the SNs, where q inputs are connected to T^q and the remaining r are connected to T^r (the particular assignment is not relevant).

The outputs of each of the two SNs can be classified into three categories based on their role in the final CD. We define T^x_y as

(i) *unused* if $y > m$

(ii) *certain* if $y \leq x - (n - m)$

(iii) *indicating* otherwise.

An *unused* output can never be asserted, because there are not enough ones in the input code word to ever set this output. This means that it can be removed from the corresponding SN (again with all gates driving it). Since of x inputs to T^x at most $n - m$ can be zero, the rest (if existent) must be asserted for every (valid) input code word.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

These *certain* outputs can, consequently, be directly connected to the output C gate. The *indicating* outputs can, depending on the input code word, be zero or one. However, for each of the networks they are guaranteed to be sorted binary vectors, i.e., vectors encoded with a thermometer code.

For the next steps we define functions to calculate the number of outputs which fall into each of these categories. Let $u(T^x)$, $c(T^x)$ and $i(T^x)$ denote the number of unused, certain and indicating outputs of the SN T^x (Equations (4.22) to (4.24)).

$$u(T^x) = \begin{cases} x - m & \text{if } x > m \\ 0 & \text{otherwise} \end{cases} \tag{4.22}$$

$$c(T^x) = \begin{cases} x - (n - m) & \text{if } x > (n - m) \\ 0 & \text{otherwise} \end{cases} \tag{4.23}$$

$$i(T^x) = x - c(T^x) - u(T^x) \tag{4.24}$$

In the following we will show that the number of indicating outputs is the same for both SNs (i.e., $i(T^q) = i(T^r)$). Moreover, we will show that this number also matches the total number of transitions expected on all indicating outputs, denoted by $I(T^q, T^r)$. This value can be calculated simply by subtracting the number of certain transitions from the total number of input transitions m.

$$I(T^q, T^r) = m - c(T^q) - c(T^r) \tag{4.25}$$

If we can show that $i(T^q) = i(T^r) = I(T^q, T^r)$ always holds, then it is possible to use the indicating outputs to build a selection-network-like structure that outputs the $I(T^q, T^r)$ largest (binary) values of the total $i(T^q) + i(T^r)$ indicating outputs with $I(T^q, T^r)$ comparator cells. This is achieved by merging the indicating outputs of both SNs using the SNML structure shown in Figure 4.15. However, since we are only interested in the $I(T^q, T^r)$ outputs of the merging network that are actually asserted for valid code words, only the OR gates of the comparators are needed.

Without loss of generality we assume that $q \geq r$. The following cases can be distinguished.

(i) $m \leq r$:
$c(T^r) = 0$, $u(T^r) = r - m \Rightarrow i(T^r) = r - (r - m) = m$
$c(T^q) = 0$, $u(T^q) = q - m \Rightarrow i(T^q) = q - (q - m) = m$
$\Rightarrow I(T^q, T^r) = m$

(ii) $r < m \leq q$ (where $r < q$):
$u(T^r) = 0$, $c(T^r) = 0 \Rightarrow i(T^r) = r$
$u(T^q) = q - m$, $c(T^q) = m - r \Rightarrow i(T^q) = r$
$\Rightarrow I(T^q, T^r) = m - c(T^q) = r$

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

(iii) $m > q$:
$u(T^r) = 0,\ c(T^r) = r - (n - m) \Rightarrow i(T^r) = n - m$
$u(T^q) = 0,\ c(T^q) = q - (n - m) \Rightarrow i(T^q) = n - m$
$\Rightarrow I(T^q, T^r) = n - m$

This gives evidence that in all three possible cases we have $i(T^q) = i(T^r) = I(T^q, T^r)$, which is exactly what we wanted to prove.

Figure 4.16 shows the general overview of the proposed CD, where the SN T^q has certain, indicating and unused outputs. Note that, according to the provided proof, for every valid code word and every intermediate input pattern (with less than m ones), there can only be one of the inputs of each OR gate in the SNML set to one. This means that the proposed circuit is free from gate orphans.

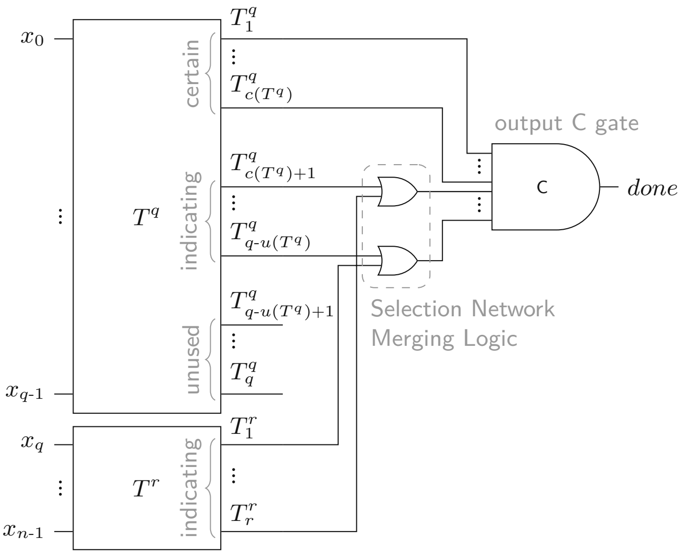


Figure 4.16: Proposed m-of-n completion detection approach

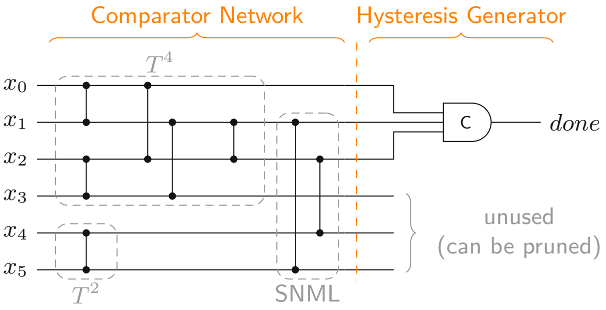
The proposed construction approach ensures that the resulting circuits can always be separated into a block composed *solely* of binary comparator cells, which we refer to as the Comparator Network (CN) and a block that implements the hysteresis behavior, called the Hysteresis Generator (HG). The HG takes some outputs of the CN and generates the *done* output. The other outputs of the HG can be pruned (i.e., the gates driving them can be removed). While this observation seems trivial for the case of m-of-n CDs, we will see this holds true for every other CD presented in this work. Moreover, it enables us to present CDs in an abstract unified form (see Figure 4.17a for an example). This also allows for the implementation of a single algorithm that finds a suitable mapping of the CN to a gate-level circuit, that minimizes the usage of non-inverting gates (i.e., AND and OR gates), automating the CD generation process.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

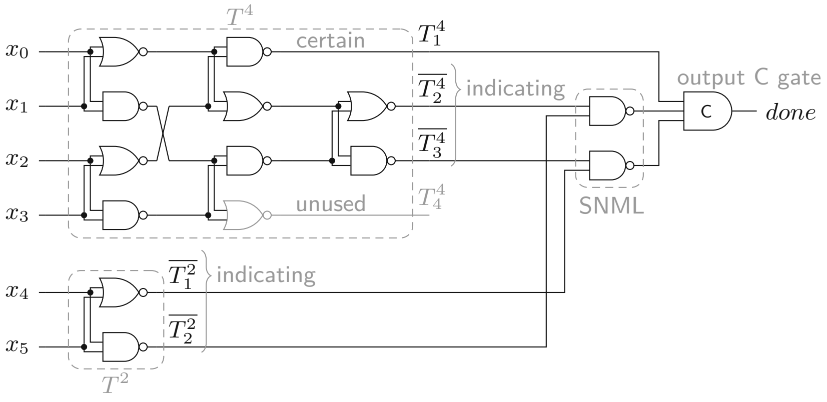
To optimize for a low transistor count and delay the CN should be implemented predominantly with NAND and NOR gates. In our analysis we observed that SNs with an even number of inputs can often be implemented more efficiently, since because of their symmetrical structure no additional inverters inside the network are required. Hence, if $\frac{n}{2}$ is an odd integer, it is beneficial to use an SN partition with $q = \frac{n}{2} + 1$ and $r = \frac{n}{2} - 1$. On top of that it is also often the case that the costs for two identical SNs of some particular *uneven* size m are higher (in terms of comparators) than the combination two SNs of sizes $m + 1$ and $m - 1$. To illustrate that consider the example of a CD for the 5-of-10 code. Two T^5 SNs require 18 comparator cells. However, a T^4 combined with a T^6 only need 16. Furthermore, we know the T_1^6 is a certain output, and is hence directly connected to the HG, which simplifies the SNML.

Figure 4.17b shows another example CD for the 3-of-6 code. Here the partition $q = 4$ and $r = 2$ was chosen. Notice that the circuit does not contain any explicit inverters.

The proposed fully generic CD construction approach yields circuits that are guaranteed to be QDI (i.e., free from gate orphans). At least for the class of m-of-$2m$ codes the generated CD are also the most area-efficient in literature (to the best of our knowledge).



(a) Abstract representation



(b) Optimized gate-level implementation

Figure 4.17: Proposed 3-of-6 CD ($q = 4$, $r = 2$)

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.4.2 Berger Codes (RZ)

Piestrak also proposed a SN-based CD for Berger codes, which is shown in Figure 4.18. The basic idea behind this circuit is that a SN is used to determine the Hamming weight of the data part **d** of the code word, while the Unate Product Generator (UPG) sets the signals $w_1, ..., w_b$ according to the value of the parity bits **p**. For this purpose the signal w_i is generated by a conjunction over those rails of **p**, which are set if **p** carries the binary representation of i (e.g., w_5 is generated by a C gate over the inputs p_0 and p_2). Note that for every $T^b_{h(\mathbf{d})}$ asserted by the SN for a certain Hamming weight of **d**, a corresponding $w_{b-h(\mathbf{d})}$ will eventually be asserted by the UPG. The C gates are used to detect these conditions. Their outputs are connected to an output OR gate generating the *done* signal. For the two special cases T^b_b and w_b, there is no corresponding signal from the respective other block. Hence, these signals are directly connected to the OR gate.

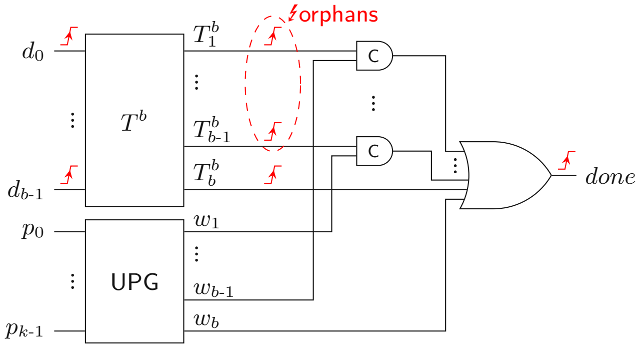


Figure 4.18: CD for Berger codes by Piestrak [Pie98]

However, as with the m-of-n CD discussed in the previous section, there is a similar problem with orphans in this circuit. Notice that, if the data part of a code word has a certain Hamming weight h, none of the outputs $T^m_x|_{x<h}$ of the SN is observed by any part of the circuit. Thus, transitions occurring on them constitute orphan transitions. A similar problem arises in the UPG, but we won't go into further detail on that because our proposed CD does not use this component anyway. Figure 4.18 shows the extreme case where the CD processes a code word, whose data part only contains ones.

An overview of our proposed CD architecture is depicted in Figure 4.19. It uses the same basic idea as discussed in the previous section. The data part **d** is processed by the T^b SN at the top that fulfills the same purpose as in Piestrak's design, giving us a unary encoding of the Hamming weight of **d**. The bottom block $BUC^{2^k\text{-}1}$, referred to as the binary-to-unary converter (BUC), is connected to the parity bits **p** and yields a unary representation of the binary value carried by **p**. For now assume that the BUC is itself implemented as a SN with $2^k - 1$ inputs where each rail p_i is connected to the exact number of inputs of this SN that represents its binary value 2^i (i.e., p_i is connected to 2^i inputs).

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

From the definition of the Berger code we know that the sum of the Hamming weight of **d** and the binary value represented by **p** must be b. Therefore, we again have the situation that there are two sorted binary vectors (i.e., unary encoded values) of length b where exactly b bits must be one for valid code words. This means that in order to generate the final output of the CD an SNML is connected to the b outputs of the SN and the BUC. The outputs of the resulting CN are then fed into a b-input C gate representing the HG. We thus need b comparator cells between the signals T_i^b and $T_{b+i-1}^{2^k-1}$ for $1 \leq x \leq b$, from which only the OR gates remain after pruning. Again, it is important to stress that for every valid code word and every intermediate input pattern only one of the inputs to each of these OR gate can be one. Every internal transition is observed by this circuit; thus, it is free from orphans.

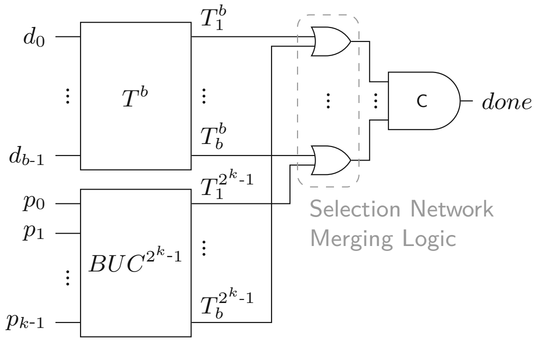
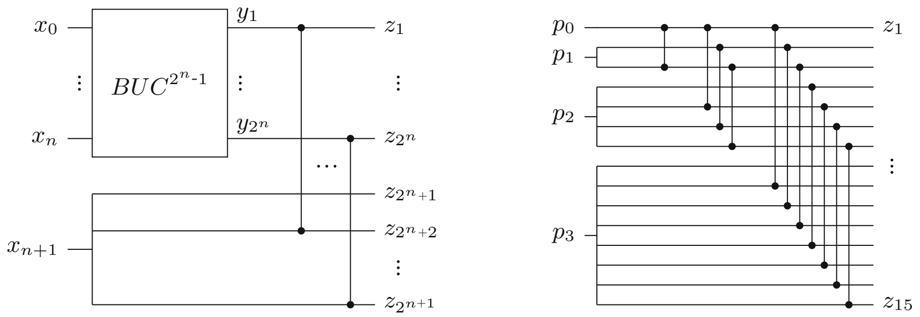


Figure 4.19: Orphan-free CD for Berger codes

From a functional point of view this CD design works. However, the implementation of the BUC is highly inefficient and needs to be improved. Consider the following inductive definition of a BUC using a special CN. Converting a single-bit number x_0 to unary is trivial. Assume we have a BUC with the inputs x_0 to x_n (where x_n is the most significant bit (MSB)) and the outputs y_1 to y_{2^n}. To extend this circuit to also process the input signal x_{n+1}, we need to add $2^{n+1} - 1$ comparators as illustrated in Figure 4.20a. We denote the new outputs of the resulting circuit with z_1 to $z_{2^{n+1}}$. To generate the outputs z_i and z_{2^n-i+2} we need the maximum and minimum output of the comparator connected to y_i and $x^{2^{n+1}}$ for $1 \leq i \leq 2^n$. The output z_{2^n+1} is generated directly from the input $x^{2^{n+1}}$. Note that the newly added layer of comparators basically performs a unary addition of the unary vector **y** and the newly created unary vector which can only hold the values 0 or 2^{n+1}. Figure 4.20b shows an example 4-bit BUC represented as a CN.

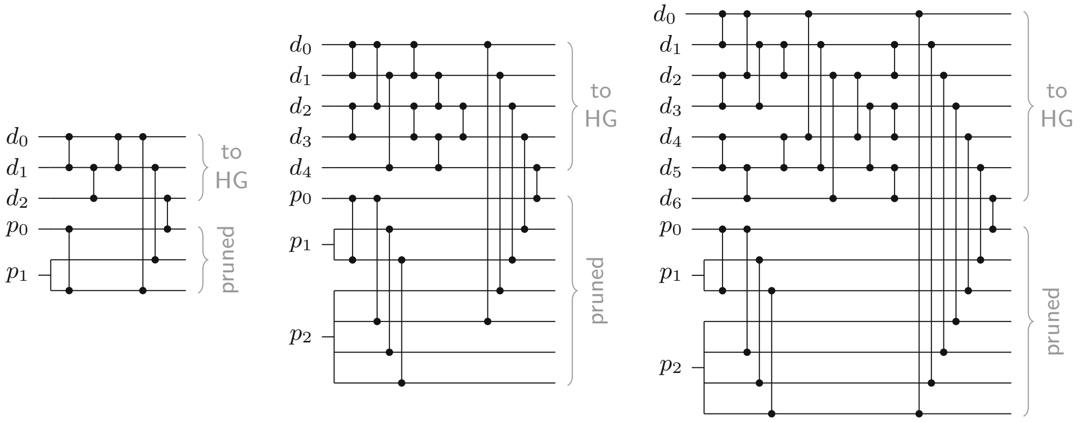
To further illustrate the construction approach Figure 4.21 shows three Berger code CD examples represented by their CNs. The presented technique is completely generic and guarantees fully QDI CD circuits, which are (to the best of our knowledge) the most area-efficient in literature for this class of codes.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.



(a) Inductive BUC construction (b) 4-bit BUC example

Figure 4.20: CN-based BUC



(a) $b = 3$ (b) $b = 5$ (c) $b = 7$

Figure 4.21: Abstract CN specification for Berger CDs with 3, 5 and 7 data bits

4.4.3 Hybrid Protocols

Now, to extend the CDs proposed in the two previous sections to also cope with the hybrid protocols, we need to be able to detect the second spacer (or set thereof). We will first show how this works for m-of-n codes and then generalize the approach to Berger codes.

Again, consider the circuit in Figure 4.16 with a valid m-of-n code word at its input. In this case all *certain* outputs of the SNs T^q and T^r are one and *exactly* one input of every OR gate in the SNML is asserted. Now we assume that the input transitions to the special spacer. Therefore, by the construction of the circuit, for every additional one that appears at the input one of two things can happen:

(i) An additional *indicating* output goes high

(ii) An *unused* output on one of the SNs goes high

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Finally, if all bits of the input vector were set to one (as would be the case for the all-one spacer) all of the outputs of the two SNs T^q and T^r are set to one. Thus, every (previously) *unused* output and every OR gate input is asserted.

Note that case (i) implies that the additional one in the input bit pattern causes the assertion of both inputs of exactly one of the OR gates in the SNML. This condition can easily be detected if we don't prune the AND gates of the SNML.

Hence, for detecting $k \leq (n-m)$ additional ones in the input pattern we propose to use a second-level CD connected to the AND gates of the SNML and the previously *unused* outputs (if present). For that the following cases have to be distinguished:

(i) In the simplest case no SN has *unused* outputs. Then, we basically only have to connect another k-of-i CD to the outputs of the i AND gates of the SNML that would otherwise have been pruned from the circuit.

(ii) In the second case, namely when T^q is the only SN with *unused* outputs, we can simply use a k-of-j CD to which we connect the i AND gates as before, plus up to k of the $u(T^q)$ originally *unused* outputs of T^q, i.e., $j = i + \min(k, u(T^q))$.

(iii) Finally, if both T^q and T^r have *unused* outputs, care must be taken because some of the *unused* outputs might only be asserted in a mutually exclusive way. These can be merged by an OR gate (i.e., a comparator) before being connected to the second-level CD. Consider the case of a CD for the 2-of-7 code with $q = 4$ and $r = 3$. Hence, T_3^4, T_4^4 and T_3^3 are *unused*. If this CD is extended to an SDS CD with $d = 1$, the outputs T_3^4 and T_3^3 could never be asserted at the same time, and can consequently be merged.

We use $done_2$ to refer to the output of the second-level CD, which is again generated by a C gate. This signal needs to be merged with the output of the original CD, which we now refer to as $done_1$ into the final *done* output of the hybrid protocol CD. Here we need to distinguish three cases.

(i) all-zero spacer: $done_1$ is low (which implies $done_2$ is low as well); *done* has to be zero

(ii) special spacer: $done_1$ and $done_2$ are both high; *done* has to be zero

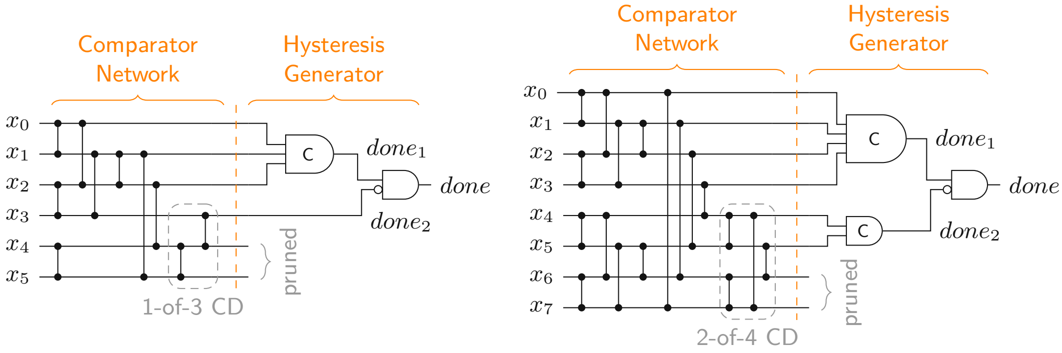
(iii) valid data: $done_1$ is high and $done_2$ low; *done* has to be one

This behavior can be implemented using a simple AND gate with the $done_2$ input inverted. Note that the case where $done_2$ is high and $done_1$ is low can never occur.

Figure 4.22 shows two example CDs for the SDS protocol. Note that it is again possible to make a clean distinction between the CN and the HG. The 3-of-6 CD constitutes a

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

special case, where no second C gate is required. Since here $d = 1$ the second-level CD only needs to detect a 1-of-3 code, which can be implemented by a three input OR gate. Another special case are CDs for the DS protocol, where only the all-one spacer needs to be detected. Hence, it is sufficient to connect the second C gate to all *unused* outputs of the SNs as well as all AND gates of the SNML to generate the $done_2$ signal, because this essentially creates an $(n\text{-}m)$-of-$(n\text{-}m)$ CD.



(a) 3-of-6, $d = 1$

(b) 4-of-8, $d = 2$

Figure 4.22: CD examples for the SDS protocol

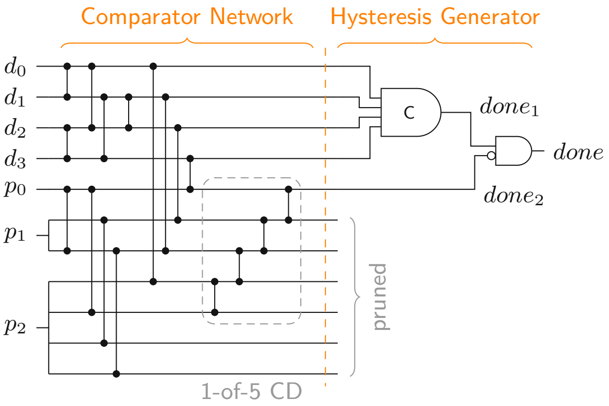
For Berger codes a very similar approach can be used. Let's first consider the DS protocol. Instead of pruning the respective base CN (see Figure 4.21), we use a $2^k - 1$-input C gate to combine all these previously pruned outputs signals into the signal $done_2$. Note that it is not possible to prune any of the outputs in this case, because it must be possible to detect the case where *all* bits in the parity part $\mathbf{p}$ are set to one. If we would, for example, only use the AND gate outputs of the SNML, gate orphans would be introduced.

For the UBS protocol a second-level d-of-x CD is added to the AND gate outputs of the SNML and some of the outputs of the (previously) unused and pruned outputs of the BUC. The variable x is given by the maximal numerical value the parity part of all possible spacers for a given code can take (i.e., $x = \max_{\mathbf{d_s}:\mathbf{p_s} \in S_{b,d}}(\llbracket \mathbf{p_s} \rrbracket)$), while d again denotes the chosen imbalance between the code words and the unbalanced spacer. Note that outputs of the BUC that were previously unused, must be *directly* connected to the second-level CD, since a one at these outputs directly contributes to the spacer balance. Figure 4.23 shows two example CDs for the UBS protocol.

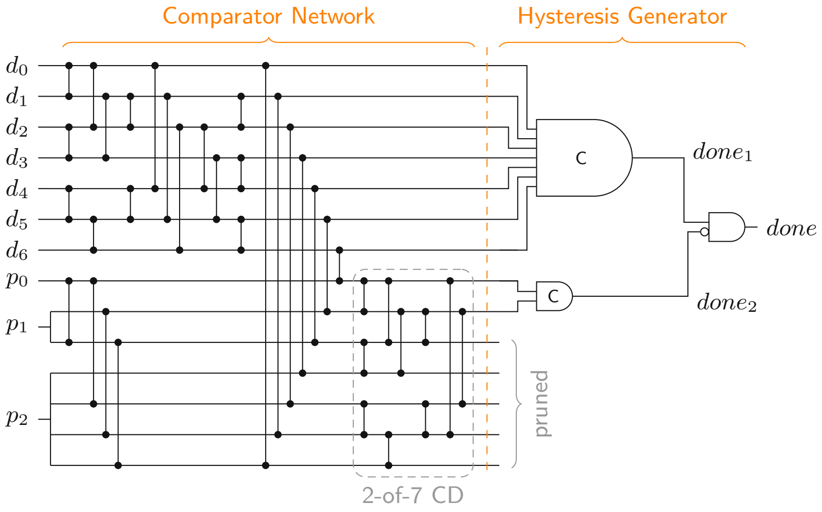
4.5 Link Architecture

This section briefly discusses how the proposed protocols impact the transmitter, receiver and repeater (i.e., buffer) design of a (pipelined) DI link, such as the one shown in Figure 4.1. We assume that the protocols have to be converted to and from 4-phase BD channels. However, adapting the circuits to 2-phase BD protocols is quite straightforward,

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub



(a) $b = 4,\ d = 1$



(b) $b = 7,\ d = 2$

Figure 4.23: CD examples for the UBS protocol

mostly only the presented STGs have to be revised to accommodate the protocol change. Note that we don't claim that these circuits are in any way optimal, we just want to (i) show that the protocols can actually be implemented and (ii) have some basis for the area estimations, we conduct in Section 4.6. For that purpose, we try to take similar design decisions for all the circuits.

4.5.1 Pipeline Design

The first point we want to address is the actual pipeline architecture that can be used to implement intermediate stages on a link. Since the hybrid protocols don't use a single spacer, it is no longer possible to use 4-phase pipeline buffers like the WCHB. What is actually needed is a circuit capable of transporting 2-phase protocols. One possilbe

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

option is a MOUSETRAP-style [SN07] pipeline, which has also been utilized for the 2-phase LETS approach [MAMN08]. Instead of C gates like in the WCHB this buffer uses D latches, whose enable input is controlled by an XNOR gate (see Figure 4.24). For this reason we will refer to this buffer as MOUSETRAP-style D latch half buffer (MDHB). Initially the latches are transparent, but are disabled as soon as data (or a spacer) arrives. To re-enable the latches the subsequent pipeline stage must acknowledge the received data (or spacer), by toggling the acknowledgment signal. This behavior implies a small timing assumption, because it must be ensured that the latches of a stage are closed before the preceding stage can invalidate the latch inputs. Notice that these two actions are triggered by the same signal, namely the output of the CD.

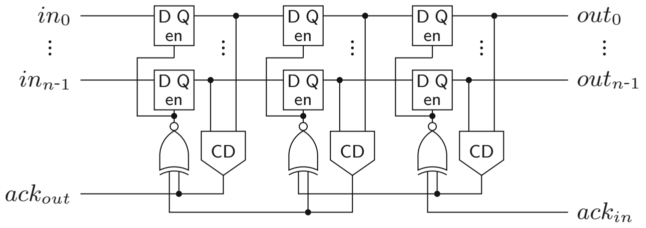


Figure 4.24: Pipeline implementation for proposed protocols (three stages)

4.5.2 RZ Link

We start with the “base-line” design for the RZ protocol. Figure 4.25 shows a possible transmitter/receiver pair. Consider the circuit in the reset state, i.e., all req_{in} and ack_{out} signals and the output register $\mathrm{R_{out}}$ contains the (all-zero) spacer. A rising transition on the transmitter’s req_{in} signal will thus set the C gate. This event is used to produce the acknowledgment for the BD input channel as well as to trigger the output register $\mathrm{R_{out}}$, which will thus be loaded with the data produced by the encoder. Eventually this data gets acknowledged (rising transition on ack_{DI}), which, if req_{in} has already been deasserted by the BD channel, in turn triggers the reset of the register (through the pulse generator formed by the delay δ_p and the AND gate). This essentially produces the all-zero spacer on the DI bus, which will again be acknowledged by a falling transition on ack_{DI}. After the C gate is reset the BD ack_{out} signal will be deasserted and the whole process may start over. The receiver works in a quite similar fashion. When the CD detects a valid DI code word on the DI bus, the receiver’s C gate is set to one (assuming ack_{in} is zero). This transition is used to capture the DI data into the input register $\mathrm{R_{in}}$, produce the acknowledgment for the DI link as well as to generate the req_{out} signal for the BD channel. The C gate will be reset again if the CD detects the spacer and if the BD side acknowledges the (decoded) output data ($ack_{in} = 1$). This produces the falling transitions on ack_{DI} and req_{out}, which in turn leads to the deassertion of ack_{in} on the BD side. The delay elements δ_{enc} and δ_{dec} ensure that the request signal is sufficiently delayed such that there is enough time tor the data to pass through the encoder and decoder, respectively.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

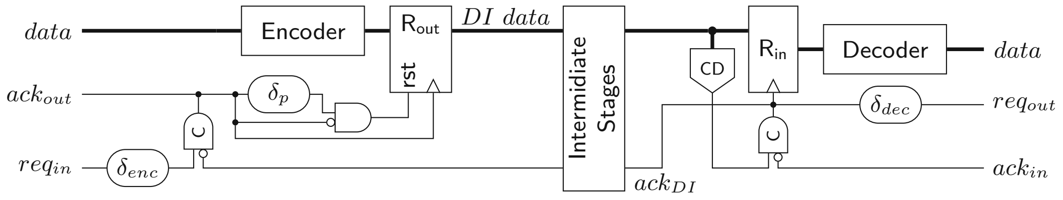


Figure 4.25: RZ protocol transmitter and receiver

Note that no actual decoder is required if the link uses a Berger code. Furthermore, there is no need for the receiver to capture the parity bits into its input register, further simplifying the circuit.

4.5.3 SDS/UBS/SDDS Link

The transmitter for the SDS and the UBS protocol is a little more tricky to implement than the RZ transmitter. Figure 4.26a shows a high-level overview of a possible transmitter circuit. The behavior of the controller is defined by the STG in Figure 4.26b. Let's first disregard the *reset controller* (i.e., the signal r is low) and assume that the circuit and *controller* are in a state where a valid code word is in the output register $\mathrm{R_{out}}$. Hence, ack_{DI} will eventually be asserted by the environment (this state is indicated by the initial marking in the STG). Now the controller waits for the next input data, i.e., a rising edge on the req signal. As soon as this edge is received the controller sets the trg output to one, which switches the multiplexer to the spacer path. The delay element δ_{enc} ensures that trg reaches the pulse generator (formed by the XOR gate and the delay element δ_p) only after *data* passed through the encoder, the spacer generator and the multiplexer and a valid SD spacer (if one could be generated for the two code words) is stable at the input of $\mathrm{R_{out}}$. If no spacer could be generated the spacer generator asserts its z output, in this case the actual value of the spacer output does not matter. Depending on the value of the signal z, the pulse that is generated at the output of the XOR gate is either relayed to the clock or the reset input of the output register. A pulse on the clock input transfers the SD spacer to the output of $\mathrm{R_{out}}$ while a reset pulse effectively generates the all-zero spacer. The spacer at the output *DI data* will cause the environment to eventually deassert ack_{DI}, which in turn causes the controller to respond by also resetting trg. This causes the multiplexer to switch to the next code word (i.e., the output of the encoder). The zero value on the control input of the demultiplexer ensures that the generated pulse will clock the output register, which results in the next code word appearing at *DI data*. After completing the input handshake ($ack+ \rightarrow req- \rightarrow ack-$) this process can start over. To optimize the cycle time of this circuit the delay element δ_{enc} can be implemented in an asymmetrical way, since for falling transitions on trg only the delay of the multiplexer must be compensated for.

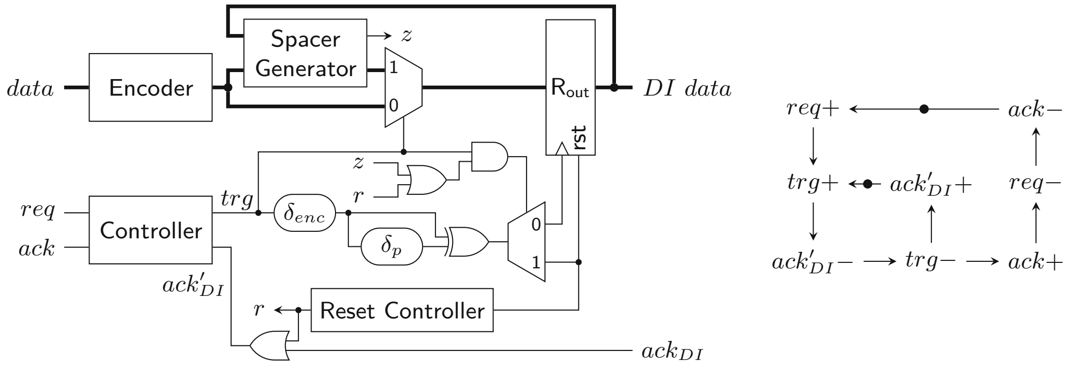
The thing that complicates the circuit is the reset controller which ensures correct start-up of the protocol. As can be seen from the STG the controller expects that initially the

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

circuit is in a state where a code word is present in R$_{\text{out}}$ and ack_{DI} is high. However, on reset we don't yet have a code word and hence ack_{DI} is also low. Furthermore, the first task the controller will execute is to reset R$_{\text{out}}$ to generate "another" (all-zero) spacer. The reset controller is, thus, used to "emulate" the circuit state expected by the controller and uses an OR gate to force ack_{DI} to a high level. Furthermore, it is ensured that the first pulse that will be generated is relayed to the reset input of R$_{\text{out}}$. After the first pulse the signal r is permanently set to low. This leads to ack_{DI} going low, fulfilling the STG specification and completing the start-up phase.

An interesting observation is that the receiver for the SDS/UBS protocol is not affected by the more complex protocol. The event that triggers the consumption of the received data is still the rising edge of the CD's output, the spacers themselves don't carry any data information and can, hence, be ignored completely behind the CD.

The transmitter for the SDDS protocol is quite similar. The main difference is that the spacer generator only has the z output. Thus, the multiplexer is not required. Furthermore, the output register now also needs an asynchronous set input (to generate the all-one spacer). The signal z is then used to decide, whether to generate a set or reset pulse for the output register (similar the DS transmitter presented in the next section).



(a) Circuit

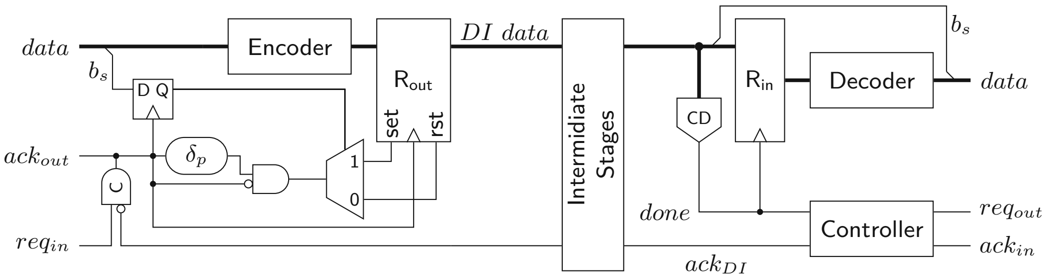
(b) Controller STG

Figure 4.26: SDS/UBS protocol transmitter

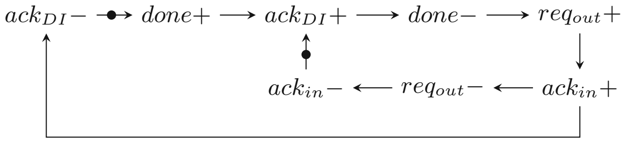
4.5.4 DS Link

A possible transmitter/receiver pair for the DS protocol is shown in Figure 4.27a. The transmitter circuit is simpler than for the SDS protocol because here the spacer does not depend on the next code word being transmitted. The different spacers are generated by using an output register (R$_{\text{out}}$) with asynchronous set and reset inputs that are activated based on the value of b_s. One thing to point out is that the bit b_s needs to be captured with the same clock signal that is used to trigger the output register. This is because after the assertion of ack_{in}, the BD input channel is allowed to invalidate the input data. To control the sequence of events in the circuit a simple C gate suffices. Its rising output

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub



(a) Circuit



(b) Controller STG

Figure 4.27: DS protocol transmitter and receiver

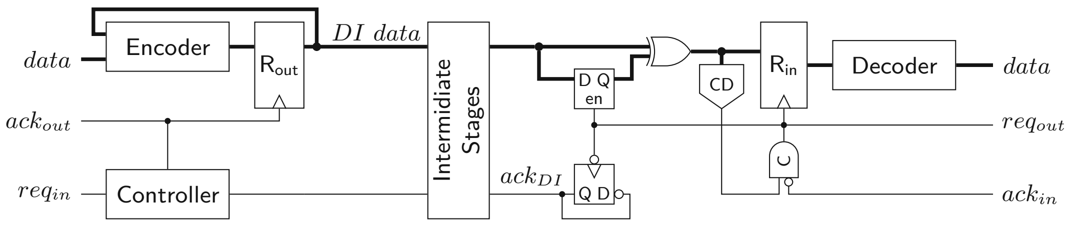
edge clocks R$_{\text{out}}$, while the falling one is used to generate a pulse that is either applied to the set or reset input of R$_{\text{out}}$.

The receiver uses the *done* output of the CD to trigger its input register R$_{\text{in}}$. The controller specified by the STG in Figure 4.27b acknowledges the data phase and waits for the spacer. When the spacer arrives the output handshake ($req_{out}+ \rightarrow ack_{in}+ \rightarrow req_{out}- \rightarrow ack_{in}-$) is initiated. As soon as the preceding logic asserts ack_{in} the spacer can be acknowledged (deassertion of ack_{DI}) and the whole process can start over. Note that we have omitted the delay elements on the BD channels for both transmitter and receiver for the sake of clarity of the figure.

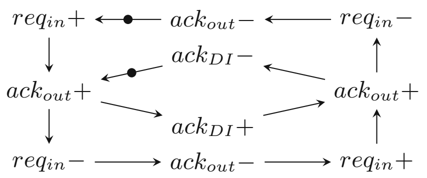
4.5.5 NRZ Link

Finally Figure 4.28a shows a possible NRZ link. The transmitter controller STG in Figure 4.28b basically performs a 4-phase/2-phase conversion between the BD input channel and the ack_{DI} signal. Note that the encoder needs the last state of the DI data, because information is only encoded in the transitions. Internally the encoder essentially uses an RZ encoder and an array of XOR gates for the transition encoding. The receiver on the other side very closely resembles that of the RZ protocol. The only difference is the 2-phase/4-phase conversion (D latches and XORs) in front of the 4-phase CD (see Figure 4.11). The toggle flip-flop again converts the 4-phase *done* signal of the (4-phase) CD to the 2-phase ack_{DI} of the link. Note that the input register already captures a 4-phase code word. Thus, the decoder is the same as for the RZ protocol.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.



(a) Circuit



(b) Controller STG

Figure 4.28: NRZ protocol transmitter and receiver

4.6 Results

There is no single, globally optimum solution for a DI protocol and encoding. Each choice has its specific place within the parameter space spanned by coding efficiency, power metric, area overhead and data throughput. Ultimately, the application needs to determine the most desirable region within this space. In the previous sections we have already investigated coding efficiency and power metric. While that was possible on a purely abstract level, area overhead and data throughput will be studied in this section, based on implementation examples.

