HAZARD-FREE CLOCK SYNCHRONIZATION

A dissertation submitted towards the degree Doctor of Engineering of the Faculty of Mathematics and Computer Science of Saarland University

by Johannes Bund
Saarbrücken, 2022

Day of Colloquium:	28.02.2023
Dean of the Faculty:	Univ.-Prof. Dr. Jürgen Steimle
Chair of the Committee:	Prof. Dr. Markus Bläser
Reporters	
First reviewer:	Dr. Christoph Lenzen
Second reviewer:	Prof. Dr. Andreas Steininger
Third reviewer:	Dr. Matthias Függer
Fourth reviewer:	Prof. Dr. Kurt Mehlhorn
Academic Assistant:	Dr. Roohani Sharma

Abstract

The growing complexity of microprocessors makes it infeasible to distribute a single clock source over the whole processor with a small clock skew. Hence, chips are split into multiple clock regions, each covered by a single clock source. This poses a problem for communication between these clock regions. Clock synchronization algorithms promise an advantage over state-of-the-art solutions, such as GALS systems. When clock regions are synchronous the communication latency improves significantly over handshake-based solutions. We focus on the implementation of clock synchronization algorithms.

A major obstacle when implementing circuits on clock domain crossings are hazardous signals. We can formally define hazards by extending the Boolean logic by a third value $\mathfrak{u}$. In this thesis, we describe a theory for designing and analyzing hazard-free circuits. We develop strategies for hazard-free encoding and construction of hazard-free circuits from finite state machines. Furthermore, we discuss clock synchronization algorithms and a possible combination of them.

In the end, we present two implementations of the GCS algorithm by Lenzen, Locher and Wattenhofer [67]. We prove by rigorous analysis that the systems implement the algorithm. The theory described above is used to prove that our clock synchronization circuits are hazard-free (in the sense that they compute the most precise output possible). Simulation of our GCS system shows that it achieves a skew between neighboring clock regions that is smaller than a few inverter delays.

Zusammenfassung

Aufgrund der zunehmenden Komplexität von Mikroprozessoren ist es unmöglich, mit einer einzigen Taktquelle den gesamten Prozessor ohne großen Versatz zu takten. Daher werden Chips in mehrere Regionen aufgeteilt, die jeweils von einer einzelnen Taktquelle abgedeckt werden. Dies stellt ein Problem für die Kommunikation zwischen diesen Taktregionen dar. Algorithmen zur Taktsynchronisation bieten einen Vorteil gegenüber aktuellen Lösungen, wie z.B. GALS-Systemen. Synchronisiert man die Taktregionen, so verbessert sich die Latenz der Kommunikation erheblich.

In Schaltkreisen zwischen zwei Taktregionen können undefinierte Signale, sogenannte Hazards auftreten. Indem wir die boolesche Algebra um einen dritten Wert $\mathfrak{u}$ erweitern, können wir diese Hazards formal definieren. In dieser Arbeit zeigen wir eine Methode zum Entwurf und zur Analyse von hazard-freien Schaltungen. Wir entwickeln Strategien für Kodierungen die Hazards vermeiden und zur Konstruktion von hazard-freien Schaltungen. Darüber hinaus stellen wir Algorithmen Taktsynchronisation vor und wie diese kombiniert werden können.

Zum Schluss stellen wir zwei Implementierungen des GCS-Algorithmus von Lenzen, Locher und Wattenhofer [67] vor. Oben genannte Mechanismen werden verwendet, um formal zu beweisen, dass diese Implementierungen korrekt sind. Die Implementierung hat keine Hazards, das heißt sie berechnet die bestmögliche Ausgabe. Anschließende Simulation der GCS Implementierung erzielt einen Versatz zwischen benachbarten Taktregionen, der kleiner als ein paar Gatter Laufzeiten ist.

Publications

This dissertation summarizes the work carried out during my doctoral studies at the Max-Planck Institute for Informatics and CISPA Helmholtz Center for Information Security. The work is published in the following research papers (sorted by the order of appearance in this thesis).

[17] Bund, J., Lenzen, C., and Medina, M. Small hazard-free transducers. In *13th Innovations in Theoretical Computer Science Conference, ITCS 2022, January 31 - February 3, 2022, Berkeley, CA, USA* (2022), M. Braverman, Ed., vol. 215 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 32:1–32:24

[16] Bund, J., Lenzen, C., and Medina, M. Optimal metastability-containing sorting via parallel prefix computation. *IEEE Trans. Computers 69*, 2 (2020), 198–211

[18] Bund, J., Lenzen, C., and Rosenbaum, W. Fault tolerant gradient clock synchronization. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019* (2019), P. Robinson and F. Ellen, Eds., ACM, pp. 357–365

[12] Bund, J., Függer, M., Lenzen, C., and Medina, M. Synchronizer-free digital link controller. *IEEE Trans. Circuits Syst. 67-I*, 10 (2020), 3562–3573

[13] Bund, J., Függer, M., Lenzen, C., Medina, M., and Rosenbaum, W. PALS: plesiochronous and locally synchronous systems. In *26th IEEE International Symposium on Asynchronous Circuits and Systems, ASYNC 2020, Salt Lake City, UT, USA, May 17-20, 2020* (2020), IEEE, pp. 36–43

In parts, the presented work is an extension of the published work. At the beginning of each chapter, I give more detail about the individual content.

Figures and graphics used in this dissertation are either taken from the stated research papers or newly created for this thesis, with one exception: Figure 2.2 was created by Ian W. Jones to help me visualize the behavior of metastable latches.

Remark. The research paper [16] is a major part of my master's thesis. It is not regarded as a contribution to achieving the doctoral degree. The research paper is included in this thesis as it is an essential part of this line of work.

Contribution. All authors contributed equally to the research papers listed above. In the following, I state my contribution in more detail. In general, I contributed to circuit design, simulation, and evaluation. However, in most works, I was also involved in correctness proofs and write-up.

[18] The work presents a purely theoretical result. I was involved in discussing the results and proofs, as well as proofreading the write-up.

[16, 12] Both works present theoretical constructions of circuits which are supported by extensive computer simulations. I was involved in developing the circuits and proving them correct. Largely, I contributed to the implementation and simulation of the circuits, including subsequent evaluation of the experiments.

[13] The work discusses the implementation of an algorithm, starting from a description of the algorithm and circuit, to correctness the of the circuit, to extensive simulations. I contributed to developing the circuits and formal proofs. As for [16, 12], I largely contributed to the implementation and simulation of the circuits, including subsequent evaluation of the experiments.

[17] The work presents a theoretical framework and its correctness proof. The idea was developed during the work on [16]. My contribution was identifying the problem, finding a solution, and proving the correctness of the construction together with my co-authors.

Acknowledgments

My gratitude goes to all people that I met along the way to completing this important step, regardless of whether they are family, friends, colleagues, or unique encounters.

I appreciate every support that I received, thank you!

Contents

Abstract **iv**

Publications **vi**

Acknowledgments **ix**

1 Introduction **1**
1.1 Basic Terms 1
1.2 Contributions 4
1.3 Outline 5

2 Circuits and Hazards **7**
2.1 Basic Notation 7
2.2 Circuits and Kleene Logic 8
2.3 Hazard-Free Circuits 10
2.4 Implementation of Basic Gates 14
2.5 Example: The Hazard-Free Multiplexer 15
2.6 Related Work 19

3 Encoding **23**
3.1 Codes 23
3.2 Preserving and Recoverable Codes 24
3.3 Example Codes 29
3.4 Follow-Up Questions 34

4 Hazard-Free Transducers **35**
4.1 Introduction and Related Work 35
4.2 Classic PPC and Hazards 40
4.3 Hazard-Free PPC 44
4.4 Extension of the Input Encoding 59
4.5 Bound on k 61
4.6 Follow-Up Questions 63

5 Hazard-Free Sorting **65**
5.1 Introduction and Related Work 65
5.2 Sorting Networks 66
5.3 Comparator Specification 67
5.4 Sorting Transducer 70
5.5 Hazard-free Implementation 73
5.6 Simulation and Results 78

6 Clock Synchronization **79**
6.1 Model . . . 79
6.2 Problem . . . 80
6.3 Lynch-Welch Algorithm . . . 83
6.4 Gradient Clock Synchronization . . . 86

7 Fault Tolerant Gradient Clock Synchronization **97**
7.1 Introduction and Related Work . . . 97
7.2 Computational Model . . . 102
7.3 Cluster Algorithm . . . 104
7.4 Inter-cluster Algorithm . . . 106

8 Implementation of Clock Synchronization Algorithms **111**
8.1 Related Work . . . 111
8.2 Hardware Modules . . . 114

9 Single Link Synchronization **119**
9.1 Introduction and Related Work . . . 119
9.2 System Specification . . . 121
9.3 Continuous Threshold Controller . . . 125
9.4 Performance Evaluation . . . 132

10 Network Synchronization **139**
10.1 Introduction . . . 139
10.2 Hardware Modules . . . 140
10.3 Hardware Implementation . . . 148
10.4 Simulation and Comparison . . . 154
10.5 Follow-Up Questions . . . 164

11 Conclusions **167**
11.1 Summary . . . 167
11.2 Vision . . . 168

List of Acronyms **172**

Introduction 1

Famously, Gordon E. Moore stated in the year 1965 what later became known as Moore's law [79]:

The number of transistors on a microprocessor doubles every two years.

While this is not an actual law but an observation formulated decades ago, we see that this is still true to date. Transistor sizes on microchips are shrinking to low single-digit nanometer ranges, allowing more transistors in the same area.

Although it seems that the physical limitations on transistor sizes will be reached very soon, Moore's law is coming to an end only in a literal sense [95]. There are plenty of other possibilities to increase the computational power of microprocessors. As long as manufacturers keep increasing the functionality, Moore's law will continue from a consumer's point of view. A more up-to-date version of Moore's law could be: *The complexity of chips doubles every two years.* The "complexity of a chip" is kept vague on purpose. It includes various factors from power management and clock distribution to application-specific circuitry.

Due to the growth of complexity, one of the fundamental design principles becomes infeasible: Having a single clock source whose signal is distributed over the entire microprocessor by a clock tree. To understand the reasons behind this challenge we now provide a basic overview of clock distribution and its problems. On the following pages, we give a brief introduction to the basic terms. We cover historical background and current developments that help to contextualize the presented work. In particular, we discuss the problems of using a single clock source.

1.1 Basic Terms

Clocks and Registers. The clock on a microprocessor is the heart of a chip. It is an internal time reference that allows for synchronous computation and communication of data between different parts. Microprocessors operate in stages where the outcome of one stage is forwarded to the next. The clock signal is a reference that determines when the previous stage is finished, i.e., when data is ready for use. The time between two clock ticks needs to be larger than the time a stage needs for its computation.

an alternative design approach is asynchronous design

Registers are used to transport data from one stage to the next, they update their data on each clock beat. Registers are such a fundamental building block of microprocessors that can be found in every part of the chip. A clock signal needs to be propagated to every register, i.e., it needs to be propagated over the complete microprocessor chip. Due to their implementation, registers pose timing constraints on when the data in relation to the clock signal must arrive. In other words, for synchronous computation, the clock signal should arrive at each register at the same

time. A common design scheme that addresses this challenge is a *clock tree*. It distributes a clock signal over the complete microprocessor.

Clock Trees. A clock tree is a series of repeaters (buffers) and forks that connect one root to a set of leaves. The clock signal is fed to the root of the tree which then propagates the signal to its leaves where the registers are located. In an idealized model, all paths from the root to a leaf that have the same length also have the same propagation delay. Hence, a clock tree is perfect if each leaf has the same distance to the root. Then, in the ideal model, the clock signal arrives at the same time at each leaf. We call the difference in arrival time of the signal between two leaves *skew*.

Clock trees do not scale well to more complex designs. In contrast to the idealized model, in the real world, the possibility of building a perfect tree vanishes. With the increasing complexity of the chip, the layout process becomes more intricate. Also, variations in the manufacturing and application affect single path delays in the clock tree significantly. The design principle of having a single clock, distributed via a clock tree becomes infeasible for complex chips.

Remark. No layout of a single clock tree can achieve a small skew for large chips. In Section 10.4.2 we prove formally that the skew between neighbors grows proportional to the side length of a chip.

Multiple Clock Sources. A solution to the skew problem follows one of the standard design methodologies in computer science: *divide and conquer*. Split the chip into small regions that can be covered by a clock tree with negligible skew, then handle each region with its own clock source.

The approach brings many advantages such as modularity. Modularity allows the integration of new components as new clock regions without the need to compile a new clock tree. Clock regions can independently slow down for less power consumption or speed up for faster computation. If one clock source fails other clock regions can continue operating.

So far, however, regions are isolated and cannot communicate with each other by sending data. Clock sources of different regions are uncorrelated. No data or information can be transferred without either synchronizing the clocks or synchronizing the data. A popular design approach for communication between clock regions are globally asynchronous locally synchronous (GALS) systems [94].

GALS systems. The term has first been coined by Chapiro [22] in 1984. The basic idea is to keep clock regions isolated, but adding asynchronous communication on top. In GALS systems, local computation within a clock region is synchronous, and global communication between clock regions is asynchronous.

a synchronizer reduces the probability of reading corrupted data

Asynchronous communication requires the addition of data buffers and synchronizers on every communication link between two clock regions. Synchronizers will add a latency of two to three clock cycles to the data transfer. Depending on the

amount of data that needs to be transferred, buffers may add a significant overhead in cost to the chip.

GALS systems overcome the skew problem by surrendering to the scalability issues, small skew is maintained only on a manageable scale. Computations are synchronous only on a local level, globally the chip runs asynchronously. Thus, on a global scale, GALS systems lose the advantages of synchronous design such as timing guarantees and low latency communication. A more detailed discussion of GALS systems follows in Section 8.1.

Hardware on clock domain crossings faces a fundamental issue: *metastability* [21]. Data that passes from one clock domain to another may get corrupted when both domains have no timing relation. This is due to metastability in hardware registers.

Metastability. Storage elements such as registers store a digital value by changing their internal state to low voltage (logical 0) or high voltage (logical 1). If registers sample data that is in transition they may become metastable. This means that the register enters an unstable equilibrium state where its internal state is somewhere between high and low voltage. The register remains unstable until the state resolves to either high or low after some (unknown) time. This spurious mode of operation cannot be avoided when there are no timing guarantees [74]. Essentially, clock domain crossings cannot maintain timing guarantees, because different clock domains may have different clock speeds or different initialization.

Recent work on metastability-containment [41] showed that the uncertainty of metastable signals can be contained without further amplification. A circuit designed for metastability-containment can limit the instability of its outputs. Limiting this instability is not a trivial task. It will be discussed in this dissertation. First, we will see that the digital abstraction is not capable of modeling metastable behavior.

Remark. The term metastability-containment is replaced by the term hazard-freedom throughout this dissertation.

Digital Abstraction. Digital circuit design relies on a fundamental abstraction of the physical world. Electric voltages transmitted by wires are abstracted by Boolean values, where the supply voltage level corresponds to logical 1 (true) and low voltage level corresponds to logical 0 (false). By this abstraction, the behavior of digital circuitry can be described with Boolean algebra. However, the abstraction does not account for the behavior of digital circuits in all cases: It offers no way of representing unstable signals, transitioning, oscillating, etc.

There is a plethora of algebras that can model different kinds of signal behaviors [10]. In this work, we study a classic extension of Boolean logic due to Kleene [58], which allows for the presence of unspecified signals. We abstract all signal behaviors that do not fit into the Boolean abstraction by a new logic value $\mathfrak{u}$. Intuitively, $\mathfrak{u}$ models an unstable signal, i.e., a voltage level that is not sufficiently close to logic 1 or 0.

Remark. Value u differs from an unknown value which is stable but not known whether it is 0 or 1. The third logical value u can be regarded as the superposition of 0 and 1. When taking a copy of u both copies may resolve to different stable values. A formal discussion follows in Chapter 2.

Hazards. Kleene logic also defines the behavior of logic operators (such as `and`, `or`, and `not`) in the presence of logic value u. A circuit is an arrangement of logic gates that compute logic operators. Given a function on Boolean values, we can define an extension of this function to Kleene values. The extension adopts the output whenever a part of the inputs defines a stable output. A circuit that computes the Boolean function does not necessarily compute its Kleene extension. If a circuit computes the Boolean function but not the Kleene extension we call this a *hazard*. A formal definition follows in Chapter 2.

the extension is called the hazard-free extension

The use of Kleene logic to model hazards dates back to Huffman [51]. The most famous example of a circuit that computes the Boolean function but not the extension to Kleene logic is the multiplexer (`MUX`). We discuss the `MUX` and a `MUX` without hazards (the `CMUX`) in Section 2.5. To our knowledge the first occurrence of the `CMUX` dates back to Goto [46].

1.2 Contributions

In our work we aim at both: maintaining the modularity of GALS systems and providing advantages of globally synchronous systems. We keep the idea of separating the chip into local clock regions, but on top, we use clock synchronization algorithms (known from distributed computing) to synchronize different regions.

Implementation of these algorithms is highly non-trivial. Clearly, clock synchronization needs to be implemented in hardware. A software solution would be too slow to guarantee small skews because computation will take multiple clock cycles before we can adjust a clock region. A software solution requires functioning hardware upfront which is not necessarily given on clock domain crossings. We aim at a hardware solution, which also is subject to the fundamental issue of metastability.

The content of this thesis can be split into two major topics. We start with the theory of hazard-free circuits. In the second part, we show applications of hazard-free circuits in clock synchronization algorithms. We present designs that achieve low skew by implementing a clock synchronization algorithm.

Hazard-Free Circuits. The first part of this dissertation focuses on the design of circuits that have no hazards (so-called hazard-free or metastability-containing circuits). Hazard-free circuits compute the most precise outputs in the face of unstable inputs. A formal definition follows in Chapter 2.

The hazard-free circuits that we present throughout this dissertation can be split into two approaches:

general construction Given a formal specification of a circuit in Boolean logic, construct a hazard-free circuit that computes the hazard-free extension. Chapter 4 presents a general construction that produces a hazard-free circuit from any given specification in the form of a finite state transducer. We also pick up the question of how circuits and their hazard-free counterparts differ in their complexity. The question recently gained interest due to the work of Ikenmeyer et al. [52].

problem specific If the problem description allows for some adjustments to the circuit specification, then we have more freedom to find a hazard-free solution. In Chapter 5, e.g., we choose an input encoding that is robust against unstable signals in the sense that it does not amplify uncertainty (like other encodings). We then define how to compare and sort inputs in the specified encoding. Finally, we provide a hazard-free circuit for sorting these inputs. Also, the clock synchronization approaches presented in Chapters 9 and 10 fall into the category of problem-specific solutions.

In several instances, we see that the type of encoding chosen is important in Kleene logic. In Boolean logic, every encoding can be translated to any other encoding without loss of information. In Kleene logic, this is not possible for every encoding. Many encodings lose information (cf. Chapter 3), the choice of encoding matters.

Hazard-Free Clock Synchronization Algorithms. After introducing hazard-free circuits we come back to the implementation of clock synchronization algorithms in the second part. A circuit implementation eventually communicates with two or more clock domains. Digital circuits that operate on clock domain crossings may face unstable signals. Without sacrificing time to reduce the probability of unstable signals, we aim to compute the most precise clock adjustments in the presence of unstable inputs. The main research question we want to answer in this work is:

Can we design hazard-free circuits that implement clock synchronization algorithms?

In the final chapter of this dissertation, we present a design that implements a clock synchronization algorithm. It is proven to be hazard-free and maintains the worst-case skew bounds of the algorithm. The design can be adjusted to any system with multiple clock regions on a chip.

1.3 Outline

This dissertation is organized in the following way. The technical content follows after this introductory chapter, it concerns two major topics. First, Chapters 2 to 5 cover the theory of hazard-free circuits. Second, Chapters 6 to 10 cover clock synchronization and its implementation using hazard-free circuits.

Chapter 2 and Chapter 3 provide necessary background on hazards by giving formal definitions and going through examples. In Chapter 2, we define Kleene logic as an extension of Boolean logic. We use Kleene logic to model hazards in Boolean circuits. We define k-bit hazards and discuss one of the most prominent examples of circuits that have a hazard: the MUX.
The extension of the Boolean logic has an unfavorable side effect: the encoding we choose for data becomes important. In Chapter 3 we discuss why the standard binary encoding is infeasible for hazard-free circuits. We continue with a discussion on desirable properties of encodings in the Kleene logic and present different encodings.

Chapter 4 presents a general construction for hazard-free circuits that is based on a well-known construction for adder circuits. However, the standard encoding used in the construction is lossy in Kleene logic. We deploy a more elaborate encoding that achieves the required precision.

In Chapter 5, we show that the construction for hazard-free circuits can be applied to a primitive that is of importance for clock synchronization algorithms: sorting of two input numbers. Moreover, we can improve over the general construction and find an asymptotically optimal implementation of the hazard-free sorting primitive.

We define the model and problem of clock synchronization in Chapter 6, including the two clock synchronization algorithms that are used in this work; the Lynch-Welch algorithm and the OffsetGCS algorithm. Both algorithms have advantages and disadvantages in terms of fault-tolerance and network structure. Chapter 7 presents an approach to combine both algorithms into a hybrid algorithm that combines the best of both worlds.

Before we present two hardware implementations of the clock synchronization approach in Chapters 9 and 10, we discuss related work and basic building blocks used for both in Chapter 8. Chapter 9 then presents an implementation on a simple network structure, the sender-receiver link. It covers communication and synchronization on a single link between a producer and a consumer. Both share a data buffer that can help to synchronize both nodes. Chapter 9 presents a detailed description and simulation of the single link implementation.

Chapter 10 highlights the implementation and simulation of a clock synchronization algorithm on an arbitrary network. It represents the state-of-the-art answer to our research question for hazard-free clock synchronization on arbitrary networks.

We conclude this dissertation in Chapter 11 with a summary of our most important findings and an outlook on further research.

Circuits and Hazards 2

In this section, we describe basic notation and tools used in the study of hazard-free circuits and the complexity of hazard-free circuits. First, we walk the reader through the important definitions before we apply them in a demonstrative example. Finally, we discuss related work. We start with a collection of common notations.

2.1 Basic Notation

Natural Numbers. The set of positive integers including 0 is denoted by $\mathbb{N}_0$. Given an integer $M \in \mathbb{N}_0$ we denote by $[M]$ the set of all integers up to (not including) M,

$$[M] = \{0, \ldots, M-1\}.$$

For $i, j \in \mathbb{N}_0$, such that $i \leq j$, the *interval* $\langle i, j \rangle_M$ is a set of consecutive integers from $[M]$. Integers are consecutive in the sense that after $M-1$ we continue again at 0. The interval is defined by

$$\langle i, j \rangle_M := \{i \bmod M, \ldots, j \bmod M\}.$$

Power Set. The power set of a set is the set of all subsets. Given a set S, we denote the power set of S by $\mathcal{P}(S)$, it is defined by

$$\mathcal{P}(S) := \{S' | S' \subseteq S\}.$$

Bits and Words. The set of *binary* values is denoted by $\mathbb{B}$ and the set of *ternary* values is denoted by $\mathbb{T}$. For the third logical value, we use the symbol $\mathfrak{u}$.

$$\begin{aligned} \mathbb{B} &:= \{0, 1\} \text{ and} \\ \mathbb{T} &:= \{0, \mathfrak{u}, 1\}. \end{aligned}$$

The set of all binary (respectively ternary) words of length $\ell \in \mathbb{N}$ is denoted by $\mathbb{B}^\ell$ (respectively $\mathbb{T}^\ell$). For a word $x \in \mathbb{B}^\ell$ or $x \in \mathbb{T}^\ell$ and index $i \in \{1 \ldots \ell\}$, x_i denotes the ith bit of x. For $i, j \in \{1 \ldots \ell\}$, where $i \leq j$, we denote the substring of x from index i to j by $x_{i,j}$,

$$\begin{aligned} x &= x_1 x_2 \ldots x_\ell, \\ x_{i,j} &= x_i x_{i+1} \ldots x_j. \end{aligned}$$

We denote by xy the concatenation of (possibly 1-bit) words x and y.

o	0	$\mathtt{u}$	1
0	0	0	0
$\mathtt{u}$	0	$\mathtt{u}$	$\mathtt{u}$
1	0	$\mathtt{u}$	1

$o = \texttt{and}(a, b)$

o	0	$\mathtt{u}$	1
0	0	$\mathtt{u}$	1
$\mathtt{u}$	$\mathtt{u}$	$\mathtt{u}$	1
1	1	1	1

$o = \texttt{or}(a, b)$

a	o
0	1
$\mathtt{u}$	$\mathtt{u}$
1	0

$o = \texttt{not}(a)$

Table 2.1: Natural extension of the basic operators `and`, `or`, and `not` to Kleene logic.

Parity. The parity of a binary word denotes whether a binary word has an even or an odd number of bits of value 1.

the parity can be computed by `xor` over all bits in the word

Definition 2.1 (Parity)**.** *The parity of an n-bit word $x \in \mathbb{B}^n$ is defined by the sum over all bits modulo* 2*.*

$$\operatorname{par}(x) := \left(\sum_{i=1}^{n} x_i \right) \bmod 2 \,.$$

Functions. For notational convenience, we extend all functions $f\colon A \to B$ to sets of inputs. For non-empty input $A' \subseteq A$ take the union of f applied to each element of A',

$$f(A') = \bigcup_{a \in A'} f(a) \,.$$

2.2 Circuits and Kleene Logic

2.2.1 Kleene Logic

the third logical value, $\mathtt{u}$, may also be called *uncertain*

Kleene logic is an extension of Boolean logic. It extends the set of Boolean values by a third logical value. We call the Boolean values *stable* values, while the value $\mathtt{u}$ is called the *unstable* value. We regard the third logical value, $\mathtt{u}$, as the superposition of the logical values 0 and 1. Intuitively, the unstable state $\mathtt{u}$ may evaluate to any stable state at any point in the circuit, regardless of previous evaluations.

Basic Gates. In Kleene logic, the basic operators compute a stable value if and only if the inputs fully determine the output. The natural extension of the basic operators `and`, `or`, and `not` is given in Table 2.1. The specific choice of basic operators does not matter if the set of basic operators is functionally complete [52]. Thus, we stick to the standard model of operators `and`, `or` and `not`.

In ternary logic familiar identities $\texttt{and}(a, 1) = a$, and $\texttt{or}(a, 0) = a$ hold. The idempotent laws $\texttt{and}(a, a) = a$, and $\texttt{or}(a, a) = a$ and null laws $\texttt{and}(a, 0) = 0$, and $\texttt{or}(a, 1) = 1$ hold. The operators `and` and `or` are associative, commutative, and

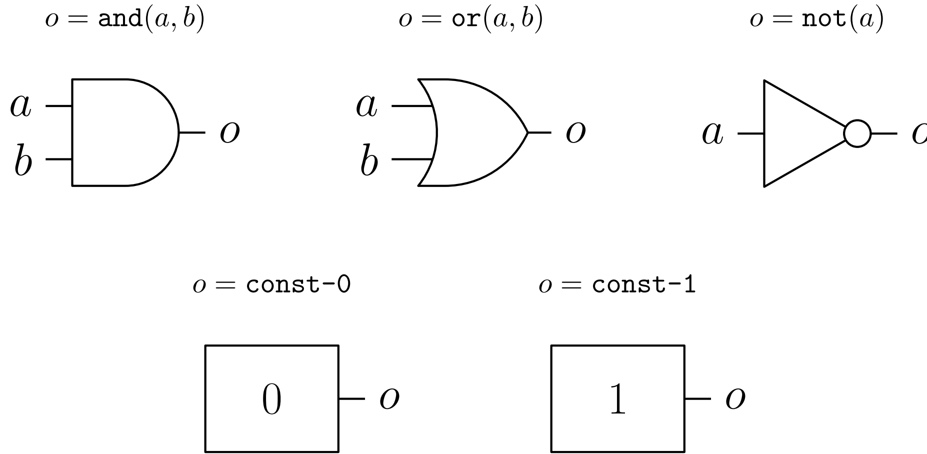

Figure 2.1: Circuit symbols for the basic gates `and`, `or`, `not`, `const-0`, and `const-1`.

distributive. Besides that, by the truth tables, we can verify that De Morgan's laws hold:

$$\texttt{not}(\texttt{and}(a, b)) = \texttt{or}(\texttt{not}(a), \texttt{not}(b)),$$
$$\texttt{not}(\texttt{or}(a, b)) = \texttt{and}(\texttt{not}(a), \texttt{not}(b)).$$

Remark. A difference between Boolean logic and Kleene logic is for example the law of non-contradiction. In Boolean logic, it holds for all $x \in \mathbb{B}$ that $\texttt{and}(x, \texttt{not}(x)) = 0$. In Kleene logic, the law is false, because $\texttt{and}(\mathtt{u}, \texttt{not}(\mathtt{u})) = \mathtt{u}$.

2.2.2 Circuits

A Boolean circuit is a directed acyclic graph, where each node is either an input node, an output node, or one of the following basic gates: `and`, `or`, `not`, `const-0`, and `const-1`. The *fan-in* of a gate is the number of its inputs. In our model, logic gates `and` and `or` have fan-in 2, `not` has fan-in 1, and constant gates `const-0` and `const-1` fan-in 0, i.e., they have no incoming edges. All basic gates have a single output. Circuit symbols for the basic gates are depicted in Figure 2.1.

Remark. For the implementation of hazard-free circuits, it is necessary to allow for constant output gates. We will show this in Definition 2.8 after the definition of hazard-free circuits.

A Boolean circuit C with n input nodes and m output nodes implements the function $C\colon \mathbb{T}^n \to \mathbb{T}^m$. The function of an output node is defined by induction over the circuit

structure and the specification of the basic gates. The function C is then defined by the concatenation of all output nodes.

Remark. We simplify the notation by using the same symbol for operators in ternary logic and basic gates: `and`, `or`, and `not`. Each basic gate is defined by the corresponding operator, they compute the same function.

2.3 Hazard-Free Circuits

We strive for circuits that behave similarly to basic gates when receiving unstable inputs. That is, given all inputs, if changing an input bit from 0 to 1 has no effect on the output, then setting this input bit to $\mathfrak{u}$ should also not affect the output. This concept is called hazard-freedom. To formalize this concept, we make use of two operations. The first operation is the *resolution*, which replaces the unstable value with both stable values. The second operation is the *superposition*, which results in the unstable value $\mathfrak{u}$ whenever its inputs do not agree on a stable value.

Resolution. The third logical value may resolve to a stable value, i.e., real circuits may interpret an unstable value as logic 0 or logic 1. The resolution is a function that maps a ternary word to a set of binary words. Intuitively, the resolution of a word includes all possible words that can be obtained by replacing each $\mathfrak{u}$ with a 0 or a 1.

Definition 2.2 (Resolution)**.** *The* resolution *is the function* $\text{res}\colon \mathbb{T}^n \to \mathcal{P}(\mathbb{B}^n)$*. For* $x \in \mathbb{T}^n$ *the resolution is defined by*

$$\text{res}(x) := \{y \in \mathbb{B}^n \mid \forall i \in \{1, \dots, n\}\colon x_i \neq \mathfrak{u} \Rightarrow y_i = x_i\}.$$

Example. The resolutions of 0101, $010\mathfrak{u}$, and $0\mathfrak{u}0\mathfrak{u}$ are given by

$$\begin{aligned}
\text{res}(0101) &= \{0101\}\,,\\
\text{res}(010\mathfrak{u}) &= \{0100, 0101\}\,,\\
\text{res}(0\mathfrak{u}0\mathfrak{u}) &= \{0000, 0001, 0100, 0101\}\,.
\end{aligned}$$

Remark. As stated before, applying a function to a set is given by the union over all applications on each element, e.g., $\text{res}(\{0\mathfrak{u}0, 1\mathfrak{u}0\}) = \{000, 010, 100, 110\}$.

Superposition. The third logical value is the superposition of both binary values. We extend this notion to words. If two bits do not agree on a value, their superposition is unstable.

Definition 2.3 (Superposition)**.** *Denote the* superposition *by the operator* $*$*. The superposition is the function* $*\colon \mathbb{T}^n \times \mathbb{T}^n \to \mathbb{T}^n$*. For* $x, y \in \mathbb{T}^n$ *the superposition is defined by bit-wise application. Let* $i \in \{1, \dots, n\}$*, then*

$$(x * y)_i = \begin{cases} x_i & \text{if } x_i = y_i, \\ \mathfrak{u} & \text{otherwise.} \end{cases}$$

The $*$-operation is associative and commutative. Hence, we can compute the superposition of a non-empty set $A \subseteq \mathbb{T}^n$ regardless of the order of application. Consider $A = \{a_0, \ldots, a_{\ell-1}\}$, we use the following equivalent notations for the superposition of all elements of A;

$$*A := \mathop{*}_{a \in A} a = a_0 * \cdots * a_{\ell-1} .$$

Example. Some superpositions of the words 0000, 0001, 0100, and 0101 are given by

$$\begin{aligned} 0101 * 0100 &= 010\mathfrak{u} , \\ 0001 * 0100 &= 0\mathfrak{u}0\mathfrak{u} , \\ *\{0000, 0001, 0100, 0101\} &= 0\mathfrak{u}0\mathfrak{u} . \end{aligned}$$

Resolution and superposition are functions that allow to compare the behavior of operations in the ternary logic to their behavior in binary logic. However, resolution and superposition are not inverse functions. The resolution of the superposition of a set $A \subseteq \mathbb{T}^n$ may add further words.

Observation 2.4. *Let $A \subseteq \mathbb{T}^n$ be a non-empty set, then $A \subseteq \operatorname{res}(*A)$.*

However, if a ternary word $*A$ has at most one $\mathfrak{u}$ bit, then its resolution is equal to A. Note that a word a has at most one $\mathfrak{u}$ bit iff $|\operatorname{res}(a)| \leq 2$.

Observation 2.5. *For any subset of words $A \subseteq \mathbb{B}^n$, if $|\operatorname{res}(*A)| \leq 2$, then*

$$\operatorname{res}(*A) = A .$$

Proof. Since $*A$ can contain at most one $\mathfrak{u}$ bit, we know that A can contain at most two words that differ in one position. It is then straightforward to show that every word in $\operatorname{res}(*A)$ is an element of A. Together with Observation 2.4 this shows equality. □

Observation 2.6. *Let $a \in \mathbb{T}^n$, then $a = * \operatorname{res}(a)$.*

Remark. Note that a strict superset relation is possible in Observation 2.4. For example let $A = \{101, 110\}$, then $\operatorname{res}(*A) = \operatorname{res}(1\mathfrak{u}\mathfrak{u}) = \{100, 101, 110, 111\}$.

2.3.1 Hazards

A circuit has as a hazard if it computes a $\mathfrak{u}$ although the stable inputs determine a stable output. With the definitions of the resolution and the superposition at our disposal, we can define how to extend a binary function to unstable inputs. We call this extension the *hazard-free extension.* The hazard-free extension gives the most precise output we can hope for when using Boolean circuits; no correct implementation can do better.

Definition 2.7 (Hazard-free Extension)**.** *Given a function* $f\colon \mathbb{B}^n \to \mathbb{B}^m$*, denote by* $f_\mathtt{u}\colon \mathbb{T}^n \to \mathbb{T}^m$ *its* hazard-free extension*, where*

$$f_\mathtt{u}(x) := \mathop{*}_{y \in \operatorname{res}(x)} f(y)\,.$$

Hazards are defined by the hazard-free extension. A circuit that does not implement the hazard-free extension has a hazard. If the number of $\mathtt{u}$'s in the input can be bounded by a number k we call it a k-bit hazard.

Definition 2.8 (k-Bit Hazards)**.** *Any circuit* C *implementing* $f\colon \mathbb{B}^n \to \mathbb{B}^n$ *has a* hazard *at* $x \in \mathbb{T}^n$ *iff* $C(x) \neq f_\mathtt{u}(x)$*. If* x *contains at most* k $\mathtt{u}$*'s, for some* $k \in \mathbb{N}$*, and* C *has a hazard at* x*, then* C *has a* k-bit hazard *at* x*.*

A circuit is *hazard-free* if it has no hazard, i.e., if it implements the hazard-free extension. Likewise, it is *k-bit hazard-free* if it has no k-bit hazards.

Definition 2.9 (k-Bit Hazard-Free)**.** *Any circuit* C *implementing* $f\colon \mathbb{B}^n \to \mathbb{B}^m$ *is* k*-bit hazard-free iff for all* $x \in \mathbb{T}^n$ *that contain at most* k $\mathtt{u}$*'s,*

$$C(x) = f_\mathtt{u}(x)\,.$$

Boolean circuits can be implemented using two types of gates only. A minimal set of basic gates includes `and` and `not` gates only. Hazard-free circuits, however, require additionally a constant gate. Consider a function $f\colon \mathbb{B} \to \mathbb{B}$ that gets one input and has a constant output. E.g., let $f(x) = 0$ and its hazard-free extension $f_\mathtt{u}(x) = 0$. None of the basic gates `and` and `or` produce a constant output for input $(\mathtt{u}, \mathtt{u})$. Note that $\mathtt{not}(\mathtt{u}) = \mathtt{u}$. Hence, if $x = \mathtt{u}$, no combination of basic gates can produce a stable output. Thus, any hazard-free circuit implementing f requires a constant output gate. The simplest hazard-free implementation of $f(x)$ is a single `const-0` gate.

2.3.2 Metastability

A textbook example of instability is the metastable state of flip-flops. Digital storage elements such as data flip-flops and latches are bistable elements. In other words, they store a binary value. Physically a flip-flop stores a value by setting the voltage level of its inner storage loop to ground level (GND) or supply level ($\mathsf{V_{DD}}$). In the Boolean abstraction, we map GND level to logic 0 and $\mathsf{V_{DD}}$ level to logic 1. By Marino [74], no bistable element can avoid an intermediate, metastable, state. The metastable state is an unstable equilibrium state between two stable states.

the metastable state cannot be avoided, detected, or resolved!

Figure 2.2 shows a series of simulations where a latch is operated close to its metastable state. If the signal on the data input arrives before the enable signal changes, then the latch output transitions to logic 1. This behavior can be seen in the first blue and red transitions. Accordingly, if the signal arrives late the latch output remains stable at logic 0. The latch becomes metastable if the data input arrives shortly before the enable signal is switched off. Each simulation moves the arrival

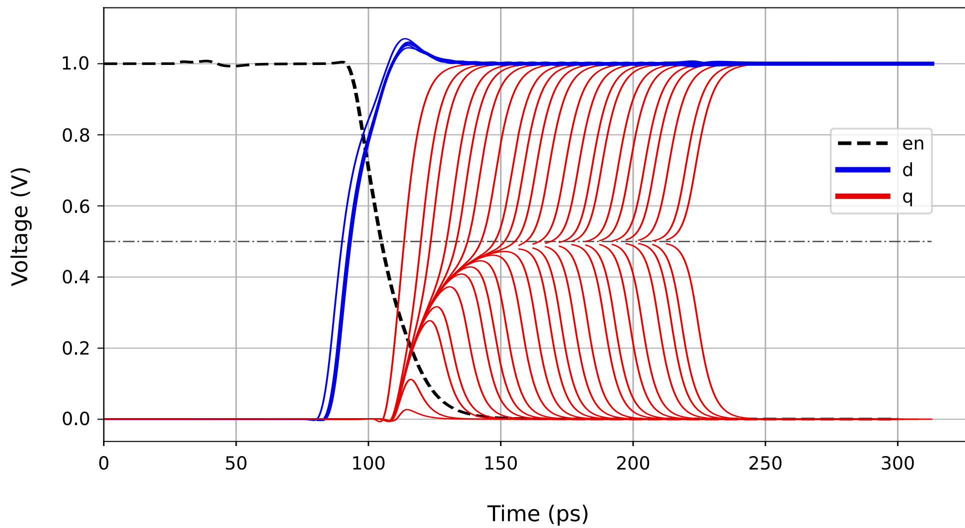

Figure 2.2: Simulation traces of a latch with enable signal *en*, data input *d* and output *q*.

q traces on longer metastability are partial only, due to the mode of simulation (which achieves higher precision)

time of the data input closer to the metastable point. We observe that the output of the latch rises to roughly the midpoint between GND (0 V) and V_{DD} (1 V) before it resolves to either GND or V_{DD}.

Any subsequent circuit that obtains inputs from storage elements may receive a metastable signal. We denote a metastable signal by M. We denote any signal that cannot be interpreted as logic 0 or logic 1 by M. That is for example signals with long transitions between GND and V_{DD} or oscillating signals. The input signal to the circuit, hence, can be any value in $\{\mathsf{GND}, \mathtt{M}, \mathsf{V_{DD}}\}$. This perfectly matches the logic values $\{0, \mathfrak{u}, 1\}$. Thus, we make use of the ternary logic to model metastable behavior.

If we disregard the possibility that metastability resolves over time, i.e., if we do a worst-case analysis, then we see that metastability propagates through the circuit. CMOS gates that have a metastable input can have a metastable output. Metastable signals may be interpreted as any stable signal at any point in the circuit, e.g., after forks the same signal can have different interpretations.

This makes the metastable signal M a perfect real-world example for the unstable logic value $\mathfrak{u}$. In the worst-case model, metastability is an unstable value between GND (logic 0) and V_{DD} (logic 1). It may evaluate to any stable signal at any point in the circuit.

Basically, this means that we can use $\mathfrak{u}$ to model M. The line of research we present in this work is motivated by metastability as it occurs in latches. Practical results such as [12, 16] use the M notation. More theoretical results, e.g. [17], use the

u notation in order to fit in with the notation of related work. Contributions [12],[13], and [16] use the M notation and contribution [17] uses the u notation. Throughout this work, we use u for consistency.

2.3.3 Metastability-Containment.

The theory of hazard-free circuits can be applied to compute the most precise output in the face of metastability. A circuit that has no hazard can also be called a *metastability-containing* circuit. It implements the hazard-free extension of its function, which is also known as the *metastable closure.* A hazard-free circuit will propagate metastability only if the stable inputs do not determine the output, i.e., the uncertainty of metastable signals is not amplified but contained. Superposition and resolution can be applied to M in the same way as for u. A circuit that computes the metastable closure of a function is called *metastability-containing* (mc).

Remark. We are aware that the notion of a hazard is ambiguous. Sometimes, especially in work that studies metastable behavior, a hazard describes a short transient fault. For example an electrical pulse on a signal that is supposed to be stable. This behavior is also called a glitch. In order to remain consistent with related work (cf. Section 2.6), we stick to the notion of a hazard denoting the misbehavior of circuits on unstable inputs.

The notion of a hazard has been coined by Huffman in the study of glitches in switching networks [51]. Circuits that prevent glitches are called glitch-free. When an input changes its value a circuit might produce a glitch at the output due to internal delays. If the output is the same before and after the input transition Huffman called this behavior a static hazard. If the output changes its value Huffman called this a dynamic hazard. A hazard in our sense corresponds to a static hazard in the sense of Huffman, we do not handle dynamic hazards.

dynamic hazards cannot be captured by a ternary logic [11]

Hazard-freedom in our sense describes a more general resilience to unstable signals. An unstable signal can be glitching, oscillating, or any other arbitrary behavior between GND and V_{DD}. As a consequence, every hazard-free circuit is also static glitch-free.

2.4 Implementation of Basic Gates

In this section, we make a short excursion to the implementation of basic gates on the transistor-level. We want to show that it is feasible to assume that there are basic gates that implement the behavior of the basic operators in Table 2.1. To keep this section brief, we expect basic knowledge of complementary metal-oxide-semiconductor (CMOS) technology. The knowledge is needed for this section only. The main takeaway of this section is that hazard-free basic gates are not artificial but can be built in real world. These basic building blocks are part of standard processes already.

o	0	u	1
0	1	1	1
u	1	u	u
1	1	u	0

$o = \texttt{nand}(a, b)$

o	0	u	1
0	1	u	0
u	u	u	0
1	0	0	0

$o = \texttt{nor}(a, b)$

a	o
0	1
u	u
1	0

$o = \texttt{not}(a)$

Table 2.2: Natural extension of the basic operators `nand`, `nor`, and `not`.

CMOS Gates. Standard CMOS gates are suitable choices for basic gates. The functions `nand` and `nor` are defined in Table 2.2, where $\texttt{nand}(a, b) = \texttt{not}(\texttt{and}(a, b))$ and $\texttt{nor}(a, b) = \texttt{not}(\texttt{or}(a, b))$. Implementations of `nand`, `nor`, and `not` gates in CMOS logic are given in Figure 2.3. Formally we show the following theorem in [16].

Theorem 2.10. *The CMOS gates depicted in Figure 2.3 implement the basic gates defined in Table 2.2.*

Intuitively, the theorem states that the standard CMOS implementation of `nand`, `nor`, and `not` gates implement the natural extension to the respective ternary operators.

Remark. We stress that the property of implementing the metastable closure is not universal for CMOS logic. A CMOS gate is not necessarily mc. For instance, standard transistor-level multiplexer implementations are not mc. However, custom implementations can be tailored for metastability. An mc CMOS implementation of the multiplexer is given in [42].

2.5 Example: The Hazard-Free Multiplexer

In this section, we walk through an example that demonstrates the application of the basic definitions. The example emphasizes that designing hazard-free circuits is not straightforward.

We discuss the specification and implementation of a `MUX` and its hazard-free implementation `CMUX`, where `CMUX` actually abbreviates: metastability-containing multiplexer. To our knowledge, the first appearance of a `CMUX` dates back to Goto [46]. The metastability-containing multiplexer (`CMUX`) is a very handy device in hazard-free circuit complexity, it will appear oftentimes later in this work.

Specification of a `MUX`. The `MUX` has two data inputs a and b, a select input s, and a single output o. The select input determines which data input is forwarded to the output.

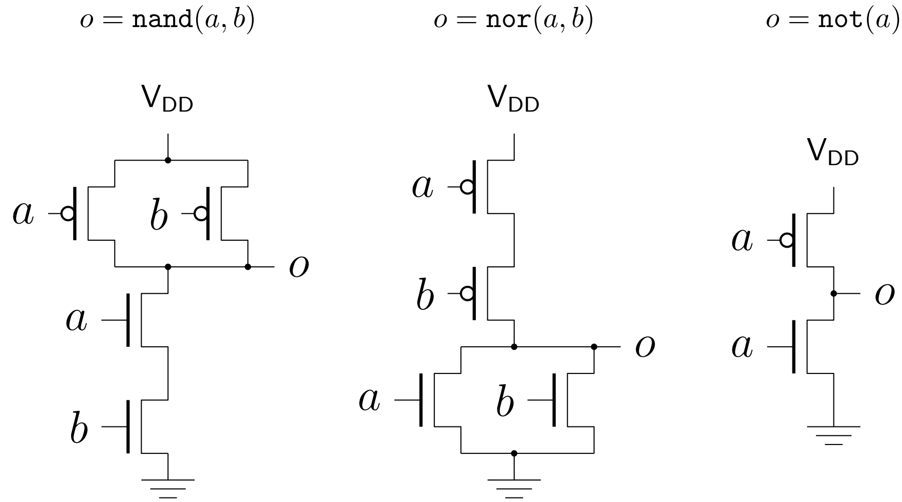

Figure 2.3: Standard transistor-level implementations of `nand`, `nor`, and `not` in CMOS technology.

Definition 2.11 (MUX)**.** *Let* $a, b, s \in \mathbb{B}$*, then*

$$\mathtt{MUX}(a, b, s) := \begin{cases} a & \textit{if } s = 0\,, \\ b & \textit{if } s = 1\,. \end{cases}$$

The hazard-free extension determines the most precise output of the MUX for any kind of unstable inputs. We obtain for $a, b, s \in \mathbb{T}$,

$$\mathtt{MUX}_{\mathtt{u}}(a, b, s) := \begin{cases} a & \text{if } s = 0\,, \\ a * b & \text{if } s = \mathtt{u}\,, \\ b & \text{if } s = 1\,. \end{cases}$$

2.5.1 Implementation of a MUX

The standard implementation of the MUX uses a `not` gate to invert the select input, two `and` gates to mask inputs a and b, and an `or` gate to combine the results. The implementation is depicted in Figure 2.4 (left).