4.6.1 Area Analysis

The synthesis results and area estimations in this section are generated using the NanGate 45 nm Open Cell Library. However, to abstract away from the library details, we use the gate equivalents (GEs) metric, which relates the actual area to the one of a single 2-input NAND gate. Encoders and decoders have been synthesized from Very High Speed Integrated Circuit Hardware Description Language (VHDL) descriptions with the Synopsys Design Compiler (version 2018.06), with high effort on area optimization (we only consider the pre-layout results for our analysis). The CDs are already generated on the gate level by our CD construction approach, hence no logic synthesis is required to estimate their area overhead. Since the library does not contain C gates, we assumed an area overhead of 3 GE (12 transistors) for a 2-input version of this gate [SEE98]. For multi-input C gates we further assume an implementation using a single 2-input C gate (as state-holding element) which is set and reset with two carefully routed AND/OR networks.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Table 4.8 lists the hardware costs for the encoders and decoders[3] for all codes analyzed in this thesis. Table 4.9 provides the accompanying information for the respective CDs. The numbers in parentheses in the Berger code rows denote the number of data bits b and parity bits k, respectively. All values given use the GE/bit metric, because this makes it easier to compare codes with different bit widths.

Let's first concentrate on the encoders and decoders. For the RZ and the NRZ protocols, it can be seen that the encoders for the PSCWCs are always more expensive than for a Berger code with the same bit width. Furthermore, since Berger codes are systematic no decoders are required. However, the table also shows that the PSCWCs codes generally have a better coding efficiency R (with the exception of the 5-of-10 and 7-bit Berger code) and as can be seen in Table 4.9 also have smaller CDs. The decoders for the PSCWCs are also considerably simpler than their respective encoders.

The values for the SDS, UBS and SDDS protocols also include the logic for the spacer generation. The encoder costs for the SDS and UBS protocol require very similar hardware efforts for codes with a certain bit width. This also holds true for different values of the parameter d. It is obvious that these protocols require a very large amount of additional logic when compared to (simple) RZ or even NRZ encoders. However, their CD costs are still below that of the NRZ protocol. Another interesting fact is that the encoders for the SDDS protocol are only marginally more expensive than the ones for the RZ protocol.

Note that we did not include the encoding costs for the DS protocol. Recall that this protocol basically uses the exact same encoder as the RZ protocol but can encode one additional bit via the use of a special output register. Since this table does not include the costs for the output register, we did not include the values because they would give a skewed picture of the actual costs[4].

The CD implementation costs in Table 4.9 always list two values per entry. The first one corresponds to the combinational costs, i.e., mainly the CNs and the XORs for the NRZ CDs, while the second one includes the costs for the C gates and the latches in case of the NRZ CDs. It is immediately apparent that the NRZ CDs require the most logic, since the 2-phase/4-phase wrapper circuit basically adds an additional D latch and XOR gate for every input rail. Also notice the entries for the DS and SDDS protocols. These protocols use the exact same CD. However, the values for the DS protocol are smaller because one additional bit of data can be transported.

With the link architecture established in Section 4.5 we now want to calculate the total combined link costs for each protocol and code. This not only includes the encoder, decoder and CD costs but also the overhead for input and output registers and pipeline stages. However, in this analysis we don't include the static costs for the control logic of the links (i.e., controllers, delay-lines, etc.), since these costs are very similar for all the

---

[3] Recall that the decoders are always the same regardless of the protocol.

[4] To some extent this argument also applies to the SDDS protocol.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Table 4.8: Hardware overhead for encoders and decoders

Code	# rails	# bits	R	Encoder overhead [GE/bit]						Decoder overhead [GE/bit]
				RZ	SDS/UBS (d) 1	2	3	SDDS	NRZ	
PS 3-of-6	6	4	0.67	3.67	14.67	–	–	–	7.33	1.67
PS 4-of-8	8	6	0.75	6.61	16.44	18.94	–	–	9.28	4.89
PS 5-of-10	10	7	0.70	5.33	16.14	18.67	20.52	–	8.52	1.71
PS 6-of-12	12	9	0.75	6.63	16.33	18.78	20.48	–	10.33	4.63
Berger (3,2)	5	3	0.60	2.22	12.33	–	–	2.56	5.67	0.00
Berger (4,3)	7	4	0.57	2.75	16.33	19.58	–	3.42	6.50	0.00
Berger (5,3)	8	5	0.62	2.87	15.40	17.93	–	3.33	6.47	0.00
Berger (6,3)	9	6	0.67	3.39	16.06	19.67	–	3.56	7.11	0.00
Berger (7,3)	10	7	0.70	3.33	16.05	18.90	–	3.86	7.00	0.00
Berger (8,4)	12	8	0.67	4.04	17.04	19.62	20.88	5.25	7.25	0.00
Berger (9,4)	13	9	0.69	3.67	16.30	18.63	20.85	4.41	6.85	0.00

Table 4.9: Hardware overhead for CDs

	CD overhead [GE/bit] (combinational/sequential costs)						
Code	RZ	SDS/UBS (d) 1	2	3	SDDS	DS	NRZ
PS 3-of-6	3.25/1.50	4.67/1.50	–	–	–	3.53/2.40	6.25/6.50
PS 4-of-8	4.11/1.56	5.39/1.56	6.17/2.06	–	–	4.43/2.67	6.78/6.00
PS 5-of-10	5.43/1.43	6.90/1.43	7.76/1.86	7.95/2.29	–	5.71/2.50	8.29/6.19
PS 6-of-12	6.07/1.33	7.44/1.33	8.37/1.67	8.44/2.00	–	6.30/2.40	8.74/5.78
Berger (3,2)	4.00/2.00	6.00/2.00	–	–	5.67/4.00	4.25/3.00	7.33/7.56
Berger (4,3)	5.42/2.33	7.92/2.33	10.58/3.08	–	8.25/5.50	6.60/4.40	8.92/8.17
Berger (5,3)	6.53/2.00	8.53/2.00	11.53/2.60	–	8.73/4.53	7.28/3.78	9.73/7.33
Berger (6,3)	6.72/2.00	8.83/2.00	10.78/2.50	–	8.44/4.11	7.24/3.52	9.72/7.00
Berger (7,3)	7.33/1.81	9.19/1.81	10.86/2.24	–	8.76/3.62	7.67/3.17	10.19/6.57
Berger (8,4)	8.38/1.67	10.42/1.67	12.96/2.04	14.42/2.42	11.08/4.71	9.85/4.19	11.38/6.67
Berger (9,4)	9.22/1.56	11.04/1.56	13.81/1.89	14.63/2.22	11.63/4.26	10.47/3.83	12.11/6.37

presented links. We are only interested in the dynamic cost that are directly impacted by the choice of a certain protocol and code. Figure 4.29 shows the results of this analysis.

The base bar of each bar stack corresponds to the combined costs of a transmitter/receiver pair. Hence, this bar includes the encoder, decoder, input and output register as well as one CD. Each additional section represents the costs for one intermediate pipeline stage, which includes the pipeline D latches (or C gates in the case of the RZ protocol because of the simple WCHB design) and one CD.

It can be seen that for all codes the hop costs for the NRZ protocol are the most expensive. However, with greater initial costs the cheaper CDs of the SDS and UBS protocols often only pay off after a certain amount of pipeline stages. The DS protocol performs quite well, as it only requires a little more hardware investment than the RZ protocol and still

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

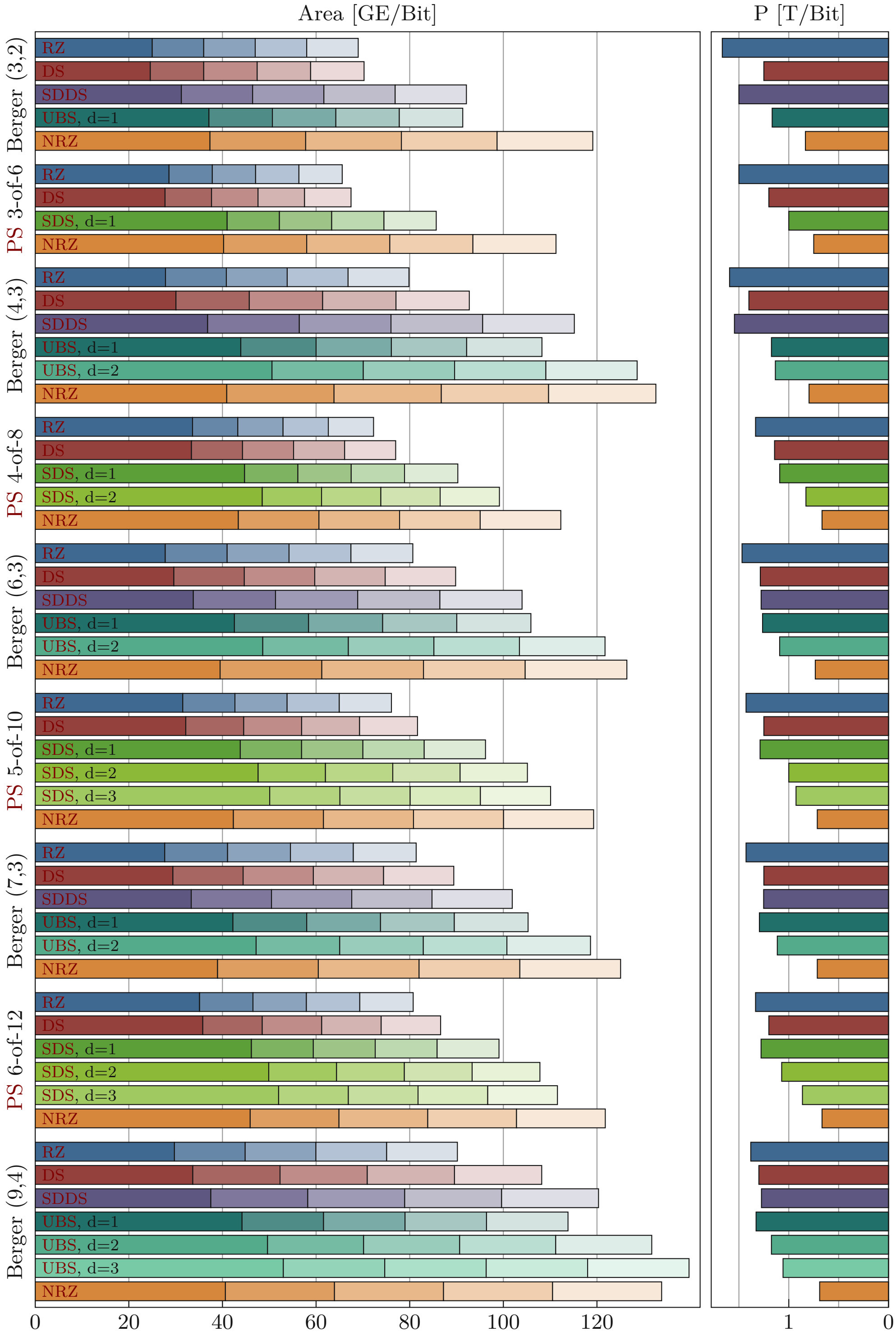


Figure 4.29: Hardware overhead for different link lengths, codes and protocols (left) and the associated power metric (right)

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

improves the power metric quite significantly (see bars on the right hand side), especially for codes with a small bit width. When the PSCWCs are compared to the Berger codes it can be seen that the higher initial costs for encoding and decoding pay off after just a few hops, regardless of the protocol.

4.6.2 Performance/Delay Analysis

This section discusses how the hybrid protocols impact the data transmission performance, i.e., the throughput, of a DI link. We start out by comparing the "classic" RZ and NRZ protocol. For this purpose we analyze the WCHB as well as the MDHB pipeline style (see Section 4.5.1) by creating a model for their dynamic behavior. After that we show how the hybrid protocols change the attainable performance when compared to the RZ protocol.

To quantify the pipeline performance, we use the *local cycle time* metric [BOF10]. The *local cycle time* corresponds to the minimal time required for a single pipeline stage to complete one handshake cycle with its neighbors. This, hence, gives a lower bound for the *system cycle time*, which is basically the inverse of a pipeline's throughput.

For this analysis we consider DI links as homogeneous linear pipelines, i.e., every pipeline stage is implemented identically and hence has similar delays. Because of the fact that handshaking protocols involve the communication of a pipeline stage with the next and the previous stage the *local cycle time* is usually a function of the delays of three neighboring blocks. This is reflected by the model circuits we use in this analysis shown in Figures 4.30 and 4.32. The environments shown in these figures are assumed to be ideal, i.e., they generate immediate responses to the inputs they are presented with. Hence, they are no limiting factor for the cycle time.

Let's first consider a classic 4-phase WCHB pipeline as shown in Figure 4.30. The delay Δ_{wire} models the wire delay[5] on the data bus D_i connecting two pipeline stages. Adding Δ_{wire} and Δ_C (i.e., the delay through the C gates comprising the buffer) thus yields the forward latency of a pipeline stage. The delay Δ_{ack} corresponds to the delay of the acknowledgment signal measured from the output of the CD to the C gates of the previous pipeline stage. To simplify the analysis we assume equal delays for rising and falling transitions.

To extract an analytical expression for the cycle time of this circuit, its dynamic behavior can be modeled by a PN as discussed in more detail in [BOF10]. For the WCHB pipeline this yields the graph shown in Figure 4.31. This type of graph can be interpreted in a similar way as an STG. However, here the nodes don't (always) correspond to transitions of single signal wires but model more abstract events, like the transition of the data bus from the spacer (i.e., null) phase to the data phase (D_i^{data}) or vice versa (D_i^{null}). This allows to capture the behavior of the pipeline in a compact way, independent of the actual data traversing it.

---

[5] In this chapter we focus on data *transport*, so we do not account for computations performed on the data and the associated delay.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

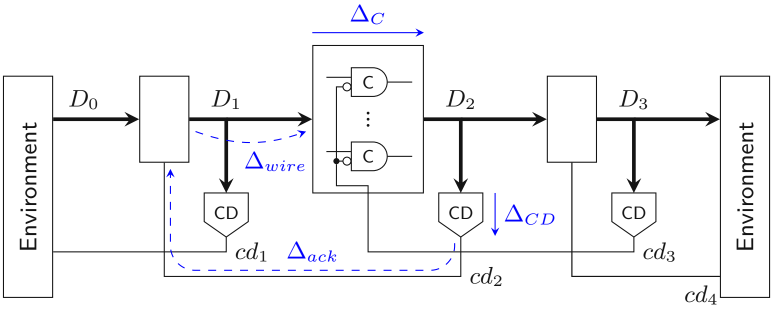


Figure 4.30: WCHB pipeline circuit model with delays (three stages)

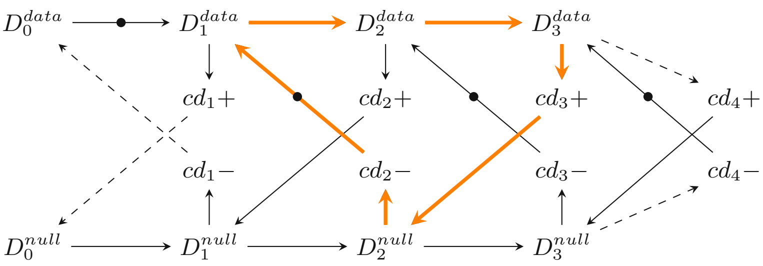


Figure 4.31: PN model for the WCHB pipeline (three stages)

Every node (event) of the graph is associated with a certain delay/latency: The nodes cd_i+ and cd_i- add the delay Δ_{CD}, and each node D_i^x adds Δ_C. However, note that some of the arcs also cause a delay (e.g., $cd_i+ \rightarrow D_{i-1}^{null}$, which adds Δ_{ack} or $D_i^{data} \rightarrow D_{i+1}^{data}$, which adds Δ_{wire}). These particular delays are marked with dashed lines in Figure 4.30.

The *local cycle time* is now obtained by analyzing the longest cycle in this graph, which is marked by the orange arrows in the figure. Equation (4.26) shows the resulting expression for the *local cycle time* of the WCHB pipeline, which corresponds to the time it takes for one code word and one spacer to pass though one pipeline stage.

$$T_{WCHB} = 4\Delta_C + 2\Delta_{CD} + 2\Delta_{wire} + 2\Delta_{ack} \tag{4.26}$$

For the MDHB pipeline we use the circuit model shown in Figure 4.32.

The associated graph model is shown in Figure 4.33. Since this pipeline works with both RZ and NRZ protocols we refer to the data events as $D_i^{\varphi 1}$ and $D_i^{\varphi 2}$.

Again the longest cycle is marked orange and the resulting cycle time expression is shown in Equation (4.27).

$$T_{MDHB} = 4\Delta_L + 2\Delta_{wire} + 2\Delta_{CD} + 2\Delta_{XNOR} + 2\Delta_{ack} \tag{4.27}$$

This expression yields the time it takes one pipeline stage to go though the two phases φ_1 and φ_2. In NRZ protocols both of these phases transmit actual data, while in RZ

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

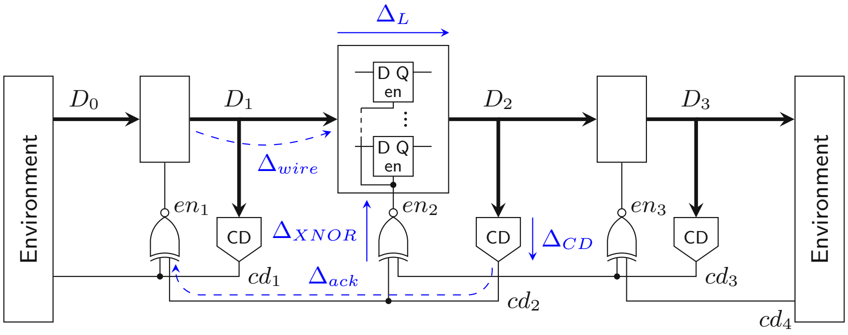


Figure 4.32: MDHB pipeline circuit with delays (three stages)

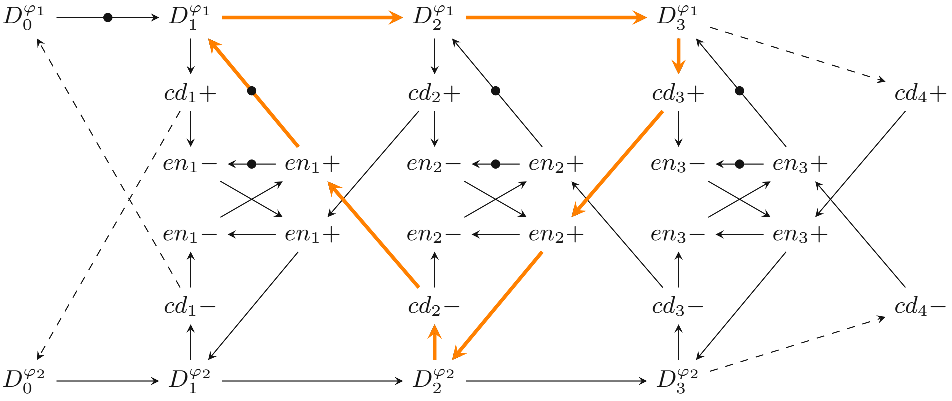


Figure 4.33: PN model for the MDHB pipeline (three stages)

protocols φ_2 corresponds to the spacer phase. Hence, to make the protocols comparable this fact must be taken into account. We do this by introducing a factor of $\frac{1}{2}$ for the actual cycle time of the NRZ protocol. Equations (4.28) and (4.29) show the resulting expressions.

$$T_{MDHB}^{RZ} = 4\Delta_L + 2\Delta_{wire} + 2\Delta_{CD}^{RZ} + 2\Delta_{XNOR} + 2\Delta_{ack} \tag{4.28}$$

$$\begin{aligned} T_{MDHB}^{NRZ} &= \frac{1}{2}(4\Delta_L + 2\Delta_{wire} + 2\Delta_{CD}^{NRZ} + 2\Delta_{XNOR} + 2\Delta_{ack}) \\ &= 2\Delta_L + \Delta_{wire} + \Delta_{CD}^{NRZ} + \Delta_{XNOR} + \Delta_{ack} \end{aligned} \tag{4.29}$$

When Equation (4.28) is compared to the cycle time of the WCHB pipeline (Equation (4.26)), it can be seen that the expressions are very similar. The only difference is the delay for the additional XNOR gate (assuming $\Delta_L \approx \Delta_C$). This reveals a first small downside of the hybrid protocols because they have to use the MDHB pipeline.

Notice that in Equations (4.28) and (4.29) Δ_{CD} has been replaced by variables denoting the actual delays of CDs for the specific protocol. Section 4.4 discussed how an NRZ CD

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

can be implemented using an RZ CD and an appropriate wrapper circuit consisting of shadow latches and XOR gates to detect input transitions. From the circuit in Figure 4.11 we can, thus, derive the following equation for the delay of NRZ CDs:

$$\Delta_{CD}^{NRZ} = \Delta_{TFF} + \Delta_L + 2(\Delta_{XOR} + \Delta_{CD}^{RZ}) \tag{4.30}$$

Plugging this into Equation (4.29) yields:

$$T_{MT}^{NRZ} = 3\Delta_L + \Delta_{wire} + \Delta_{TFF} + 2(\Delta_{XOR} + \Delta_{CD}^{RZ}) + \Delta_{XNOR} + \Delta_{ack} \tag{4.31}$$

When this expression is now compared to Equation (4.28) (or Equation (4.26)), it can be seen that the main difference is that the terms Δ_{wire} and Δ_{ack} appear without the factor 2. Depending on how large these values are (compared to the sum of the other delays of the expression) this can of course have a large impact on the overall performance gains that can be achieved using the NRZ protocol.

For a very detailed picture of the NRZ protocol one might also investigate the impact of the protocol on the delay Δ_{wire}. Even if the signal wires between two pipeline stages have the same geometrical dimensions and the same driver strength is used, it makes a difference whether an RZ or NRZ protocol is used. If neighboring wires of a bus switch in opposite directions capacitive crosstalk effects [PD08] can have a negative impact on the delay. For the RZ and hybrid protocols such a situation can never occur since in one protocol phase *all* transitioning wires must switch to the same value.

To calculate the cycle time of the hybrid protocols, we can basically take Equation (4.27) and plug in the correct value for Δ_{CD}. Hence, in the following we will examine which factors contribute to the CD delay and how to estimate it. We start off with the analysis of the CDs for constant-weight codes and then briefly discuss Berger CDs as well.

From the general structure of the RZ CDs (see Figure 4.17) we can deduce that the delay $\Delta_{CD}^{cw|RZ}$ can be divided into the delay Δ_{C_m} of the HG (i.e., the m-input C gate at the output) and the delay of the purely combinational CN Δ_{CN}. The latter delay is bounded by the depth of the CN, denoted by D_{CN} (i.e., the maximum number of comparator cells an input signal has to pass through in order to reach the HG), multiplied by the delay of a single comparator cell Δ_{CC}, which amounts to roughly one gate delay.

$$\Delta_{CD}^{cw|RZ} = D_{CN} * \Delta_{CC} + \Delta_{C_m} \tag{4.32}$$

Table 4.10 lists the CN depths for the PSCWCs investigated in this chapter. However, note that for asymmetrical CDs (like the one for the 3-of-6 code) the actual value of Δ_{CN} is data-dependent. Hence, the actual selection of the code word set also plays a role. This is because for certain input vectors there are paths through the CN that are shorter than its (worst-case) depth. For the partially-systematic 3-of-6 code an exhaustive analysis of every critical path for every code word reveals that the average number of comparator cells an input vector has to pass through is actually only 3.5 comparators instead of 4. However, for simplicity's sake we only consider the worst-case path in our analysis.

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

For CDs for the SDS protocol the data dependency is an even bigger issue, because depending on whether the all-zero or the short-distance spacer is used two different paths through the CD are relevant. Equation (4.33) shows how the average CD delay can be calculated. Recall that the variable p denotes the percentage of cases in which the short-distance spacer is used, which can either be estimated using Equation (4.18) or be calculated exactly by considering the actual code word set. For the cases where the input of the CD transitions from the all-zero spacer to a code word (or vice versa) the normal depth D_{CN} must be used. When the input of the CD switches from a code word to the short-distance spacer or vice versa, the second-level CD must be considered, which increases the depth of the CN to D_{CN_2}. However, in this case only the delay of the d-input C gate in the HG is relevant, since the m-input C gate (with the delay Δ_{C_m}) is already set. Finally the delay Δ_{AND} of the output AND gate of the HG must be added, to arrive at the following equation:

$$\Delta_{CD}^{cw|SDS} = (1-p) * (D_{CN} * \Delta_{CC} + \Delta_{C_m}) + p * (D_{CN_2} * \Delta_{CC} + \Delta_{C_d}) + \Delta_{AND} \quad (4.33)$$

Table 4.10 shows the parameters for p and D_{CN_2} extracted from our CD circuits. Note that, for the case where $d = 1$, there is no second C gate in the HG (hence $\Delta_{C_1} = 0$). Furthermore, the second-level CD only consists of an m-input OR gate for which we estimated 1 (for $m = 3$) and 2 (for $3 < m < 10$) comparator delays, respectively.

Table 4.10: Delay estimation parameters for m-of-n CDs (RZ and SDS protocols)

Code	D_{CN}	D_{CN_2}/p (d) 1	2	3
PS 3-of-6	4	5/0.50	–	–
PS 4-of-8	4	6/0.24	6/0.76	–
PS 5-of-10	6	8/0.12	9/0.5	9/0.88
PS 6-of-12	6	8/0.05	10/0.29	10/0.71

Generally it can be concluded that $\Delta_{CD}^{m\text{-of-}2m|SDS}|_{d=1}$ will only be marginally larger than $\Delta_{CD}^{m\text{-of-}2m|RZ}$, since the delay of an m-input OR gate (for the second-level 1-of-m CD) will certainly not exceed the delay of an m-input C gate. If the delay of the OR gate is significantly lower it can even compensate for Δ_{AND}. For higher values of d it strongly depends on whether the smaller C gate in the short-distance spacer path is sufficiently faster than the m-input C gate in the regular path to make up for the increased CN depth D_{CN_2}.

Because of a similar reason $\Delta_{CD}^{m\text{-of-}2m|DS}$ is only marginally larger than $\Delta_{m\text{-of-}2m|CD}^{RZ}$. Both possible paths to the output AND gate contain the same circuit elements, i.e., a CN with the same depth and an m-input C gate. Hence, the only difference in terms of delay is the output AND gate itself.

$$\Delta_{CD}^{m\text{-of-}2m|DS} = \Delta_{CD}^{m\text{-of-}2m|RZ} + \Delta_{AND} \quad (4.34)$$

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

The CDs for protocols using Berger codes are by their nature very asymmetric, which again hints on some data-dependent delay behavior. However, in most cases the overall depth of their CNs is dominated by the depth of the SN T^b used to determine the Hamming weight of the data part of the code words. Equation (4.35) shows the CD delay for the RZ protocol. Table 4.11 lists the CN depths for the Berger codes with $3 \leq b \leq 9$ data bits.

$$\Delta_{CD}^{B|RZ} = D_{CN} * \Delta_{CC} + \Delta_{C_b} \tag{4.35}$$

Similar to $\Delta_{CD}^{cw|SDS}$, $\Delta_{CD}^{B|UBS}$ can be defined as:

$$\Delta_{CD}^{B|UBS} = (1-p) * (D_{CN} * \Delta_{CC} + \Delta_{C_b}) + p * (D_{CN_2} * \Delta_{CC} + \Delta_{C_d}) + \Delta_{AND} \tag{4.36}$$

The variable p again denotes the percentage of cases where the unbalanced spacer can be used and the second-level CD is activated. The parameters D_{CN_2} and p are listed in Table 4.11. Again an argument can be made that for $d = 1$ the delay of the CD is only marginally increased compared to $\Delta_{CD}^{B|RZ}$.

Table 4.11: Delay estimation parameters for Berger CDs (RZ and UBS protocols)

Code	D_{CN}	D_{CN_2}/p (d) 1	2	3
Berger (3,2)	4	5/0.50	–	–
Berger (4,3)	4	6/0.38	8/0.65	–
Berger (5,3)	6	8/0.27	10/0.53	–
Berger (6,3)	6	8/0.18	10/0.44	–
Berger (7,3)	7	9/0.12	11/0.36	–
Berger (8,4)	7	9/0.07	12/0.29	13/0.48
Berger (9,4)	8	10/0.05	13/0.22	14/0.45

Recall that for the CD for the DS (and SDDS) protocol, the same CN as for the RZ CD is used. The only difference is that the $2^k - 1$ outputs that would be pruned from the network in case of an RZ CD, are merged using a C gate with $2^k - 1$ inputs. Depending on the spacer either this C gate or the usual b-input C gate of the base circuit contributes to the critical path. Assuming equally distributed spacer-types (all-zero and all-one) we arrive at the following equation.

$$\Delta_{CD}^{B|DS} = D_{CN} * \Delta_{CC} + \frac{\Delta_{C_b} + \Delta_{C_{2^k-1}}}{2} + \Delta_{AND} \tag{4.37}$$

Notice that in the case where $b = 2^k - 1$ (i.e., in the case where Berger codes offer the best coding efficiency), both C gates have the same number of inputs. In this case the only difference to $\Delta_{CD}^{B|RZ}$ is the delay of the output AND gate. In all other cases we have that $\Delta_{C_b} < \Delta_{C_{2^k-1}}$, which (depending on b) can significantly worsen the delay of the CD.

Overall we can conclude from our analysis that the more (power) efficient encodings and protocols do incur a performance penalty. We have, however, also seen that with a careful selection of the protocol parameters this penalty can be made negligible.

4.7 Conclusion

In this chapter we have tried to supply the designer of a DI communication channel with a systematic approach for finding the most efficient solution for a given purpose. To this end we have made contributions along several lines:

Observing that traditional DI codes are either very efficient with respect to completion detection (like the constant-weight codes) or with respect to decoding (like systematic codes), but not both at the same time, we have tried to approach a global optimization by careful composition of the DI code as a constant-weight code that includes a number of systematic bits. More specifically, we have elaborated a method for systematically deciding upon the number of systematic bits plus the generation of the non-systematic bits required to make the code constant weight. The degrees of freedom we use for optimization are the mapping between data words and code words, as well as the selection of unused code words present in our incomplete coding approach. We have presented guidelines for codes up to the 6-of-12 code, which covers the practically relevant range.

We have proposed the use of multiple spacers in the 4-phase protocol, either to obtain a higher energy efficiency (by saving transitions when going to the spacer and onward to the next data phase), or to encode additional information through the specific choice of the spacer. The latter can be viewed as a blend of the 4-phase protocol with its relatively low implementation overhead and the 2-phase protocol with its high coding and energy efficiency.

For the completion detection we have presented construction guidelines based on comparator networks. Our solution not only surpasses related approaches in terms of area efficiency, it also avoids pitfalls with orphan transitions sometimes found. Apart from CDs for constant-weight codes, which are immediately useful for the presented partially systematic codes, we also elaborate optimized solutions for Berger codes. Furthermore, our completion detection approach also works for all of the newly proposed protocols.

Building on all these contributions, we have explored the code space relevant for typical DI communication channels and have identified the respective efforts for the diverse options and devised highly optimized solutions with respect to code construction and implementation of encoders, decoders and CDs. Our comprehensive analysis results allow the designer of a DI channel to quickly check the available options for a given problem and immediately compare the efforts implied by different alternatives, as well as the attainable data throughput.

Error detection and error correction have not been covered here. If these properties are an issue, the concepts presented in [LSH15] and [HLS16] can be consulted additionally. In this

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

context it should also be mentioned that the extra bit encoded by the DS protocol is very robust, which might be advantageous for transmitting specifically sensitive information.

Considering that DI channels are very convenient for inter- and intra-chip communication between function blocks, our hope is, that this work can thus provide the designer a useful reference for selecting the appropriate coding scheme along with implementation for encoder, decoder and CD, to ultimately come up with an efficient overall solution.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 5

Asynchronous Circuit Description

This chapter introduces `pypr`, the Python-based asynchronous circuit description framework, developed throughout the course of this thesis, to conveniently create, analyze, modify and simulate asynchronous (and specifically dual-rail QDI) circuits. The main application area of this framework is to generate QDI circuits for the fault-injection experiments carried out in Chapter 6. For that purpose, it is paramount to be able to easily parameterize the circuit generation to, e.g., iterate through different circuit implementation styles, (input) bit widths or delay configurations. For the design entry we target the data-flow (see Section 2.4) and gate level, since we want to have very specific control over the circuit structure and guarantee predictable results for procedurally generated circuit parts. Hence, we opted for a low-level production-rule-inspired circuit representation.

A *production rule*, as defined in [Mar89, Mar91], is a construct of the form $G \mapsto S$. G is a Boolean expression referred to as the *guard* of the production rule, while S is a set of simple signal assignments. A signal assignment is given by a variable and an arrow (↑ or ↓) that indicates whether the respective signal switches to high or low. A production rule can "fire" if its guard evaluates to true, which means that all of its signal assignments are executed. Consider the following example.

$$a \wedge b \mapsto x\uparrow, y\downarrow$$

If a and b are both true the signal x is asserted while y is deasserted.

Two production rules that make statements about opposite transitions of a signal are called *complementary*. Together they implement an *operator*:

$$G_1 \mapsto x\uparrow$$
$$G_2 \mapsto x\downarrow$$

Complementary production rules must be *non-interfering*, which means their guards (i.e., G_1 and G_2 in the example above) must never evaluate to true at the same time (i.e.,

$\neg G_1 \lor \neg G_2$ must always hold). If it is always the case that either G_1 or G_2 holds the operator they implement is *combinational*. If $\neg G_1 \land \neg G_2$ ever holds, the associated signal retains the value of the last assignment, which makes the operator *state-holding*.

Guards of production rules must be *stable*, which means that if a guard becomes true (because of some other signal change) it must remain true until the assignments are executed. A Production Rule Set (PRS) is defined to be the "concurrent composition" of multiple production rules. Under stability and non-interference a PRS is executed by continuously selecting a rule (the selection must be fair, i.e., each rule must be selected infinitely often) and executing its signal assignments if its guard evaluates to true.

The mapping of a PRS to a CMOS circuit is straightforward, if the guards of its production rules can be directly mapped to suitable p- and n-stacks. If this is not the case the respective rule must be implemented using multiple CMOS gates. State-holding operators can for example be implemented using signal keepers, constructed from weak-feedback inverter loops.

The way we use this formalism in our circuit description framework is as follows. We define a simple (production-rule-based) hardware description language and implement an accompanying Python package that operates on said language. This package must, therefore, offer data structures able to represent circuits specified in this language and provide the means to generate and modify them. The use of a dedicated language also allows to conveniently load and store circuits, which in turn enables different tools to process and analyze them. Moreover, it enables the package user to dump quite a complex data structure into a more easily digestible human readable format, which is vital during circuit development and debugging.

Besides the central requirement of being simple to parse, understand and process we also want this PRS language to have support for the following features:

- embedded delay information
- hierarchical designs (decomposition)
- datatype-level dual-rail signals and vectors
- extensibility through attachable general-purpose (user) information for various language constructs

Any form of parameterization or support for generic programming constructs is not needed since code (especially for larger designs) will mainly be generated by Python scripts anyway. Since the user can leverage the full power of the Python programming language to make generic and parameterizable circuit designs, adding such features would unnecessarily complicate the language and thus interfere with the main goal of simplicity.

This is also partly the reason for why we decided to use a dedicated language and did not build on some standard hardware description language like VHDL or Verilog. In

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

addition, it would have been complicated to realize and integrate some of the desired special purpose features. There is also no real benefit in using an established language, since it is trivial to export the dedicated PRS language to other formats. This also has the added benefit that the way in which production rules or gates are modeled and represented in the target language is fully customizable.

The structure of this chapter is as follows. First, Section 5.1 gives a formal language definition of our dedicated PRS language. Then, Section 5.2 continues with a brief overview of the actual Python package, its features, core data structure and algorithms. Section 5.3 details how logic synthesis for QDI dual-rail circuit is handled within our framework. Finally, Section 5.4 discusses the formal verification features of the package.

5.1 Language Specification

This section formally describes the PRS language used internally by the `pypr` package. To define the language we present its context-free grammar (CFG) and describe the semantics of the various language constructs [Wat17].

For the sake of brevity we first define some essential non-terminal symbols used throughout the grammar in form of POSIX Extended Regular Expressions.

<table>
<tr><th>Non-terminal</th><th>Regular expression</th></tr>
<tr><td>⟨<i>ident</i>⟩</td><td><code>[a-zA-Z][a-zA-Z_0-9]*</code></td></tr>
<tr><td>⟨<i>int</i>⟩</td><td><code>[-+]?(0x|0b)?\d+</code></td></tr>
<tr><td>⟨<i>float</i>⟩</td><td><code>[-+]?((\d*\.\d+|\d+\.)([Ee][-+]?\d+)?|\d+[Ee][+-]?\d+)</code></td></tr>
<tr><td>⟨<i>string</i>⟩</td><td><code>"\.*"</code></td></tr>
</table>

Identifiers (⟨*ident*⟩) are given by an arbitrary sequence of letters, numbers and underscores. The first character must always be a letter and reserved keywords are obviously not allowed. Integers (⟨*int*⟩) may be specified as binary, decimal or hexadecimal values. The base is determined by the prefix to the number (i.e., `0b` for binary and `0x` for hexadecimal numbers). Float values (⟨*float*⟩) can either be numbers containing a decimal point (e.g., `1.2`, `.2`, `1.`) followed by an optional exponent (e.g., `1.2e-1`, `.2e2`) or a number without a decimal point, which must be followed be an exponent (e.g., `1e-1`, `1e2`). This approach allows a clear and unambiguous distinction between integers and floats. A string (⟨*string*⟩) is simply an arbitrary sequence of characters delimited by quotation marks.

Before we can specify the actual grammar rules[1], we first need to introduce the notation used throughout the section. A grammar rule consists of a right- and left-hand part, where the left-hand side contains a single non-terminal symbol. We use an extended

---

[1] The rules of a (context-free) grammar are also often referred to as *production rules*. However, to avoid confusion with the way this term is used in the context of this work we avoid this term and simply use *grammar rules* instead.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Backus–Naur form (EBNF) for (the right-hand side of) the grammar rules, since this allows a more compact and natural way of expressing the syntax [Wat17]. Hence, the right-hand side may contain terminal and non-terminal symbols in arbitrary sequence as well as the operators “|”, “[...]” and “{...}”. To better illustrate the meaning of the notation and the used operators, consider the following example grammar.

⟨*start*⟩ ::= ⟨*nt*⟩ {‘`c`’} [‘`d`’]

⟨*nt*⟩ ::= ‘`a`’ ⟨*nt*⟩ | ‘`b`’ ⟨*nt*⟩ | ⟨*empty*⟩

The start symbol of the grammar is ⟨*start*⟩. Additionally there is one other non-terminal symbol ⟨*nt*⟩. Non-terminal symbols are marked with diamond brackets. The grammar contains four terminal symbols ‘a’, ‘b’, ‘c’ and ‘d’, which are marked using single quotation marks. The special symbol ⟨*empty*⟩ corresponds to the empty word (often also written as ε) and is used to, e.g., break rule recursions.

The brace operator means that the inner expression may be repeated (an arbitrary number of times) but may also appear not at all. The bracket operator marks optional expressions[2]. Finally, the vertical bar operator denotes alternatives.

Hence, the two-rule example grammar shown above generates the language specified by the regular expression `[ab]*c*d?`. However, note that CFGs and regular expressions are not equally powerful in the languages that they can generate (Type-2 vs. Type-3 in the Chomsky hierarchy of languages). This can easily be demonstrated with another simple example.

⟨*start*⟩ ::= ‘`a`’ ⟨*start*⟩ ‘`b`’ | ⟨*empty*⟩

This grammar generates the language consisting of all words that start with an arbitrary number of ‘a’ symbols followed by the exact same number of ‘b’ symbols. It is not possible to specify this language as a regular expression.

With the notation out of the way we can start to define the PRS language. Let’s start by defining two important non-terminal symbols, for numerical and Boolean constants.

⟨*number*⟩ ::= ⟨*int*⟩ | ⟨*float*⟩

⟨*bool*⟩ ::= ‘`true`’ | ‘`false`’

Input files are processed one PRS at a time. Thus, the start symbol of the grammar is the (non-terminal) ⟨*prs*⟩.

---

[2] Note that the bracket and brace operators have a completely different semantic in regular expressions.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

⟨*prs*⟩ ::= ‘`prs`’ ⟨*ident*⟩ ‘`is`’ [⟨*attributes*⟩ ‘`;`’] ⟨*inputs*⟩ ⟨*outputs*⟩ [⟨*locals*⟩] [⟨*instances*⟩]
‘`begin`’ { ⟨*rule*⟩ } [⟨*constraints*⟩] ‘`end`’ ‘`prs`’ ‘`;`’

The start of a PRS is marked with the keyword `'prs'` followed by an identifier specifying the name of the PRS and the keyword `'is'`. After that an optional set of attributes can be specified (see Section 5.1.5). Then, the declarations for the inputs, outputs and (optional) locals signals are listed, which are in turn followed by an optional list of instances of other (sub-) PRSs. The beginning of the actual rule section of the PRS is marked with the `'begin'` keyword. Following the rule section an optional set of constraints may be listed (see Section 5.1.4). The end of the PRS is indicated using the keywords `'end prs;'`

The language supports line comments which use the `#` symbol. Everything after this symbol, until the next newline, is ignored.

5.1.1 Signal Declarations and Datatypes

Input and output signals must always be declared, as these signals define the interface to the PRS. If a PRS is instantiated by another one, the inputs and output of the inner PRS (i.e., the instance) can be driven (inputs) or read (outputs) by the outer one. Local signals also have to be declared, unless their data type is `Bit`, in which case the declaration may be omitted. The keywords `'inputs'`, `'outputs'` and `'locals'` are used to mark the beginning of the respective declaration sections.

⟨*inputs*⟩ ::= ‘`inputs`’ { ⟨*signal-decl*⟩ }

⟨*outputs*⟩ ::= ‘`outputs`’ { ⟨*signal-decl*⟩ }

⟨*locals*⟩ ::= ‘`locals`’ { ⟨*signal-decl*⟩ }

⟨*signal-decl*⟩ ::= ⟨*ident*⟩ ‘`:`’ ⟨*ident*⟩ [‘`(`’ ⟨*integer*⟩ ‘`)`’] [⟨*attributes*⟩] ‘`;`’

A signal declaration is given by an identifier (i.e., the signal name) followed by a colon and another identifier, which specifies the datatype. The most basic data type is `Bit`. The language currently also support the datatype `DRBit` to represent dual-rail signals. This datatype is a so called multi-rail type and encompasses two single-bit signals (i.e., the true and the false rail). Further datatypes could be added in the future. However, datatypes are always built-in types of the `pypr` package, i.e., they are directly defined in the Python package since there is no possibility to declare new datatypes within the language itself.

By adding an optional integer in parentheses after the datatype identifier, it is also possible to declare vectors (the integer specifying the vector width). Hence, in the context of the `pypr` type system the datatypes `Bit` and `DRBit` are considered scalar types while, e.g., `Bit(2)` would be considered a vector.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Finally, an optional set of attributes can be added to the signal declaration, which will be discussed in more detail in Section 5.1.5.

Consider the following example PRS, named `demo`, which specifies four single-bit inputs `a`-`d` and a single 2-element dual-rail output vector `x`. In the rules section of the PRS each of those four inputs is assigned to one of the four rails that make up the output `x` using so called wire rules. For one of the dual-rail bits a (temporary) local signal `temp` is used. The exact rule syntax and semantic will be explained in more detail in Section 5.1.3. For now it is sufficient to know that the `'wire'` command simply connects two signals.

```
 1 prs demo is
 2 inputs a : Bit; b : Bit; c : Bit; d : Bit;
 3 outputs x : DRBit(2);
 4 locals temp : DRBit;
 5 begin
 6   x(0).T := wire(a);
 7   x(0).F := wire(b);
 8   temp.T := wire(c);
 9   temp.F := wire(d);
10   x(1) := wire(temp);
11 end prs;
```

This example also shows the syntax for accessing certain bits or elements of multi-rail and vector signals. Vector elements are accessed using parentheses enclosing the desired index. The elements of an n-element vector have the indices 0 to $n-1$. Individual rails of a multi-rail signal can be accessed via the point operator. The dual-rail datatype `DRBit` has two rails named T (true) and F (false). The formal grammar rule for that is shown below.

⟨*signal*⟩ ::= ⟨*ident*⟩ ['`->`' ⟨*ident*⟩] ['`(`' ⟨*integer*⟩ '`)`'] ['`.`' ⟨*ident*⟩]

This grammar rule also shows how an interface signal of an instance can be accessed. For that purpose, the arrow operator `'->'` is used. The identifier to the left of the arrow operator refers to an instance name while the right identifier refers to some input or output of this instance. These arrow expressions are referred to as *instance connectors*. For more details see Section 5.1.2.

Note that every signal expression has a distinct datatype. Wire rules, as used in the example above, can only connect signals of the same datatype. The datatype of the signal expression `x(0)` is `DRBit`, since it is an element of a `DRBit` vector. Hence, it can be connected to the `temp` signal using a wire rule (Line 10). The individual rails of a multi-rails signal, like `x(0).T` always have the datatype `Bit`, which makes it possible to connect them to the `Bit`-type inputs (Lines 6 to 9).

5.1.2 Instances

After the signal declarations follows an optional list of instances, where other PRSs can be instantiated and connected. The beginning of the instance section is marked with the keyword `'instances'`.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

⟨*instances*⟩ ::= ‘`instances`’ {⟨*instance*⟩}

⟨*instance*⟩ ::= ⟨*ident*⟩ := ⟨*ident*⟩ ‘`(`’ [⟨*inline-connector-list*⟩]‘`)`’ [⟨*attributes*⟩] ‘`;`’

An instance is declared by specifying an identifier (i.e., the instance name) followed by the assignment operator `':='` and the name of the PRS that should be instantiated. After that a set of parentheses with an optional list of *inline connectors*, which will be explained shortly, is expected. Like signal declarations, instances may also be augmented with a set of attributes (see Section 5.1.5). Consider the two example PRSs `simple_and_gate` and `top` shown below.

```
prs simple_and_gate is
inputs x : Bit; y : Bit;
outputs z : Bit;
begin z := rule(a and b);
end prs;

prs top is
inputs a : Bit; b : Bit;
outputs c : Bit;
instances and_inst := simple_and_gate();
begin
  and_inst->x := wire(a);
  and_inst->y := wire(b);
  c := wire(and_inst->z);
end prs;
```

The PRS `simple_and_gate` simply implements an AND gate with two inputs `x` and `y` and a single output `z`. The top-level PRS `top` instantiates this PRS and connects its own interface signals to it using three wire rules. Notice how instance connectors are used to access the I/O signals of the instance `and_inst`. Instance connectors can be viewed as implicitly declared local signals.

Another way of connecting instances are *inline connectors*, which can be specified directly when the instance is declared.

⟨*inline-connector-list*⟩ ::= ⟨*inline-connector*⟩ {‘`,`’ ⟨*inline-connector*⟩}

⟨*inline-connector*⟩ ::= ⟨*signal*⟩ ‘`:=`’ ⟨*signal*⟩

This language feature is more or less a short-hand notation for wire rules and does not add expressive power to the language. An inline connector is given by a signal expression followed by the assignment operator `':='` and another signal expression. The left signal expression always refers to an interface signal of the instantiated PRS (no matter whether the signal is an input or an output). Hence, here instance connectors are not allowed (although the grammar allows that for the sake of simplicity). The right-hand side can be an arbitrary signal expression. Of course the datatypes of both sides must match for an inline connector to be valid.

The example below shows how the PRS `top` can also be implemented using inline connectors.

```
prs top is
inputs a : Bit; b : Bit;
outputs c : Bit;
instances inst := P(x := a, y := b, z := c);
begin
end prs;
```

Of course, if an inline connector drives an input signal then there must not be other rules driving the corresponding instance connector, as this would create a driver conflict.

5.1.3 Rules

The rules section of the PRS may contain an arbitrary number of rules but may also be left completely empty. A rule always start with a signal expression followed by the assignment operator `':='`. This signal expression must be a signal that can be driven, i.e., not an input or instance connector connecting some output signal of an instance. Moreover, there must always only be one rule (or inline-connector) that writes to a particular signal. The right-hand side of the assignment operator can be divided into four distinct parts. The first one, which is also the only non-optional part, is the actual rule expression (⟨*rule-expr*⟩). After that we have the optional initialization (⟨*init-expr*⟩) and delay expressions (⟨*delay-expr*⟩), followed by a set of supplemental attributes.

⟨*rule*⟩ ::= ⟨*signal*⟩ ‘`:=`’ ⟨*rule-expr*⟩ [⟨*init-expr*⟩] [⟨*delay-expr*⟩] [⟨*attributes*⟩] ‘`;`’

From this specification it is clear that the language in its current form only supports rules that define the behavior of exactly one signal, in contrast to what the production rule formalism introduced in the beginning of the chapter is capable of. For the scope of the investigations of this thesis this is sufficient. An extension of the PRS language to support a more general form of production rules would be trivial, but would also require suitable modifications to the algorithms of the accompanying Python package.

5.1.3.1 Rule Types

The PRS language supports three different types of rules, which are referred to as wire, standard and function rules.

⟨*rule-expr*⟩ ::= ⟨*wire-rule*⟩ | ⟨*standard-rule*⟩ | ⟨*function-rule*⟩

⟨*wire-rule*⟩ ::= ‘`wire`’ ‘`(`’ ⟨*signal*⟩ ‘`)`’

⟨*standard-rule*⟩ ::= ‘`rule`’ ‘`(`’ ⟨*expr*⟩ [‘`,`’ ⟨*expr*⟩] ‘`)`’

⟨*function-rule*⟩ ::= ⟨*ident*⟩ ‘`(`’ ⟨*expr*⟩ {‘`,`’ ⟨*expr*⟩} ‘`)`’

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Let's start with the wire rule, which has already been used in some of the examples shown above. As the name suggest a wire rule simply connects two signals. What makes it special, is the ability to connect signals that consist of multiple physical bits, i.e., it can connect signal expressions with arbitrary datatypes as long as the datatypes on the left-hand and the right-hand side of the assignment operator match. This is not possible with the two other rule types, where the signal expression on the left-hand side must always have the datatype `Bit`. However, wire rules cannot implement any logic, they just model physical connections between signals. Consider the following two PRSs `A` and `B`, which both describe the exact same circuit.

```
prs A is
inputs a : DRBit(2)
outputs x : DRBit(2);
begin x := wire(a);
end prs;

prs B is
inputs a : DRBit(2)
outputs x : DRBit(2);
begin
  x(0).T := wire(a(0).T);
  x(0).F := wire(a(0).F);
  x(1).T := wire(a(1).T);
  x(1).F := wire(a(1).F);
end prs;
```

The standard rule is declared with the `'rule'` keyword and two Boolean expressions (⟨*expr*⟩) in parentheses separated by a comma. The first one specifies the condition that must be satisfied for the target signal to switch to high (i.e., the up condition), while the second condition is responsible for switching the signal to low again (i.e., the down condition). If the second condition is omitted the rule is considered to be purely combinational, i.e., the down condition is derived automatically by inverting the up condition. Rules that specify up and down conditions that can be satisfied simultaneously are invalid, as this would basically correspond to a short-circuit. If there is a signal assignment where both the up and down condition are not satisfied the rule is considered to be *state-holding*. Note that it is also possible to model wires using standard rules, if the Boolean expression just consists of a single signal expression. However, as already mentioned, using this approach only single-bit signals can be connected.

To make to notation more compact and readable the language also supports so called function rules. Instead of the `'rule'` keyword a function rule uses an identifier followed by a list of arguments (i.e., Boolean expressions) separated by commas. Thereby the identifier refers to some specific function defined in the Python package. Additional function rules can be added by providing the necessary information and semantics in the form of a Python class which must adhere to certain interface specifications. These classes then also define how many arguments a respective function rule supports and what type of Boolean expressions are permissible (most of the function rules only allow plain signal expressions, i.e., no Boolean operators). Furthermore, they must provide a way to convert the function rule into a standard rule. Most of the standard gates (AND,

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

OR, XOR, etc.), including (asymmetric) C gates as well as threshold gates are already included in the `pypr` framework.

The example below shows three (equivalent) ways of defining a 2-input OR gate.

```
prs three_or_gates is
inputs a : Bit; b : Bit;
outputs x : Bit(3);
begin
  x(0) := rule(a or b, not a and not b);
  x(1) := rule(a or b);
  x(2) := or_gate(a, b);
end prs;
```

5.1.3.2 Rule Initialization

Rules can be equipped with an optional initialization expression, which basically defines the initial value for the target variable and offers a possibility to reset the rule and thereby a whole circuit to a defined state. The initialization expression is marked with the `'init'` keyword followed by an initialization value (Boolean constant or 0/1) and an optional initialization condition in parenthesis. The initialization condition may be a signal expression (high-active reset), an inverted signal expression (low-active reset) or a Boolean constant. A constant initialization condition with the value false is semantically equivalent to not having an initialization condition at all, while the constant true means that the whole rule can be replaced with a wire rule, which simply writes the initialization value to the target signal (those conditions might turn up during circuit optimization).

⟨*init-expr*⟩ ::= '`init`' '`(`' ⟨*init-value*⟩ ['`,`' ⟨*init-cond*⟩] '`)`'

⟨*init-value*⟩ ::= ⟨*integer*⟩ | ⟨*bool*⟩

⟨*init-cond*⟩ ::= ⟨*signal*⟩ | '`not`' ⟨*signal*⟩ | ⟨*bool*⟩

The following example shows a PRS modeling a resetable 2-input C gate.

```
prs resetable_cgate is
inputs a : Bit; b : Bit; reset : Bit;
outputs x : Bit;
begin x := cgate(a, b) init(0, reset);
end prs;
```

Initialization expressions are not supported/allowed for wire rules or combinational rules, i.e., only state-holding rules can be equipped with them. It is of course also possible to encode the reset directly into the Boolean equations for the up and down condition of a standard rule. However, the rationale behind the decision to allow to keep the initialization condition separate, is to not clutter the rule expressions with boilerplate reset code and to clearly mark and distinguish the initialization code from the actual logic.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

5.1.3.3 Rule Delays

The delay expression allows to associate a delay value with a rule. A delay expression starts with the (optional) selection of a delay model using the `'inertial'` or `'transport'` keywords (delay model specifier). After that the `'delay'` keyword is used followed by one or two actual time values enclosed in parentheses and separated by a comma. A time value is simply a number (integer or float) followed by a time unit. If two time values are specified the delay is asymmetric. This means that the first value measures the time it takes the rule to transition to high given that the up condition has been satisfied while the second value specifies the delay of falling transitions given that the down condition has been satisfied. These time values are referred to as up and down delays. If only one time value is specified, the delay is symmetric, meaning that the up and down delays are equal.

If no delay model is selected, the actual delay model depends on the code generator and/or the target language. When a VHDL model is generated out of a PRS with rules without an explicit delay model specifier, the delays in the VHDL code also won't explicitly specify the model, which means the default VHDL delay model (i.e., inertial) is used.

⟨*delay-expr*⟩ ::= [⟨*delay-model*⟩] '`delay`' '`(`' ⟨*time-value*⟩ ['`,`' ⟨*time-value*⟩] '`)`'

⟨*delay-model*⟩ ::= '`inertial`' | '`transport`'

⟨*time-value*⟩ ::= ⟨*number*⟩ ⟨*time-unit*⟩

⟨*time-unit*⟩ ::= '`ps`' | '`ns`' | '`us`' | '`ms`' | '`s`'

The PRS in the listing below shows some examples of delay expressions.

```
prs demo is
inputs a : Bit; b : Bit;
outputs x : Bit(3);
begin
  x(0) := or_gate(a, b) delay(1 ns);
  x(1) := or_gate(a, b) transport delay(1 ns);
  x(2) := or_gate(a, b) inertial delay(1 ns, 2 ns);
end prs;
```

5.1.3.4 Boolean Expressions

Boolean expressions can be any combination of the binary operators `'and'`, `'or'` and `'xor'` and the unary operator `'not'` as well as Boolean constants and signal expressions. Note that the `'xor'` operation has the highest and the `'or'` operator the lowest precedence. Parentheses can be used to change the precedence of operations and/or to make them more explicit. The expression `a or b and c xor d` is interpreted as `a or (b and (c xor d))`. Of course, for a Boolean expression to be valid it must only use signal expressions with the `Bit` datatype.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

⟨*expr*⟩ ::= ⟨*unary-expr*⟩
  | ‘`not`’ ⟨*unary-expr*⟩
  | ⟨*expr*⟩ ‘`or`’ ⟨*expr*⟩
  | ⟨*expr*⟩ ‘`and`’ ⟨*expr*⟩
  | ⟨*expr*⟩ ‘`xor`’ ⟨*expr*⟩

⟨*unary-expr*⟩ ::= ⟨*signal*⟩ | ‘`(`’ ⟨*expr*⟩ ‘`)`’ | ⟨*bool*⟩

5.1.4 Constraints

The constraints section allows to specify certain assumptions about how the inputs of a PRS should behave (`'assume'`) and which circuit states are valid (`'assert'`). Currently, assert and assume statements can only refer to the current state of a circuit (no temporal operators). Assignments can be used to calculate temporary variables that can be used in the other expressions in the `'constraints'` sections. Expressions (i.e., ⟨*expr*⟩) used by constraint statements may use all inputs, output and local signals from the PRS as well as the temporary variables created by assignments in the constraints section itself. As with the rules, the sequence of the statements has no semantic value.

⟨*constraints*⟩ ::= ‘`constraints`’ { ⟨*constraint*⟩ }

⟨*constraint*⟩ ::= ⟨*ident*⟩ ‘`:=`’ ⟨*expr*⟩ [⟨*attributes*⟩] ‘`;`’
  | ‘`assert`’ ‘`(`’ ⟨*expr*⟩ ‘`)`’ [⟨*attributes*⟩] ‘`;`’
  | ‘`assume`’ ‘`(`’ ⟨*expr*⟩ ‘`)`’ [⟨*attributes*⟩] ‘`;`’

Consider the following trivial example. Since the assume constraint forbids that a dual-rail value where both true rails of the vector a are asserted is applied to the circuit, the assertion can state that the output will also never attain this value.

```
prs demo is
inputs a : DRBit(2);
outputs x : DRBit(2);
begin
  x := wire(a);
constraints
  assume(not (a(0).T and a(1).T));
  assert(not (x(0).T and x(1).T));
end prs;
```

The content of the constraints section has no influence on the functionality of a PRS. It can, however, be used by a model checker for verification purposes (see Section 5.4).

5.1.5 Attributes

For prototyping and scientific purposes as well as to support data-flow abstractions (see Section 5.2.4) it is vital to be able to attach additional information to language constructs

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

that can easily be accessed from Python code. Hence, every signal declaration, instance, rule or constraint as well as PRSs themselves can be associated with a set of attributes, which can be used to store additional information about the respective object.

Attributes are specified using the `'attributes'` keyword, which is followed by a comma-separated list of key/value pairs in parentheses. Attribute keys must be (valid) identifiers, attribute values can be other identifiers, strings, numbers, Boolean values or lists of values. Lists are expressed using the `'list'` keyword followed by a comma-separated list of values in parentheses. They can contain an arbitrary mix of different value types.

⟨*attributes*⟩ ::= ‘`attributes`’ ‘`(`’ [⟨*attr-list*⟩] ‘`)`’

⟨*attr-list*⟩ ::= ⟨*attr*⟩ {‘`,`’ ⟨*attr*⟩ }

⟨*attr*⟩ ::= ⟨*attr-key*⟩ ‘`:=`’ ⟨*attr-value*⟩

⟨*attr-key*⟩ ::= ⟨*ident*⟩

⟨*attr-value*⟩ ::= ⟨*ident*⟩ | ⟨*string*⟩ | ⟨*number*⟩ | ⟨*bool*⟩ | ⟨*list*⟩

⟨*list*⟩ ::= ‘`list`’ ‘`(`’ ⟨*attr-value*⟩ {‘`,`’ ⟨*attr-value*⟩} ‘`)`’

The language itself does not specify any semantic meaning regarding attributes. However, the `pypr` package uses some attributes for special purposes. Those attributes should, therefore, only be used in the correct context.

Some examples for reserved attributes include:

- `keep`: The (Boolean) `keep` attribute can be set for signal declarations to indicate that the respective signal must not be removed during optimization (see Section 5.2.2).
- `needs_cd`: This attribute is used on input signals of PRSs to indicate that the respective signal needs a CD to avoid gate orphans and is used during logic synthesis (see Section 5.3).
- Channel related attributes (like `channel`, `channel_type`, `role`, etc.) are used to group signals into channels and support data-flow abstractions (see Section 5.2.4).

5.2 Python Production Rule Package

This section discusses the Python production rule package `pypr` that has been developed to create, analyze, modify and simulate asynchronous (and specifically dual-rail QDI)

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Table 5.1: pypr sub-packages

Sub-Package	Description
ast	Basic data structures to represent and manipulate the abstract syntax tree
bmc	Bounded model checking related data structures and functions
cg	Code generators (VHDL, Verilog)
dfg	Channel and data-flow graph related code and data structures
lib	Function rule definitions for standard gates (C gate, AND, OR, XOR, etc.), asymmetric C gates and threshold gates
opt	PRS optimizer
pg	PRS generators and circuit synthesis functions
pt	PRS transformation and modification algorithms
tb	Testbench generator

circuits. Since the package is built around the PRS description language presented in the previous section, it contains data structures that directly map to the syntax constructs of the language. The goal of this section is not to provide a detailed application programming interface (API) documentation of the package, but to give a general overview of its features, core data structures and algorithms. Some aspects of the package are more thoroughly discussed in Sections 5.3 and 5.4. For a detailed documentation please refer to the git repository[3].

The `pypr` package consists of several sub-packages which are listed in Table 5.1.

5.2.1 Basic Data Structures

To demonstrate how circuits are created and represented in `pypr`, we present a simple example script in Listing 5.1, that creates a WCHB pipeline for a single dual-rail bit. This circuit will be used as a running example in the following sections. All data structures used in this example script are provided by the `pypr.ast` sub-package.

```
wchb = PRS("wchb")
wchb.AddInput("reset", Bit)
wchb.AddInput("d_in", DRBit)
wchb.AddInput("ack_in", Bit)
wchb.AddOutput("d_out", DRBit)
wchb.AddOutput("ack_out", Bit)
wchb.AddRule(ProductionRule(var="en", up=BoolOpNode.NOT("ack_in")))
wchb.AddRule(FunctionRule(var="d_out.T", name="cgate",
  arguments=["d_in.T", "en"], init_condition="reset", init_value=False))
wchb.AddRule(FunctionRule(var="d_out.F", name="cgate",
  arguments=["d_in.F", "en"], init_condition="reset", init_value=False))
wchb.AddRule(FunctionRule(var="ack_out", name="or_gate",
  arguments=["d_out.T", "dout.F"]))

pl_length = 3
```

---

[3] https://gitlab.ecs.tuwien.ac.at/eda/pypr.git

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

```
16 pl = PRS("pl")
17 pl.AddInput("reset", Bit)
18 pl.AddInput("d_in", DRBit)
19 pl.AddInput("ack_in", Bit)
20 pl.AddOutput("d_out", DRBit)
21 pl.AddOutput("ack_out", Bit)
22 last_ack = "ack_out"
23 last_data = "d_in"
24 for i in range(pl_length):
25   pl.AddInstance(f"s{i}", wchb, inline_connectors={
26     "d_in": last_data, "ack_out": last_ack, "reset":"reset"})
27   last_ack = f"s{i}->ack_in"
28   last_data = f"s{i}->d_out"
29 pl.AddWireRule(dest="d_out", src=last_data)
30 pl.AddWireRule(dest=last_ack, src="ack_in")
31 pl.Attributes["desciption"] = f"A {pl_length}-stage WCHB pipeline."
32
33 lib = PRSLib()
34 lib.AddPRS(wchb)
35 lib.AddPRS(pl)
```

Listing 5.1: Python script to generate a 3-stage single-bit WCHB pipeline

Lines 1 to 13 create a WCHB named `wchb`. The input and output acknowledgment wires of this PRS are single-bit signals while the actual data signals use the `DRBit` data type. Then, four rules (i.e., gates) are created, two C gates as storage elements, one OR gate as CD and one inverter to control the C gates.

The actual pipeline is generated in Lines 16 to 31. The pipeline length (i.e., the number of buffers) is controlled by the variable `pl_length`. The overall interface of the pipeline is the same as for the individual buffers. The for loop starting in Line 24 adds `wchb` instances to the `pl` PRS and creates inline connectors to connect the individual stages with each other. Finally, the last pipeline stage is connected to the `ack_in` and `d_out` interface signals.

In order to keep track of multiple PRSs, `pypr` organizes them into libraries, which basically serve as containers for `PRS` objects. Hence, in Line 33 a `PRSLib` object is created and both created PRSs (i.e., `wchb` and `pl`) are added to this library.

All data structures provided by the `pypr` package feature the `ToCode` function, which converts the respective abstract syntax tree element represented by the object to a string according to the grammar rules from Section 5.1. Calling this function on the `PRSLib` object from the example script yields Listing 5.2. Using this function PRSs can be dumped to files, either simply for storage or to enable another tool to process them. Moreover, the function is quite useful for debugging purposes, since it allows to quickly create a human readable representation of complex data structures. The package naturally also offers the possibility to parse the PRS language. For that purpose, `pypr` provides the functions `ParsePRS` and `ParsePRSFile`, which are able to parse (Python) strings or whole files, respectively, and return `PRSLib` objects. Hence, taking the string produced by the `ToCode` function and passing it to the `ParsePRS` function again yields an equivalent `PRSLib` object.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

```
prs wchb is
inputs
  reset : Bit;
  d_in : DRBit;
  ack_in : Bit;
outputs
  d_out : DRBit;
  ack_out : Bit;
begin
  en := rule(not ack_in);
  d_out.T := cgate(d_in.T, en) init(0, reset);
  d_out.F := cgate(d_in.F, en) init(0, reset);
  ack_out := or_gate(d_out.T, d_out.F);
end prs;

prs pl is
  attributes(description := "A 3-stage single-bit WCHB pipeline.");
inputs
  reset : Bit;
  d_in : DRBit;
  ack_in : Bit;
outputs
  d_out : DRBit;
  ack_out : Bit;
instances
  s0  := wchb(d_in:=d_in, ack_out:=ack_out, reset:=reset);
  s1  := wchb(d_in:=s0->d_out, ack_out:=s0->ack_in, reset:=reset);
  s2  := wchb(d_in:=s1->d_out, ack_out:=s1->ack_in, reset:=reset);
begin
  dout  := wire(s2->d_out);
  s2->ack_in := wire(ack_in);
end prs;
```

Listing 5.2: PRS language representation of the 3-stage WCHB pipeline

5.2.2 Optimization

The pypr framework contains an optimizer in the `pypr.opt` sub-package that is able to perform basic circuit simplifications. An important application of the optimizer is the removal of superfluous wire rules introduced during flattening (Section 5.2.3). However, besides that it also resolves constant expressions and removes unused or redundant logic. In particular the steps performed by the optimizer are the following:

- Wire rule removal:
  Generally the optimizer removes all wire rules that don't have a delay associated with them. Since such wire rules essentially state that two signals are equal the optimizer will pick one of them and replace all occurrences of it in all other rules (and inline connectors) with the other one. However, if none of the signals can be removed (e.g., an input that is directly assigned to an output or an instance connector) the wire rule is kept.

```
b := wire(a);
x := rule(b or c);
```

↦

```
x := rule(a or c);
```

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

- Constant expression resolution:
  In this optimization pass the optimizer repeatedly goes through all rules of the PRS and applies simple combinational optimizations on a per rule basis (i.e., optimizing the up and down conditions) while simultaneously creating a list of all constant signals and their values, which are in turn used for further combinational optimizations. Constant signals are identified by wire rules connecting the signal to either true or false (e.g., `a := wire(true);`). All such wire rules are immediately removed from the PRS (the only exception are assignments to output signals). Consider the following example, showing an excerpt of the rules section of some PRS (assume that all involved signals are locals).

  ```
  x := rule(a or a);
  y := rule(a or true);
  z := rule(a and y);
  ```

  ↦

  ```
  x := wire(a);
  z := wire(a);
  ```

  The first rule is simplified to a wire rule, while the second one is removed altogether. With the acquired knowledge that the signal `y` is true, the third rule can also be simplified to a wire rule. The optimizer repeats this process until no further changes are possible.

- Inverter removal:
  If the optimizer encounters an inverter, whose input signal is only read by this particular inverter, it will try to merge it into the gate driving its input.

  ```
  x := and_gate(a, b);
  y := rule(not x);
  ```

  ↦

  ```
  y := nand_gate(a, b);
  ```

- Equal function rule removal:
  If a PRS contains two (or more) function rules that perform the exact same operation, one will be replaced by a wire rule. The newly created wire rule will then be removed by next wire rule removal pass.

  ```
  x := or_gate(a, b);
  y := or_gate(a, b);
  ```

  ↦

  ```
  x := or_gate(a, b);
  y := wire(x);
  ```

All these steps are applied until no further changes to the circuit are possible.

The optimizer also allows to prevent certain signals from being removed using the `keep` attribute. If a local signal declaration (inputs and output cannot be removed anyway) has this attribute set to true, the optimizer will preserve that signal. However, this does not mean that the rule writing to that signal is preserved as is.

5.2.3 Flattening

From the language specification and the example in Listing 5.2 we know that `pypr` supports hierarchical designs, which means that a PRS can contain other PRSs as instances. However, many algorithms (e.g., the model checker, see Section 5.4) only operate on flat PRSs, i.e., PRSs that don't contain any instances. This restriction often significantly simplifies the implementation of the respective algorithm. Furthermore,

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

the optimizer, although fully capable to operate on non-flattened designs, generally also produces better (more efficient) results, when it operates on flat designs. This is because the circuit can be analyzed and optimized as a whole and the interfaces to instances don't have to be kept in place.

Hence, pypr provides the `Flatten` function in the `pypr.pt` sub-package. This function takes a `PRSLib` object and the name the top-level PRS, collapses all hierarchical circuit levels and returns a single flattened PRS, with the same functionality as the original. For that purpose, it recursively goes through all instances of a PRS, copies all rules of an instantiated PRS to the instantiating one and renames the used signals to avoid name clashes. This process generates a lot of wire rules which can then be removed in a subsequent optimization step.

Listing 5.3 shows the result produced by the `Flatten` function and a subsequent optimization pass when applied to the example from Listing 5.2. It can be seen that all instances and wire rules have been removed. Moreover, notice that inverters have been merged into the OR gates driving them.

```
prs pl is
  attributes(description := "A 3-stage single-bit WCHB pipeline.");
inputs
  reset : Bit;
  d_in : DRBit;
  ack_in : Bit;
outputs
  d_out : DRBit;
  ack_out : Bit;
locals
  s1__d_in : DRBit;
  s2__d_in : DRBit;
begin
  s1__d_in.T := cgate(d_in.T, s0__en) init(0, reset);
  s1__d_in.F := cgate(d_in.F, s0__en) init(0, reset);
  ack_out := or_gate(s1__d_in.T, s1__d_in.F);
  s2__d_in.T := cgate(s1__d_in.T, s1__en) init(0, reset);
  s2__d_in.F := cgate(s1__d_in.F, s1__en) init(0, reset);
  s0__en := nor_gate(s2__d_in.T, s2__d_in.F);
  s2__en := rule(not ack_in);
  d_out.T := cgate(s2__d_in.T, s2__en) init(0, reset);
  d_out.F := cgate(s2__d_in.F, s2__en) init(0, reset);
  s1__en := nor_gate(d_out.T, d_out.F);
end prs;
```

Listing 5.3: Flattened and optimized 3-stage WCHB pipeline

5.2.4 Channels and Data-Flow Structures

In pypr attributes are used to annotate input and output signals of PRSs in order to group them into channels. For that purpose, the `channel` attribute specifies the name of the channel to which a certain signal belongs. The `role` attribute then further defines the function of the signal within the channel. For at least one signal of a channel (per convention the acknowledgment signal should be used) the `channel_type` attribute must be added. Currently the channel types `BD` and `DIDR` are supported, representing BD and

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

(4-phase) DI dual-rail channels, respectively[4]. Some signals, like reset inputs, have a `role` attribute but are not associated with any particular channel.

Listing 5.4 shows the interface signals of the WCHB PRS from Listing 5.2 with channel annotations.

```
prs wchb is
inputs
  reset : Bit attributes(role:=reset);
  d_in : DRBit attributes(channel:=chin, role:=data);
  ack_in : Bit attributes(channel:=chout, channel_type:=DIDR, role:=ack);
outputs
  d_out : DRBit attributes(channel:=chout, role:=data);
  ack_out : Bit attributes(channel:=chin, channel_type:=DIDR, role:=ack);
rules
  # [...]
end prs;
```

Listing 5.4: WCHB PRS annotated with channel attributes

Channels can also be created independently from an actual PRS using the `DIDRChannel` and `BDChannel` classes provided by the `pypr.dfg` sub-package. These data structures basically manage a set of (data and control) signals as well as some other important channel attributes (like, e.g., the channel direction, or information about the used protocol).

The channel information is essential within the `pypr` framework to be able to specify and create asynchronous circuits on the data-flow level (see Section 2.4). In particular this functionality is provided by the `DataFlowGraph` class, which is also located in the `pypr.dfg` sub-package. The script shown in Listing 5.5 generates the exact same PRS as Listing 5.1. However, here the individual signals of the instances don't have to be painstakingly connected manually, but are handled on the much more convenient channel level.

```
wchb = [...] # WCHB with channel annotations
channel = DIDRChannel(data=["d"])
pl_dfg = DataFlowGraph()
pl_dfg.AddInputChannel(channel, name="chin")
pl_dfg.AddOutputChannel(channel, name="chout")
for i in range(pl_length):
  pl_dfg.AddHandshakingBlock(f"s{i}", wchb)
pl_dfg.Pipeline(*["chin"] + [f"s{i}" for i in range(pl_length)] + ["chout"])
pl = pl_dfg.CreatePRS("pl")
```

Listing 5.5: WCHB pipeline created using the `DataFlowGraph` class

First, a channel object is created, which is then used to define the inputs and outputs of the top-level circuit represent by the `DataFlowGraph` object (`pl_dfg`). Then, the `AddHandshakingBlock` function is called repeatedly in the for loop to create the individual pipeline stages. The buffers created this way, are then connected with each other and the input and output channels using the `Pipeline` function.

---

[4] Note that the `channel_type` attributes actually makes some of the `role` attributes redundant. However, we still use this verbosity, to (i) make the code easier to read and (ii) avoid mistakes, since using this additional information some sanity checks can be performed.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Note that this example only shows a small subset of the features of the `DataFlowGraph` class. It is, for example, also able to implement forks and joins of channels and can handle multiple input and output channels on a data-flow element. The latter feature is vital to be able to process flow control elements like multiplexers and demultiplexers.

The class also distinguishes between handshaking and logic (i.e., function) blocks. Logic blocks don't have acknowledgment (or request) signals, they basically only modify the data signals of a channel.

To generate the individual (handshaking) blocks that can then be connected using the presented method, `pypr` provides a range of PRS generators in the `ast.pg` sub-package. The buffer used in the previous example can, for example, be automatically generated using the `DRBuffer` function from a suitable channel specification (i.e., a `DIDRChannel` object). Listing 5.6 demonstrates this feature.

```
wchb = DRBuffer(
  name="wchb",
  output_channel=DIDRChannel(data=["d"]),
  buffer_style=DRBufferStyle.WCHB
)
```

Listing 5.6: QDI dual-rail buffer PRS generator

The sub-package also provides generator functions for (QDI dual-rail) flow control components (multiplexers, demultiplexers, etc.). Moreover, it also supports logic synthesis, which will be explained in more detail in Section 5.3.

5.2.5 Code Generation

The `pypr.cg` sub-package provides code generators for VHDL and Verilog to export circuits specified in `pypr` to formats that can be processed by standard EDA tools, like digital simulators. Listing 5.7 shows the VHDL entity created from the WCHB pipeline example using the `ExportVHDL` function. A record type is used to represent the multi-rail type `DRBit`[5], for `Bit`-type signals `std_logic` is used. Also notice that the signal attributes are preserved in VHDL comments.

```
entity pl is
  port (
    d_in : in DRBit; --attributes(channel := chin, role := data)
    ack_in : in std_logic; --attributes(channel := chout, role := ack,
    ↪ channel_type := DIDR)
    reset : in std_logic; --attributes(role := reset)
    ack_out : out std_logic; --attributes(channel := chin, role := ack,
    ↪ channel_type := DIDR)
    d_out : out DRBit --attributes(channel := chout, role := data)
  );
end entity;
```

Listing 5.7: Entity produced by the VHDL code generator for the WCHB pipeline

---

[5] The Verilog code generator creates multiple single-bit signals to represent multi-rail datatypes, where the rail names are used to suffix signal names (e.g., `d_in_T` and `d_in_F`). For multi-rail vector types, a Verilog vector is created for each rail.

The way the actual rules of a PRS are translated is highly configurable. Per default a VHDL process is created for each rule. This process basically checks the up and down conditions (and if present the initialization condition) and sets the associated variable accordingly. However, it is also possible to change this behavior to create an actual instance of some gate library component instead.

5.2.6 Testbench Generation

To verify a circuit created with `pypr` through simulations in digital simulators, a testbench is required which models the environment of the circuit and checks whether the generated outputs are correct. Such testbenches often contain a lot of (repeating) boilerplate code. Hence, it makes sense to automate the process of testbench generation. For that purpose, the sub-package `pypr.tb` provides the `GenerateTestbench` function, which is able to generate complex VHDL testbenches from quite compact specifications.

This specification is a fully custom format and must be supplied in the form of a (nested) Python dictionary, which can either be directly constructed in the respective script or be loaded from a separate YAML file. Listing 5.8 shows an example testbench specification for the WCHB pipeline.

```yaml
channels:
  chin:
    log_tokens: true
    data: random
    timing: 1 ns
    token_limit: 8
  chout:
    log_tokens: true
    ack_delay: 2 ns
    check:
      type: function
      function: |
        is
        begin
          return chout.GetLatestToken.d_out =
            chin.GetToken(chout.GetLatestTokenIndex).d_in;
        end function;
general:
  reset: 10 ns
  start: 20 ns
```

Listing 5.8: YAML-based testbench specification for the WCHB pipeline

This example specifies a testbench that will generate eight random tokens on the input channel `chin`. The data signals of the input channel will take 1 ns to switch from the data to the spacer phase and vice versa (measured from the point in time of the last acknowledgment transition). For the output channel a function must be defined (as embedded VHDL code), that is then used to check the output tokens. This function has full access to the internal VHDL data structures of the testbench (`chout` and `chin`) that provide a level of abstraction over the asynchronous protocol and the raw interface signals of the unit under test (UUT) (i.e., the simulated PRS). It is also possible to access the full token history of each channel. Note that this example only shows a very limited

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

subset of the actual features of the testbench generator. In particular we are able to model much more complex channel behaviors, which includes the exact timing of the involved signal transitions as well as the actual content of data tokens and the causal dependencies between tokens on different channels.

The testbench generator can also handle VHDL entities as UUTs. To determine which interface signal belongs to which channel, the ports must be annotated by comments using the attributes syntax of the PRS language (an example for that is shown in Listing 5.7). This feature facilitates the use of the testbench generator for circuits *not* created by `pypr`.

Since, we are using the automatically generated testbenches for the fault-injection experiments performed in Chapter 6, monitors are included that check for various effects (transient) faults can have on the outputs of the UUT.

The testbenches created using the described process should work with all digital simulators supporting at least VHDL 2008. In particular we used and tested Questa and GHDL.

5.3 Dual-Rail Logic Synthesis

For synthesizing QDI dual-rail combinational circuits from a high-level hardware description in Verilog, `pypr` provides the `DRSynthesize` function in the `pypr.pg.qdi` sub-package. The reason for why we only support Verilog (rather than VHDL, which we use for our testbenches) is because Yosys [Wol], the underlying open-source synthesis tool, currently only supports Verilog. However, it will be trivial to extend the support to VHDL as soon as an appropriate front-end is added to Yosys.

The Verilog input must be a purely combinational single-rail description of the desired dual-rail function. Hence, it cannot contain any flip-flops or latches.