$\mathtt{MUX}(a, b, s)$ denotes the function, $\mathtt{mux}(a, b, s)$ denotes the circuit

When comparing the output of $\mathtt{mux}(a, b, s)$ to the output of $\mathtt{MUX}_{\mathtt{u}}(a, b, s)$ we observe that the standard implementation has a hazard. On one hand, the standard

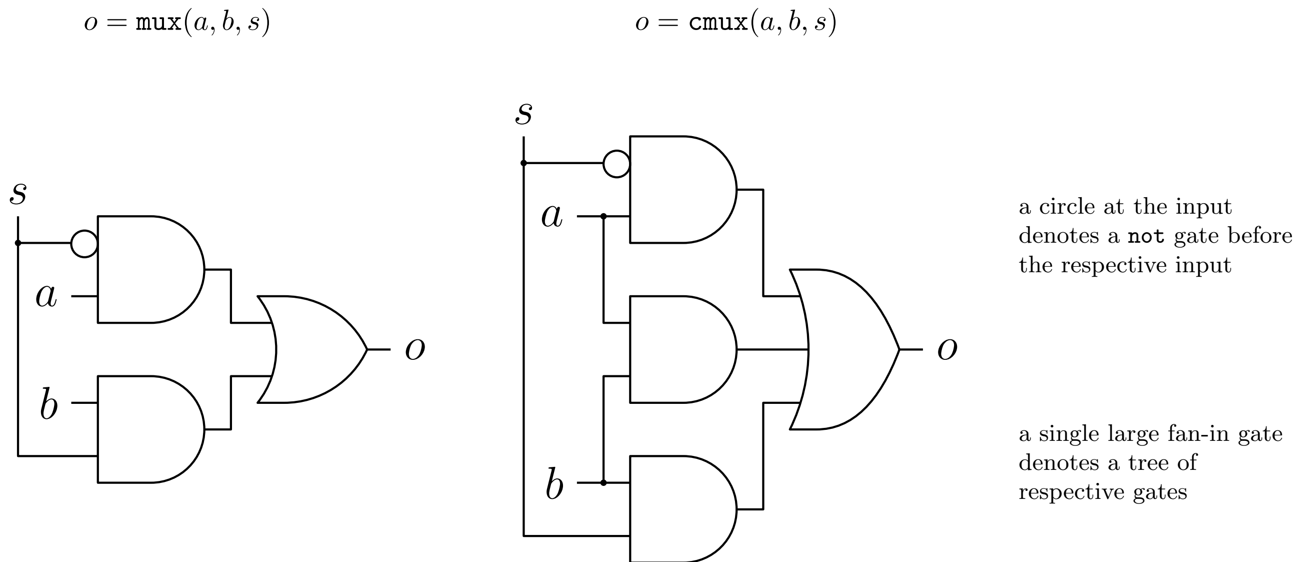

Figure 2.4: Standard implementation of a `MUX` with basic gates (left) and implementation of a `CMUX` with basic gates (right).

implementation computes on input $a = b = 1,\ s = \mathfrak{u}$

$$\begin{aligned}\texttt{mux}(1,1,\mathfrak{u}) &= \texttt{or}(\texttt{and}(1,\texttt{not}(\mathfrak{u})),\texttt{and}(1,\mathfrak{u}))\\ &= \texttt{or}(\texttt{and}(1,\mathfrak{u}),\texttt{and}(1,\mathfrak{u}))\\ &= \texttt{or}(\mathfrak{u},\mathfrak{u})\\ &= \mathfrak{u}\,.\end{aligned}$$

On the other hand, the hazard-free extension requires on input $a = b = 1,\ s = \mathfrak{u}$

$$\begin{aligned}\texttt{MUX}_{\mathfrak{u}}(1,1,\mathfrak{u}) &= *\{\texttt{MUX}(1,1,0),\texttt{MUX}(1,1,1)\}\\ &= 1 * 1\\ &= 1\,.\end{aligned}$$

Remark. This is the only input on which $\texttt{mux}(a, b, s)$ has a hazard. Hence, we can extend the standard implementation to account for this case, given that we do not introduce new hazards.

2.5.2 Implementation of a `CMUX`

The hazard-free implementation of the `MUX` is similar to the standard implementation. It adds an `and` gate and an `or` gate to the standard implementation. The $\texttt{cmux}(a, b, s)$ is depicted in Figure 2.4 (right).

Hazard-freedom of the $\texttt{cmux}(a, b, s)$ circuit can be verified by checking all possible input pairs. Here, we only show that the hazard in $\texttt{mux}(a, b, s)$ disappeared. On input $a = b = 1$, $s = \mathfrak{u}$ we obtain

$$\begin{aligned}\texttt{cmux}(1,1,\mathfrak{u}) &= \texttt{or}(\texttt{and}(1,\texttt{not}(\mathfrak{u})),\texttt{and}(1,1),\texttt{and}(1,\mathfrak{u}))\\ &= \texttt{or}(\texttt{and}(1,\mathfrak{u}),\texttt{and}(1,1),\texttt{and}(1,\mathfrak{u}))\\ &= \texttt{or}(\mathfrak{u},1,\mathfrak{u})\\ &= 1\end{aligned}$$

2.5.3 ℓ-select Multiplexer

For later use, we define the ℓ-select multiplexer $\texttt{MUX}_\ell$, i.e., a multiplexer that selects one of 2^ℓ inputs according to a dedicated select input of width ℓ. The select input encodes index $i \in [2^\ell]$.

we formally define the standard binary encoding γ_n^{bin} in Chapter 3

Definition 2.12. *Let $\ell, b \in \mathbb{N}_{>0}$. An ℓ-input multiplexer $\texttt{MUX}_\ell$ receives inputs $x_i \in \mathbb{B}^b$ for $i \in [2^\ell]$, and select input $s \in \mathbb{B}^\ell$. Let $\langle\cdot\rangle : \mathbb{B}^\ell \to [2^\ell]$ be the standard binary decoding function. Interpreting the select input s as an index, $\texttt{MUX}_\ell$ outputs $x_{\langle s\rangle}$, i.e.,*

$$\texttt{MUX}_\ell(x_0, \dots, x_{2^\ell-1}, s) \coloneqq x_{\langle s\rangle}\,.$$

By [52, Lemma 5.1] the ℓ-input multiplexer can be implemented by a hazard-free circuit.

Lemma 2.13. *Let $\ell, b \in \mathbb{N}_{>0}$, there is a hazard-free implementation of $\texttt{MUX}_\ell$ computing $(\texttt{MUX}_\ell)_\mathfrak{u}(x_0, \dots, x_{2^\ell-1}, s)$ with $x_i \in \mathbb{T}^b$ for $i \in [2^\ell]$ and $s \in \mathbb{T}^\ell$. The implementation has*

$$\begin{aligned}\textit{size}\;\; &\mathcal{O}(2^\ell b)\\ \textit{and depth}\;\; &\mathcal{O}(\ell)\,.\end{aligned}$$

Proof. The hazard-free implementation of a multiplexer receiving two input bits and a single select bit is given by $\texttt{MUX}_\mathfrak{u}$ in Figure 2.4; it has constant size and and depth. We first extend it to ℓ select bits and then to b-wide inputs.

The multiplexer can be extended to ℓ select bits by a recursive construction. For $\ell = 1$ we compute $(\texttt{MUX}_\ell)_\mathfrak{u}(x_0, x_1, s)$ by $\texttt{MUX}_\mathfrak{u}(x_0, x_1, s)$. For $\ell > 1$ we recursively compute

$$\begin{aligned}(\texttt{MUX}_\ell)_\mathfrak{u}(x_0, \dots, x_{2^{\ell-1}-1}, s_{2,\ell}) &= x_{\langle s_{2,\ell}\rangle} \text{ and}\\ (\texttt{MUX}_\ell)_\mathfrak{u}(x_{2^{\ell-1}}, \dots, x_{2^\ell-1}, s_{2,\ell}) &= x_{2^{\ell-1}+\langle s_{2,\ell}\rangle}\,.\end{aligned}$$

Then, we can compute $(\texttt{MUX}_\ell)_\mathfrak{u}(x_0, \dots, x_{2^\ell-1}, s)$ by $\texttt{MUX}_\mathfrak{u}(x_{\langle s_{2,\ell}\rangle}, x_{2^{\ell-1}+\langle s_{2,\ell}\rangle}, s_1)$. Intuitively, we build a tree of multiplexers where each layer is controlled by one bit of the select input. The tree uses 2^ℓ instances of $\texttt{MUX}_\mathfrak{u}$ and it has depth ℓ.

Extension to inputs of width b is simply done by copying the tree of multiplexers b times. For $i \in \{1 \dots b\}$, we can compute the ith output bit by a $(\texttt{MUX}_\ell)_\mathfrak{u}$ selecting from the ith bits of inputs x_0 to $x_{2^\ell-1}$. □

2.6 Related Work

The `cmux` example (in Section 2.5) demonstrates that an implementation of the hazard-free extension might be more complex than an implementation of the function itself. A measure of *complexity* of hazard-free circuits is the overhead a hazard-free implementation has compared to its Boolean counterpart. In general, an exponential blow-up cannot be avoided [52]. In this section, we discuss related work that studies the complexity of hazard-free circuits.

Switching Networks. The strive for hazard-free circuits naturally arises from switching networks. Glitches might occur at the output, when delays in a network do not match. Consider for example the `mux` from Figure 2.4. Let $a = b = 1$ and input s switches from 1 to 0. If the upper `and` has a greater delay than the lower one, then there is a short time where both inputs to the `or` are 0. A short 1-0-1 glitch occurs at the output, although it should be constant 1 as we saw in the example. This behavior is called a *static hazard*. In contrast to a 1-bit hazard, a static hazard determines also the input (bit) that causes the glitch. A circuit that eliminates all static hazards is also hazard-free.

electronic glitches are sometimes known as hazards

Huffman [51] shows that every Boolean function can be computed by a hazard-free circuit. Essentially the proof shows that the disjunctive normal form (DNF) of a Boolean function whose terms are prime implicants is 1-bit hazard-free. Huffman also notes that the results carry over to higher-order hazards. Yoeli and Rinon [96] relate the notion of hazards to ternary logic.

Hazard-free Complexity. More recently, Ikenmeyer et al. [52] showed that the hazard-free complexity of a function is equal to its monotone complexity. A monotone circuit has only `and` and `or` gates, `not` gates are not allowed. From lower bounds on monotone circuit complexity, they can infer an exponential blow-up for the hazard-free complexity. Also Ikenmeyer et al. [52] showed a conditional lower bound, which states the exponential blow-up for non-monotone circuits.

Jukna [55] presents an upper bound on the complexity of the hazard-free implementation of a Boolean function f. We state the theorem for later use.

Theorem 2.14. *Given a Boolean function $f\colon \mathbb{B}^n \to \mathbb{B}$, there is a hazard-free circuit implementing f that has size $\mathcal{O}(2^n/n)$ and depth $\mathcal{O}(n)$.*

Jukna presents a construction that formally proves the upper bound. The formal argument only concerns the circuit size, a bound on the depth follows by the following argument. The presented construction consists of two parts, that are both recursively defined. The first part uses at most m recursions and the second part uses $n - m$ recursions, where each recursion step has constant depth. Outputs of the first part are inputs to the second part. Hence, the construction has depth $\mathcal{O}(n)$.

Hazard Detection. After the attention to static hazards rose, detecting and eliminating these has been extensively studied [37, 76, 96]. Only a recent result of Komarath and Saurabh [61] shows that detecting hazards is computationally hard. The authors show a lower bound, which proves that no algorithm can decide in polynomial time whether a circuit with n inputs has a hazard.

Metastability-Containment. Metastability is a spurious mode of operating storage elements. When a storage element, e.g., a flip-flop, is metastable it is in an unstable equilibrium state that is neither 0 nor 1. The term metastability-containing is coined by Friedrichs et al. [41]. They apply techniques known from hazard-free switching circuits to prevent the propagation of metastability beyond the unavoidable upsets.

An mc circuit in the sense of Friedrichs et al. neglects the need for synchronizers. A synchronizer trades time for reliability [5]. A signal that is suspected to be metastable can be piped through a synchronizer to increase the probability that metastability resolves. The synchronization approach comes with no deterministic guarantees, usually a figure of merit is the mean time between failures (MTBF).

Remark. Synchronizers and clock synchronization should not be confused. A synchronizer is a device, which might be used for clock synchronization.

Furthermore, Friedrichs et al. introduce a new kind of register: the *masking register* that can hide internal metastability. There are two types of masking registers; one that outputs 0 and one that outputs 1 on internal metastability. Essentially, masking registers trade metastable upsets for late transitions. A late transition is a clean transition from a stable value to the other. The transition can have an arbitrary delay. It enters the metastable state only for a short time. The authors show that circuits using masking registers have more computational power than conventional circuits. Hazard-free circuits using masking registers are significantly smaller than conventional circuits. Unfortunately, masking registers are not part of standard circuit design flows. It is not obvious how to build such registers. Further research on implementations is required. When using masking registers elaborate timing of signals is required as each type of register can only hide metastability occurring on an up or a down transition. The register has to be reset when it is not read.

Subsequently, more mc circuits have been introduced. The mc `MUX` of [42] presents an improvement of the `cmux` on transistor-level. The mc sorting networks from [16] are discussed in detail in Section 5.2.

Tarawneh et al. [93] explore the interplay of mc design and resolution of metastability. In a state machine, metastable signals are synchronized during the state/data computation. Assuming that metastability resolves after sufficient time, they present a framework for modeling synchronous state machines. Metastable upsets can be reduced to a subset of the computations, such that mc design needs to be applied only partially.

Cybersecurity. Studying hazard-free circuits also became interesting for applications in cybersecurity. Techniques like gate level information flow tracking (GLIFT) are proposed to secure information flow on the hardware level. Hu et al. [50] relates GLIFT logic to hazard-free circuits, also showing that constructing GLIFT logic is NP-complete.

Summary. The study of hazard-free circuits naturally rises from the study of switching networks. The field gained a lot of attention in the 1960's. More recently it became interesting as new lower bounds were shown and there are modern applications such as GLIFT logic. We are interested in a general construction for hazard-free circuits. Known constructions add significant overhead to the circuit size. Ikenmeyer et al. show that it is impossible to avoid these large overheads. However, in Chapter 4 we present a construction for small hazard-free circuits from small transducers. In our construction, we encounter a basic problem of hazard-free circuits: encoding matters. The type of encoding chosen is important in ternary logic. Superposition of values may lose important information about the original data. In the following chapter, we discuss in detail the problems and properties of codes in ternary logic.

a point in the power set is a set again

the set of codewords is only the intersection of extended codewords and binary strings

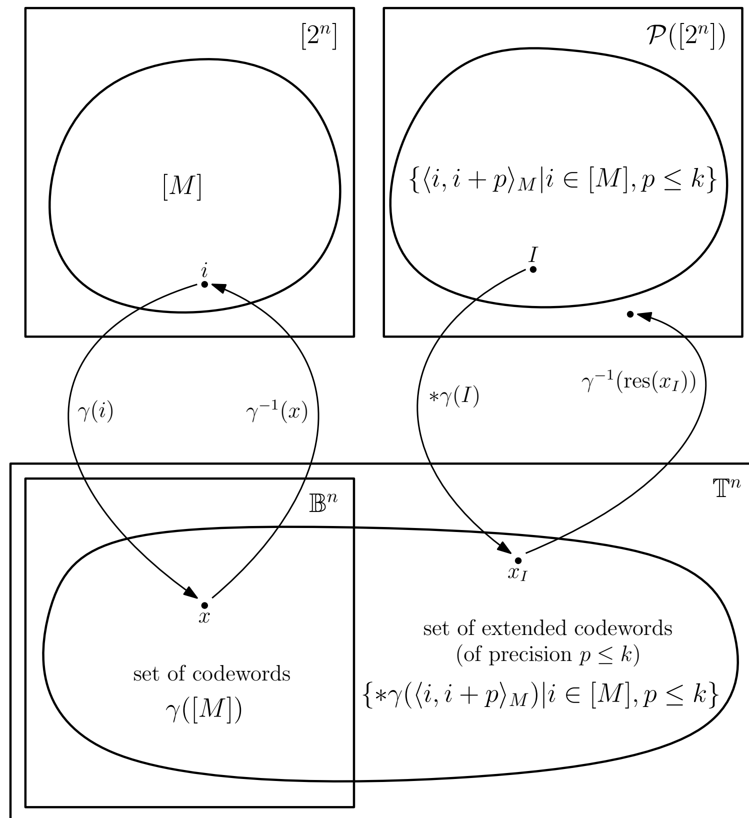

Figure 3.1: Overview of encoding and decoding functions and their extension to the ternary world. The encoding function γ maps a number i to a codeword x of length n. Its inverse, the decoding function γ^{-1} maps x back to i. The superposition of all codewords of an interval I is the extended codeword x_I. In Kleene logic decoding all resolutions of x_I may result in a set of numbers not equal to I.

Encoding 3

In hazard-free circuits the type of encoding chosen for data is important. Some encodings lose precision when superpositioning multiple values. In this section, we discuss encodings in detail. We formally define encodings and identify two desirable properties of codes. The properties provide information on whether a code is appropriate for hazard-free circuits or not.

At the end of the chapter, we define three popular codes that will be used throughout this thesis and examine their preservation and recoverability properties. The codes are standard binary code, unary thermometer code, and Binary Reflected Gray Code (BRGC).

3.1 Codes

A code is a mapping between a finite set of numbers and strings of bits of a fixed length. Any code is defined by its encoding function. The encoding function defines the set of codewords and the decoding function.

Definition 3.1 (Codes)**.** *A* code *is given by an injective function* $\gamma\colon [M] \to \mathbb{B}^n$ *that maps integers from a set* $[M]$ *to* n*-bit strings. We call* γ *the* encoding function *and its image* $\gamma([M])$ *the* set of codewords*. The inverse of* γ *on the set of codewords, denoted by* $\gamma^{-1}\colon \mathbb{B}^n \to [M]$*, is called the* decoding function*. The decoding function is only defined on the domain* $\gamma([M])$*.*

A subset of the domain can be mapped to a string of ternary bits by taking the superposition over all codewords. We call this an extended codeword of the subset. In this work, we always consider subsets of consecutive integers, which we call intervals By definition (cf. Chapter 2) intervals count modulo M, hence, 0 is the successor of $M-1$. An overview of sets and mappings defined by a code is part of Figure 3.1.

Definition 3.2. *For an interval* $I = \langle i, i+k\rangle_M$ *we define*

the extended codeword x_I	*by* $x_I := *\,\gamma(I)$*,*
the range r_{x_I} *of* x_I	*by* $r_{x_I} := I$*, and*
the imprecision *of* x_I	*by* $p := \vert I\vert - 1$*.*

Depending on the encoding function such an extended codeword does not preserve the information about the original set. It cannot be mapped back in every case.

Example 3.3 (Binary Code)**.** *Let* $\gamma_n^{\mathrm{bin}}\colon [2^n] \to \mathbb{B}^n$ *be the encoding function of the standard binary encoding. Regard the interval* $\langle 3,4\rangle_M = \{3,4\}$*, where codewords are given by* $\gamma_n^{\mathrm{bin}}(3) = 0011$ *and* $\gamma_n^{\mathrm{bin}}(4) = 0100$*. The extended codeword is the superposition of all codewords, it is* $0\mathfrak{uuu}$*. Resolution of the extended codeword yields more codewords, such that* $\mathrm{res}(0\mathfrak{uuu}) \neq \{\gamma_n^{\mathrm{bin}}(3), \gamma_n^{\mathrm{bin}}(4)\}$*.*

As stated before (cf. Observation 2.4) the superposition is not reversible in general because it is not an injective function. For codes, this means that there may be multiple intervals that map to the same extended codeword. Thus, mapping an interval to ternary strings loses information about the original interval.

Example (Binary Code (continued)). The binary encoding γ_n^{bin} maps intervals $\langle 5, 6\rangle_M$ and $\langle 4, 7\rangle_M$ to the same ternary string. From the binary encoding, we obtain the following codewords:

$$\gamma_n^{\text{bin}}(4) = 0100\,,\ \gamma_n^{\text{bin}}(5) = 0101\,,$$
$$\gamma_n^{\text{bin}}(6) = 0110\,,\ \gamma_n^{\text{bin}}(7) = 0111\,.$$

Hence, extended codewords of intervals $\langle 5, 6\rangle_M$ and $\langle 4, 7\rangle_M$ are given by

$$\langle 5, 6\rangle_M = *\{0101, 0110\} = 01\mathtt{uu} \text{ and}$$
$$\langle 4, 7\rangle_M = *\{0100, 0101, 0110, 0111\} = 01\mathtt{uu}\,.$$

Remark. Thus, the range (as given in Definition 3.2) is not well defined. For an extended codeword, there may be multiple ranges that map to the codeword, as shown in the above example. We use the range notation to denote the interval only when the interval is clear from context.

3.2 Preserving and Recoverable Codes

We define two properties of codes that prevent loss of information when mapping to extended codewords. Parameter k describes the maximum size of an interval that can be mapped to an extended codeword without loss of information.

Preserving Codes. First, preservation ensures that the original interval is preserved, i.e., a resolution of an extended codeword will not add new codewords to the original interval. The resolution may, however, add strings that are non-codewords. A positive and a negative illustration are given in Figure 3.2.

Definition 3.4 (k-preserving Codes)**.** *A code γ on domain $[M]$ is k-preserving iff for each interval $I = \langle i, i+p\rangle_M$, where $p \leq k$, the resolution $\operatorname{res}(x_I)$ does not contain codewords which are not in $\gamma(I)$. Formally,*

$$\gamma([M]) \cap \operatorname{res}(x_I) = \gamma(I)\,.$$

Recoverable Codes. Second, the recoverability property makes sure that every non-codeword in the resolution of an extended codeword can be mapped back to the original interval. The property is illustrated in Figure 3.3 and defined in Definition 3.5.

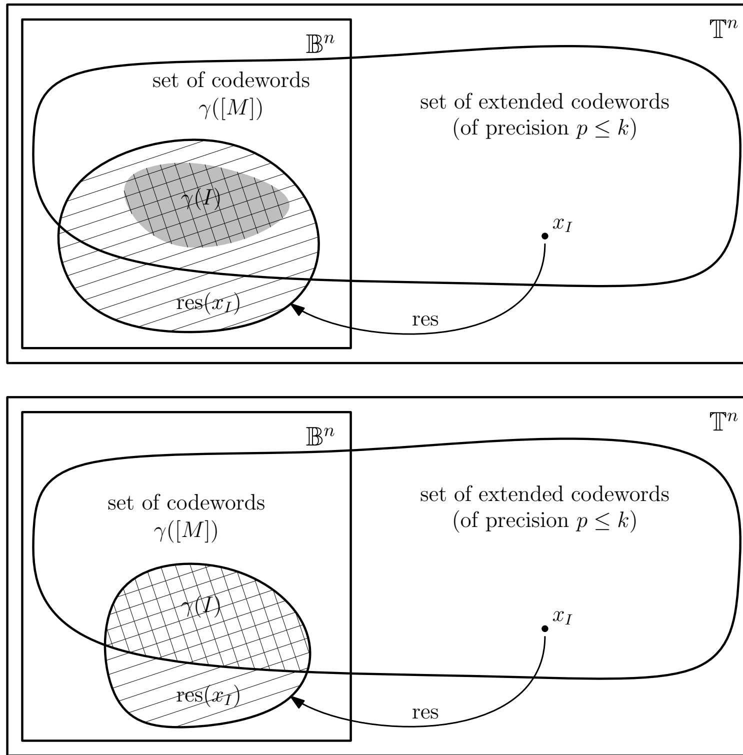

Figure 3.2: Illustration of a code that is not preserving (top) and a code that is preserving (bottom). Hatched areas denote $\mathrm{res}(x_I)$ and checkered areas denote $\gamma(I)$. For a code that is preserving, the resolution of extended codeword x_I must not add new codewords. It may add binary strings that are not codewords.

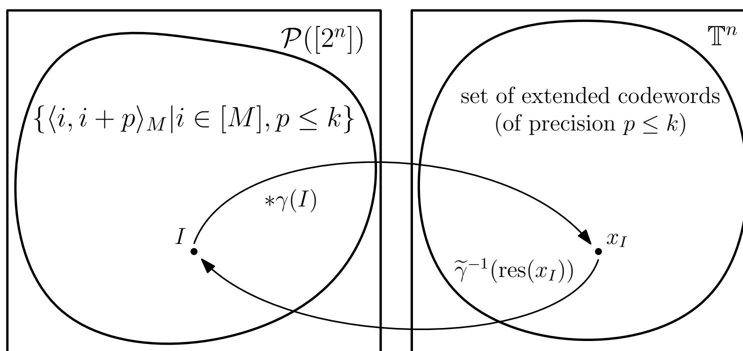

Figure 3.3: Illustration of the recoverability property.

Before the definition of k-recoverable codes, we need to define extensions to functions. Let $f\colon A \to C$ and $g\colon B \setminus A \to C$ be two functions and let $\widetilde{f}$ be the extension of f from A to B by g. Then $\widetilde{f}\colon B \to C$ is given by

$$\widetilde{f}(x) \coloneqq \begin{cases} f(x) & \text{, if } x \in A \\ g(x) & \text{, if } x \in B \setminus A \,. \end{cases}$$

Definition 3.5 (k-recoverable Codes). *Let x be an extended codeword of imprecision $p_x \leq k$. A code γ is k-recoverable iff there exists an extension of γ^{-1} from codewords to bit strings, such that each resolution of x is mapped to the range of x. Denote the extension of $\gamma^{-1}\colon \mathbb{B}^n \to [M]$ by $\widetilde{\gamma}^{-1}\colon \mathbb{B}^n \to [M]$. Formally, for $I = \langle i, i+p \rangle_M$ and $p \leq k$,*

$$\widetilde{\gamma}^{-1}(\mathrm{res}(x_I)) \subseteq I \,.$$

Observations on Preserving and Recoverable Codes. We observe for any code γ and any extension $\widetilde{\gamma}^{-1}$ that $I \subseteq \widetilde{\gamma}^{-1}(\mathrm{res}(x_I))$. This leads us to the observation that in any k-recoverable code the extended decoding function maps to the original interval.

Observation 3.6. *Consider $I = \langle i, i+p \rangle_M$ and $p \leq k$. If γ is k-recoverable, in fact*

$$\widetilde{\gamma}^{-1}(\mathrm{res}(x_I)) = I \,.$$

Proof. As γ is k-recoverable we have that $\widetilde{\gamma}^{-1}(\mathrm{res}(x_I)) \subseteq I$. We show that also $I \subseteq \widetilde{\gamma}^{-1}(\mathrm{res}(x_I))$. Consider an arbitrary $i \in I$. By definition of superposition and resolution (Definition 2.3 and Definition 2.2) we know that $\gamma(i)$ must be element of $\mathrm{res}(x_I)$. Hence, i is also an element of $\gamma^{-1}(\mathrm{res}(x_I))$. As $\widetilde{\gamma}^{-1}$ is only an extension to γ^{-1}, i is also an element of $\widetilde{\gamma}^{-1}(\mathrm{res}(x_I))$. Thus, for each $i \in I$ we have that $i \in \widetilde{\gamma}^{-1}(\mathrm{res}(x_I))$. □

Moreover, we note that preservation is a necessary condition for recoverability.

Lemma 3.7. *Every code that is k-recoverable is also k-preserving.*

Proof. We prove the contrapositive, i.e., no code is k-recoverable if it is not k-preserving. Assume that a code γ is not k-preserving. By Definition 3.4 there is an interval I such that there is some $x \in \gamma([M]) \cap \mathrm{res}(x_I)$ that is not an element of $\gamma(I)$. By Definition 3.1 the decoding function satisfies $\gamma^{-1}(\gamma(I)) = I$. Hence, since $x \notin \gamma(I)$ it follows that $\gamma^{-1}(x) \notin I$. Thus, no extension to γ^{-1} can satisfy the requirement $\widetilde{\gamma}^{-1}(\mathrm{res}(x_I)) \subseteq I$. □

the inverse of the redundancy is also known as the rate of a code

Optimal Codes. In error-correcting codes, the efficiency of a code is measured by its *redundancy*. The redundancy of a code is the ratio of the number of bits in use compared to the minimum number necessary to encode the same information [49].

Definition 3.8. *The redundancy of a code $\gamma\colon [2^m] \to \mathbb{B}^\ell$ is the ratio $\ell/m \geq 1$.*

A code that has redundancy 1 is a bijection. Each binary string is a codeword that can be mapped to a natural number. From Definition 3.4 we can infer that no code with redundancy 1 can be more than 1-preserving. Consider an arbitrary interval of three elements, the corresponding extended codeword must have at least 2 unstable bits. The resolution of the extended codeword thus has four elements, each of them is a codeword as there are no redundant non-codewords. By Lemma 3.7 this also applies to recoverability.

Corollary 3.9. *Any code with (optimal) redundancy* 1 *is at most* 1*-recoverable.*

A class of codes that is of special interest to us are Gray codes. They are named after Frank Gray who presented the most prominent Gray code in [48]. A code is a Gray code if two consecutive codewords differ in exactly one bit.

the code presented by Gray was introduced a decade earlier by George R. Stibitz [90]

Definition 3.10 (Gray codes)**.** *A code* $\gamma\colon [M] \to \mathbb{B}^n$ *is a Gray code if for* $i \in [M]$ *the codewords* $\gamma(i)$ *and* $\gamma(i+1 \bmod M)$ *have Hamming distance* 1*. The Hamming distance between two strings is the number of positions at which the bits are different.*

Every Gray code is 1-preserving and 1-recoverable. In Section 3.3 we present two Gray codes. The code γ_n^{s} with suboptimal redundancy and the code γ_n^{g} with optimal redundancy 1.

Recoverability Requires Redundancy. Larger recoverability requires also more redundancy. The main result we present in this chapter is a lower bound on the redundancy required for recoverability. We show the statement for codes that are Gray codes.

It is feasible to assume that recoverable codes with low redundancy are Gray codes. Codes that flip more than one bit per step are likely to require more redundancy. For example, consider a code that flips two bits on the step from $\gamma(i)$ to $\gamma(i+1)$. Then the extended codeword of $\langle i, i+1\rangle_M$ has two $\mathfrak{u}$'s and its resolution has size four. Hence, for the code to be 1-preserving we use four words in the codomain, whereas a Gray code uses two.

An n-bit code which is k-recoverable can encode at most $2^{n-k}(k+1)$ values. The redundancy of a k-recoverable code is bounded below by $n/(n-k+\log(k+1))$. We first show that two codewords of distance $\ell < k$ have at least Hamming distance ℓ.

Lemma 3.11. *For* $k, n \in \mathbb{N}$*, let* γ *be an* n*-bit,* k*-preserving Gray code. Let* $i \in [M-k]$ *and* $\ell \leq k$*, then the two codewords* $\gamma(i)$ *and* $\gamma(i+\ell)$ *have at least Hamming distance* ℓ*.*

Proof. We prove the claim by induction on ℓ. The base case $\ell = 0$ is trivial as codewords $\gamma(i)$ and $\gamma(i+\ell)$ are identical and have Hamming distance 0.

Next, we show the induction step from $\ell - 1$ to ℓ, where $0 < \ell \leq k$. Intuitively, the claim follows from the observation that when going from $\gamma(i+\ell-1)$ to $\gamma(i+\ell)$, a k-preserving code has to flip a bit that remains unchanged when going from $\gamma(i)$ to $\gamma(i+\ell-1)$. Formally, let $I = \langle i, i+\ell-1\rangle_M$ be an interval containing all numbers

i to $i+\ell-1$. The extended codeword x_I is the superposition of all codewords $\gamma(i)$ to $\gamma(i+\ell-1)$. As γ is k-preserving, we have that $\gamma([M]) \cap \mathrm{res}(x_I) = \gamma(I)$, i.e., the codewords in $\mathrm{res}(x_I)$ are the codewords $\gamma(i)$ to $\gamma(i+\ell-1)$. In particular, codeword $\gamma(i+\ell)$ is not an element of $\mathrm{res}(x_I)$.

Each bit that is flipped when going from $\gamma(i)$ to $\gamma(i+\ell-1)$ becomes $\mathfrak{u}$ in the extended codeword x_I. We obtain the resolution $\mathrm{res}(x_I)$ by replacing each $\mathfrak{u}$ in x_I by 0 and 1. As codeword $\gamma(i+\ell)$ is not an element of $\mathrm{res}(x_I)$, there is at least one stable bit that differs for x_I and $\gamma(i+\ell)$. Since the bit is stable in x_I we obtain that, for all $j \in I$, all codewords $\gamma(j)$ agree on this bit. Hence, $\gamma(i+\ell-1)$ and $\gamma(i+\ell)$ differ in at least one bit on which $\gamma(i)$ and $\gamma(i+\ell-1)$ agree.

It remains to show that that $\gamma(i)$ and $\gamma(i+\ell)$ differ in all bits that are different for $\gamma(i)$ and $\gamma(i+\ell-1)$. As γ is a Gray code $\gamma(i+\ell-1)$ and $\gamma(i+\ell)$ differ in exactly one bit, which is the bit that is stable in x_I. Hence, going from $\gamma(i)$ to $\gamma(i+\ell)$ flips one bit more than going from $\gamma(i)$ to $\gamma(i+\ell-1)$. The induction hypothesis states that $\gamma(i)$ and $\gamma(i+\ell-1)$ have Hamming distance at least $\ell-1$. By the induction hypothesis, we obtain that $\gamma(i)$ and $\gamma(i+\ell)$ have at least Hamming distance ℓ. □

With Lemma 3.11 at hand, we can show that a k-recoverable Gray code can encode at most $2^{n-k}(k+1)+1$ codewords.

Theorem 3.12. *For $k, n \in \mathbb{N}$, the domain of any Gray code $\gamma\colon [M] \to \mathbb{B}^n$ has at most size $M \leq 2^{n-k}(k+1)$, if γ is k-recoverable.*

Proof. Consider a k-recoverable Gray code γ on domain $[M]$ for some $M \in \mathbb{N}$. We show that for a fixed $M = 2^{n-k}(k+1)$ the codomain has at least size 2^n. Consider a family of intervals I_ℓ of size $k+1$. For $\ell \in [2^{n-k}]$ define $I_\ell = \langle \ell(k+1), \ell(k+1)+k \rangle_M$. The family of intervals I_ℓ is a partition of $[M]$. Since γ is k-recoverable, there is an extension $\widetilde{\gamma}^{-1}$ such that $\widetilde{\gamma}^{-1}(\mathrm{res}(x_{I_\ell})) \subseteq I_\ell$, for all ℓ. As $I_\ell \cap I_{\ell'} = \emptyset$, for $\ell, \ell' \in [2^{n-k}]$ and $\ell \neq \ell'$, we obtain that $\mathrm{res}(x_{I_\ell}) \cap \mathrm{res}(x_{I_{\ell'}}) = \emptyset$. Hence, there are 2^{n-k} many distinct intervals that have extended codewords with non-overlapping resolutions.

We show that for each I_ℓ the resolution $\mathrm{res}(x_{I_\ell})$ has size at least 2^k. The resolution $\mathrm{res}(x_{I_\ell})$ contains codewords $\gamma(i)$ and $\gamma(i+k)$, which have at least Hamming distance k (by Lemma 3.11). Thus, x_I has at least k-many $\mathfrak{u}$'s. It follows that $|\mathrm{res}(x_I)| \geq 2^k$. Hence, there are $[2^{n-k}]$ many distinct intervals I_ℓ, with resolutions $\mathrm{res}(x_{I_\ell})$ of size at least 2^k. Accordingly, there have to be at least $2^{n-k} \cdot 2^k = 2^n$ possible words in the codomain. Thus, if $M = 2^{n-k}(k+1)$ the codomain has at least size 2^n.

Finally, fix codomain $\mathbb{B}^n$, i.e., a codomain of size 2^n. Assume for contradiction a domain $M > 2^{n-k}(k+1)$. We partition M into intervals of size $k+1$. There are more than 2^{n-k} intervals with one interval possibly smaller than $k+1$. By the conclusion above we use 2^n words in the codomain for the first 2^{n-k} intervals. The remaining intervals require further words, but all words in the codomain are allocated already. □

Theorem 3.12 shows that, for k-recoverability, an n-bit Gray code γ can encode at most $2^{n-k}(k+1)$ codewords. We conclude that any Gray code with domain $[2^{n-k}(k+1)]$ and codomain $\mathbb{B}^n$ is at most k-recoverable.

Corollary 3.13. *For $k, \ell, n \in \mathbb{N}$, no code $\gamma\colon [M] \to \mathbb{B}^n$ with $|M| > 2^{n-k}(k+1)$ is k-recoverable.*

A direct result of Corollary 3.13 is that no Gray code with $M = 2^n$ can be more than 1-recoverable. In the following section, we present two optimal redundancy codes, one that is not 1-recoverable (standard binary code) and one that is 1-recoverable (BRGC).

3.3 Example Codes

In this section, we present three common codes: standard binary code, unary thermometer code, and BRGC. They differ in their redundancy, preservation, and recoverability properties. Each property is discussed for the respective code. While the standard binary code serves as an example of a well-known code that is unsuitable for hazard-free circuits, the BRGC is extensively used throughout the remainder of this thesis. The unary thermometer code comes in two flavors that can be combined into a new code, the snake-in-the-box code, which we also review.

3.3.1 Standard Binary Code

the standard binary code is also known as the base-2 numeral system

We denote the encoding function of standard binary code by γ_n^{bin}

Definition 3.14. *For $n \in \mathbb{N}_{>0}$ and $i \in [2^n]$ we define the binary encoding function $\gamma_n^{\text{bin}}\colon [2^n] \to \mathbb{B}^n$ by*

$$\gamma_n^{\text{bin}}(i) \coloneqq \gamma_{n-1}^{\text{bin}}(\lfloor i/2 \rfloor) x_n \,,$$

where $x_n \coloneqq (i \bmod 2)$.

The standard binary code has (optimal) redundancy 1, each number in $[2^n]$ can be mapped to a binary string in $\mathbb{B}^n$ and vice versa. From Example 3.3 we observe that the code is not 1-preserving.

Observation 3.15. *The standard binary encoding is not k-preserving, for any $k > 0$.*

Proof. As indicated in Example 3.3, the codewords of $2^{n-1} - 1$ and 2^{n-1} differ in all bits. For any n let $k = 1$ and $I = \left\langle 2^{n-1} - 1, 2^{n-1} \right\rangle_M$. The extended codeword

$$x_I = \gamma_n^{\text{bin}}(2^{n-1} - 1) * \gamma_n^{\text{bin}}(2^{n-1}) = \mathfrak{u} \dots \mathfrak{u}$$

consists of $\mathfrak{u}$'s only. Hence, the resolution of x_I contains all codewords of length n,

$$\gamma_n^{\text{bin}}([2^n - 1]) \cap \operatorname{res}(x_I) = \gamma_n^{\text{bin}}([2^n - 1]) \neq \gamma_n^{\text{bin}}(\{2^{n-1} - 1, 2^{n-1}\}) \,. \qquad \square$$

From Lemma 3.7 we conclude that the standard binary encoding is not recoverable.

Corollary 3.16. *The code γ_n^{bin} is not k-recoverable, for any $k > 0$.*

i	$\gamma_4^{\mathrm{u}}(i)$	$\overline{\gamma}_4^{\mathrm{u}}(i)$
0	0000	1111
1	1000	0111
2	1100	0011
3	1110	0001
4	1111	0000

Table 3.1: Unary thermometer codes γ_4^{u} and $\overline{\gamma}_4^{\mathrm{u}}$.

3.3.2 Unary Thermometer Codes

Next, we define the unary thermometer code in two different flavors indicated by γ_n^{u} and $\overline{\gamma}_n^{\mathrm{u}}$. The code $\overline{\gamma}_n^{\mathrm{u}}$ is obtained from γ_n^{u} by flipping all bits. Hence, both are equally expressive.

Definition 3.17 (Unary Thermometer Codes)**.** *For $n \in \mathbb{N}$ and $i \in [n+1]$ we define the encoding function $\gamma_n^{\mathrm{u}} \colon [n+1] \to \mathbb{B}^n$ for a unary thermometer code by*

$$\gamma_n^{\mathrm{u}}(i) \coloneqq 1^i 0^{n-i} .$$

We also define a unary thermometer code with flipped bits $\overline{\gamma}_n^{\mathrm{u}} \colon [n+1] \to \mathbb{B}^n$ by

$$\overline{\gamma}_n^{\mathrm{u}}(i) \coloneqq \overline{\gamma_n^{\mathrm{u}}(i)} = 0^i 1^{n-i} .$$

Both γ_n^{u} and $\overline{\gamma}_n^{\mathrm{u}}$ for $n = 4$ are listed in Table 3.1. From the table, we observe that any extended codeword of an interval without a wrap-around equals the superposition of its start and end points.

Observation 3.18. *Let $n, p \in \mathbb{N}$ and $i \in [n+1]$. For any $I = \langle i, i+p \rangle_M$, where $i + p \leq n$, it holds that*

$$\begin{aligned} * \gamma_n^{\mathrm{u}}(I) &= \gamma_n^{\mathrm{u}}(i) * \gamma_n^{\mathrm{u}}(i+p) = 1^i \mathtt{u}^p 0^{n-i-p} , \text{ and} \\ * \overline{\gamma}_n^{\mathrm{u}}(I) &= \overline{\gamma}_n^{\mathrm{u}}(i) * \overline{\gamma}_n^{\mathrm{u}}(i+p) = 0^i \mathtt{u}^p 1^{n-i-p} . \end{aligned}$$

The observation shows that, in this case, each extended codeword has i-many stable bits in the front part and $(n - i - p)$-many stable bits in the rear part. In particular, each resolution has the same front and rear parts as the extended codeword. It follows that the resolution of an interval without a wrap-around does not add new codewords.

Observation 3.19. *Let $n, p \in \mathbb{N}$ and $i \in [n+1]$. For any $I = \langle i, i+p \rangle_M$, where $i + p \leq n$, it holds that*

$$\begin{aligned} \gamma_n^{\mathrm{u}}([n+1]) \cap \operatorname{res}(* \gamma_n^{\mathrm{u}}(I)) &= \gamma_n^{\mathrm{u}}(I) , \text{ and} \\ \overline{\gamma}_n^{\mathrm{u}}([n+1]) \cap \operatorname{res}(* \overline{\gamma}_n^{\mathrm{u}}(I)) &= \overline{\gamma}_n^{\mathrm{u}}(I) . \end{aligned}$$

i	$\gamma^{\mathrm{s}}_4(i)$	i	$\gamma^{\mathrm{s}}_4(i)$
0	0000	4	1111
1	1000	5	0111
2	1100	6	0011
3	1110	7	0001

Table 3.2: Snake-in-the-box code γ^{s}_4.

Both codes γ^{u}_n and $\overline{\gamma}^{\mathrm{u}}_n$ are n-preserving up to intervals that wrap around the maximum value, i.e., up to the point where $i + p > n$. As our definition of preserving codes also requires the wrap-around, codes γ^{u}_n and $\overline{\gamma}^{\mathrm{u}}_n$ would only satisfy a weaker notion of n-preserving.

We can overcome this issue by combining both codes by appending the codewords of γ^{u}_n to the codewords of $\overline{\gamma}^{\mathrm{u}}_n$ (cf. Table 3.2). The result is a so-called *snake-in-the-box* code.

Definition 3.20. *For $n \in \mathbb{N}$ and $i \in [2n]$ we define the encoding function $\gamma^{\mathrm{s}}_n \colon [2n] \to \mathbb{B}^n$ for a snake-in-the-box code by*

$$\gamma^{\mathrm{s}}_n(i) := \begin{cases} \gamma^{\mathrm{u}}_n(i) & \text{, if } i \in [n] \\ \overline{\gamma}^{\mathrm{u}}_n(i-n) & \text{, otherwise.} . \end{cases}$$

Snake-in-the-box codes were introduced by Kautz [56] motivated by the research on error-correcting codes. The snake-in-the-box code overcomes the issue of the unary thermometer code, which was not recoverable on intervals with a wrap-around.

Remark. The snake-in-the-box code is a Gray code (cf. Definition 3.10).

Observation 3.21. *Snake-in-the-box code γ^{s}_n is $(n-1)$-preserving.*

Proof. We first prove that γ^{s}_n is $(n-1)$-preserving on intervals that use only one of both encodings. From Observation 3.19 we follow for interval $I = \langle i, i+p \rangle_M$, with either $i \geq 0$ and $i+p \leq n-1$ or $i \geq n$ and $i+p \leq 2n-1$, that $\gamma^{\mathrm{s}}_n(I) = \gamma^{\mathrm{s}}_n([M]) \cap \mathrm{res}(x_I)$.

We show the claim for the remaining intervals in a similar fashion. From Table 3.2 we observe for the interval $I = \langle i, i+p \rangle_M$, where $i \leq n-1$ and $i+p > n-1$, that

$$x_I = \mathtt{u}^{i+p-n} 1^{n-p} \mathtt{u}^{n-i} .$$

Accordingly, for the interval $I = \langle i, i+p \rangle_M$ where $n < i \leq 2n-1$ and $p \in [n]$, we get that

$$x_I = \mathtt{u}^{i+p-n} 0^{n-p} \mathtt{u}^{n-i} .$$

Hence, in both cases, each codeword in the resolution of x_I is a codeword in $\gamma^{\mathrm{s}}_n(I)$, and no codeword outside of $\gamma^{\mathrm{s}}_n(I)$ is added. □

The Snake-in-the-box code γ_n^{s} is an example of a code that increases redundancy to be preserving. Code γ_n^{s} has redundancy $n/(\log n + 1)$. According to Corollary 3.13 this is far from optimal. Despite being $(n-1)$-preserving, γ_n^{s} it is not $(n-1)$-recoverable.

Observation 3.22. *Snake-in-the-box code γ_n^{s} is not $(n-1)$-recoverable.*

Proof. Let $\widetilde{\gamma}^{-1}$ be an extension to the decoding function $(\gamma_n^{\mathrm{s}})^{-1}$. We need to show that there is no $\widetilde{\gamma}^{-1}$ that, for every interval, maps non-codewords to the original interval.

Let $I = \langle 0, n-1\rangle_M$ and $J = \langle n+1, 2n\rangle_M$ such that $x_I = \mathfrak{u}^{n-1}0$ and $x_J = 0\mathfrak{u}^{n-1}$. Note that $I \cap J = \{0\}$. Now regard the word $01^{n-2}0$, which is in both, the resolution of x_I and the resolution of x_J. Hence, any extension $\widetilde{\gamma}^{-1}$ has to map $01^{n-2}0$ to a number that is in both intervals. Thus,

$$\widetilde{\gamma}^{-1}(01^{n-2}0) = 0\,.$$

However, the word $01^{n-2}0$ is also in the resolution of extended codeword $\mathfrak{u}1^{n-2}\mathfrak{u}$, i.e., the extended codeword of $\langle n-1, n+1\rangle_M$. As 0 is not an element of the interval, the extension $\widetilde{\gamma}^{-1}$ does not correctly map the resolutions back to the original interval. Moreover, this proves that there is no extension that can do so. □

3.3.3 Gray Code

Gray codes have the property that two consecutive codewords have Hamming distance 1. That is, to obtain the successor of a codeword exactly one bit is flipped. This is also true in the case of a wrap-around, i.e., when defining 0 as the successor of $M-1$. An example of a Gray code is the snake-in-the-box code γ_n^{s}. In this section, we will discuss the optimal redundancy Binary Reflected Gray Code, which is arguably the most famous Gray code.

Binary Reflected Gray Code. As the name suggests, the code is built from the standard binary code with a reflection process. Intuitively, for an n bit code, we count through all codewords of length $n-1$ with the first bit fixed to 0. Next, the first bit is flipped to 1, while fixing the remaining $n-1$ bits. Afterwards, we reflect the counting order. With the first bit fixed to 1, we count backward through all codewords of length $n-1$. For example, this can be seen in the first two columns of Table 3.3. The code counts through bits three and four before flipping bit two and reflecting the order. Formally, we define γ_n^{g}, the BRGC of length n, by recursion.

Definition 3.23 (BRGC)**.** *For $n \in \mathbb{N}$ we recursively define the encoding function $\gamma_n^{\mathrm{g}} \colon [2^n] \to \mathbb{B}^n$ as follows. A 1-bit BRGC is given by $\gamma_1^{\mathrm{g}}(0) \coloneqq 0$ and $\gamma_1^{\mathrm{g}}(1) \coloneqq 1$. For $n > 1$ and $i \in [2^n]$, the encoding function γ_n^{g} is defined by*

$$\gamma_n^{\mathrm{g}}(i) \coloneqq \begin{cases} 0\gamma_{n-1}^{\mathrm{g}}(i) & \text{, if } i \in [2^{n-1}] \\ 1\gamma_{n-1}^{\mathrm{g}}(2^n - i - 1) & \text{, if } i \in [2^n] \setminus [2^{n-1}]\,. \end{cases}$$

i	$\gamma^{g}_{4}(i)$	i	$\gamma^{g}_{4}(i)$	i	$\gamma^{g}_{4}(i)$	i	$\gamma^{g}_{4}(i)$
0	0000	4	0110	8	1100	12	1010
1	0001	5	0111	9	1101	13	1011
2	0011	6	0101	10	1111	14	1001
3	0010	7	0100	11	1110	15	1000

Table 3.3: Binary Reflected Gray Code γ^{g}_{4}.

For example, the code γ^{g}_{4} is listed in Table 3.3. The BRGC has (optimal) redundancy 1; each number in $[2^n]$ can be mapped to a binary string in $\mathbb{B}^n$ and vice versa. The code γ^{g}_{n} is a Gray code (cf. Definition 3.10).

Lemma 3.24. *For $n \in \mathbb{N}$, given two numbers $i \in [2^n]$ and $j = i + 1 \bmod 2^n$, the codewords $\gamma^{g}_{n}(i)$ and $\gamma^{g}_{n}(j)$ have Hamming distance 1.*

Proof. We show the claim by an inductive argument over n, the length of the code. The base case $n = 1$ is simple, all possible codewords (i.e. 0 and 1) have Hamming distance 1. For the induction step $(n > 1)$ we make a case distinction on i, where in the case $i \in [2^n] \setminus \{2^{n-1} - 1, 2^n - 1\}$ the claim follows from the recursive definition of γ^{g}_{n}. In case $i = 2^{n-1} - 1$, we obtain that

$$\gamma^{g}_{n}(i) = 0\gamma^{g}_{n-1}(2^{n-1} - 1) \text{ and}$$
$$\gamma^{g}_{n}(j) = 1\gamma^{g}_{n-1}(2^{n-1} - 1)\,.$$

Hence, the codewords differ in the first bit and the claim holds. In case $i = 2^n - 1$, we obtain that

$$\gamma^{g}_{n}(i) = 1\gamma^{g}_{n-1}(0) \text{ and}$$
$$\gamma^{g}_{n}(j) = 0\gamma^{g}_{n-1}(0)\,.$$

Hence, the codewords differ in the first bit and the claim holds. □

As BRGC is a Gray code it follows that BRGC is 1-preserving.