The synthesis process is split into two parts. First, a standard logic synthesis tool is used to process the Verilog specification (Section 5.3.1). The resulting intermediate netlist is then processed by a simple dual-rail expansion algorithm described in Section 5.3.2 that generates the final QDI dual-rail circuit.

To optimize the handling of arithmetic operations (such as addition, subtraction, comparisons) during the synthesis process, special care must be taken of adders, which will be addressed in Section 5.3.4.

5.3.1 Cell Library Specification and Synthesis

Before the actual synthesis process can be started a target dual-rail cell library must be defined. This library is then automatically converted to a single-rail library that contains the exact same gates and can be used by standard logic synthesis tools. This single-rail library is used to synthesize and technology-map the Verilog design, which is then further processed using the dual-rail expansion algorithm.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The cell library is specified as a `PRSLib` object containing a PRS for each available cell. Listing 5.9 shows an example in the form of a 2-input NCLX AND gate.

```
prs nclx_and is
  attributes(function:="and", area:=12);
inputs
  a : DRBit attributes(needs_cd := true);
  b : DRBit attributes(needs_cd := true);
outputs
  x : DRBit;
begin
  x.T := and_gate(a.T, b.T);
  x.F := or_gate(a.F, b.F);
end prs;
```

Listing 5.9: NCLX AND gate library element

The `function` attribute is used to indicate the actual logic function implemented by the PRS. Possible values are `"and"`, `"xor"`, `"fa"` and `"ha"`, where the latter two represent full adders (FAs) and HAs, respectively. Note that an inverter cell does not need to be (and actually must not be) contained in the cell library. The reason for this is that inverters are not "real" gates in dual-rail logic, but are rather just implemented by switching the true and false rail of a particular dual-rail bit. Hence, an inverter is always implicitly added to the library anyway (as we will see in the next section, they also require special treatment during dual-rail expansion). This is also the reason why it is sufficient for a minimal cell library to just contain a single (2-input) AND gate and why it is not necessary to add OR, NAND or NOR gates. Because the inversion is essentially "free" in dual-rail logic, those gates can simply be implemented using the AND gate. For each function (and number of inputs) the library may only contain one PRS.

The PRS attributes can also be used to specify the area requirements of a library cell, which will be used during technology mapping[6]. The meaning of the data signals is automatically detected (only relevant for full and half adders).

The data inputs and outputs of a cell must have the type `DRBit` and may also have attributes associated with them. Relevant for the synthesis (or more specifically the dual-rail expansion) is the Boolean `needs_cd` attribute on input signals (if it is missing it is assumed to be false). As discussed in Section 2.6.2 some dual-rail logic gates (e.g., NCLX gates) require CDs on their inputs to avoid orphans. The outputs of those CDs are then combined into an additional single-bit `done` output signal of the circuit. The reason for why the input CDs are not directly integrated into the library cells themselves, is efficiency. Depending on the actual use of a cell in the circuit the input CDs may in fact not be required after all. Hence, integrating them would potentially unnecessarily increase the area footprint of the circuit. Consider the following two scenarios:

- Imagine a circuit where two NCLX gates are connected to the same input signal. If both gates would bring their own input CDs the input completion of this signal would be checked twice.

---

[6] For HA and FA cells the `area` attribute is not used.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- Now consider the case where one of those NCLX gates would be changed to, e.g., a DIMS gate, which does not need to check for input completion. This would mean that the input CD of the remaining NCLX gate is not needed anymore since input completion is implicitly checked by the second (DIMS) gate.

To summarize these observations: In order to avoid orphans all signals in a circuit must be observed. Hence, a CD is only required for those signals which are *only* read by cells that need input completion and then only a single CD is required.

Library cells may also have an additional single-bit output, whose `role` attribute is set to `"done"` (or `"done_n"` referring to a low active version of this signal). These signals represent the outputs of internal CDs and are combined into a single `done` output during dual-rail expansion. Some of the FAs presented in Section 5.3.4 feature this output.

After the specified dual-rail cell library passed to the `DRSynthesize` function has been processed and checked, the single-rail version of the library is created. This library is then used during synthesis and technology mapping of the Verilog design which is performed by Yosys [Wol]. The resulting netlist is then passed to the dual-rail expansion algorithm.

5.3.2 Dual-Rail Expansion Algorithm

After the Verilog code has been synthesized and technology mapped to (single-rail versions of) our available library cells, the dual-rail expansion can be performed. The technology mapped circuit is represented by a set of instances of (library) cells and a set of (single-bit) signals connecting them. Each input and output of a cell instance is connected to exactly one signal. Of course for every signal there must be exactly one cell *output* connected to it (i.e., there can only be a single driver). However, signals can be connected to an arbitrary amount of cell *inputs* (i.e., they can be read multiple times). Inputs and outputs of the overall circuit (i.e., the interface signals of the original Verilog module) are represented as special input and output cells. Input cells only have outputs, while output cells only have inputs. For vector inputs (outputs) the corresponding interface cell has multiple outputs (inputs). Each of these inputs and outputs is again connected to exactly one signal. For the inputs of output cells the `needs_cd` attribute is assumed to be false. The rationale behind this is, that the outputs of a function block are always observed by some other part of the circuit because, if they weren't they should not exist in the first place.

The algorithm keeps a list of all signals and associates a Boolean value with them, representing whether they are observed or not. Initially all signals are marked as unobserved.

Before the dual-rail cell instances can be created, a data structure, which we refer to as the observation equivalence map is created. This data structure maps every signal to a set of signals, which are equivalent regarding the requirement for when they can be considered observed. If one of the signals in this set is observed, then all members of the set can be considered observed. The reason for why this is necessary are the inverters in the circuit. Recall that in dual-rail logic inverting a signal does not involve any actual

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

gates but just a rewiring of the true and false rails. Hence, when the output (input) of an inverter is observed the input (output) is observed as well, since physically we are dealing with the same pair of signals. Whenever a signal is marked as observed, the observation equivalence map is consulted, to check if any other signals have to be marked as observed as well.

First, the algorithm creates an empty PRS and adds local signal declarations for all signals. Obliviously the type of these signals must be `DRBit`. Then, it goes through the list of all cell instances and performs one of the following actions depending on the cell type.

- Output cells:
  For each output cell an output signal is added to the PRS. If the cell only has a single input the type of this output is `DRBit`, for multiple inputs a `DRBit` vector is used. Then, wire rules are added to connect the actual output signals to the signals driving them. Finally, the relevant signals are marked as observed.

- Input cells:
  For each input cell an input signal is added to the PRS. Similar to outputs cells a `DRBit` vector will be used for multi-output input cell. Wire rules connect the cell outputs to the relevant signals.

- Inverter cells:
  If an inverter is encountered two wire rules are added to the PRS, which basically assign the crossed-out true and false rails of the input signal of the inverter to the output signal.

- Other cells:
  For all other cells an instance of the actual library cell PRS is created. Wire rules are used to connect the local signals to the data inputs and data outputs of the instance. If the library cell has a `done` output, a new local signal of type `Bit` is declared and connected to it. This signal is then added to a list containing all the `done` signals. Each signal that is connected to an input of the library cell where the `needs_cd` attribute is set to false, will be marked as observed (of course taking into account the observation equivalence map).

When all cells have been processed the algorithm again goes through all the PRS inputs and sets the `needs_cd` attribute. For signals that are still unobserved after all cells have been instantiated the inputs driving these signals have their `needs_cd` attribute set to true. For input vectors where the individual dual-rail bits have different observation states a list of Boolean elements is used. If the `needs_cd` attribute is set to true the signal is also marked as observed.

Now the algorithm takes the list of signals and creates a list of unobserved signals, where it also needs to take the observation equivalence map into account. For each unobserved signal an internal CD in the form of a NOR gate is created.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Finally, a multi-input C gate is used to merge all the `done` outputs of the cell instances and the outputs of the internal CDs. The output of this C gate is used to drive the `done` output of the generated PRS. If the circuit does not contain cells with `done` outputs or internal CDs no top-level `done` output will be created.

To complete the process the generated PRS is flattened and optimized.

5.3.3 Synthesis Example

To demonstrate the described synthesis process, consider the Verilog specification of a simple combinational circuit over four inputs in Listing 5.10 consisting of an OR, an AND and an XOR gate.

```
module comb (a, b, c, d, z);
  input a, b, c, d;
  output z;
  assign z = (a | b) ^ (c & d);
endmodule
```

Listing 5.10: Verilog description of a simple combinational circuit

Synthesizing this circuit for an NCLX target library yield the PRS shown in Listing 5.11. The PRS has not been flattened to preserve its structure. It can be seen that for the three Boolean operations that are required to compute the output `z`, three PRS instances are created. The OR gate is implemented using an AND gate with inverted inputs (Line 20) and outputs (Line 21). Internal CDs are created for the outputs of the AND and the OR gate and their outputs are then merged into the `done` output of the PRS (Lines 27-30). Also notice that all inputs have their `needs_cd` attribute set to `true`.

```
 1 prs comb is
 2 inputs
 3   a : DRBit attributes(needs_cd := true);
 4   b : DRBit attributes(needs_cd := true);
 5   c : DRBit attributes(needs_cd := true);
 6   d : DRBit attributes(needs_cd := true);
 7 outputs
 8   z : DRBit;
 9   done : Bit attributes(role := done);
10 locals
11   int_sig_10 : DRBit;
12   int_sig_11 : DRBit;
13   int_sig_9_done_n : Bit;
14   int_sig_11_done_n : Bit;
15 instances
16   cell_0 := nclx_and(
17     a.F:=d.F, a.T:=d.T, b.F:=c.F, b.T:=c.T,
18     x.F:=int_sig_10.T, x.T:=int_sig_10.F);
19   cell_1 := nclx_and(
20     a.F:=b.T, a.T:=b.F, b.F:=a.T, b.T:=a.F,
21     x.F:=int_sig_11.F, x.T:=int_sig_11.T);
22   cell_2 := nclx_xor(
23     a.F:=int_sig_10.F, a.T:=int_sig_10.T,
24     b.F:=int_sig_11.F, b.T:=int_sig_11.T,
25     x.F:=z.F, x.T:=z.T);
26 begin
27   int_sig_9_done_n := nor_gate(int_sig_10.F, int_sig_10.T);
```

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
  int_sig_11_done_n := nor_gate(int_sig_11.T, int_sig_11.F);
  internal_done_n_merged := cgate(int_sig_9_done_n, int_sig_11_done_n);
  done := inv(internal_done_n_merged);
end prs;
```

Listing 5.11: Synthesis result

5.3.4 Adder Synthesis

Binary adders are very versatile generic function blocks that are heavily used in digital circuits. They are not only required to implement arithmetic addition, but are also needed for subtraction and comparison operations. Hence, to achieve well-performing and area-efficient circuits, it is reasonable to give adders special attention during synthesis. Over the years many different design approaches for adders have been proposed [WH11]. These range from the simple ripple-carry adder (RCA) to more sophisticated circuits like carry-lookahead adders and related strategies.

Since the operation speed of synchronous circuits is limited by the critical path of their combinational logic, a special focus in the design of adders is placed on the reduction of their worst-case delay. In that regard RCAs, although very area-efficient, perform particularly bad, because in the worst case the complete carry chain has to be traversed. However, as will be discussed in this section, this is not necessarily the case for QDI circuits. A well-designed QDI RCA can take full advantage of average case performance and does not suffer from the same issue as its synchronous counterpart.

Hence, our logic synthesis always implements adders as RCAs (if the supplied cell library provides an FA). Since, unfortunately the tool internally used by Yosys for technology mapping does not support multi-output function blocks like FAs, adders are implemented in a separate additional processing step during synthesis, which slightly complicates the synthesis process.

This section first gives some background on QDI RCAs in general in Section 5.3.4.1 and then shows how their basic building block, i.e., the FA, can be implemented in Section 5.3.4.2. Additionally, some novel FA implementations based on the use of SNs are presented in Section 5.3.4.3. Finally, the different FA versions are compared to each other in terms of speed and area-efficiency in Section 5.3.4.4.

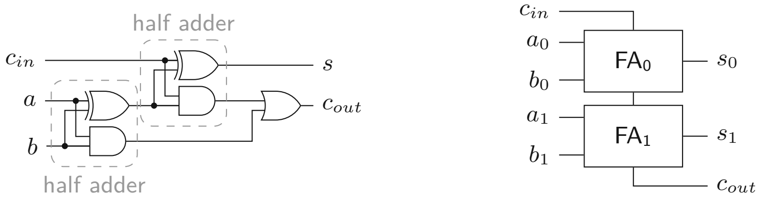
For the FA designs presented in this work we restrict our investigation to logic styles that only use basic gates readily available in standard libraries (with no more than 3 inputs) and 2-input C gates. The aim of this investigation is not to give a definite answer to the question which adder design is the best. We rather want to give an overview of the available designs and present some estimates on their area overhead and performance. The selection of a particular design depends on the available target technology. Therefore, we keep our analysis on a more abstract level.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

5.3.4.1 Ripple Carry Adders

RCAs are constructed form FAs (and HAs). An FA, such as the one shown in Figure 5.1a, is a digital circuit with three inputs a, b, and c_{in} (carry input) and two outputs s (sum) and c_{out} (carry output). Its purpose is to add the three (single-bit) input signals and produce a 2-bit result consisting of the sum bit (i.e., the LSB of the 2-bit result) and a carry output. To add binary numbers of arbitrary width n, an RCA consisting of n FAs (FA$_0$,...,FA$_{\mathrm{n-1}}$) can be used. As shown for the example of $n = 2$ in Figure 5.1b this is done by connecting the i-th bits of the input vectors $\mathbf{a} = (a_0, ..., a_{n-1})$ and $\mathbf{b} = (b_0, ..., b_{n-1})$ to the a and b inputs of FA$_\mathrm{i}$, while the carry output of FA$_\mathrm{i}$ is connected to the carry input FA$_{\mathrm{i+1}}$ to form a so called carry chain. The delay of this carry chain represents the critical path through an RCA in a synchronous circuit. Therefore, its optimization is critical for the attainable speed of the circuit.

As illustrated in Figure 5.1a, a single FA can also be viewed as being composed of two HA components. An HA just adds two single-bit numbers and is, hence, considerably simpler. For RCAs that don't require a c_{in} signal the first FA can be replaced by an HA.



(a) Full adder

(b) 2-bit RCA

Figure 5.1: Binary addition circuits

Recall from Section 2.6.2 that multi-output QDI function blocks can be classified as strongly and weakly indicating. In the context of QDI FAs in RCAs this is an important performance-determining property. Using strongly indicating FAs the carry signal *always* has to travel through the complete carry chain, because every FA only starts producing output values when it received its carry input.

To achieve good average performance a weakly indicating FA is preferable, because it shortens the average distance a signal has to travel in the carry chain: If inputs a and b match (which is the case for statistically 50% of the inputs), then c_{out} is already determined irrespective of the value of c_{in}. This allows an early activation of c_{out} therefore effectively shortening the carry path; in essence it breaks the carry path into segments (depending on the data inputs) that can be processed independently and, more importantly, concurrently. This also occurs in synchronous implementations, but there the reliance on the worst-case timing prohibits making use of this effect. QDI circuits, in contrast, can fully benefit from this average case timing – if they are weakly indicating.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

5.3.4.2 Full Adder Designs from Literature

The first QDI FA we want to discuss is the version using 2-input DIMS gates. A first naive approach to such an implementation is to replace each gate in Figure 5.1a with its DIMS version, which each consist of four C gates and one or two OR gates, depending on the Boolean function. However, notice that the two pairs of XOR and AND gates (i.e., the two HAs) share the same inputs, meaning that the C gates in their DIMS versions would perform the exact same operation. Hence, these gates can be shared between the DIMS AND and XOR gates. A different optimization for this design style called “Input Completeness Relaxation” was proposed by Jeong and Nowick in [JN07]. This optimization essentially allows to replace the two DIMS AND gates with the simpler NCLX version consisting of just an OR and an AND gate. This circuit is the one we will use for our evaluation and refer to as DIMS-FA. We don’t consider a potential FA variant using a single 3-input DIMS gate. This is because such a solution would require 3-input C gates, which we want to avoid, and would bring little to no benefit when compared to other presented circuits.

Another possibility is to replace every gate in Figure 5.1a with its NCLX equivalent, resulting in the circuit shown in Figure 5.2. This circuit needs CDs for the three input signals as well as the three internal signals, which are also depicted in the figure. Notice that the marked gates can be shared between the XOR and AND gates. We will refer to this circuit as NCLX2-FA.

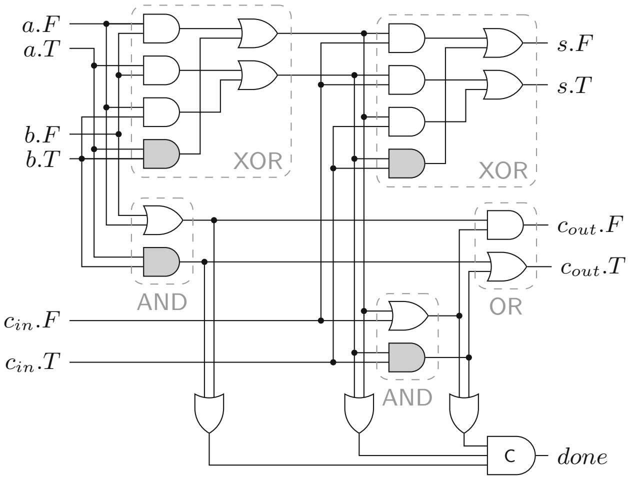


Figure 5.2: NCLX FA (input CD omitted from figure)

The NCLX design style can also be applied using 3-input gates, resulting in the circuit shown in Figure 5.3, which we will refer to as NCLX3-FA. For every input pattern only one of the AND gates is activated, the OR gates at the output then determine which output rails must be asserted. Since the AND gates switch to zero again as soon as one input is deasserted an input CD is required for this circuit. Notice that using C gates instead of AND gates would yield a DIMS circuit.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

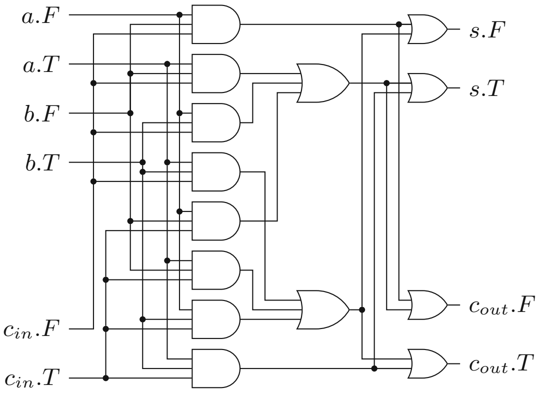


Figure 5.3: NCLX FA with 3-input gates (input CD omitted from figure)

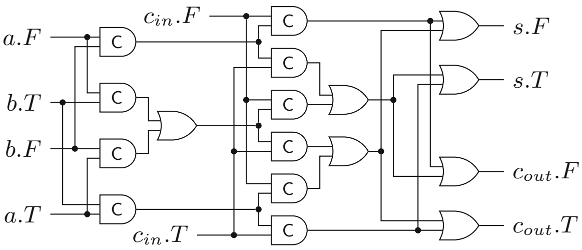


Figure 5.4: Toms' FA [Tom06]

In [Tom06] Toms presents a synthesis algorithm for QDI combinational circuits, which yields the FA shown in Figure 5.4. For every input value exactly one of the C gates in the first column and one in the second column switch to one. The circuit does not need an input CD and is strongly indicating. We will refer to this circuit as Toms-FA.

If this circuit is equipped with an input CD and an internal CD to collect gate orphans after the first column of C gates, the C gates can be replaced with AND gates. The resulting circuit, which we will refer to as TomsX-FA, is shown in Figure 5.5. Note that this circuit can no longer be classified as strictly strongly indicating. While it still only generates output data on s and c_{out} when all three input values have switched to the data phase, the *done* output may be asserted with only two inputs present. Moreover, when one input switches to the null phase the outputs will immediately switch to the null phase as well, only the *done* signal switches to low when all inputs are in the null phase. This behavior also applies to the NCLX3-FA.

5.3.4.3 Sorting-Network-based Full Adders

This section presents three of our own FA designs that are based on the use of binary SNs (see Section 4.4). Figure 5.6 shows the first of these circuits, which we will refer to as SN-FA. The SNs are essentially used to count the number of asserted true and false

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

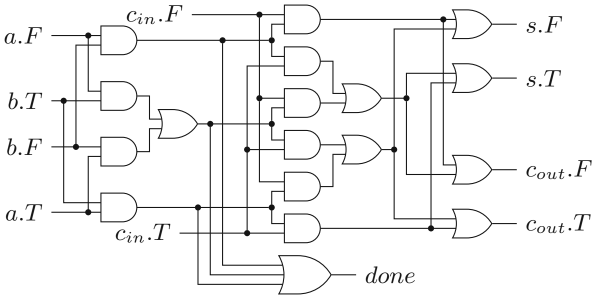


Figure 5.5: Modification of Toms' FA (input CD omitted from figure)

rails in the set of dual-rail input signals. If any of the SNs detects two asserted inputs (i.e., T_1^3 and T_2^3 switch to one), the respective carry output can be generated, which means that the circuit is weakly indicating. To generate the sum output, all three inputs must be present and one of two conditions must be satisfied. Either one SN detects three asserted inputs, which means that the inputs are either all true (i.e., their sum is one) or all false (i.e., the sum is zero). The other possibility is that one SN detects two asserted inputs and the other SN one. If two true rails and one false rail are detected, the sum is zero, in the complementary case it is one.

It is important to point out that this circuit does not need input or internal CDs. As already discussed in Section 4.4, SNs themselves don't contain gate orphans, if all outputs are observed correctly. For every combination of input values exactly three outputs of the combined set of outputs of both SNs will transition to one. When all those three intermediate transitions are involved in the generation of output values it is clear that there are no more orphan transitions that could happen at an output of a T^3. After the last output transitioned to data or null, one can be sure that all switching activity inside the circuit has completed.

Specifically, if on one SN T_1^3 is the only activated output (first case), it takes T_2^3 from the other side to activate the respective sum rail (according to our assumption, the lower input of the OR must be 0). The T_1^3 on the side where T_2^3 is asserted, is required to activate the corresponding carry rail. In case of all 3 outputs activated on one SN (second case), the respective carry rail collects the transitions from T_1^3 and T_2^3, while the transition at T_3^3 is observed at the respective sum rail. Hence, it is clear that in both cases all internal transitions are observed.

Notice that for generating the carry rail $c_{out}.x$ the corresponding output T_2^3 alone would be sufficient, as $T_2^3 = 1$ implies $T_1^3 = 1$. In fact the C gates joining these two signals are only there to keep the T_1^3 signals involved, and specifically avoid introducing gate orphans when switching to the null phase. Unfortunately, these additional C gates in the carry path introduce additional delay in the carry chain of an RCA, which is exactly where we need to optimize.

The circuit shown in Figure 5.7, which we refer to as the SNFC-FA, improves this issue

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

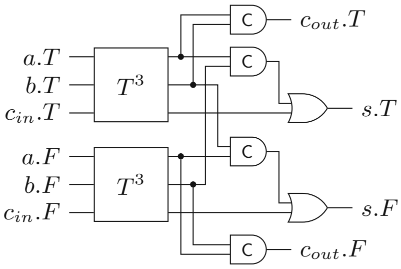


Figure 5.6: SN-based FA (SN-FA)

by removing the C gate from the carry path. However, to still keep the T^3_1 outputs involved in generating an output signal in the case where both SNs assert this output, additional C gates have to be introduced in the signal paths generating the sum output.

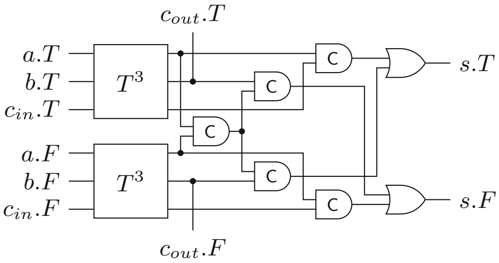


Figure 5.7: SN-based FA with fast carry generation (SNFC-FA)

The other possibility to create a fast carry path, but without compromising on the sum output delay, is presented in Figure 5.8 and will be referred to as the SNX-FA. This FA variant does not use C gates to implement the actual Boolean function and directly uses the T^3_2 outputs for c_{out}. In order for this to work correctly, an internal CD must be added to collect all orphan transitions that may arise. Having thus again ensured that all three output transitions of the two T^3 SNs are involved in generating the outputs for every possible input pattern, the circuit still does not require an input CD.

Note carefully, that we have now made the carry generation fast and by removing the C gates from the data path also improved the sum output delay. This optimizations came at the price of introducing the *done* signal, which, for some input patterns, resembles the slowest path in the cell now. Recall, however, that *done* is moving downstream, thus bypassing the next stage and its CD, and hence has some delay margin.

Table 5.2 shows an overview of the properties of the FA circuits presented in this section. Keep in mind that if an FA needs an input CD then every internal carry signal along the carry chain of an RCA also needs a CD.

---

[7] Weakly indicating on the data rails when switching to the spacer, see explanation above.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

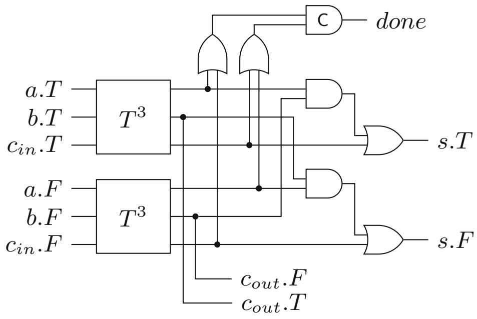


Figure 5.8: SN-based FA with explicit completion detection (SNX-FA)

Table 5.2: QDI FA properties

Circuit	Indication	Input CD	Internal CD
DIMS-FA	weak	no	no
NCLX2-FA	weak	yes	yes
NCLX3-FA	full[7]	yes	no
Toms-FA	full	no	no
TomsX-FA	full[7]	yes	yes
SN-FA	weak	no	no
SNFC-FA	weak	no	no
SNX-FA	weak	no	yes

5.3.4.4 Evaluation and Performance Considerations

This section evaluates and compares the presented FA circuits in terms of their area overhead and performance.

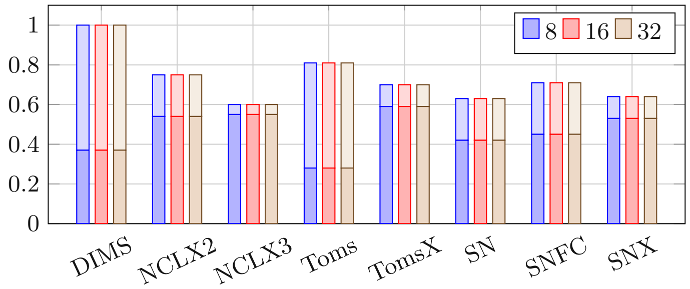
Area Analysis For the area analysis, we optimized the individual FAs using standard CMOS optimizations, reducing the number of non-inverting gates. As mentioned before we don't target any specific gate library with this analysis. Hence, for the evaluation we assume an area requirement of 1 GE for 2-input NAND/NOR gates and 0.5 GEs for inverters. C gates are assumed to require 2 GE, which, depending on the actual implementation may even be considered generous [SEE98]. Our analysis only considers the gate costs, other parameters like drive strength/fanout or routing are not considered.

First, we consider a classic RCA with a varying data width of 8, 16 and 32 bits and carry input and output signals. Figure 5.9a shows the area overhead of the presented implementation variants relative to the DIMS version, which we will use as the baseline for this analysis. The stacked bars in the figure indicate the proportion of the hardware dedicated to C gates (upper part) and combinational gates (lower part). Note that the cost for input CDs (needed for the NCLX2-FA, NCLX3-FA and TomsX-FA versions) is not included in these results, since we assume that the adder is directly fed from a

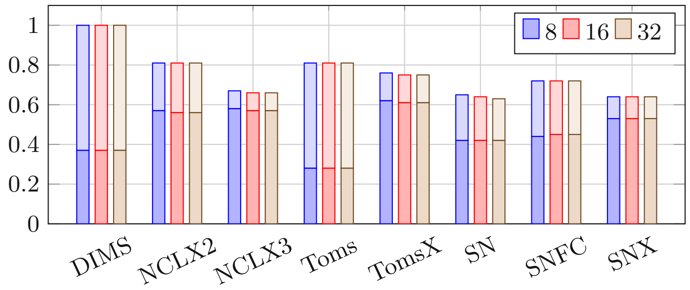
Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

buffer stage[8]. It can be seen that all the SN-based FA variants have a competitive area footprint. The fact that the NCLX3-FA variant performs best in this analysis is a little misleading. From the discussed circuits, this variant has the highest fanout on its input signals, which might require additional drivers, and quite complex routing requirements which are not factored in here. Unlike all other FAs presented in this work the maximum fanout for any signal in the SN-based FAs is three.

For the second evaluation we implemented an adder tree consisting of two levels of RCAs adding four numbers. Here we used HAs for the lower-most bits of the individual RCAs, where an NCLX-style HA was used for the NCLX2-FA, NCLX3-FA, TomsX-FA and SNX-FA versions, while the other used DIMS HAs. Figure 5.9b shows the results. Since some circuits need CDs at the input of the second-level adder, the NCLX2-FA, NCLX3-FA, TomsX-FA version perform worse than in the previous test. The SNX-FA version now even surpasses the NCLX3-FA version. It is also worth noting that with the exception of the NCLX3-FA the SNX-FA version requires the least amount of C gates. This means that assuming a more expensive C gate implementation, the SNX-FA version would perform even better when compared to the other variants.



(a) Single RCA



(b) 2-level adder tree of RCAs

Figure 5.9: Area overhead of the presented adder circuits (normalized to the DIMS RCA)

---

[8] Note that this assumption combined with the “Input Completeness Relaxation” approach from [JN07] could be used to further optimize the DIMS-FA and the Toms-FA variants resulting in a hybrid NCLX/DIMS solution, which relies on an input CD provided by the source buffer. However, such specific circuit-level optimizations go beyond the scope of this analysis. Hence, we restrict the discussion to the “pure” DIMS solution.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Performance For the performance evaluation of the presented circuits we derive a simple mathematical model that describes their behavior in an RCA with input data width n. For that purpose, we assume that all inputs arrive at the RCA simultaneously and that their values are uniformly distributed (i.e., every input data bit is equally likely to be zero or one regardless of the other input bits).

Let us define the following delay parameters.

- Early carry delay Δ_E: The delay to generate c_{out} out of the inputs a and b for the case where $a = b$ (this only applies to weakly indicating FAs).
- Late carry delay Δ_L: The delay to generate c_{out} when c_{in} arrives, assuming that a and b have already propagated through the circuit as far as possible. Hence, internal nodes only dependent on a and b have already settled to a value.
- Sum delay Δ_S: The delay to generate the s output when c_{in} arrives, again assuming the same condition for a and b as before.

Given these parameters we can model the average delay Δ_{RCA} the RCA needs to produce its outputs. As already mentioned, the most important factor for the performance of an RCA is the length of its carry chain. If the RCA consists of strongly indicating FAs, always the complete carry chain has to be traversed, which means that their average case delay is equal to the worst case. Thus, their overall delay (Equation (5.1)) is given by the carry delay through $(n-1)$ FAs and the final delay to generate the MSB s signal or the c_{out} signal (depending on which circuit path is slower).

$$\Delta^F_{RCA} \approx (n-1)\Delta_L + \max(\Delta_L, \Delta_S) \tag{5.1}$$

We only use the approximate symbol, because the equation does not fully consider the delay through the first FA of the chain. However, for large n this discrepancy is not significant.

The calculation of the RCA delay for the case of weakly indicating FAs is a little bit more involved. In a first step we need to determine the average length of the longest carry chain segment for a given input data width. Because we assumed uniformly distributed input values, for every possible input value combination there is an equal chance that a particular FA in the adder is able to generate its c_{out} signal with just the inputs a and b or that it has to wait for the c_{in} signal. This means that every possible input value configuration fits one of these 2^n equally likely cases. Hence, we can simply enumerate every possible binary vector of length n, determine the longest sequence of zeros and take the average of these values. We denote this function with $l_c(n)$. A numerical analysis showed that $l_c(n)$ grows quite slowly $(l_c(8) = 2.1, l_c(16) = 3.2, l_c(24) = 3.8)$. The final adder delay $\Delta^W_{RCA}(n)$ is then given by Equation (5.2). Again the approximate symbol is used for the same reason as before.

$$\Delta^W_{RCA}(n) \approx l_c(n) * \Delta_L + \Delta_E + \Delta_S \tag{5.2}$$

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

When comparing the two equations it is apparent that the weakly indicating FAs have quite a large performance advantage over the strongly indicating ones. Table 5.3 shows estimations for the delay parameters[9] for the weakly indicating FAs based solely on the gate delays involved (routing/wire delays are not considered).

Table 5.3: Estimated full adder delay parameters

Circuit	Δ_E	Δ_L	Δ_S
DIMS-FA	8.5	4.75	3.75
NCLX2-FA	3	2	2
SN-FA	5.75	4	5.9
SNFC-FA	3.75	2	6.6
SNX-FA	3.75	2	3.5

As can be seen from the table, all our proposed approaches perform better than the DIMS version. The SNX-FA even has comparable delay to the NCLX2-FA version. However, these numbers must be interpreted cautiously, because circuit layout and the target technology also play an important role in the real-world performance of these circuits.

Another important point we need to address here is the delay when switching to the null phase. Again an RCA consisting of fully indicating FAs is at a disadvantage, because also in the null phase the carry signal has to traverse the complete carry chain before every individual FA is reset. However, some of the weakly indicating circuits suffer from a similar albeit less severe issue. If the carry output of the SN-FA was generated solely on the basis of the inputs a and b (i.e., for the case where $a = b$), then resetting just those inputs also resets c_{out}. If on the other hand the c_{in} signal was involved in generating c_{out} the circuits waits until c_{in} resets before letting c_{out} enter the null phase. Hence, the same carry segments that determined the delay of the circuit in the data phase also affect the null phase. Notice that the SNFC-FA and SNX-FA variants don't behave like that. A similar issue arises in the DIMS-FA, where if $c_{in}.F$ is asserted, this rail must be deasserted before c_{out} can enter the null phase.

5.4 Bounded Model Checking

The `pypr` framework also has some basic bounded model checking (BMC) capabilities to verify the correctness of SI circuits with regard to certain properties (within certain bounds). For that purpose, we use Z3. The advantage of Z3 is that it is not "just" a Boolean satisfiability (SAT) solver, but a satisfiability modulo theories (SMT) solver, allowing to reason about more general classes of problems. This feature can come in handy for future extensions of the model checker. Moreover, the solver can handle arbitrary Boolean input formulas, i.e., it is not restricted to an input clause set in conjunctive

---

[9] For this estimation, we assumed a unit delay of 1 for 2-input NAND/NOR gates, 1.25 for 3-input NAND/NOR gates, 0.75 for inverters, and 2 for C gates.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

normal form (CNF). This is a convenient feature as it makes it a lot easier to formulate the SAT problem.

After a brief introduction to BMC for synchronous circuits in Section 5.4.1, Section 5.4.2 goes on to show how the presented techniques can be adapted and applied to SI/QDI circuits. Then, some of the checkable properties supported by our framework are discussed in Section 5.4.3. Section 5.4.4 shows how to model transient faults within the model checker. For a practical demonstration of the framework Section 5.4.5.1 includes two simple application examples. Finally, Section 5.4.6 provides some insight into the computational complexity of our approach.

5.4.1 Introduction and Related Work

Model checking is a well established technique to verify the correctness of digital circuits or to find (potential) problems in them [BK18]. For BMC the core idea (as also used in this work) is to convert a circuit into a SAT problem and model its state progression for a certain (bounded) amount of time. In the domain of synchronous circuits the SAT problem simply models how the circuit's register values change with each clock cycle (given some initial state). To visualize this approach consider the circuit in Figure 5.10 showing a 3-bit LFSR, consisting of three registers r_1 to r_3 and an XOR gate (disregard the NOR gate and the *err* output for now).

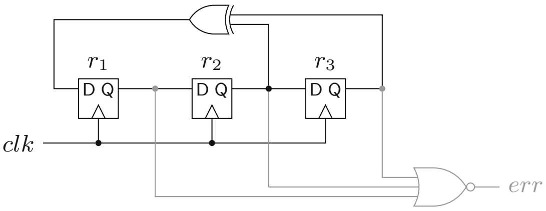


Figure 5.10: 3-bit LFSR

In the following we will use the notation $r_{[n]}$ to denote the value of the register r in clock cycle n, where $r_{[0]}$ refers to the initial (i.e., reset) value of a register. The clause set that models the behavior of this LFSR is, thus, given by[10]:

$$\begin{aligned}
\neg r_{1[0]},\ \neg r_{2[0]},\ r_{3[0]}, &\qquad \text{inital state}\\
r_{1[1]} = (r_{2[0]} \oplus r_{3[0]}),\ r_{2[1]} = r_{1[0]},\ r_{3[1]} = r_{2[0]}, &\qquad \text{state after first clock cycle}\\
r_{1[2]} = (r_{2[1]} \oplus r_{3[1]}),\ r_{2[2]} = r_{1[1]},\ r_{3[2]} = r_{2[1]}, &\qquad \text{state after second clock cycle}\\
\dots &\\
r_{1[b]} = (r_{2[b-1]} \oplus r_{3[b-1]}), r_{2[b]} = r_{1[b-1]}, r_{3[b]} = r_{2[b-1]} &\qquad \text{state after } b \text{ clock cycles}
\end{aligned}$$

---

[10] For a clause set to be considered satisfiable *all* clauses must evaluate to true, hence a clause set can also be viewed as one large Boolean logic formula given by the conjunction of all clauses.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Note that we don't use a CNF here, since this would unnecessarily complicate the example. To convert the clause set into a CNF the Tseytin transformation can be used. However, modern solvers such as Z3 can often directly handle clauses in propositional logic.

The first three unit clauses[11]encode the initial (i.e., reset) state of the circuit, which is given by $r_1 = r_2 = 0$ and $r_3 = 1$. The following three clauses model the state after the first clock cycle. It can be seen that the state in cycle n is derived from the previous state (i.e., the state in cycle $n-1$) through some Boolean transformations which are determined by the combinational logic of the circuit. Hence, given a suitable representation of a circuit (often and-invert-graphs are used for this purpose) the creation of this clause set is quite straightforward. However, it is apparent that during creation we must choose a certain bound b up to which the circuit progression is modeled. This is where the boundedness comes from.

Circuit inputs can, e.g., be modeled using free variables (i.e., variables unconstrained by any clause). However, as we will see in the following section, inputs must often adhere to a certain protocol, which must then also be modeled using appropriate clauses.

Now that we modeled and encoded the behavior of the circuit itself, we also need some clauses that actually verify certain properties that we are interested in. A common question is whether a given circuit can ever enter an unsafe/forbidden or invalid state (safety property). For our simple example, we want to check if the circuit can ever enter a state where all registers are zero. Since LFSRs cannot produce this value, this would indicate a problem with the circuit. To do that we can add some extra logic to the circuit (i.e., the NOR gate) that checks for this state and produces a single output err that indicates whether the state has been entered. This additional logic is basically a circuit-level representation of an assertion, in this specific case $r_1 \vee r_2 \vee r_3$. The clause set is then extended accordingly:

$$\begin{aligned}
&err_{[0]} = \neg(r_{1[0]} \vee r_{2[0]} \vee r_{3[0]}),\\
&err_{[1]} = \neg(r_{1[1]} \vee r_{2[1]} \vee r_{3[1]}),\\
&\ldots\\
&err_{[b]} = \neg(r_{1[b]} \vee r_{2[b]} \vee r_{3[b]})
\end{aligned}$$

Finally, we add a clause that states that the err signal will be asserted at least once (i.e., for at least one clock cycle).

$$\bigvee_{0 \leq n \leq b} err_{[n]}$$

If we now run the SAT solver and it is not able to find a model (i.e., a variable assignment that satisfies the SAT problem), we can conclude that the circuit never enters the unsafe state (within b clock cycles). If a model is found, we can immediately get a concrete (counter) example of how the circuit got into the state and can further investigate it.

---

[11] A unit clause only consists of a single literal and states whether a variable must be true or false. For example, the unit clause $\neg a$ states that the variable a must be false.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

This also brings us to the limitation of *simple* BMC. The obtained results are only valid up to the chosen bound b. Even if a circuit behaves correctly up to bound b, it may still violate some assertion at step $b + 1$. Hence, selecting the "right" bound (completeness threshold) is a non-trivial problem. On the other hand, it is guaranteed that if there is the possibility to enter a certain (e.g., unsafe) state within the chosen bound, the model checker will find it. However, there are further techniques, that allow to overcome this bound limitation and derive universal results which hold in general and not just up to some bound. In particular this includes induction- or interpolation-based approaches [BK18].

Regarding model checking of asynchronous and more specifically SI circuits the discussed approach needs to be modified to a certain degree. How this is done is subject of the next sections. There are some other model checking approaches for asynchronous circuits in literature. In [TM18] a commercial synchronous tool is used to verify SI (and mixed-style) circuits, by first creating a suitable synchronous transformation of the circuit. An STG-based approach is presented in [KKY06, SKM+15]. Another approach is presented in [BRG+18], which, however, operates on a higher abstraction layer (SystemVerilog is used for design entry). For verification the circuit is transformed such that it can be processed by the tool CADP (an acronym for "construction and analysis of distributed processes").

The use of a SAT-based approach for verifying the delay insensitivity of circuits (absence of orphans) is presented in [KNR+02]. However, the focus is on partitioning large circuits to make model checking feasible and the construction of the SAT problem is not described.

In this section we want to provide a clear path from a circuit description and properties to be verified, to a SAT problem understood by a SAT solver to facilitate formal verification of (asynchronous) circuits created using `pypr`.

5.4.2 Encoding PRSs as SAT problems

This section describes how an SI PRS can be converted into a SAT problem. The process that will be discussed assumes a flat PRS (i.e., a PRS without instances) with only single-bit wire rules.

The bound to which the circuit is unrolled is denoted by the variable b. Since there is no clock signal in asynchronous circuits that can be used to model the progression of time in discrete steps a different abstraction has to be found. Moreover, as already implicitly assumed in the previous section, when model checking synchronous circuits gate delays are completely disregarded. It is simply assumed that the combinational logic evaluates within one clock cycle. This assumption is perfectly fine for synchronous circuits as it represents the core abstraction for the design style. However, recall that in the SI delay model, gate delays can be arbitrarily long – a fact that must somehow be reflected by our model. Hence, the circuit's (state) progression is not modeled using clock cycles, but production rule evaluation rounds. During a step each rule of the PRS may evaluate if its up or down condition is satisfied but it doesn't have to. This way the SI delay model

TU Bibliothek, WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

is accommodated by the model checker. Similar techniques have been used in literature [TM18]. The resulting SAT problem encodes exactly b such rule evaluation steps. The SAT solver tasked with this problem will, thus, explore every possible rule firing sequence corresponding to every possible delay configuration of the circuit. As a consequence of this approach the computational effort of model checking per gate of an SI circuit is much higher than for a synchronous circuit.

The state of a circuit and its environment for some step n is defined by the values of a set of variables. These variables include all input, output and local signals of a circuit as well as some auxiliary variables required to describe and constrain the dynamic behavior of the circuit or to formulate and check certain properties. Thus, we use $x_{[n]}$ to denote the value of the variable x in rule evaluation step n, where $0 \leq n \leq b$. Using this notation $x_{[0]}$ refers to the initial value of some variable x, while $x_{[1]}$ models the state after the first rule evaluation step. Since internally Z3 is employed as the actual SAT solver, all variables can conveniently be represented using the built-in bit vector type for Boolean variables.

In the context of the model checker a DI RZ channel C is defined by the tuple (D, ack), consisting of a set of (multi-rail) data signals $D(C)$ and an acknowledgment signal $ack(C)$. Each data signal $d \in D(C)$ is represented by a set of rails. For a dual-rail signal, this rail set is given by $\{T, F\}$, i.e., the true and the false rail. To denote the individual rails $r \in d$, the notation $d.r$ is used. This means that a dual-rail bit d consists of the rails $d.T$ and $d.F$. Currently the model checker only supports dual-rail input and output channels.

We use the variables C^I and C^O to refer to the set of input and output channels of a PRS, respectively.

A production rule r, in the context of model checker, is defined by the tuple $(x, up, down)$, where $x(r)$ refers to the variable the rule writes to while $up(r)$ and $down(r)$ refer to the Boolean expressions that determine if $x(r)$ should be asserted or deasserted, respectively. If $up(r) = \neg down(r)$ the rule r is *combinational*, otherwise it is *state-holding*. A combinational rules, where $up(r)$ just contains a single signal, is a wire rule.

5.4.2.1 Initialization

To encode the initial condition of the circuit into the SAT problem, the value of each input, output and local signal x of the PRS must be determined and encoded using a unit clause, effectively fixing the value of $x_{[0]}$. However, in some cases the initial value of a signal cannot be derived, which results in a not fully constrained initial state. If this signal then fans out to other production rules this uncertainty can spread.

Consider the following example PRS consisting of an inverter, a C gate (without `init` expression) and an XOR gate.

```
prs demo is
inputs a : Bit; b : Bit; c : Bit;
outputs z : Bit;
rules
```

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

```
  not_b := inv(b);
  x := cgate(a, not_b); # no init expression
  z := xor_gate(x, c);
end prs;
```

Assume that initially all inputs are zero, which would be encoded using three unit clauses:

$$\neg a_{[0]}, \quad \neg b_{[0]}, \quad \neg c_{[0]}$$

The value for the local *not_b* is unambiguously true, hence we have

$$not\_b_{[0]}$$

However, the value of the local signal x cannot be derived because the C gate is in its state-holding mode of operation, where it outputs the value of its internal storage loop. Hence, we cannot constrain $x_{[0]}$ using a SAT clause, it is a free variable that will be treated as such by the SAT solver. Consequently, it is also not possible to derive an initial value for the output z, because it directly depends on the value of x. However, we still need to constrain $z_{[0]}$, because, although $x_{[0]}$ is not constrained, the value of $z_{[0]}$ is still related to it (e.g., $x_{[0]} = true$ and $z_{[0]} = false$ would not be a valid variable assignment, since we assumed all inputs are false). Hence, we add the following clause to the SAT problem (remember that because we are using Z3 not all our clauses will be specified as simple disjunctions):

$$x_{[0]} = z_{[0]}$$

We will now describe the process we use to ascertain the initial signal values. The model checker takes a (Python) dictionary with signal assignments, where initial values for inputs and internal signals can be specified. An initial value can either be a Boolean constant or `None`, indicating that no assumption about the initial value of the respective signal shall be made. Specifying an initial value for an internal signal is only possible for signals driven by state-holding rules. Doing so then overrides the initialization expression of the respective rule and replaces it with an initialization expression *only* containing the specified reset value, the initialization condition is removed. Note that overriding the initial values of state-holding rules can be used to bring the circuit into arbitrary initial states. Using this approach a circuit can even be brought into a state not reachable during normal operation. This could for example be useful to explore how a circuit behaves in a state that was caused by a fault.

Per default all inputs are assumed to be zero initially. However, this behavior can be deactivated, such that the model checker does not make assumptions about the initial values of inputs.

Now, to derive the initial signal assignment the model checker first makes a copy of the original PRS and replaces all initialized inputs with local signals driven by a constant wire rule. Furthermore, reset input signals are assumed to be active (reset inputs are identified using the role attribute). Hence, those inputs are likewise replaced by constant wire rules. In the next step, every rule with an initialization expression which just contains a value

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

(`init(true)` or `init(false)`) is also replaced with a constant wire. Then, the keep attribute is set to true for all locals (undeclared locals are explicitly declared for that purpose) and the PRS is passed through the optimizer. This yields a PRS with initial signal assignments (i.e., constant wire rules) for all inputs, locals and outputs, which can be expressed as constants. The remaining signals, which are not driven by constant wire rules, are either themselves indeterminable because they are driven by state-holding rules (like the C gate in the example above) or are driven by a combinational rule depended on an indeterminable signal (like the XOR gate). For the former case no clauses are generated, while for the latter case we simply generate a clause that states that the signal the rules writes to is equal to the combinational expression represented by the rule.

The model checker reports signals with indeterminable initial values, because such situations can potentially lead to unintended consequences or a malfunction of the circuit (see Section 5.4.5.1 for an example).

5.4.2.2 Rule Translation

Before the clauses for the rules of a PRS are generated the reset inputs are replaced with local signals driven by constants ensuring the reset is not active. The PRS is then optimized accordingly, which will remove all occurrences of the reset signals in the rules, which in turn removes all init expressions.

The simplest form of production rules are wire rules, which are expressed using simple equivalence clauses. This means that wires are always assumed to have zero delay, which is consistent with the SI delay model. Hence, a wire rule of the form `x := wire(y)` yields:

$$\bigwedge_{1 \leq n \leq b} x_{[n]} = y_{[n]}$$

For each non-wire rule r we introduce an auxiliary variable (vector) s_r, that determines whether the rule is evaluated in a particular step or not. A rule may only be activated if activating it would change the associated variable $x(r)$. If a rule is activated, $x(r)$ simply changes its value, otherwise it keeps its old value. This is expressed with the following two clauses.

$$\bigwedge_{1 \leq n \leq b} s_{r\,[n]} \rightarrow (x(r)_{[n]} = \neg x(r)_{[n-1]})$$

$$\bigwedge_{1 \leq n \leq b} \neg s_{r\,[n]} \rightarrow (x(r)_{[n]} = x(r)_{[n-1]})$$

Hence, the entirety of these auxiliary variables determine the sequence in which the rules of a PRS “fire”, which basically corresponds to different delays associated with each rule. Since we evaluate SI circuits this sequence must not have an impact on the validity and correctness of the result the circuit produces. It can, however, be relevant in the case of faults. Notice that in step $n = 0$ no rule evaluation takes place. Hence, we can set all $s_{r\,[0]}$ variables to false using unit clauses, effectively “deactivating” them.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

To specify the clauses that determine when a production rule may fire we distinguish state-holding and combinational rules. As we will see in the following, we only constrain when a rule is *not* allowed to fire, leaving the actual firing sequence (which as mentioned above actually corresponds to different/varying gate delays) to the underlying SAT solver.

Combinational Rules: The following constraint ensures that a combinational rule r may only fire in some step n, if the value of $up(r)_{[n-1]}$ is different from $x(r)_{[n-1]}$.

$$\bigwedge_{1 \le n \le b} (up(r)_{[n-1]} = x(r)_{[n-1]}) \rightarrow \neg s_{r[n]}$$

Consider the following example of a simple OR gate (`x := or_gate(y, z)`), which translates into the following clauses:

$$\bigwedge_{1 \le n \le b} ((y_{[n-1]} \vee z_{[n-1]}) = x_{[n-1]}) \rightarrow \neg s_{r[n]}$$

$$\bigwedge_{1 \le n \le b} s_{r[n]} \rightarrow (x_{[n]} = \neg x_{[n-1]})$$

$$\bigwedge_{1 \le n \le b} \neg s_{r[n]} \rightarrow (x_{[n]} = x_{[n-1]})$$

State-holding Rules: There are three distinct cases when a state-holding rule r is *not* allowed to fire.

- Neither $up(r)$ nor $down(r)$ evaluate to true:

$$\bigwedge_{1 \le n \le b} (\neg up(r)_{[n-1]} \wedge \neg down(r)_{[n-1]}) \rightarrow \neg s_{r[n]}$$

- When $up(r)$ evaluates to true and $x(r)$ is already asserted:

$$\bigwedge_{1 \le n \le b} (up(r)_{[n-1]} \wedge x(r)_{[n-1]}) \rightarrow \neg s_{r[n]}$$

- When $down(r)$ evaluates to true and $x(r)$ is already deasserted:

$$\bigwedge_{1 \le n \le b} (down(r)_{[n-1]} \wedge \neg x(r)_{[n-1]}) \rightarrow \neg s_{r[n]}$$

Consider the example of a C gate (`x := cgate(y, z)`), which translates to:

$$\bigwedge_{1 \le n \le b} (\neg(y_{[n-1]} \wedge z_{[n-1]}) \wedge (y_{[n-1]} \vee z_{[n-1]})) \rightarrow \neg s_{r[n]}$$

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

$$\bigwedge_{1\leq n\leq b} ((y_{[n-1]} \wedge z_{[n-1]}) \wedge x_{[n-1]}) \rightarrow \neg s_{r[n]}$$

$$\bigwedge_{1\leq n\leq b} ((\neg(y_{[n-1]} \vee z_{[n-1]}) \wedge \neg x_{[n-1]}) \rightarrow \neg s_{r[n]}$$

$$\bigwedge_{1\leq n\leq b} s_{r[n]} \rightarrow (x_{[n]} = \neg x_{[n-1]})$$

$$\bigwedge_{1\leq n\leq b} \neg s_{r[n]} \rightarrow (x_{[n]} = x_{[n-1]})$$

Notice that the last two lines are the same as for the OR gate example above.

5.4.2.3 I/O Channel Constraints

This section explains how the constraints describing the behavior of DI dual-rail input and output channels are formulated. For that purpose, consider Figure 5.11 that shows the different protocol phases of a dual-rail channel consisting of n dual-rail bits d_0,...,$d_{n\text{-}1}$.

- *Phase A* represents the spacer phase, where all data rails and the acknowledgment signal are zero. The channel waits for transitions on the data rails.
- In *Phase B* the data rails transition from the spacer to the data phase. It ends when all dual-rails bits are in the data phase
- *Phase C* represents the data phase, the channel now waits for an acknowledgment.
- In *Phase D* the acknowledgment is asserted and the channel waits for the data rails to return to the spacer again.
- The actual data rail transitions happen in *Phase E*. It ends with the last data rail transitioning to zero.
- Finally, the channel again enters the spacer phase in *Phase F* and waits for a falling edge on the acknowledgment signal.

We will use this figure and the described protocol phases to explain where a particular clause applies and which behavior it models.

Recall the notation for channels introduced above: The expressions $ack(C)$ and $D(C)$ refer to the acknowledgment wire and set of dual-rail data bits of some channel C, respectively. C^I (C^O) refers to the set of input (output) channels of a PRS.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

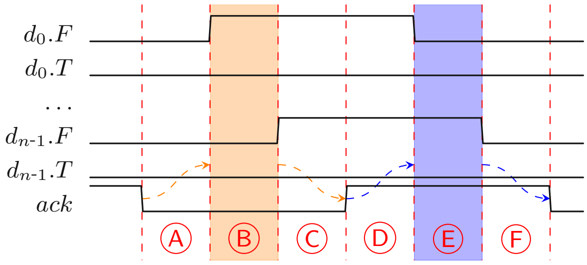


Figure 5.11: DI dual-rail channel protocol phases

Input Channel Constraints For input channels $C \in C^I$ we distinguish between coding and protocol constraints. The coding constraints specify *how* inputs are allowed to change, while protocol constraints state *when* the inputs are allowed to change.

Regarding the coding constraints, since we only deal with dual-rail inputs it suffices to demand that the true and false rails cannot be asserted at the same time (i.e., at the same step n).

$$\bigwedge_{d \in D(C)} \bigwedge_{0 \le n \le b} \neg(d.T_{[n]} \wedge d.F_{[n]})$$

For the protocol constraints we distinguish between the data and the spacer phases.

- Data Phase: If $ack(C)$ is deasserted (Phases A, B, C) and a rail is one, it must stay one until $ack(C)$ is asserted, i.e., in the data phase only rising transitions are allowed.

  $$\bigwedge_{d \in D(C)} \bigwedge_{r \in d} \bigwedge_{1 \le n \le b} (d.r_{[n-1]} \wedge \neg ack(C)_{[n-1]}) \rightarrow d.r_{[n]}$$

  This means that rails that are zero are allowed to switch to high during the phases A and B, and if they do, they have to stay one. Note that in phase C rising transitions are prevented by the coding constraints.

- Null Phase: If $ack(C)$ is asserted (Phases D, E, F) and a rail is zero, it must stay zero until $ack(C)$ is deasserted, i.e., in the null phase only falling transitions are allowed.

  $$\bigwedge_{d \in D(C)} \bigwedge_{r \in d} \bigwedge_{1 \le n \le b} (\neg d.r_{[n-1]} \wedge ack(C)_{[n-1]}) \rightarrow \neg d.r_{[n]}$$

  This constraint has two functions:

  - If a rail was not set during the data phase, it has to remain zero
  - If a rail was set during the data phase, it may switch to zero, but then it has to remain zero

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

To support other 4-phase DI codes (see Section 4.1), only the coding constraints need to be adjusted. For example, to support a 1-of-4 code the coding constraint must state that it must never be the case that two or more rails are asserted at any given time.

Output Channel Constraints For output channels $C \in C^O$ we have to model the behavior of the $ack(C)$ signal based on the value of $D(C)$. For that purpose, for each channel, two auxiliary variables oc_C and oe_C are required, that essentially model the behavior of a CD. The *output complete* indicator signal oc_C is given by:

$$oc_{C[n]} = \bigwedge_{d \in D(C)} d.T_{[n]} \vee d.F_{[n]}$$

The *output empty* indicator signal oe is given by:

$$oe_{C[n]} = \neg \bigvee_{d \in D(C)} d.T_{[n]} \vee d.F_{[n]}$$

- If $ack(C)$ is asserted and the output data is not empty, (Phases D, E), it must stay asserted.
$$\bigwedge_{1 \leq n \leq b} (ack(C)_{[n-1]} \wedge \neg oe_{C[n-1]}) \rightarrow ack(C)_{[n]}$$
- If $ack(C)$ is deasserted and the output data is not complete (Phases A, B) it must stay deasserted.
$$\bigwedge_{1 \leq n \leq b} (\neg ack(C)_{[n-1]} \wedge \neg oc_{C[n-1]}) \rightarrow \neg ack(C)_{[n]}$$

Hence, we see that $ack(C)$ may only change in Phases C (rising transition) and F (falling transition), which effectively ends these phases.

5.4.3 Checkable Properties

5.4.3.1 Static Assertions

As described in Section 5.1.4 the `pypr` PRS language also supports a constraints section where assumptions and assertions about the circuit can be formulated. Assumptions are useful to constrain the inputs of a circuit (e.g., forbid certain input values) or to prevent the model checker from exploring certain circuit states. Assertions on the other hand make statements about valid states of a circuit – a violation of an assertion means that a circuit has entered an illegal state, which should not have been possible to enter in the first place.

Let E (U) denote the set of assert (assume) statements in the constraints section. Each assertion $e \in E$ and assumption $u \in U$ is represented by a Boolean expression.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

The following clause states that there is at least one step n where at least one assertion is violated. Hence, if the model checker cannot find a model we can conclude that all assertions hold.

$$\bigvee_{e \in E} \bigvee_{0 \leq n \leq b} \neg e_{[n]}$$

Assumptions can also easily be incorporated into the SAT problem. We simply state that every assumption holds in every step:

$$\bigwedge_{u \in U} \bigwedge_{0 \leq n \leq b} u_{[n]}$$

5.4.3.2 Liveness/Deadlocks

A deadlock is defined as a state where no rule of a PRS can fire and no input signal can transition because the I/O channel constraints forbid it. Since no signal changes are possible, once entered, this state can never be left. Let $\mathit{deadlock}(n)$ denote the clause set that encodes this condition for the step n, i.e., a PRS is deadlocked in step n when $\mathit{deadlock}(n)$ evaluates to true. From the definition of the deadlock state it is clear that

$$\mathit{deadlock}(n) \rightarrow \mathit{deadlock}(n+1)$$

must hold. Hence, to check for a deadlock, we simply add the clause set created by $\mathit{deadlock}(b)$ to our SAT problem. If the solver is unable to satisfy the presented problem, it can therefore be concluded that the circuit does not deadlock (up to bound b). If a model is found, we can immediately extract an example (i.e., a rule firing sequence) that shows how the deadlock was caused.

Now what remains is to define the deadlock condition itself:

$$\mathit{deadlock}(n) = \neg \left(\left(\bigvee_{r \in R} \mathit{live}_r(r,n) \right) \vee \left(\bigvee_{C \in C^I} \mathit{live}_{ic}(C,n) \right) \vee \left(\bigvee_{C \in C^O} \mathit{live}_{oc}(C,n) \right) \right)$$

This formula can be subdivided into three parts, each expressing the liveness constraints for production rules, input channels and output channels. To get to deadlock the disjunction of all these constraints must be false.

A rule is live if it could fire, which is expressed as:

$$\mathit{live}_r(r,n) = \begin{cases} (\mathit{up}(r)_{[n]} \wedge \neg x(r)_{[n]}) \vee (\mathit{down}(r)_{[n]} \wedge x(r)_{[n]}) & \text{if } r \text{ is state-holding} \\ \mathit{up}(r)_{[n]} \neq x(r)_{[n]} & \text{if } r \text{ is combinational} \\ \mathit{false} & \text{if } r \text{ is a wire} \end{cases}$$

For dual-rail input channels we have liveness if (i) the acknowledgment signal is deasserted and there is at least one dual-rail signal pair that is still in the spacer phase or if (ii) the

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

acknowledgment signal is asserted and there is at least one dual-rail signal pair still in the data phase (i.e., with one asserted rail).

$$live_{ic}(C, n) = \left(\neg ack(C)_{[n]} \wedge \bigvee_{d \in D(C)} \neg d.T_{[n]} \wedge \neg d.F_{[n]}\right) \vee \left(ack(C)_{[n]} \wedge \bigvee_{d \in D(C)} d.T_{[n]} \vee d.F_{[n]}\right)$$

For output channels the situation is a little bit simpler, as it is only required to check the value of the auxiliary variables $oc_{C[n]}$ and $oe_{C[n]}$ in comparison to $ack(C)_{[n]}$ to determine if a transition on the acknowledgment is expected.

$$live_{oc}(C, n) = (oc_{C[n]} \wedge \neg ack(C)_{[n]}) \vee (oe_{C[n]} \wedge ack(C)_{[n]})$$

5.4.3.3 Gate Orphans

Another very interesting property to check in an SI/QDI circuit is whether it contains gate orphans (see Section 2.2) as this would invalidate the SI/QDI property in the first place. In our SAT problem gate orphans are discoverd by searhing for rules that could have fired in step $n-1$ (but didn't) and can no longer fire in step n.

Hence, for each combinational rule the following clause is added to the problem, which exactly captures this condition.

$$\bigvee_{1 \leq n \leq b} up(r)_{[n-1]} \neq x(r)_{[n-1]} \wedge up(r)_{[n]} = x(r)_{[n]} \wedge x(r)_{[n]} = x(r)_{[n-1]}$$

For state-holding rules the situation is similar, but here a distinction has to be made between up and down transitions. If the up condition is fulfilled in step $n-1$ and no longer fulfilled in step n, then we have a gate orphan if the output value of the rule $x(r)$ was zero in both steps. The down condition is handled analogously.

$$\bigvee_{1 \leq n \leq b} \left(up(r)_{[n-1]} \wedge \neg up(r)_{[n]} \wedge \neg x(r)_{[n-1]} \wedge \neg x(r)_{[n]}\right) \vee \left(down(r)_{[n-1]} \wedge \neg down(r)_{[n]} \wedge x(r)_{[n-1]} \wedge x(r)_{[n]}\right)$$

If the solver finds a model for a SAT problem with these clauses the circuit contains at least one gate orphan.

5.4.4 Fault Injection

The model checker can also be used to investigate the effects of transient faults on a circuit. A fault in the context of the model checking problem is defined as the change of

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

a signal without the associated rule firing. We distinguish between SETs and SEUs. The former can be injected into both state-holding and combinational rules and only changes the value of the affected signal for one step. SEUs on the other hand can only be injected into state-holding rules. If neither the up nor the down condition is fulfilled the fault makes a lasting change to the value of the associated signal (i.e., until it gets overridden).

To mark a rule as the target of a fault injection its `fault_victim` attribute must be set to true. This can be set for multiple rules at once. The `fault_type` attribute is used to choose between SETs and SEUs.

When the SAT problem is generated and a rule r is encountered that should be the target of a fault injection, a special variable f is introduced. This variable models the actual fault-injection event and is constrained to be asserted for exactly one step n using the clause $once(f)$[12].

$$once(v) = \left(\bigvee_{1 \le n \le b} v_{[n]} \right) \wedge \left(\bigwedge_{n \le b} \left(v_{[n]} \rightarrow \neg \bigvee_{j \le b, n \ne j} v_{[j]} \right) \right)$$

To model the effects of a fault only minor modifications to the rule clauses introduced in Section 5.4.2.2 are necessary. For SETs the same set of clauses is generated but a newly introduced variable x'_r is used as the target instead of $x(r)$. Then, an additional clause is added, which basically states that the actual signal $x(r)_{[n]}$ is always equal to $x'_{[n]}$ except for the fault-injection event:

$$\bigvee_{1 \le n \le b} x(r)_{[n]} = x'_{r[n]} \oplus f_{[n]}$$

To model SEUs the clauses that determine what happens when the auxiliary variable s_r is asserted or deasserted have to be modified. In the case of a fault the variable f is used to force the variable $x(r)$ to flip its state, regardless of the value of s_r.

$$\bigwedge_{1 \le n \le b} (s_{r[n]} \vee f_{[n]}) \rightarrow (x(r)_{[n]} = \neg x(r)_{[n-1]})$$

$$\bigwedge_{1 \le n \le b} (\neg s_{r[n]} \wedge \neg f_{[n]}) \rightarrow (x(r)_{[n]} = x(r)_{[n-1]})$$

5.4.5 Model Checking Examples

This section demonstrates two possible use cases for the model checker.

---

[12] Notice that $f_{[0]}$ is always false, because we don't allow faults to be injected in step 0.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

5.4.5.1 Checking Circuit Initialization

Consider the circuit in Figure 5.12, showing a 3-stage dual-rail WCHB pipeline ring, that constantly outputs a stream of tokens on its output channel $C = (\{x\}, ack)$. Since the data rails are crossed-over in the feedback connection in the pipeline (i.e., the two top-most signal lines in the figure) the value of the single-bit output token toggles between true and false. The PRS encoding this circuit is shown in Listing 5.12. Note that the listing uses vectors where the labels in the figure use subscript indices.

```
prs atg is
inputs
  ack : Bit attributes(channel:=C, role:=ack, channel_type:=DIDR);
outputs
  x : DRBit attributes(channel:=C, role:=data);
locals
  d : DRBit(2); c : Bit(3);
  ack_n : Bit; en : Bit;
begin
  #1. stage (data rails are crossed-over)
  d(0).T := cgate(x.F, c(1)) init(true);
  d(0).F := cgate(x.T, c(1)) init(false);
  c(0) := nor_gate(d(0).T, d(0).F);
  #2. stage
  d(1).T := cgate(d(0).T, c(2)) init(false);
  d(1).F := cgate(d(0).F, c(2)) init(false);
  c(1) := nor_gate(d(1).T, d(1).F);
  #3. stage (output)
  x.T := cgate(d(1).T, en) init(false);
  x.F := cgate(d(1).F, en) init(false);
  c(2) := nor_gate(x.T, x.F);
  #join input acknowledgment
  ack_n := inv(ack);
  en := cgate(ack_n, c(0));
constraints
  assume(not (x.T and x.F));
end prs;
```

Listing 5.12: Alternating token generator PRS

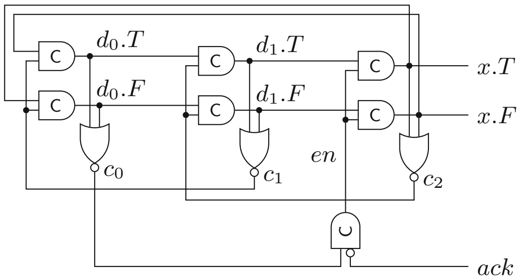


Figure 5.12: Alternating token generator circuit

As can be seen in the listing all but two C gates are initialized to false (`init(false)`). The C gate driving the signal $d_0.T$ is initialized to true, as the buffers comprising this C gate initially starts with a data token. The other stages are initialized with the spacer value.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

The C gate joining the input acknowledgment *ack* of the output channel C with the acknowledgment of the left-most pipeline stage is intentionally left uninitialized. We will now use the model checker to show that this can lead to a problem in the form of a deadlock. Listing 5.13 shows how to run the model checker to find this particular issue from within the `pypr` framework. First, the PRS is loaded and a bounded model checking problem is created with a bound of ten. We then assert that there exists a deadlock in the circuit and run the solver. Note the `assume` statement in the constraints section of the PRS. This assumption is added to exclude models that would produce an illegal output (i.e., $x.T = x.F = 1$), which is also a possible scenario in this circuit. Every model found by the model checker corresponds to a sequence of events that brings the circuit into a deadlocked state. To examine the model the `Plot` function allows to create a timing diagram, which is shown in Figure 5.13. It can be seen that if the initial value of the C gate driving the *en* signal is one (which is a valid initial value since it has not been set explicitly), there exists a trace that leads to a deadlock. If this C gate is initialized to zero, the circuit is deadlock-free. This example shows how the model checker can be used to find potential initialization problems in SI circuits.

```python
prs_lib = ParsePRSFile("atg.prs")
problem = BMCProblem(prs_lib["atg"], 10)
problem.ConstrainIOChannels()
problem.ImplementPRSConstraints()
AssertDeadlock(problem)
model = problem.Check()
model.Plot("counter_example.png")
```

Listing 5.13: Example Python code demonstrating how to run the model checker

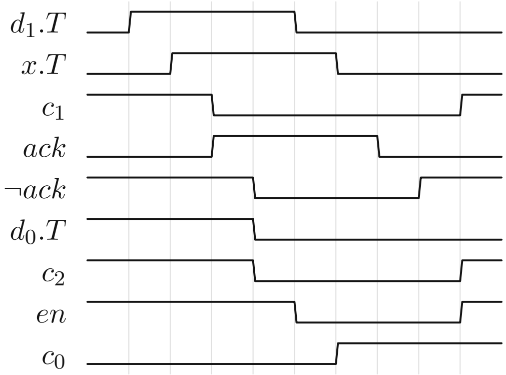


Figure 5.13: Timing diagram representing the model found by the model checker

5.4.5.2 Generating Fault Scenarios

The model checker can also be used to investigate the potential effects of faults on a system. Consider the 3-stage 2-bit WCHB pipeline shown in Figure 5.14 and its corresponding PRS in Listing 5.14.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