Observation 3.25. *The Binary Reflected Gray Code γ^{g}_{n} is 1-preserving.*

Proof. Consider any interval $I = \langle i, i+1 \rangle_M$ with $i \in [M]$. By Lemma 3.24 we know that $\gamma^{g}_{n}(i)$ and $\gamma^{g}_{n}(i+1)$ differ in exactly one bit. Hence, x_I has exactly one $\mathfrak{u}$ bit and thus $\operatorname{res}(x_I) = \{\gamma^{g}_{n}(i), \gamma^{g}_{n}(i+1)\}$. □

We make an observation on extended codewords x_I that have exactly one $\mathfrak{u}$ bit. If $(x_I)_m = \mathfrak{u}$ at position m, then the remaining $n - m$ bits are the maximum codeword of a $(n - m)$-bit code. Note that the maximum codeword of a $(n - m)$-bit code is

$$\gamma^{g}_{n-m}(2^{n-m} - 1) = 10^{n-m-1}\,.$$

The observation is important in the correctness proof of the sorting primitive in Chapter 5.

Observation 3.26. *For x_I, where $I = \langle i, i+1 \rangle_M$, if there is a single index $1 \leq m < n$ such that $(x_I)_m = \mathfrak{u}$, then $(x_I)_{m+1,n} = 10^{n-m-1}$.*

Proof. List the codewords in order. By the recursive definition of the code, removing the first $m-1$ bits of the code leaves us with 2^{m-1} repetitions of a $(n-m+1)$-bit BRGC code alternating between correct order and reflected order. Hence, we only need to regard the case $m = 1$, where the first bit is unstable, the other cases follow from this. Also by the recursive definition, the first bit toggles when the remaining $(n-1)$-bit code reaches its last codeword. The last codeword is $\gamma^{\mathrm{g}}_{n-1}(2^{n-1}-1) = 10^{n-2}$. Thus, for any interval I, where $(x_I)_m = \mathfrak{u}$ for only one m, we get that $(x_I)_{m+1,n} = 10^{n-m-1}$. □

3.4 Follow-Up Questions

Hazard-Free Arithmetic. This chapter leans towards a formal understanding of properties that are desirable for hazard-free circuits. While there are surely circuits that compute the hazard-free extension of a binary increment, Example 3.3 shows that the result of such a circuit is useless.

It is out of the scope of this thesis to evaluate and elaborate more on the properties of encodings that are used in hazard-free circuits. A deeper understanding of such properties is desirable for further study of a hazard-free arithmetic. Hazard-free addition requires a code that is k-recoverable and $2k$-preserving, assuming the same encoding is used for both inputs and the output. If both addends have up to $2k$ unstable bits in total the output has uncertainty up to $2k$. Hence, the encoding should be $2k$-preserving. If each addend has at most k unstable bits, k-recoverability suffices.

Hazard-free arithmetic can be of interest for the Lynch-Welch algorithm from Section 6.3. The algorithm computes the the average of two numbers. In a practical implementation, each input can become metastable. Hence, the algorithm requires hazard-free addition and hazard-free division by 2.

Error Correcting Codes. We claim that further study of error-correcting codes is required. As we point out several times in this chapter, the field of error-correcting codes is closely related to the problems described here. Preservation and recoverability properties are similar to error detection and error correction properties.

The field of error-correcting codes has seen extensive research. It is of interest to formally relate both fields, as a relation shows whether innovative results in one field apply to the other.

Conclusion. Finally, we state the take-home message for this chapter that became a theme throughout this work:

Encoding matters!

Hazard-Free Transducers 4

This chapter presents the results of our work published in ITCS 22 (cf. [17]). We describe a general construction of hazard-free circuits from finite-state transducers. The construction is based on a framework of Ladner and Fischer [65] that constructs Boolean circuits from transducers. It is best known for its construction of fast adders.

Outline. After an introduction in Section 4.1 we discuss related work in Section 4.1.3. With the help of a toy example, Section 4.2 builds intuition and presents the key ideas needed to obtain the main result Theorem 4.20. That is, Section 4.2.1 explains why the Ladner and Fischer framework fails, and Section 4.3.1 introduces the encoding used to resolve the main shortcoming of their approach. Finally, we prove Theorem 4.20 in Section 4.3.3. Afterward, we discuss in detail how to extend the input encoding (cf. Section 4.4) and how to bound parameter k (cf. Section 4.5). Last, we give an outlook in Section 4.6.

4.1 Introduction and Related Work

4.1.1 Introduction

Ikenmeyer et al. proved unconditional lower bounds on the complexity of hazard-free circuits implementing explicit functions [52]. More precisely, they show exponential gaps between the size of several Boolean circuits and their hazard-free counterparts. They show that hazard-free verification circuits for NP-hard problems cannot be of polynomial size unless the circuit equivalent of $\mathrm{P} = \mathrm{NP}$ holds. On the other hand, as we prove in Chapter 5, there are efficient implementations of sorting networks that avoid certain hazards. This shows that hazard-free implementations do not always come at a high cost. That contrast leads to the following question:

Which classes of Boolean functions allow for an efficient hazard-free implementation?

Ladner and Fischer [65] presented a general framework providing an efficient circuit implementation of arbitrary (small) transducers, giving rise to the most efficient adder circuits known to date. While the Ladner and Fischer framework fails to yield hazard-free circuits, the result from [16] suggests the possibility of a general hazard-free construction.

Transducers. A deterministic finite-state transducer is a finite state machine that outputs a symbol on each state transition. We phrase our results for *Mealy machines* [77], but our techniques are not specific to this type of transducer. A Mealy

Mealy and Moore automata have the same computational power

machine is defined by a finite set of states, a starting state, the input and output alphabets, a state transition function, and an output function.

Definition 4.1 (Mealy Machine)**.** *A Mealy machine, denoted by T, is given by the 6-tuple $T = (S, s_0, \Sigma, \Lambda, t, o)$, where*

S	*is the finite set of states,*
$s_0 \in S$	*is the starting state,*
Σ	*is the finite input alphabet,*
Λ	*is the finite output alphabet,*
$t\colon S \times \Sigma \to S$	*is the state transition function, and*
$o\colon S \times \Sigma \to \Lambda$	*is the output function.*

Each Mealy machine induces a *transcription function* mapping a string of input symbols to a string of output symbols of the same length.

Definition 4.2 (Transcription Function τ)**.** *Let $T = (S, s_0, \Sigma, \Lambda, t, o)$ be a Mealy machine and $n \in \mathbb{N}$. The* transcription function *$\tau_{T,n}\colon \Sigma^n \to \Lambda^n$ is given in the following way. Define for $i \in \{1, \dots, n\}$ and $x \in \Sigma^n$ the* state s_i after i steps *inductively via $s_i \coloneqq t(s_{i-1}, x_i)$. Then $\tau_{T,n}(x)_i \coloneqq o(s_{i-1}, x_i)$.*

Remark. Every Boolean function $f\colon \mathbb{B}^n \to \mathbb{B}^m$ can (essentially) be realized by a deterministic finite-state transducer. A simple implementation could read the entire input string x and output $f(x)$ on reception of the last input symbol. This approach, however, requires an exponential number of states $|S| \in \mathcal{O}(2^n)$ to memorize the input. Accordingly, it is of interest to consider *small* transducers. In particular, important basic operations, like addition, max, and min, can be implemented by constant-size transducers.

Parallel Prefix. In the course of presenting their framework, Ladner and Fischer present also the parallel prefix circuit that makes the implementation possible. Essentially, the parallel prefix circuit is a construction that computes the application of a given subcircuit on all prefixes of an input word. An essential insight is that if the function of the subcircuit is associative then the prefixes can be computed in parallel. A parallel prefix circuit hence requires associativity of the given operator.

Definition 4.3 (Parallel Prefix Circuit)**.** *Given input $x \in \Sigma^n$ and an implementation of the associative operator $\circ\colon \Sigma \times \Sigma \Rightarrow \Sigma$, a parallel prefix circuit computes for each $i \in \{1, \dots, n\}$*

$$\pi_i \coloneqq x_1 \circ \ldots \circ x_i \,.$$

The construction of Ladner and Fischer and others are not discussed further in this work. We assume that there is a construction with asymptotically optimal size $\mathcal{O}(n)$ and depth $\mathcal{O}(\log n)$, e.g., the construction of Ladner and Fischer. Parallel prefix circuits are applied as a black box.

4.1.2 Contribution

In this chapter, we establish that constant-size transducers allow for an efficient hazard-free circuit implementation. Denoting by ℓ and m the (constant) number of bits encoding an input symbol and an output symbol, respectively. Denoting by n the length of the input string, by $|S|$ the number of states of the transducer, and by k an upper bound on the number of unstable bits in the input, our main result is as follows.

Theorem 4.20. *For any integers $k \in \mathbb{N}$, $\ell, m, n \in \mathbb{N}_{>0}$ (with $k \leq n$) and Mealy machine $T = (S, s_0, \Sigma = \mathbb{B}^\ell, \Lambda \subseteq \mathbb{B}^m, t, o)$, there is a k-bit hazard-free circuit implementing $\tau_{T,n}$. For $\kappa := \sum_{i=0}^{\min\{|S|,2^k\}} \binom{|S|}{i}$ and $\lambda := \min\{m, 2^{|S|\cdot|\Sigma|}\}$ the circuit has*

$$\text{size } \mathcal{O}\left((\kappa^3 + (2^\ell/\ell)\kappa^2 + 2^\ell\kappa\lambda)n\right)$$
$$\text{and depth } \mathcal{O}\left(\log\kappa \log n + \ell\right) .$$

Remark. The proof of Theorem 4.20 shows that we can save a factor of κ in the third term, provided that the preimage of 1 under $o(\cdot, \sigma)_j$ (i.e., bit j of the output function with the second input fixed to σ) has size at most 2^k for each $\sigma \in \Sigma$ and $j \in [m]$. In this case, there is a k-bit hazard-free circuit implementing $\tau_{T,n}$ of size $\mathcal{O}\left((\kappa^3 + 2^\ell\kappa^2 + 2^\ell\lambda)n\right)$.

The asymptotic complexity depends on k, i.e., the upper bound on the numbers of $\mathfrak{u}$'s. We identify two scenarios that lead to different corollaries of Theorem 4.20. For $k \in \mathbb{N}$ we consider the cases: $2^k \geq |S|$ and $2^k < |S|$, i.e., whether the largest possible resolution (bounded by 2^k) has more elements than the state space. In both cases let $\lambda := \min\{m, 2^{|S|\cdot|\Sigma|}\}$.

First Scenario. If $2^k \geq |S|$, we apply the trivial bound of $2^{|S|}$ for the sum over the binomial coefficients κ. Note that in this case, trivially the preimage of 1 under $o(\cdot, \sigma)$ has size at most 2^k. Hence, as discussed above, the factor of κ in the third term of the size bound can be removed. This gives us the following size and depth bounds for a fully hazard-free implementation.

Corollary 4.4. *For any integers $\ell, n \in \mathbb{N}$ and Mealy machine T, with $\Sigma = \mathbb{B}^\ell$ and $2^k < |S|$, the transcription function $\tau_{T,n}$ can be implemented by a hazard-free circuit*

$$\text{of size } \mathcal{O}((2^{3|S|} + 2^{2|S|+\ell}/\ell + 2^\ell\lambda)n)$$
$$\text{and depth } \mathcal{O}(|S| \log n + \ell) .$$

We stress that this result stands out against the lower bound from [52], which proves an exponential dependence of the circuit size on n, for any general construction of hazard-free circuits. While the above theorem incurs exponential overheads in terms of the size of the transducer, the dependence on n is asymptotically optimal. Thus, for constant-size transducers, we obtain asymptotically optimal hazard-free

implementations of their transcription functions, both for size and depth. More generally, Theorem 4.20 shows that the task of implementing transcription functions is fixed-parameter tractable with respect to $\max\{\ell, |S|\}$.

Second Scenario. If $2^k < |S|$, the Binomial Theorem [47] provides a stronger bound for κ, the sum over the binomial coefficients. Note that the respective factor in the third term of the size bound can still be removed if the output function satisfies the above requirement, but this does not hold in general.

Corollary 4.5. *Given integers $k, \ell, n \in \mathbb{N}$ and Mealy machine T, with $\Sigma = \mathbb{B}^\ell$ and $2^k < |S|$, the transcription function $\tau_{T,n}$ can be implemented by a k-bit hazard-free circuit*

$$\text{of size } \mathcal{O}((|S|^{3\cdot 2^k} + (2^\ell/\ell)|S|^{2\cdot 2^k} + 2^\ell |S|^{2^k} \lambda)n)$$
$$\text{and depth } \mathcal{O}(2^k \log(|S|) \log n + \ell)\,.$$

Encoding Matters. The main insight underlying the proof of Theorem 4.20 is an understanding of how the encoding of a piece of information (such as an input) affects the ability of the circuit to keep track of this information. Due to the ambiguity presented by $\mathfrak{u}$ signals, naive encodings may lose information crucial for determining a stable output, which cannot be recovered later. We tackle this problem by introducing a *universal encoding* that explicitly stores for each $A \subseteq S$ (of size at most 2^k) whether the state machine is currently in a state from A. This redundancy is sufficient to completely eliminate k-bit hazards, yet it is affordable when $|S|$ or k are small.

Remark. The main result holds also for $\Sigma \subseteq \mathbb{B}^\ell$ when choosing an (arbitrary) extension for the transition function to domain $S \times \mathbb{B}^\ell$. However, the choice of how to extend the transition function t matters for the behavior of the hazard-free state machine. The decision on how to treat non-input symbols is important because it is possible that an unstable input resolves to such a non-input symbol. If this happens, the choice of where t maps such resolutions to affects the value the hazard-free extension takes. A bad extension may result in unstable output without need, decreasing the utility of the constructed circuit. The task of finding a useful extension is nontrivial and depends on the application. A detailed discussion at hand of an example is given in Section 4.4.

Note that in some sense the solution is only an incomplete or partial answer to the posed research question. We clarify that every Boolean formula can be computed by a transducer, hence the lower bound applies also to this general construction. However, the presented work gives more insight into which functions have an efficient hazard-free implementation.

Functions that can be implemented by a small transducer have an efficient hazard-free implementation.

Transducer notation introduces various parameters which have different trade-offs and which influence the size of the hazard-free implementation.

4.1.3 Related Work

We already describe related work on the complexity of hazard-free circuits in Section 2.6. In this section, we discuss how our work relates to the work of Ladner and Fischer and how it relates to the lower bound of Ikenmeyer.

Transducers. Our approach can be seen as an extension of the work of Ladner and Fischer [65]. This celebrated result yields the only asymptotically optimal adder constructions known to date, cf. [92]. Alongside the result for binary addition, the authors point out the general applicability of their parallel prefix computation (PPC) framework: for any transcription function, it allows constructing a circuit implementing it. However, as we discuss in detail in Section 4.2, their approach cannot be applied to our setting, as it does not take into account the uncertainty imposed by unstable inputs.

Our approach might also remind the reader of the power set construction [86, Thm. 1.39], which translates a non-deterministic finite-state automation into a deterministic one operating on the power set of the state space. This analogy is correct to the extent that we seek to maintain information on the set of states that are reachable by resolutions of the input. However, Kleene logic has the fundamentally different characteristic that the choice of encoding (e.g. of states) affects to what extent the circuit can keep track of the encoded information. In a nutshell, we prove that it is sufficient to maintain a bit vector indicating for each element $A \subseteq S$ of the power set whether, given the input, all states that could have been reached by the state machine are a subset of A. This resolves an issue that has no connection to the original power set construction.

Parallel Prefix Circuit. The parallel prefix circuit is an arrangement of subcircuits implementing a function $\circ \Sigma \times \Sigma \Rightarrow \Sigma$. The circuit computes all prefixes $x_1 \circ \ldots \circ x_i$ for $i \in \{1, \ldots, n\}$ and input $x \in \Sigma^n$ (cf. Definition 4.3).

In [65] Ladner and Fischer present an implementation of the parallel prefix circuit that is parametrized by $k < \log n$. It achieves

$$\begin{aligned} \text{depth: } & (\log n + k) \cdot \text{depth}(\circ), \text{ and} \\ \text{size: } & (2(1 + 1/2^k)n - k - F_{\log n + 5 - k} + 1) \cdot \text{size}(\circ)\,, \end{aligned}$$

where F_ℓ is the ℓth Fibonacci number. I.e., parameter k offers to trade a smaller size for a larger depth. Note that for $k = 0$ the construction has optimal depth $\log n$.

There is a body of work presenting different constructions for parallel prefix circuits starting from the probably best known constructions [9, 60, 87]. However, there is not a clear state-of-the-art parallel prefix circuit as different constructions optimize for different parameters, such as the maximum fan-out and the number of cross wires (besides size and depth). In addition, we propose a modification of Ladner and Fischer that offers a trade-off between size and maximum fan-out [16].

In this work, we apply the parallel prefix circuit as a black box. We assume that there is a construction with asymptotically optimal size $\mathcal{O}(n)$ and depth $\mathcal{O}(\log n)$, e.g., Ladner and Fischer with $k = \log n - 1$.

Lower Bound. In Section 2.6 we state the result from Ikenmeyer et al. [52], i.e., that an exponential blow-up cannot be avoided for general hazard-free constructions. The lower bound also applies to our contribution. However, it is not perfectly clear to us how the lower bound exactly applies as it is not fixed how a transducer for a function is constructed. A computationally hard function might have an exponential size state space or input alphabet. Both lead to an exponential blow-up in our construction. We further discuss this issue in Section 4.6.

4.2 Classic PPC and Hazards

In this section, we walk the reader through the main idea of the PPC framework by Ladner and Fischer [65]. At the hand of a simple running example, we demonstrate how a naive application of the PPC framework results in circuits that are not hazard-free. We then use the running example to illustrate how to overcome this hurdle and to obtain a hazard-free circuit by making use of the universal encoding.

Running Example. The running example we use throughout this section is an extremely simple transducer: It simply shifts the input sequence by one bit, outputting a 0 on reception of the first symbol; see Figure 4.1 for an illustration. It has two states (referred to as 0 and 1), they are used to keep track of the most recently processed input bit. Hence, the transition and output functions are obvious: the automaton transitions to state $s \in \{0, 1\}$ on reception of input s, and outputs s when leaving state s. Thus, the transducer is formally specified by the 6-tuple

$$(S \coloneqq \{0, 1\},\ s_0 \coloneqq 0,\ \Sigma \coloneqq \{0, 1\},\ \Lambda \coloneqq \{0, 1\},\ t(s, i) = i,\ o(s, i) = s).$$

Clearly, this transducer is a toy example, and it is pointless to construct a circuit implementing its transcription function. This is easily achieved by suitable rewiring of the inputs instead. However, the shift transducer serves as a minimal example for illustrating both the obstacle we need to overcome and the general solution we provide for doing so.

4.2.1 The Classic PPC Framework

In their work, Ladner and Fischer observe that any transcription function $\tau_{T,n}$ on inputs $x \in \mathbb{B}^n$ can be efficiently implemented as a circuit by following four steps. The transition function can be restricted to an input symbol. For input symbol $\sigma \in \Sigma$ the function $t_\sigma : S \to S$ denotes the transition function t restricted to σ; $t_\sigma \coloneqq t(\cdot, \sigma)$. Given an encoding of t_σ the four steps compute for each $i \in \{1, \ldots, n\}$:

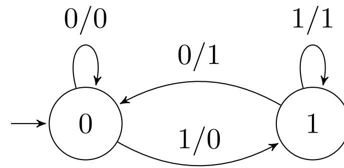

Figure 4.1: The shift transducer delays the input by one symbol. It serves as a running example.

(Step 1) compute (the encoding of) the restricted transition function t_{x_i} of ith input,

(Step 2) compute the composition $\pi_i \coloneqq t_{x_i} \circ \ldots \circ t_{x_1}$ of restricted transition functions for the prefix of i,

(Step 3) compute the ith state, i.e., evaluate $s_i = \pi_i(s_0)$, and

(Step 4) compute the ith output $o(s_{i-1}, x_i) = \tau_{T,n}(x)_i$.

Asymptotics. Steps 1, 3, and 4 can be performed independently and hence in parallel for each i. This means that each of them can be performed by n parallel copies of a circuit whose size (and depth) are independent of n. In contrast, Step 2, the computation of all prefixes, inherently relies on information across all i's. A circuit computing Step 2 will never achieve size and depth independent of n. To achieve a small depth without blowing up the circuit size, Ladner and Fischer exploit the associativity of function composition.

For a constant-size Mealy machine, Steps 1, 3, and 4 can be performed by circuits of size $\mathcal{O}(n)$ and depth $\mathcal{O}(1)$, and Step 2 can be done by a circuit of size $\mathcal{O}(n)$ and depth $\mathcal{O}(\log n)$. In their argument showing this, Ladner and Fischer encode the space of functions $S \to S$ as Boolean $|S| \times |S|$ matrices. Functional composition (Step 2), hence, becomes Boolean matrix multiplication, while evaluation of functions (Step 3) becomes Boolean matrix-vector multiplication.

4.2.2 Running Example.

Applying this to our example, states 0 and 1 of our transducer are represented by the column unit vector $e^{(0)} = \binom{1}{0}$ and $e^{(1)} = \binom{0}{1}$, respectively. Representing $t_\sigma \colon S \to S$ as a Boolean matrix in the natural way, the column corresponding to state s is the unit vector $e^{(t(s,\sigma))}$. Assuming that we make the best effort and use a hazard-free circuit for computing the encodings of t_σ, the hazard-free extension determines what our circuit will compute when receiving $\mathfrak{u}$ as an input symbol. Denote by $\mathcal{M}_{t_\sigma}$ the matrix computed by this hazard-free circuit for (the encoding of) the transition function

	i	0	1	2	3	4
input	x_i	-	0	0	1	0
Step 1, function encoding	$\mathcal{M}_{t_{x_i}}$	-	$\mathcal{M}_{t_0}$	$\mathcal{M}_{t_0}$	$\mathcal{M}_{t_1}$	$\mathcal{M}_{t_0}$
Step 2, function composition	$\mathcal{M}_{\pi_i}$	-	$\mathcal{M}_{t_0}$	$\mathcal{M}_{t_0}$	$\mathcal{M}_{t_1}$	$\mathcal{M}_{t_0}$
Step 3, function evaluation	$s_i = \pi_i(s_0)$	$e^{(0)}$	$e^{(0)}$	$e^{(0)}$	$e^{(1)}$	$e^{(0)}$
Step 4, output	$o(s_{i-1}, x_i)$	-	0	0	0	1

stable input word 0010

	i	0	1	2	3	4
input	x_i	-	0	$\mathfrak{u}$	1	0
Step 1, function encoding	$\mathcal{M}_{t_{x_i}}$	-	$\mathcal{M}_{t_0}$	$\mathcal{M}_{t_\mathfrak{u}}$	$\mathcal{M}_{t_1}$	$\mathcal{M}_{t_0}$
Step 2, function composition	$\mathcal{M}_{\pi_i}$	-	$\mathcal{M}_{t_0}$	$\mathcal{M}_{t_\mathfrak{u}}$	$\mathcal{M}_{t_\mathfrak{u}}$	$\mathcal{M}_{t_\mathfrak{u}}$
Step 3, function evaluation	$s_i = \pi_i(s_0)$	$e^{(0)}$	$e^{(0)}$	$\binom{\mathfrak{u}}{\mathfrak{u}}$	$\binom{0}{\mathfrak{u}}$	$\binom{0}{\mathfrak{u}}$
Step 4, output	$o(s_{i-1}, x_i)$	-	0	0	$\mathfrak{u}$	$\mathfrak{u}$

unstable input word 0u10

Table 4.1: Application of the Ladner and Fischer approach to the example transducer for stable input word 0010 (top) and unstable input word 0u10 (bottom). Gray area: these values do not match the results from a hazard-free computation.

restricted to the input symbol σ. We thus obtain

$$\mathcal{M}_{t_0} = \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}, \quad \mathcal{M}_{t_1} = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}, \quad \mathcal{M}_{t_\mathfrak{u}} = \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix} * \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} \mathfrak{u} & \mathfrak{u} \\ \mathfrak{u} & \mathfrak{u} \end{pmatrix},$$

where the $*$ operator is applied component-wise: as each entry of the computed matrix depends on whether the input symbol was 0 or 1, $\mathfrak{u}$ must result in the all-$\mathfrak{u}$ matrix.

Function composition corresponds to Boolean matrix multiplication, i.e., for restricted functions t_σ and $t_{\sigma'}$ ($\sigma, \sigma' \in \Sigma$), $\mathcal{M}_{t_\sigma \circ t_{\sigma'}} = \mathcal{M}_{t_\sigma} \cdot \mathcal{M}_{t_{\sigma'}}$, where $\cdot$ denotes the Boolean matrix multiplication operator. Similarly, function evaluation corresponds to matrix-vector multiplication, meaning that the framework stipulates to compute the encoding of π_i as $\mathcal{M}_{t_{x_i}} \cdot \ldots \cdot \mathcal{M}_{t_{x_1}}$ and hence s_i as $\mathcal{M}_{t_{x_i}} \cdot \ldots \cdot \mathcal{M}_{t_{x_1}} \cdot e^{(s_0)}$. Again, we make the best effort, i.e., assume that hazard-free circuits are used. Therefore, the circuit will compute $\mathcal{M}_{t_{x_{i-1}}} \cdot_\mathfrak{u} \ldots \cdot_\mathfrak{u} \mathcal{M}_{t_{x_1}} \cdot_\mathfrak{u} e^{(s_0)}$.

Finally, the ith output bit is computed according to the output function by mapping $e^{(0)}$ to output 0 and $e^{(1)}$ to output 1. Note that the redundant representation allows for some freedom: we can choose for $\binom{0}{0}$ and $\binom{1}{1}$ whether to map them to 0 or 1, respectively. As these state vectors cannot occur anyway, this choice has no impact on stable inputs. It might, however, affect the behavior of a (hazard-free) circuit confronted with unstable inputs.

Example Runs. Now consider Table 4.1(a), which breaks down the computation for a stable input string 0010. Any hazard-free circuit should output 0001 in this case. In contrast, Table 4.1(b) breaks down the computation for an input containing a single unstable bit, 0𝔲10. A hazard-free circuit should output 00𝔲1 here. However, multiplication of any function encoding with the all-𝔲 matrix results again in the all-𝔲 matrix, such that any further step of function composition will return $\mathcal{M}_{t_\mathfrak{u}}$. Hence, for the second input, the hazard-free extension of the established approach will compute 𝔲 as the last symbol.

The Problem. To identify the key issue, examine the sequence of matrices determined from the input symbols, which represent the transition functions restricted to the respective input bit. $\mathcal{M}_{t_0}$ will map any vector corresponding to a stable state, i.e., each unit vector, to $e^{(0)}$. This reflects the fact that a 0 is guaranteed to result in state 0. Accordingly, multiplying $\mathcal{M}_{t_0}$ with any matrix representing the transition function restricted to a stable input symbol will result in $\mathcal{M}_{t_0}$: no matter what happened to the state machine before, the state after receiving input symbol 0 is 0 (represented by $e^{(0)}$).

On the other hand, $\mathcal{M}_{t_\mathfrak{u}}$ is the "correct" representation for input symbol 𝔲: regardless of the previous state, the resolutions 0 and 1 of input symbol 𝔲 reach state 0 or 1 respectively, and $\binom{1}{0} * \binom{0}{1} = \binom{\mathfrak{u}}{\mathfrak{u}}$. Unfortunately, (the hazard-free extension of) Boolean matrix multiplication of any matrix with the all-𝔲 matrix $\mathcal{M}_{t_\mathfrak{u}}$ can never yield a matrix that is composed of column unit vectors. The circuit computes

$$\begin{aligned}\mathcal{M}_{t_1} \cdot_\mathfrak{u} \mathcal{M}_{t_\mathfrak{u}} \cdot_\mathfrak{u} \mathcal{M}_{t_0} \cdot_\mathfrak{u} e^{(s_0)} &= \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix} \cdot_\mathfrak{u} \begin{pmatrix} \mathfrak{u} & \mathfrak{u} \\ \mathfrak{u} & \mathfrak{u} \end{pmatrix} \cdot_\mathfrak{u} \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix} \cdot_\mathfrak{u} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 & 0 \\ \mathfrak{u} & \mathfrak{u} \end{pmatrix} \cdot_\mathfrak{u} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 \\ \mathfrak{u} \end{pmatrix}\end{aligned}$$

as the (encoding of) state s_3 in Step 3. Thus, in Step 4 the circuit at its best effort can only output

$$\begin{aligned} o_\mathfrak{u}\left(\begin{pmatrix} 0 \\ \mathfrak{u} \end{pmatrix}, 0\right) &= o\left(\begin{pmatrix} 0 \\ 0 \end{pmatrix}, 0\right) * o\left(\begin{pmatrix} 0 \\ 1 \end{pmatrix}, 0\right) \\ &= o\left(\begin{pmatrix} 0 \\ 0 \end{pmatrix}, 0\right) * 1\,, \end{aligned}$$

which is 1 if $o\left(\binom{0}{0}, 0\right) = 1$ and 𝔲 if $o\left(\binom{0}{0}, 0\right) = 0$.

This might give the false hope of escaping the problem by leveraging our aforementioned freedom to choose $o\left(\binom{0}{0}, 0\right)$ by setting it to 1, but this is a red herring. If $o\left(\binom{0}{0}, 0\right) = 1$, then the input string 0𝔲00 forces the circuit to incorrectly output $o_\mathfrak{u}\left(\binom{\mathfrak{u}}{0}, 0\right) = o\left(\binom{0}{0}, 0\right) * 0 = \mathfrak{u}$ as the final bit.

Intuitively, the main takeaway from this example is that encoding the transition function as an $|S| \times |S|$ matrix is insufficient to keep track of the set of reachable states. The crucial problem is that uncertainty about the transducer's state can be removed (or reduced) by later input symbols. In our example, we have a very simple case: any stable input symbol fully determines the attained state, regardless of the previous state.

4.3 Hazard-Free PPC

In this section, we start with an example to give an intuition to the new encoding of transition functions. We formally define the universal encoding in Section 4.3.2 and we outline the proof of the main Theorem in Section 4.3.3. Afterwards, we prove correct the main Theorem. We exclude two important discussions from the main proof for better readability. We assign them to external sections that are appended to the proof section. Section 4.4 discusses the importance of the input encoding and Section 4.5 presents the effects of bounding k.

In our approach, we keep track of a strict subset of states $A \subset S$ the state machine could have reached when facing some inputs with some uncertainty, such that we can infer the set of states $B \subset S$ (ideally a singleton, if the uncertainty has been completely masked) that can be reached by the current state transition.

4.3.1 Running Example

Before formalizing our encoding, we provide some intuition, by using, again, our toy example, the shift transducer. As we kept the example tiny, the number of subsets of the state space, i.e., the power set of S, is small: the four possible subsets are $\emptyset$, $\{0\}$, $\{1\}$ and $\{0, 1\}$. Already a single $\mathfrak{u}$ input leads to the largest possible uncertainty about the state of the transducer, hence we choose $k = 1$ throughout the example.

New Encoding. To avoid the pitfall discussed in Section 4.2.1, we now choose a highly redundant matrix representation. Fix an input symbol $\sigma \in \{0, 1\}$. The corresponding $2^{|S|} \times 2^{|S|}$ Boolean matrix encodes for each pair of sets $A, B \subseteq S$ whether for each state in A receiving σ as the next input symbol will result in a state from B. Again, assuming that a hazard-free circuit is used to compute the matrix representation, this choice fully determines the matrix $\mathcal{M}_{t_\mathfrak{u}} = \mathcal{M}_{t_0} * \mathcal{M}_{t_1}$ resulting from input symbol $\mathfrak{u}$. Labeling the rows by subsets B and columns by the subsets A, the resulting matrices $\mathcal{M}_{t_0}$, $\mathcal{M}_{t_1}$, and $\mathcal{M}_{t_\mathfrak{u}}$ are

$$\mathcal{M}_{t_0} = \begin{array}{c} \\ \emptyset \\ \{0\} \\ \{1\} \\ \{0,1\} \end{array} \begin{array}{c} \begin{array}{cccc} \emptyset & \{0\} & \{1\} & \{0,1\} \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \end{array},$$

$$\mathcal{M}_{t_1} = \begin{array}{c} \\ \emptyset \\ \{0\} \\ \{1\} \\ \{0,1\} \end{array} \begin{array}{c} \begin{array}{cccc} \emptyset & \{0\} & \{1\} & \{0,1\} \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \end{array},$$

$$\mathcal{M}_{t_\mathfrak{u}} = \begin{array}{c} \\ \emptyset \\ \{0\} \\ \{1\} \\ \{0,1\} \end{array} \begin{array}{c} \begin{array}{cccc} \emptyset & \{0\} & \{1\} & \{0,1\} \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & \mathfrak{u} & \mathfrak{u} & \mathfrak{u} \\ 1 & \mathfrak{u} & \mathfrak{u} & \mathfrak{u} \\ 1 & 1 & 1 & 1 \end{pmatrix} \end{array},$$

respectively. Consider, for example, input symbol 0 and the corresponding matrix $\mathcal{M}_{t_0}$. Each set of states $\{0\}$, $\{1\}$ and $\{0,1\}$ will transition to state 0. As 0 is a subset of $\{0\}$ and $\{0,1\}$ the respective column vectors of the matrix are $\begin{pmatrix}0 & 1 & 0 & 1\end{pmatrix}^\intercal$.

Remark. Each matrix maintains the trivialities that the empty set will always be mapped to a subset of any set (leftmost column), no non-empty set is mapped to a subset of the empty set (top row), and any set will be mapped to a subset of $S = \{0,1\}$ (bottom row). Note also that the submatrices induced by the rows and columns of singleton sets equal those we got in Section 4.2.1. Since we opted for a minimal example with only two states, none of the additional entries depend on the specific transition function. Note that this changes for $|S| > 2$. Crucial to us is the point that the encoding now reflects that even when an input symbol is $\mathfrak{u}$, it remains certain that any resolution of the input must end up in some state. This is reflected in the bottom row of $\mathcal{M}_{t_\mathfrak{u}}$.

Example Run. Applying the framework of Section 4.2.1 with the new encoding to the input string $0\mathfrak{u}10$, this time Step 2 yields for π_3:

$$\begin{aligned}
\mathcal{M}_{t_1} \cdot_\mathfrak{u} \mathcal{M}_{t_\mathfrak{u}} \cdot_\mathfrak{u} \mathcal{M}_{t_0} &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \cdot_\mathfrak{u} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & \mathfrak{u} & \mathfrak{u} & \mathfrak{u} \\ 1 & \mathfrak{u} & \mathfrak{u} & \mathfrak{u} \\ 1 & 1 & 1 & 1 \end{pmatrix} \cdot_\mathfrak{u} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \cdot_\mathfrak{u} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & \mathfrak{u} & \mathfrak{u} & \mathfrak{u} \\ 1 & \mathfrak{u} & \mathfrak{u} & \mathfrak{u} \\ 1 & 1 & 1 & 1 \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \\
&= \mathcal{M}_{t_1}.
\end{aligned}$$

As we can see, multiplying with $\mathcal{M}_{t_1}$ from the left now correctly recovers $\mathcal{M}_{t_1}$, i.e., regardless of previous possibly unstable input symbols, the computed matrix reflects that reading input symbol 1 results in state 1.

Remark. The above behavior is not a corner case due to the small size of our example, a larger example is given in Section 4.5.

A fundamental problem in hazard-free circuits is that the resolution of the superposition may add undesired values (Observation 2.4). Recall that for a Boolean function $f\colon \mathbb{B}^n \to \mathbb{B}^m$, we obtain $f_{\mathfrak{u}}(x)$ by mapping each $y \in \operatorname{res}(x)$ using f and then taking the $*$ operation over the resulting set. The latter might, depending on x and the encoding, lose information, as $\operatorname{res}(f_{\mathfrak{u}}(x))$ might be a strict superset of $f(\operatorname{res}(x))$. This becomes problematic when we subsequently apply some function, e.g., $g : \mathbb{B}^m \to \mathbb{B}$, that is constant on $f(\operatorname{res}(x))$, but not on $\operatorname{res}(f_{\mathfrak{u}}(x))$ for some $x \in \mathbb{T}^n$; we then get that

$$\mathfrak{u} = g_{\mathfrak{u}}(f_{\mathfrak{u}}(x)) = *(g(\operatorname{res}(f_{\mathfrak{u}}(x)))) \neq *(g(f(\operatorname{res}(x))) = (g \circ f)_{\mathfrak{u}}(x) \in \mathbb{B}\,.$$

4.3.2 Universal Encoding

The key idea underlying our solution to this problem is to maintain the information that f maps $\operatorname{res}(x)$ to $f(\operatorname{res}(x))$. As illustrated by the example, the encoding stores for each $A \subseteq \mathbb{B}^n$ and $B \subseteq \mathbb{B}^m$ whether $f(A) \subseteq B$. When composing functions, we then can retrieve the information that $g \circ f$ is constant on $\operatorname{res}(x)$.

The size of the encoding can be reduced if we only regard k-bit hazards. If the number of $\mathfrak{u}$'s in the input x can be bounded by an integer k, then there is also an upper bound on $|f(\operatorname{res}(x))|$. As each $\mathfrak{u}$ has two stable resolutions, we can readily bound $|f(\operatorname{res}(x))| \leq |\operatorname{res}(x)| \leq 2^k$. Hence, the encoding can be reduced to sets $A \subseteq \mathbb{B}^n$ and $B \subseteq \mathbb{B}^m$, where $|A| \leq 2^k$ and $|B| \leq 2^k$. This leads to the following encoding, which is universal in the sense that it gives rise to k-bit hazard-free implementations of arbitrary transducers.

Definition 4.6 (Universal Function Encoding)**.** *Denote by $\mathcal{P}_t(A)$ the set of all subsets of A with cardinality smaller equal to $t \in \{0, \dots, |A|\}$:*

$$\mathcal{P}_t(A) \coloneqq \{A' \subseteq A \,|\, |A'| \leq t\}\,.$$

Given a function $f\colon S \to T$ and $k \in \mathbb{N}$, define

$$\forall A \in \mathcal{P}_{2^k}(S), B \in \mathcal{P}_{2^k}(T)\colon\ (\mathcal{M}_f)_{BA} \coloneqq \begin{cases} 1 & \text{if } f(A) \subseteq B \\ 0 & \text{else.} \end{cases}$$

The Boolean matrix $\mathcal{M}_f$ has dimension

$$|\mathcal{P}_{2^k}(T)| \times |\mathcal{P}_{2^k}(S)| = \sum_{i=0}^{\min\{|T|,2^k\}} \binom{|T|}{i} \times \sum_{i=0}^{\min\{|S|,2^k\}} \binom{|S|}{i}\,.$$

For $s \in S$ and $A \in \mathcal{P}_{2^k}(S)$, define the state vector $e^{(s)}$ via $e^{(s)}_A := 1$ if $s \in A$ and $e^{(s)}_A := 0$ otherwise. Hence, for all $B \in \mathcal{P}_{2^k}(T)$ we have that $\left(\mathcal{M}_f \cdot e^{(s)}\right)_B = 1$ if $f(s) \in B$ and $\left(\mathcal{M}_f \cdot e^{(s)}\right)_B = 0$ otherwise.

Remark. We are mostly interested in the case where $T = S$, since for the restricted transition functions computed in Step 1 we only need to represent functions from S to S. However, we prove the more general statement.

4.3.3 Proof Outline

Our goal in this subsection is to show Theorem 4.20. To this end, we first establish that the above encoding indeed keeps track of all required information to remove uncertainty in case the input allows it. We show that the above matrix representation is a suitable encoding, i.e., we show that the representation is capable of encoding the transition function without dropping information, here the transition function is restricted to a single input symbol.

Ladner and Fischer. Recall that the classic PPC framework computes states s_i by (Step 1) translating input symbol x_i into the matrix representation of t_{x_i}, (Step 2) determining by matrix multiplication the transition function π_i resulting from a sequence of input symbols, and (Step 3) evaluating the transition function π_i on $e^{(s_0)}$ via matrix-vector multiplication. Given state s_i the output at position i can be determined by application of o_{x_i} (Step 4). In the PPC framework we replace the original matrix representation with the new universal function encoding. We show that the universal encoding overcomes the issue of information loss during function composition.

If function composition does not lose information, then repeated application of function composition gives hazard-free transition functions π_i, which will be formalized in Corollary 4.13.

Key Stepping Stone. To show that for the universal encoding the strategy also succeeds in the face of unstable inputs, we need to prove that composing functions and translating the composed function into its matrix representation is equivalent to first translating each function to its matrix representation and then multiplying these matrices. This is captured by the following theorem, which is our key stepping stone towards Theorem 4.20.

Theorem 4.7. *Let $k \in \mathbb{N}$, $f_j \colon S \to T$ for all $j \in J$, $g_i \colon T \to U$ for all $i \in I$, $A \in \mathcal{P}_{2^k}(S)$, and $C \in \mathcal{P}_{2^k}(U)$. If $|J| \cdot |A| \leq 2^k$, then*

$$\left(\left(\mathop{*}_{i \in I} \mathcal{M}_{g_i}\right) \cdot_{\mathfrak{u}} \left(\mathop{*}_{j \in J} \mathcal{M}_{f_j}\right)\right)_{CA} = \left(\mathop{*}_{(i,j) \in I \times J} \mathcal{M}_{g_i \circ f_j}\right)_{CA}.$$

The condition $|J| \cdot |A| \leq 2^k$ may seem non-intuitive at first. The product $|J| \cdot |A|$ corresponds to the number of resolutions of the input that has been processed so far.

The product is bounded by 2^k, i.e., the number of resolutions of k many $\mathfrak{u}$'s. In the application of Theorem 4.7, set J corresponds to the resolutions of respective parts of the input. Set A corresponds to the current state of the transducer, and its size depends on the uncertainty of previous transitions. More intuition is provided in the example in Section 4.5.

Before we prove the key stepping stone we discuss tools that are used in the proof of the main result and the key stepping stone. First, we define hazard-free multiplexers which are used in Step 4 of the PPC framework. Second, we show that there is an efficient implementation of hazard-free matrix multiplication. Third, we introduce monotone resolutions, a technique used in the proofs. Last, we state a recent result on the complexity of hazard-free circuits for general functions, which is applied in the proof of the main theorem.

4.3.4 Hazard-free Matrix Multiplication

For Theorem 4.7 to be of use, we need a circuit implementing $\cdot_{\mathfrak{u}}$, i.e., hazard-free matrix multiplication. The standard Boolean matrix multiplication algorithm is known to be appropriate.

Corollary 4.8 (of [52, Lemma 4.2])**.** *There is a circuit of size* $(2\beta - 1)\alpha\gamma$ *and depth* $\lceil \log \beta \rceil + 1$ *that computes* $\mathcal{A} \cdot_{\mathfrak{u}} \mathcal{B}$ *for matrices* $\mathcal{A} \in \mathbb{T}^{\alpha \times \beta}$ *and* $\mathcal{B} \in \mathbb{T}^{\beta \times \gamma}$*.*

Proof. The standard algorithm for Boolean matrix multiplication is monotone, i.e., does not use negations, and requires for each of the $\alpha\gamma$ entries of $\mathcal{A} \cdot_{\mathfrak{u}} \mathcal{B}$ a binary tree of β `and` gates (the leaves) and $\beta - 1$ `or` gates (internal nodes); monotone circuits are hazard-free. □

We observe that hazard-free Boolean matrix multiplication is associative.

Observation 4.9 ($\cdot_{\mathfrak{u}}$ is associative)**.** *For all* $A \in \mathbb{T}^{\alpha \times \beta}$*,* $B \in \mathbb{T}^{\beta \times \gamma}$*, and* $C \in \mathbb{T}^{\gamma \times \delta}$*, we have that* $(A \cdot_{\mathfrak{u}} B) \cdot_{\mathfrak{u}} C = A \cdot_{\mathfrak{u}} (B \cdot_{\mathfrak{u}} C)$*.*

($\mathbb{T}$, `or`, `and`) is only a (commutative) semiring, as its addition, i.e., `or`, has no inverses.

Proof. As `or` and `and` are associative also on $\mathbb{T}$, this follows by the same straightforward calculation as for matrices over arbitrary (semi)rings. □

To prove Theorem 4.7, we first need to establish that matrix multiplication indeed is equivalent to function composition for stable inputs.

Lemma 4.10. *Let* f *and* g *be functions* $f \colon S \to T$ *and* $g \colon T \to U$*. For all* $A \in \mathcal{P}_{2^k}(S)$ *and* $C \in \mathcal{P}_{2^k}(U)$*, it holds that* $(\mathcal{M}_g \cdot \mathcal{M}_f)_{CA} = (\mathcal{M}_{g \circ f})_{CA}$*.*

Proof. Suppose that $(g \circ f)(A) = g(f(A)) \subseteq C$, such that the matrix $\mathcal{M}_{g \circ f}$ has entry 1 at position CA. If the set A is element of $\mathcal{P}_{2^k}(S)$ then applying the function f to set A results in a set contained in $\mathcal{P}_{2^k}(T)$, i.e., $f(A) \in \mathcal{P}_{2^k}(T)$. By Definition 4.6 we

obtain for matrix $\mathcal{M}_f$ at position $f(A)A$ that $(\mathcal{M}_f)_{f(A)A} = 1$. Additionally, we have that $(\mathcal{M}_g)_{Cf(A)} = 1$, because $g(f(A)) \subseteq C$. Thus,

$$\begin{aligned}((\mathcal{M}_g) \cdot (\mathcal{M}_f))_{CA} &= \sum_{B \in \mathcal{P}_{2^k}(T)} (\mathcal{M}_g)_{CB} (\mathcal{M}_f)_{BA} \\ &\geq (\mathcal{M}_g)_{Cf(A)} (\mathcal{M}_f)_{f(A)A} \\ &= 1 \cdot 1 \\ &= 1\,.\end{aligned}$$

Now consider the case that $(\mathcal{M}_{g\circ f})_{CA} = 0$, i.e., there exists $a \in A$ so that $g(f(a)) \nsubseteq C$. Accordingly,

$$\forall B \in \mathcal{P}_{2^k}(T)\colon\ (\mathcal{M}_f)_{BA} = 1 \Rightarrow (\mathcal{M}_g)_{CB} = 0\,,$$

because

$$(\mathcal{M}_f)_{BA} = 1 \Leftrightarrow f(A) \subseteq B \Rightarrow g(B) \nsubseteq C \Leftrightarrow (\mathcal{M}_g)_{CB} = 0\,.$$

Thus, for all $B \in \mathcal{P}_{2^k}(T)$, we have that $(\mathcal{M}_f)_{BA} (\mathcal{M}_g)_{CB} = 0$, leading to

$$(\mathcal{M}_g \cdot \mathcal{M}_f)_{CA} = \sum_{B \subseteq T} (\mathcal{M}_g)_{CB} (\mathcal{M}_f)_{BA} = 0\,. \qquad \square$$

Monotone Resolution. To understand how multiplying matrices plays out when the multiplicands are not stable, we exploit the monotonicity of matrix multiplication. Because flipping matrix entries of multiplicands from 0 to 1 can only flip entries from 0 to 1 in the product, we can restrict our attention to only two resolutions of each matrix: we simultaneously replace all $\mathfrak{u}$ entries by either 0 or 1, respectively.

Definition 4.11. *For $A \in \mathbb{T}^{\alpha\times\beta}$ and $b \in \mathbb{B}$, define $A^{(b)} \in \mathbb{B}^{\alpha\times\beta}$ via*

$$\forall (i,j) \in \{1,\dots,\alpha\} \times \{1,\dots,\beta\}\colon A^{(b)}_{ij} := \begin{cases} b & \text{if } A_{ij} = \mathfrak{u} \\ A_{ij} & \text{else}\,. \end{cases}$$

With this definition, the above intuition is formalized by the following lemma.

Lemma 4.12. *For all $G \in \mathbb{T}^{\alpha\times\beta}$, $F \in \mathbb{T}^{\beta\times\gamma}$ and all $i \in \{1,\dots,\gamma\}$, $j \in \{1,\dots,\alpha\}$, we have*

$$(G \cdot_{\mathfrak{u}} F)_{ij} = \mathfrak{u} \Leftrightarrow \left(G^{(0)} \cdot F^{(0)}\right)_{ij} = 0 \wedge \left(G^{(1)} \cdot F^{(1)}\right)_{ij} = 1\,.$$

Proof. Fix $i \in \{1,\dots,\gamma\}$ and $j \in \{1,\dots,\alpha\}$. If $(G \cdot_{\mathfrak{u}} F)_{ij} = b \in \mathbb{B}$, i.e., for all G', F' such that $G' \in \operatorname{res}(G)$ and $F' \in \operatorname{res}(F)$, $(G' \cdot F')_{ij} = b$, then since $G^{(0)}, G^{(1)} \in \operatorname{res}(G)$ and $F^{(0)}, F^{(1)} \in \operatorname{res}(F)$ it holds that that

$$\left(G^{(0)} \cdot F^{(0)}\right)_{ij} = \left(G^{(1)} \cdot F^{(1)}\right)_{ij} = b\,.$$

Now consider the case that $(G \cdot_{\mathfrak{u}} F)_{ij} = \mathfrak{u}$. Thus, there are $G', G'' \in \operatorname{res}(G)$ and $F', F'' \in \operatorname{res}(F)$ satisfying that $(G' \cdot F')_{ij} = 0$ and $(G'' \cdot F'')_{ij} = 1$, respectively. It follows that

$$\left(G^{(0)} \cdot F^{(0)}\right)_{ij} = \sum_{k=1}^{\beta} G^{(0)}_{ik} F^{(0)}_{kj} \leq \sum_{k=1}^{\beta} G'_{ik} F'_{kj} = \left(G' \cdot F'\right)_{ij} = 0$$

and, analogously,

$$\left(G^{(1)} \cdot F^{(1)}\right)_{ij} \geq \left(G'' \cdot F''\right)_{ij} = 1\,.$$

□

4.3.5 Proving the Key Stepping Stone

Using Lemma 4.12, proving Theorem 4.7 is reduced to showing correct behavior for matrices $\left(*_{j\in J}\, \mathcal{M}_{f_j}\right)^{(0)}$ and $\left(*_{j\in J}\, \mathcal{M}_{f_j}\right)^{(1)}$ instead of all resolutions of $*_{j\in J}\, \mathcal{M}_{f_j}$ and $*_{i\in I}\, \mathcal{M}_{g_i}$.

Proof of Theorem 4.7. Define $\preceq$ as the partial order $b \prec \mathfrak{u}$ for $b \in \mathbb{B}$ and observe that $*\,X \preceq *\,Y$ for $X \subseteq Y \subseteq \mathbb{B}$. By Observation 2.4, we obtain for $\mathcal{M}_{g_i}$ (and accordingly $\mathcal{M}_{f_j}$) that

$$\{\mathcal{M}_{g_i} | i \in I\} \subseteq \operatorname{res}\left(\mathop{*}_{i\in I} \mathcal{M}_{g_i}\right)\,.$$

Thus, by the definition of the hazard-free extension (cf. Definition 2.7) and the resolution (Definition 2.2),

$$\begin{aligned}
\left(\left(\mathop{*}_{i\in I} \mathcal{M}_{g_i}\right) \cdot_{\mathfrak{u}} \left(\mathop{*}_{j\in J} \mathcal{M}_{f_j}\right)\right)_{CA} &= *\left(\operatorname{res}\left(\mathop{*}_{i\in I} \mathcal{M}_{g_i}\right) \cdot \operatorname{res}\left(\mathop{*}_{j\in J} \mathcal{M}_{f_j}\right)\right)_{CA} \\
&\succeq *\left(\{\mathcal{M}_{g_i} | i \in I\} \cdot \{\mathcal{M}_{f_j} | j \in J\}\right)_{CA} \\
&= \left(\mathop{*}_{(i,j)\in I\times J} \mathcal{M}_{g_i} \cdot \mathcal{M}_{f_j}\right)_{CA} \\
&= \left(\mathop{*}_{(i,j)\in I\times J} \mathcal{M}_{g_i \circ f_j}\right)_{CA}\,,
\end{aligned}$$

where the last equality follows from Lemma 4.10. The claimed equality follows if the l.h.s. equals $b \in \{0, 1\}$.

It remains to show the claimed equality assuming that the l.h.s. equals $\mathfrak{u}$. By application of Lemma 4.12 with $G = *_{i\in I}\, \mathcal{M}_{g_i}$ and $F = *_{j\in J}\, \mathcal{M}_{f_j}$, we obtain that

$$\left(\left(G^{(0)} \cdot F^{(0)}\right)_{CA} = 0\right) \wedge \left(\left(G^{(1)} \cdot F^{(1)}\right)_{CA} = 1\right)\,.$$

By the definition of matrix multiplication, this is equivalent to

$$\forall B \in \mathcal{P}_{2^k}(T)\colon G^{(0)}_{CB} = 0 \vee F^{(0)}_{BA} = 0\,, \tag{4.1}$$

$$\exists B \in \mathcal{P}_{2^k}(T)\colon G^{(1)}_{CB} = 1 \wedge F^{(1)}_{BA} = 1\,. \tag{4.2}$$

We observe from Definition 4.11 that for $b \in \mathbb{B}$ we have that

$$\left(\mathop{*}_{i \in I} \mathcal{M}_{g_i}\right)^{(b)}_{CB} = b \Leftrightarrow \exists i \in I \colon \left(\mathcal{M}_{g_i}\right)_{CB} = b\,;$$

an analogous statement holds for $\mathcal{M}_{f_j}$. We can plug this observation into equations (4.1) and (4.2), then we get that

$$\forall B \in \mathcal{P}_{2^k}(T)\ \exists (i,j) \in I \times J \colon \left(\mathcal{M}_{g_i}\right)_{CB} = 0 \vee \left(\mathcal{M}_{f_j}\right)_{BA} = 0 \tag{4.3}$$

$$\exists B \in \mathcal{P}_{2^k}(T)\ \exists (i,j) \in I \times J \colon \left(\mathcal{M}_{g_i}\right)_{CB} = 1 \wedge \left(\mathcal{M}_{f_j}\right)_{BA} = 1\,. \tag{4.4}$$

Let $B_0 = \bigcup_{j \in J} f_j(A)$ be the subset of T, to which states in A are mapped to by any f_j. As $|f_j(A)| \leq |A|$, cardinality $|B_0|$ is at most $|J| \cdot |A|$. By assumption $|J| \cdot |A| \leq 2^k$, we obtain $B_0 \in \mathcal{P}_{2^k}(T)$. Since $f_j(A) \subseteq B_0$ by construction, it holds that $\left(\mathcal{M}_{f_j}\right)_{B_0 A} = 1$ for all $j \in J$. Equation Equation (4.3) thus entails that

$$\exists i \in I \colon \left(\mathcal{M}_{g_i}\right)_{CB_0} = 0 \Leftrightarrow \exists i \in I \colon g_i(B_0) \nsubseteq C\,.$$

Hence, there are $i_0 \in I$ and $x \in B_0$ such that $g_{i_0}(x) \notin C$. By construction, $x \in f_{j_0}(A)$ for some $j_0 \in J$, yielding that $(g_{i_0} \circ f_{j_0})(A) = g_{i_0}(f_{j_0}(A)) \nsubseteq C$. We conclude that $\left(\mathcal{M}_{g_{i_0} \circ f_{j_0}}\right)_{CA} = 0$.

Now consider equation Equation (4.4), which says that there are indices $i_1 \in I$ and $j_1 \in J$ such that $g_{i_1}(B_1) \subseteq C$ and $f_{j_1}(A) \subseteq B_1$. This immediately yields that $(g_{i_1} \circ f_{j_1})(A) \subseteq C$ and thus $\left(\mathcal{M}_{g_{i_1} \circ f_{j_1}}\right)_{CA} = 1$. The desired equality now follows, because

$$\begin{aligned}
\left(\mathop{*}_{(i,j) \in I \times J} \mathcal{M}_{g_i \circ f_j}\right)_{CA} &\succeq \left(* \left\{\mathcal{M}_{g_{i_0} \circ f_{j_0}}, \mathcal{M}_{g_{i_1} \circ f_{j_1}}\right\}\right)_{CA} \\
&= * \left\{\left(\mathcal{M}_{g_{i_0} \circ f_{j_0}}\right)_{CA}, \left(\mathcal{M}_{g_{i_1} \circ f_{j_1}}\right)_{CA}\right\} \\
&= *\{0, 1\} = \mathfrak{u}
\end{aligned}$$

and $b \succeq \mathfrak{u}$ only holds if $b = \mathfrak{u}$. □

4.3.6 Proving the Main Result

With Theorem 4.7 at our disposal, we are ready to prove our main result, Theorem 4.20. Following Step 1 and Step 2 of the parallel prefix framework given in Section 4.2.1, we need to compute $\pi_i = t_{x_i} \circ \ldots \circ t_{x_1}$ for all prefixes $x_i \ldots x_1$ of the input string. This computation can be phrased in terms of matrix multiplications, which is shown by the following corollary. It readily follows by inductive application of Theorem 4.7.

Corollary 4.13. *Suppose that for $i \in [n]$, we are given mappings $E_i \colon \mathbb{B}^\ell \to F_i$ from input symbols $\mathbb{B}^\ell$ to function spaces F_i. Moreover, for all $i \in [n-1]$ the codomain of functions from F_i equals the domain of functions from F_{i+1}. Let E denote a function,*

with $E\colon \mathbb{B}^{n\ell} \to (F_0 \to F_{n-1})$*, that maps a binary string* $x \in \mathbb{B}^{n\ell}$ *to the composition of the corresponding functions,* $E(x) \coloneqq \circ_{i=0}^{n-1} E_i(x_i)$*. Then, for all* $x \in \mathbb{T}^{n\ell}$*,*

$$\left(\mathcal{M}_{E(\cdot)}\right)_{\mathfrak{u}}(x) = \left(\mathcal{M}_{E_{n-1}(\cdot)}\right)_{\mathfrak{u}}(x_{n-1}) \cdot_{\mathfrak{u}} \left(\mathcal{M}_{E_{n-2}(\cdot)}\right)_{\mathfrak{u}}(x_{n-2}) \cdot_{\mathfrak{u}} \dots \cdot_{\mathfrak{u}} \left(\mathcal{M}_{E_0(\cdot)}\right)_{\mathfrak{u}}(x_0)\,.$$

Following this insight we observe that the evaluation of the functions corresponds to hazard-free matrix-vector multiplication.

Corollary 4.14. *For* $j \in J$*, let* $f_j\colon S \to T$*. Assume that* $A \in \mathcal{P}_{2^k}(T)$*,* $S' \in \mathcal{P}_{2^k}(S)$*, and* $|J| \cdot |S'| \leq k$*. Then*

$$\left(\left(\mathop{*}_{j\in J} \mathcal{M}_{f_j}\right) \cdot_{\mathfrak{u}} \left(\mathop{*}_{s\in S'} e^{(s)}\right)\right)_A = \left(\mathop{*}_{(j,s)\in J\times S'} e^{(f_j(s))}\right)_A .$$

Proof. Define $g_s\colon \{\bullet\} \to S$ by $g_s(\bullet) \coloneqq s$ for $s \in S$, such that $(\mathcal{M}_{g_s})_{\{\bullet\}A} = \left(e^{(s)}\right)_A$. By Theorem 4.7, we thus get that

$$\begin{aligned}
\left(\left(\mathop{*}_{j\in J} \mathcal{M}_{f_j}\right) \cdot_{\mathfrak{u}} \left(\mathop{*}_{s\in S'} e^{(s)}\right)\right)_A &= \left(\left(\mathop{*}_{j\in J} \mathcal{M}_{f_j}\right) \cdot_{\mathfrak{u}} \left(\mathop{*}_{s\in S'} \mathcal{M}_{g_s}\right)\right)_{\{\bullet\}A} \\
&= \left(\mathop{*}_{(j,s)\in J\times S'} \mathcal{M}_{g_s\circ f_j}\right)_{\{\bullet\}A} \\
&= \left(\mathop{*}_{(j,s)\in J\times S'} e^{(f_j(s))}\right)_A .
\end{aligned}$$

□

Prefix Multiplication. Corollary 4.13 uses the hazard-free matrix product of all input prefixes. This can be efficiently implemented, similarly to the parallel prefix computation approach of Ladner and Fischer.

Corollary 4.15 (of [65, Section 2])**.** *For input matrices* $\mathcal{A}_{n-1}, \dots, \mathcal{A}_0$ *of size* $\alpha \times \alpha$*, there is a circuit computing the hazard-free Boolean matrix multiplication of all prefixes;* $\mathcal{A}_i \cdot_{\mathfrak{u}} \dots \cdot_{\mathfrak{u}} \mathcal{A}_0$*, for each* $i \in [n]$*. The circuit*

$$\begin{aligned}
&\textit{has size } \mathcal{O}\left(\alpha^3 n\right) \\
&\textit{and depth } \mathcal{O}\left(\log\alpha \log n\right) .
\end{aligned}$$

Proof. By Observation 4.9, $\cdot_{\mathfrak{u}}$ is associative. For an associative operator, Ladner and Fischer [65] present a family of circuits computing the application of all prefixes of the input. Let c be the size and d the depth of a circuit implementing the operator. The family has asymptotically optimal size $\mathcal{O}(cn)$ and depth $\mathcal{O}(d \log n)$. Corollary 4.8 offers an implementation of hazard-free Boolean $\alpha \times \alpha$ matrix multiplication of size $c = \alpha^3$ and depth $d = \log\alpha$. □

Output Step. Before putting the above pieces together to derive our main results, we need to address how the final output is computed, i.e., Step 4. Here, we can exploit that (i) the output does not need to be represented in the universal encoding, and (ii) the universal encoding used in the previous computations holds additional information that simplifies determining the output in a hazard-free way. We leverage these points in a case analysis to minimize the cost of the output stage.

The input to Step 4 is the vector encoding state s_{i-1} and input x_i for each $i \in \{1 \dots n\}$; it computes output $o(s_{i-1}, x_i)$. We restrict the output function to an input symbol, as we did for the transition function, i.e., $o_\sigma : S \to \Lambda$ for $\sigma \in \Sigma$ is defined by $o_\sigma(s) \coloneqq o(s, \sigma)$. In what follows we describe the computation of output bit $o_\sigma(s)_j$ $(= o(s, \sigma)_j)$, for $j \in [m]$. We remark that the preimage of 1 under $o_\sigma(s)_j$ is a set of states. Recall that by Definition 4.6, the state vector $e^{(s_{i-1})}$ encodes not only state s_{i-1}, but indicates for each subset of states $S' \subseteq S$ (with cardinality less or equal to 2^k) whether $s_{i-1} \in S'$. Thus, we can simply check whether s_{i-1} lies in the set that of states which $o_\sigma(s_{i-1})_j$ maps to 1 (provided its cardinality is at most 2^k).

Definition 4.16. *For an input symbol $\sigma \in \Sigma$ and $j \in [m]$, we define*

$$A_{\sigma,j} \coloneqq \{s \in S | o(s, \sigma)_j = 1\} .$$

Note that if $2^k < |S|$, we reduce the size of the universal encoding by encoding only sets of size less or equal to 2^k. Hence, we might not have computed a bit indicating whether $s_{i-1} \in A_{\sigma,j}$ as an entry of the vector $e^{(s_{i-1})}$. However, essentially all output bits are conveniently available if $\max_{\sigma \in \Sigma, j \in [m]} |A_{\sigma,j}| \leq 2^k$, which is captured by the following lemma.

Lemma 4.17. *For $k \in \mathbb{N}$ and $S' \subseteq S$, $\Sigma' \subseteq \Sigma$, if $\max_{\sigma \in \Sigma, j \in [m]} |A_{\sigma,j}| \leq 2^k$ we have that*

$$\mathop{\ast}_{s \in S', \sigma \in \Sigma'} o(s, \sigma)_j = \mathop{\ast}_{s \in S', \sigma \in \Sigma'} e^{(s)}_{A_{\sigma,j}} . \tag{4.5}$$

Proof. First, as $2^k \geq \max_{\sigma \in \Sigma', j \in [m]} |A_{\sigma,j}|$, by Definition 4.6 every $A_{\sigma,j}$ is encoded in the vector $e^{(s)}$, i.e., every entry $e^{(s)}_{A_{\sigma,j}}$ exists.

Next, we distinguish three cases for every evaluation of the l.h.s. of Equation (4.5): 1, 0, and $\mathfrak{u}$. In the first case, $\ast_{s \in S', \sigma \in \Sigma} o(s, \sigma)_j = 1$, we apply Definition 4.16 and Definition 4.6 to show the claim, as follows.

$$\begin{aligned}
\mathop{\ast}_{s \in S', \sigma \in \Sigma'} o(s, \sigma)_j = 1 \quad &\Leftrightarrow \quad \forall s \in S', \sigma \in \Sigma' : o(s, \sigma)_j = 1 \\
&\Leftrightarrow \quad \forall s \in S', \sigma \in \Sigma' : s \in A_{\sigma,j} && \textit{by Def. 4.16} \\
&\Leftrightarrow \quad \forall s \in S', \sigma \in \Sigma' : e^{(s)}_{A_{\sigma,j}} = 1 && \textit{by Def. 4.6} \\
&\Leftrightarrow \quad \mathop{\ast}_{s \in S', \sigma \in \Sigma'} e^{(s)}_{A_{\sigma,j}} = 1
\end{aligned}$$

The second case, $*_{s\in S',\sigma\in\Sigma} o(s,\sigma)_j = 0$, is treated analogously.

$$\begin{aligned}
\mathop{*}_{s\in S',\sigma\in\Sigma'} o(s,\sigma)_j = 0 \quad &\Leftrightarrow \quad \forall s\in S', \sigma\in\Sigma' : o(s,\sigma)_j = 0 \\
&\Leftrightarrow \quad \forall s\in S', \sigma\in\Sigma' : s\notin A_{\sigma,j} && \textit{by Def. 4.16} \\
&\Leftrightarrow \quad \forall s\in S', \sigma\in\Sigma' : e^{(s)}_{A_{\sigma,j}} = 0 && \textit{by Def. 4.6} \\
&\Leftrightarrow \quad \mathop{*}_{s\in S',\sigma\in\Sigma'} e^{(s)}_{A_{\sigma,j}} = 0
\end{aligned}$$

In case the l.h.s. of Equation (4.5) evaluates to $\mathfrak{u}$ the statement follows from the first two cases. As we showed equivalence in case 1 and 0, we deduce, that in case the l.h.s. of Equation (4.5) evaluates to $\mathfrak{u}$, the r.h.s. of Equation (4.5) also evaluates to $\mathfrak{u}$. □

If the size of the largest preimage ($\max_{\sigma\in\Sigma,j\in[m]}|A_{\sigma,j}|$) exceeds 2^k not all preimages have a corresponding entry in $e^{(s_{i-1})}$. We need to compute whether s_{i-1} is in the preimage. To this purpose, we define a cover of $A_{\sigma,j}$ with sets of size at most 2^k.

Definition 4.18. *For an input symbol $\sigma\in\Sigma$, $j\in[m]$, and $k\in\mathbb{N}$ we define $\mathcal{A}^k_{\sigma,j}$, the cover of $A_{\sigma,j}$ containing sets of cardinality smaller or equal to 2^k:*

$$\mathcal{A}^k_{\sigma,j} := \begin{cases} \{A_{\sigma,j}\} & \text{if } |A_{\sigma,j}| \le 2^k\,, \\ \{A\subseteq A_{\sigma,j} \,|\, |A| = 2^k\} & \text{else.} \end{cases}$$

If the state of the transducer is in one of the sets in $\mathcal{A}^k_{\sigma,j}$, then it is also in $A_{\sigma,j}$. All sets in $\mathcal{A}^k_{\sigma,j}$ have a corresponding entry in the state vector. We can take the `or` over all entries to see whether the transducer is in a state of $A_{\sigma,j}$ and hence whether it outputs 1.

Lemma 4.19. *For $k\in\mathbb{N}$, $j\in[m]$, $S'\subseteq S$, and $\Sigma'\subseteq\Sigma$ with $2^k < \max_{\sigma\in\Sigma,j\in[m]}|A_{\sigma,j}|$, if $|S'|\le 2^k$ we have that*

$$\mathop{*}_{s\in S',\sigma\in\Sigma'} o(s,\sigma)_j = \mathop{*}_{\sigma\in\Sigma'} \bigvee_{A\in\mathcal{A}^k_{\sigma,j}} \mathop{*}_{s\in S'} e^{(s)}_A \tag{4.6}$$

Proof. Every $A\in\mathcal{A}^k_{\sigma,j}$ has cardinality smaller or equal to 2^k. Hence, there is an entry in vector $e^{(s)}$ corresponding to A, i.e., entry $e^{(s)}_A$ exists in the encoding. From its definition we observe that $\mathcal{A}^k_{\sigma,j}$ is indeed a cover of $A_{\sigma,j}$, i.e.,

$$\bigcup_{A\in\mathcal{A}^k_{\sigma,j}} A = A_{\sigma,j}\,. \tag{4.7}$$

Moreover, Definition 4.18 ensures that every subset of $A_{\sigma,j}$ of size at most 2^k is contained in at least one set from $\mathcal{A}^k_{\sigma,j}$. On the other hand, each $A\in\mathcal{A}^k_{\sigma,j}$ is a subset of $A_{\sigma,j}$. Hence, the assumption that $|S'|\le 2^k$ implies that

$$S'\subseteq A_{\sigma,j} \Leftrightarrow \exists A\in\mathcal{A}^k_{\sigma,j} : S'\subseteq A\,. \tag{4.8}$$

We make a case distinction on every possible evaluation of the l.h.s. of Equation (4.6) and show the equality for each case. The first case is $*_{s\in S',\sigma\in\Sigma'} e^{(s)}_{A_{\sigma,j}} = 1$. We get that

$$
\begin{aligned}
\underset{s\in S',\sigma\in\Sigma'}{*} o(s,\sigma)_j = 1 \quad &\Leftrightarrow \quad \forall s \in S', \sigma \in \Sigma' : o(s,\sigma)_j = 1 \\
&\Leftrightarrow \quad \forall s \in S', \sigma \in \Sigma' : s \in A_{\sigma,j} \\
&\Leftrightarrow \quad \forall \sigma \in \Sigma' : S' \subseteq A_{\sigma,j} \\
&\Leftrightarrow \quad \forall \sigma \in \Sigma' : \exists A \in A_{\sigma,j} : S' \subseteq A \qquad \textit{by Eq. } (4.8) \\
&\Leftrightarrow \quad \forall \sigma \in \Sigma' : \exists A \in A_{\sigma,j} : \forall s \in S' : s \in A \\
&\Leftrightarrow \quad \forall \sigma \in \Sigma' : \exists A \in A_{\sigma,j} : \forall s \in S' : e^{(s)}_A = 1 \\
&\Leftrightarrow \quad \forall \sigma \in \Sigma' : \exists A \in A_{\sigma,j} : \underset{s\in S'}{*} e^{(s)}_A = 1 \\
&\Leftrightarrow \quad \forall \sigma \in \Sigma' : \bigvee_{A\in\mathcal{A}^k_{\sigma,j}} \underset{s\in S'}{*} e^{(s)}_A = 1 \\
&\Leftrightarrow \quad \underset{\sigma\in\Sigma'}{*} \bigvee_{A\in\mathcal{A}^k_{\sigma,j}} \underset{s\in S'}{*} e^{(s)}_A = 1.
\end{aligned}
$$

The second case, $*_{s\in S',\sigma\in\Sigma'} e^{(s)}_{A_{\sigma,j}} = 0$, is treated similarly, where now $S' \cap A_{\sigma,j} = \emptyset$ for each $\sigma \in \Sigma'$.

$$
\begin{aligned}
\underset{s\in S',\sigma\in\Sigma}{*} o(s,\sigma)_j = 0 \quad &\Leftrightarrow \quad \forall s \in S', \sigma \in \Sigma' : o(s,\sigma)_j = 0 \\
&\Leftrightarrow \quad \forall s \in S', \sigma \in \Sigma' : s \notin A_{\sigma,j} \\
&\Leftrightarrow \quad \forall s \in S', \sigma \in \Sigma', A \in \mathcal{A}^k_{\sigma,j} : s \notin A \\
&\Leftrightarrow \quad \forall s \in S', \sigma \in \Sigma', A \in \mathcal{A}^k_{\sigma,j} : e^{(s)}_A = 0 \\
&\Leftrightarrow \quad \forall \sigma \in \Sigma', A \in \mathcal{A}^k_{\sigma,j} : \underset{s\in S'}{*} e^{(s)}_A = 0 \\
&\Leftrightarrow \quad \forall \sigma \in \Sigma' : \bigvee_{A\in\mathcal{A}^k_{\sigma,j}} \underset{s\in S'}{*} e^{(s)}_A = 0 \\
&\Leftrightarrow \quad \underset{\sigma\in\Sigma'}{*} \bigvee_{A\in\mathcal{A}^k_{\sigma,j}} \underset{s\in S'}{*} e^{(s)}_A = 0
\end{aligned}
$$

In the final case, i.e., that the l.h.s. of Equation (4.6) evaluates to $\mathfrak{u}$, equality follows from the equivalence established in the previous two cases. □

Main Theorem. We are left with the task of showing that indeed the obtained circuit is correct, i.e., prove Theorem 4.20. The correctness of the construction is

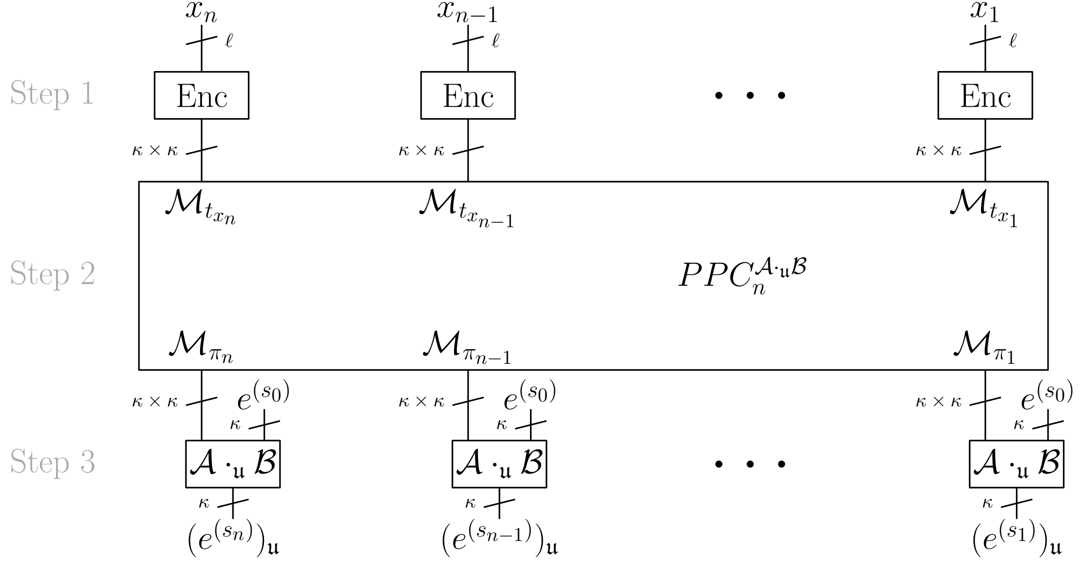

Figure 4.2: Steps 1 to 3 of the circuit implementing the transcription function. Enc denotes the computation of the universal encoding by the generic construction of [55]. Hazard-free Boolean matrix multiplication is denoted by $\mathcal{A} \cdot_{\mathfrak{u}} \mathcal{B}$.

proven foremost by the application of Corollary 4.13 and Corollary 4.14. The proof is mainly concerned with establishing the size and depth bound for the obtained circuit. Without going into detail, for constant $|S|$ the reader should be convinced that the depth of the circuit is logarithmic in n, because all operations except for Step 2 can be computed in parallel, while Step 2 exploits the associativity of matrix multiplication to obtain a circuit of depth logarithmic in n. The size of the circuit is linear in n, as Step 1, Step 3 and Step 4 each use a constant number of operations for each input symbol and Step 2 can be performed asymptotically optimally with a linear number of operations.

Theorem 4.20. *For any integers $k \in \mathbb{N}$, $\ell, m, n \in \mathbb{N}_{>0}$ (with $k \leq n$) and Mealy machine $T = (S, s_0, \Sigma = \mathbb{B}^\ell, \Lambda \subseteq \mathbb{B}^m, t, o)$, there is a k-bit hazard-free circuit implementing $\tau_{T,n}$. For $\kappa := \sum_{i=0}^{\min\{|S|,2^k\}} \binom{|S|}{i}$ and $\lambda := \min\{m, 2^{|S| \cdot |\Sigma|}\}$ the circuit has*

$$\text{size } \mathcal{O}\left((\kappa^3 + (2^\ell/\ell)\kappa^2 + 2^\ell \kappa\lambda)n\right)$$
$$\text{and depth } \mathcal{O}\left(\log \kappa \log n + \ell\right) .$$

Proof. We show that there is a circuit computing the hazard-free extension of the transcription function $(\tau_{T,n})_{\mathfrak{u}}(x)$ for every $x \in \Sigma^n$, where we replace at most k many

bits with $\mathtt{u}$'s. We follow the steps of the PPC framework presented in Section 4.2.1 to compute output $((\tau_{T,n})_{\mathtt{u}}(x))_i = o_{\mathtt{u}}(s_{i-1}, x_i)$ at each position $i \in \{1 \dots n\}$.

The block diagram of steps 1 to 3 of the circuit is depicted in Figure 4.2. The circuit mostly consists of hazard-free matrix multiplication blocks. In particular, the n-input parallel prefix circuit is an arrangement of hazard-free matrix multiplication blocks.

Step 1 computes the universal encoding of the restricted transition function t_{x_i} (a $\kappa \times \kappa$ matrix) from input x_i. Noting that the computation in Step 1 evaluates a function from $\mathbb{B}^\ell$ to $\mathbb{B}^{\kappa^2}$, we can directly apply Theorem 2.14 for each output bit separately. Hence, there is a hazard-free encoding circuit of size $\mathcal{O}((2^\ell/\ell)\kappa^2)$ and depth $\mathcal{O}(\ell)$ implementing this step. Thus, the computation of the universal encoding of t_{x_i} for each i in parallel has size $\mathcal{O}((2^\ell/\ell)\kappa^2 n)$ and depth $\mathcal{O}(\ell)$.

Step 2 computes the encoding $\mathcal{M}_{\pi_i}$ of the composition $\pi_i = t_{x_i} \circ \ldots \circ t_{x_1}$. We define $E_j(x_j) = t(\cdot, x_j)$ for $j \in \{1, \ldots, i\}$ such that for $E(x) = \circ_{j=1}^{i} E_j(x_j)$ we have

$$(\mathcal{M}_{\pi_i})_{\mathtt{u}} = \left(\mathcal{M}_{E(x)}\right)_{\mathtt{u}} = \left(\mathcal{M}_{E(\cdot)}\right)_{\mathtt{u}}(x)\,.$$

Application of Corollary 4.13 then yields

$$\left(\mathcal{M}_{E(\cdot)}\right)_{\mathtt{u}}(x) = \left(\mathcal{M}_{E_j(\cdot)}\right)_{\mathtt{u}}(x_j) \cdot_{\mathtt{u}} \left(\mathcal{M}_{E_{j-1}(\cdot)}\right)_{\mathtt{u}}(x_{j-1}) \cdot_{\mathtt{u}} \cdots \cdot_{\mathtt{u}} \left(\mathcal{M}_{E_1(\cdot)}\right)_{\mathtt{u}}(x_1)\,. \quad (4.9)$$

By Corollary 4.15, there is an efficient circuit computing the r.h.s. of (4.9) for each i. The Corollary gives also size $\mathcal{O}(\kappa^3 n)$ and depth $\mathcal{O}(\log\kappa \log n)$ for Step 2.

Step 3 computes the column unit vector corresponding to the ith state, i.e., evaluation of the composition π_i on the initial state s_0. By Corollary 4.14, we can compute

$$\left(e^{(s_i)}\right)_{\mathtt{u}} = \left(e^{(\pi_i(s_0))}\right)_{\mathtt{u}}$$

by matrix-vector multiplication of $(\mathcal{M}_{\pi_i})_{\mathtt{u}}$ and $\left(e^{(s_0)}\right)_{\mathtt{u}} = e^{(s_0)}$. Hence, by Corollary 4.8 there is a k-bit hazard-free circuit that computes s_i. Evaluation of π_i in parallel for each i yields size $\mathcal{O}(\kappa^2 n)$ and depth $\mathcal{O}(\log\kappa)$ for Step 3.

Finally, Step 4 computes the ith output $o(s_{i-1}, x_i)$ for each $i \in \{1 \dots n\}$. W.l.o.g., assume that the width of an output symbol is 1, i.e., $m = 1$ and hence $\Lambda \subseteq \mathbb{B}$ (otherwise repeat the computation for each output bit separately). Viewing the computation in Step 4 as a function from $\mathbb{B}^{\kappa+\ell}$ to $\mathbb{B}$, we could apply Theorem 2.14 to obtain a circuit of size $\mathcal{O}(2^{\kappa+\ell}/(\kappa+\ell))$ and depth $\mathcal{O}(\kappa+\ell)$. We now show how to obtain better results, bounding the size by $\mathcal{O}(2^\ell\kappa)$ with a circuit of depth $\mathcal{O}(\log\kappa + \ell)$.

Recall Definition 4.16. As $j = 0$ we omit j from the notation and write A_σ and $\mathcal{A}_\sigma^k$ instead of $A_{\sigma,j}$ and $\mathcal{A}_{\sigma,j}^k$. First, consider the case that $\max_{\sigma\in\Sigma} |A_\sigma| \leq 2^k$. A depiction of this case is given in Figure 4.3 (1). By directly using the outputs of the gates computing the respective bits of (the universal encoding of) the state vector, we readily obtain $o(s_{i-1}, \sigma)$ for each $\sigma \in \Sigma$. Thus, we are left with the task of choosing the output corresponding to input x_i. We can do so by using a $\mathtt{MUX}$. By Lemma 2.13, the implementation $\mathtt{MUX}(e_{A_\sigma}^{(s_{i-1})}, \ldots, e_{A_{\sigma'}}^{(s_{i-1})}, x_i)$ has size $\mathcal{O}(2^\ell)$ and depth $\mathcal{O}(\ell)$, where

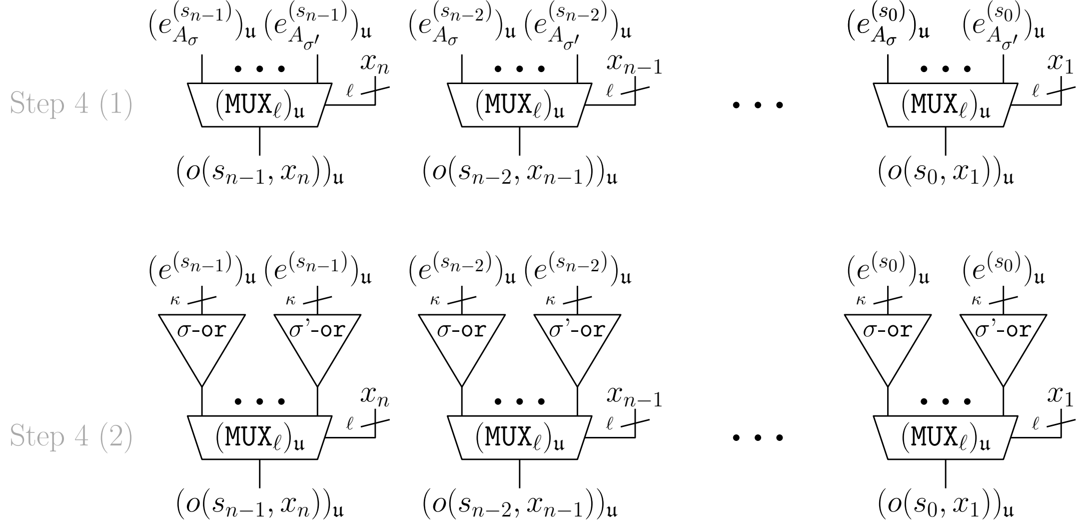

Figure 4.3: Step 4, assuming $m = 1$. (1) Every preimage is encoded in the state vector. (2) At least one preimage is not encoded in the state vector. To increase readability we do not enumerate all elements of Σ but use σ, σ', and dots to denote that the step is repeated for every element of Σ.

σ, σ' are representatives for every input symbol in Σ. If $m > 1$, the multiplexer is copied m times and wired accordingly. Step 4 can be performed in parallel for each i, hence the resulting size and depth bounds are $\mathcal{O}(2^\ell mn)$ and $\mathcal{O}(\ell)$, respectively.

The other case is that $\max_{\sigma\in\Sigma} |A_\sigma| > 2^k$. A depiction of the corresponding circuit is given in Figure 4.3 (2), where σ-`or` denotes the `or`-tree over all $e_A^{(s_i)}$ for $A \in \mathcal{A}_\sigma^k$. Here, we compute the output $o(s_{i-1}, \sigma)$ as the `or` over all entries corresponding to $\mathcal{A}_\sigma^k$ in the state vector. Correctness readily follows from Lemma 4.19. To bound the size and depth of the resulting circuit, observe that the cardinality of $\mathcal{A}_\sigma^k$ is bounded by $\binom{|S|}{2^k}$, hence each `or`-tree has size $\mathcal{O}(\binom{|S|}{2^k})$ and depth $\mathcal{O}(\log \binom{|S|}{2^k})$. As in the previous case, the ith output is selected by a multiplexer. Hence, applying Lemma 2.13, we get that in this case Step 4 requires size $\mathcal{O}(\binom{|S|}{2^k}2^\ell mn)$ and depth $\mathcal{O}(\log \binom{|S|}{2^k} + \ell)$.

Furthermore, there is a natural cap on m. Similar to the previous paragraph, consider each bit of the output separately, i.e., for each position of an output symbol consider the function $o\colon S \times \Sigma \to \mathbb{B}$. For inputs from S and Σ there are $2^{|S|\cdot|\Sigma|}$ different one bit functions. Hence, we can enumerate all possible output functions. If $m > 2^{|S|\cdot|\Sigma|}$, we simply compute and reuse as often as needed the output of each of these possible functions, rather than replicating computations for identical output bits.

Finally, we can derive the asymptotic size and depth of the presented k-bit hazard-free implementation of the transcription function $\tau_{T,n}$. We distinguish two cases

depending on the size of the largest preimage of o. In both cases m is capped at $2^{|S|\cdot|\Sigma|}$ as discussed above. Assume $\max_{\sigma\in\Sigma}|A_\sigma| \leq 2^k$, then the presented circuit has

$$\begin{aligned} \text{size } & \mathcal{O}\left((\kappa^3 + (2^\ell/\ell)\kappa^2 + 2^\ell m)n\right), \text{ and} \\ \text{depth } & \mathcal{O}\left(\log\kappa \log n + \ell\right). \end{aligned}$$

In case $\max_{\sigma\in\Sigma}|A_\sigma| > 2^k$ (and hence $|S| > 2^k$) we bound $\binom{|S|}{2^k}$ by κ, such that the circuit has

$$\begin{aligned} \text{size } & \mathcal{O}\left((\kappa^3 + (2^\ell/\ell)\kappa^2 + 2^\ell \kappa m)n\right), \text{ and} \\ \text{depth } & \mathcal{O}\left(\log\kappa \log n + \ell\right). \end{aligned}$$

□

Remark. The main result holds also for $\Sigma \subseteq \mathbb{B}^\ell$, when choosing an (arbitrary) extension for the transition function to domain $S\times\mathbb{B}^\ell$. However, the choice of how to extend the transition function t matters for the behavior of the hazard-free state machine. The decision on how to treat non-input symbols is important because it is possible that an unstable input resolves to such a non-input symbol. If this happens, the choice of where t maps such resolutions to affects the value the hazard-free extension takes. A bad extension may result in unstable output without need, decreasing the utility of the constructed circuit. The task of finding a useful extension is nontrivial and depends on the application. A detailed discussion at hand of an example is given in Section 4.4.

4.4 Extension of the Input Encoding

The main result, Theorem 4.20, assumes that each binary string of length ℓ is part of the input alphabet, i.e., $\Sigma = \mathbb{B}^\ell$. In this section, we discuss the case where Σ is a strict subset of $\mathbb{B}^\ell$. We choose extensions of the transition function $t\colon \Sigma \times S$ and output function $o\colon \Sigma \times S$ to domain $\mathbb{B}^\ell \times S$. The choice of extension is arbitrary, for any transition function and output function defined on $\mathbb{B}^\ell$ the construction described in the proof of the main theorem computes the hazard-free extension of the transcription function. An arbitrary extension may lead to undesirable results, however. We discuss this issue at hand of an example. Note that both, t and o have to be extended and the same problems may arise in both functions. We construct an example, that provides intuition on the problem, by extending the output function o. The transition function t is not discussed in further detail.

Running Example. Consider a simple finite-state transducer with a single state ($S = \{0\}$) with single bit outputs (output alphabet $\Lambda = \mathbb{B}$). Input symbols have three bits ($\ell = 3$) and the output function computes the `and` over the three input bits. See Table 4.2 for the output table. For the sake of the example, assume that input 000 never occurs, such that the input alphabet is defined by $\Sigma = \mathbb{B}^3 \setminus \{000\}$. Even though

σ	000	001	010	100	011	101	110	111
$o(\sigma, 0)$	-	0	0	0	0	0	0	1
(1)	0	0	0	0	0	0	0	1
(2)	1	0	0	0	0	0	0	1

Table 4.2: Output function of the example transducer and extensions (1) and (2).

the example is artificial, in the sense that it could be solved by simpler circuits, it is worth the discussion as it provides intuition to the problem.

Unknown inputs (which we model by $\mathfrak{u}$) may arise from transitioning input signals. Due to the analog behavior of signal transitions, it might be the case that a bit is neither digital 0 nor 1 for some time during the transition. The signal can float near the threshold for 0 or 1 and the circuit designer can not be sure whether the electronics interpret the signal as digital 0 or 1.

Extension. For an application of the framework, we have to define an extension to o on input 000. As there is only a single output bit, we are left with exactly two choices:

$$o(0, 000) = 0\,, \text{ or} \tag{1}$$

$$o(0, 000) = 1\,. \tag{2}$$

Assume the input switches from 001 to 010, such that the input transitions via $0\mathfrak{u}\mathfrak{u}$ for some time. We choose extension (2), then the framework computes the hazard-free extension of the output function

$$\begin{aligned} o_{\mathfrak{u}}(0, 0\mathfrak{u}\mathfrak{u}) &= *\{o(0, 000), o(0, 001), o(0, 010), o(0, 011)\} \\ &= *\{1, 0\} = \mathfrak{u}\,. \end{aligned}$$

In the sense of hazard-freedom, this is perfectly fine, because we compute the most precise output for all specified inputs and the extension. However, input 001 and 010 both produce output 0. Hence, the output has no transition when switching from 001 to 010. But, due to the chosen extension, the output is $\mathfrak{u}$ during the transition.

Contrary to that, if we choose extension (1) we obtain that the output signal remains 0 during the transition of the inputs;

$$\begin{aligned} o_{\mathfrak{u}}(0, 0\mathfrak{u}\mathfrak{u}) &= *\{o(0, 000), o(0, 001), o(0, 010), o(0, 011)\} \\ &= *\{0\} = 0\,. \end{aligned}$$

Also one can verify that for each input transition, the output remains stable unless there is a transition in the output (for any input transition to or from 111). Hence, the advisable choice of the extension for the specific application is extension (1).

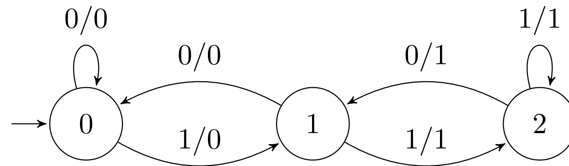

Figure 4.4: Extension of the first example transducer, that shifts to outputting ones only after the reception of two consecutive ones. Vice versa the transducer shifts to outputting zeros upon reception of two consecutive zeros.

The example shows that finding an extension of the transition function, in case Σ is a strict subset of $\mathbb{B}^\ell$, is not a straightforward task. We refrain from defining a notion of the optimum choice and generating the according extension, as this is beyond the scope of this work.

4.5 Bound on k

Running Example. We present an extension of our earlier example, the shift transducer. Here, we switch from output 0 to output 1 only if we see two consecutive 1's in the input. Vice versa, we switch from 1's to 0's only we see two consecutive 0 inputs. The transducer has three states $S = \{0, 1, 2\}$. It is depicted in Figure 4.4.

Bound. What we seek to demonstrate is the impact of bounding k. Accordingly, we choose $k = 1$, such that $2^k = 2 < 3 = |S|$. Thus, the encoding takes into account $\mathcal{P}_2(S) = \mathcal{P}(S) \setminus S$, i.e., all proper subsets of S. For restricted transition functions t_0, t_1, and $t_\mathfrak{u}$, the encoding yields the matrices given in Figure 4.5.

In matrix $\mathcal{M}_{t_\mathfrak{u}}$, we see again the same behavior as in the naive encoding of the shift transducer. Columns corresponding to $\{0, 1\}$, $\{1, 2\}$, and $\{0, 2\}$ contain only 0 and $\mathfrak{u}$ entries. These columns hence cannot prevent the resolution from containing the all 0's vector, which cannot be mapped to a single state in a way that guarantees correct output.

However, columns corresponding to $\{0\}$, $\{1\}$, and $\{2\}$ can be mapped to states without uncertainty, presuming the next input symbol allows it. Taking a look at Figure 4.4, we can see that the transducer goes back to state 0 after the reception of two consecutive 0 inputs, regardless of the previous state. For the sake of this example, consider the input sequence $\mathfrak{u}00$. Hence, we need to calculate the matrix corresponding to $t_0 \circ t_0 \circ t_\mathfrak{u}$. The calculation can be seen in Figure 4.6.

Here it becomes apparent why we need the assumption $|A| \cdot |J| \leq 2^k$ in Theorem 4.20. In matrix $\mathcal{M}_{t_0 \circ t_0 \circ t_\mathfrak{u}}$ only column unit vectors corresponding to singletons can be mapped to stable states. Column vectors corresponding to sets $\{0, 1\}$, $\{1, 2\}$ and $\{0, 2\}$ contain only 0's and $\mathfrak{u}'s$, no 1's. Hence, they cannot be "properly" mapped

$$\mathcal{M}_{t_0} = \begin{array}{c} \\ \emptyset \\ \{0\} \\ \{1\} \\ \{2\} \\ \{0,1\} \\ \{1,2\} \\ \{0,2\} \end{array} \begin{array}{ccccccc} \emptyset & \{0\} & \{1\} & \{2\} & \{0,1\} & \{1,2\} & \{0,2\} \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 \end{array}$$

$$\mathcal{M}_{t_1} = \begin{array}{c} \\ \emptyset \\ \{0\} \\ \{1\} \\ \{2\} \\ \{0,1\} \\ \{1,2\} \\ \{0,2\} \end{array} \begin{array}{ccccccc} \emptyset & \{0\} & \{1\} & \{2\} & \{0,1\} & \{1,2\} & \{0,2\} \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \end{array}$$

$$\mathcal{M}_{t_\mathfrak{u}} = \begin{array}{c} \\ \emptyset \\ \{0\} \\ \{1\} \\ \{2\} \\ \{0,1\} \\ \{1,2\} \\ \{0,2\} \end{array} \begin{array}{ccccccc} \emptyset & \{0\} & \{1\} & \{2\} & \{0,1\} & \{1,2\} & \{0,2\} \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & \mathfrak{u} & \mathfrak{u} & 0 & \mathfrak{u} & 0 & 0 \\ 1 & \mathfrak{u} & 0 & \mathfrak{u} & 0 & 0 & 0 \\ 1 & 0 & \mathfrak{u} & \mathfrak{u} & 0 & \mathfrak{u} & 0 \\ 1 & 1 & \mathfrak{u} & \mathfrak{u} & \mathfrak{u} & \mathfrak{u} & \mathfrak{u} \\ 1 & \mathfrak{u} & \mathfrak{u} & 1 & \mathfrak{u} & \mathfrak{u} & \mathfrak{u} \\ 1 & \mathfrak{u} & 1 & \mathfrak{u} & \mathfrak{u} & \mathfrak{u} & 0 \end{array}$$

Figure 4.5: Matrix encodings of the transition functions t_0, t_1, and $t_\mathfrak{u}$.

$$\mathcal{M}_{t_0 \circ t_0 \circ t_\mathfrak{u}} = \mathcal{M}_{t_0} \cdot \mathcal{M}_{t_0} \cdot \mathcal{M}_{t_\mathfrak{u}}$$
$$= \mathcal{M}_{t_0} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & \mathfrak{u} & \mathfrak{u} & \mathfrak{u} & \mathfrak{u} & \mathfrak{u} \\ 1 & 0 & \mathfrak{u} & \mathfrak{u} & 0 & \mathfrak{u} & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & \mathfrak{u} & \mathfrak{u} & \mathfrak{u} \\ 1 & 0 & \mathfrak{u} & \mathfrak{u} & 0 & \mathfrak{u} & 0 \\ 1 & 1 & \mathfrak{u} & \mathfrak{u} & \mathfrak{u} & \mathfrak{u} & \mathfrak{u} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & \mathfrak{u} & \mathfrak{u} & \mathfrak{u} \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & \mathfrak{u} & \mathfrak{u} & \mathfrak{u} \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & \mathfrak{u} & \mathfrak{u} & \mathfrak{u} \end{pmatrix}.$$

Figure 4.6: Matrix multiplication of the encoding of transition functions t_0 and $t_\mathfrak{u}$.

to any state. We apply Theorem 4.20 to the last step, i.e., the multiplication of $\mathcal{M}_{t_0}$ and $\mathcal{M}_{t_0 \circ t_\mathfrak{u}}$. Here, set J is mapped to res(0$\mathfrak{u}$) yielding $|J| = 2$. As $k = 1$, condition $|A| \cdot |J| \leq 2^k$ is only satisfied for $|A| = 1$ (A cannot be the empty set). Thus, the condition is only satisfied for singleton sets, which is what we observed from the example.