```
prs pl is
inputs
  a : DRBit(2) attributes(channel:=Cin, role:=data);
  ack_in : Bit attributes(channel:=Cout, role:=ack, channel_type:=DIDR);
outputs
  ack_out : Bit attributes(channel:=Cin, role:=ack, channel_type:=DIDR);
  d : DRBit(2) attributes(channel:=Cout, role:=data);
locals
  b : DRBit(2); c : DRBit(2);
begin
  #stage 1
  b(0).T := cgate(a(0).T, b_en) init(0);
  b(0).F := cgate(a(0).F, b_en) init(0);
  b(1).T := cgate(a(1).T, b_en) init(0);
  b(1).F := cgate(a(1).F, b_en) init(0);
  b0_done_n := nor_gate(b(0).F, b(0).T);
  b1_done_n := nor_gate(b(1).F, b(1).T);
  b_done_n := cgate(b0_done_n, b1_done_n);
  ack_out := inv(b_done_n);
  #stage 2
  c(0).T := cgate(b(0).T, c_en) init(0)
    attributes(fault_victim:=true, fault_type:=SET);
  c(0).F := cgate(b(0).F, c_en) init(0);
  c(1).T := cgate(b(1).T, c_en) init(0);
  c(1).F := cgate(b(1).F, c_en) init(0);
  c0_done_n := nor_gate(c(0).F, c(0).T);
  c1_done_n := nor_gate(c(1).F, c(1).T);
  b_en := cgate(c0_done_n, c1_done_n);
  #stage 3
  d(0).T := cgate(c(0).T, d_en) init(0);
  d(0).F := cgate(c(0).F, d_en) init(0);
  d(1).T := cgate(c(1).T, d_en) init(0);
  d(1).F := cgate(c(1).F, d_en) init(0);
  d0_done_n := nor_gate(d(0).F, d(0).T);
  d1_done_n := nor_gate(d(1).F, d(1).T);
  c_en := cgate(d0_done_n, d1_done_n);
  d_en := inv(ack_in);
end prs;
```

Listing 5.14: 3-stage 2-bit WCHB pipeline PRS with fault injector

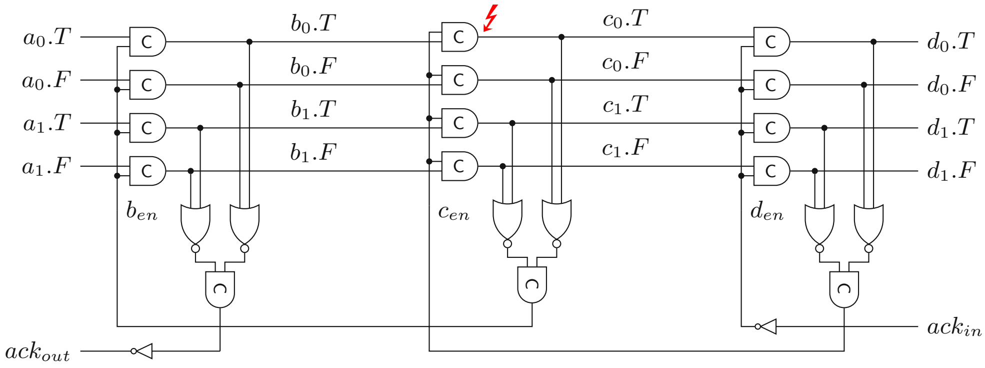


Figure 5.14: 3-stage 2-bit WCHB pipeline circuit

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Using `pypr` this circuit can be generated with a script similar to the one shown in Listing 5.5. In Line 22 the rule producing the signal `c(0).T` ($c_0.T$ in Figure 5.14), which represents one of the buffer C gates in the second WCHB, is marked as a fault victim.

Now the model checker can be run in order to, e.g., find a scenario where a fault on this signal leads to a deadlock of the circuit, using a similar script as the one shown in Listing 5.13. Figure 5.15 shows the resulting traces extracted form the model found by the model checker (some input and internal signals have not been included in the figure in order not to unnecessarily clutter the timing diagram). It can be seen that the input transition at $a_1.T$ propagates through the pipeline and appears at the output $d_1.T$. In the second time step an SET is injected into $c_0.T$ which in turn propagates to the output $d_0.T$. Simultaneously this fault also triggers the completion detector in the second WCHB leading to the deassertion of b_{en}. This event is what causes the deadlock, since even if an input transition on $a_0.T$ or $a_0.F$ arrives, it can no longer pass though the buffer C gates of the first WCHB.

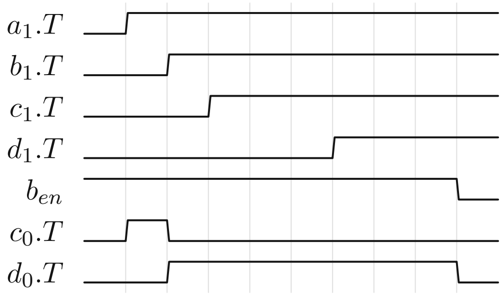


Figure 5.15: Timing diagram representing the model found by the model checker

Another possible fault scenario that can be visualized by the model checker is an illegal output state, where both rails of a single dual-rail bit are asserted. For that purpose, the assertion `assert(d(0).T and d(0).F)` must be added to the constraints section of the PRS in Listing 5.14.

The presented scenarios might seem trivial, but this approach can, e.g., be used to find edge cases for fault-mitigation strategies or to show that a particular signal in a circuit is not susceptible to faults.

5.4.6 Performance and Limitations

To demonstrate the computational overhead of this model checking approach we apply it to two simple QDI/SI dual-rail circuits – a 4-bit adder and a 4x4-bit unsigned multiplier circuit. Both circuits have one input and one output channel and use DIMS for their combinational logic. The multiplier is implemented using a 4-stage WCHB pipeline, and comprises 440 rules of which 296 represent C gates. The combinational logic of the adder (122 rules, 77 C gates) is also placed between two WCHB stages.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

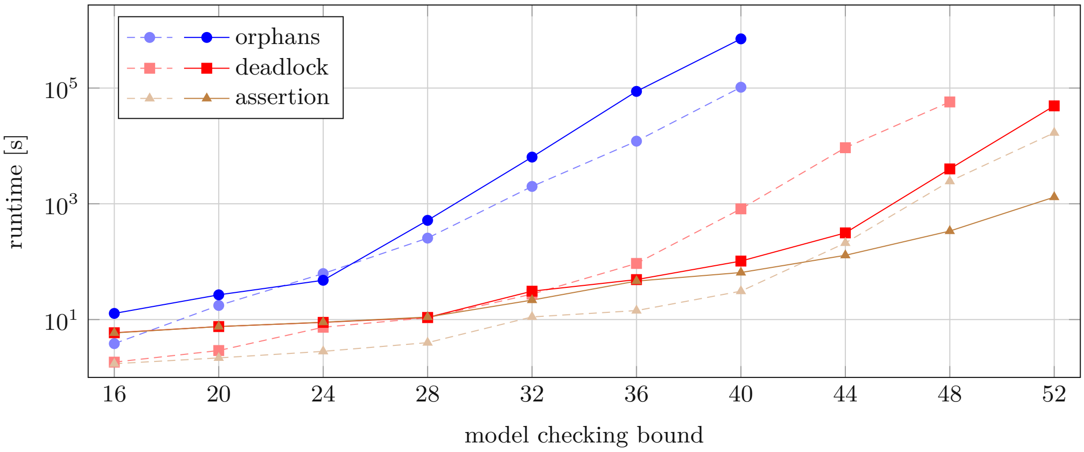


Figure 5.16: Model checker runtime for the multiplier (solid lines) and adder (dashed lines) circuits

We performed three different model checking tasks, which include a check for a deadlock possibility, gate orphans and the verification of a simple static assertion[13]. Since there are no issues with the circuits, no models can be found for any of these tests. Figure 5.16 shows the runtime (in seconds) for these tests for different bounds b as executed on an Intel Core i7-10700K CPU at 3.80 GHz with 32 GB of dual-channel DDR4 memory at 3200 GHz. It is obvious that checking for orphans is the hardest problem, while verifying a static assertion is much less expensive, and can hence be executed for much larger bounds in reasonable time. The results also show that the computational overhead is not only dependent on the size of the circuits.

For our future research, we want to address the boundedness of the model checker, and extend it to be able to verify SI circuits to an unbounded depth. Moreover, we plan to further explore applications of model checking for the analysis of SI/QDI circuits with respect to their behavior under transient faults.

5.5 Conclusion

This chapter introduced our Python-based design and development tool for asynchronous circuits, called `pypr`. To summarize its various features, Figure 5.17 shows an example tool-flow for a simple linear (QDI) pipeline. The combinational logic of this pipeline is described using Verilog code, which is synthesized to PRSs using Yosys and our integrated dual-rail expansion algorithm. The resulting PRSs are collected in a `PRSLib` object. To generate the pipeline buffers, the PRS generator function `DRBuffer` is used. Those sub-components are then connected using the `DataFlowGraph` class. The resulting PRS is then flattened and optimized. In this form, it can be passed to our Z3-based model

---

[13] The result of the multiplier (adder) circuit cannot be larger than 15^2 $(2 * 15)$.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub

checker to verify certain desired properties. Using the code generator, the PRS is then converted to a VHDL design, which can be simulated with a standard digital simulator (like, e.g., GHDL or Questa) For that purpose, also a testbench is required, which is automatically generated from a compact, yet powerful YAML-based specification.

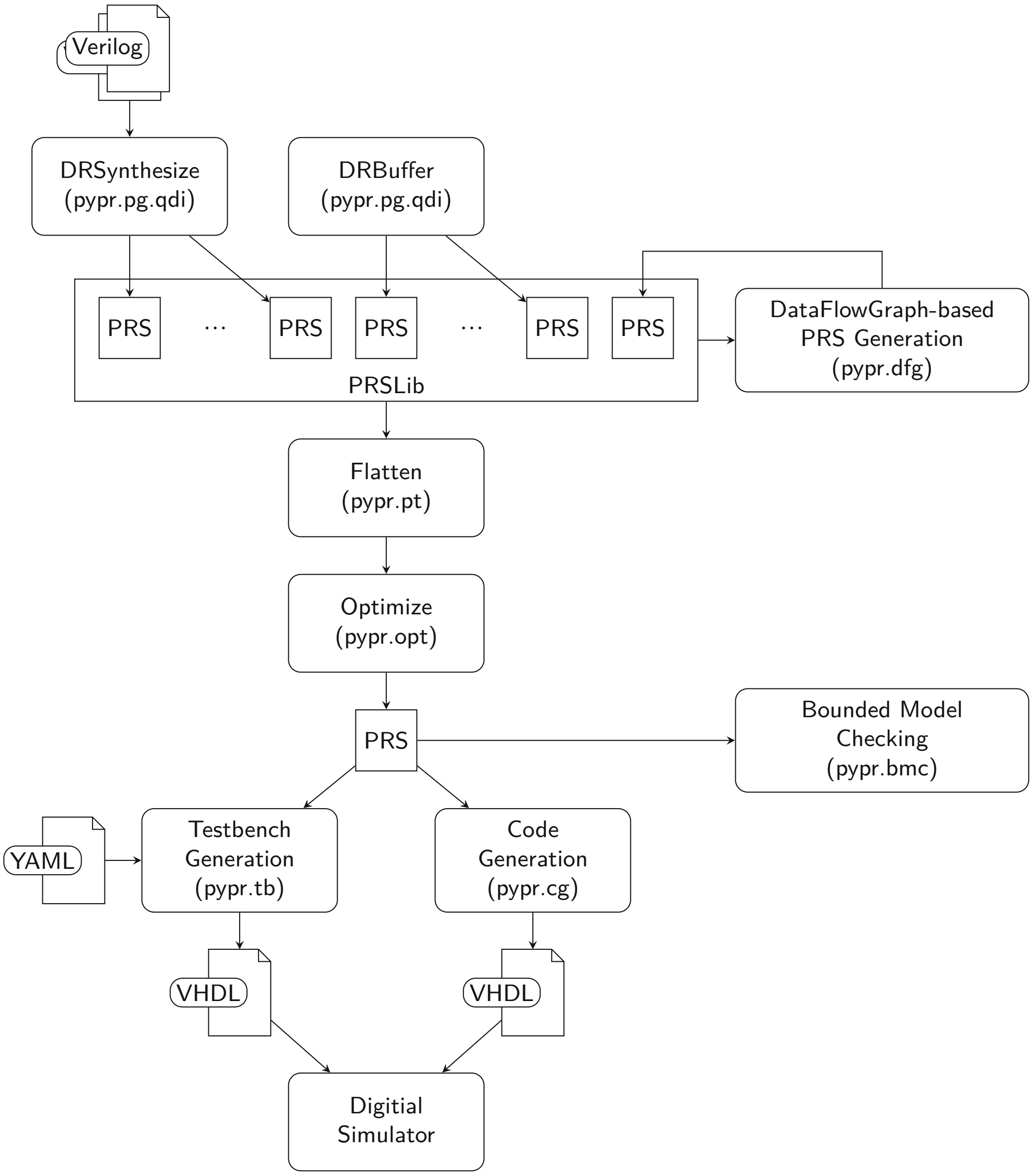


Figure 5.17: `pypr` tool-flow example

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 6

Fault-Tolerance in QDI Circuits

The overall goal of the research presented in this chapter is to perform a meaningful comparison of the resilience of different QDI pipeline and design styles to transient faults and to analyze their respective strengths and weaknesses. To this end we provide an elaborate fault-injection simulation framework, that allows insightful conclusions regarding our research questions. As part of a larger project, this work also represents the foundation for further research into this area.

As their name already suggests transient faults only affect a circuit for a certain amount of time, without actually causing physical harm to it. They may, however, lead to a change of the contents of some storage elements (e.g., latches or flip-flops), which is then usually referred to as a soft error. This is in contrast to permanent faults that always entail some form of physical damage to a circuit (e.g., a permanently open connection between two circuit nodes). Transient faults can have a variety of causes, including but not limited to cross-talk, electrostatic discharge or radiation effects. Specifically the sensitivity of semiconductors to (cosmic) radiation is a major point of concern, which continuously increases in severity due to the ongoing technology scaling [DDC05, Bau05]. We won't to go into detail on the exact causes and composition of cosmic radiation and how it manages to induce state changes in a circuit, since these topics are not particularly relevant for this work [AM15]. For our discussion here it suffices to say that an energetic particle strike (e.g., by a neutron) can cause a (short) current/voltage pulse on some node of a circuit, which is referred to as SET. Whether an SET actually has an impact on the behavior of the victim circuit depends on various factors, including the location of the hit and the current state of the circuit. In particular SETs may be filtered by logical, temporal or electrical masking effects as they spread through a circuit.

For our fault-tolerance research we focus on the investigation of the *digital effects* of SETs on QDI/SI circuits and ways to harden them against such effects. In the context of this work, faults are considered to be binary events, that (instantly) appear, affect some part of the circuit for the fault duration and then disappear again. The immediate effect of a fault is a change in the binary state of the affected node. We exclude analog effects from our analysis. Hence, electrical masking is not covered.

To investigate fault effects we perform extensive fault-injection simulations. The circuits we subject to these experiments are modeled and created using the `pypr` framework presented in Chapter 5. We consider every production rule as a single possible fault-injection target. This means that an SET can change the output of exactly one production rule, which essentially represents the output of a CMOS gate. This change then propagates to *all* gates that are connected to the respective signal. Although certain things like stack-sharing of gates cannot be modeled this way, our circuit model is a good compromise between a detailed and realistic circuit representation and a reasonable computational overhead required for the actual simulation, where we want to use a digital circuit simulator.

The chapter is structured as follows: First, Section 6.1 presents an overview of the relevant related work. Then, Section 6.2 examines possible effects transient faults can have on QDI circuits and identifies certain weak points of the classic WCHB. With this analysis in mind Section 6.3 presents some relatively low-overhead fault mitigation and hardening techniques from literature that can be used as drop-in replacements for the WCHB. To evaluate those buffer styles Section 6.4 performs a case study using fault-injection simulations on a simple pipeline without any processing logic. Building upon the knowledge gathered form this experiment we then propose our own improved QDI buffer designs in Section 6.5. A second case study on more complex target circuits then shows how the different buffer styles perform in a more realistic real-world scenario and allows for better insights in the fault resilience of different designs. Finally, Section 6.7 summarizes our findings and concludes the chapter.

6.1 Related Work

As already discussed in Section 2.6, a core difference between BD and QDI circuits lies in the separation of control logic and the data path. While they are considerably intertwined for QDI circuits there is a more or less strict separation in BD circuits. Nevertheless, the timing/delay model that is applied for the control logic of BD circuits is often similarly strict to the QDI model, i.e., the control logic is often QDI itself or at least SI.

One very important characteristic of QDI or SI circuits is glitch-free operation. Every transition that happens inside such a circuit, or at its primary inputs and outputs has a certain meaning and must have a direct causal dependence on some other input or output transition. Hence, although SI and QDI circuits are by their nature very robust against delay variations, disturbances in the value domain caused by transient faults can have severe consequences. A single transition at the wrong time can bring a circuit into an undefined or illegal state and cause a deadlock or some other undesired behavior. The simplest form of faulty behavior is data corruption (similar to what can also be observed in synchronous circuits). However, with asynchronous circuits it is also possible that complete handshaking cycles get extinguished or wrongly created (token creation and

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

deletion) [LM04, MRL07]. For these cases there is no direct equivalent in the synchronous world. Therefore, it is vital to take these special characteristics of asynchronous and specifically QDI circuits into account, when investigating and analyzing their behavior under faults and when it comes to implementing appropriate countermeasures.

However, this sensitivity to faults in the value domain also has advantages. It is well established that under (certain) permanent faults QDI circuits eventually deadlock [DGY95, YCCP04]. Intuitively this is because of the fact that a certain transition can no longer occur. Since every transition carries some meaning the system can no longer progress, if it waits for it. Of course permanent faults can also be dormant, were they don't have an effect unless invoked. This means that QDI circuits have inherent self-checking abilities, which can, for example, be used for circuit testing. A test run that triggers every possible transition in a QDI circuit and does so without deadlocking, proves that the circuit does not contain any stuck-at faults. Some fault tolerance techniques for QDI circuits try to leverage this fail-stop behavior also for transient faults [PM05]. One of the techniques proposed in this chapter also falls into this category.

Other methods for tolerating transient faults in QDI pipelines are presented in [JM05, MRL05, FS09]. Here the circuits are basically duplicated and the resulting replicas are carefully interlocked, such that they can only operate in a lock-step way. If one replica is affected by a fault, the other one still works correctly and the whole system can only proceed if the fault disappears. A concrete implementation example where this technique was used is a radiation-hard microcontroller presented in [KMM15].

Similar techniques, although for BD circuits, have been proposed in [VM02, MR07, KHS$^+$20, KK20]. Here full or partial duplication of the target circuit is used to detect and (for the latter two publications) also correct faults caused by SETs.

In [GYB07] a special buffer style for QDI circuits is proposed that relies on checking for the illegal dual-rail state (i.e., both rails set simultaneously) to ensure that faulty data is not captured into the buffer. Regarding the hardening of pipeline buffers [BS09] presents a variety of different techniques based around the WCHB and discusses their strengths and weaknesses. Some of these approaches will be investigated thoroughly throughout this chapter.

To accurately understand and assess the behavior of QDI circuits under the influence of (transient) faults, some form of fault model is required. Hence, it must be defined what a fault is capable of doing to a system. This of course depends on the chosen level of abstraction used to represent the circuit, which can range from transistor or gate-level representations to more high-level token-based circuit modeling techniques. This means that a fault can, for example, cause a transistor to become conductive (with various possible consequences on the higher abstraction layers), flip the state of some storage element or cause the corruption, deletion or insertion of tokens.

In an early approach [Dil88] built an automated verifier for SI circuits based on trace theory, which is a natural means for describing the sequences of transitions relevant for the operation in asynchronous circuits. Later, [Yak93] also used trace theory (among

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

others) for modeling the correct (and incorrect) behavior of asynchronous circuits, more specifically interfaces.

Another approach to model the effects faults can have on a circuit are STGs, which have, e.g., been used in [BS09].

In [MRL04] a simulation-based approach is presented, which acquires statistics about the fault sensitivity of individual C gates in QDI circuits. A C gate is considered sensitive if only one input needs to change for the output of the C gate to change. A similar approach that, instead of C gates, also considers threshold gates is proposed in [KZYD10]. Another simulation-based fault-injection technique is presented in [BMS+09]. All these approaches show that, similar to synchronous circuits, there is a certain degree of fault masking. However, none of the papers consider a wide range of different QDI design styles.

In [LM04] LaFrieda and Manohar show how permanent and transient faults affect production rules.

On a higher abstraction level [MRL07] analyzes how faults affect the token flow in QDI systems. For this they use a set of token and fault rules to model the circuit operation.

6.2 Fault Effects and Sensitivity Windows

Figure 6.1 shows a timing diagram of a multi-bit WCHB over a single handshake cycle. The shown data signals correspond to the data outputs of the buffer, i.e., the outputs of the C gates. The dual-rail bits d_0 and d_n symbolize the pair of data signals with the largest skew between them (windows B-C and E-F).

As soon as the input acknowledgment ack_{in} goes low (A) a WCHB waits for the data phase. Thus, all its C gates are armed for rising transitions. The C gates are only disabled when the next stage acknowledges the received data (D), which can only happen after the slowest dual-rail signal pair transitioned to the data phase. This leaves quite a large time window (A-D) where the buffer is susceptible to faulty rising input transitions. We say that the buffer is now in a (fault-)accumulating state, since all input transitions (faulty and valid ones alike) will be captured into the C gates. This is in stark contrast to, e.g., synchronous designs, where there is only a single point in time (i.e., the clock edge) where data is captured by the storage elements.

Considering a WCHB pipeline we can see that the sensitive time window for each stage depends on various factors. The time between when the first buffer of the pipeline is armed for rising input transitions (A) and a valid data token actually starts to arrive (B) depends on the overall speed the circuit is operated with, i.e., the rate with which input tokens arrive. This factor alongside the general timing of each stage and the overall delay balance of the pipeline contribute to the individual duration of the window (A-B) for each stage. A token arriving at a stage has to pass through its buffer C gates and the combinational logic on the data path to the next stage, which also adds delay. The

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

skew between the individual data rails (B-C) also affects this delay. In the worst case the combinational logic cannot start processing until all dual-rail bits arrived. When the token arrives at the next WCHB it has to pass through its buffer C gates and CD. However, if this buffer is currently not yet ready to receive a new token, additional wait time is added (this again depends on the overall load and delay balance of the pipeline). Finally, there is the delay of the acknowledgment path back to the source stage.

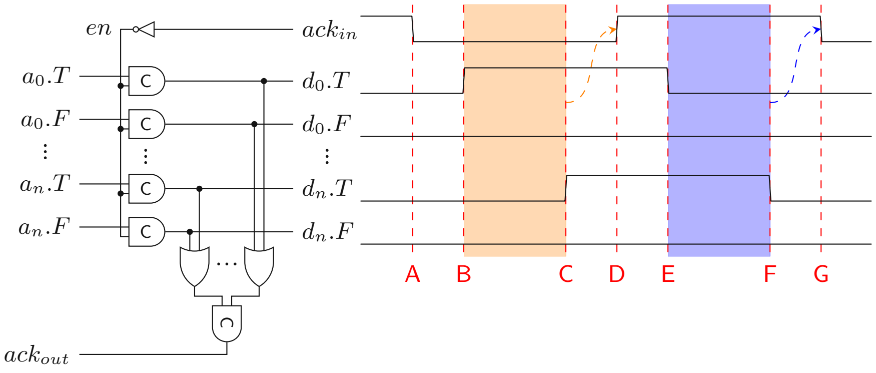


Figure 6.1: WCHB time windows

Depending on when exactly a fault strikes a data wire during the data phase and erroneously sets a C gate, one can observe vastly different effects. If a fault strikes a data wire that should transition anyway, in the best case only the point in time when this transition happens is shifted, which is completely tolerable in the context of QDI circuits. However, given sufficient skew between the data rails, in the worst case such a fault can deadlock the whole circuit. Striking a data rail that should remain zero, a fault can (besides the deadlock) cause an invalid output pattern for a dual-rail bit (i.e., both rails set to one) or a valid but incorrect data token, depending on whether the valid transitions on the respective other dual-rail signal arrives before the buffer is closed. These scenarios have been explored in Section 5.4.5.2 using the BMC feature of `pypr`.

After ack_{in} has been asserted (D) to acknowledge the received data the C gates in the buffer are again armed for falling input transitions. Hence, there is a time window (D-G) where the buffer is susceptible to faulty (falling) input transitions. However, since the null phase does not carry any actual data, faults don't affect the transmitted information but can again lead to deadlocks.

Since the input acknowledgment signal ack_{in} basically represents the enable signal for all the C gates in a buffer, faults affecting this signal can have severe consequences on a circuit[1]. These range from deadlocks to lost or additional data tokes or illegal output patterns. The ack_{in} (or en) signal is vulnerable during phases where the inputs to the C gates differ in their logical value (i.e., the C gate is in state-holding mode). In such a situation a single input change can flip the value of the C gate. As Figure 6.1 shows, at time (C) the output of the buffer is complete. Thus, eventually the output

---

[1] In synchronous systems this is in a way comparable to a clock glitch.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

acknowledgment ack_{out} would be asserted, which in turn leads to the deassertion of all input data rails. However, the output of the buffer must not enter the null phase until ack_{in} is asserted, which means that there is a sensitive window for the acknowledgment signal between (C) and (D). A similar situation arises during the null phase between the points (F) and (G).

We see that although in general delays are not relevant for the correct behavior of QDI circuits, the exact fault behavior of such a circuit is highly dependent on the input skew, the involved gate and wire delays, the speed of operation and the actual data that is being processed. This means that the sensitivity of a buffer cannot be evaluated completely statically but must incorporate environmental conditions as well as concrete circuit details.

6.3 Fault Mitigation Strategies

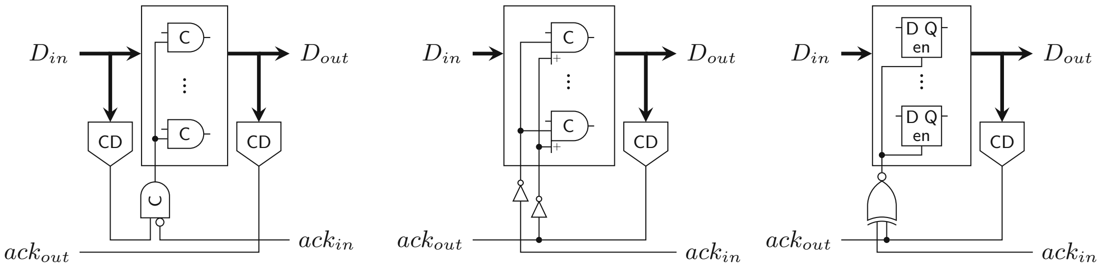
To mitigate the shortcomings of the classic WCHB with regard to its fault sensitivity, various modifications and improvements have been proposed. A selection of those strategies will be presented in this section. Most WCHB modifications aim at shortening the time window in which the buffer stores input transitions, in one way or another.

One possibility to achieve this, is to use two CDs for every buffer, one for the input channel and another one for the output channel (see Figure 6.2a). Using this configuration, the C gates of the buffer are only armed when there is actually data at the input. Consequently, we refer to this buffer as the Dual-CD WCHB. This idea is presented in more detail in [BS09], where it is referred to as *normally closed latch*.

Moreover, [BS09] proposes another slightly different approach, which uses asymmetric C gates as storage elements for the buffer (see Figure 6.2b). These C gates have one additional (asymmetric) input that must be asserted in order to set the gate, but does not need to be deasserted to reset it again, like a normal input would have to be. The asymmetric input is then fed by the (inverted) output of the buffer's CD, which ensures that the C gates are disarmed as soon as all input transitions arrived, effectively closing the sensitivity window at point (C) in Figure 6.1. In case of the dual-rail code this can be done on a per-bit basis by using the output of the individual OR (or NOR) gates in the CD, which has the benefit of closing the C gate of the respective other rail as soon as one transition arrived. This approach prevents the capturing of the invalid dual-rail state, where true and false rails are asserted simultaneously. However, blindly capturing the first transition creates a potential of forwarding a wrong value (50% when assuming random faults). In that sense it operates similarly to a mutex, with the important difference that depending on the timing of the feedback path, there may still be cases where both outputs are set simultaneously. This buffer modification can be beneficial in combination with an error correcting code on top of the dual-rail code: It prevents the protocol from being upset by an illegal dual-rail state, while the potential corruption of the data value it causes can be undone by the error correction. In the context of this work we refer to this buffer style as the Locking WCHB.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Another approach to avoid the lasting consequence of a faulty transition is to use different storage elements altogether (see Figure 6.2c). For that purpose, the MDHB (already presented in Section 4.5.1) can be used. Here a simple buffer control circuit, consisting of just a single XOR gate, is responsible to enable and disable the buffer's D latches. When the D latches are transparent, input glitches caused by SETs can freely propagate through the latch to the output. Unless the latch is closed in exactly this time instance, the latch will not store the faulty value. Similar to the previous approach the buffer is closed as soon as the CD detects the data phase. Although, strictly speaking, this circuit is not QDI because it introduces a small timing constraint, we still want to include it in our survey since it should show quite a different behavior in the analysis. However, note that this buffer has not been proposed to improve fault-tolerance.



(a) Dual-CD WCHB (b) Locking WCHB (c) MDHB

Figure 6.2: Alternative buffer styles

Finally, we also want to mention a completely different approach proposed by Jang and Martin in [JM05]. This is not really a design or buffer style on its own, rather an all-encompassing technique for hardening existing QDI designs. Here the original design is duplicated and both copies are interlocked with C gates that essentially vote on every intermediate signal of the circuit. The authors formally show that this scheme is able to tolerate (single) faults on any internal signal, which means that in our analysis it should not show any erroneous behavior. However, it is easy to see that this approach more than doubles the area footprint of the original design. Moreover, the additional logic for synchronization of the two replicas also results in a slightly slower circuit. Following the terminology used in [JM05] we refer to a WCHB using the described technique as a DD WCHB.

6.4 Case Study: Pipeline

To experimentally analyze the fault sensitive windows of the QDI buffer styles presented in the previous section, we conducted a fault-injection simulation experiment on a simple 4-stage, 2-bit, dual-rail pipeline. The rationale behind using a pipeline without any processing logic is to enable an unobstructed analysis of only the pipeline buffers.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

6.4.1 Experiment Setup

Figure 6.3 shows our fault-injection simulation target and its testbench. Both pipeline and testbench were generated using `pypr`, which makes it straightforward to iterate over the various buffer designs. For our circuit model we use inertial gate delays and no wire delays. Since we don't target any specific circuit technology with our simulations, the delays of all gates are set to some nominal value, which is then (randomly) varied by ±20% in a uniform way, to account for routing and process variations. This is important to get some skew between the data rails to be able to observe how the different buffers styles handle that.

The victims for the fault injection are the 4 data rails between buffers 1 and 2 as well as the acknowledgment input to buffer 2 originating from buffer 3 (i.e., all the input signals to buffer 2). Moreover, for buffer types that use multi-input gates to generate the enable signals for their actual buffer elements (i.e., the Dual-CD WCHB and the MDHB), this enable signal is targeted as well. The actual fault-injection simulations are carried out using Questa (version 10.6c).

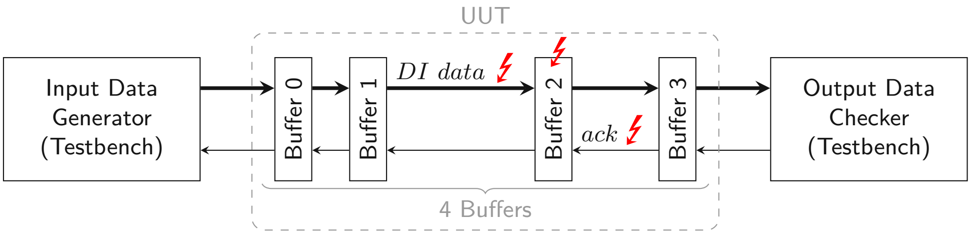

UUT
Input Data Generator (Testbench)
Buffer 0
Buffer 1
DI data
Buffer 2
ack
Buffer 3
Output Data Checker (Testbench)
4 Buffers

Figure 6.3: Simulation Setup

The testbench generates a pseudo-random token stream for the pipeline and checks the produced output tokens for correctness. Whether an injected fault had an effect on the circuit is *only* checked on the primary outputs of the pipeline (i.e., the data outputs of buffer 3 and the acknowledgment output of buffer 0). The motivation behind this decision is to let the pipeline stages after the fault-injection point perform logical and temporal masking as it would occur in a normal pipeline.

To classify the effects of faults we use the following categories:

- Timing deviation: A transition happened earlier or later than expected, when compared to a reference simulation run (without fault injection). The circuit being QDI, this is not a fault, but rather an observation.
- Value fault: A wrong data value was delivered to the output.
- Code fault: An invalid DI code word was observed at the output (i.e., both rails of a dual-rail bit were high).

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

- Glitch: A signal changed its value twice during a protocol phase. This includes protocol violations, e.g., acknowledgment before data completion.
- Deadlock: The circuit reached a state where no further transitions are possible.

One minor downside of only checking the output channel of the target circuit and categorizing the recorded effects into the listed fault classes is that token insertion or deletion cannot be observed directly. When a data token is erroneously removed from or inserted into the pipeline due to the injected fault, the testbench would only see a timing deviation and a value fault for some of the following words extracted from the pipeline[2].

For the result visualization in the next section, we consider the presented list of fault classes to be ordered with ascending importance, i.e., the timing deviation is the least important effect. Whenever a fault injection triggered faults from different classes, only the one with the higher importance was plotted and counted towards the results. An example would be a dual-rail output x that is expected to change from the null phase ($x.T = x.F = 0$) to a logic 1 ($x.T = 1$) which, as a result of a fault injection, instead changes to a logic 0 ($x.F = 1$) at an unexpected time, generating both a value fault and a timing deviation event in the testbench. If subsequently the expected transition arrived causing a code fault ($x.T = x.F = 1$) we would consider the fault injection to yield a code fault and disregard the value fault and timing deviation.

6.4.2 Results

Figure 6.4a shows a single simulation trace of the reference run of the classic WCHB. A single fault-injection simulation always starts with a pre-run, where several rising transitions on the ack_{in} signals are counted in order to skip the initial phase, where the pipeline only starts to fill up (this part is not visible in the figure). Afterwards a single 400 ps pulse is injected into one of the victim signals in the time frame between 0 ns and 120 ns from the end of the pre-run (marked by the 0 ns point on the x-axis). Multiple simulation runs are executed to perform a sweep over this injection window in 250 ps steps. Markers in the figure indicate injection times of pulses that had an observable effect on the pipeline when injected into the respective signal. Their colors represent the classification of that effect.

As we have seen from the analysis in the previous section, the timing of the input signals to a pipeline has a high impact on the fault sensitivity windows. By operating the circuit at a specific speed (i.e., handshake rate) the external signals determine how much time the circuit spends in the different protocol phases, and hence in its sensitive windows. This is again in stark contrast to synchronous designs, where the input signals simply don't have that much "power" over the circuit. Hence, a single simulation trace (Figure 6.4a) alone only yields very little information about fault behavior of a circuit, since it only shows one specific operation point. For that reason, an essential part of the simulation

---

[2] Hence, for the second case study presented in Section 6.6 we use a separate fault class for this purpose.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

setup was the possibility to choose delays of the source and sink when generating new input tokens and acknowledgments, respectively. It allowed us running the fault-injection simulation of the same pipeline with a variety of timing settings, gradually changing its operation from token-limited (where the pipeline stages mostly wait for valid data words to arrive) to bubble-limited (where the pipeline stages receive valid data at their inputs, but need to wait for the acknowledgment from the succeeding stage before being allowed to store the new data word).

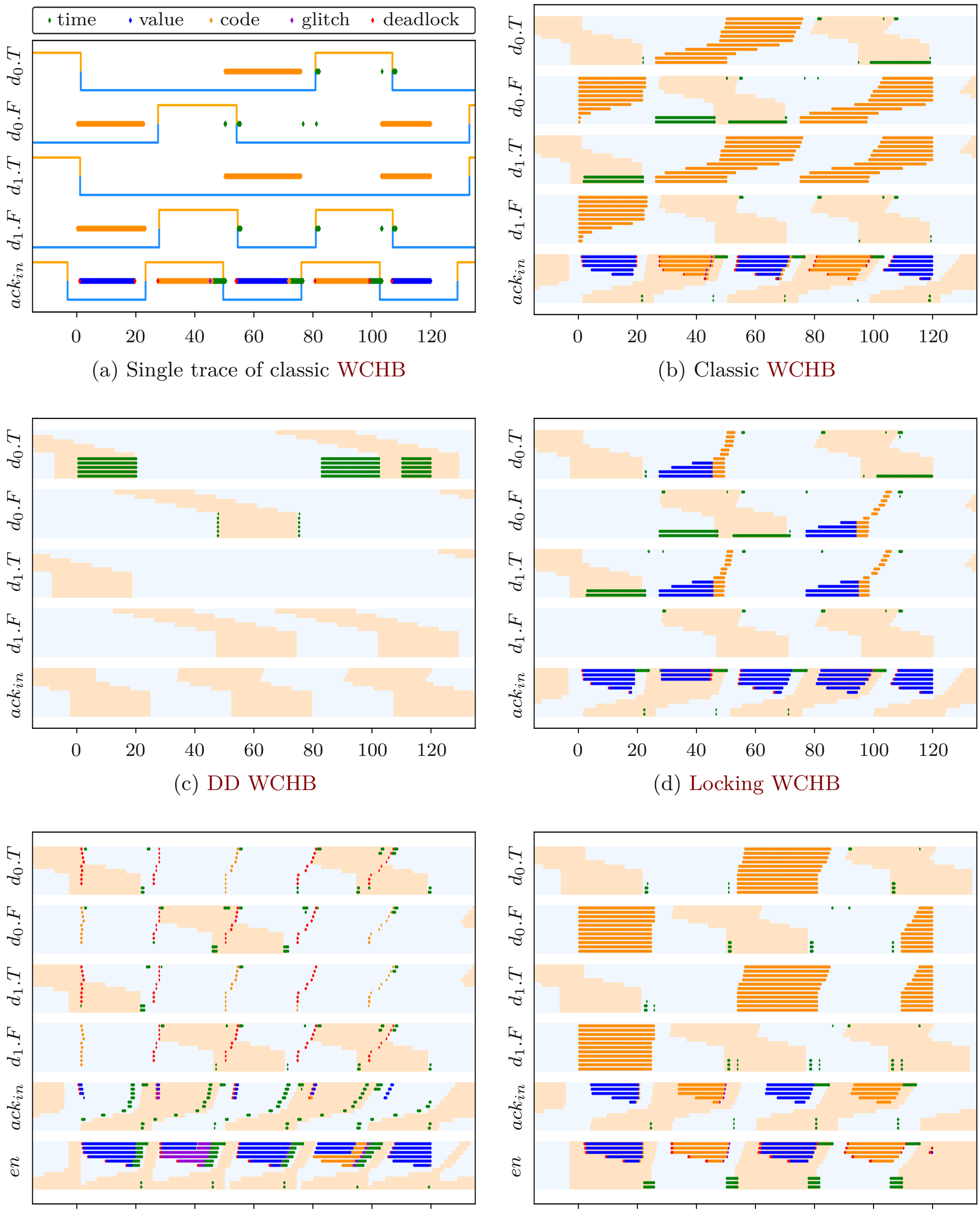
Figure 6.4b shows the results of such a timing variation for the same WCHB pipeline simulated in Figure 6.4a: For each signal, horizontal stripes represent the results of eleven different simulation runs, in which the pipeline transitioned from bubble-limited (top) to token-limited (bottom) operation. A stripe is colored light blue where the respective signal (in the reference run) is supposed to be low and light orange where should be high. Note that Figure 6.4a shows the topmost WCHB simulation stripe from Figure 6.4b. In the same way the other sub-graphs in Figure 6.4 illustrate the behavior of the pipeline using the alternative pipeline implementation styles presented in Section 6.3. This representation style, that allows to pack a large amount of information about the fault behavior of a buffer into a single figure, is one of the contributions of this work.

It allows us to see how the external interface timing affects the sensitivity to faults of the different pipeline implementations when subject to SETs. The data rails for the classic WCHB implementation show how the inactive rail is sensitive to produce a code fault the entire time the receiving C gate is accumulating, irrespective of whether the pipeline runs token- or bubble-limited. The Locking WCHB significantly reduces the sensitive windows by correctly preventing code faults in bubble-limited operation after a transition on one of the two rails was captured by a C gate. It only fails to prevent code faults for a short time corresponding to the feedback delay for locking. In token-limited operation, the injected pulse is captured and the correct and expected transition on the other data rail is prevented from turning the valid, albeit incorrect, value into a code fault.

Unsurprisingly, the DD WCHB style proves to be insensitive to SETs in all operation modes, whereas the Dual-CD WCHB style brings little to no improvement to the sensitivity windows. The MDHB shows very narrow sensitivity windows on the data rails while the enable signal (the signal that activates the D latches of the buffer) is sensitive most of the time. Faults on this signal also have a wide range of possible effects.

Figure 6.5 shows the ratio of injected faults that had an observable effect (other than a timing deviation) to all injected faults. Note that buffer styles which will be introduced in the following section are also already included in this figure. To make a fair comparison and prevent speed differences from influencing the results, faults are only considered during one handshake cycle between two rising edges of the acknowledgment wire. For each considered buffer style, the eleven bars show the results for the eleven simulation runs depicted in Figure 6.4 (going from a bubble-limited mode of operation to a token-limited one). It is apparent that the robustness of the simulated pipelines clearly depends on the external timing. By observing a sweep of simulations with varying timing, one can qualitatively assess the effectiveness of fault mitigation strategies like, e.g., the Locking

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



(a) Single trace of classic WCHB (b) Classic WCHB

(c) DD WCHB (d) Locking WCHB

(e) MDHB (f) Dual-CD WCHB

Figure 6.4: Simulation results for the 4-stage pipeline visualizing the sensitive windows of the considered buffer styles (x-axis shows the simulation time in nanoseconds)

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

WCHB design instead of just looking at numbers without the knowledge of whether the circuit was simulated with token- or bubble-limited timing. If this factor is not taken into account correctly it might lead to a situation where some fault mitigation technique appears advantageous in simulation but turns out to be ineffective in practice. This could be caused by the mere fact that introducing additional logic and therefore delay, changes the circuit operation mode from token-limited to balanced which can (as shown in the figure) in itself bring a significant improvement to the fault sensitivity windows. The figure also shows that, depending on the operation mode a circuit is actually used in, it may not pay off to invest in very high overhead fault mitigation strategies, because simple and comparatively cheap approaches also yield quite good robustness.

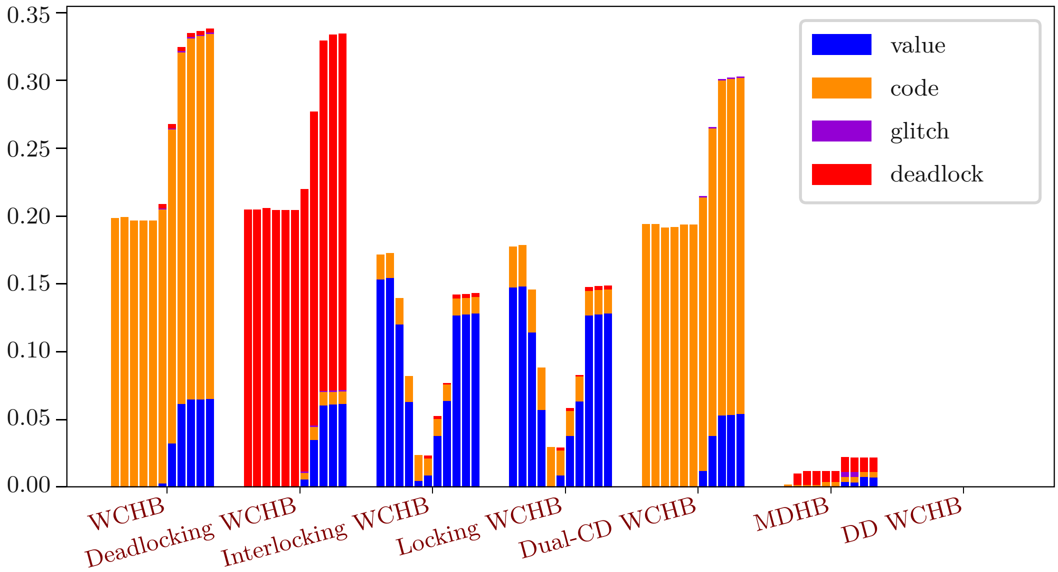


Figure 6.5: Simulation results for the 4-stage pipeline showing the number of fault-injection simulations with an observable effect on the pipeline

6.5 Improved Buffer Designs

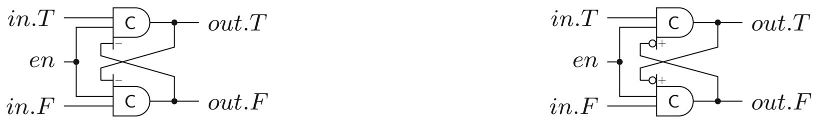
The analysis in the previous section shows that the accumulation behavior of buffers is crucial when it comes to their susceptibility to faults. This section introduces two improved buffer designs that aim to mitigate this fault-accumulating behavior.

6.5.1 Proposed Circuits

Figure 6.6 shows two simple and comparatively low-hardware-overhead modifications to the classic WCHB design. The core idea for both of these modifications is to replace the C gate pairs used as the storage elements for the individual dual-rail bits in the WCHB with cross-coupled asymmetric C gates. As shown in the figure the asymmetric inputs of these C gates are fed by the output of the respective other C gate in the pair.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub

Depending on the type of asymmetric C gate (i.e., whether the asymmetric input is positive or negative), two different circuit behaviors can be achieved. We refer to the resulting circuits as *Deadlocking* and *Interlocking* WCHB.



(a) Deadlocking buffer (b) Interlocking buffer

Figure 6.6: Proposed WCHB modifications

For the Deadlocking WCHB the feedback inhibits the buffer from entering the null phase if it is ever the case that both C gates are set (erroneously) because of some transient event. Hence, the feedback path effectively forces the circuit into a deadlock and prevents it from processing possibly faulty data. This can be useful in applications where the correctness of the output is crucial, while deadlocking is not harmful (fail-stop behavior). A deadlocked system could, e.g., be detected using a timeout and reset to a known working state.