4.6 Follow-Up Questions

Small Transducers. In the introduction, we state that the presented work is not a complete answer to the question of which classes of functions allow for an efficient hazard-free implementation. We show that small transducers can be efficiently implemented in a hazard-free manner. So far the design of the transducer is left to the reader.

The construction we present introduces some parameters that influence the complexity of the resulting circuit. First and foremost the size of the state space, the sizes of the input and the output encoding. We do not fully understand how the complexity of a function relates to the parameters. With a better understanding, we would be able to show how the lower bound of Ikenmeyer et al. [52] applies.

To answer the question of which classes of functions allow for small transducers further study of finite-state machines is required. Transducers are very powerful tools and to us, it is not clear how to achieve small transducers in general. A simple transducer could for example read the whole input before computing the output upon reception of the last symbol. Another transducer could read the whole input at once and output the result in one step. However, as these approaches either require a large state space or a large in-/output alphabet they will be inefficient in the presented construction.

The question that arises can be formulated as follows: How does the complexity of a function relate to the size of the smallest transducer computing the function?

This question is solely related to the study of finite-state machines, it is out of scope for this work.

Applications. An application of the framework can be seen in Chapter 5. We apply the construction to the sorting primitive, such that we obtain an efficient hazard-free sorting circuit. Despite the application to the sorting primitive, we are interested in finding further applications to the framework.

Hazard-free implementations of arithmetic functions are of interest, e.g., for further steps towards an implementation of the Lynch-Welch algorithm in Chapter 6. After sorting the algorithm computes the average of clock values. This requires hazard-free addition and hazard-free division by 2. Needless to say, hazard-free arithmetic goes hand in hand with the development of preserving and recoverable codes (Chapter 3). In the scope of this work, we can only cover the groundwork for this goal.

Hazard-Free Sorting 5

This chapter presents (a part of) the results published in Transactions on Computers [16]. We describe how to sort N inputs, where each input is a binary word that may have an unstable bit. The key to the sorting primitive is a comparator circuit (COMP) that sorts two inputs. The circuit construction makes use of the framework presented in Chapter 4. Furthermore, we improve the construction by adjusting the framework.

Outline. Section 5.1 covers an introduction to the topic and related work. We discuss Sorting networks in Section 5.2. Formally the specification of the circuit is given in Section 5.3, before a transducer implementation and the hazard-free implementation are given in Section 5.4 resp. Section 5.5.

5.1 Introduction and Related Work

5.1.1 Introduction

We describe a variant of the clock synchronization algorithm by Lundelius-Welch and Lynch in Section 6.3. The Lynch-Welch algorithm is a distributed fault-tolerant algorithm that runs on fully connected networks. Essential for fault-tolerance of the algorithm is an approximate agreement step [35]. Differences between each clock in the network are measured and sorted. The algorithm picks measurements $f+1$ and $n-f$ to compute the adjustments. Hence, up to f faulty nodes can be tolerated.

Clocks in the system are uncorrelated. Hence, any digital measurement of the difference between two clocks inherently bears the risk of un-/metastable signals [74]. Any circuit sorting these measurements has to account for possible instability. Friedrichs et al. show in [41] that it is possible to implement the Lynch-Welch algorithm with digital components, given that the components are metastability-containing. Inputs are restricted to binary words that may have an unstable bit only in specific positions. Such inputs result from a suitable choice of measurement circuits [43].

Instability or metastability may or may not resolve during computation. As described in Chapter 2 we consider the worst-case propagation of unstable signals, i.e., outputs of the sorting circuit also might be unstable. According to Friedrichs et al. the subsequent step, frequency adjustment, maintains the precision of unstable signals for careful choice of oscillators. The step involves mapping digital to analog values. Suitable oscillators are those that are controlled by analog effects, e.g., voltage controlled oscillators (VCOs) such as [2, 91] that will be discussed in Section 8.2. The implementation has a guaranteed end-to-end uncertainty of a single bit.

Our goal is to sort any number of inputs that are encoded in BRGC encoding. We first show that we can sort an arbitrary number of inputs by designing a circuit

that sorts two inputs and plugging them into a sorting network. Then we show how to implement an asymptotically optimal, hazard-free comparator.

5.1.2 Related Work

We cover related work on transducers in Section 4.1.3 and related work on hazard-free circuits in Section 2.6. Sorting networks are discussed in Section 5.2.

Lynch-Welch Algorithm. The first to suggest a hazard-free implementation of the Lynch-Welch algorithm were Friedrichs et al. [41]. They identify 4 steps to implement the algorithm, where step 1 (mapping from analog to digital) bears the risk of metastable signals. Hence, subsequent steps either need to use synchronizers or metastability-containing circuits. As the use of synchronizers, which only lower the risk of metastability, is too time-consuming, it is advisable to use metastability-containing circuits.

Inputs to the sorting circuit are called valid strings. Friedrichs et al. show the existence of hazard-free circuits comparing two valid strings and returning the maximum and minimum. The valid strings may arise from a suitable design of time to digital converters (TDCs) and appropriate encoding. Memory-efficient TDC circuits that already return a valid string at their output are presented by Függer et al. [43].

Sorting. The presented work is the last part in a line of research focusing on the implementation of a valid string comparator. After Friedrichs et al. showed that it is possible to implement the circuit [41], a first implementation has been given in [68]. The authors describe a recursive circuit design that has size $\mathcal{O}(n^2)$ and depth $\mathcal{O}(n)$.

We present in [15] an improvement to the recursive design of [68]. By splitting up the recursive design into a sorting and a controlling part we achieve near-optimal size $\mathcal{O}(n \log n)$ and asymptotically optimal depth $\mathcal{O}(\log n)$.

The best implementation, to our knowledge, is given in [16]. We present the design, based on a transducer implementation, in this chapter. Practical simulations in [16] show that the design improves delay by 48.46% and chip area by 71.58% in a 16-bit design. Also, the simulations indicate that the design performs similarly to a (non-hazard-free) binary design, after transistor-level optimization of both.

5.2 Sorting Networks

Sorting networks (see, e.g., [59]) sort N-many inputs from a totally ordered universe by feeding them into N parallel wires that are connected by `COMP` elements, i.e., subcircuits sorting two inputs; these can act in parallel whenever they do not depend on each other's output. A correct sorting network sorts all possible inputs, i.e., the wires are labeled 1 to N such that the ith wire outputs the ith element of the sorted list of inputs. The *size* of a sorting network is its number of `COMP` elements and its

depth is the maximum number of `COMP` elements an input may pass through until reaching the output.
The 0-1-principle [59] states that a sorting network, assuming the `COMP` circuits are correct, is correct if and only if it sorts 0-1 inputs correctly. Thus, we obtain sorting networks for inputs that may suffer from unstable inputs by constructing `COMP` circuits (w.r.t. a suitable order on such inputs) and plugging them into existing sorting networks.

Sorting networks have been extensively studied. Tight lower bounds of depth $\mathcal{O}(\log N)$ (trivial) and size $\mathcal{O}(N \log N)$ (see, e.g., [27]) are known and can be simultaneously asymptotically matched [3]. More practically, for small values of N optimal depth and/or size networks are known [20, 26, 59]. Accordingly, our task boils down to finding optimal (or close to optimal) hazard-free `COMP` circuits. For n-bit inputs, our `COMP` circuits have depth and size $\mathcal{O}(\log n)$ and $\mathcal{O}(n)$, respectively, which is (trivially) optimal up to constants.

5.3 Comparator Specification

In this section, we formally define the comparator primitive which will be implemented by the circuit presented in Section 5.5. The primitive sorts two inputs according to their encoded value. It has two outputs; (1) the input that encodes the larger value will be called the maximum, and (2) the input that encodes the smaller value will be called the minimum. We restrict the set of possible inputs and denote it by the set of valid strings.

Valid Strings. The set of valid strings is a subset of all BRGC encoded strings that have no or one unstable bit. The BRGC encoding is defined in Chapter 3. We denote the set of valid strings of length $n \in \mathbb{N}$ by $\mathcal{S}_n^{\mathrm{g}}$. It contains all BRGC strings of length n and all superpositions of two consecutive codes. As shown in Lemma 3.24, two consecutive BRGC strings have Hamming distance one, such that their superposition has exactly one unstable bit. For ease of notation we denote the set of all BRGC strings of length n by Γ_n^{g}, with

$$\gamma_n^{\mathrm{g}}([2^n]) = \Gamma_n^{\mathrm{g}} = \mathbb{B}^n\,,$$

and we denote the decoding function of the BRGC code by

$$(\gamma_n^{\mathrm{g}})^{-1}(\cdot) = \langle \cdot \rangle_n^{\mathrm{g}}\,.$$

Intuitively, Γ_n^{g} is the set of all inputs that purely consists of binary values, and $\mathcal{S}_n^{\mathrm{g}}$ extends it to ternary inputs. For example, $\mathcal{S}_4^{\mathrm{g}}$ the set of valid strings of length 4 is given in Table 5.1.

I	$*\gamma_4^{\mathrm{g}}(I)$	I	$*\gamma_4^{\mathrm{g}}(I)$	I	$*\gamma_4^{\mathrm{g}}(I)$	I	$*\gamma_4^{\mathrm{g}}(I)$
$\{0\}$	0000	$\{4\}$	0110	$\{8\}$	1100	$\{12\}$	1010
$\{0,1\}$	000u	$\{4,5\}$	011u	$\{8,9\}$	110u	$\{12,13\}$	101u
$\{1\}$	0001	$\{5\}$	0111	$\{9\}$	1101	$\{13\}$	1011
$\{1,2\}$	00u1	$\{5,6\}$	01u1	$\{9,10\}$	11u1	$\{13,14\}$	10u1
$\{2\}$	0011	$\{6\}$	0101	$\{10\}$	1111	$\{14\}$	1001
$\{2,3\}$	001u	$\{6,7\}$	010u	$\{10,11\}$	111u	$\{14,15\}$	100u
$\{3\}$	0010	$\{7\}$	0100	$\{11\}$	1110	$\{15\}$	1000
$\{3,4\}$	0u10	$\{7,8\}$	u100	$\{11,12\}$	1u10		

Table 5.1: All valid strings of length 4, $\mathcal{S}_4^{\mathrm{g}}$.

Definition 5.1 (Set of Valid Strings)**.** *The set of valid strings $\mathcal{S}_n^{\mathrm{g}}$ of length n is given by the union of all BRGC codes of length n and the superposition of any two consecutive BRGC codes. Formally,*

$$\mathcal{S}_n^{\mathrm{g}} := \Gamma_n^{\mathrm{g}} \cup \bigcup_{x \in [2^n-1]} *\{\gamma_n^{\mathrm{g}}([x]), \gamma_n^{\mathrm{g}}([x+1])\} \,.$$

Remark. The set of valid strings is more restricted than the set of all BRGC encoded strings that have no or one unstable bit, because u may only occur in certain positions. However, the set naturally emerges from the outputs of TDCs [43]. Hence, restricting the inputs to valid strings is suitable for the intended application.

5.3.1 Hazard-free Sorting

Total Order on Valid Strings. We aim to build a circuit that computes the maximum and minimum of two valid strings. First, however, we need to answer the question of what it means to ask for the maximum or minimum of valid strings. While there is a natural order on Γ_n^{g}, we need to define an order on the valid strings that contain an unstable bit as well. To this end, suppose a valid string is $*\{\gamma_n^{\mathrm{g}}(x), \gamma_n^{\mathrm{g}}(x+1)\}$ for some $x \in [N-1]$. The unstable bit makes it impossible to interpret this string as x or $x+1$. Resolution of the valid string will cover $\gamma_n^{\mathrm{g}}(x)$ and $\gamma_n^{\mathrm{g}}(x+1)$. Accordingly, it makes sense to consider $*\{\gamma_n^{\mathrm{g}}(x), \gamma_n^{\mathrm{g}}(x+1)\}$ in between $\gamma_n^{\mathrm{g}}(x)$ and $\gamma_n^{\mathrm{g}}(x+1)$, resulting in the following total order on valid strings.

Definition 5.2 (Total order on valid strings ($\succ$))**.** *Let $g, h \in \Gamma_n^{\mathrm{g}}$ be two BRGC strings. We define the total order $\succ$ by*

$$g \succ h \Leftrightarrow \langle g \rangle_n^{\mathrm{g}} > \langle h \rangle_n^{\mathrm{g}} \,.$$

Furthermore, let $x \in [2^n-1]$, then

$$\gamma_n^{\mathrm{g}}(x+1) \succ *\{\gamma_n^{\mathrm{g}}(x), \gamma_n^{\mathrm{g}}(x+1)\} \succ \gamma_n^{\mathrm{g}}(x) \,.$$

We extend the resulting relation on $\mathcal{S}_n^{\mathrm{g}} \times \mathcal{S}_n^{\mathrm{g}}$ to a total order by taking the transitive closure.

Remark. From Definition 5.2 we can define $\succeq$ via $g \succeq h \Leftrightarrow (g = h \vee g \succ h)$.

Our aim, hence, amounts to building a circuit that computes the maximum and the minimum with respect to this order. It turns out that a circuit that sorts two BRGC codes and that is hazard-free also sorts valid strings.

Comparator circuit $\texttt{COMP}_n$. First, we formally define $\texttt{COMP}_n$. Any circuit that implements $\texttt{COMP}_n$ computes the maximum and minimum of two BRGC codes.

Definition 5.3 (Maximum and minimum of BRGC codes)**.** *Let $g, h \in \Gamma_n^{\mathrm{g}}$, then $\max^{\mathrm{g}}(g,h)\colon \Gamma_n^{\mathrm{g}} \times \Gamma_n^{\mathrm{g}} \to \Gamma_n^{\mathrm{g}}$ and $\min^{\mathrm{g}}(g,h)\colon \Gamma_n^{\mathrm{g}} \times \Gamma_n^{\mathrm{g}} \to \Gamma_n^{\mathrm{g}}$ are defined by*

$$\max^{\mathrm{g}}(g,h) := \begin{cases} g & \text{if } \langle g\rangle_n^{\mathrm{g}} \geq \langle h\rangle_n^{\mathrm{g}}\,, \\ h & \text{otherwise.} \end{cases} \qquad \min^{\mathrm{g}}(g,h) := \begin{cases} h & \text{if } \langle g\rangle_n^{\mathrm{g}} \geq \langle h\rangle_n^{\mathrm{g}}\,, \\ g & \text{otherwise.} \end{cases}$$

The comparator circuit $\texttt{COMP}_n$ gets two inputs and computes two outputs. It sorts the inputs w.r.t. the natural order on Γ_n^{g}, i.e., it computes $\max^{\mathrm{g}}$ and $\min^{\mathrm{g}}$.

Definition 5.4 ($\texttt{COMP}_n$)**.** *Let $g, h \in \Gamma_n^{\mathrm{g}}$, then*

$$\texttt{COMP}_n(g,h) := (\max^{\mathrm{g}}(g,h), \min^{\mathrm{g}}(g,h))\,.$$

Sorting Valid Strings. We now show that any circuit that implements the hazard-free extension of $\texttt{COMP}_n$ sorts its inputs according to the order ($\succ$). To this end, we prove that the hazard-free extension of functions $\max^{\mathrm{g}}$ and $\max^{\mathrm{g}}$ compute the maximum and minimum respectively.

Lemma 5.5. *Let $g, h \in \mathcal{S}_n^{\mathrm{g}}$. If and only if $g \succeq h$, then*

$$\max_{\mathrm{u}}^{\mathrm{g}}(g,h) = g\,,$$
$$\min_{\mathrm{u}}^{\mathrm{g}}(g,h) = h\,.$$

Proof. Let $g, h \in \mathcal{S}_n^{\mathrm{g}}$, we prove the claim by case distinction on the cases $g \succ h$, $g \prec h$, and $g = h$. First, if $g \succ h$, Definition 5.1 and Definition 5.2 imply for all $g' \in \mathrm{res}(g)$ and all $h' \in \mathrm{res}(h)$ that $g' \succeq h'$. Hence, $\max^{\mathrm{g}}(g',h') = g'$ and $\min^{\mathrm{g}}(g',h') = h'$. From Definition 2.7 and Observation 2.6, we can thus conclude that

$$\max_{\mathrm{u}}^{\mathrm{g}}(g,h) = *\,\mathrm{res}(g) = g\,,$$
$$\min_{\mathrm{u}}^{\mathrm{g}}(g,h) = *\,\mathrm{res}(h) = h\,.$$

Next, if $g \prec h$, analogous reasoning shows that

$$\max_{\mathrm{u}}^{\mathrm{g}}(g,h) = h \neq g\,,$$
$$\min_{\mathrm{u}}^{\mathrm{g}}(g,h) = g \neq h\,.$$

The remaining case is that $g = h$. Assume g and h do not contain an $\mathfrak{u}$ bit, then we have $\max^{\mathrm{g}}_{\mathfrak{u}}(g, h) = \max^{\mathrm{g}}\{g, h\} = g = h$. Now assume $g = h = \gamma^{\mathrm{g}}_n(x) * \gamma^{\mathrm{g}}_n(x+1)$, for $x \in [2^n - 1]$. We get that $\max^{\mathrm{g}}_{\mathfrak{u}}(g, h) = *\{\gamma^{\mathrm{g}}_n(x), \gamma^{\mathrm{g}}_n(x+1)\} = h$. Likewise, $\min^{\mathrm{g}}_{\mathfrak{u}}(g, h) = g$. □

In other words, $\max^{\mathrm{g}}_{\mathfrak{u}}$ and $\min^{\mathrm{g}}_{\mathfrak{u}}$ are the max and min operators w.r.t. the total order on valid strings shown in Table 5.1.

Corollary 5.6. *Let $g, h \in \mathcal{S}^{\mathrm{g}}_n$. If and only if $g \succeq h$, then*

$$(\texttt{COMP}_n(g, h))_{\mathfrak{u}} = (g, h)\,.$$

Example. Let $n = 4$ then the $\max^{\mathrm{g}}_{\mathfrak{u}}$ function computes

$$\max^{\mathrm{g}}_{\mathfrak{u}}\{1001, 1000\} = \max^{\mathrm{g}}_{\mathfrak{u}}\{\gamma^{\mathrm{g}}_4(14), \gamma^{\mathrm{g}}_4(15)\} = \gamma^{\mathrm{g}}_4(15) = 1000\,,$$
$$\max^{\mathrm{g}}_{\mathfrak{u}}\{0\mathfrak{u}10, 0010\} = \max^{\mathrm{g}}_{\mathfrak{u}}\{*\,\gamma^{\mathrm{g}}_4(\{3,4\}), \gamma^{\mathrm{g}}_4(3)\} = \gamma^{\mathrm{g}}_4(3) * \gamma^{\mathrm{g}}_4(4) = 0\mathfrak{u}10\,, \text{ and}$$
$$\max^{\mathrm{g}}_{\mathfrak{u}}\{0\mathfrak{u}10, 0110\} = \max^{\mathrm{g}}_{\mathfrak{u}}\{*\,\gamma^{\mathrm{g}}_4(\{3,4\}), \gamma^{\mathrm{g}}_4(4)\} = \gamma^{\mathrm{g}}_4(4) = 0110\,.$$

5.4 Sorting Transducer

We define a finite state machine performing the comparison of two BRGC codes. In each step, the transducer is fed the pair of ith input bits. That is, when processing inputs $g, h \in \Gamma^{\mathrm{g}}_n$, in step i it receives input pair $g_i h_i$. The transducer performs a four-valued comparison. When processing the inputs it determines whether g and h are equal with even parity, with odd parity, or whether the encoded value of g is larger than h, or smaller. A depiction of the transducer is given in Figure 5.1.

Definition 5.7 (Sorting Transducer)**.** *The transducer comparing two BRGC codes step by step is given by state space $S = \{0, 1, 2, 3\}$, starting state $s_0 = 0$, $\Sigma = \mathbb{B}^2$, and $\Lambda = \mathbb{B}^2$. The transition function $t\colon S \times \Sigma \to S$ and the output function $o\colon S \times \Sigma \to \Lambda$ are given in Table 5.2.*

The transducer describes a transcription function τ (cf. Definition 4.2). To show the correctness of the transducer we show that $\tau(g, h)$ computes $\texttt{COMP}(g, h)$.

Example. A run of the transducer of Definition 5.7 on inputs $g = 1011$ and $h = 1001$ is given in Table 5.3.

5.4.1 Correctness of the Transducer

In order to show the correctness of the transducer for BRGC strings we first show that the transition function is correct, i.e., that the transducer transitions to the state according to the natural order on BRGC strings. Subsequently, we show the correctness of the output function, i.e., that in each state the transducer outputs the correct pair of bits. In combination, these statements yield the correctness of the transducer

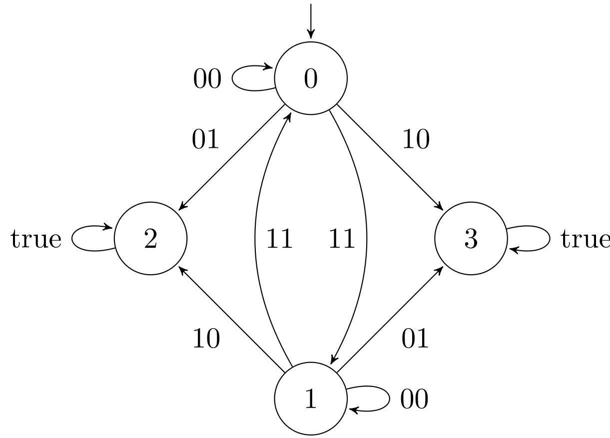

Figure 5.1: Visualization of the transducer that, at step i, processes the pair of ith input bits.

$t(s,\sigma)$	00	01	11	10
0	0	2	1	3
1	1	3	0	2
2	2	2	2	2
3	3	3	3	3

$o(s,\sigma)$	00	01	11	10
0	00	10	11	10
1	00	01	11	01
2	00	10	11	01
3	00	01	11	10

Table 5.2: The transition function $t(s,\sigma)$ (left) and output function $o(s,\sigma)$ (right).

i	0	1	2	3	4
$g_i h_i$		11	00	10	11
$s^{(i)} = t(s^{(i-1)}, g_i h_i)$	0	1	1	2	2
$o(s^{(i-1)}, g_i h_i)_1$		1	0	0	1
$o(s^{(i-1)}, g_i h_i)_2$		1	0	1	1

Table 5.3: Example run of the sorting transducer on inputs $g = 1011$ and $h = 1001$.

state	encoding	meaning
0	00	$g_{0,i-1} = h_{0,i-1}$ and $\mathrm{par}(g_{0,i-1}) = 0$
1	11	$g_{0,i-1} = h_{0,i-1}$ and $\mathrm{par}(g_{0,i-1}) = 1$
2	01	$g \prec h$
3	10	$g \succ h$

Table 5.4: Binary encoding and intuitive meaning of the states of the transducer from Definition 5.7.

Correctness of the Transition Function. The meaning of the states is indicated in Section 5.4.1. In BRGC encoding the parity indicates whether the remaining bits are to be compared w.r.t. the standard or reflected order. If the processed parts of the inputs are equal, then the transducer also keeps a record of their parity. Such that the state machine performs the comparison correctly w.r.t. the meaning of the states.

Lemma 5.8. *Let $g, h \in \Gamma_n^{\mathrm{g}}$ and $i \in [n+1]$, then the following equivalences hold*

$$s^{(i)} = 0 \vee s^{(i)} = 1 \Leftrightarrow g_{1,i} = h_{1,i}$$
$$s^{(i)} = 0 \Leftrightarrow g \prec h, \textit{ if and only if } g_{i+1,n} \prec h_{i+1,n}$$
$$s^{(i)} = 1 \Leftrightarrow g \prec h, \textit{ if and only if } g_{i+1,n} \succ h_{i+1,n}$$
$$s^{(i)} = 2 \Leftrightarrow g \prec h$$
$$s^{(i)} = 3 \Leftrightarrow g \succ h$$

Proof. We show the claim by induction on i. For the induction basis, $i = 0$, we need to show the claim for $s^{(i)} = s^{(0)} = 0$. The first equivalence holds as $g_{1,0} = h_{1,0}$ is the empty string. Also the second equivalence holds as $g_{i+1,n} = g$ and $h_{i+1,n} = h$. For the induction step from $i - 1 \in [n]$ to i, we make a case distinction on $s^{(i-1)}$.

Case $s^{(i-1)} = 0$: By the induction hypothesis, $g_{1,i-1} = h_{1,i-1}$ and $g \prec h$ if and only if $g_{i,n} \prec h_{i,n}$. Thus, if $g_i h_i = 00$, $s^{(i)} = 0$, $g_{1,i} = h_{1,i}$, and by the recursive definition of the code, $g_{i,n} \prec h_{i,n} \Leftrightarrow g_{i+1,n} \prec h_{i+1,n}$. Similarly, if $g_i h_i = 1$, also $g_{1,i} = h_{1,i}$, but the code for the remaining bits is "reflected", i.e., $g \prec h \Leftrightarrow g_{i+1,n} \succ h_{i+1,n}$. If $g_i h_i = 2$, the definition implies that $g \prec h$ regardless of further bits, and if $g_i h_i = 3$, $g \succ h$ regardless of further bits.

Case $s^{(i-1)} = 1$: Analogously to the previous case, noting that reflecting a second time results in the original order.

Case $s^{(i-1)} = 2$: By the induction hypothesis, $g \prec h$. For any $\sigma \in \Sigma$, we obtain from the definition of the transition function that $s^{(i)} = t(2, \sigma) = 2$.

Case $s^{(i-1)} = 3$: By the induction hypothesis, $g \succ h$. For any $\sigma \in \Sigma$, we obtain from the definition of the transition function that $s^{(i)} = t(3, \sigma) = 3$. □

Correctness of the Output Function. It is left to show that the output function is correct, i.e., that upon reception of an input pair, the transducer returns the correct output (depending on its state).

Lemma 5.9. *Let $g, h \in \Gamma_n^{\mathrm{g}}$, then for all $i \in [n+1]$*

$$\max{}^{\mathrm{g}}(g,h)_i = o(s^{(i-1)}, g_i h_i)_1$$
$$\min{}^{\mathrm{g}}(g,h)_i = o(s^{(i-1)}, g_i h_i)_2$$

Proof. We distinguish two cases. The proof is simple in case $g_i = h_i$, i.e. $g_i h_i = 00$ and $g_i h_i = 11$. We know that $\max^{\mathrm{g}}(g,h)_i = \min^{\mathrm{g}}(g,h)_i$. By Table 5.2 we can verify for each $s^{(i-1)} \in S$ that $o(s^{(i-1)}, g_i h_i)_1 = g_i = h_i = \max^{\mathrm{g}}(g,h)_i$ and $o(s^{(i-1)}, g_i h_i)_2 = g_i = h_i = \min^{\mathrm{g}}(g,h)_i$. In case $g_i \neq h_i$ we verify the claim by going over all possible states $s^{(i-1)} \in S$.

Case $s^{(i-1)} = 0$: We know by Lemma 5.8 that $\max^{\mathrm{g}}(g,h)_i = \max\{g_i, h_i\}$ and $\min^{\mathrm{g}}(g,h)_i = \min\{g_i, h_i\}$. Table 5.2 verifies that $o(s^{(i-1)}, g_i h_i)_1 = \max\{g_i, h_i\}$ and $o(s^{(i-1)}, g_i h_i)_2 = \min\{g_i, h_i\}$.

Case $s^{(i-1)} = 1$: We know by Lemma 5.8 that $\max^{\mathrm{g}}(g,h)_i = \min\{g_i, h_i\}$ and $\min^{\mathrm{g}}(g,h)_i = \max\{g_i, h_i\}$. Table 5.2 verifies that $o(s^{(i-1)}, g_i h_i)_1 = \min\{g_i, h_i\}$ and $o(s^{(i-1)}, g_i h_i)_2 = \max\{g_i, h_i\}$.

Case $s^{(i-1)} = 2$: We know by Lemma 5.8 that $\max^{\mathrm{g}}(g,h)_i = g_i$ and $\min^{\mathrm{g}}(g,h)_i = h_i$. Table 5.2 verifies that $o(s^{(i-1)}, g_i h_i)_1 = g_i$ and $o(s^{(i-1)}, g_i h_i)_2 = h_i$.

Case $s^{(i-1)} = 3$: We know by Lemma 5.8 that $\max^{\mathrm{g}}(g,h)_i = h_i$ and $\min^{\mathrm{g}}(g,h)_i = g_i$. Table 5.2 verifies that $o(s^{(i-1)}, g_i h_i)_1 = h_i$ and $o(s^{(i-1)}, g_i h_i)_2 = g_i$. □

Together Lemma 5.8 and Lemma 5.9 imply that the transducer sorts two inputs according to the BRGC encoding. And $\tau(g,h)$ computes $\texttt{COMP}(g,h)$.

Corollary 5.10. *Let $g, h \in \Gamma_n^{\mathrm{g}}$ and define $x \in \Sigma^n$ such that $(x)_i = g_i h_i$, where $i \in [1, n]$. For all i we have that*

$$\max{}^{\mathrm{g}}(g,h)_i = (\tau(x)_i)_1$$
$$\min{}^{\mathrm{g}}(g,h)_i = (\tau(x)_i)_2$$

5.5 Hazard-free Implementation

We now move on to implement a circuit that sorts two valid strings according to the order ($\succ$). In Corollary 5.6 we show a hazard-free COMP circuit suffices to sort valid strings. The previous section shows a finite state transducer that computes COMP. Hence, we can readily apply Theorem 4.20 from Chapter 4 to obtain the desired hazard-free circuit from the transducer.

5.5.1 Small Hazard-Free Transducer.

We apply the construction from Chapter 4 to the transducer defined in Definition 5.7 as a black box. Valid strings have one unstable bit at most, we get two valid strings as input. Hence, we can choose $k = 2$, such that the framework constructs a 2-bit hazard-free circuit. By Theorem 4.20 we obtain a hazard-free COMP circuit.

Corollary 5.11. *Let $g, h \in \mathcal{S}_n^{\mathrm{g}}$, then there is a circuit that computes* $\mathtt{COMP}_{\mathtt{u}}(g, h)$ *that has asymptotically optimal*

$$\text{size } \mathcal{O}(n)$$
$$\text{and depth } \mathcal{O}(\log n) \ .$$

Unfortunately, Theorem 4.20 yields large constants for an implementation of the transducer with four states and an in-/output alphabet using two bits. By observing the structure of valid strings we can improve and construct a smaller circuit.

5.5.2 Improvements.

The framework allows both u's in one input word.

The framework of Chapter 4 does not take into account that the position of an unstable bit in a valid string is restricted. It can only occur in certain positions. Moreover, it does not take into account that there is only one unstable bit per valid string. Fortunately, we can make use of the restrictions on the inputs. If we do not use the framework as a black box, but adjust it to the restrictions, we can show that there is a better implementation in terms of size and depth. In the following, we describe the adjustments and the resulting improvements. The implementation described in the following has the same asymptotic behavior, but the constants are smaller.

The key step to the improvements is to observe that the transition function is associative. Instead of function composition (of the transition functions), we can use the transition function directly as the PPC operator. The correctness proof then uses the fact that the position of unstable bits is restricted.

Three Hurdles. We want to replace the universal encoding and matrix multiplication. We can do so by specifying an encoding of the states and designing a hazard-free circuit of the transition function. But, even with this modification, it is not obvious that our approach yields correct outputs. There are three hurdles to overcome in the hazard-free extension. For $i \in \{1, \dots, n\}$ and $g, h \in \mathcal{S}_n^{\mathrm{g}}$ we need to show that

$$t_{\mathtt{u}}(s, \sigma) \text{ is associative} \tag{P1}$$

$$t_{\mathtt{u}}(\cdot, g_1h_1) \circ \ldots \circ t_{\mathtt{u}}(\cdot, g_ih_i)(s^{(0)}) = s_{\mathtt{u}}^{(i)} \tag{P2}$$

$$o_{\mathtt{u}}(s_{\mathtt{u}}^{(i-1)}, g_ih_i) = (\max{}_{\mathtt{u}}^{\mathrm{g}}(g, h)_i, \min{}_{\mathtt{u}}^{\mathrm{g}}(g, h)_i) \tag{P3}$$

The encoding of the states is given in Section 5.4.1. In the following pages, we prove propositions (P1) to (P3). In combination, these propositions then show that the improvement correctly implements $\mathtt{COMP}(g, h)$.

Associativity of the Transition Function. Regarding (P1), we note the statement that $t_{\mathfrak{u}}(s,\sigma)$ is associative does not depend on n. In other words, it can be verified by checking for all possible $x,y,z \in \mathbb{T}^2$ whether $t_{\mathfrak{u}}(t_{\mathfrak{u}}(x,y),z) = t_{\mathfrak{u}}(x,t_{\mathfrak{u}}(y,z))$. While it is tractable to manually verify all $3^6 = 729$ cases (exploiting various symmetries and other properties of the operator), it is tedious and prone to errors. Instead, we verified that both evaluation orders result in the same outcome by a short computer program.

Theorem 5.12. (P1) *holds, i.e.,* $t_{\mathfrak{u}}(s,\sigma)$ *is associative.*

Apart from being essential for our construction, this theorem simplifies notation; in the following, we may write

$$(\pi_i)_{\mathfrak{u}} \coloneqq t_{\mathfrak{u}}(\cdot, g_1h_1) \circ t_{\mathfrak{u}}(\cdot, g_2h_2) \circ \ldots \circ t_{\mathfrak{u}}(\cdot, g_ih_i)\,,$$

where the order of evaluation does not affect the result. Because $t_{\mathfrak{u}}(s^{(0)},\sigma) = \sigma$,

$$(\pi_i)_{\mathfrak{u}} = (\pi_i)_{\mathfrak{u}}(s^{(0)})\,.$$

Remark. We stress that in general the hazard-free extension of an associative operator is not associative. A counter-example is given by binary addition modulo 4:

$$(0\mathfrak{u} +_{\mathfrak{u}} 01) +_{\mathfrak{u}} 01 = \mathfrak{u}\mathfrak{u} \neq 1\mathfrak{u} = 0\mathfrak{u} +_{\mathfrak{u}} (01 +_{\mathfrak{u}} 01).$$

Determining State i. For hurdle (P2) we need to show that repeated application of the transition function to the input pairs g_jh_j, $j \in [1,i]$, actually results in $s_{\mathfrak{u}}^{(i)}$. We make use of Observation 3.26, i.e., if there is an $\mathfrak{u}$ bit at position m of a valid string, then the remaining $n-m$ bits of a valid string are the maximum codeword of a $(n-m)$-bit code.

Our reasoning will be based on distinguishing two main cases: one is that $s_{\mathfrak{u}}^{(i)}$ contains at most one $\mathfrak{u}$ bit, the other that $s_{\mathfrak{u}}^{(i)} = \mathfrak{u}\mathfrak{u}$. For each case, we formulate a technical statement.

Observation 5.13. *If $s_{\mathfrak{u}}^{(i)}$ contains at most one $\mathfrak{u}$ bit, then resolution of the ith state equates to repeated application of the transition function to the resolution of the inputs. For $i \in [n+1]$,*

$$\left|\operatorname{res}\left(s_{\mathfrak{u}}^{(i)}\right)\right| \leq 2 \Rightarrow \operatorname{res}(s_{\mathfrak{u}}^{(i)}) = t(\cdot, \operatorname{res}(g_1h_1)) \circ \ldots \circ t(\cdot, \operatorname{res}(g_ih_i))(s^{(0)})\,.$$

Proof. Let $S \coloneqq t(\cdot, \operatorname{res}(g_1h_1)) \circ \ldots \circ t(\cdot, \operatorname{res}(g_ih_i))(s^{(0)})$. From the hazard-free extension we obtain $s_{\mathfrak{u}}^{(i)} = *\,S$. Plugging this in, the l.h.s. becomes $\operatorname{res}\left(s_{\mathfrak{u}}^{(i)}\right) = \operatorname{res}(*\,S)$. The claim thus follows from Observation 2.5. □

Lemma 5.14. *Suppose that for valid strings $g,h \in \mathcal{S}_n^{\mathrm{g}}$, it holds that $s_{\mathfrak{u}}^{(i)} = \mathfrak{u}\mathfrak{u}$, for some $i \in [1,n]$. Then $g = h$ and $s_{\mathfrak{u}}^{(j)} = \mathfrak{u}\mathfrak{u}$, for all $j \in [i,n]$.*

Proof. As in the previous proof let $S \coloneqq t(\cdot, \text{res}(g_1h_1)) \circ \ldots \circ t(\cdot, \text{res}(g_ih_i))(s^{(0)})$ such that $s_\mathfrak{u}^{(i)} = *\,S$. It must hold that (i) $\{00, 11\} \subseteq S$, or (ii) $\{01, 10\} \subseteq S$. By Lemma 5.8, (i) implies that there are stabilizations $g', g'' \in \text{res}(g_{1,i})$ and $h', h'' \in \text{res}(h_{1,i})$ such that $g' = h'$, $\text{par}(g') = 0$, $g'' = h''$, and $\text{par}(g'') = 1$, while (ii) implies such g', g'', h', h'' with $g' \prec h'$ and $g'' \succ h''$. Checking Definition 5.2 and Table 5.1, we see that both options necessitate that $g_{1,i} = h_{1,i}$ with some $\mathfrak{u}$ bit. Denoting by $m \in [1, i-1]$ the index such that $g_m = h_m = \mathfrak{u}$. Observation 3.26 shows that $g_{m+1,n} = h_{m+1,n} = 10^{n-m-1}$. In particular, $g = h$, showing (again by Lemma 5.8) that (i) or (ii) (in fact both) also apply to $t(\cdot, \text{res}(g_1h_1)) \circ \ldots \circ t(\cdot, \text{res}(g_jh_j))(s^{(0)})$ for any $j \in [i, n]$. We conclude that $s_\mathfrak{u}^{(j)} = \mathfrak{uu}$ for any such j. ☐

Equipped with these tools, we are ready to prove (P2). Note that we can abbreviate $t_\mathfrak{u}(\cdot, g_1h_1) \circ \ldots \circ t_\mathfrak{u}(\cdot, g_ih_i)$ by $(\pi_i)_\mathfrak{u}$ because of Theorem 5.12.

Theorem 5.15. (P2) *holds, i.e., for all* $g, h \in \mathcal{S}_n^{\mathrm{g}}$ *and* $i \in [1, n]$, $s_\mathfrak{u}^{(i)} = (\pi_i)_\mathfrak{u}$.

Proof. We show the claim by induction on i. Trivially, we have that $s_\mathfrak{u}^{(0)} = s^{(0)} = 00$ and thus for $i = 1$ that

$$s_\mathfrak{u}^{(1)} = *(t(s^{(0)}, \text{res}(g_1h_1))) = *(\text{res}(g_1h_1)) = g_1h_1 = t_\mathfrak{u}(s^{(0)}, \text{res}(g_1h_1)) \,.$$

Hence, suppose that the claim has been established for $i - 1 \in [1, n-1]$ and consider index i. If $\left|\text{res}\left(s_\mathfrak{u}^{(i-1)}\right)\right| \leq 2$, then Observation 5.13 and the induction hypothesis yield that

$$\begin{aligned} t_\mathfrak{u}(\cdot, g_1h_1) \circ \ldots \circ t_\mathfrak{u}(\cdot, g_ih_i)(s^{(0)}) &= t_\mathfrak{u}(s_\mathfrak{u}^{(i-1)}, g_ih_i) \\ &= *\left(t(\text{res}\left(s_\mathfrak{u}^{(i-1)}\right), \text{res}(g_ih_i))\right) \\ &= *\,t(\cdot, \text{res}(g_1h_1)) \circ \ldots \circ t(\cdot, \text{res}(g_ih_i)) \\ &= s_\mathfrak{u}^{(i)} \,. \end{aligned}$$

It remains to consider the case that $s_\mathfrak{u}^{(i-1)} = \mathfrak{uu}$. By Lemma 5.14, $s_\mathfrak{u}^{(i)} = \mathfrak{uu}$, too. Thus,

$$(\pi_i)_\mathfrak{u} = t_\mathfrak{u}(s_\mathfrak{u}^{(i-1)}, g_ih_i) = t_\mathfrak{u}(\mathfrak{uu}, g_ih_i) = \mathfrak{uu} = s_\mathfrak{u}^{(i)} \,.$$ ☐

Obtaining the Outputs from State i. Recall that $o\colon \mathbb{B}^2 \times \mathbb{B}^2 \to \mathbb{B}^2$ is the operator given in Table 5.2 computing $\max^{\mathrm{g}}\{g, h\}_i \min^{\mathrm{g}}\{g, h\}_i$ out of $s^{(i-1)}$ and g_ih_i. We derive (P3) by again distinguishing cases where $s^{(i-1)}$ has at most one unstable bit and $s^{(i-1)} = \mathfrak{uu}$.

Theorem 5.16. (P3) *holds, i.e., given valid inputs* $g, h \in \mathcal{S}_n^{\mathrm{g}}$ *and* $i \in [1, n]$,

$$o_\mathfrak{u}(s_\mathfrak{u}^{(i-1)}, g_ih_i) = \max{}_\mathfrak{u}^{\mathrm{g}}(g, h)_i \min{}_\mathfrak{u}^{\mathrm{g}}(g, h)_i \,.$$

i	0	1	2	3	4
g_ih_i		00	u0	11	00
$s_\mathfrak{u}^{(i)} = t_\mathfrak{u}(s_\mathfrak{u}^{(i-1)}, g_ih_i)$	00	00	u0	1u	1u
$o_\mathfrak{u}(s_\mathfrak{u}^{(4)}, g_ih_i)$		00	uu	11	00
$o_\mathfrak{u}(s_\mathfrak{u}^{(i-1)}, g_ih_i)$		00	u0	11	00

Table 5.5: Run of the FSM on inputs $g = 0\mathfrak{u}10$ and $h = 0010$, showing that computation of only the last state is insufficient. The computation yields $o_\mathfrak{u}(1\mathfrak{u}, \mathfrak{u}0) = *\{00, 01, 10\} = \mathfrak{uu}$, where $o_\mathfrak{u}(00, \mathfrak{u}0) = *\{00, 10\} = \mathfrak{u}0$ is the correct output at the second position.

Proof. Assume first that $\left|\text{res}\left(s_\mathfrak{u}^{(i-1)}\right)\right| \leq 2$, then

$$\begin{aligned}
&o_\mathfrak{u}(s_\mathfrak{u}^{(i-1)}(g,h), g_ih_i)\\
&= *\,o\left(\text{res}\left(s_\mathfrak{u}^{(i-1)}(g,h)\right), \text{res}(g_ih_i)\right)\\
&= *\,o\left(t(\cdot, \text{res}(g_1h_1)) \circ \ldots \circ t(\cdot, \text{res}(g_{i-1}h_{i-1})), \text{res}(g_ih_i)\right) && \textit{by Obs. 5.13}\\
&= *\left(\max{}^\text{g}\{\text{res}(g), \text{res}(h)\}_i \min{}^\text{g}\{\text{res}(g), \text{res}(h)\}_i\right) && \textit{by Cor. 5.10}\\
&= \max{}^\text{g}_\mathfrak{u}(g,h)_i \min{}^\text{g}_\mathfrak{u}(g,h)_i\,.
\end{aligned}$$

Otherwise, $s_\mathfrak{u}^{(i-1)} = \mathfrak{uu}$. Then, by Lemma 5.14, $g = h$. In particular, $g_i = h_i$. We observe that for all $b \in \mathbb{T}$, it holds that $o_\mathfrak{u}(\mathfrak{uu}, bb) = bb$. Therefore,

$$o_\mathfrak{u}(s_\mathfrak{u}^{(i-1)}(g,h), g_ih_i) = g_ih_i = \max{}^\text{g}_\mathfrak{u}(g,h)_i \min{}^\text{g}_\mathfrak{u}\{g,h\}_i\,. \qquad \square$$

Remark. The reader may ask why we compute $s_\mathfrak{u}^{(i)}$ for all $i \in [0, n-1]$ instead of computing only $s_\mathfrak{u}^{(n)}$ with a simple tree of $t_\mathfrak{u}$ elements, which would yield a smaller circuit. State $s_\mathfrak{u}^{(n)}$ determines the result of the comparison of the entire strings. It could be used to compute all outputs, i.e., we could compute the output by $o_\mathfrak{u}(s_\mathfrak{u}^{(n)}, g_ih_i)$ instead of $o_\mathfrak{u}(s_\mathfrak{u}^{(i-1)}, g_ih_i)$. However, in case of instability, this may lead to incorrect results. We show an example run of the transducer and the outputs for $o_\mathfrak{u}(s_\mathfrak{u}^{(n)}, g_ih_i)$ and $o_\mathfrak{u}(s_\mathfrak{u}^{(i-1)}, g_ih_i)$ in Table 5.5. The example shows that, in this case, a single $\mathfrak{u}$ in the input can affect both outputs. We thus need to compute every intermediate state $s_\mathfrak{u}^{(i)}$.

Putting it Together. We can combine propositions (P1) to (P3) to show that the improvement correctly implements `COMP`(g,h). By Theorem 5.12 and Definition 4.3 we can compute $(\pi_i)_\mathfrak{u} = t_\mathfrak{u}(\cdot, g_1h_1) \circ \ldots \circ t_\mathfrak{u}(\cdot, g_ih_i)$ by a PPC circuit for each $i \in \{1, \ldots, n\}$. Given a hazard-free implementation of $t_\mathfrak{u}(s, gh)$, Theorem 5.15 states that we can

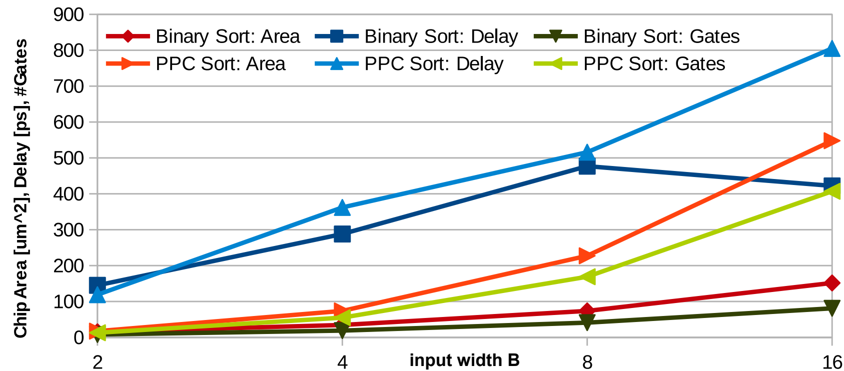

Figure 5.2: Simulation results of [16]. Comparison of `COMP` (here "PPC Sort") and a binary benchmark in terms of chip area, delay, and number of gates for input widths 2, 4, 8, and 16.

compute $s_{\mathrm{u}}^{(i)}$ from $(\pi_i)_{\mathrm{u}}(s^{(0)})$. Finally, Theorem 5.16 shows that any hazard-free implementation of $o_{\mathrm{u}}(s, gh)$ can be used to compute the output of $\mathtt{COMP}(g, h)$.

Hazard-free implementations of $t_{\mathrm{u}}(s, gh)$ and $o_{\mathrm{u}}(s, gh)$ do not offer significant novelty. In fact, they can be implemented by the same circuit, which is similar to the `CMUX`. Gate-level implementations are deferred to [16].

5.6 Simulation and Results

In [16] we perform simulations to compare `COMP` to a binary benchmark. Here we shortly highlight the results. The binary benchmark is a simple circuit sorting two inputs in the standard binary encoding. The circuit is implemented in VHDL and synthesized by computer aided design (CAD) tools.

Both designs are layed out for input widths of 2, 4, 8, and 16 bits and plugged in sorting networks for 2, 3, 7, and 10 inputs. Results are depicted in Figure 5.2. Even though the comparison disfavors `COMP` the simulations show that it performs on par with the benchmark. The comparison favors the benchmark in the sense that the benchmark undergoes standard optimization while `COMP` is not optimized because we have to ensure hazard-freedom. The optimization deploys faster gates for input width 16, which reduces the delay of the benchmark. `COMP` could be further optimized by the use of the transistor-level implementation of the `CMUX` in [42].

The results show that (application-specific) hazard-free circuits are able to perform on par with conventional circuits. Furthermore, there is room to improve the results by incorporating hazard-free solutions in the design flow. For example, hazard-free optimization and integration of the `CMUX` into standard cell libraries are expected to yield better hazard-free transistor-level implementations.

Clock Synchronization 6

In the following pages, we introduce the basic terminology and definitions for the problem of clock synchronization. We present two algorithms for clock synchronization: the Lynch-Welch algorithm and the OffsetGCS algorithm. We defer a discussion of related work on clock synchronization algorithms to Section 7.1.3 in the following chapter.

Outline. We present the computational model in Section 6.1 followed by the problem specification in Section 6.2. The Lynch-Welch algorithm is discussed in Section 6.3 and the OffsetGCS algorithm in Section 6.4.

6.1 Model

Network, Communication, and Timing. A network is modeled by a graph $G = (V, E)$, where V is the set of nodes and E is the set of edges. An edge is a pair of nodes v, w in V. Edges in a network are bidirectional, for each edge (v, w) E also contains (w, v). Furthermore, for each $v \in V$, set E contains edge (v, v). The diameter D of a network is the maximum length of a shortest path in the network.

We denote real time, i.e., an external reference time for analysis of the algorithms, by *Newtonian* time. A node has no access to Newtonian time, each node has its own (internal) time reference.

Nodes communicate by sending content-less messages, known as *pulses*. A pulse is sent via broadcast to all neighboring nodes. The message delay is the time a pulse travels between sender and receiver. It is constrained by a maximum delay d and a minimum delay $d - U$, where U is the delay uncertainty. A pulse sent by a node at Newtonian time t is received between time $t + d - U$ and time $t + d$. The pulse arrives at Newtonian time $t' \in [t + d - U, t + d]$.

Computations are event-driven, an incoming message or a timing threshold triggers the next step. The model does not account for computational delay. Any computations are instantaneous, i.e., calculations take no time. An incoming message triggers immediate transmission of the next message. We can account for computational delays in the message delay.

Faults. Below we present an algorithm that is robust against faulty nodes. Formally, we denote the set of faulty nodes by $F \subseteq V$. All nodes $v \in F$ are Byzantine, i.e., we make no assumptions whatsoever about their behavior. In particular, they are not required to communicate by broadcast, pulses can be sent to different neighbors at different times. Usually, we assume that the number of faulty nodes in the network is bounded by a parameter f, such that $|F| \leq f$

Hardware Clock. Each node has an associated time reference, given by its *hardware clock*. The hardware clock is prone to uncertainty, which we model by a variable rate that may change over time. The uncertainty is called the *hardware clock drift* (short *clock drift*). For each $v \in V$ there is an integrable function $h_v \colon \mathbb{R}_{\geq 0} \to \mathbb{R}$ called the hardware clock rate. Parameter $\rho > 0$ is an upper bound on the one-sided hardware clock drift of all nodes. The hardware clock rate satisfies $1 \leq h_v(t) \leq 1 + \rho$ for all $t \in \mathbb{R}_{\geq 0}$. Formally, $H_v(t)$, the hardware clock value of v at time t, is defined by

$$H_v(t) = \int_{t_v^0}^{t} h_v(\tau) d\tau + H_v^0 ,$$

where H_v^0 is the, possibly negative, initial value of v's hardware clock and t_v^0 is the Newtonian time at which v initializes. The hardware clock is v's internal reference of time. A node can only access its own hardware clock value, the rate of the hardware clock remains unknown to a node. The main purpose of a hardware clock is to estimate the time difference between local events.

6.2 Problem

Even when perfectly initialized, neighboring nodes may drift apart at rate ρ. Without further tools, the system cannot cope with the drift of the hardware clock. The longer the system runs, the longer nodes would have to wait for incoming messages. The response time of an event-based system would increase with its running time.

different approaches that go beyond adjusting the frequency of the logical clock are discussed in Section 8.1

In order to cope with this issue we introduce an artificial clock, the *logical clock*. It is artificial in the sense that it is computed by the node itself. In general, the logical clock follows the rate of the hardware clock with slight adjustments. The difference between two logical clock values is called skew. The problem of clock synchronization lies in computing synchronous logical clocks with small skew, despite uncertainties in the hardware clock and message delays.

Logical Clock. The logical clock value is given by $L_v \colon \mathbb{R}_{\geq 0} \to \mathbb{R}$. A node's logical clock follows the rate of its hardware clock, but it is adjustable by a constant factor. It may speed up in order to catch up to other nodes. The logical clock $L_v(t)$ is initialized to 0, when $H_v(t)$ reaches 0.

The formal specification of the logical clock slightly varies between both algorithms that we present. According definitions are given in the specific sections. Intuitively, the GCS algorithm (cf. Section 6.4) controls a binary switch that puts the logical clock in either slow or fast mode; whereas in slow mode the logical clock follows the hardware clock and in fast mode, it advances at the hardware clock rate adjusted by a constant factor larger than 1. The Lynch-Welch algorithm on the other hand is round-based. In each round it computes an adjustment, which is applied to the logical clock.

We require that the logical clocks progress at least at the normalized rate 1, we do not allow to halt a clock. In other words, each node advances at all times. This guarantees progress of the system at all times.

Remark. Without the minimum rate requirement, the task becomes trivial: all nodes can simply set $L_v(t) = 0$ for all times t to achieve perfect synchronization.

Skew. The skew between two nodes describes the difference in their logical clock value. The upper bound on the skew is a figure of merit for clock synchronization algorithms. For a graph, we define two types of skew, the *global skew* and the *local skew*.

Global Skew The global skew $\mathcal{G}(t)$ is the maximum skew between any two nodes in the network. Formally, it is defined by

$$\mathcal{G}(t) \coloneqq \max_{v,w \in V} \{L_w(t) - L_v(t)\} \ .$$

Local Skew The local skew $\mathcal{L}(t)$ is the maximum skew between any two neighboring nodes in the network. Formally, it is defined by

$$\mathcal{L}(t) \coloneqq \max_{(v,w) \in E} \{L_w(t) - L_v(t)\} \ .$$

Clock Synchronization. The goal of a clock synchronization algorithm is to provide a distributed algorithm that keeps clocks synchronized. A clock synchronization algorithm should be able to tolerate clock drift and varying message delays. When executed on each node, the algorithm computes a logical clock minimizing $\mathcal{L}(t)$ (or $\mathcal{G}(t)$) at any time t.

Initialization. The analysis of the algorithms requires that at time 0 neighboring nodes have a small initial skew. The initial local skew is bounded by Δ,

in Section 8.2 we further discuss initial requirements on the clocks

$$\mathcal{L}(0) = \max_{(v,w) \in E} \{H_v(0) - H_w(0)\} \leq \Delta \,. \tag{6.1}$$

To achieve good initialization, we could, e.g., perform an initial flooding of the system [73]. For an initial flooding, we require a spanning tree with low depth. Assuming we are given a root node we can construct a Breadth-First Search (BFS) tree using the Bellman-Ford algorithm [6]. BFS trees are spanning trees with optimal depth, the longest path from the root to a leaf has length D, the diameter of the network. The algorithm can be executed asynchronously, it requires asymptotic running time in $\mathcal{O}(dD)$. We can afford the running time because the algorithm only needs to be executed once. After execution of the algorithm, each node v has a record of D and its distance to the root, i.e., its depth in the tree, which we denote by depth(v).

After constructing the optimal-depth spanning tree we can perform the flooding as follows: The root node r starts up, initializes $H_r^0 \coloneqq -(d-U)D$, and sends a

message to all its children. When receiving a message a node initializes and forwards a message to all its children. On initialization node v sets its initial value according to its depth in the tree; $H_v^0 := -(d-U) \cdot (D - \text{depth}(v))$. We define Newtonian time 0 as the time the last node in the system initializes. At time 0 the hardware clock of each node v is non-negative, i.e., $H_v(0) \geq 0$. We now show that the flooding achieves a local skew that is bounded by $(1+\rho)UD$.

Lemma 6.1. *The initial local skew is bounded by* $\Delta = (1+\rho)UD$*.*

Proof. Assume we are given a BFS tree, we perform a flooding as described above. We denote by t_f the Newtonian time the flooding starts. We show the claim by showing that neighboring nodes in the graph are located on the same or adjacent levels. Then we show skew bounds for nodes on the same or adjacent levels.

First, we show that neighboring nodes in the graph are located on the same or adjacent levels in the BFS tree. A level of the tree is the set of all nodes with the same depth. For contradiction assume nodes v and w are neighboring in the graph but $\text{depth}(w) > \text{depth}(v) + 1$. As v and w are neighbors there exists a path in the graph from w to the root r of length at most $\text{depth}(v) + 1$. As the tree is a BFS tree we obtain that $\text{depth}(w) \leq \text{depth}(v) + 1$. This contradicts the assumption that $\text{depth}(w) > \text{depth}(v) + 1$.

Next, we show the skew bound for nodes that are on the same level. Nodes on the same level initialize H_v^0 to the same value, and skew emerges from the different arrival times of the message. Each node v at depth $\text{depth}(v)$ receives its message earliest at time $t_f + (d-U) \cdot \text{depth}(v)$ and latest at time $t_f + d \cdot \text{depth}(v)$. Hence, two nodes at the same depth receive their message with a time difference of at most $U \cdot \text{depth}(v)$. As the hardware clock may drift, the skew of two nodes at the same depth is bounded by $(1+\rho)U \cdot \text{depth}(v)$. Bounding the depth by D results in a skew bound of $(1+\rho)UD$.

Last, we show the skew bound for nodes that are on adjacent levels. Node v at $\text{depth}(v)$ receives its message earliest at time $t_f + (d-U) \cdot \text{depth}(v)$. Node w at $\text{depth}(v) + 1$ receives its message latest at time $t_f + d(\text{depth}(v) + 1)$. Hence, the initializations have a time difference of at most $U \cdot \text{depth}(v) + d$. Node v initializes its hardware clock value to $-(d-U)(D-\text{depth}(v))$ and advances at least at rate 1. Hence, when w initializes, H_v has value at least $-(d-U)(D-\text{depth}(v)) + U \cdot \text{depth}(v) + d$. Node w initializes its hardware clock to $-(d-U)(D-(\text{depth}(v)+1))$. Hence, the difference in clock values of nodes in adjacent levels is at most $(\text{depth}(v)+1)U$. As $\text{depth}(v)$ is at most D, we obtain a bound of $(D+1)U$. Assuming that $\rho D > 1$, we obtain that the skew is bounded by the skew of nodes on the same level. □

For the gradient clock synchronization (GCS) algorithm the bound on the local skew achieved by a flooding does not match the requirement(cf. Theorem 6.8). However, in [14] we show that given an arbitrary global skew $\mathcal{G}(0)$ the system will converge to the skew bounds claimed in Theorem 6.8. We now show that the flooding technique can bound the initial global skew by $\mathcal{G}(0) \leq (\rho d + U)D$. Further evaluation is deferred to Section 6.4.1.

Lemma 6.2. *The initial global skew $\mathcal{G}(0)$ is bounded by $(\rho d + U)D$.*

Proof. Assume we are given a BFS tree, we perform a flooding as described above. We denote by t_f the Newtonian time the flooding starts. In the proof of Lemma 6.1 we derived skew bounds for nodes on the same or adjacent levels. Analogous argumentation shows that the skew of two nodes v and w is bounded by $U \cdot \text{depth}(w)$, where $\text{depth}(w) > \text{depth}(v)$ and v is not the root node. The skew of any two nodes in the graph is bounded by the skew of the node at a larger depth to the root node. At time t_f the root initializes to $-(d-U)D$. A node v receives its message latest at time $t_f + d \cdot \text{depth}(v)$. Hence, the root is initialized at most $\text{depth}(v) \cdot d$ time before v. It may drift at rate $1+\rho$. Hence, the value of the hardware clock of the root is at most $-(d-U)D + (\text{depth}(v) \cdot d)(1+\rho)$ when v initializes. Node v initializes its hardware clock to $-(d-U)(D - \text{depth}(v))$. Subtraction of the clock values results in $(\rho d + U)\text{depth}(v)$. The diameter D is an upper bound on $\text{depth}(v)$. Thus, the global skew after flooding is bounded by $(\rho d + U)D$. □

We do not further discuss initialization procedures. Although it is an important part, initialization is not the main focus of this work. It complicates the description of the algorithms and circuits. Hence, we assume the bounds on global and local skew given above.

6.3 Lynch-Welch Algorithm

In this section, we describe the algorithm of Lundelius-Welch and Lynch [71]. We present a variant of the algorithm similar to the one described by Khanchandani and Lenzen [57]. The algorithm is restricted to fully connected networks, also known as *cliques*. It is *fault-tolerant*, in the sense that it allows for up to $f \leq (n-1)/3$ nodes to be faulty.

Following the description in [57], the algorithm proceeds in rounds. During each round, each (correct) node pulses once at a specified logical time. Once a node records the relative times of its neighbors' pulses, it computes an adjustment to its logical clock using an "approximate agreement" step (cf. [35]). The clock adjustment is then made by adjusting the logical clock to an appropriate rate for the remainder of the round. The length of each round is inductively defined, where the initial round's length depends on (an upper bound on) the initial clock skew in the network.

Remark. Jennifer Lundelius Welch appears in the bibliography as Jennifer Lundelius or Jennifer L. Welch due to her marriage. For consistency with previous work, we stick with Lynch-Welch algorithm.

6.3.1 Algorithm

We assume that the network $G = (V, E)$ is fully connected, i.e., for each $v, w \in V$ the edge (v, w) is in set E. Nodes communicate by broadcast, a message that is sent by (non-faulty) node v is received by each node in the network (including v).

The algorithm controls its logical clock by adjusting the logical clock rate relative to the hardware clock. Specifically, the algorithm controls the parameter $\delta_v(t) \in \mathbb{R}_{\geq 0}$. The logical clock value $L_v(t)$ with offset L_v^0 at time 0 is computed to be

$$L_v(t) = \int_0^t (1 + \varphi \cdot \delta_v(\tau))h_v(\tau)\, d\tau + L_v^0 \,. \tag{6.2}$$

The parameter φ is a constant whose value will be determined later on, we now already fix that $0 < \varphi < 1$.

The algorithm is executed in rounds on a cluster of nodes. When the network is a clique, the cluster spans the whole network. In Chapter 7 clusters will be subgraphs of the network. Hence, we call our variant of the Lynch-Welch algorithm the ClusterSync algorithm. Each round $r \in \mathbb{N}$ consists of three phases, of logical durations $\tau_1(r)$, $\tau_2(r)$, and $\tau_3(r)$, respectively; the total round length is $T(r) = \tau_1(r) + \tau_2(r) + \tau_3(r)$. The phases play the following roles:

Phase 1. This phase is sufficiently long for all non-faulty nodes within a cluster to have transitioned to round r by the end of Phase 1. Each node sends a pulse at the end of Phase 1.

Phase 2. During this phase, each node waits to receive pulses from its neighbors. Phase 2 is sufficiently long that by the end of this phase, each node will have received pulses from all of its (correct) neighbors within its cluster. At the end of the phase, each node v computes an adjustment $\Delta_v(r)$ to its own logical clock.

Phase 3. During this final phase, v implements the clock adjustment computed at the end of Phase 2 by setting δ_v to an appropriate value for the duration of the phase.

During Phase 1 and Phase 2, each node sets $\delta_v = 1$. We give the pseudo-code in Algorithm 1. The algorithm uses the following notation. For each round r and nodes $v, w \in C$, at Newtonian time

- $t_v(r)$ node v begins round r,
- $p_v(r)$ node v sends its pulse in round r, and
- $t_{wv}(r)$ node v receives the pulse node w sent in (w's) round r.

In case node v or node w is faulty, the values above may not be well-defined. In such a case, we can assign their values arbitrarily. While the notation above is convenient for describing the algorithm, we emphasize that nodes cannot access the values $t_v(r)$, $p_v(r)$, and $t_{wv}(r)$ directly. Instead, v stores, for example, $L_v(t_{wv}(r))$, the logical time at which it receives w's round r pulse.

We state the ClusterSync algorithm executed at each node in Algorithm 1. The algorithm is parametrized by τ_1, τ_2, and τ_3.

In [57], L_v is discontinuously adjusted by $-\Delta_v(r)$ at the end of Phase 2, and increases at the nominal rate at all other times. The advantage of our formulation

Algorithm 1 ClusterSync at node v

```
1: L_v ← 0
2: for each round r ∈ ℕ do
3:     δ_v ← 1
4:     at time L_v(t_v(r)) + τ_1(r) do
5:         broadcast clock pulse
6:     at time L_v(t_v(r)) + τ_1(r) + τ_2(r) do
7:         S_v ← ∅                                   ▷ multiset, ordered ascendingly
8:         at each node w ∈ C_v do
9:             τ_wv ← L_v(t_wv(r)) − L_v(t_vv(r))
10:            S_v ← S_v ∪ {τ_wv}
11:        Δ_v(r) ← (S_v^{f+1} + S_v^{n−f})/2        ▷ S_v^i is the i-th element of S_v
12:        δ_v ← 1 − (1 + 1/φ) Δ_v(r)/(τ_3(r) + Δ_v(r))
13:    at time L_v(t_v(r)) + τ_1(r) + τ_2(r) + τ_3(r) do
14:        end round r
```

of ClusterSync is that $L_v(t)$ is continuous and increases at a rate that can be kept close to the nominal rate by choosing $\tau_3(r)$ sufficiently large. This becomes essential in Chapter 7.

6.3.2 Analysis

In [19], we employ the analysis from [57] with the adjustments necessary for our variant of the algorithm. We define the following parameters:

$$\begin{aligned}
\vartheta_g &= 1+\rho\,,\\
\xi_g &= (\vartheta_g-1)(1+\varphi)\,,\\
\alpha_g &= \frac{1}{1-\xi_g}\left(\frac{2\vartheta_g^2+5\vartheta_g-5}{2(\vartheta_g+1)}+\xi_g\cdot(1+1/\varphi)\right)\,,\\
\beta_g &= \frac{1}{1-\xi_g}(3\vartheta_g-1+\xi_g/\varphi)\cdot U+\frac{\xi_g}{1-\xi_g}d\,,\text{ and}\\
e_g^\infty &= \frac{\beta_g}{1-\alpha_g}\,.
\end{aligned} \tag{6.3}$$

Suppose $\alpha_g < 1$ (which is possible for sufficiently small ρ and suitable choice of φ). Then the analysis implies the upper bound in Lemma 6.3 on the skew, for suitable choices of τ_1, τ_2, and τ_3. In what follows, we define $\mathbf{p}(r)$ to be the multiset of (Newtonian) times at which the non-faulty nodes in the network generate their round

r pulses. That is,

$$\mathbf{p}(r) = \{p_v(r) \mid v \in C \setminus F\}.$$

We define the *pulse width* of round r, denoted $\|\mathbf{p}(r)\|$, to be the difference between maximum and minimum times in $\mathbf{p}(r)$:

$$\|\mathbf{p}(r)\| = \max \mathbf{p}(r) - \min \mathbf{p}(r).$$

Since every node produces its round r pulse at the same logical time, i.e.,

$$L_v(p_v(r)) = T(0) + T(1) + \cdots + T(r-1) + \tau_1(r),$$

bounds on the pulse width $\|\mathbf{p}(r)\|$ together with bounds on $T(r)$ and logical clock rates imply corresponding bounds on the skew in round r. Our analysis in [19] yields the following bound on skew.

Lemma 6.3. *Suppose logical clocks are initialized so as to ensure $\|\mathbf{p}(0)\| \leq e_g^\infty$, and parameters are chosen such that $\alpha_g < 1$ in Equation* (6.3)*. Then for all $r \in \mathbb{N}$ take*

$$\begin{aligned} \tau_1(r) &= \tau_1 = \vartheta_g \cdot e_g^\infty, \\ \tau_2(r) &= \tau_2 = \vartheta_g \cdot (e_g^\infty + d), \\ \tau_3(r) &= \tau_3 = \vartheta_g \cdot \frac{1}{\varphi} \cdot (e_g^\infty + U). \end{aligned} \tag{6.4}$$

Suppose $\rho > 0$ is sufficiently small. Then for suitable choices of μ and φ, for all $t \in \mathbb{R}^+$ and non-faulty nodes $v, w \in C \setminus F$, we have

$$\mathcal{L}(t) \leq 6e_g^\infty. \tag{6.5}$$

We note that $\mathcal{L}(t)$ is bounded by $\mathcal{O}(\rho \cdot d + U)$, which asymptotically matches the best possible bound for fault-tolerant clock synchronization in a fully connected network [72].

Remark. In fully connected networks $\mathcal{L}(t) = \mathcal{G}(t)$.

6.4 Gradient Clock Synchronization

In this section, we present the GCS algorithm of Lenzen et al. [67]. The algorithm achieves close synchronization between neighboring nodes in an arbitrary network, i.e., it minimizes $\mathcal{L}(t)$. Let δ be an upper bound on how precisely the skew between neighbors is known. Provided that the global skew does not exceed a bound of $\mathcal{O}(\delta D)$, it achieves asymptotically optimal local skew bounded by $\mathcal{O}(\delta \log_{\mu/\rho} D)$. In other words, local skew grows only logarithmic in the hop diameter of the network, still, we have clocks that progress at a minimum rate of 1. The local and global skew bounds are optimal up to roughly factor 2 [67].

Intuitively, the algorithm executed by node v continuously measures the skew to each neighbor w. By a set of rules, the algorithm decides whether to progress

the logical clock at a fast or a slow rate. The set of rules basically defines pairs of thresholds on the measured skews. Skew is measured in discrete steps. When there is a neighbor that has larger skew (negative or positive) than all other neighbors, then the algorithm changes the rate accordingly. We associate a slow rate with waiting for a neighbor and a fast rate with catching up to a neighbor.

6.4.1 Algorithm

The GCS algorithm by Lenzen et al. computes a logical clock from the hardware clock in two different modes; fast and slow. In slow mode, the logical clock follows the rate of the hardware clock. In fast mode, the logical clock advances at the hardware clock rate and *speedup factor* $\mu > 0$. Formally, a node in fast mode advances its logical clock with rate $(1+\mu)h_v(t)$. Factor μ is not constrained by the system, it is a parameter adjustable by the designer. A node controls a binary switch $\gamma_v(t) \in \{0,1\}$ to adjust its logical clock. In *fast mode* γ_v is set to 1 and, accordingly, in *slow mode* γ_v is set to 0. The logical clock value of v at time t with offset L_v^0 at time 0 is determined by

$$L_v(t) = \int_0^t (1 + \mu \cdot \gamma_v(\tau))\, h_v(\tau)\, d\tau + L_v^0 \,.$$

Clearly, a node in fast mode must be able to catch up to a node in slow mode. Hence, we pose the constraint that fast mode (without clock drift) can never be slower than slow mode (with clock drift). This can be formalized as

$$1 + \rho < 1 + \mu \,. \tag{6.6}$$

The algorithm specifies two conditions that control when to switch between fast and slow mode. Accordingly, conditions are named *fast condition* (FC) and *slow condition* (SC). The algorithm is parametrized by κ, which determines the quality of synchronization.

Definition 6.4 (fast and slow condition)**.** *Let $\kappa \in \mathbb{R}^+$ be a positive, non-zero, real number. A node $v \in V$ satisfies the fast condition at time t if there is a natural number $s \in \mathbb{N}$ such that the following two statements hold:*

$$\exists (v,x) \in E : L_x(t) - L_v(t) \geq (2s+1)\kappa \tag{FC-1}$$

$$\forall (v,y) \in E : L_y(t) - L_v(t) \geq -(2s+1)\kappa \tag{FC-2}$$

Node $v \in V$ satisfies the slow condition at time t if there is a natural number $s \in \mathbb{N}$ such that the following two conditions hold:

$$\exists (v,x) \in E : L_x(t) - L_v(t) \leq -2s\kappa \tag{SC-1}$$

$$\forall (v,y) \in E : L_y(t) - L_v(t) \leq 2s\kappa \tag{SC-2}$$

Intuitively, skew is measured in steps of κ. Node v satisfies FC if there is at least one node u, that is ahead of v and no other node behind v exceeds the absolute skew

between v and u. Node v satisfies SC if there is a node u behind v that has a larger absolute skew to v than all nodes ahead of v. The thresholds use odd multiples of κ for FC and even multiples of κ for SC to ensure mutual exclusion.

If v is the node with the largest logical clock value in the network, then all other nodes are behind v, they have negative offset. Thus, SC is satisfied for $s = 0$. Accordingly, if v is the node with the smallest clock value in the network, then it satisfies FC as all offsets to other nodes are positive.

Definition 6.5. *An algorithm is a* GCS algorithm *with parameters ρ, μ, κ if the following invariants hold, for every node $v \in V$ and all times t, t':*

$$\mu > \rho\,, \tag{I1}$$

$$H_v(t') - H_v(t) \leq L_v(t') - L_v(t) \leq (1+\mu)(H_v(t') - H_v(t))\,, \tag{I2}$$

if v satisfies FC at time t then v is in fast mode at time t, (I3)

if v satisfies SC at time t then v is in slow mode at time t. (I4)

Invariant (I2) only states that the rate of the logical clock is at least the rate of the hardware clock and at most $(1 + \mu)$ times the rate of the hardware clock.

Remark. Every algorithm that meets Definition 6.5 is a GCS algorithm. In this work we often regard the algorithm by Lenzen, Locher, and Wattenhofer [67], we will present it later as the OffsetGCS algorithm. The analysis of [62], however, applies to every GCS algorithm.

Offset Estimates. Nodes have no access to the logical clocks of their neighbors. Hence, precise skews remain unknown to the node. In order to fulfill the invariants of the algorithm a node maintains an estimate of each offset to a neighbor. The *offset estimate* of node v to its neighbor w is denoted by $\widehat{O}_w$. Intuitively, a node maintains $\widehat{O}_w(t) \approx L_w(t) - L_v(t)$. Parameter δ gives a two-sided bound on the quality of the estimates.

$$\left|\widehat{O}_w(t) - (L_w(t) - L_v(t))\right| \leq \delta \tag{6.9}$$

Given an estimate of each neighboring clock, the GCS algorithm specifies the *fast trigger* (FT). Every node that satisfies fast condition (FC) must satisfy FT, but a node that satisfies slow condition (SC) must not satisfy FT. A node determines by FT whether to go to fast or slow mode.

Definition 6.6 (fast trigger)**.** *Let $\kappa \in \mathbb{R}^+$ be a positive, non-zero, real number. A node $v \in V$ satisfies the fast trigger at time t if there is a natural number $s \in \mathbb{N}$ such that the following two statements hold:*

$$\exists (v, x) \in E : \widehat{O}_x(t) \geq (2s+1)\kappa - \delta \tag{FT-1}$$

$$\forall (v, y) \in E : \widehat{O}_y(t) \geq -(2s+1)\kappa - \delta \tag{FT-2}$$

Algorithm 2 GCS algorithm at node v

```
at each time t do
    for each neighbor w do
        o_w ← Ô_w(t)                              ▷ save offset estimate to w
    ft1(s) ← ∃w : o_w ≥ (2s + 1)κ − δ
    ft2(s) ← ∀w : o_w ≥ −(2s + 1)κ − δ
    if ∃s : ft1(s) ∧ ft2(s) then
        γ_v(t) ← 1                                ▷ switch to fast mode
    else
        γ_v(t) ← 0                                ▷ switch to slow mode
```

We are now able to state the GCS algorithm in Algorithm 2. Intuitively, the algorithm checks the fast trigger (FT) at all times. If v satisfies FT then v switches to fast mode, otherwise v defaults to slow mode.

Remark. As the decision to run fast or slow is a discrete decision, a hardware implementation will be prone to metastability [74]. We discuss how to deal with this problem in Chapters 9 and 10.

Maximal and Minimal Offset. The fast and slow condition of Lenzen et al. can be reformulated using the largest and smallest offset to node v. The maximal offset at node v is given by

$$O_{\max}(t) := \max_{(v,x)\in E}\{L_x(t) - L_v(t)\}\,.$$

The node with largest offset to v is the node most ahead of v. Similarly, the minimal offset is defined by

$$O_{\min}(t) := \min_{(v,x)\in E}\{L_x(t) - L_v(t)\}\,.$$

A node $v \in V$ satisfies FC if there is a neighbor that reaches a certain (positive) threshold and no offset to any neighbor crosses the corresponding negative offset. If the maximal offset reached a certain threshold we are certain that there is a corresponding neighbor with this offset. Accordingly, if the minimal offset is larger than the corresponding negative offset, we are certain that there is no neighbor with an offset smaller than the threshold. Formally, we restate the fast condition as

$$O_{\max}(t) \geq (2s+1)\kappa \tag{FC-1}$$

$$O_{\min}(t) \geq -(2s+1)\kappa \tag{FC-2}$$

Accordingly, we can restate the slow condition as

$$O_{\min}(t) \leq -2s\kappa \tag{SC-1}$$

$$O_{\max}(t) \leq 2s\kappa \tag{SC-2}$$

Algorithm 3 OffsetGCS algorithm at node v

```
at each time t do
    for each adjacent node w do
        Float o_w ← Ô_w(t)                          ▷ save offset estimate to w
    o_max ← max{o_w}                                ▷ compute maximal estimate
    o_min ← min{o_w}                                  and minimal estimate
    ft1(s) ← o_max ≥ (2s + 1)κ − δ
    ft2(s) ← o_min ≥ −(2s + 1)κ − δ
    if ∃s : ft1(s) ∧ ft2(s) then
        γ_v(t) ← 1                                  ▷ switch to fast mode
    else
        γ_v(t) ← 0                                  ▷ switch to slow mode
```

We define the maximal and the minimal estimate of v's offset estimates by

$$\widehat{O}_{\max} \coloneqq \max_{(v,x)\in E}\{\widehat{O}_x\}\,, \text{ and}$$
$$\widehat{O}_{\min} \coloneqq \min_{(v,x)\in E}\{\widehat{O}_x\}\,.$$

We can bound s by a natural number ℓ. The largest skew between two neighbors is bounded by $\mathcal{L}$. Let ℓ be the largest number such that $(2\ell+1)\kappa - \delta \le \mathcal{L}$. Then node $v \in V$ satisfies the fast trigger at time t if there is an $s \in [\ell+1]$ such that the following two statements hold:

$$\widehat{O}_{\max}(t) \ge (2s+1)\kappa - \delta \tag{FT-1}$$
$$\widehat{O}_{\min}(t) \ge -(2s+1)\kappa - \delta \tag{FT-2}$$

We are now able to restate the GCS algorithm with respect to $\widehat{O}_{\max}$ and $\widehat{O}_{\min}$ rather than quantifiers "exists" and "for all" over all neighbors. The algorithm is presented in Algorithm 3. It is called OffsetGCS, it will be used throughout the rest of this dissertation.

Figure 6.1 visualizes conditions FC and SC. In a coordinate system where we mark $O_{\min}$ along the x-axis and $O_{\max}$ along the y-axis we can mark FC and SC as colored regions. We mark maximal and minimal offsets as point $(O_{\min}, O_{\max})$ in the plane. Offsets are denoted by a small rectangle. An algorithm that satisfies Definition 6.5 has to go to fast mode at any time $(O_{\min}, O_{\max})$ falls into the FC (yellow) region. Similarly, if $(O_{\min}, O_{\max})$ falls into the SC (blue) region, the algorithm needs to go to slow mode.

Due to the uncertainty in message delay the actual point $(\widehat{O}_{\min}, \widehat{O}_{\max})$ may lie in the δ surrounding of a measurement, which is depicted by a larger box surrounding

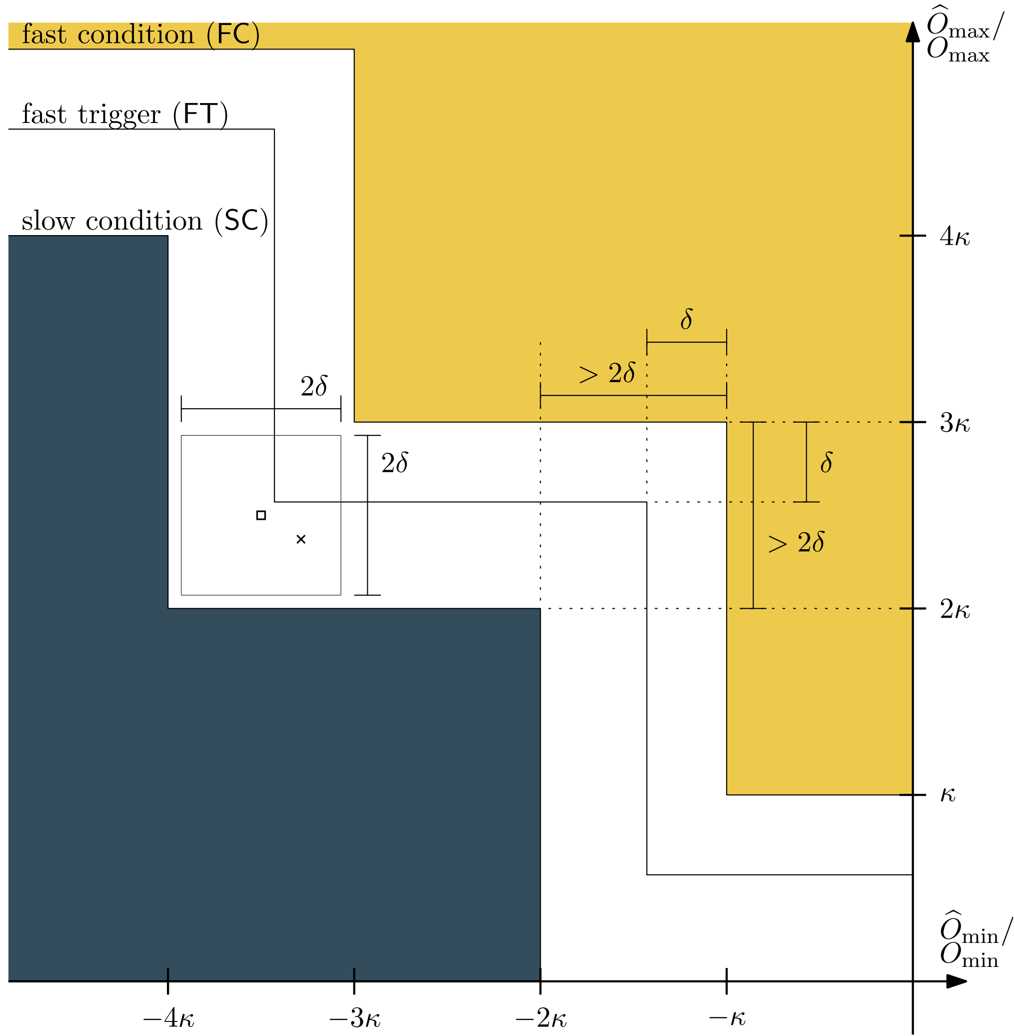

Figure 6.1: Visualization of FC and SC in the plane of $O_{\min}$ and $O_{\max}$. The exact measurement $(O_{\min}, O_{\max})$ is denoted by a small rectangle. FT is marked only by a line in the plane of $\widehat{O}_{\min}$ and $\widehat{O}_{\max}$. The actual measurement $(\widehat{O}_{\min}, \widehat{O}_{\max})$ is depicted as a small cross. According to Definition 6.5, the logical clock must be fast when $(O_{\min}, O_{\max})$ is within the FC region. Respectively, it must be slow when $(O_{\min}, O_{\max})$ is within the SC region. The OffsetGCS algorithm switches to fast when $(\widehat{O}_{\min}, \widehat{O}_{\max})$ is within the FT region and to slow otherwise.

$(O_{\min}, O_{\max})$. The point $(\widehat{O}_{\min}, \widehat{O}_{\max})$ is denoted by a cross. With regard to $\widehat{O}_{\min}$ and $\widehat{O}_{\max}$ along the x-axis and y-axis we can also mark FT as a region. A node that executes OffsetGCS chooses fast or slow mode depending on whether $(\widehat{O}_{\min}, \widehat{O}_{\max})$ lies in the FT region. Intuitively, every cross that is above the FT border causes the OffsetGCS algorithm to go to fast mode.

Furthermore, Figure 6.1 visualizes that for any $(O_{\min}, O_{\max})$ in the FC region, all possible points $(\widehat{O}_{\min}, \widehat{O}_{\max})$ will be within the FT region, they can never cause OffsetGCS to go slow. Similarly any $(O_{\min}, O_{\max})$ in the SC region can never cause OffsetGCS to go fast.

Remark. Only the second quadrant of the coordinate plane is of interest. As we also include all edges (v, v), each node measures the offset to itself. Thus, we also include an offset measurement of 0, disregarding the measurement error δ. Hence, $\widehat{O}_{\max}$ is bounded below and $\widehat{O}_{\min}$ is bounded above by 0.

Example 6.7. *Figure 6.2 shows an example trajectory of $(O_{\min}, O_{\max})$ for an execution of OffsetGCS. Small rectangles denote the position of $(O_{\min}, O_{\max})$ when measuring. Large boxes denote the set of all possible estimates and a cross denotes the actual measurement $(\widehat{O}_{\min}, \widehat{O}_{\max})$. Measurements are continuous, there is a measurement at every point along the trajectory. Exemplary, we depict only the measurements where the algorithm changes mode. Fast mode is indicated by a dash-dot-dot line and slow mode by a dash-dot line. In Chapter 10 we implement and simulate the OffsetGCS algorithm. A trajectory of $(\widehat{O}_{\min}, \widehat{O}_{\max})$ from a simulation of our implementation is depicted in Figure 10.15. The simulation setup is described in Section 10.4.1.*

6.4.2 Analysis

In what follows, we show that for a suitable choice of parameters, OffsetGCS is a GCS algorithm in the sense of Definition 6.5. Thus, OffsetGCS maintains the skew bounds of [13], which can be stated as follows.

Theorem 6.8 (from [13])**.** *Suppose algorithm A is a GCS algorithm in the sense of Definition 6.5 with $\mu > 2\rho$. Then A maintains global and local skew*

$$\mathcal{G}(t) \leq \frac{\mu\kappa D}{\mu - 2\rho} \qquad\qquad \mathcal{L}(t) \leq \left(2\left\lceil \log_{\mu/\rho} \frac{\mu D}{\mu - 2\rho} \right\rceil + 1\right)\kappa$$

for initial skew $\Delta \leq \mu\kappa D/(\mu - 2\rho)$.

Remark. In the formal analysis, the precise local and global skew bounds achieved by a GCS algorithm at an arbitrary time t depend on the initial state of the system. GCS algorithms are self-stabilizing in the sense that starting from an arbitrary initial state, the algorithm will eventually achieve the skew bounds claimed in Theorem 6.8 (cf. [62]). For the implementations in Chapters 9 and 10 this is not true in general. Initial skew exceeding the capability of the measurement circuit will lead to non-stabilizing behavior.

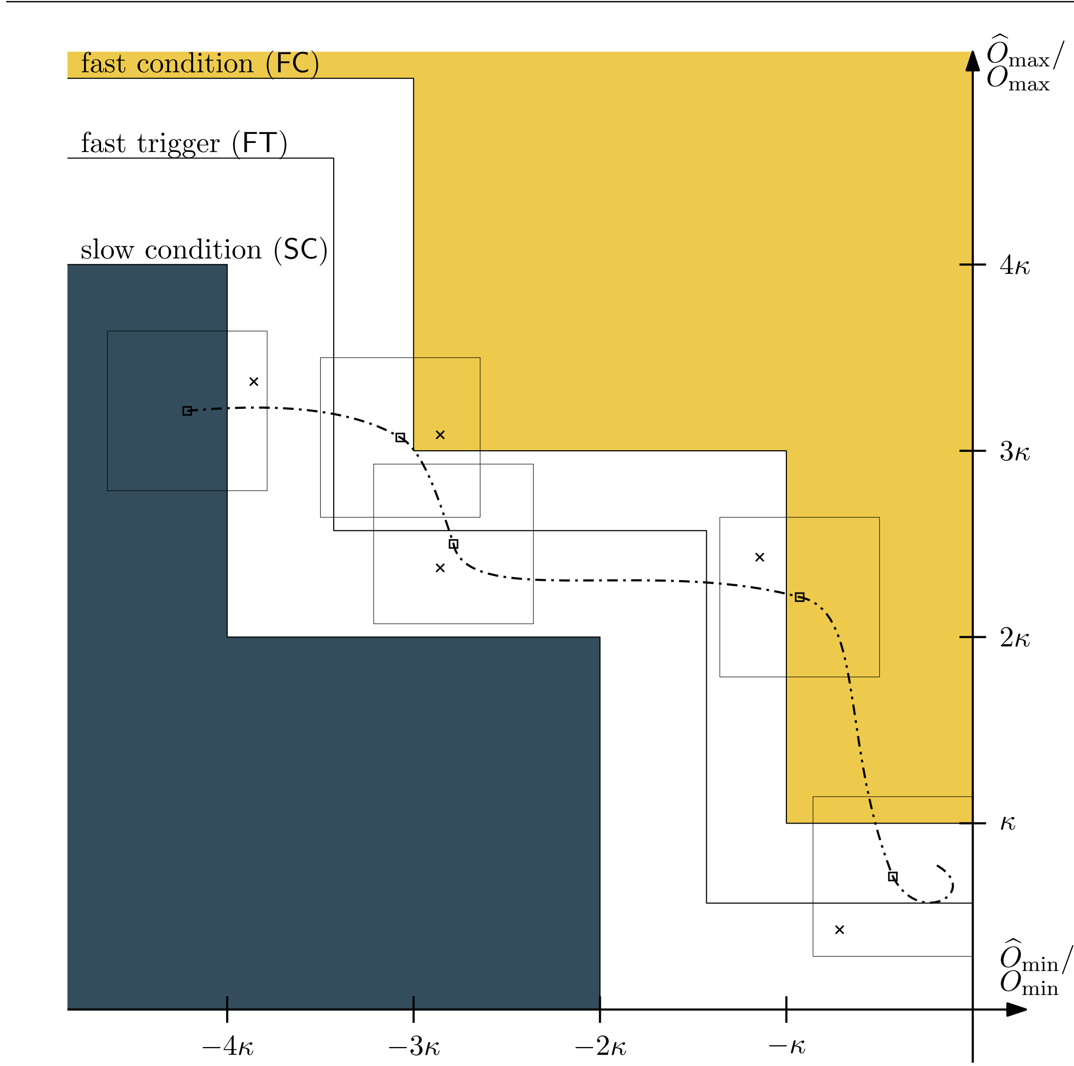

Figure 6.2: Example trajectory of $(O_{\max}, O_{\min})$ of a node executing OffsetGCS. The node is starting with large offsets to its neighbors, during execution it is converging to smaller offsets. Measurements are taken continuously, we only depict the ones that change the mode of the clock. Fast mode is indicated by a dash-dot-dot line and slow mode by a dash-dot line.

Initialization. The bound on the local skew achieved by flooding described in Section 6.2 does not match the requirement $\Delta \leq \mu\kappa D/(\mu - 2\rho)$. As the GCS algorithm is self-stabilizing we can start the execution with a larger initial skew and the system will converge to the claimed skew bounds. In [14] we show that given an arbitrary global skew of $\mathcal{G}(0)$ the system will converge to the skew bounds claimed in Theorem 6.8 within time $\mathcal{O}(\mathcal{G}(0)/\mu)$. The flooding technique can bind the initial global skew by $\mathcal{G}(0) \leq (\rho d + U)D$. We could choose $\mu = U/d$, such that convergence takes $\mathcal{O}(dD)$ time. Then convergence is as fast as the initialization procedure itself, i.e., it takes $\mathcal{O}(dD)$ time to achieve the bounds claimed in Theorem 6.8.

Uncertainty. In our analysis, it will be helpful to distinguish two sources of uncertainty faced by any implementation of the GCS algorithm. The first is the *propagation delay uncertainty*, which is the absolute timing variation in signal propagation adding to the measurement error. We use the parameter $\delta_0 > 0$ to denote an upper bound on this value.

The second source of error is the time between initiating a measurement and actually using it in control of the logical clock speed. During this time, the logical clocks advance at rates that are not precisely known. Here, we can exploit that the maximum rate difference between any two logical clocks is $(1+\rho)(1+\mu)-1 = \rho+\mu+\rho\mu$. Thus, denoting the *maximum end-to-end latency* by $T_{\max}$, this contributes an error of at most $(\rho + \mu + \rho\mu)T_{\max}$ at any given time. Time $T_{\max}$ includes the time for the logical clock to respond to the control signal. Once suitable values of δ_0 and $T_{\max}$ are determined, δ can be computed easily.

Lemma 6.9. *When $\delta = \delta_0 + (\rho + \mu + \rho\mu) \cdot T_{\max}$, then inequality (6.9) holds.*

Based on δ, we now seek to choose κ as small as possible to realize the invariants given in Definition 6.5. The basic idea is to ensure that if a node v satisfies FC at time t, then it must satisfy FT. In turn, if SC is satisfied, we must make sure that FT is not satisfied.

Theorem 6.10. *Suppose for all times t an implementation of OffsetGCS satisfies*

$$\mu > \rho\,, \qquad \text{(Equation (I1))}$$

$$\delta \geq \left|\widehat{O}_w(t) - (L_w(t) - L_v(t))\right|\,, \qquad \text{(Equation (6.9))}$$

and additionally

$$\kappa > 2\delta\,,$$

then OffsetGCS is a GCS algorithm.

Proof. We verify the conditions of Definition 6.5. By assumption condition (I1) is satisfied. Condition (I2) is a direct consequence of the algorithm specification. For Condition (I3), suppose first that v satisfies the fast condition at time t. Therefore,

there exists some $s \in \mathbb{N}$ and neighbor x of v such that $L_x(t) - L_v(t) \geq (2s+1)\kappa$. Therefore, by (6.9), $\widehat{O}_x(t) \geq (2s+1)\kappa - \delta$, so that (FT-1) is satisfied. Similarly, since v satisfies the fast condition, all of its neighbors y satisfy $L_v(t) - L_y(t) \leq (2s+1)\kappa - \delta$. Therefore, $\widehat{O}_y(t) \geq -(2s+1)\kappa$, hence (FT-2) is satisfied for the same value of s and v runs in fast mode at time t.

It remains to show that if v satisfies the slow condition at time t, then it does not satisfy FT at time t and, accordingly, is in slow mode. Suppose that v satisfies SC and FT at time t. Then by SC,

$$\exists x\colon L_v(t) - L_x(t) \geq 2s\kappa - \delta\,, \tag{6.11}$$

$$\forall y\colon L_y(t) - L_v(t) \leq 2s\kappa + \delta\,. \tag{6.12}$$

Since v is assumed to satisfy FT at time t, combining (FT-1) and (FT-2) with (6.9) imply that there exists some $s' \in \mathbb{N}$ with

$$\exists x\colon L_x(t) - L_v(t) \geq (2s'+1)\kappa - \delta\,, \tag{6.13}$$

$$\forall y\colon L_v(t) - L_y(t) \leq (2s'+1)\kappa + \delta\,. \tag{6.14}$$

Combining (6.12) and (6.13), we must have

$$\begin{aligned}(2s'+1)\kappa - \delta &\leq 2s\kappa + \delta \\ \Leftrightarrow 2s'\kappa &\leq 2s\kappa - \kappa + 2\delta\,.\end{aligned}$$

Since $2\delta < \kappa$, the previous expression implies that $s' < s$. Similarly, combining (6.11) and (6.14) gives

$$\begin{aligned}2s\kappa - \delta \cdot T_{\max} &\leq (2s'+1)\kappa + \delta \\ \Leftrightarrow 2s\kappa &\leq 2s'\kappa + 2\delta \\ \Leftrightarrow 2s\kappa &< 2(s'+1)\kappa\end{aligned}$$

Thus, $s < s'+1$, or equivalently (since s and s' are integers), that $s \leq s'$. However, this contradicts the previous conclusion, that $s' < s$. Thus FT cannot be satisfied at time t if the slow condition is satisfied at time t, as we assumed. □

The OffsetGCS algorithm is a GCS algorithm, hence we can apply an analysis similar to [62]. The analysis is given in [19], and it yields the skew bounds mentioned in Theorem 6.8.

Fault Tolerant Gradient Clock Synchronization 7

This chapter presents the results published at PODC 2019 (cf. [18]). The complete proof of the main result is given in the full version [19]. We show how to combine the Lynch-Welch and a variant of the GCS algorithm by Lenzen, Locher, and Wattenhofer. The combination gets the best of both worlds; a fault-tolerant algorithm on arbitrary networks, assuming that we can extend the network.

Outline. We introduce the topic and discuss related work in Section 7.1. We formalize our computational model in Section 7.2. Section 7.3 provides further description of the intra-cluster synchronization portion of our algorithm, ClusterSync, and states our main technical lemmas. Section 7.4 contains a description of the inter-cluster synchronization portion of our algorithm, InterclusterSync, and provides a proof of our main result, Theorem 7.1. Due to space restrictions, we do not provide proofs of the technical lemmas leading to the proof of Theorem 7.1. Technical details are provided in the full version of this paper, which is available online [19].

7.1 Introduction and Related Work

7.1.1 Introduction

Synchronizing clocks across distributed systems is a fundamental task. Clock synchronization may be used for coordination, e.g. in a time division multiple access scheme to a shared resource such as a wireless channel. Clock synchronization also plays a crucial role in (distributed) measurements by enabling a system to correctly correlate data, and it can form the basis of a decentralized system clock for a System-on-Chip or Network-on-Chip.

Gradient clock synchronization (GCS) algorithms aim to minimize the *local skew,* i.e., the worst-case phase difference between the logical clocks computed by neighbors in a network graph. While the *global skew* is linear in the network diameter D [7], a bound of $\Theta(\log_b D)$ on the local skew was established in [67]. Here, the base is $b = \mu/\rho$, where $1 + \mu$ is the maximum factor by which a node speeds up its logical clocks relative to its hardware clock. The bound of $\Theta(\log_b D)$ is tight, subject to the constraints that (i) the nodes' hardware clocks always run at rates between 1 and $1 + \rho$, (ii) nodes' logical clocks always run at rates between 1 and $(1 + \rho)(1 + \mu)$, and (iii) $\mu/\rho = 1 + \Omega(1)$.

The problem of minimizing the local skew is well-studied. Under worst-case assumptions (on hardware clock rates, message delays, etc), small bounds on the local skew can be achieved with logical clocks whose parameters (e.g., maximum

and minimum rates) are only slightly worse than the parameters of the underlying hardware clocks. Results in the same vein hold for dynamic graphs [63, 62]. The results for dynamic networks immediately imply analogous results for networks with crash faults. From the perspective of the remaining system, a node crashing is equivalent to removing all of its incident links. The algorithm achieving small local skew is inherently self-stabilizing, provided one ensures that excessive global skews are detected and reduced, e.g., by a reset procedure.