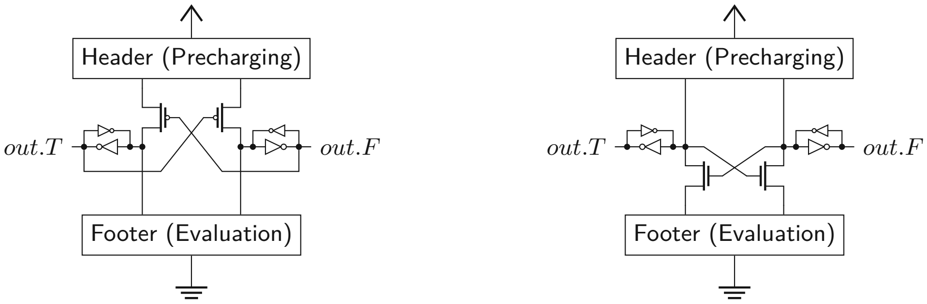
The Interlocking WCHB only allows the first transition at its input to propagate, thus, prohibiting the invalid dual-rail state where both the true and the false rail are asserted simultaneously. In this sense it is similar to the Locking WCHB design discussed in Section 6.3. However, one important difference is that their approach uses the output of the CD to deactivate the corresponding C gates (i.e., prohibiting them from switching to one) in the buffer. While this allows the use of arbitrary DI codes, it also prolongs the feedback path by the delay of the CD, which keeps the buffer open and thus susceptible to SETs on its inputs for a longer time window. Another more subtle difference to the approach in [BS09] is that in order to prevent an erroneous input transition from setting a C gate, after completion *all* C gates are switched to a state-holding mode (i.e., the output is driven by the internal storage loop). In our approach only the C gate connected to the rail that did not transition to high is switched to the state-holding mode (keeping its zero value).

A similar design can also be applied to buffers used in QDI circuits based on precharged/-domino logic, like the PCHB. As shown in Figure 6.7, depending on the desired behavior (deadlocking or interlocking), an additional layer of transistors can be added to either the p- or the n-stack of the buffer. However, these circuits are only presented for the sake of completeness and not further analyzed in this work, since we focus on static designs based on and around the WCHB.

6.5.2 Evaluation

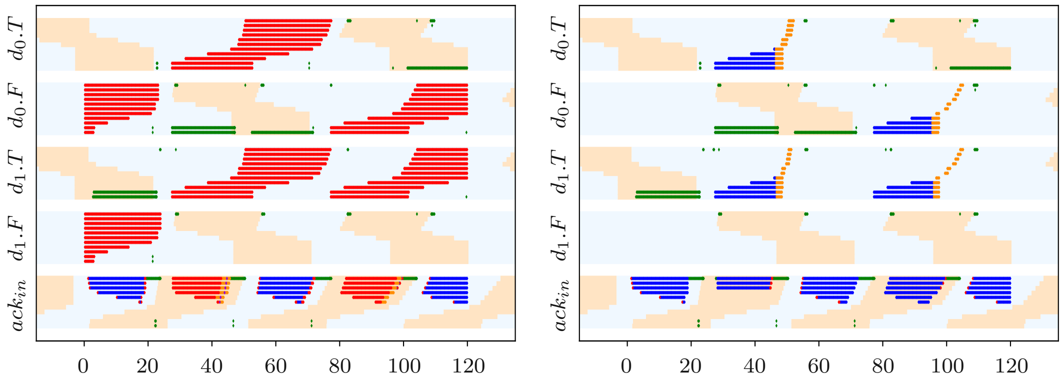
To evaluate the proposed buffers we subjected them to the same experiments as the other buffer styles in Section 6.4. Figures 6.5 and 6.8 show the results of this analysis. It is

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



(a) Deadlocking precharged buffer (b) Interlocking precharged buffer

Figure 6.7: Proposed buffer modifications for precharged/domino logic



(a) Deadlocking WCHB (b) Interlocking WCHB

Figure 6.8: Simulation results for the proposed buffer visualizing their sensitive windows

clearly visible how the Deadlocking WCHB, as expected, turns all code faults seen on the data rails in the classic WCHB into deadlocks, albeit without changing the sensitive window. The Interlocking WCHB in turn significantly shortens the sensitive windows, since in the bubble-limited case the correct transition appears early, and afterwards the interlocking mechanism closes the sensitive window. The non-zero size of the remaining window is due to the propagation delay for the interlocking to become active. Note that these windows are slightly shorter than those found for the Locking WCHB, due to the shorter feedback path. In the token-limited case, there is the potential for a fault on the non-switching rail to arrive before the correct transition on the other rail and thus lock the buffer in an incorrect (but valid) state. This is indicated by the windows with value faults that match the size of the sensitive windows in the original WCHB.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

6.6 Case Study: Multiplier

In this case study we want to deepen our investigation regarding the resilience of different QDI buffer styles. For that purpose, we need more sophisticated target circuits for our fault-injection experiments that, most importantly, also include some actual data processing elements (i.e., not just a simple FIFO pipeline) as such a circuit is more representative of a real-world application. This will then allow us to perform a more comprehensive and meaningful comparison of different QDI buffer (and logic) styles, and to analyze their respective strengths and weaknesses under different operation scenarios and circuit topologies. In order to fulfill this goal we have to conduct a statistically significant number of experiments in reasonable time which requires a highly automated setup for which a special distributed simulation framework had to be developed.

6.6.1 Experiment Setup

For our experiments we chose an unsigned multiplier as target circuit, as it is an elementary function in many applications, and it comprises an appreciable amount of combinational logic – partly in the shape of adders, which by themselves represent another elementary function block. This choice is a trade-off between a highly realistic, complex target circuit that, however, requires excessive computational performance for running the anticipated high number of fault injections, and a too simplistic target, that impairs the significance of the results. Another valuable property of the multiplier is its regularity, which conveniently enables us to either implement it as a linear pipeline or use an iterative approach. For the former case, different degrees of pipelining are possible, a fact that we are using for our experiments.

Figure 6.9 shows the fully pipelined multiplier circuit, where each stage calculates a partial product and adds the result to a sum variable that is then passed to the next stage. The number of pipeline stages is given by the input bit width (plus one additional output buffer). Both factors of the multiplication have the same bit width n while the result has a width of $2n$.

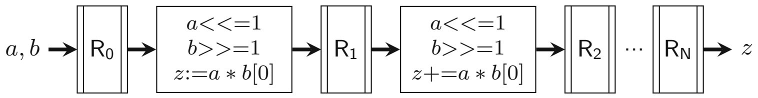


Figure 6.9: Pipelined multiplier

The iterative version of the multiplier has already been discussed in Section 2.4.3. In this circuit variant the partial products are calculated in a feedback loop. Because the hardware to calculate and add up the partial products is shared, the resulting circuit has less area overhead when compared to the pipelined version. However, this also leads to lower throughput and higher latency.

As a rough point of reference: The (fully) pipelined version of the multiplier using the classic WCHB has approximately 500 production rules for an input data width of 4 bit and 2200 rules for the 8-bit version. The iterative multiplier (also using the classic WCHB) requires approximately 540 for the 4-bit version and 1200 for the 8-bit version of the circuit.

Again the target circuits and the required testbenches are created via `pypr`. To implement the combinational logic the DIMS design style is used, the required adders are implemented as RCAs. The script-based circuit generation again allows for an easy change of buffer styles. The generated PRSs are annotated with static (inertial) delays, which we randomly varied by 10% to model PVT variations in the circuit. Wires in the PRSs are considered ideal (i.e., zero delay).

So overall we believe that these multipliers represent reasonably complex targets of which we can, thanks to our tool flow, easily generate numerous variants. Of course the high simulation efforts for the controlled fault injection and detailed tracing limit the attainable input width to (currently) 8 bit. While we are aware that 32 bit or even larger may be desirable, such values are out of reach here. Still we believe that our results can give initial insights, especially since we also include a 4-bit variant, whose comparison with the 8-bit variant facilitates a first judgment of the impact of bit width on the fault effects.

6.6.1.1 Tools and Automation

To obtain the necessary statistical coverage of the experiment space in our desired comparison of pipeline styles, and their dependence on certain parameters, we have to execute over 100 million simulations on a total of 120 different target variations. To make this feasible we extensively rely on automation of target generation and parameterization as well as simulation and result extraction. Figure 6.10 shows an overview of this process.

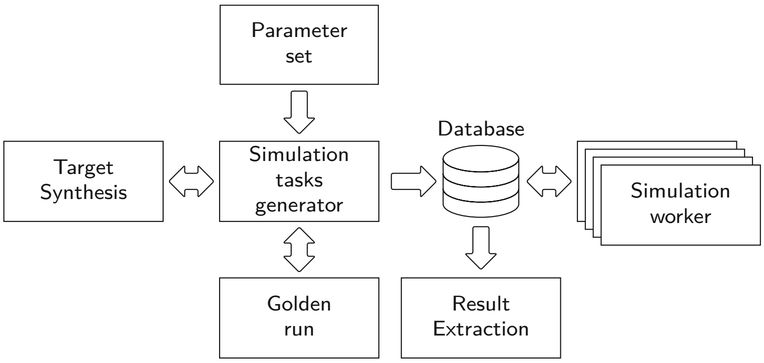


Figure 6.10: Simulation setup

Everything is built around a central Structured Query Language (SQL) database, that stores the simulation tasks that need to executed as well as the results of those simulations. In a first step a parameter set has to be defined which is then issued to the simulation task generator, which uses `pypr` to generate the appropriate circuit as well as an accompanying testbench, and configures all the required simulation parameters.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Some of those parameters must be configured on a per-target and per-experiment basis. Consider for example the delays of the sink and the source in the testbench that interface with the target circuit, which have to be adapted to bring the target into a given load scenario. Since those values depend on the actual target-circuit timing, they are determined using some preliminary simulations during circuit generation. Another example is the amount of simulations to be executed to get an adequate coverage with randomized injection times and targets. Furthermore, the injection window (confining the allowed injection time) is dependent on the target's timing and needs to be adjusted on a per-target basis. During this process also the reference (i.e., fault-free) simulation run is performed.

After all the necessary data for an experiment (circuit, testbench, simulation parameters) has been generated the simulation task is divided into several reasonable-sized work packages that are added to the database. Those work packages can then be processed by multiple simulation workers (on multiple physical machines) in parallel. Simulation workers can be added and removed dynamically from our setup, which helps with restricting and balancing the computational load. Every active simulation worker periodically checks the database for open work packages. If one is present, it is claimed and the associated simulations are performed. To save space only results which deviate from the reference run are saved in the database.

To run the simulation for this case study we use a network of ten physical machines (3.5 GHz 7th generation Intel i5 processor, 16 GB RAM) each running four workers in parallel (one worker per core). The combined runtime of all simulations across all machines is approximately 1200 hours. The actual simulator used is Questa (version 10.6c).

After all simulations are complete the final results can be extracted from the database using SQL queries. The information stored in the database also allows for each individual simulation to be rerun, such that unexpected behavior or interesting effects can be investigated more closely.

A large part of this simulation setup was developed by Patrick Behal in the context of a master thesis [Beh21] and is discussed in more detail in [BHNS21].

6.6.1.2 Fault-Victims and Effect Classes

During a simulation run, we inject faults into all internal signals of the target circuit that are visible on the PRS level, i.e., we again consider production rules to be atomic and do not resolve their internal implementation. However, we don't inject faults on primary inputs and gates driving primary outputs. Like with the previous case study the testbench only monitors the primary outputs of the circuit and records all deviations from the reference run. Again, this choice was made to include the fault masking capability of the circuits in the results. The more masking the circuit provides, the fewer effects will propagate to and be observable at the primary outputs, thus reducing the effective set of signals sensitive to faults.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

We are using the same fault classification as for the previous case study, extended by one additional category, the *token fault*. A token fault occurs when the total number of tokens at an output channel did not match the number of tokens in the reference run. For the last case study we did not explicitly test for this situation and basically only observed its side effects (mostly in the form of value errors and timing deviations). However, for the much more complex target circuits used for this experiment, it makes sense to record them separately.

Note that the effects of a single simulation run can fall into multiple categories, e.g., the circuit may produce a coding fault and then deadlock. We will show all of these effects in our results. This is different from the way we presented the results in the previous case study, where we established a fault hierarchy and only considered the “worst” fault for each simulation.

6.6.1.3 Comparison Parameters

Since our highly automated target circuit generation and simulation framework allows a seamless adaptation of the fault-injection experiment setup, one of our goals was to study what effects changes of certain parameters have on the resilience of a circuit and identify the important ones. For this purpose we systematically varied the following design parameters during our analysis:

- Buffer style (classic WCHB, Dual-CD WCHB, MDHB, Deadlocking WCHB, Interlocking WCHB, DD WCHB)
- Circuit topology (pipelined, iterative)
- Input data width (4 bit, 8 bit)
- Operations per stage (only applies to the pipelined version)
- Pipeline load factor

We use a similar set of buffers as in the previous case study. The only difference is that we don’t include the Locking WCHB here, since it has very similar behavior to the Interlocking WCHB. The buffer styles also list the DD WCHB. However, note that this is not “just” another buffer style, as the duplication and double-checking is applied to the whole PRS not just the production rules/gates implementing buffers. For the previous case study this did not make a difference, since there was no processing logic involved. We included this circuit variant in our experiment as a sanity check for our tool and simulation flow, as it is proven that duplicated and double-checked circuits are tolerant to single faults. Hence, we should not be able to observe any effects with this circuit variant except for timing deviations.

The *operations per stage* parameter of the pipelined implementation dictates, how many stages of computational logic are placed between two pipeline buffers. The fully pipelined

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

multiplier computing one partial product in each pipeline stage has one operation per stage. When set to two, every other buffer is removed and its input and output signals are wired together, leaving the logic for two partial product computations in each pipeline stage.

Both, the operations per stage and the data width parameters allow us to change the ratio of logic related gates to gates used to implement the buffers. When only the input data width is increased the sizes of the used adders increase but so does the width of data words stored in the buffers along with their CDs. By using fewer buffer stages we are able to vary the amount of logic between pipeline stages while keeping the implementation and width of the buffers unchanged.

The *pipeline load factor* is a metric that specifies whether the circuit is operated in a more bubble- or token-limited way. It is the averaged ratio of the time the individual buffers of a circuit spend waiting for the next data or null phase and of the time waiting for the acknowledgment. A well balanced pipeline should have load factor of one when operated at maximum speed; delaying the acknowledgments on the output channel will make the circuit bubble-limited, thus, increasing the load factor. While having the nature of a measurement rather than a design parameter, it can be varied by changes of the average response time of the input and output channels in the testbench. However, note that our simulation setup automation which determines the testbench speed to reach a certain pipeline load factor averages the pipeline load measurement over all buffers while 100 tokens pass through them, while the actual fault-injection simulation is significantly shorter and the measured pipeline load factor of that shorter simulation time can differ from the desired setting.

In the analysis, we differentiate between injection victims being *control* or *data* signals. Control signals, like the acknowledgment and latch enable signals, are responsible for value and spacer token migration through the circuit, which is not to be confused with the full control part of the iterative multiplier implementation, i.e., the upper portion of the circuit depicted in Figure 2.15.

Of course, our tool conveniently allows adapting the relevant fault and delay parameters to a given technology and a given physically grounded fault model. For our more general study here, we performed preliminary experiments to assess the impact of the *width of the injected fault pulse* (relative to the circuit delays) on the observed effect classes. It turned out that pulses shorter than the gate delays were filtered by the inertial delay model we used for the gates (corresponding to electrical masking), while arbitrarily increasing the pulse width did not bring any new insight. Thus, we only used a fixed width of 1.5 ns for the injected pulses that was slightly above the range of randomization we used for most of the gate delays.

6.6.2 Results

The results are presented with plots showing the number of injections that provoked observable effects in each of the respective fault classes, divided by the total number of

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

injections into the considered set of signals (when differentiating between control and data signals). We refer to this metric as the *fault sensitivity*. Figure 6.11 shows the results for the pipelined multiplier (one operation per stage) for 4 and 8 bit input data width and the six considered buffer styles. Control and data signals are shown in separate subplots. Each subplot shows the configured pipeline load factor on the x-axis and the fault sensitivity on the y-axis.

6.6.2.1 Effects of Parameter Changes

Figure 6.11 clearly shows that the *pipeline load factor* is an important parameter with a significant influence on the resilience of the studied circuits. The control signals of all buffer styles have lower fault sensitivity in the token-limited operation, i.e., with a low pipeline load factor. Also for data signals, most buffers show better resilience in token-limited operation. The exception here is the MDHB, which has its data latches transparent while waiting for data and thus naturally provides less temporal fault masking in token-limited operation, which also increases the chance that glitches are propagated to the output.

Comparing the results of the different data widths (4 vs. 8 bit), it can be noticed that the sensitivity of the control signals decreases with the higher data width. This is due to the CD signals being part of the control signal group. A larger data width requires larger CD trees which are less sensitive to faults than for example acknowledgments and buffer enable signals. Increasing the number of CD wires in the control signal group while keeping the number of other control signals mostly unchanged causes a relative decrease of overall control signal sensitivity. For the data signals, we observe an increase of value faults across all the buffer styles, as data width increases – which corresponds with intuition.

Figure 6.12 shows the results for the 8-bit pipelined multiplier with 1 and 2 operations per stage as well as the results for the iterative multiplier variant. The pipeline load factor of the iterative multiplier is not practically controllable by varying handshake delays at the interface to the circuit because of its self-timed operation while computing all partial products in a loop. Thus, for the iterative implementation, the pipeline load factor was only measured, rather than controlled, and found to be 1.25 on average for the different buffer styles, ranging between 0.98 and 1.36. Given that we have seen the pipeline load factor having a significant influence on the results, for a fair comparison, when plotting the results for the pipelined multiplier, we only used data with the pipeline load factor fixed at 1.2, the closest value simulated to that of the iterative multiplier. Figure 6.12a can, thus, be considered as a vertical cut through the 8 bit values from Figure 6.11 at load factor 1.2, slightly right of the center, both for data signals (upwards of 0 on the y-axis in the bar plot) and control signals (downwards from 0 in the bar graph).

Figure 6.12 also demonstrates that increasing the operations per stage does not have a remarkable impact on the fault sensitivity of the circuit. The most significant difference for data signals can be observed with the Interlocking WCHB. However, in this case the

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

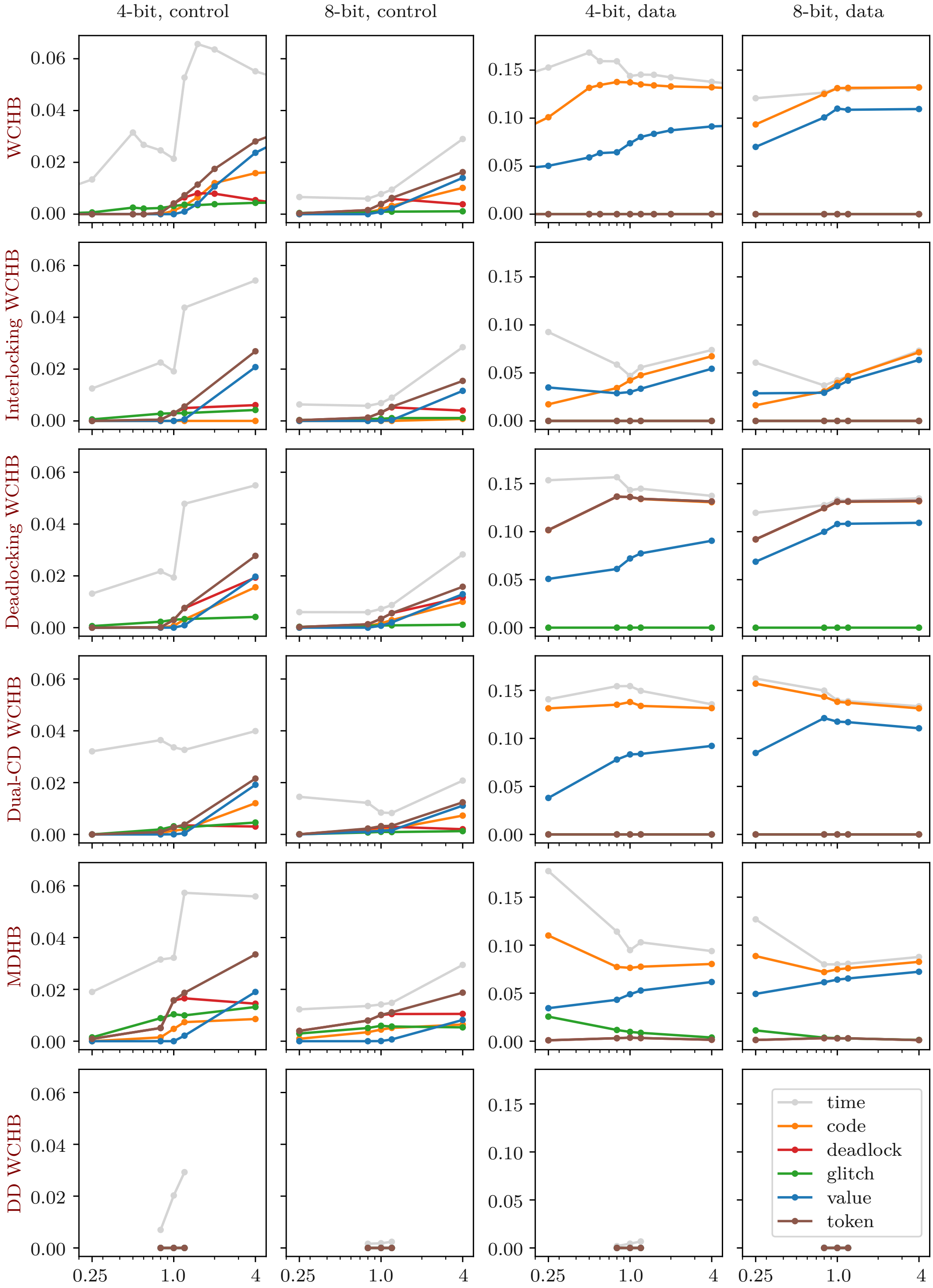
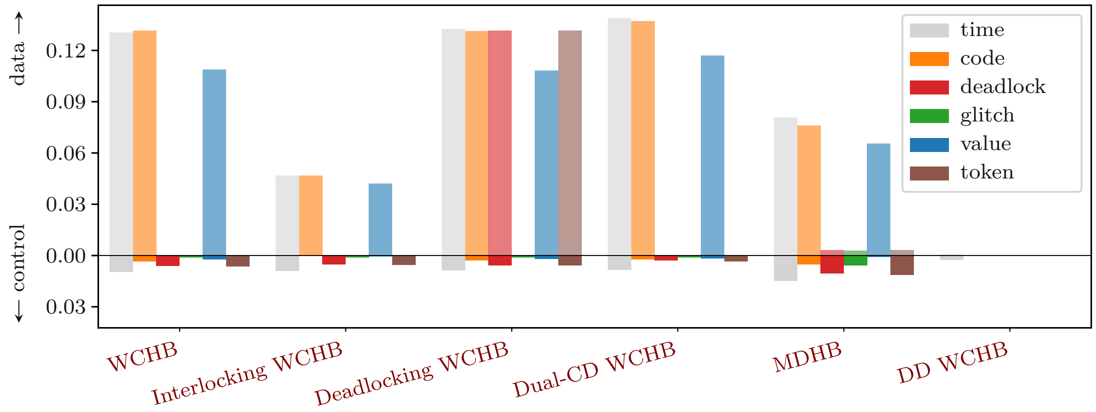
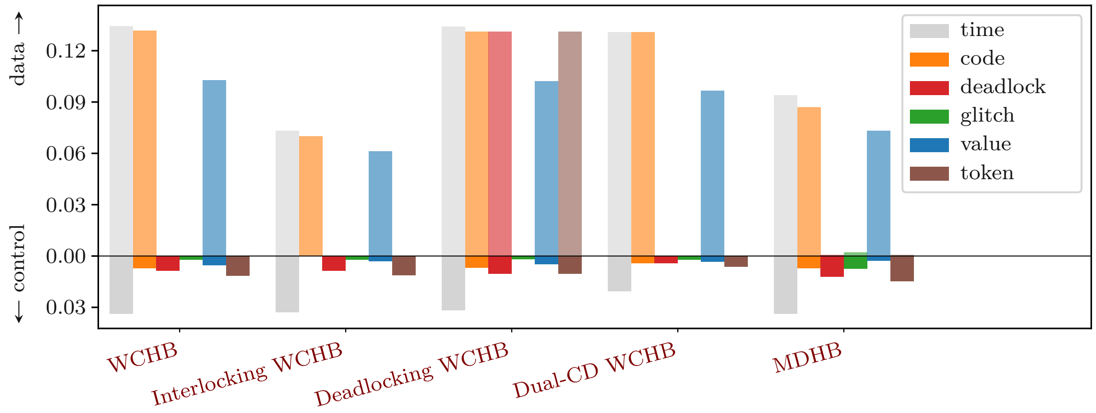


Figure 6.11: Simulation results for the pipelined multiplier circuits, one operation per stage – fault sensitivity to the different fault classes (y-axis) over pipeline load factor (x-axis)

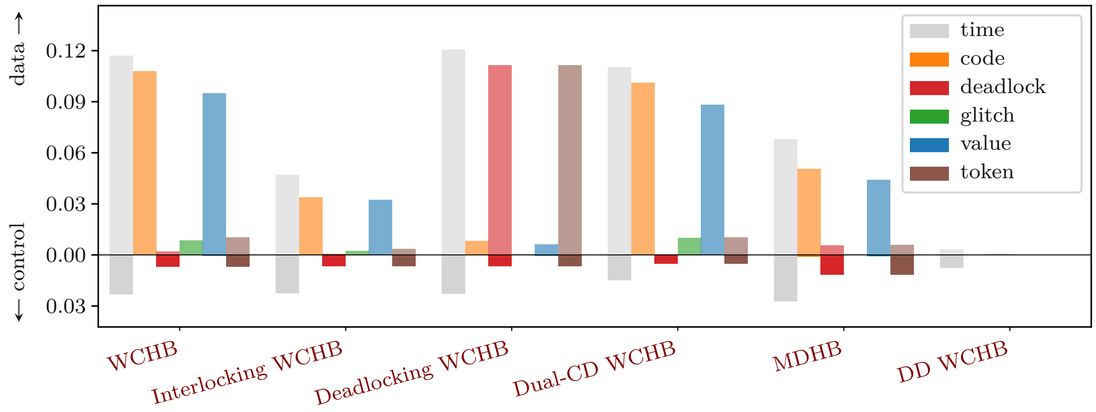
Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.



(a) Pipelined, 1 operation per stage



(b) Pipelined, 2 operations per stage



(c) Iterative

Figure 6.12: Simulation results for the 8-bit multiplier circuit variants, pipelined (1 and 2 operations per stage) and iterative

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

measured pipeline load factors during the fault-injection were 0.78 and 1.28 for the 1 and 2 operations per stage simulations, respectively. Taking the effect of the pipeline load factor from Figure 6.11 into consideration, the deteriorated performance of the Interlocking WCHB can be explained by the imperfection of our automated testbench setup yielding a discrepancy between the targeted pipeline load factor of 1.2 and the actual pipeline load factor during fault injection.

For the iterative implementation we observe a better resilience. One notable difference lies in the reduced coding faults for the Deadlocking WCHB, because the iterative nature of the circuit prevents the coding faults detected in the internal loop to be propagated to the output when a deadlock occurs. Where the Interlocking WCHB more or less stochastically masks a fault completely or converts it into a value fault, the deadlocking variant always stops the operation of the circuit altogether.

6.6.2.2 Buffer style comparison

Compared to the WCHB, the masking provided by the Interlocking WCHB version successfully reduced the number of faults that can propagate through the buffer. While being seemingly worse, the Deadlocking WCHB in fact performs as expected and, by not allowing a spacer into a buffer once a coding fault has been detected, it turns coding faults into some residual coding, deadlock and token events logged by the testbench.

The Dual-CD WCHB performs similar to the WCHB albeit showing less sensitivity for control signals. Similar to the increase of the data width, also here the added CDs increase the relative number of signals in the group of the control signals thus making the overall result relatively better.

It can also be seen that the DD WCHB (and logic) style successfully withstands all fault-injections into the circuit. The only observable effect compared to the reference run are timing deviations. The resilience of this style along with its very large size compared to the other styles is also the reason why we did not simulate all circuit variations with this style. The Dual-CD WCHB performs similar to the WCHB buffer, while the MDHB buffer is more resilient albeit propagating more glitches and being more sensitive on the control signals.

6.7 Conclusion

In this chapter we performed two simulation-based fault-injection case studies with the goal of providing a first insight into the sensitivity of different QDI circuits to transient faults.

Given that this sensitivity strongly depends on the operation speed of a circuit (e.g., the sink and source speed in a pipeline), our conclusion was that a systematic analysis requires a visualization of this dependence. Our proposed solution here is a graphical representation of the sensitive windows for each relevant signal, aggregated for different settings of source and sink speed, and showing a color code for the observed effect of a

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

fault at the point corresponding to its injection. The required data for these plots were generated, in our first case study on linear pipeline circuits which use different variants of the WCHB. Using this compact representations we have identified a key vulnerability of the classic WCHB and proposed two enhancements. For both circuit variants, namely the Interlocking and the Deadlocking WCHB, we have sketched target use cases and given evidence for their proper operation through our analysis.

To extend the significance of our results we performed a second case study on a much more complex target circuit which also incorporates processing logic. For this purpose we have selected a multiplier, since its complexity, while not being trivial, still allows understanding and tracing all its operation details and simulation with reasonable computational efforts. Our sophisticated fully automated circuit generation and simulation framework allowed us to perform many millions of fault injections that are still well controlled and reproducible in all detail for later analysis of interesting cases. By varying several parameters like data width, pipeline structure, buffer style and operation speed we were able to directly observe the differences in the effects that the same types of injected faults cause under these different conditions. This allowed a direct and meaningful comparison of the sensitivity of different buffer styles under different operation modes.

Since this work is part of an ongoing research project, the tools developed for the experiments in this thesis also lay the ground work for further investigations. Our setup will, with some minor refinements, allow us numerous further investigations like taking the effect analysis to the level of a single buffer, or quantifying the degree of masking between pipeline stages. The vision is to understand the masking effects well enough to be able to make quantitative predictions for a given parameter set, which would be a valuable foundation for optimizations of the circuits' resilience.

In addition, once we have gained a better understanding of the impact of certain parameter choices, we can reduce the parameter space and use the available computational power to extend our list of target circuits towards more complex ones.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 7

Conclusion and Future Work

In this thesis several contribution to the field of asynchronous and specifically QDI circuits have been presented. This chapter briefly summarizes our findings and lays out possible paths for future research.

Starting with **Chapter 3** we proposed a novel timing (and clock) domain crossing scheme based on the use of Muller pipelines. To this end, we developed a synchronous sampling model for the inherently asynchronous Muller pipeline and showed what timing constraints are necessary in order to minimize the risk for a metastable upset. The resulting circuits proved to be quite simple, have low area overhead and are straightforward to implement – even on an FPGA platform not at all meant for asynchronous design. Our measurements demonstrated that the approach is capable of delivering high throughput, with acceptable latency and is resilient against severe clock jitter. An interesting direction for future research would be to test the circuits for higher performance target technologies. Another possible extension would be to develop versions of the circuits which don't rely on C gates (e.g., based on MOUSETRAP) and only use standard library elements. This would make the proposed approach more attractive for traditional target technologies not featuring this special asynchronous component.

Chapter 4 was dedicated to the investigation of various important aspects of DI communication links and made several contribution to this field. First, we proposed a (semi-generic) strategy for mapping data words to code words in constant-weight codes. The approach separates the code words into a systematic and a non-systematic part to reduce the encoding and decoding overhead. We also explored a new class of DI protocols which represent a hybrid between the classic RZ and NRZ schemes. The protocols improve the power and/or coding efficiency over RZ protocols (using the same DI code), but without the hardware overhead that would be entailed by a "full" NRZ protocol implementation. Regarding the design of completion detectors for the 4-phase protocols we built on the work of Piestrak [Pie98] and provided a fully generic orphan-free design approach for (arbitrary) constant-weight and Berger codes. The proposed technique yields the most area-efficient circuits in literature for many DI codes and is also applicable to the hybrid protocols. The chapter concluded with an extensive case-study which compared the different protocols and codes as well as the circuits required to implement them and highlighted their potential strengths and weaknesses. A potential path for future work would be to look at the implementation of fully QDI links, i.e., replacing the BD input and output channel of the link model used in this work with dual-rail DI channels.

In **Chapter 5** we introduced a comprehensive Python-based tool set called `pypr`, which is able to generate, analyze, simulate and verify asynchronous circuits and is built around a custom PRS-inspired circuit description language. The framework allows the description of asynchronous circuits on the gate-level but also supports the use of the data-flow level for a more abstract and convenient design entry. QDI combinational circuits can be synthesized from Verilog specifications using a dual-rail synthesis algorithm, that internally relies on the open-source EDA tool Yosys. Moreover, we also presented a survey of QDI dual-rail full adder circuits and proposed our own low-area-overhead variants, which are based on the use of binary sorting networks. Regarding the verification of QDI/SI circuits, we showed how a basic model checking system is quite straightforward to implement using standard open-source tools.

The short- to midterm strategy for our design framework is to continue to use and extend it, because a lot of our internal tools (for our ongoing fault-tolerance research) are built around it and depend on it. To broaden its applicability and enable direct comparisons of different design styles extensions for BD and maybe even synchronous circuits would be very useful. Another feature that would definitely benefit a lot from further refinement is the model checker. In our opinion, this is an interesting and promising direction for future work, especially with regard to the analysis of fault behavior. When evaluating fault-mitigation strategies such a tool could, e.g., be used to automate the process of identifying edge cases where a particular strategy fails or to find vulnerable signals. Some important aspects that could be addressed in a first step are the completeness of the model checking approach (using, e.g., inductive techniques or Craig interpolation [BK18]) and the overall performance. It would also be very expedient if it would be possible to express and verify temporal properties of circuits.

For the longer term it might be beneficial to direct our ambitions towards ACT [AHY+21] to facilitate the standardization of the tools used within the asynchronous research community and to benefit from the mature tools and algorithms provided by it. We currently also have an ongoing master thesis that looks into how ACT can be used for our fault-injection experiments. In the course of that work a code generator has been developed, that is able to export circuits created using `pypr` to a format that can be processed by ACT.

However, we definitely also see long-term value in our Python-based asynchronous design methodology and think that it can be useful to introduce people to asynchronous design. Using a quite common programming language that many people are already familiar with

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

can lower the barrier of entry into the field. Hence, we also plan to use our framework for teaching asynchronous design.

Finally, we performed an extensive study on the behavior of QDI circuits under transient faults in **Chapter 6**. Due to the vast size of the design space of QDI circuits we restricted our investigation to WCHB-based circuits and variations thereof. We used large-scale fault-injection simulations, which gave first insights into the fault-sensitivity of different QDI circuits and their inherent masking capabilities. We showed that the sensitivity of a given circuit strongly depends on its operation speed and proposed a compact but very expressive visualization of the fault-sensitive windows of the different buffer styles. Furthermore, two new buffer designs have been proposed that aim to alleviate some of the shortcomings of a classic WCHB. The experiments demonstrated the viability of our proposed solutions.

Since the work in this chapter is part of a larger and ongoing research project, we already have plans for how to proceed with it. The results presented here focused on how buffers react to faults and how they can be hardened. However, another important factor contributing to the overall fault resilience of a circuit is the combinational logic. Here we want to investigate differences between different logic styles and implementation variants for basic blocks (e.g., full adders). For example, since NCLX logic contains no storage elements (i.e., C gates) directly in the data path of combinational logic there should be less potential for capturing a fault than, e.g., for DIMS. Moreover, we are currently working on functionality in `pypr` that allows us to track the sensitive time frames for all gates in a circuit. The basic idea is to use this information in to assess and optimize the resilience of a circuit with less need for complex fault-injection simulations.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

List of Figures

2.1 Asynchronous handshaking protocols . . . 9
2.2 RZ (4-phase) DI protocol . . . 11
2.3 NRZ (2-phase) transition signaling DI protocol . . . 12
2.4 Asynchronous circuit model . . . 13
2.5 Examples for gate and wire orphans . . . 14
2.6 Possible effect of a non-isochronic wire fork . . . 15
2.7 2-input Muller C gate . . . 16
2.8 Asymmetric C gate with 4 inputs . . . 17
2.9 4-stage Muller pipeline . . . 18
2.10 C gate specifications . . . 19
2.11 STG describing a 3-stage MPL . . . 20
2.12 SDFS operations . . . 20
2.13 Token progression in a 4-phase 3-stage pipeline ring SDFS . . . 22
2.14 Fork and join implementations for 4-phase BD channels . . . 23
2.15 SDFS of an iterative multiplier circuit . . . 24
2.16 4-phase BD pipeline with MPL as handshake control circuit . . . 25
2.17 3-stage WCHB pipeline . . . 26
2.18 WCHB pipeline with combinational logic . . . 28
2.19 QDI dual-rail AND gates . . . 28
2.20 NCL AND gates . . . 29
2.21 NCLX AND-OR structure with 3 inputs and input/internal CDs . . . 30
2.22 PCHB pipeline stage . . . 31

3.1 Synchronous system with asynchronous output channel . . . 34
3.2 FIFO interfaces . . . 34
3.3 MPL circuits . . . 36
3.4 Setup/hold window . . . 39
3.5 Desynchronizer circuit . . . 44
3.6 Resynchronizer circuit . . . 47
3.7 Example timing diagram ($n = 2$) . . . 48
3.8 Bisynchronous FIFO prototype . . . 50

4.1 Delay-insensitive link overview . . . 56
4.2 PSCWC encoder for $C^{ps}_{j,k,s}$ . . . 63

4.3 Circuits for the PS 3-of-6 code 63
4.4 DS protocol state diagram 67
4.5 SDS protocol state diagram 68
4.6 SDS protocol example timing diagram 69
4.7 Power metric improvement for the SDS protocol 70
4.8 SDDS protocol state diagram 71
4.9 UBS protocol state diagram 72
4.10 Power metric comparison for Berger code protocols 73
4.11 NRZ CD constructed from RZ CD with 2-phase wrapper circuit 74
4.12 Comparator cells 75
4.13 T^4 sorting networks 75
4.14 4-phase 2-of-4 CDs 76
4.15 Selection network 77
4.16 Proposed m-of-n completion detection approach 79
4.17 Proposed 3-of-6 CD ($q = 4$, $r = 2$) 80
4.18 CD for Berger codes by Piestrak 81
4.19 Orphan-free CD for Berger codes 82
4.20 CN-based BUC 83
4.21 Abstract CN specification for Berger CDs 83
4.22 CD examples for the SDS protocol 85
4.23 CD examples for the UBS protocol 86
4.24 Pipeline implementation for proposed protocols (three stages) 87
4.25 RZ protocol transmitter and receiver 88
4.26 SDS/UBS protocol transmitter 89
4.27 DS protocol transmitter and receiver 90
4.28 NRZ protocol transmitter and receiver 91
4.29 Hardware overhead for different DI links 94
4.30 WCHB pipeline circuit model with delays (three stages) 96
4.31 PN model for the WCHB pipeline (three stages) 96
4.32 MDHB pipeline circuit with delays (three stages) 97
4.33 PN model for the MDHB pipeline (three stages) 97

5.1 Binary addition circuits 130
5.2 NCLX FA (input CD omitted from figure) 131
5.3 NCLX FA with 3-input gates (input CD omitted from figure) 132
5.4 Toms' FA 132
5.5 Modification of Toms' FA (input CD omitted from figure) 133
5.6 SN-based FA (SN-FA) 134
5.7 SN-based FA with fast carry generation (SNFC-FA) 134
5.8 SN-based FA with explicit completion detection (SNX-FA) 135
5.9 Area overhead of the presented adder circuits 136
5.10 3-bit LFSR 139
5.11 DI dual-rail channel protocol phases 147

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.12 Alternating token generator circuit 152
5.13 Timing diagram representing the model found by the model checker . . . 153
5.14 3-stage 2-bit WCHB pipeline circuit 154
5.15 Timing diagram representing the model found by the model checker . . . 155
5.16 Model checker runtime for the multiplier and adder circuits 156
5.17 `pypr` tool-flow example 157

6.1 WCHB time windows 163
6.2 Alternative buffer styles 165
6.3 Simulation Setup 166
6.4 Simulation results for the 4-stage pipeline (sensitive windows) 169
6.5 Simulation results for the 4-stage pipeline (total fault count) 170
6.6 Proposed WCHB modifications 171
6.7 Proposed buffer modifications for precharged/domino logic 172
6.8 Simulation results for the proposed buffer (sensitive windows) 172
6.9 Pipelined multiplier 173
6.10 Simulation setup 174
6.11 Simulation results for the pipelined multiplier circuits 179
6.12 Simulation results for the 8-bit multiplier circuit variants 180

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

List of Tables

4.1 Examples for Partially Systematic Codes 62
4.2 {1,2,3}-of-4 multi-encoder for the PS 3-of-6 code C_3^{ps} 65
4.3 {3,4}-of-7 multi-encoder for PS 4-of-8 code C_4^{ps} 65
4.4 {2,3,4,5}-of-7 multi-encoder for the PS 5-of-10 code C_5^{ps} 65
4.5 {3,4,5,6}-of-9 multi-encoder for the PS 6-of-12 code C_6^{ps} 65
4.6 d values used for the power metric evaluation of the UBS protocol 73
4.7 SN implementation costs (minimal depth) 75
4.8 Hardware overhead for encoders and decoders 93
4.9 Hardware overhead for CDs 93
4.10 Delay estimation parameters for m-of-n CDs 99
4.11 Delay estimation parameters for Berger CDs 100

5.1 `pypr` sub-packages 116
5.2 QDI FA properties 135