Unfortunately, the GCS algorithm utterly fails in the face of non-benign faults. Even a single node that refuses to adjust its logical clock rate would invalidate any non-trivial bound on skews. Given that distributed systems of sufficient size invariably tend to violate (overly optimistic) specifications [85], this raises the following question.

Can small local skew be achieved despite Byzantine faults?

When posing this question, obviously we need to restrict attention to skews between non-faulty nodes only. Nonetheless, the answer is trivially "no" in general: a node with exactly two neighbors, one of which is Byzantine, cannot reliably decide which neighbor's clock it should follow. More generally, a node with up to f faulty neighbors must have at least $2f+1$ neighbors to avoid the trivial impossibility of synchronization. A more careful argument shows that, without cryptographic assumptions, $n > 3f$ is necessary even if the network is a clique [32]. In the case of a clique, the Lynch-Welch algorithm [71] achieves asymptotically optimal skew for any n and f satisfying $n > 3f$.

The Lynch-Welch algorithm, like other fault-tolerant clock synchronization algorithms designed for cliques, can be extended to networks of larger diameter in a straightforward way. The approach is discussed in [33] for the synchronous setting, but the argument transfers to the "semi-synchronous" model we consider without issue. The idea is to set up a central cluster running the algorithm and synchronize "slave" clusters of nodes to the central cluster. That is slave clusters simply "echo" the clock pulses generated by the central cluster. Clusters farther away from the central cluster echo the messages of slave clusters, and so on, resulting in a tree structure with each tree node corresponding to a cluster of nodes. So long as in each cluster, less than one third of the nodes is faulty, subsequent groups can synchronize both to their parent cluster in the tree and among themselves, using the same technique based on approximate agreement [35] that lies at the heart of the Lynch-Welch algorithm.

If clusters have uniform size of $3f+1$ and nodes fail independently with probability p, then the probability that more than f nodes in a cluster are faulty is

$$\sum_{i=f+1}^{3f+1} \binom{3f+1}{i} p^i (1-p)^{3f+1-i} \leq \binom{3f+1}{f+1} p^{f+1} \leq (3ep)^{f+1} . \tag{7.1}$$

Thus, if the *distribution* of faults across the system is benign, even small choices of f can improve reliability dramatically, without causing impractically large degrees. For $f \in \Theta(\log n)$, the system as a whole operates correctly with high probability even for a constant failure probability p of individual nodes.

The simplistic approach of synchronizing clusters to a central cluster succeeds in the sense that it achieves asymptotically optimal global skew in a sparse network (assuming that at most f nodes fail in each cluster). However, it does not offer a non-trivial bound on the local skew. Setting $f = 0$, the algorithm specializes in a simple master-slave synchronization algorithm on (fault-free) tree topologies. If a clock pulse propagates through a line network with the global skew equally distributed over the line, this will "compress" the full global skew onto a single edge, cf. [70]. In contrast, GCS algorithms need to take into account whether neighbors are lagging behind when deciding how to adjust their logical clocks. More sophisticated strategies are needed for a fault-tolerant GCS algorithm!

7.1.2 Contribution

We present a simple and general transformation that takes an arbitrary network $\mathcal{G}$ and yields a larger network on which we can achieve fault-tolerant GCS with asymptotically optimal local skew. The basic idea is simple enough: each node is replaced by a (fully connected) cluster of $3f + 1$ nodes and each edge is replaced by a complete bipartite graph between the respective clusters. We then use the Lynch-Welch algorithm [71] to synchronize within clusters and simulate the (non-fault-tolerant) GCS algorithm from [67] on virtual clocks defined for the clusters.

Our approach is simplified by employing the two algorithms (almost) as black boxes. This is made possible by leveraging the worst-case assumptions on clock rates the Lynch-Welch algorithm can handle: Our algorithm treats the speed adjustments made by the concurrently running GCS algorithm as changes in "hardware" clock speeds. Thus we fully exploit the analysis of the Lynch-Welch algorithm without having to prove statements again from scratch. Similarly, the GCS algorithm from [67] can be phrased such that each node acts solely on estimates of real-time differences between events (which any node gets from its hardware clock) and estimates of neighboring logical clocks. We exploit the latter by having nodes use their own logical clocks in the Lynch-Welch algorithm as stand-ins for the (virtual) cluster clocks, which they can never know precisely.

The main obstacle to this approach is that using both algorithms as black boxes results in the problem with the "additional" (amortized) clock drift induced by the Lynch-Welch algorithm's corrections to nodes' clocks. This additional drift is proportional to the (intra-cluster) synchronization quality divided by the length of the resynchronization interval. The synchronization quality is proportional to the difference between the maximum and minimum clock rate times the length of the resynchronization interval. This means that a naive analysis would require that the clock drift the GCS algorithm needs to combat is at least as large as the increase in clock speed the GCS algorithm is willing to use to do so. In other words, fast-running clocks would need to be able to outrun other fast-running clocks, leaving the GCS algorithm with no way of reducing skews at all.

We resolve this circularity in the naive implementation by exploiting the convergence properties of the Lynch-Welch algorithm in combination with the flexibility

of the GCS algorithm. The GCS algorithm allows for slack between conditions under which logical clocks *must* run fast or slow, and *triggers* that indicate when a node actually switches from fast to slow mode, or vice versa. This slack enables an implementation despite the fact that the clock values of neighbors are never known exactly. We also use this slack to buy some time for responding to critical skews. By the time a cluster is required to be in, say, fast mode, *all* (correct) nodes in the cluster will have satisfied the fast trigger for some time. The period of time between when the fast triggers are unanimously satisfied and when the node is required to run fast is sufficiently long that the Lynch-Welch algorithm converges to a smaller skew within the cluster. Thus, when the cluster is required to run fast, the Lynch-Welch corrections are significantly smaller than the worst-case bounds for non-unanimous clusters. This behavior allows us to bind the amortized clock rates of individual nodes in clusters satisfying the fast or slow conditions, thereby showing that fast clusters indeed run faster than slow clusters. The gap is sufficiently large that we can apply the analysis of the GCS algorithm as a black box. Specifically, we show the following.

Theorem 7.1. *Let $\mathcal{G} = (\mathcal{C}, \mathcal{E})$ be an arbitrary network. Let $G = (V, E)$ be the augmented graph formed by replacing each node in $\mathcal{G}$ with a clique of $k \geq 3f + 1$ nodes, and fully connecting such cliques if they correspond to neighbors in $\mathcal{G}$. Suppose messages in G are subject to maximum delay d, delay uncertainty U, and hardware clock drifts are at most ρ, where ρ is sufficiently small. Suppose further that within each cluster corresponding to a node in $\mathcal{G}$, at most f of the nodes are faulty. Then there exists an algorithm that computes logical clocks $L_v(t)$ for each correct node $v \in V \setminus F$ such that (i) for each $v \in V \setminus F$, L_v increases at rates between 1 and $1 + \mathcal{O}(\rho)$ and (ii) for all $\{v, w\} \in E$ with $v, w \in V \setminus F$, and for all $t \in \mathbb{R}^+$, we have*

$$|L_v(t) - L_w(t)| = \mathcal{O}((\rho \cdot d + U) \log D)\,,$$

where D is the network diameter.

7.1.3 Related Work

Srikanth and Toueg [89] introduced a basic synchronization algorithm that can cope with Byzantine faults. In a fully connected network, the Srikanth-Toueg algorithm maintains synchronization among the nodes by a propose-and-pull mechanism: Nodes propose to resynchronize either after a local timeout or upon having received at least $f + 1$ proposals. The nodes then resynchronize after receiving at least $n - f$ proposals. This procedure achieves asymptotically optimal [72] skew of $\mathcal{O}(d)$ despite $f < 3n$ Byzantine faults in the case where there is no guaranteed *lower* bound on the communication delay. The Lynch-Welch algorithm [71] improves the skew to $\mathcal{O}(U + (\vartheta - 1)d)$ under the additional assumption that message delay is at least $d - U$ time. The skew $\mathcal{O}(U + (\vartheta - 1)d)$ is asymptotically optimal, and is strictly better than the $\mathcal{O}(d)$ skew attained by the Srikanth and Toueg algorithm when there is some a priori lower bound on the minimum message delay. The Lynch-Welch algorithm achieves synchronization by simulating synchronous rounds, each of which is used to

perform an approximate agreement [35] step on when the round should have started and adjusting clocks accordingly.

Either the Srikanth-Toueg or the Lynch-Welch algorithm could be employed in our construction, where we chose Lynch-Welch for its better skew. Both algorithms also share the characteristic that, in their basic variants, logical clocks “jump” to implement phase corrections. This discontinuous behavior is incompatible with the requirement that logical clocks satisfy lower and upper bounds on their rates in a GCS algorithm. This issue is easily addressed by amortizing clock adjustments over sufficient periods of time [66]. The amortization process requires that we adjust the “round length” of these algorithms, but this change has no asymptotic impact on skews.

A series of works considers synchronization algorithms that are simultaneously resilient to $f < n/3$ Byzantine faults and self-stabilizing [29, 34, 36, 57, 69]. That is, synchronization is re-established despite the (ongoing) interference from Byzantine faulty nodes after transient faults cease. Dolev and Welch [36] proposed the problem, proving that it can actually be solved. However, their algorithm has exponential stabilization time, i.e., $2^{\Omega(f)}d$ time may pass after transient faults cease before the logical clocks meet the synchronization and progress requirements. The stabilization time was improved to polynomial [29], then linear [34], and finally (randomized) logarithmic [69]. The latter construction transforms any synchronous R-round consensus algorithm into a solution to the problem that stabilizes in $\mathcal{O}(R \log n)$ time and sends $\mathcal{O}(M \log n)$ bits over each link in $\Theta(d)$ time, where M is the message size of the consensus algorithm. If the consensus algorithm is randomized, the transformation works the same way, but the stabilization time bound holds with high probability (instead of deterministically). Besides the smallest known stabilization time, this transformation also yields the best-known trade-offs between stabilization time and the amount of communication. All of these algorithms have in common that they achieve $\mathcal{O}(d)$ skew, as they rely on the propose-and-pull mechanic underlying the Srikanth-Toueg algorithm. However, such algorithms can be used to make the Lynch-Welch algorithm self-stabilizing, by utilizing the inaccurate (and typically also infrequent) synchronization events to “jump-start” the simulation of synchronous approximate agreement rounds upon which the Lynch-Welch algorithm is based [57]. The result is a routine that combines the extreme resilience of the self-stabilizing routine, with the asymptotically optimal skew of the Lynch-Welch algorithm.

To date, GCS has been studied in fault-free networks only. The problem was introduced by Fan and Lynch [39], alongside a surprising lower bound of $\Omega(\log D/ \log\log D)$ on the local skew. The first non-trivial upper bound of $\mathcal{O}(\sqrt{D})$ on the local skew is due to Locher and Wattenhofer [70]. In the algorithm of [70], nodes try to catch up with the maximum logical clock value among their neighbors, but under the constraint that they never run faster than their hardware clock rate when there is a neighbor whose clock lags $\Theta(\sqrt{D})$ or more behind. As the global skew is bounded by $\mathcal{O}(D)$, at most $\mathcal{O}(\sqrt{D})$ consecutive nodes can be “blocked” from catching up. Thus, an individual node is not prevented from speeding up for more than $\Theta(\sqrt{D})$ time. This gives rise to the bound on the skew. Subsequently, the tight bound of $\Theta(\log D)$ on the

local skew was established in [67]. The GCS algorithm in [67] can be seen as switching between the "catching up" and "blocking" strategy depending on the amount of local skew witnessed by a node. Specifically, each node compares the largest $s \in \mathbb{N}_0$ such that some neighbor's clock is at least $s\kappa$ ahead to the largest $s' \in \mathbb{N}_0$ such that some neighbor's clock is at least $s'\kappa$ behind, for some suitably chosen κ. One can then show that the length of paths with a sufficient skew to "block" nodes from catching up decreases exponentially with s, which yields the stated bound.

The algorithmic approach of the GCS algorithm [67] turns out to be quite robust and flexible. The GCS algorithm generalizes to networks in which edges $e = \{v, w\}$ have weights ε_e indicating the accuracy with which v and w can estimate each other's clock values, by simply choosing κ proportional to ε_e. The generalization to heterogeneous networks was described by Kuhn and Oshman in [64], who also introduced the description of the GCS algorithm in terms of the fast and slow conditions we use in our exposition (see Section 6.4). The GCS algorithm is also "almost" self-stabilizing, in the sense that it will re-establish its local skew bound from any state in $\mathcal{O}(\mathcal{S}/\mu)$ time, provided that a global skew bound of $\mathcal{S}$ is satisfied. As logical clocks must not run more than factor $1 + \mu$ faster than hardware clocks, this time bound is optimal so long as we impose this restriction on the maximum rate of logical clocks. Moreover, the stabilization property can be leveraged to allow for dynamic topologies. That is, edges may appear and disappear in a worst-case fashion, yet the algorithm maintains its skew bounds on all paths that consist only of edges that have been present for $\Omega(\mathcal{S}/\mu)$ time. Adding a mechanism to carefully "activate" the consideration of newly arriving edges level by level (i.e., for increasing values of s) in a well-timed fashion, the algorithm guarantees the claimed skew bounds with no further modification [63, 62]. In addition, choosing $\mu = \Theta(1)$ and using that the algorithm achieves $\mathcal{S} = \mathcal{O}(D)$, where D is the (weighted, dynamic) diameter of the graph, we see that the algorithm stabilizes new edges in $\mathcal{O}(D)$ time. Again, this bound is worst-case optimal [62]. The dynamic version of the algorithm in particular shows that crash failures can be tolerated, as repeatedly checking the liveness of nodes (which is implicit, as estimating clock values necessitates communication) enables the mapping of crash failures. Live nodes can then "delete" all incident links to the crashed node.

In this work, we provide the first (non-trivial) GCS algorithm resilient to non-benign faults. As it is based on the same algorithmic concept and a generic construction, we anticipate that all of the results mentioned in the preceding discussion can be carried over, even though we confine ourselves to the static setting in this paper.

7.2 Computational Model

The computational model used in this chapter largely follows the model described in Chapter 6. Nodes communicate by sending messages with minimum delay $d - U$ and maximum delay d.

Fault-tolerant GCS is not possible on arbitrary graphs. We augment a given graph to a graph that allows for fault-tolerant GCS. Given a graph $\mathcal{G}$ we extend it to graph G as described in the following.

Network. Let $\mathcal{G} = (\mathcal{C}, \mathcal{E})$ be an arbitrary graph. We consider a network $G = (V, E)$ constructed from $\mathcal{G}$ in the following way. We identify each $C \in \mathcal{C}$ with a set of k nodes $C = \{v_1, v_2, \ldots, v_k\}$. We refer to the sets $C \in \mathcal{C}$ as *clusters*. For distinct clusters $B, C \in \mathcal{C}$, the corresponding sets of nodes in V are disjoint, i.e., $V = \bigcup_{C \in \mathcal{C}} C$. The edge set E contains two different "types" of edges defined as follows:

cluster edges: for each $C \in \mathcal{C}$ and $v, w \in C$, we have $(v, w) \in E$;

intercluster edges: for each $(B, C) \in \mathcal{E}$, $v \in B$ and $w \in C$ we have $(v, w) \in E$.

Thus, each cluster $C \in \mathcal{C}$ is a clique in G, and for each pair of adjacent clusters $(B, C) \in \mathcal{E}$, E contains all possible edges between B and C. We refer to G as the *physical network*.

Remark. We assume that each vertex $v \in V$ knows the identities of its neighbors, as well as the identity of the cluster to which each neighbor belongs.

Logical Clock. While h_v determines the (unknown) rate of v's hardware clock, our algorithm controls its logical clock by adjusting the logical clock rate relative to the hardware clock. Specifically, the algorithm controls two parameters: $\delta_v(t) \in \mathbb{R}_{\geq 0}$, and $\gamma_v(t) \in \{0, 1\}$. The parameter $\delta_v(t)$ is adjusted by the intra-cluster synchronization algorithm in order to maintain synchronization within each cluster in accordance with the Lynch-Welch algorithm (see Section 7.3). The parameter $\gamma_v(t)$ indicates whether or not a node runs in "fast mode" in accordance with the inter-cluster algorithm simulating the GCS algorithm between clusters (see Section 7.4). The logical clock value $L_v(t)$, with offset L_v^0 at time 0, is computed to be

$$L_v(t) = \int_0^t (1 + \varphi \cdot \delta_v(\tau))(1 + \mu \cdot \gamma_v(\tau)) h_v(\tau)\, d\tau + L_v^0 \,. \tag{7.2}$$

The parameters φ and μ are constants whose values will be determined later on, where we already fix that $0 < \varphi < 1$ and $\mu > 0$. The parameters $\delta_v(t)$ and $\gamma_v(t)$ are controlled by two algorithms independently operating at each node.

Given Newtonian times $t < t'$, we define the *logical duration* or *logical length* of the interval $[t, t']$ for v to be

$$L_v(t') - L_v(t) = \int_t^{t'} (1 + \varphi \cdot \delta_v(\tau))(1 + \mu \cdot \gamma_v(\tau)) h_v(\tau)\, d\tau. \tag{7.3}$$

7.2.1 Faults

We assume that the network G contains a fixed subset $F \subseteq V$ of *faulty* processes. The nodes $v \in F$ are Byzantine. We assume that the number of faulty nodes *within each*

cluster is bounded by a parameter f, i.e., $\forall C \in \mathcal{C}\,|F \cap C| \leq f$, and we require that $k \geq 3f+1$, for the clusters of size k. Recall that Ineq. (7.1) relates this deterministic requirement to a setting in which nodes fail uniformly and independently at random with probability p; in this case, one can tolerate a value of p up to roughly $n^{-1/f}$. In contrast, an adversarial placement of faults would necessitate degrees larger than the total number of faults.

7.3 Cluster Algorithm

In this section, we only consider nodes and edges within a fixed cluster $C \in \mathcal{C}$. Within C, the logical clocks maintain synchronization by the variant of the Lynch-Welch algorithm [71] described in Section 6.3.

Algorithm ClusterSync adjusts $\delta_v(t)$ to manipulate the logical clock and hence the round length. We define the *nominal clock rate* of node v as

$$h_v^{\text{nom}}(t) = (1+\varphi)(1+\mu \cdot \gamma_v(t))h_v(t). \tag{7.4}$$

The *nominal length* of an interval $[t,t']$ is defined to be $\int_t^{t'} h_v^{\text{nom}}(\tau)\,d\tau$.

The following lemma shows that ClusterSync simulates an execution of the Lynch-Welch algorithm (cf. [57]), where the nominal clock in ClusterSync plays the role of the hardware clock in the Lynch-Welch algorithm.

Lemma 7.2. *Fix a non-faulty node $v \in C \setminus F$ and a round $r \in \mathbb{N}$. Then the nominal length of round r for v is $T(r) + \Delta_v(r)$, where $T(r) = \tau_1(r) + \tau_2(r) + \tau_3(r)$ is the logical length of round r. That is, $\int_{t_v(r)}^{t_v(r+1)} h_v^{\text{nom}}(\tau)\,d\tau = T(r) + \Delta_v(r)$.*

Lemma 7.2 shows that Algorithm 1 achieves the same effect at the end of the round as the variant of the Lynch-Welch algorithm given in [57]. In [57], L_v is discontinuously adjusted by $-\Delta_v(r)$ at the end of Phase 2 and increases at the nominal rate at all other times. The advantage of our formulation of Algorithm 1 is that $L_v(t)$ is continuous and increases at a rate that can be kept close to the nominal rate by choosing $\tau_3(r)$ sufficiently large. This is essential for the inter-cluster algorithm and its analysis, as the analysis of GCS assumes clock rates are bounded from above and below.

7.3.1 Cluster clocks and estimates

Here we define the notion of a "cluster clock" which is a function of the logical clocks of correct nodes in the cluster. In Section 7.4 we describe an algorithm such that the cluster clocks simulate an execution of the GCS algorithm from [62, 67]. Our main result then follows from the analysis of the GCS algorithm, and Corollary 7.5 below, which shows that logical clocks within each cluster approximate the cluster clock.

Definition 7.3. *Fix a cluster C. For each time t define $L_C^+(t)$ and $L_C^-(t)$ to be the maximum and minimum values (respectively) of logical clock values of non-faulty clocks in C at time t. That is, the maximum value $L_C^+(t) = \max\{L_v(t) \mid v \in C \setminus F\}$*

and the minimum value $L_C^-(t) = \min\{L_v(t) \mid v \in C \setminus F\}$*. We define* C*'s* cluster clock L_C *by the formula* $L_C(t) = (L_C^+(t) + L_C^-(t))/2$*.*

At several points, we will appeal to the following result about cluster clocks:

Observation 7.4. *Suppose that over some interval* $[t', t]$ *the logical clock of each* $v \in C \setminus F$ *increases at a rate at most* ϑ*. Then we have* $L_C(t) - L_C(t') \leq \vartheta \cdot (t - t')$*. Indeed, if each clock individually satisfies some upper (lower) bound on its rate, then in particular the rates of* $L_C^+(t)$ *and* $L_C^-(t)$ *satisfy the same bound, hence so does* $L_C(t)$*. Symmetrically, if all logical clock rates are at least* ϑ*, then we have that the logical duration* $L_C(t) - L_C(t') \geq \vartheta \cdot (t - t')$*.*

Suppose C is a cluster and w a node adjacent to C (i.e., $w \in B$ where $(B, C) \in \mathcal{E}$). Then w computes an estimate $\widetilde{L}_C^w(t)$ of $L_C(t)$ as follows. The node w listens to the pulses of nodes in C and simulates the ClusterSync algorithm, without sending pulses itself. Then w takes $\widetilde{L}_C^w(t)$ to be the logical clock value computed in its simulation of ClusterSync. Applying the analysis of ClusterSync (unchanged!) to w's estimate $\widetilde{L}_C^w(t)$, we obtain the following guarantee.

Corollary 7.5. *Let* C *be a cluster and* w *a node adjacent to* C*. Suppose* $\mathcal{L}$ *is as in* (6.5)*. Then for all* $v \in C \setminus F$ *and times* t *we have* $|\widetilde{L}_C^w(t) - L_v(t)| \leq \mathcal{L}$*. Thus for all* t*, we have* $|\widetilde{L}_C^w(t) - L_C(t)| \leq \mathcal{L}/2$*.*

7.3.2 Bounds for unanimous clusters

In this section, we state bounds on the amortized rates of cluster clocks when all correct nodes are running in fast or slow modes. We say that a cluster C is *unanimous at time* t if either (1) for all $v \in C \setminus F$, $\gamma_v(t) = 1$ or (2) for all $v \in C \setminus F$, $\gamma_v(t) = 0$. In the former case, we call C *(unanimously) fast*, and in the latter case, C is *(unanimously) slow*. For a round $r \in \mathbb{N}$, we say that C is (unanimously) fast (resp. slow) in round r if every $v \in C \setminus F$ is in fast mode (resp. slow mode) for all $t \in [t_v(r), t_v(r+1)]$.

In order to implement the GCS algorithm for cluster clocks, we must show that unanimously fast clusters can "catch up" to unanimously slow clusters. This is true for individual nominal clocks: if v is in fast mode and w is in slow mode, then $h_v^{\text{nom}}(t)/h_w^{\text{nom}}(t) \geq (1+\mu)/(1+\rho) = 1 + \Omega(\mu - \rho)$. So as long as $\mu \gg \rho$, a fast node can always catch up to a slow node.

The story for clusters is, unfortunately, more complicated. Suppose C is (unanimously) fast for an entire round r. Even though the individual *nominal* clocks in $C \setminus F$ all run at rates at least $(1+\varphi)(1+\mu)$, the amortized rate of L_C may be significantly slower because of the adjustment made to the logical clocks in the Lynch-Welch step. Specifically, this adjustment could be as large as $e_g^\infty \sim \mu T(r)$. Thus, the amortized rate of L_C over round r could be as small as $(1+\varphi)(1+\mu - e_g^\infty/T(r)) \sim (1+\varphi)$. Thus, the logical clock of a cluster in fast mode may increase *slower* than a cluster in slow mode!

To address this potential problem, observe that if a cluster is unanimous, then the nominal clocks for $v \in V$ satisfy $\zeta \leq h_v^{\text{nom}} \leq \zeta \cdot \vartheta_u$, where $\vartheta_u = 1 + \rho$ and $\zeta = 1 + \varphi$

or $(1+\varphi)(1+\mu)$ depending on whether the cluster is unanimously slow or fast. Thus, the nominal clock drift between nodes is at most ρ, rather than $(1+\rho)(1+\mu)-1$. This smaller hardware clock drift allows the cluster synchronization algorithm to converge to a smaller skew. In particular, ClusterSync achieves skews of size εe_g^∞ for an arbitrarily small $\varepsilon > 0$ for suitable choices of parameters, assuming that the cluster is unanimous for sufficiently many rounds. Thus, we ensure that the amortized rate of L_C when C is unanimously fast is at least $(1+\varphi)(1+7\mu/8)$. Similarly, we show that any cluster in slow mode increases at an amortized rate between $(1+\varphi)(1-\mu/8)$ and $(1+\varphi)(1+\mu/8)$, assuming it has been in slow mode for sufficiently long.

We denote the steady-state error for a unanimous execution (in which all nodes are unanimously fast or slow in all rounds) to be e_u^∞. We derive explicit bounds on e_u^∞ in the full version [19]. The following lemma shows that by choosing appropriate values of μ and φ, we can ensure that e_u^∞ is significantly smaller than e_g^∞. To make the dependence on ρ in all expressions explicit, we define parameters c_1 and c_2 and take

$$\varphi = \rho/c_1 \quad \text{and} \quad \mu = c_2 \cdot \rho\,.$$

Lemma 7.6. *Let $\varphi = \rho/c_1$ and $\mu = c_2 \cdot \rho$, and define τ_1, τ_2, and τ_3 by* (6.4)*. Suppose logical clocks are initialized so as to ensure $\|\mathbf{p}(0)\| \leq e_g^\infty$, and parameters are chosen such that $\alpha_g < 1$ in Equation* (6.3)*. Then there exist constants c_1, c_2, and k with $k \geq 1$ such that for all sufficiently small $\rho > 0$ the following hold:*

(1) If C is unanimously fast for rounds $r-k, r-k+1, \ldots, r$ then for all $v \in C \setminus F$ we have

$$(1+\varphi)\left(1+\frac{7}{8}\mu\right) \leq \frac{L_v(t_v(r+1)) - L_v(t_v(r))}{t_v(r+1) - t_v(r)}\,.$$

(2) If C is unanimously slow for rounds $r-k, r-k+1, \ldots, r$, then for all $v \in C \setminus F$ we have

$$(1+\varphi)\left(1-\frac{1}{8}\mu\right) \leq \frac{L_v(t_v(r+1)) - L_v(t_v(r))}{t_v(r+1) - t_v(r)} \leq (1+\varphi)\left(1+\frac{1}{8}\mu\right).$$

If additionally C is unanimously fast (resp. slow) in round $r+1$, then the cluster clock $L_C(t)$ satisfies conclusion 1 (resp. 2) above.

7.4 Inter-cluster Algorithm

In this section, we describe an algorithm that synchronizes clocks between adjacent clusters. The algorithm simulates an execution of the GCS algorithm [62, 67], where each cluster plays the role of a single node in the GCS algorithm. Specifically, our algorithm ensures that the cluster clocks L_C (cf. Definition 7.3) faithfully simulate an execution of the GCS algorithm, in a sense made precise below. While L_C is only defined implicitly from the logical clocks of individual nodes within a cluster, Lemma 6.3 and Corollary 7.5 ensure that every node maintains good estimates of

its own and neighboring cluster clocks. In our algorithm, each node individually simulates the GCS algorithm. We then apply Lemma 7.6 to argue that when the behavior of nodes within a cluster is unanimous, the corresponding cluster clocks behave as required by the GCS algorithm.

7.4.1 Fast and slow conditions and triggers

We know fast and slow conditions already from Chapter 6. For the remainder of this chapter, we reformulate them for clusters. Let κ be a parameter (to be chosen later), and let C be a cluster. We denote the set of neighboring clusters of C by N_C. The following definitions give conditions under which the cluster C should be in fast mode or slow mode in order to implement the GCS algorithm.

Definition 7.7. *We say that C satisfies the* fast condition *(FC) at time t if there exists $s \in \mathbb{N}$ such that the following conditions hold:*

$$\text{There exists } A \in N_C \text{ such that } L_A(t) - L_C(t) \geq 2s\kappa\,. \tag{FC-1}$$

$$\text{For all } B \in N_C\,.\, L_C(t) - L_B(t) \leq 2s\kappa\,. \tag{FC-2}$$

Definition 7.8. *We say that C satisfies the* slow condition *(SC) at time t if there exists $s \in \mathbb{N}$ such that the following two conditions hold:*

$$\text{There exists } A \in N_C \text{ such that } L_C(t) - L_A(t) \geq (2s-1)\kappa\,. \tag{SC-1}$$

$$\text{For all } B \in N_C\,.\, L_B(t) - L_C(t) \leq (2s-1)\kappa\,. \tag{SC-2}$$

In order to implement the GCS algorithm, we must guarantee that if a cluster satisfies the fast (resp. slow) condition, then the cluster is unanimously fast (resp. slow). Further, in order to maintain the guarantees on *amortized* rates of cluster clocks of Lemma 7.6, our implementation of the fast and slow conditions should guarantee that clusters remain unanimously fast or slow for several rounds before any round in which the respective (fast or slow) condition is satisfied. To this end, for each $v \in C$ and $B \in N_C$, v maintains an estimate $\widetilde{L}_B^v$ of L_B (see Corollary 7.5 and preceding discussion). In order to implement the fast and slow conditions, we define the following triggers. The parameter δ will be chosen later.

Definition 7.9. *We say that $v \in C$ satisfies the* fast trigger *(FT) at time t if there exists $s \in \mathbb{N}$ such that the following two conditions hold:*

$$\text{There exists } A \in N_C \text{ such that } \widetilde{L}_A^v(t) - L_v(t) \geq 2s\kappa - \delta\,. \tag{FT-1}$$

$$\text{For all } B \in N_C\,.\, L_v(t) - \widetilde{L}_B^v(t) \leq 2s\kappa + \delta\,. \tag{FT-2}$$

Definition 7.10. *We say that $v \in C$ satisfies the* slow trigger *(ST) at time t if there exists $s \in \mathbb{N}$ the following two conditions hold:*

$$\text{There exists } A \in N_C \text{ such that } L_v(t) - \widetilde{L}_A^v(t) \geq (2s-1)\kappa - \delta\,. \tag{ST-1}$$

$$\text{For all } B \in N_C\,.\, \widetilde{L}_B^v(t) - L_v(t) \leq (2s-1)\kappa + \delta\,. \tag{ST-2}$$

Algorithm 4 InterclusterSync(v, κ, δ, T)

```
1: for each round r ∈ ℕ do
2:     at time L_v(t_v(r)) do
3:         if v satisfies FT then
4:             γ_v ← 1
5:         if v satisfies ST then
6:             γ_v ← 0
```

The following lemma shows that for all $\delta < 2\kappa$ the triggers above cannot both be simultaneously satisfied. In particular, taking $\delta = 0$, the following lemma implies that the fast and slow conditions are also mutually exclusive.

Lemma 7.11 ([64]). *The conditions FT and ST are mutually exclusive. That is, if C satisfies FT, then C does not satisfy ST, and vice versa.*

With the fast and slow triggers defined, we describe the inter-cluster algorithm, InterclusterSync (Algorithm 4). InterclusterSync differs from other descriptions of the GCS algorithm [67, 64, 62] in that it can only switch from fast to slow mode, or vice versa, at predetermined discrete times. For the analysis of the GCS algorithm, we require that the fast (resp. slow) trigger implements the fast (resp. slow) condition in the sense that whenever the condition is satisfied, the corresponding trigger is also satisfied. For *clusters* we require more: Even if a cluster is unanimously fast or slow, we cannot immediately infer sufficiently tight bounds on the rate of L_C. Instead, we must apply Lemma 7.6. In particular, we must wait until $k = \mathcal{O}(1)$ unanimous rounds have elapsed until we can guarantee sufficiently tight bounds on the rate of L_C to apply the GCS algorithm analysis as a black box. The following definition describes sufficient conditions under which InterclusterSync guarantees that a node has been in fast (resp. slow) mode "sufficiently long" whenever the fast (resp. slow) condition is satisfied.

Definition 7.12. *Let $T : \mathbb{N} \to \mathbb{R}$ be a sequence of round lengths, C a cluster, and $v \in C$. For every $r \in \mathbb{N}$, let $t_v(r)$ denote the time at which v begins round r in an execution of the* ClusterSync *algorithm. Let k be a constant such that the conclusion of Lemma 7.6 is satisfied. For any time t, let $r_t = \max\{r \mid t_v(r) \leq t\}$. We say that an execution of* InterclusterSync *is* faithful *for v if the following conditions hold:*

(1) For all $t \in \mathbb{R}^+$ such that C satisfies FC at time t, v satisfies FT at all $t' \in [t_v(r_t - k - 1), t_v(r_t)]$.

(2) For all $t \in \mathbb{R}^+$ such that C satisfies SC at time t, v satisfies ST at all $t' \in [t_v(r_t - k - 1), t_v(r_t)]$.

We say that the execution is faithful *for C if it is faithful for every node $v \in C \setminus F$.*

Applying Definition 7.12 we obtain the following consequence of Lemma 7.6:

Corollary 7.13. *Suppose X is a faithful execution for C. Then for every $t \in \mathbb{R}$ the following holds. If C satisfies* FC *at time t then*

$$(1+\varphi)\left(1+\frac{7}{8}\mu\right) \leq \frac{L_v(t_v(r_t+1)) - L_v(t_v(r_t))}{t_v(r_t+1) - t_v(r_t)}.$$

If C satisfies SC *at time t then*

$$(1+\varphi)\left(1-\frac{1}{8}\mu\right) \leq \frac{L_v(t_v(r_t+1) - L_v(t_v(r_t))}{t_v(r_t+1) - t_v(r_t)} \leq (1+\varphi)\left(1+\frac{1}{8}\mu\right).$$

We will show that Corollary 7.13 is strong enough that we can apply the analysis of the GCS algorithm as a black box to bound skew between adjacent clusters. In the remainder of this section, we will give sufficient conditions under which every execution is guaranteed to be faithful. We fix k to be a constant such that the conclusion of Lemma 7.6 is satisfied.

Lemma 7.14. *Suppose the hypotheses of Lemma 6.3 are satisfied, and that k is sufficiently large that the conclusion of Lemma 7.6 holds. Let $\delta = (k+5)\mathcal{L}$ and $\kappa = 3\delta$. Then for every cluster $C \in \mathcal{C}$, every execution X is faithful for C.*

7.4.2 InterclusterSync **simulates GCS**

We now show that a faithful execution X of InterclusterSync on the (physical) network G *simulates* an execution $\overline{X}$ of the GCS algorithm on the network $\mathcal{G}$ in a sense made precise below. As a result, we can apply the analysis of the GCS algorithm to $\overline{X}$ to derive bounds on the local skew between adjacent cluster clocks in X. Before defining simulation formally, we adapt the axioms required by the GCS algorithm (cf. Definition 6.5) to clusters.

Definition 7.15. *Suppose $\mathcal{G} = (\mathcal{C}, \mathcal{E})$ is a network, and each $C \in \mathcal{C}$ computes a logical clock $L_C : \mathbb{R} \to \mathbb{R}$. We say that the logical clocks satisfy the* GCS axioms *if there exist constants $\rho, \mu > 0$ such that the following hold for all times $t \in \mathbb{R}$ and nodes $C \in \mathcal{C}$:*

$$1 \leq \frac{\mathrm{d}}{\mathrm{dt}} L_v(t) \leq (1+\rho)(1+\mu). \tag{A1}$$

If C satisfies SC *at time t, then* $\frac{\mathrm{d}}{\mathrm{dt}} L_v(t) \leq 1+\rho$. (A2)

If C satisfies FC *at time t, then* $1+\mu \leq \frac{\mathrm{d}}{\mathrm{dt}} L_v(t)$. (A3)

$$\mu/\rho > 1. \tag{A4}$$

Any execution of any algorithm satisfying these axioms is said to implement GCS*.*

In general, an execution of our algorithm does not satisfy the GCS axioms for the values of ρ and μ as specified in the previous sections. However, for suitable choices of these parameters, we can find different parameters $\overline{\rho}$, $\overline{\mu}$ for which our logical clocks do satisfy the GCS axioms. In the full version [19], we prove the following.

Lemma 7.16. *Suppose X is a faithful execution of* InterclusterSync *on G. Then the cluster clocks $\{L_C \mid C \in \mathcal{C}\}$ satisfy the GCS axioms for $\overline{\rho} = (1+\varphi)(1+(1/4)\mu) - 1$ and $\overline{\mu} = (1+\varphi)(1+(7/8)\mu) - 1$.*

7.4.3 Derivation of main result

So far, we have neglected the global skew $\mathcal{S}$. We can bound $\mathcal{S} = \mathcal{O}(D)$, where D is the network diameter as follows. Each node v maintains an (under)-estimate M_v of the maximum logical clock value of any node in the network. If L_v is not too far behind M_v, and v does not satisfy FT-1 for any s, then v defaults to slow mode, even if ST is not satisfied. This behavior ensures that if $L_v(t)$ is the maximal logical clock in the network, then v runs in slow mode. On the other hand, if L_v is significantly behind M_v ($M_v(t) - L_v(t) \geq c \cdot D$ for some suitable constant c) and v does not satisfy ST-1 for any s, then v defaults to fast mode. This behavior ensures that if the global skew is large, the slowest node runs in fast mode. Together, the two additional triggers described above ensure that the global skew satisfies $\mathcal{S} = \mathcal{O}(D)$.

We now have all the pieces in place to prove our main result, Theorem 7.1. We assume the parameters ρ, d, and U are given with ρ sufficiently small, and choose suitable values of μ and φ such that the conclusions of all required lemmas hold.

Proof of Theorem 7.1. Let $v, w \in V \setminus F$ with $\{v, w\} \in E$. We first consider the case where $v, w \in C$ for some cluster C. Then by Lemma 6.3, we have $|L_v(t) - L_w(t)| \leq \mathcal{L}$ for all t, where $\mathcal{L} = 6\beta_g/(1-\alpha_g) = \mathcal{O}(\rho \cdot d + U)$.

Now consider the case where $v \in B$ and $w \in C$ with $(B, C) \in \mathcal{E}$. By Lemma 7.16, InterclusterSync implements the GCS algorithm on $\mathcal{G}$ for cluster clocks L_C, with $\overline{\rho} = (1+\varphi)(1+(1/4)\mu) - 1$ and $\overline{\mu} = (1+\varphi)(1+(7/8)\mu) - 1$. Since $\varphi = \rho/c_1$ and $\mu = c_2 \cdot \rho$ for some absolute constants c_1 and c_2, there exists an absolute constant $c > 0$ such that

$$\frac{\overline{\mu}}{\overline{\rho}} = 1 + c + \mathcal{O}(\rho).$$

Thus $\overline{\mu}/\overline{\rho} \geq 1 + c/2$ for sufficiently small ρ. By Theorem 6.8, we therefore have

$$|L_B(t) - L_C(t)| = \mathcal{O}(\kappa \log_{\overline{\mu}/\overline{\rho}} \mathcal{S}) = \mathcal{O}(\mathcal{L} \log D).$$

We then bound the skew to obtain the desired conclusion.

$$\begin{aligned}
|L_v(t) - L_w(t)| &\leq |L_v(t) - L_B(t)| + |L_B(t) - L_C(t)| + |L_C(t) - L_w(t)| \\
&\leq \frac{1}{2}\mathcal{L} + \mathcal{O}(\mathcal{L} \log D) + \frac{1}{2}\mathcal{L} \\
&= \mathcal{O}(\mathcal{L} \log D)\,.
\end{aligned}$$

□

8 Implementation of Clock Synchronization Algorithms

In the subsequent chapters we present two implementations of the GCS algorithm. First, an implementation for a single link, i.e., two neighbors connected by an edge. Second, an implementation for arbitrary networks. Both implementations are different, e.g., only the link implementation includes data communication. However, there is a large overlap for both concepts.

In this chapter we summarize shared features of the subsequent chapters. That is: the related work on GALS systems and the hardware modules that are used as building blocks.

Outline. We discuss related work on GALS systems in Section 8.1. Subsequently we discuss three modules necessary for the implementations in Section 8.2.

8.1 Related Work

As discussed earlier, GALS systems overcome the scalability issues of centralized clocking. The microchip is divided into smaller regions that can be covered by a clock tree with small skew. Communication between clock regions happens asynchronously by handshakes, synchronizers and buffers. Hence, on a global scale GALS systems do not provide a synchronous abstraction and timing guarantees. Moreover, they add latency and cost to the communication paths.

A synchronizer reduces the probability of failure due to metastability. The probability of reading a meta-/unstable signal drops exponentially with the time allocated for resolution. A simple synchronizer circuit connects one or more flip-flops in series. With every flip-flop the resolution time increases by one clock cycle. Synchronizer circuits provide a large mean time between failures (MTBF).

According to [94], GALS systems can be classified by their clocking schemes: pausible clocked systems, asynchronous systems with uncorrelated clocks, and loosely synchronous systems, with (partially) synchronized clocks. We shortly review communication in these three approaches.

Pausible Clocking. Pausible Clocking overcomes synchronization issues by halting the clock until metastability is resolved [80]; e.g., the design in [81] guarantees no glitches on stopping and starting. This requires that the clock cannot be started again before metastability has been resolved. Metastability inside the control loop may lead to an arbitrary delay of the final pulse on stopping. This approach has no guarantees on progress of the system as resolution of metastability can have an

arbitrary delay in our worst case model. However, because metastability has been resolved when continuing the pausible clocking provides an infinite MTBF.

Uncorrelated Clocks. Communication between uncorrelated clock frequencies and phases is traditionally done by combining classical two-flop synchronizers with buffers and flow-control circuitry. A downside of these approaches is that the latency and the throughput are determined by the handshake cycle that has to include (at least) two synchronizer cycles at both sides. Clearly, this approach has a non-zero upset probability and thus finite MTBF. In [23], a mixed-clock first-in first-out pipeline (FIFO) with flow control logic is proposed. Instead of classical handshaking, synchronized full/empty and almost full/empty signals are used. The throughput is one data item per clock cycle until the almost full signal is raised; afterwards, the true full signal has to be considered, at the cost of increased latency and lower throughput. The approach has finite MTBF.

In [25], a ripple FIFO solution with almost full/empty signals is proposed. The approach requires slow sender/receiver speeds compared to data propagation within the ripple FIFO. Moreover, full/empty flags have to be synchronized, which leads to increased latency and finite MTBF. In [31], a locally delayed latching (LDL) approach is proposed: conflicting read/write operations are delayed by an asynchronous controller with a mutual exclusion (MUTEX) element. Controller latency is in the order of 20 gate delays, and the minimum feasible clock cycle is no less than 69 gate delays, accounting for sufficient time for the MUTEX to stabilize. Gradual synchronization [53] allows fine-grained interweaving of synchronization and computation, also shifting conflicting ripple FIFO requests by MUTEX elements at each stage. Like synchronizer chains, this approach has finite MTBF that can be increased at the cost of higher latency.

Dally and Tell [30] propose a scheme in which the MTBF can be made arbitrarily large without increasing latency. They use synchronizers to continually determine phase offsets between sender and receiver clocks only. A drawback is that the frequency and phase measurement circuits require accurate phase tracking (64 bit in their implementation) and can account for slow phase drifts only.

Loosely Synchronous Systems. In contrast to the approaches above, synchronizing clocks allows obtaining worst-case guarantees on latency and throughput together with provable absence of metastable upsets. Our approaches also fall into this class. The closest work to our approach in Chapter 9 presumably is proposed in [83]. By using the Distributed Algorithms for Robust Tick Synchronization (DARTS) clock generation mechanism [44], a buffer size of 9 and latency of 9 clock cycles was achieved for a receiver-sender clock shift of 4 ticks at around 25 MHz in an FPGA. While these numbers clearly can be improved in ASIC designs, DARTS inherently is slower than our approach.

Teehan et al. [94] present the term plesiochronous designs, a subclass of loosely synchronous GALS design style. Here, clock regions operate at the same nominal

frequency but may have a slight frequency mismatch. The mismatch may lead to a drifting phase, which in turns leads to an unbounded skew. We also call the system in Chapter 10 a plesiochronous system, although there is a slight mismatch in terminology. In our system clocks can speed up by a small factor, but essentially they run at the same nominal frequency. We allow for a slight frequency drift of oscillators, which may always happen due to manufacturing. In contrast to [94], we bound the skew by measuring the phase difference and adjusting the speed-up. The phase difference is bounded well below one clock cycle, such that communication without buffers and synchronizers is possible.

Phase Locked Loops. Our control module in Chapter 9 has some similarities to a phase locked loop (PLL) with an all-digital phase detector; see e.g., [1, 24] for all-digital PLLs designs. We briefly summarize commonalities and differences in the following.

Classical PLLs lock a slave clock to a typically more stable master clock. In Chapter 9 we do not distinguish between a slave and a master, but our controller treats both receiver and sender clocks equally; one might think of this as a peer-to-peer PLL. The reason is that our goal is not to stabilize the absolute frequency of a poor clock by ensuring a bounded phase offset to a more stable master clock, but rather to bound the phase offset between a sender and receiver clock of similar quality. Additionally, we provide lower and upper bounds on the frequency of the clocks which are close to the frequency bounds free-running oscillators of the same quality have. For example, this is useful when communicating with the environment.

The initial stage of a classical PLL is a phase frequency detector (PFD), which measures the phase difference between the master and slave clock signals. Designs range from conventional PFDs, which measure negative and positive phase offsets on separate binary output signals by producing pulses whose width is the negative/positive phase offset, to more advanced setups [82]. Phase differences are then either forwarded to charge pumps (analog PLLs) [4] or converted to digital counter offsets (digital PLLs). For the latter, an unstable phase difference poses a risk for increased power consumption and likelihood of metastable upsets; see [1], where a filter on phase difference signals for a low-power digital PLL is proposed.

In Chapter 9, there is a (digital) unary-encoded up/down-counter at the heart of the controller, allowing to measure the phase difference between both clocks. Note that since our goal is not to lock to a highly stable oscillator, our design is much simpler: our circuit only determines whether the actual phase offset is larger or smaller than the desired phase offset. It is also worth noting that, while our oscillators are analog components, our circuit relies on the ability to switch between fast and slow oscillator mode only. This binary decision may become meta-/unstable frequently. In stark contrast to a classical digital PLL with a binary counter, this does not pose a problem for our design. We ensure that the potentially metastable output signal of our controller is only used to control the oscillator. The oscillator frequency is required to remain in the range spanned by the frequencies possible under stable

operation (slow and fast mode) in presence of a metastable, or in general, unstable signal. This is the case for starved inverter ring oscillators.
The use of local clocks in our design has a further advantage over locking to a centralized clock that is assumed to provide a highly stable frequency reference. In our system the sender and receiver are not impaired by the failure of the respective other's clock. While correct communication between the two nodes inherently requires both oscillators to work correctly, our design guarantees that if one of the oscillators fails, the respective other keeps running within the same frequency bounds. Potential top-level error-detection based on the (non-)communicated data then provides adequate application-specific reaction to such scenarios.

8.2 Hardware Modules

In this section we describe hardware modules that are utilized in the hardware implementations described in Chapters 9 and 10. We identify three modules in the algorithm that are required for a hardware implementation: a tunable oscillator, an offset measurement, and a control module. In the following pages we describe each module, its specification and constraints to its hardware realization.

The modules have a cyclic dependency/connection on each other. The output of one module is connected to the input of the next module. The measurement modules captures offset estimates $\widehat{O}_w(t)$ of tunable oscillators, the controller module checks the fast trigger FT and forwards a mode signal to the tunable oscillator.

The measurement module captures the phase difference of two different clock regions. Any synchronous circuit that operates on clock domain crossings of uncorrelated clocks cannot avoid meta-/unstable signals. Hence, the module cannot avoid corrupted measurements. In turns it may produce a meta-/unstable output that is forwarded to the tunable oscillator. We discuss in Section 8.2.1 that the tunable oscillator, when chosen carefully, can cope with meta-/unstable inputs. In general, we can say that meta-/instability arises in the offset measurement, it propagates through the control module and is handled in the oscillator. Hence, we require the control module to be hazard-free, i.e., we require it to only propagate meta-/unstable signals if the output is undetermined.

Implementations of the single link and the network synchronization differ also in the communication structure. In consequence they have partly different requirements on the modules. Constraints and solutions specific to each implementation are discussed further in the respective chapter. We now discuss confirming constraints in the remainder of this chapter.

8.2.1 Tunable Oscillator

Each node maintains a logical clock on top of a hardware clock. The hardware clock is prone to uncertainty that is modeled by drift ρ. The logical clock can be adjusted by factor μ to catch up to neighbors that are known to be ahead.

In the implementations we derive the logical clock from a tunable oscillator. Each node is associated with its own oscillator that can be tuned in its frequency. In some sense the oscillator combines both the hardware and the logical clock, it is prone to drift but also adjustable in its frequency.

Remark. An oscillator alone does not make a clock in the sense that is described in Chapter 6. It is a simple pulse giving device that has no sense of how much time has passed in total. In fact no hardware can implement a clock as described in Chapter 6, as no hardware can count indefinitely, eventually it will overflow. A solution, that comes close, is to add a modulo counter. The counter counts the rising transitions of the oscillator. In Chapter 9 we see this approach, where the modulo counter can be as small as modulo 2. In Chapter 10 we see that an oscillator alone suffices for the application we have in mind.