5.3 Estimated full adder delay parameters 138

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Acronyms

API Application Programming Interface. 116

ASIC Application-Specific Integrated Circuit. 53

BD Bundled-Data. 2, 3, 10, 13, 20, 22–25, 27, 33, 34, 55, 56, 85, 87, 89, 90, 120, 160, 161, 184, 187

BMC Bounded Model Checking. 138, 139, 141, 163

BUC Binary-to-Unary Converter. 81–83, 85, 188

CD Completion Detector. 3–5, 10–12, 26–32, 56, 57, 73, 74, 76, 77, 79–87, 89–93, 95, 97–102, 115, 117, 125–128, 131–136, 148, 163–165, 171, 177, 178, 181, 183, 187, 188, 191, 197

CFG Context-Free Grammar. 105, 106

CMOS Complementary Metal-Oxide-Semiconductor. 16, 29, 53, 104, 135, 160

CN Comparator Network. 79, 80, 82–85, 92, 98–100, 188

CNF Conjunctive Normal Form. 138, 140

DI Delay-Insensitive. ix, xi, 3–6, 10–15, 17, 20, 23, 25, 27, 31, 55–58, 64, 66, 67, 69, 74, 85, 87, 90, 91, 95, 101, 102, 121, 142, 146–148, 166, 171, 183, 184, 187, 188

DIMS Delay-Insensitive Minterm Synthesis. 28–30, 126, 131, 135, 136, 138, 155, 174, 185, 197

DS Data Spacer. 66–68, 70, 73, 85, 89, 90, 92–94, 100, 102, 188

EBNF Extended Backus–Naur Form. 105

EDA Electronic Design Automation. 1, 30, 122, 184

FA Full Adder. 125, 126, 129–138, 184, 185, 188, 191, 197, 198

FIFO First In, First Out (Buffer). 17, 34, 35, 41–44, 46, 47, 49–52, 173, 187, 197

FPGA Field-Programmable Gate Array. 49–51, 53, 183

GALS Globally Asynchronous Locally Synchronous. 2, 3, 5, 55

GE Gate Equivalent. 91, 92, 135

HA Half Adder. 63, 125, 130, 131, 136

HG Hysteresis Generator. 79, 80, 82, 84, 98, 99

I/O Input/Output. 109, 149

LEDR Level-Endcoded Dual-Rail. 12, 67

LETS Level-Encoded Transition Signaling. 12, 87

LFSR Linear Feedback Shift Register. 50, 139, 140, 188

LSB Least Significant Bit. 62, 130

LUT Lookup Table. 49, 50

MDHB MOUSETRAP-style D latch Half Buffer. 87, 95–97, 165, 166, 168–170, 176, 178–181, 188

MPL Muller Pipeline. 17, 19, 20, 24–27, 34–38, 41–45, 48, 50–52, 183, 187

MSB Most Significant Bit. 82, 137

MTBU Mean-Time Between Upsets. 43, 46, 49, 52

NCL Null Convention Logic. 29, 187, 197

NRZ Non-Return-to-Zero. 10, 12, 57, 59, 66, 70, 73, 74, 90–98, 183, 187, 188

PCB Printed Circuit Board. 55

PCFB Prechared Full Buffer. 32

PCHB Prechared Half Buffer. 31, 32, 171, 187

PN Petri Net. 18, 19, 95–97, 188

PRS Production Rule Set. 104–125, 127, 128, 141–144, 146, 148, 149, 152–157, 174–176, 184

PS Partially-Systematic. 62–65, 94, 188, 191

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

PSCWC Partially-Systematic Constant-Weight Code. 57, 59, 62, 63, 70, 92, 95, 98, 187

PVT Process, Voltage and Timing. ix, 1, 3, 55, 174

QDI Quasi Delay-Insensitive. ix–xi, 3–7, 11, 13–15, 23, 25–31, 56, 57, 74, 76, 80, 82, 103, 105, 115, 122, 124, 129–132, 135, 139, 150, 155, 156, 159–166, 171, 173, 181, 183–185, 187, 191, 197, 198

RCA Ripple-Carry Adder. 129, 130, 133–138, 174

RTL Register-Transfer Level. 20, 199

RZ Return-to-Zero. 10–12, 31, 57–59, 66, 67, 69–71, 73, 74, 87, 88, 90, 92–96, 98–100, 142, 183, 187, 188

S/H Setup/Hold. 37, 39, 50

SAT Boolean Satisfiability. 138–143, 145, 149–151

SDDS Short Distance Dual Spacer. 71, 73, 89, 92, 94, 100, 188

SDFS Static Data-Flow Stucture. 20–24, 187

SDS Short Distance Spacer. 68–72, 84, 85, 88, 89, 92–94, 99, 188

SET Single-Event Transient. 3, 4, 151, 155, 159–161, 165, 168, 171

SEU Single-Event Upset. 3, 151

SI Speed-Independent. 13–15, 18, 138, 139, 141, 142, 144, 150, 153, 155, 156, 159–161, 184

SMT Satisfiability Modulo Theories. 138

SN Sorting Network. 74–85, 100, 129, 132–136, 184, 188, 191, 197

SNML Selection Network Merging Logic. 77–80, 82–85

SoC System on a Chip. 33, 55

SQL Structured Query Language. 174, 175

ST Self-Timed. 13

STG Signal Transition Graph. 18–20, 86, 88–91, 95, 141, 162, 187

UBS Unbalanced Spacer. 71–73, 85, 86, 88, 89, 92–94, 100, 188, 191

UPG Unate Product Generator. 81

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

UUT Unit Under Test. 123, 124

VHDL Very High Speed Integrated Circuit Hardware Description Language. 91, 104, 113, 116, 122–124, 157, 199

WCHB Weak-Conditioned Half Buffer. 25–28, 31, 32, 86, 87, 93, 95–97, 116–118, 120–123, 152–155, 160–165, 167–174, 176, 179–182, 185, 187–189, 197

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Glossary

A/S FIFO A FIFO buffer with a synchronous write and an asynchronous read port (or vice versa). 34, 35, 41, 52

DD WCHB A WCHB where the duplication and double-checking (DD) modification introduced in [JM05] has been applied . 165, 168–170, 176, 179–181

Deadlocking WCHB A WCHB variant using cross-coupled asymmetric C gates introduced in Section 6.5. 170–172, 176, 179–181

DIMS-FA A QDI FA constructed using 2-input DIMS gates (using the “Input Completeness Relaxation” optimization[JN07]). 131, 135, 136, 138

Dual-CD WCHB A WCHB variant with CDs on both its input and its output channel [BS09]. 164–166, 168–170, 176, 179–181

Interlocking WCHB A WCHB variant using cross-coupled asymmetric C gates introduced in Section 6.5. 170–172, 176, 178–181

Locking WCHB A WCHB variant using asymmetric C gates proposed in [BS09] . 164, 165, 168–172, 176

NCLX NCL with explicit completion detection – a QDI logic style (also referred to as NCL_X or NCL-X). 29, 30, 125, 126, 128, 131, 132, 136, 185, 187, 188, 197

NCLX2-FA A QDI FA constructed using the NCLX design style using only 2-input basic gates. 131, 135, 136, 138

NCLX3-FA A QDI FA constructed using the NCLX design style using basic gates with up to three inputs. 131, 132, 135, 136

SN-FA A QDI FA based on the use of binary SNs. 132, 134, 135, 138, 188, 198

SNFC-FA A QDI FA based on the use of binary SNs with a specially optimized fast carry (FC) signal path. 133–135, 138, 188

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

SNX-FA A QDI FA based on the SN-FA, but without C gates in the signal paths to the data outputs and with additional explicit completion detection. 134–136, 138, 188

Toms-FA A QDI FA constructed using Toms' synthesis approach [Tom06]. 132, 135, 136, 198

TomsX-FA A QDI FA based on the Toms-FA, but with AND gates instead of C gates and additional explicit completion detection. 132, 135, 136

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Software and Tools

pypr Python Production Rule Package – The Python-based asynchronous circuit modeling and analysis framework developed for this thesis, https://gitlab.ecs.tuwien.ac.at/eda/pypr.git. 103, 105, 107, 112, 115–124, 138, 141, 148, 153, 155–157, 160, 163, 166, 174, 184, 185, 189, 191

ACT Asynchronous Circuit Toolkit, an open-source asynchronous design suite developed at Yale University R. Manohar and his research group, https://avlsi.csl.yale.edu/act. 184

GHDL an open-source VHDL simulator, https://github.com/ghdl/ghdl. 124, 157

Questa a commercial digital simulator, https://eda.sw.siemens.com. 124, 157, 166, 175

Synopsys Design Compiler a commercial RTL synthesis suite by Synopsys, https://www.synopsys.com. 91

Workcraft an open-source framework for interpreted graph models, https://workcraft.org. 18

Yosys Yosys Open SYnthesis Suite, an open-source Verilog synthesis tool, https://github.com/YosysHQ/yosys. 124, 126, 129, 156, 184

Z3 an open-source theorem prover from Microsoft Research, https://github.com/Z3Prover/z3. 138, 140, 142, 143, 156

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Bibliography

[AG17] A. M. S. Abdelhadi and M. R. Greenstreet. Interleaved Architectures for High-Throughput Synthesizable Synchronization FIFOs. In *23rd IEEE International Symposium on Asynchronous Circuits and Systems*, pages 41–48, May 2017.

[AHY$^+$21] Samira Ataei, Wenmian Hua, Yihang Yang, Rajit Manohar, Yi-Shan Lu, Jiayuan He, Sepideh Maleki, and Keshav Pingali. An open-source eda flow for asynchronous logic. *IEEE Design Test*, 38(2):27–37, April 2021.

[Ale69] V. E. Alekseev. Sorting algorithms with minimum memory. *Cybernetics*, 5(5):642–648, Sep. 1969.

[Alt16] Altera Corporation. *Cyclone IV Device Handbook*, March 2016.

[AM15] Jean-Luc Autran and Daniela Munteanu. *Soft Errors: From Particles to Circuits*, volume 39 of *Devices, circuits, and systems*. CRC Press, Baton Rouge, 2015.

[AYM$^+$07] R. W. Apperson, Z. Yu, M. J. Meeuwsen, T. Mohsenin, and B. M. Baas. A Scalable Dual-Clock FIFO for Data Transfers Between Arbitrary and Haltable Clock Domains. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(10):1125–1134, Oct. 2007.

[Bau05] R.C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability*, 5(3):305–316, Sep. 2005.

[Beh21] Patrick Behal. Quantitative Comparison of the Sensitivity of Delay-Insensitive Design Templates to Transient Faults. Master's thesis, Institut für Computer Engineering, TU Wien, 2021.

[Ber61] J.M. Berger. A note on error detection codes for asymmetric channels. *Information and Control*, 4(1):68–73, 1961.

[BF02] J. Bainbridge and S. Furber. Chain: a delay-insensitive chip area interconnect. *IEEE Micro*, 22(5):16–23, Sep. 2002.

[BHN+21] Patrick Behal, Florian Huemer, Robert Najvirt, Andreas Steininger, and Zaheer Tabassam. Towards Explaining the Fault Sensitivity of Different QDI Pipeline Styles. In *27th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 25–33, 2021.

[BHNS21] Patrick Behal, Florian Huemer, Robert Najvirt, and Andreas Steininger. An Automated Setup for Large-Scale Simulation-Based Fault-Injection Experiments on Asynchronous Digital Circuits. In *24th Euromicro Conference on Digital System Design (DSD)*, pages 541–548, Sep. 2021.

[BK18] Armin Biere and Daniel Kröning. *SAT-Based Model Checking*, pages 277–303. Springer International Publishing, Cham, 2018.

[BMS+09] R. P. Bastos, Y. Monnet, G. Sicard, F. Kastensmidt, M. Renaudin, and R. Reis. Comparing transient-fault effects on synchronous and on asynchronous circuits. In *15th IEEE International On-Line Testing Symposium*, pages 29–34, June 2009.

[BOF10] Peter A. Beerel, Recep O. Ozdag, and Marcos Ferretti. *A Designer's Guide to Asynchronous VLSI.* Cambridge University Press, 2010.

[BRG+18] Aymane Bouzafour, Marc Renaudin, Hubert Garavel, Radu Mateescu, and Wendelin Serwe. Model-checking synthesizable systemverilog descriptions of asynchronous circuits. In *24th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 34–42, May 2018.

[BRWG05] G. F. Bouesse, M. Renaudin, A. Witon, and F. Germain. A clock-less low-voltage AES crypto-processor. In *Proceedings of the 31st European Solid-State Circuits Conference (ESSCIRC 2005).*, pages 403–406, Sep. 2005.

[BS09] W. J. Bainbridge and S. J. Salisbury. Glitch Sensitivity and Defense of Quasi Delay-Insensitive Network-on-Chip Links. In *15th IEEE Symposium on Asynchronous Circuits and Systems*, pages 35–44, May 2009.

[BTEF03] W.J. Bainbridge, W. B. Toms, D.A. Edwards, and S.B. Furber. Delay-insensitive, point-to-point interconnect using m-of-n codes. In *Ninth International Symposium on Asynchronous Circuits and Systems*, pages 132–140, 2003.

[BV06] E. Beigne and P. Vivet. Design of on-chip and off-chip interfaces for a GALS NoC architecture. In *12th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 183–193, March 2006.

[CC13] F. C. Cheng and C. Chen. Can QDI Combinational Circuits be Implemented without C-elements? In *IEEE 19th International Symposium on Asynchronous Circuits and Systems*, pages 134–141, May 2013.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[CD03] W. S. Coates and R. J. Drost. Congestion and starvation detection in ripple fifos. In *Ninth International Symposium on Asynchronous Circuits and Systems*, pages 36–45, May 2003.

[Cha84] Daniel M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems.* PhD thesis, Stanford University, Oct. 1984.

[Chu87] Tam-Anh Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications.* PhD thesis, MIT Laboratory for Computer Science, June 1987.

[CJN10] M. Cannizzaro, Weiwei Jiang, and S.M. Nowick. Practical completion detection for 2-of-N delay-insensitive codes. In *IEEE International Conference on Computer Design (ICCD)*, pages 151–158, 2010.

[CN04] T. Chelcea and S. M. Nowick. Robust interfaces for mixed-timing systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(8):857–873, Aug. 2004.

[DDC05] Y. S. Dhillon, A. U. Diril, and A. Chatterjee. Soft-error tolerance analysis and optimization of nanometer circuits. In *Design, Automation and Test in Europe*, pages 288–293, March 2005.

[DGY92] I. David, R. Ginosar, and M. Yoeli. An efficient implementation of Boolean functions as self-timed circuits. *IEEE Transactions on Computers*, 41(1):2–11, 1992.

[DGY95] Ilana David, Ran Ginosar, and Michael Yoeli. Self-timed is self-checking. *Journal of Electronic Testing*, 6(2):219–228, April 1995.

[DIBM03] M. Donno, A. Ivaldi, L. Benini, and E. Macii. Clock-tree power optimization based on RTL clock-gating. In *Design Automation Conference, 2003. Proceedings*, pages 622–627, June 2003.

[Dil88] David L. Dill. *Trace Theory for Systematic Verification of Speed-Independent Circuits.* PhD thesis, Carnegy Mellon University, 1988.

[DLD$^+$14] M. Davies, A. Lines, J. Dama, A. Gravel, R. Southworth, G. Dimou, and P. Beerel. A 72-Port 10G Ethernet Switch/Router Using Quasi-Delay-Insensitive Asynchronous Design. In *20th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 103–104, May 2014.

[DS09] Jia Di and Scott C. Smith. *Designing Asynchronous Circuits Using NULL Convention Logic (NCL).* Morgan and Claypool Publishers, 2009.

[DWD91] Mark E. Dean, Ted E. Williams, and David L. Dill. Efficient self-timing with level-encoded 2-phase dual-rail (ledr). In *Proceedings of the 1991 University of California/Santa Cruz Conference on Advanced Research in VLSI*, page 55–70, Cambridge, MA, USA, 1991. MIT Press.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

[Fan05] Karl M. Fant. *Logically Determined Design – Clockless System Design with NULL Convention Logic™*. John Wiley & Sons, Ltd, 2005.

[FD96] S.B. Furber and P. Day. Four-phase micropipeline latch control circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4(2):247–253, June 1996.

[FFL18] S. Friedrichs, M. Függer, and C. Lenzen. Metastability-Containing Circuits. *IEEE Transactions on Computers*, 67(8):1167–1183, Aug. 2018.

[FMRM10] M. Fattah, A. Manian, A. Rahimi, and S. Mohammadi. A High Throughput Low Power FIFO Used for GALS NoC Buffers. In *IEEE Computer Society Annual Symposium on VLSI*, pages 333–338, 2010.

[Fri01] E.G. Friedman. Clock distribution networks in synchronous digital integrated circuits. *Proceedings of the IEEE*, 89(5):665–692, May 2001.

[FS09] W. Friesenbichler and A. Steininger. Soft Error Tolerant Asynchronous Circuits Based on Dual Redundant Four State Logic. In *12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*, pages 100–107, Aug. 2009.

[Gin11] R. Ginosar. Metastability and synchronizers: A tutorial. *IEEE Design Test of Computers*, 28(5):23–35, 2011.

[GYB07] K. T. Gardiner, A. Yakovlev, and A. Bystrov. A C-element Latch Scheme with Increased Transient Fault Tolerance for Asynchronous Circuits. In *13th IEEE International On-Line Testing Symposium (IOLTS 2007)*, pages 223–230, July 2007.

[HLS16] Florian Huemer, Jakob Lechner, and Andreas Steininger. A new Coding Scheme for Fault-Tolerant 4-Phase Delay-Insensitive Codes. In *IEEE 34th International Conference on Computer Design (ICCD)*, pages 392–395, Oct. 2016.

[HNS20] Florian Huemer, Robert Najvirt, and Andreas Steininger. Identification and Confinement of Fault Sensitivity Windows in QDI Logic. In *Austrochip Workshop on Microelectronics (Austrochip)*, pages 29–36, Oct. 2020.

[HNS22] Florian Huemer, Robert Najvirt, and Andreas Steininger. On SAT-Based Model Checking of Speed-Independent Circuits. In *IEEE 25th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 100–105, April 2022.

[HPL^{+}16] W. Ho, A. A. Pammu, N. Liu, K. Z. L. Ne, K. Chong, and B. Gwee. Security analysis of asynchronous-logic QDI cell approach for differential power analysis attack. In *International Symposium on Integrated Circuits (ISIC)*, pages 1–4, Dec. 2016.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[HS18a] Florian Huemer and Andreas Steininger. Advanced Delay-Insensitive 4-Phase Protocols. In *Austrochip Workshop on Microelectronics (Austrochip)*, pages 50–55, Sep. 2018.

[HS18b] Florian Huemer and Andreas Steininger. Partially Systematic Constant-Weight Codes for Delay-Insensitive Communication. In *24th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 17–25, May 2018.

[HS19] Florian Huemer and Andreas Steininger. Novel Approaches for Efficient Delay-Insensitive Communication. *Journal of Low Power Electronics and Applications*, 9(2), 2019.

[HS20a] Florian Huemer and Andreas Steininger. Sorting Network based Full Adders for QDI Circuits. In *Austrochip Workshop on Microelectronics (Austrochip)*, pages 21–28, Oct. 2020.

[HS20b] Florian Huemer and Andreas Steininger. Timing Domain Crossing using Muller Pipelines. In *26th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 44–53, May 2020.

[HSS15] Florian Huemer, Markus Schütz, and Andreas Steininger. Revisiting Sorting Network Based Completion Detection for 4 Phase Delay Insensitive Codes. In *Austrian Workshop on Microelectronics (Austrochip)*, pages 3–8, 2015.

[Huf54] David A. Huffman. The synthesis of sequential switching circuits. *Journal of the Franklin Institute*, 257(3):161–190,275–303, March/April 1954.

[IW10] K. A. Schouhamer Immink and J. H. Weber. Very efficient balanced codes. *IEEE Journal on Selected Areas in Communications*, 28(2):188–192, Feb. 2010.

[JM05] Wonjin Jang and A. J. Martin. SEU-tolerant QDI circuits [quasi delay-insensitive asynchronous circuits]. In *11th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 156–165, March 2005.

[JN07] C. Jeong and S. M. Nowick. Optimization of Robust Asynchronous Circuits by Local Input Completeness Relaxation. In *Asia and South Pacific Design Automation Conference*, pages 622–627, Jan. 2007.

[JN08] C. Jeong and S. M. Nowick. Technology Mapping and Cell Merger for Asynchronous Threshold Networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(4):659–672, April 2008.

[KC87] L. Kleeman and A. Cantoni. Metastable Behavior in Digital Systems. *IEEE Design Test of Computers*, 4(6):4–19, 1987.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[KFK15] B. Keller, M. Fojtik, and B. Khailany. A Pausible Bisynchronous FIFO for GALS Systems. In *21st IEEE International Symposium on Asynchronous Circuits and Systems*, pages 1–8, May 2015.

[KHS+20] F. A. Kuentzer, M. Herrera, O. Schrape, P. A. Beerel, and M. Krstic. Radiation Hardened Click Controllers for Soft Error Resilient Asynchronous Architectures. In *26th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 78–85, May 2020.

[KK20] F. A. Kuentzer and M. Krstic. Soft Error Detection and Correction Architecture for Asynchronous Bundled Data Designs. *IEEE Transactions on Circuits and Systems I: Regular Papers*, pages 1–12, 2020.

[KKY98] A. Kondratyev, M. Kishinevsky, and A. Yakovlev. Hazard-free implementation of speed-independent circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(9):749–771, 1998.

[KKY06] Victor Khomenko, Maciej Koutny, and Alex Yakovlev. Logic synthesis for asynchronous circuits based on STG unfoldings and incremental SAT. *Fundamenta Informaticae*, 70(1, 2):49–73, 2006.

[KL02] A. Kondratyev and K. Lwin. Design of Asynchronous Circuits by Synchronous CAD Tools. In *Proceedings 2002 Design Automation Conference (IEEE Cat. No.02CH37324)*, pages 411–414, June 2002.

[KMM15] S. Keller, A. J. Martin, and C. Moore. DD1: A QDI, Radiation-Hard-by-Design, Near-Threshold 18uW/MIPS Microcontroller in 40nm Bulk CMOS. In *21st IEEE International Symposium on Asynchronous Circuits and Systems*, pages 37–44, May 2015.

[KNR+02] A. Kondratyev, L. Neukom, O. Roig, A. Taubin, and K. Fant. Checking delay-insensitivity: 10^4 gates and beyond. In *Eighth International Symposium on Asynchronous Circuits and Systems*, pages 149–157, April 2002.

[Knu86] D. E. Knuth. Efficient balanced codes. *IEEE Transactions on Information Theory*, 32(1):51–53, Jan. 1986.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Edition) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[KSS+16] E. Kasapaki, M. Schoeberl, R. B. Sørensen, C. Müller, K. Goossens, and J. Sparsø. Argo: A Real-Time Network-on-Chip Architecture With an Efficient GALS Implementation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(2):479–492, Feb. 2016.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[KZYD10] W. Kuang, P. Zhao, J. S. Yuan, and R. F. DeMara. Design of Asynchronous Circuits for High Soft Error Tolerance in Deep Submicrometer CMOS Circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 18(3):410–422, March 2010.

[LG01] D.W. Lloyd and J.D. Garside. A practical comparison of asynchronous design styles. In *Seventh International Symposium on Asynchronus Circuits and Systems*, pages 36–45, 2001.

[LHCG17] J. Lim, W. Ho, K. Chong, and B. Gwee. DPA-resistant QDI dual-rail AES S-Box based on power-balanced weak-conditioned half-buffer. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, May 2017.

[Lin03] A. Lines. Nexus: an asynchronous crossbar interconnect for synchronous system-on-chip designs. In *11th Symposium on High Performance Interconnects, 2003. Proceedings.*, pages 2–9, Aug. 2003.

[LKM10] Dong-Jin Lee, Myung-Chul Kim, and Igor L. Markov. Low-power clock trees for cpus. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 444–451, Nov. 2010.

[LM04] C. LaFrieda and R. Manohar. Fault detection and isolation techniques for quasi delay-insensitive circuits. In *International Conference on Dependable Systems and Networks, 2004*, pages 41–50, June 2004.

[LSH15] Jakob Lechner, Andreas Steininger, and Florian Huemer. Methods for Analysing and Improving the Fault Resilience of Delay-Insensitive Codes. In *33rd IEEE International Conference on Computer Design (ICCD)*, pages 519–526, 2015.

[MAMN08] P.B. McGee, M.Y. Agyekum, M.A. Mohamed, and S.M. Nowick. A Level-Encoded Transition Signaling Protocol for High-Throughput Asynchronous Global Communication. In *14th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 116–127, 2008.

[Mar89] Alain J. Martin. Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits. 1989.

[Mar90] Alain J. Martin. *The Limitations to Delay-Insensitivity in Asynchronous Circuits*, pages 302–311. Springer New York, New York, NY, 1990.

[Mar91] Alain J. Martin. Synthesis of asynchronous VLSI circuits. 1991.

[MBSC18] M. T. Moreira, P. A. Beerel, M. L. L. Sartori, and N. L. V. Calazans. NCL Synthesis With Conventional EDA Tools: Technology Mapping and Optimization. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65(6):1981–1993, June 2018.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[MG20] M. T. Moreira and S. Giaconi. Chronos link: A qdi interconnect for modern socs. In *26th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 67–68, May 2020.

[MM15] R. Manohar and Y. Moses. Analyzing Isochronic Forks with Potential Causality. In *2015 21st IEEE International Symposium on Asynchronous Circuits and Systems*, pages 69–76, May 2015.

[MMC14] Matheus Trevisan Moreira, Fernando Gehm Moraes, and Ney Laert Vilar Calazans. Beware the dynamic c-element. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(7):1644–1647, July 2014.

[MMK+02] Y. Massoud, S. Majors, J. Kawa, T. Bustami, D. MacMillen, and J. White. Managing on-chip inductive effects. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10(6):789–798, Dec. 2002.

[MOMC12] Matheus Moreira, Bruno Oliveira, Fernando Moraes, and Ney Calazans. Impact of c-elements in asynchronous circuits. In *Thirteenth International Symposium on Quality Electronic Design (ISQED)*, pages 437–343, March 2012.

[MOPC13] M. T. Moreira, C. H. M. Oliveira, R. C. Porto, and N. L. V. Calazans. Ncl+: Return-to-one null convention logic. In *IEEE 56th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 836–839, Aug. 2013.

[MR07] M. Marshall and G. Russell. A Low Power Information Redundant Concurrent Error Detecting Asynchronous Processor. In *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, pages 649–656, Aug. 2007.

[MRL04] Y. Monnet, M. Renaudin, and R. Leveugle. Asynchronous circuits sensitivity to fault injection. In *10th IEEE International On-Line Testing Symposium*, pages 121–126, July 2004.

[MRL05] Y. Monnet, M. Renaudin, and R. Leveugle. Hardening techniques against transient faults for asynchronous circuits. In *11th IEEE International On-Line Testing Symposium*, pages 129–134, July 2005.

[MRL07] Y. Monnet, M. Renaudin, and R. Leveugle. Formal Analysis of Quasi Delay Insensitive Circuits Behavior in the Presence of SEUs. In *13th IEEE International On-Line Testing Symposium*, pages 113–120, July 2007.

[MTMR02] S. Moore, G. Taylor, R. Mullins, and P. Robinson. Point to point GALS interconnect. In *Proceedings Eighth International Symposium on Asynchronous Circuits and Systems*, pages 69–75, April 2002.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

[Mul59] David Muller. A Theory of Asynchronous Circuits. In *Proceedings of the International Symposium on Theory of Switching*, volume 29, pages 204–243, 1959.

[NFG+13] Javier Navaridas, Steve Furber, Jim Garside, Xin Jin, Mukaram Khan, David Lester, Mikel Luján, José Miguel-Alonso, Eustace Painkras, Cameron Patterson, Luis A. Plana, Alexander Rast, Dominic Richards, Yebin Shi, Steve Temple, Jian Wu, and Shufan Yang. SpiNNaker: Fault tolerance in a power- and area- constrained large-scale neuromimetic architecture. *Parallel Computing*, 39(11):693–708, 2013.

[Now96] S. M. Nowick. Design of a low-latency asynchronous adder using speculative completion. *IEE Proceedings - Computers and Digital Techniques*, 143(5):301–307, Sep. 1996.

[PD08] Sudeep Pasricha and Nikil Dutt. *On-Chip Communication Architectures: System on Chip Interconnect.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten.* PhD thesis, Technische Hochschule Darmstadt, 1962.

[Pet77] James L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–252, Sep. 1977.

[PFT+07] L. A. Plana, S. B. Furber, S. Temple, M. Khan, Y. Shi, J. Wu, and S. Yang. A GALS Infrastructure for a Massively Parallel Multiprocessor. *IEEE Design Test of Computers*, 24(5):454–463, Sep. 2007.

[PG07] I. Miro Panades and A. Greiner. Bi-Synchronous FIFO for Synchronous Circuit Communication Well Suited for Network-on-Chip in GALS Architectures. In *First International Symposium on Networks-on-Chip (NOCS'07)*, pages 83–94, May 2007.

[Pie98] S. J. Piestrak. Membership test logic for delay-insensitive codes. In *Proceedings Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 194–204, March 1998.

[PKY09] Ivan Poliakov, Victor Khomenko, and Alex Yakovlev. Workcraft – a framework for interpreted graph models. In Giuliana Franceschinis and Karsten Wolf, editors, *Applications and Theory of Petri Nets*, pages 333–342, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[PM05] Song Peng and R. Manohar. Efficient failure detection in pipelined asynchronous circuits. In *20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'05)*, pages 484–493, 2005.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[PS13] P. Palangpour and S. C. Smith. Sleep Convention Logic using partially slept function blocks. In *IEEE 56th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 17–20, Aug. 2013.

[PtBdWM10] Ad Peeters, Frank te Beest, Mark de Wit, and Willem Mallon. Click elements: An implementation style for data-driven compilation. In *IEEE Symposium on Asynchronous Circuits and Systems*, pages 3–14, May 2010.

[RMGW09] E. Rotem, A. Mendelson, R. Ginosar, and U. Weiser. Multiple clock and Voltage Domains for chip multi processors. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 459–468, Dec. 2009.

[RY85] L. Ya. Rosenblum and A. V. Yakovlev. Signal graphs: from self-timed to timed ones. In *International Workshop on Timed Petri Nets*, 1985.

[SEE98] M. Shams, J. C. Ebergen, and M. I. Elmasry. Modeling and comparing CMOS implementations of the C-element. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 6(4):563–567, Dec. 1998.

[SF98] G. E. Sobelman and K. Fant. CMOS circuit design of threshold gates with hysteresis. In *Proceedings of the 1998 IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 2, pages 61–64, May 1998.

[SF01] I. Sutherland and S. Fairbanks. GasP: a minimal FIFO control. In *Proceedings Seventh International Symposium on Asynchronous Circuits and Systems*, pages 46–53, 2001.

[SFGP09] Yebin Shi, S.B. Furber, J. Garside, and L.A. Plana. Fault Tolerant Delay Insensitive Inter-chip Communication. In *15th IEEE Symposium on Asynchronous Circuits and Systems*, pages 77–84, May 2009.

[SKM+15] Danil Sokolov, Victor Khomenko, Andrey Mokhov, Alex Yakovlev, and David Lloyd. Design and verification of speed-independent multiphase buck controller. In *21st IEEE International Symposium on Asynchronous Circuits and Systems*, pages 29–36, 2015.

[SKYL18] D. Sokolov, V. Khomenko, A. Yakovlev, and D. Lloyd. Design and verification of speed-independent circuits with arbitration in workcraft. In *24th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 30–31, May 2018.

[SMB+02] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *Proceedings Eighth International Symposium on High Performance Computer Architecture*, pages 29–40, Feb. 2002.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

[SMBY05] D. Sokolov, J. Murphy, A. Bystrov, and A. Yakovlev. Design and analysis of dual-rail circuits for security applications. *IEEE Transactions on Computers*, 54(4):449–460, 2005.

[Smi02] S. C. Smith. Speedup of self-timed digital systems using Early Completion. In *Proceedings IEEE Computer Society Annual Symposium on VLSI. New Paradigms for VLSI Systems Design. ISVLSI*, pages 98–104, 2002.

[SN07] M. Singh and S. M. Nowick. MOUSETRAP: High-Speed Transition-Signaling Asynchronous Pipelines. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(6):684–698, 2007.

[Spa20] Jens Sparsø. *Introduction to Asynchronous Circuit Design.* DTU Compute, Technical University of Denmark, 2020. Paperback edition available here: https://www.amazon.com/dp/B08BF2PFLN.

[SPY07] Danil Sokolov, Ivan Poliakov, and Alex Yakovlev. Asynchronous data path models. In *Seventh International Conference on Application of Concurrency to System Design (ACSD 2007)*, pages 197–210, July 2007.

[SS93] Jens Sparsø and Jørgen Staunstrup. Delay-insensitive multi-ring structures. *Integration, the VLSI Journal*, 15(3):313–340, 1993. Special Issue on asynchronous systems.

[Sut89] I. E. Sutherland. Micropipelines. *Commun. ACM*, 32(6):720–738, June 1989.

[TBV09] Y. Thonnart, E. Beigné, and P. Vivet. Design and Implementation of a GALS Adapter for ANoC Based Architectures. In *15th IEEE Symposium on Asynchronous Circuits and Systems*, pages 13–22, 2009.

[TGL07] P. Teehan, M. Greenstreet, and G. Lemieux. A Survey and Taxonomy of GALS Design Styles. *IEEE Design Test of Computers*, 24(5):418–428, Sep. 2007.

[TM18] G. Tarawneh and A. Mokhov. Formal verification of mixed synchronous asynchronous systems using industrial tools. In *24th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 43–50, May 2018.

[Tom06] William Benjamin Toms. *Synthesis of quasi-delay-insensitive datapath circuits.* PhD thesis, Department of Computer Science, University of Manchester, 2006.

[vB92] Kees van Berkel. Beware the isochronic fork. *Integr. VLSI J.*, 13(2):103–128, June 1992.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[Ver88] Tom Verhoeff. Delay-insensitive codes — an overview. *Distributed Computing*, 3(1):1–8, March 1988.

[VM02] T. Verdel and Y. Makris. Duplication-based concurrent error detection in asynchronous circuits: shortcomings and remedies. In *Proceedings of the 17th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, pages 345–353, 2002.

[VSB10] Santosh N. Varanasi, Kenneth S. Stevens, and Graham Birtwistle. Concurrency reduction of untimed latch protocols - theory and practice. In *IEEE Symposium on Asynchronous Circuits and Systems*, pages 26–37, 2010.

[Wat17] Des Watson. *A Practical Approach to Compiler Construction*. Springer, 2017.

[WH11] Neil H. E Weste and (author.) Harris, David Money. *Integrated circuit design*. Boston : Pearson, fourth edition, global edition edition, 2011. Previous ed.: 2005.

[Wol] Claire Wolf. Yosys open synthesis suite. `https://yosyshq.net/yosys/`.

[Yak93] A. Yakovlev. Structural technique for fault-masking in asynchronous interfaces. *IEEE Proceedings - Computers and Digital Techniques*, 140(2):81–91, March 1993.

[YBA96] K. Y. Yun, P. A. Beerel, and J. Arceo. High-performance asynchronous pipeline circuits. In *Proceedings Second International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 17–28, March 1996.

[YCCP04] Jing-Ling Yang, Chiu-Sing Choy, Cheong-Fat Chan, and Kong-Pong Pun. A high-efficiency strongly self-checking asynchronous datapath. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(10):1484–1494, Oct. 2004.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Curriculum Vitae

Florian Huemer

Personal Information

Nationality	Austrian
Email	fhuemer@ecs.tuwien.ac.at
ORCID	0000-0002-2776-7768

Education

2003-2008	Technical college (telecommunication), HTL Mössingerstraße, Klagenfurt, Austria
2009-2013	Bachelor's degree Computer Engineering, TU Wien, Austria
2013-2017	Master's degree Computer Engineering, TU Wien, Austria, Master Thesis: *"Protecting 4-Phase Delay-Insensitive Communication Against Transient Faults"*
2017-2022	PhD Student at the Institute of Computer Engineering, TU Wien, Austria

Professional Experience

2011-2017 (winter terms)	Tutor at the Institute of Computer Engineering (TU Wien) for digital design related courses
2014-2017	Student researcher at the Institute of Production Engineering and Photonic Technologies (TU Wien)
2015/2016 (Jul - Sept)	Student assistant at the Institute of Computer Engineering (TU Wien) for digital design related courses
2017-2022	University/project assistant at the Institute of Computer Engineering (TU Wien)

Academic Achievements/Awards

2016	Distinguished Young Alumnus Award, TU Wien (Master Thesis Award)
2017	Best Paper Award, IEEE International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)
2021	Best Paper Award, IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)

Community Services

2015	Publication Chair for the Austrian Workshop on Microelectronics (Austrochip)

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

since 2016	Reviewing for various conferences (DDECS, Austrochip, ASYNC, DSD)
2018	Local Organization Chair for the IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)
2021	Local Organization Chair for the IEEE International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)

Journal Papers (peer-revieved)

1. F. Huemer and A. Steininger. Novel Approaches for Efficient Delay-Insensitive Communication. *Journal of Low Power Electronics and Applications*, 9(2), 2019. ISSN 2079-9268. doi: 10.3390/jlpea9020016.
2. T. Polzer, F. Huemer, and A. Steininger. An Experimental Study of Metastability-Induced Glitching Behavior. *Journal of Circuits, Systems and Computers*, 28, Suppl. 1 (2019) 1940006, 2019. doi: 10.1142/S0218126619400061.
3. T. Polzer, F. Huemer, and A. Steininger. Refined metastability characterization using a time-to-digital converter. *Microelectronics Reliability*, 80:91–99, 2018. ISSN 0026-2714. doi: 10.1016/j.microrel.2017.11.017.
4. F. Huemer, M. Jamalieh, F. Bammer, and D. Hönig. Inline imaging-ellipsometer for printed electronics. *tm - Technisches Messen*, 83(10):549–556, 2016. doi: https://doi.org/10.1515/teme-2015-0067.

Conference Papers (peer-revieved)

5. F. Huemer, R. Najvirt, and A. Steininger. On SAT-Based Model Checking of Speed-Independent Circuits. In *IEEE 25th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 100–105, April 2022. doi: 10.1109/DDECS54261.2022.9770165.
6. P. Behal, F. Huemer, R. Najvirt, and A. Steininger. An Automated Setup for Large-Scale Simulation-Based Fault-Injection Experiments on Asynchronous Digital Circuits. In *24th Euromicro Conference on Digital System Design (DSD)*, pages 541–548, Sep. 2021. doi: 10.1109/DSD53832.2021.00087.
7. P. Behal, F. Huemer, R. Najvirt, A. Steininger, and Z. Tabassam. Towards Explaining the Fault Sensitivity of Different QDI Pipeline Styles. In *27th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 25–33, 2021. doi: 10.1109/ASYNC48570.2021.00012.
8. F. Huemer, R. Najvirt, and A. Steininger. Identification and Confinement of Fault Sensitivity Windows in QDI Logic. In *Austrochip Workshop on Microelectronics (Austrochip)*, pages 29–36, Oct. 2020. doi: 10.1109/Austrochip51129.2020.9232985.
9. F. Huemer and A. Steininger. Sorting Network based Full Adders for QDI Circuits. In *Austrochip Workshop on Microelectronics (Austrochip)*, pages 21–28, Oct. 2020. doi: 10.1109/Austrochip51129.2020.9232987.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

10. F. Huemer and A. Steininger. Timing Domain Crossing using Muller Pipelines. In *26th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 44–53, May 2020. doi: 10.1109/ASYNC49171.2020.00014.

11. M. Schütz, A. Steininger, F. Huemer, and J. Lechner. State Recovery for Coarse-Grain TMR Designs in FPGAs Using Partial Reconfiguration. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–6, Oct. 2018. doi: 10.1109/DFT.2018.8602984.

12. F. Huemer and A. Steininger. Advanced Delay-Insensitive 4-Phase Protocols. In *Austrochip Workshop on Microelectronics (Austrochip)*, pages 50–55, Sep. 2018. doi: 10.1109/Austrochip.2018.8520702.

13. F. Huemer and A. Steininger. Partially Systematic Constant-Weight Codes for Delay-Insensitive Communication. In *24th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 17–25, May 2018. doi: 10.1109/ASYNC.2018.00014.

14. F. Huemer, T. Polzer, and A. Steininger. Using a Duplex Time-to-Digital Converter for Metastability Characterization of an FPGA. In *IEEE 21st International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 141–146, April 2018. doi: 10.1109/DDECS.2018.00032.

15. T. Polzer, F. Huemer, and A. Steininger. Measuring Metastability using a Time-to-Digital Converter. In *IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 116–121, 2017. doi: 10.1109/DDECS.2017.7934582.

16. T. Polzer, F. Huemer, and A. Steininger. A Programmable Delay Line for Metastability Characterization in FPGAs. In *Austrochip Workshop on Microelectronics (Austrochip)*, pages 51–56, 2016. doi: 10.1109/Austrochip.2016.021.

17. A. Kinali, F. Huemer, and C. Lenzen. Fault-Tolerant Clock Synchronization with High Precision. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 490–495, 2016. doi: 10.1109/ISVLSI.2016.88.

18. F. Huemer, J. Lechner, and A. Steininger. A new Coding Scheme for Fault-Tolerant 4-Phase Delay-Insensitive Codes. In *IEEE 34th International Conference on Computer Design (ICCD)*, pages 392–395, Oct. 2016. doi: 10.1109/ICCD.2016.7753311.

19. 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)*, pages 519–526, 2015. doi: 10.1109/ICCD.2015.7357160.

20. M. Schütz, F. Huemer, and A. Steininger. A Practical Comparison of 2-Phase Delay Insensitive Communication Protocols. In *Austrian Workshop on Microelectronics (Austrochip)*, pages 15–20, 2015. doi: 10.1109/Austrochip.2015.11.

21. 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)*, pages 3–8, 2015. doi: 10.1109/Austrochip.2015.16.