There are two modes of the oscillator, fast and slow mode. The tunable oscillator has one input and one output port. The input port is called md, it is a binary input that controls the mode of the oscillator. The output, called clk, of the oscillator is its clock signal that pulses either at fast or slow frequency if md is stable for at least the time the oscillator requires to respond. For properly chosen response time of the tunable oscillator $T_{\mathrm{osc}} \geq 0$ and properly chosen initialization, we require the following conditions.

We assume a small initial skew of the system. The oscillators are started roughly at the same time, the global skew is bounded by $c \cdot \kappa$.

$$\mathcal{G}(0) \leq c \cdot \kappa \,. \tag{C1}$$

we discuss a possible initialization scheme in Section 6.1

Denote by $\dot{\mathrm{clk}}$ the rate of the signal clk. If mode signal of node v is constantly 0 for time T_{osc}, the oscillator with output clk_v is in *slow mode* at time t:

$$\forall t' \in [t - T_{\mathrm{osc}}, t].\ \mathrm{md}_v(t') = 0 \Rightarrow \dot{\mathrm{clk}}_v(t) \in [1, 1+\rho] \,. \tag{C2}$$

If a mode signal is constantly 1 for time T_{osc}, the respective oscillator is in *fast mode* at time t:

$$\forall t' \in [t - T_{\mathrm{osc}}, t].\ \mathrm{md}_v(t') = 1 \Rightarrow \dot{\mathrm{clk}}_v(t) \in [1+\mu, (1+\mu)(1+\rho)] \,. \tag{C3}$$

If a mode signal is neither constantly 0 nor constantly 1 for time T_{osc}, then the respective oscillator is *unlocked* at time t:

$$\exists t', t'' \in [t - T_{\mathrm{osc}}, t].\ \mathrm{md}_v(t') \neq \mathrm{md}_v(t'') \Rightarrow \dot{\mathrm{clk}}_v(t) \in [1, (1+\mu)(1+\rho)] \,. \tag{C4}$$

The constraint that clocks in slow mode are never faster than clocks in fast mode (cf. Section 6.1) carries over to oscillators:

$$\begin{aligned} 1+\rho &< 1+\mu \\ \Leftrightarrow \rho &< \mu \,. \end{aligned} \tag{C5}$$

cf. Condition (6.6)

Our requirements on the oscillator are fairly weak, making it easy to implement: Only if the control signal is stable for T_{osc} time, the oscillator needs to guarantee the

respective frequency. At any other time, it is not locked to a fixed mode and may run at any frequency between the slowest and fastest possible. Especially the unlocked mode may be entered when the control signal is meta-/unstable or transitioned recently. I.e. an oscillator that satisfies (C4) is able to cope with meta-/unstable inputs in the sense that it produces stable outputs. Meta-/Unstable signals only affect the frequency.

It is an essential requirement of the algorithm that the skew between two nodes cannot increase if the algorithm tries to reduce that skew. Condition (C5) is a minimal requirement ensuring that the phase offset between the two clocks cannot increase further when a clock in fast mode is chasing a clock in slow mode.

Remark. A tunable oscillator that satisfies (C4) is not pausible.

8.2.2 Control Module

The control module checks whether the fast trigger is satisfied and controls the tunable oscillator accordingly. The control module outputs md. The input to the control module is given by the offset measurement. As implementations of Chapter 9 and Chapter 10 have different approaches to the offset measurement we specify the inputs in each section. The requirements to the control module are given by the GCS algorithm. They can be formulated for both implementations.

We denote the response time of the control module by T_{ctr}, the control module must satisfy the following three constraints. If OffsetGCS continuously maps the switch $\gamma(t)$ to 0 for time T_{ctr}, then the output of the control module is 0 at time t:

$$\forall t' \in [t - T_{\text{ctr}}, t].\, \gamma_v(t') = 0 \Rightarrow \text{md}_v(t) = 0\,. \tag{L1}$$

If OffsetGCS continuously maps the switch $\gamma(t)$ to 1 for time T_{ctr}, then the output of the control module is 1 at time t:

$$\forall t' \in [t - T_{\text{ctr}}, t].\, \gamma_v(t') = 1 \Rightarrow \text{md}_v(t) = 1\,. \tag{L2}$$

If $\gamma(t)$ is noncontinuous for time T_{ctr}, then the output of the control module is unconstrained:

$$\exists t', t'' \in [t - T_{\text{ctr}}, t].\, \gamma_v(t') \neq \gamma_v(t'') \Rightarrow \text{md}_v(t) \in \{0, \mathfrak{u}, 1\}\,. \tag{L3}$$

Since the result of the offset measurement may be unstable, the control module might be faced with unstable inputs. Constraints (L1) to (L3) require that the output is stable if the inputs determine the output. In other words the control module must be hazard-free.

In Chapter 9 we present a single control module that controls oscillators of both nodes in the system. Formally, we can regard the control module as two merged control modules, one for each oscillator. In contrast, Chapter 10 shows a system where each node has its own control module that steers only one oscillator.

8.2.3 Offset Measurement

In general, the offset measurement estimates the difference between two local clock sources and sends it to the control module. The phase offset of two nodes in our system is bounded by $\mathcal{L}$. Due to its nature, the offset measurement module may output meta-/unstable signals. As discussed earlier, we refrain from using synchronizers, because this comes at the cost of worse clock synchronization.

The delay of the offset measurement, T_{meas}, adds to the precision of the estimates, as clocks may drift during that time. Hence, any time spent on synchronizers comes at the burden of inaccurate measurements.

Measuring offsets to neighboring nodes is solved inherently different for implementations in Chapter 9 and Chapter 10. Hence, any constraints on the measurement module are discussed in the respective chapter.

The offset measurement in Chapter 9 utilizes a data buffer, which readily gives information about the offset by its fill level. In Chapter 10 we discuss a traditional way to measure offset between two oscillator signals. Later on we show an advanced approach that yields better results.

Single Link Synchronization 9

This chapter presents parts of the results published in Transactions on Circuits and Systems [12]. We present a producer-consumer link for a single sender and receiver scenario. Sender and receiver both have their own (independent) clock domain, hence any communication bears the risk of metastable signals. Our objective is to synchronize the clock domains while improving the latency and throughput of the communication link.

Outline. An introduction is given in Section 9.1.1. We start with presenting the problem of communication in a system of two nodes with controllable oscillators in Section 9.2. We then break the system down into modules, formally specifying their requirements. Section 9.3 discusses gate-level implementations of the modules, together with proofs that the implementations satisfy the formal requirements. In Section 9.4, we present simulations of our implementation at the gate level and transistor level. The simulation results are consistent with our formally proven results and allow us to obtain detailed performance metrics. This chapter is followed by Chapter 10 which can be seen as an extension of the work that is presented here.

9.1 Introduction and Related Work

9.1.1 Introduction

Links that enable communication between different clock domains are an important ingredient in every GALS system [94]. This communication is performed in a producer-consumer manner: In one clock domain, the producer pushes messages to the link, while in the other clock domain the consumer pulls messages from the other side. Inherently, link implementations are susceptible to failures induced by metastable upsets; Even if such errors can be handled, they negatively impact the performance of the link.

Previous digital controller designs resort to different methods to deal with metastability (cf. Section 8.1): clock-masking [25], clock-pausing [80, 81], or adding synchronizers (while sacrificing latency) to maintain a realistic (yet finite) MTBF of the link [23, 25, 30, 31, 53]. This requires almost-full flags [25], long handshake latencies that increase the dead time and affect the latency and throughput, additional slack in a controller cycle accounting for metastability resolution time in the controller's flip-flops, or MUTEX elements [31, 53]. The downside of these approaches is that synchronized fill-level flags are inherently stale by the time they affect the system.

At the heart of the problems faced in these controllers lies the impossibility of solving discrete decision problems, e.g., writing to a cell at a certain clock tick or skipping

		this work	[83]	[30]
Performance	Latency	1 ns	375 ns	1.3 ns
	Throughput	1/2 ns	1/41 ns	1/1.3 ns
	MTBF	∞	∞	Finite
Overhead	N	2	9	2
	# Gates	8	> 100	> 100
	# Flip-Flops	4	> 50	> 100
	Oscillator Type	tune	distributed	quartz

Table 9.1: Performance and hardware overhead (buffer size N, gates, flip-flops, oscillator type) of the proposed controller with a tunable 2 to 2.3 GHz oscillator, [30], and [83].

a clock cycle, under continuous inputs (i.e., arbitrary phase shifts between producer and consumer clocks) within bounded time [74]. One way out of this impossibility is to resort to *end-to-end analog designs*, e.g., by letting an analog controller apply continuous phase shifts by (slightly) tuning the producer and/or consumer oscillator. This comes at the burden of a fully-fledged analog design.

An interesting alternative was proposed in [88], where the authors advocate the use of asynchronous controllers, sensing and controlling analog processes. With this approach, analog components are required at the controller interfaces only, and the controller itself is implemented by a digital asynchronous circuit. For certain classes of controllers, this approach allows to completely circumvent metastable upsets within the controller circuit, essentially by allowing for the occurrence of (digital) controller outputs within a continuous time range, rather than at discrete clock ticks only.

Table 9.1 shows a comparison of our controller with the most closely related works, [83] and [30] (cf. Section 9.4 for details).

Contribution. We propose a fundamentally different approach, exemplifying it at the hand of highly efficient link controllers: like [88], we replace large parts of a (conceptually) analog controller by standard digital circuitry. However, we do *not* resort to asynchronous circuits. Instead, we allow unstable/metastable signal values within our circuit. Clearly, care must be taken that such values do not affect the whole controller logic, leading to unconstrained control outputs.

Specifically, we propose a digital controller that drives tunable ring oscillators as presented in [45] at the sender and receiver side and prove its correctness. The controller is small in size, has low control latency, and allows for small link buffers. We show that this guarantees high throughput and low latency communication. Most notably, while the controller may become metastable, we ensure that metastability is contained within the controller, and does not lead to metastable upsets, corruption, or

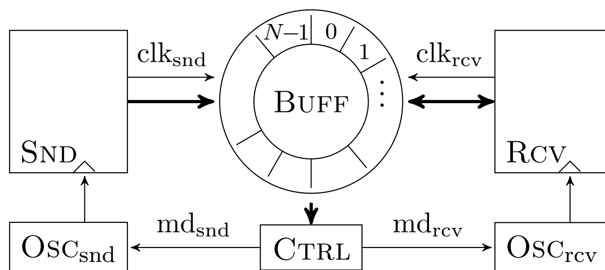

Figure 9.1: Link with Digital Controller

drops of communicated data words in the ring buffer between the sender and receiver. We complement our provable system guarantees with simulations (see Section 9.4).

In the next chapter (Chapter 10) we present a synchronization scheme for more than two nodes, i.e., a network of nodes. Here we discuss the groundwork at hand of a single link. Some components and concepts, e.g., oscillators, are equal or adjusted versions of the ones presented here. A further difference is that this work also includes communication of the nodes while the network synchronization does not.

9.2 System Specification

We specify the system requirements and functionality next. The system topology is depicted in Figure 9.1. We identify three main components:

tunable oscillators $\mathrm{OSC}_{\mathrm{snd}}$ and $\mathrm{OSC}_{\mathrm{rcv}}$, (i)

a (ring) buffer BUFF, and (ii)

a buffer controller CTRL. (iii)

The link enables communication between two parties, a sender SND and a receiver RCV, that interact with the link via prescribed interfaces, discussed later on.

The sender writes data to a ring buffer of even size $N > 0$, which is read by the receiver. Cells are numbered from 0 to $N - 1$. Read and write access is clocked: following transitions of $\mathrm{clk}_{\mathrm{snd}}$, the sender writes to the ring buffer. The register address is specified by the current value of its address pointer, which it subsequently increments (modulo N); likewise, following transitions of $\mathrm{clk}_{\mathrm{rcv}}$, the receiver reads from its current address and subsequently increments its pointer.

We remark that our design can easily be altered for bidirectional communication. Each party needs to perform a read/write sequence instead of just a read (respectively write) operation when it is accessing a buffer cell; The only effect is that the respective higher access time needs to be respected in the timing constraints on the system. For ease of presentation, we stick to the asymmetric setting in the remainder of this chapter.

9.2.1 Local Clocks

The sender and receiver clocks $\mathrm{clk}_{\mathrm{snd}}$ and $\mathrm{clk}_{\mathrm{rcv}}$ are derived from clock sources $\mathrm{Osc}_{\mathrm{snd}}$ and $\mathrm{Osc}_{\mathrm{rcv}}$, respectively. We require that these clock sources (or oscillators) are *tunable in frequency*. The frequency is controlled by the *mode signals* $\mathrm{md}_{\mathrm{snd}}$ and $\mathrm{md}_{\mathrm{rcv}}$.

Denote by $Clk(t) \in \mathbb{Z}$ a discrete clock value at Newtonian time $t \in \mathbb{R}_{\geq 0}$. This discrete clock is derived from a continuous clock $\mathrm{clk}(t) \in \mathbb{R}$ as $Clk(t) := \lfloor \mathrm{clk}(t) \rfloor$, with current frequency $\dot{\mathrm{clk}}(t)$. Let $Clk_s(t)$, $Clk_r(t)$ be the discrete clock values of sender and receiver at Newtonian time t, and $\mathrm{clk}_s(t)$, $\mathrm{clk}_r(t)$ their continuous clocks.

The local clock is realized by a tunable oscillator and a modulo-counter. It follows the constraints of the tunable oscillator in Section 8.2. In the remainder of this chapter, we denote slow and fast rates of the oscillator by s and f and by $-$ and $+$ the drift of the hardware clock, such that

$$s^- := 1,\;\; s^+ := 1+\rho,\;\; f^- := 1+\mu,\;\; f^+ := (1+\rho)(1+\mu)\,.$$

Remark. For $\delta = 1$, Condition (C1) is a fairly weak constraint. If the sender and receiver each access one element of the ring buffer per clock cycle, the condition requires that both oscillators are started within one clock cycle of each other. However, smaller values of δ may reduce the minimum feasible ring size by 2 in some cases.

9.2.2 Buffer Access Specification

Next, we specify buffer access in an abstract model with few parameters. We assume that access to a buffer cell starts when the respective clock modulo N (possibly with a fixed offset) equals the buffer index. A computational cycle is defined by the local time between accessing consecutive buffers.

Intuitively, a buffer cell is *valid* (i.e., ready to be read) if it contains stable, logical data and is currently not written. A buffer cell is *invalid* (i.e., ready to be written) if it is not valid and currently not read. Formally, we define the receiver's (discrete) address pointer as

$$P_r(t) := \lfloor p_r(t) \rfloor \bmod N = Clk_r(t) \bmod N\,, \tag{B1}$$

where the receiver's (continuous) address pointer is $p_r(t) := \mathrm{clk}_r(t)$. That is, the receiver starts to access cell ℓ at each time t when $P_r(t) = p_r(t) \bmod N = \ell$.

The sender pointer has a (nominal) offset of half the ring size relative to the receiver pointer. We define the sender's address pointer to be

$$P_s(t) := \lfloor p_s(t) \rfloor \bmod N\,, \tag{B2}$$

where $p_s(t) := \mathrm{clk}_s(t) + N/2$. In the following, we will simply drop the "starts to" and say that the receiver (sender) *accesses cell ℓ at time t* if $p_r(t) \bmod N = \ell$ ($p_s(t) \bmod N = \ell$).

Read and write operations take non-zero time. We account for setup/hold times and latency by parameters τ_{s} and τ_{r}, which denote the maximum duration of write and read operations. Concretely,

$$\begin{aligned}&\text{if the sender accesses a cell at time } t,\\&\text{the receiver must not do so during } [t, t+\tau_{\mathrm{s}}),\\&\text{and if the receiver accesses a cell at time } t,\\&\text{the sender must not do so during } [t, t+\tau_{\mathrm{r}}).\end{aligned} \tag{B3}$$

On initialization, cells $0 \leq \ell < N/2$ are valid, while cells $N/2 \leq \ell < N$ are invalid.

$$\begin{aligned}&\text{If the sender accesses an invalid cell at time } t,\\&\text{the cell \textit{becomes valid} at time } t+\tau_{\mathrm{s}}.\\&\text{If the reader accesses a valid cell at time } t,\\&\text{it \textit{becomes invalid} at time } t+\tau_{\mathrm{r}}.\end{aligned} \tag{B4}$$

This inductively defines for each cell and each time $t \geq 0$ whether it is valid or invalid.

Note that these definitions are crafted in such a way that if the sender accesses only invalid cells and the reader accesses only valid cells, we have mutual exclusion of read and write operations and for each individual cell, reads and writes alternate. This is the intended mode of operation, which we will formalize in Section 9.2.6.

Remark. This approach is a normalization of the time axis so that one computational cycle takes 1 unit of local time as measured by the sender or receiver oscillator, respectively. Note that this will typically not be 1 unit of Newtonian time, as oscillator speeds may vary.

9.2.3 Metastability

To minimize the dead time of the control loop regulating the clock speeds, we do not make use of synchronizers. Forgoing their use can result in meta-/unstable signals. At any point in time, a signal has a value in $\{0, \mathfrak{u}, 1\}$, where $\mathfrak{u}$ means that a signal is potentially unstable or in transition. In particular, a flip-flop latching when its input is $\mathfrak{u}$ will "store" an $\mathfrak{u}$ until a stable input is latched again. Note that an output signal may also be unstable due to a transitioning signal, e.g. after latching a new value different from the previously stored one.

9.2.4 Link Controller Interface Specification

The mode signals themselves are generated by the controller CTRL. Controller decisions are based on full/empty flags of the ring-buffer cells, which we describe shortly. We stress that, inherently, the controller acts at the border of two clock domains. Any digital implementation (including ours) is thus susceptible to unstable upsets. Accordingly, the voltage levels of $\mathrm{md}_{\mathrm{snd}}$ and $\mathrm{md}_{\mathrm{rcv}}$ may become meta-/unstable, denoted by $\mathfrak{u}$ (between logical 0 and 1). In order to minimize delay, we do *not* pipe them through a synchronizer chain before making use of them.

Figure 9.2: Ring buffer access at time t. The sender currently accesses cell 4. Hence, its full/empty flag is $\mathfrak{u}$. The receiver has just finished accessing cell 1. Thus, its full/empty flag is 0. In executions, we mark (potential) instability in red, cf. Figure 9.8.

Let T_{ctr} denote the maximum end-to-end delay of the controller circuit, i.e., between its input (the full/empty flags) and its output (md_{snd} and md_{rcv}). The specification of the link controller's interface follows the specification of the control module in Section 8.2.

9.2.5 Full/Empty Flags

With each buffer cell ℓ, we associate a full/empty flag F_ℓ. It is specified as

$F_\ell(t) = 1$ if the cell is valid at time t and it either has not been accessed yet or the most recent access to it was by the sender; (F1)

$F_\ell(t) = 0$ if the cell is invalid at time t and it either has not been accessed yet or the most recent access to it was by the receiver; (F2)

if neither applies at time t, then $F_\ell(t) \in \{0, \mathfrak{u}, 1\}$. (F3)

In other words, we allow for the possibility that $F_\ell(t) = \mathfrak{u}$ at any point in time during read and write operations.

Figure 9.2 depicts the state of the above-described cell pointers at time t. Observe that all cells between the sender and the receiver are full and thus their full/empty flags equal to 1, those between the receiver and the sender are empty with full/empty flags equal to 0, and the flags of those currently accessed are $\mathfrak{u}$.

9.2.6 System Correctness

Expressing the correct order of and separation in time between cell accesses, we can now succinctly state what the correct operation of the link architecture means.

Definition 9.1. *A link is* correct *if the following holds in any execution adhering to our model.*

No underrun: the receiver accesses only valid cells. (P1)

No overflow: the sender accesses only invalid cells. (P2)

Definition 9.2. *Controller* CTRL *is* correct *if it computes the signals* md_{snd} *and* md_{rcv} *out of the inputs* F_ℓ *so that the link is correct.*

The goal is now to design a (simple) controller that is correct even if the ring size N is small: this minimizes both the size of the buffer and its latency.

9.3 Continuous Threshold Controller

Our control algorithm CONTTH($\mathcal{T}$) is specified in Algorithm 5. It is parametrized by $\mathcal{T} \in \mathbb{R}_{>0}$. In the remainder of this section, we explain the intuition behind the approach.

For the purpose of exposition, denote by $\mathit{fill}(t) := p_s(t) - p_r(t)$, such that $\mathit{fill}(t) = N/2 + \mathrm{clk}_s(t) - \mathrm{clk}_r(t)$, the fill level of the buffer. Recall that one of our design goals is to have a simple digital controller. The most straightforward choice for such a control algorithm is presumably the threshold controller: If the fill level of the ring buffer is larger than $N/2$, the sender is forced to slow mode and the receiver is forced to fast mode. If the fill level is less than $N/2$, the sender and receiver are forced into fast and slow mode, respectively.

However, as the various involved circuit components incur non-zero delays, we cannot expect instantaneous (and thus also not exact) information on the fill level. Also, changing the oscillators' speeds takes non-zero time, so we cannot hope for an immediate response to a small/large fill level. Algorithm 5 takes this into account by introducing *two thresholds*.

Example. Figure 9.3 shows an execution where the controller CTRL runs the algorithm. The fill-level increases until it hits $\frac{N}{2} + \mathcal{T}$, which makes the $\mathrm{md}_{\mathrm{rcv}}$ signal drive 1 after T_{ctr} time. After another T_{osc} time, the receiver and sender clocks are required to run in fast and slow mode, respectively (cf. Section 9.2). Note that the second phase during which the threshold $\frac{N}{2} + \mathcal{T}$ is crossed is too short for CTRL and the oscillators to react with certainty.

Algorithm 5 Controller ContTh($\mathcal{T}$)

at each time t **do**
  $\text{md}_{\text{rcv}}(t) \leftarrow$ choose arbitrarily in $\{0, \mathtt{M}, 1\}$
  $\text{md}_{\text{snd}}(t) \leftarrow$ choose arbitrarily in $\{0, \mathtt{M}, 1\}$
  if $\text{clk}_s(t) - \text{clk}_r(t) \geq \mathcal{T}$ **then**
    $\text{md}_{\text{rcv}}(t) \leftarrow 1$
    $\text{md}_{\text{snd}}(t) \leftarrow 0$
  if $\text{clk}_r(t) - \text{clk}_s(t) \geq \mathcal{T}$ **then**
    $\text{md}_{\text{rcv}}(t) \leftarrow 0$
    $\text{md}_{\text{snd}}(t) \leftarrow 1$

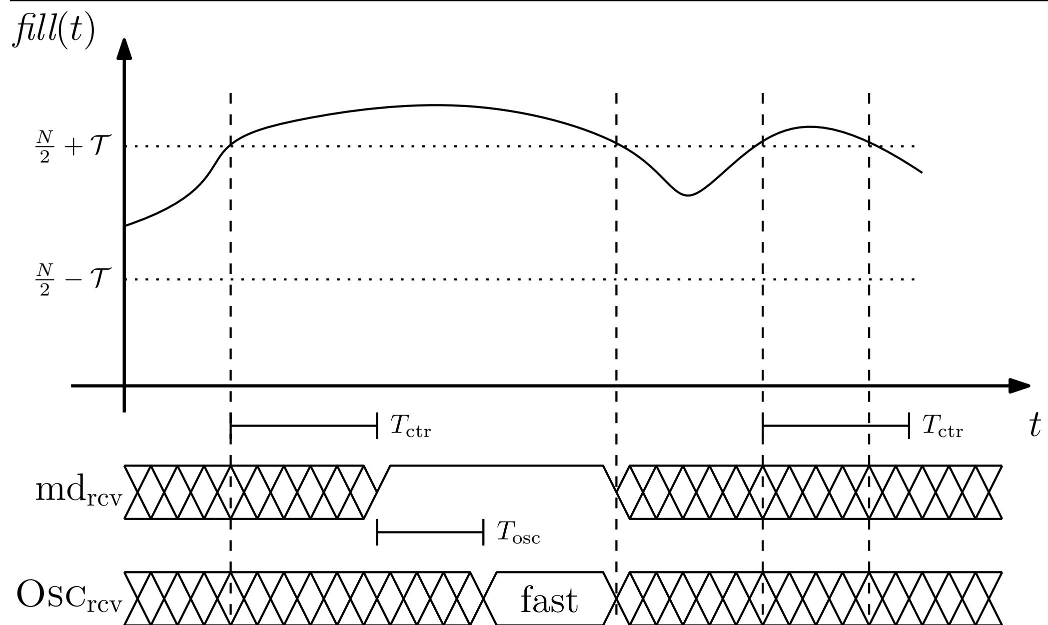

Figure 9.3: ContTh($\mathcal{T}$)'s signals of the receiver. When the fill level crosses the threshold the oscillator switches after $T_{\text{ctr}} + T_{\text{osc}}$ time.

9.3.1 Correctness of ContTh($\mathcal{T}$)

Before we show that, for a $\mathcal{T}$ that is chosen sufficiently large, ContTh($\mathcal{T}$) is *implementable* by a digital circuit in Section 9.3.2, we show that ContTh($\mathcal{T}$) indeed is *correct* (as per Definition 9.2) if $\mathcal{T}$ is chosen small enough.

Theorem 9.3. ContTh($\mathcal{T}$) *is correct if*

$$\begin{aligned}\delta &\leq \mathcal{T}\\ &\leq N/2 - (f^+ - s^-)(T_{osc} + T_{ctr}) - f^+ \max\{\tau_s, \tau_r\}.\end{aligned} \tag{9.1}$$

Recall that $p_r(t) = \mathrm{clk}_r(t)$ and $p_s(t) = \mathrm{clk}_s(t) + N/2$. Thus, when perfectly synchronized, the sender and receiver concurrently access opposite cells of the buffer. The first subtrahend accounts for the fact that the clocks remain unconstrained for $T_{\mathrm{osc}} + T_{\mathrm{ctr}}$ time even after a threshold is reached: the controller guarantees corresponding output only after T_{ctr} time, which is bound to affect clock speeds at most another T_{osc} time later; During this time period, one clock may catch up to the other at rate $f^+ - s^-$. The second subtrahend accounts for the fact that the sender must always access a cell at least τ_r time before the receiver, while the receiver must do so τ_s time before the sender (B3).

Note that these two conditions become fully symmetric when using $\max\{\tau_s, \tau_r\}$ as the minimum required separation between accesses. Translating this wall-clock time difference to the address pointers using the upper bound of f^+ on clock frequencies, we see that the following lemma is the key to showing Theorem 9.3.

Lemma 9.4. *If Eq.* (9.1) *holds, then*

$$\forall t \in \mathbb{R}_{\geq 0} \colon |\,\mathrm{clk}_s(t) - \mathrm{clk}_r(t)| \leq N/2 - f^+ \max\{\tau_s, \tau_r\}\,.$$

Proof. Assume for contradiction that $\mathrm{clk}_s(t) - \mathrm{clk}_r(t) > N/2 - f^+ \max\{\tau_s, \tau_r\} > \mathcal{T}$ for some time t. Let $t_0 \in \mathbb{R}_{\geq 0}$ be the minimal time such that $\mathrm{clk}_s(\tau) - \mathrm{clk}_r(\tau) \geq \mathcal{T}$ for all $\tau \in [t_0, t]$; as $|\,\mathrm{clk}_s(0) - \mathrm{clk}_r(0)| < \delta \leq \mathcal{T}$ by (C1) and (9.1) and both clk_s and clk_r are continuous, such a time t_0 must exist. Observe that $\mathrm{clk}_s(t_0) - \mathrm{clk}_r(t_0) = \mathcal{T}$.

By the specification of the controller (L1), we have that $\mathrm{md}_{\mathrm{snd}}(\tau) = 0$ and $\mathrm{md}_{\mathrm{rcv}}(\tau) = 1$ for all $\tau \in [t_0 + T_{\mathrm{ctr}}, t]$. Thus, we have that $\dot{\mathrm{clk}}_r(\tau) \geq f^- \geq s^+ \geq \dot{\mathrm{clk}}_s(\tau)$ for all $\tau \in [t_0 + T_{\mathrm{ctr}} + T_{\mathrm{osc}}, t]$ by the specification of the clocks ((C2), (C3), and (C5)). Recall that also $\dot{\mathrm{clk}}_r(\tau) \geq s^-$ and $\dot{\mathrm{clk}}_s(\tau) \leq f^+$ at all times τ by (C2) to (C5). We abbreviate $t_{\mathrm{clk}} = t_0 + T_{\mathrm{ctr}} + T_{\mathrm{osc}}$. If $t - t_0 \geq T_{\mathrm{ctr}} + T_{\mathrm{osc}}$, we can thus bound

$$\begin{aligned}\mathrm{clk}_s(t) - \mathrm{clk}_r(t) &= \mathrm{clk}_s(t_0) - \mathrm{clk}_r(t_0) + \int_{t_0}^{t} \dot{\mathrm{clk}}_s(\tau) - \dot{\mathrm{clk}}_r(\tau)\; d\tau\\ &\leq \mathcal{T} + \int_{t_0}^{t_{\mathrm{clk}}} f^+ - s^-\; d\tau + \int_{t_{\mathrm{clk}}}^{t} 0\; d\tau\\ &\leq \mathcal{T} + (f^+ - s^-)(T_{\mathrm{ctr}} + T_{\mathrm{osc}})\\ &\leq \frac{N}{2} - f^+ \max\{\tau_s, \tau_r\}\,. && \text{by Eq. (9.1)}\end{aligned}$$

If $t - t_0 < T_{\mathrm{ctr}} + T_{\mathrm{osc}}$, the second part of the integral vanishes and the first part becomes smaller, showing that the same bound holds. Either way, this contradicts our assumption that $\mathrm{clk}_s(t) - \mathrm{clk}_r(t)$ exceeds this bound.

Finally, in the case that $\mathrm{clk}_r(t) - \mathrm{clk}_s(t) > N/2 - f^+ \max\{\tau_s, \tau_r\}$ we argue analogously, but the roles of the sender and receiver are exchanged. □

Proof of Theorem 9.3. By Lemma 9.4,

$$\begin{aligned} |p_s(t) - p_r(t)| &= |\,\mathrm{clk}_s(t) + N/2 - \mathrm{clk}_r(t)| \\ &\in [f^+ \max\{\tau_s, \tau_r\}, N - f^+ \max\{\tau_s, \tau_r\}]. \end{aligned} \tag{9.2}$$

In particular, the (continuous) sender and receiver address pointers never have the same value modulo N and thus cannot pass each other. Moreover, by our assumptions on the initial clock values (C1), and since $\delta \le 1$, we have that $\mathrm{clk}_s(0), \mathrm{clk}_r(0) \in (-1, 0]$, i.e., $p_r(0) \in (-1, 0]$ and $p_s(0) \in (N/2 - 1, N/2]$ by (B1) and (B2), respectively. Together with (B4), this implies that (i) the first access to each cell that is invalid at time 0 is by the sender, (ii) the first access to each cell that is valid at time 0 is by the receiver, and (iii) each cell is accessed alternatingly by the sender and receiver.

It remains to show that the receiver does not access a cell less than τ_s time after a sender access to the same cell. Similarly, we need to show that the sender does not access a cell less than τ_r time after a receiver access. To this end, suppose cell ℓ is accessed by the sender and receiver at times t_s and t_r, respectively. Thus, $\ell = p_r(t_r) + aN = p_s(t_s) + bN$ for some $a, b \in \mathbb{Z}$, i.e.,

$$\begin{aligned} |p_r(t_r) - p_r(t_s)| &= |p_s(t_s) - p_r(t_s) + (b-a)N| \\ &\ge f^+ \max\{\tau_s, \tau_r\}\,. && \textit{by Eq. (9.2)} \end{aligned}$$

As $\dot{p}_r(t) = \dot{\mathrm{clk}}_r(t)$ and $\dot{\mathrm{clk}}_r(t) \le f^+$ at all times t by (C2) to (C5), we also have $|p_r(t_r) - p_r(t_s)| \le f^+|t_r - t_s|$ and therefore $|t_r - t_s| \ge \max\{\tau_s, \tau_r\}$. Thus, (P1) and (P2) are satisfied for any access to cell ℓ; Since ℓ was arbitrary, this completes the proof. □

9.3.2 Clocked Implementation **ClockedTh**

Next, we provide a simple and efficient controller implementation that works if $\mathcal{T}$ is sufficiently large. Recall that our goal is to detect when $c_s(t) - c_r(t) \ge \mathcal{T}$ or $c_r(t) - c_s(t) \ge \mathcal{T}$. By Lemma 9.4, assuming a correct implementation satisfying (9.1), it holds that the address pointers never reach each other. Together with the equality $c_r(t) - c_s(t) = p_r(t) + N/2 - p_s(t)$, it follows that all we need to check is whether one pointer is more or less than $N/2$ cells ahead of the other or not. This gives us an indication of whether the buffer is more or less than half full, and the more accurately we can decide, the smaller $\mathcal{T}$ can be for the implementation to be correct.

We use the receiver's clock to sample whether the sender's address pointer is currently by more or less than $N/2$ cells ahead of the receiver's address pointer. This is where the full/empty flags come in handy. Instead of having to communicate and

sample $c_s(t)$, the receiver simply samples the flag of cell $\ell+N/2 \bmod N$ when accessing cell $\ell \in [N]$. This occurs at each time t when $\ell = p_r(t) \bmod N = c_r(t) \bmod N$, which means that if the buffer is exactly half full, we had that $p_s(t) \bmod N = \ell+N/2 \bmod N$, i.e., the sender accesses cell $\ell + N/2 \bmod N$ at precisely the same time. This means that it starts setting the full/empty flag of the cell from 0 to 1 at time t, i.e., if the buffer is less than half full, the receiver will successfully sample a stable 0 into flip-flop 'ffa', see Figure 9.4.

Remark. For simplicity, we attribute any unstable reading to the transition of the memory flag of the cell via τ_s. Of course, the parameters of the flip-flop we sample into, the quality of the clock signal, and the delay from the flag's output to the flip-flop's input through the `MUX` all have an effect. Based on a timing analysis of the circuit and adding a suitable phase shift to the clock input of 'ffs' by, e.g., using a buffer, the abstract behavior we assume can be realized. Then, τ_s simply describes the size of the time window during which 'ffs' is vulnerable to metastability induced by a transition of the memory flag of cell ℓ.

In contrast, if the buffer is more than half full, it may be the case that the receiver reads an `M` because the sender is still writing the full/empty flag. Only if it accessed the cell at the latest at time $t - \tau_s$, we can be certain that the result of the read operation is a stable 1. To avoid this asymmetry, we sample cell ℓ at times t when $c_r(t) \bmod N = \ell + f^+\tau_s/2$.

Lemma 9.5. *Suppose time t and $\ell \in [N]$ are such that $c_r(t) \bmod N = \ell + f^+\tau_s/2$ and Eq.* (9.2) *holds. Then*

(1) $c_s(t) - c_r(t) \geq f^+\tau_s/2 \Rightarrow F_\ell(t) = 1$, *and*

(2) $c_r(t) - c_s(t) \geq f^+\tau_s/2 \Rightarrow F_\ell(t) = 0$.

Proof. We show (*1*) first, i.e., assume that $c_s(t) \geq c_r(t) + f^+\tau_s/2$. Then

$$p_s(t-\tau_s) = c_s(t-\tau_s) \geq c_s(t) - f^+\tau_s \geq c_r(t) - f^+\tau_s/2\,.$$

Note that

$$c_r(t) - f^+\tau_s/2 \bmod N = \ell\,,$$

i.e., the sender completed writing cell ℓ (for the most recent time) at time t; Here, (9.2) shows that neither the sender nor receiver cannot have accessed the cell again after the operation was complete. In other words, $F_\ell(t) = 1$, as claimed.

Now we show statement (*2*). We assume that $c_s(t) \leq c_r(t) - f^+\tau_s/2$, while also $c_r(t) - f^+\tau_s/2 \bmod N = \ell$. Hence, the most recent access to cell ℓ was by the reader (again using also (9.2)), which also completed its access (as $N \geq 2$ and we assume that operations are completed within a single clock cycle). In other words, $F_\ell(t) = 0$, as claimed. □

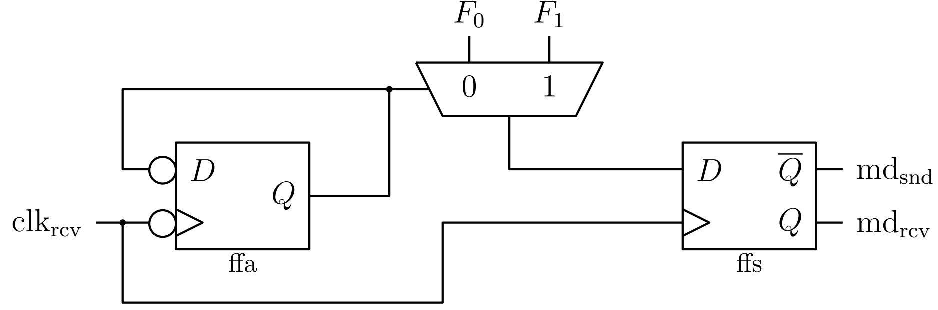

Figure 9.4: Controller ClockedTh for ring-size $N = 2$. Flip-flop 'ffa' stores the address (modulo 2) that is sampled and 'ffs' is the sampled full/empty flag.

Remark. It is worth noting that one could use a purely combinational controller to achieve the same result, i.e., there is no need to rely on clocking. Making use of the clock does also not guarantee that stable values are sampled. However, making use of the clock results in a controller with a smaller threshold value than a straightforward combinational implementation due to the known alignment of the sampling times with one of the clocks.

Based on this idea, we derive a straightforward implementation of the controller. Put simply, the receiver samples the full/empty flag of the cell opposite to the one it currently reads in the ring. More precisely, $\mathrm{md}_{\mathrm{rcv}}$ is the output of a flip-flop (flip-flop 'ffs' in Figure 9.4), into which the receiver samples $F_\ell(t)$ at times t such that $c_r(t) \bmod N = \ell + f^+\tau_s/2$. Signal $\mathrm{md}_{\mathrm{snd}}$ is obtained by negating $\mathrm{md}_{\mathrm{rcv}}$. A circuit implementing this approach for ring size $N = 2$ is shown in Figure 9.4. Here, flip-flop 'ffa' is a modulo 2 counter used to track the address to the current cell to sample. It is initialized to the opposite of the receiver address. We need to ensure that the `MUX` switches to forwarding the respective full flag before flip-flop 'ffs' latches the output of the `MUX`. We do so by computing the select bit on the negated clock signal. This shifts the computation of the select bit by half a clock cycle and ensures correct timing. Note that here we might get metastable mode signals due to switching full flags. Naturally, it is necessary that the mode signal is computed within a single clock cycle; Given the simplicity of the circuit, this is easily achieved.

In the following, denote by $\tau_{\max}$ the maximum propagation time through the circuit shown in Figure 9.4 from the full/empty flags at the top to $\mathrm{md}_{\mathrm{snd}}$ (without τ_s, which is already taken into account by Lemma 9.5). Lemma 9.5 then characterizes the proposed controller.

Corollary 9.6. *Assume that the control circuit ClockedTh is used in accordance with Lemma 9.5 and that* (P1) *and* (P2) *hold until time* $t > T_{ctr} = 1/s^- + \tau_{\max}$.

If for all $t' \in [t - T_{ctr}, t]$ *we have that*

$$\mathrm{clk}_s(t') - \mathrm{clk}_r(t') \geq f^+\tau_s/2\,,\ \textit{then}\ \mathrm{md}_{rcv}(t) = 1\ \textit{and}\ \mathrm{md}_{snd}(t) = 0 \qquad \text{(case (i))}$$

If for all $t' \in [t - T_{ctr}, t]$ *we have that*

$$\mathrm{clk}_r(t') - \mathrm{clk}_s(t') \geq f^+\tau_s/2\,,\ \textit{then}\ \mathrm{md}_{snd}(t) = 1\ \textit{and}\ \mathrm{md}_{rcv}(t) = 0 \qquad \text{(case (ii))}$$

Proof. The outputs $\mathrm{md}_{\mathrm{rcv}}(t)$ and $\mathrm{md}_{\mathrm{snd}}(t)$ at time t are derived from the output of 'ffs' at time t (or one inverter delay earlier). As the receiver clock runs at least at speed s^- (by (C2) to (C4)), flip-flop 'ffs' is latched at least every $1/s^-$ time. Hence, taking into account the propagation time through the MUX and the definition of $\tau_{\max}$, the outputs correspond to the output of one of the flags at some time $t' \in [t - T_{\mathrm{ctr}}, t]$. As the MUX selects the flag output it forwards according to Lemma 9.5, we can apply the lemma to time t', yielding in case (i) that a stable 1 is latched and in case (ii) that a stable 0 is latched. This results in the desired corresponding circuit outputs $\mathrm{md}_{\mathrm{rcv}}(t) = 1$ and $\mathrm{md}_{\mathrm{snd}}(t) = 0$ (case (i)) or $\mathrm{md}_{\mathrm{snd}}(t) = 1$ and $\mathrm{md}_{\mathrm{rcv}}(t) = 0$ (case (ii)), respectively. □

We now can derive the correctness of the controller, expressed in Theorem 9.7, conditional on simple constraints on $\mathcal{T}$.

Theorem 9.7. *Assume that Eq.* (9.1) *holds, where* $T_{ctr} = 1/s^- + \tau_{\max}$, *and* $\mathcal{T} \geq f^+\tau_s/2$. *Then* ClockedTh *is an implementation of* ContTh($\mathcal{T}$).

Proof. If there is some access to a valid cell by the sender or to an invalid cell by the reader, there must be a minimal such time (because the start of a cell access is a discrete event). Denote by $\bar{t}$ the minimal such time if such an access occurs and set $\bar{t}$ to infinity otherwise.

We claim that the circuit implements ContTh($\mathcal{T}$) at all times $0 \leq t < \bar{t}$; from this, we will infer the statement of the theorem. Recall that by (L1) and (L2), the controller implementation needs to output a specific (and stable) signal only if the condition in Line 3 or the one in Line 7 of Algorithm 5 continuously holds during the previous T_{ctr} time. According to Algorithm 5, this is the case at time t if and only if $c_s(t') - c_r(t') \geq \mathcal{T}$ for all $t' \in [t - T_{\mathrm{ctr}}, t]$ or $c_r(t') - c_s(t') \geq \mathcal{T}$ for all $t' \in [t - T_{\mathrm{ctr}}, t]$.

Consider such a time t. Note that $t > T_{\mathrm{ctr}}$, as $|c_s(0) - c_r(0)| < \delta \leq \mathcal{T}$ by (C1) and Eq. (9.1), i.e., neither condition is satisfied at time 0. We consider the two cases (i) $\mathrm{clk}_s(t') - \mathrm{clk}_r(t') \geq \mathcal{T}$ for all $t' \in [t - T_{\mathrm{ctr}}, t]$ and (ii) $\mathrm{clk}_r(t) - \mathrm{clk}_s(t) \geq \mathcal{T}$ for all $t' \in [t - T_{\mathrm{ctr}}, t]$.

Case (i): Since $\mathcal{T} \geq f^+\tau_s/2$, we may apply case (i) of Corollary 9.6. We conclude that $\mathrm{md}_{\mathrm{rcv}}(t) = 1$ and $\mathrm{md}_{\mathrm{snd}}(t) = 0$.

Case (ii): In this case we may apply case (ii) of Corollary 9.6, from which we deduce that $\mathrm{md}_{\mathrm{rcv}}(t) = 0$ and $\mathrm{md}_{\mathrm{snd}}(t) = 1$.

We conclude that the circuit meets the specification at all times $t < \bar{t}$. In particular, we can apply Lemma 9.4 at times $t < \bar{t}$, showing that

$$|c_s(t) - c_r(t)| \leq N/2 - f^+ \max\{\tau_s, \tau_r\}\,.$$

If $\bar{t} \neq \infty$, continuity of c_s and c_r implies that also

$$|c_s(\bar{t}\,) - c_r(\bar{t}\,)| \leq N/2 - f^+ \max\{\tau_s, \tau_r\}\,.$$

Reasoning analogously to the proof of Theorem 9.3, it follows that (P1) and (P2) are not violated at times $t \leq \bar{t}$, contradicting the definition of $\bar{t}$. We conclude that $\bar{t} = \infty$, implying that the circuit from Figure 9.4 indeed implements $\textsc{ContTh}(\mathcal{T})$. □

Finally, we translate the theorem into a sufficient condition for the correctness of the link implementation. To state its performance, we define the *latency* as the maximum time between consecutive accesses of the sender and receiver to the same cell, plus the setup/hold time at the receiver (as the data should be stable before it is used). The *throughput* is the guaranteed minimum rate of delivered packets; Note that no packet drops or corruptions occur in our implementations.

Corollary 9.8. *For $N \geq 2\Delta$, the given clocked link implementation is correct with latency N/s^- and throughput $1/s^-$, where*

$$\Delta = \lceil (f^+ - s^-)(T_{osc} + 1/s^- + \tau_{\max}) + f^+ \max\{\tau_s, \tau_r\} + \max\{\delta, f^+\tau_s/2\} \rceil\,.$$

Proof. Set $T_{\text{ctr}} = 1/s^- + \tau_{\max}$. We choose $\mathcal{T}$ such that (9.1) and $\mathcal{T} \geq f^+\tau_s/2$ are both satisfied. This is possible if and only if $N/2 \geq \Delta$, which holds by the prerequisites of the corollary. Then Theorem 9.7 yields that the circuit from Figure 9.4 indeed implements $\textsc{ContTh}(\mathcal{T})$, and Theorem 9.3 shows that the implemented controller is correct. The performance bounds follow immediately from correctness and the fact that the guaranteed minimum clock rate is s^-. □

9.4 Performance Evaluation

application-specific integrated circuit (ASIC)

In this section we discuss an ASIC design for which we carried out simulations. The design is implemented with the UMC 65 nm standard cell library. It operates at roughly 2 GHz. This demonstrates that the derived performance bounds indeed lead to promising results.

Metastability. In this section, we also demonstrate simulated executions that show the circuit behaving according to the specification, despite the reoccurring metastability of its control signals; see Figure 9.8. In fact metastability of the control signals is likely to be observed in an implementation, since by its attempt to synchronize the two oscillators, the controller repeatedly drives the control signals into metastability; much like the experimental setups to measure deep metastability of synchronizers

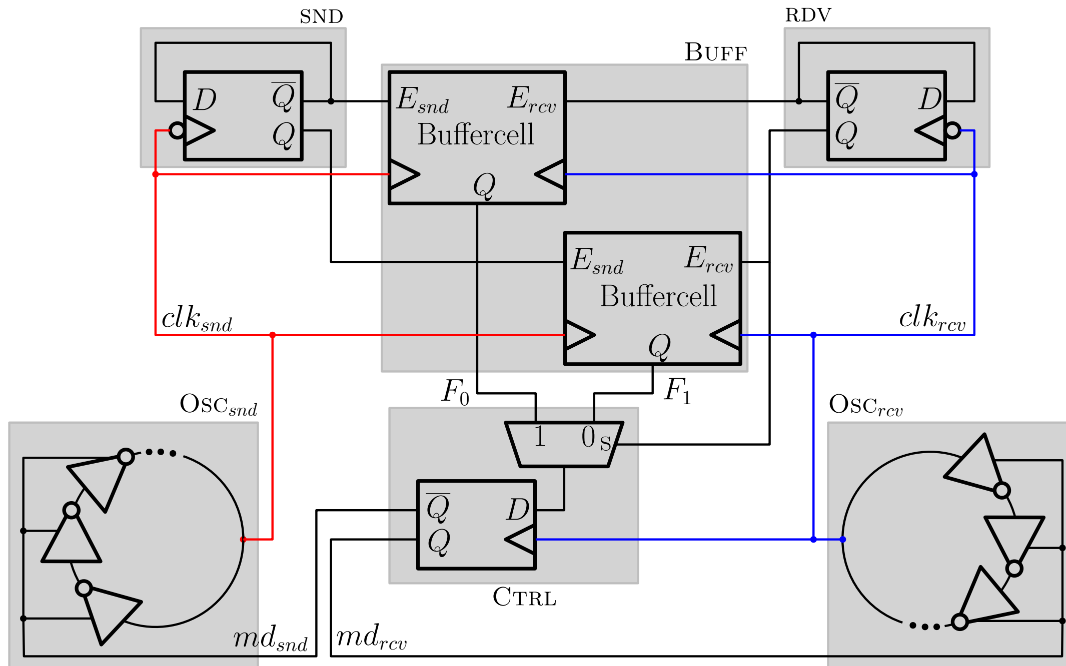

Figure 9.5: Implementation of the system with buffer size $N = 2$. Clock regions are marked red (sender) and blue (receiver).

[97, 84]. We would like to point out that any such demonstration, however, does not replace the correctness proofs in Section 9.3. Proving that metastability is not a problem would require to verify the absence of metastability (or resulting effects) in all circuit components, except for the places to which our proofs show metastability to be confined.

9.4.1 ASIC Implementation

The complete design is shown in Figure 9.5. It comprises the digital controller (Ctrl), tunable sender and receiver oscillator (Osc$_{\text{snd}}$,Osc$_{\text{rcv}}$), and the ring buffer of size $N = 2$.

At a buffer size of 2, the address logic in Snd and Rcv reduces to a simple modulo 2 counter. Hence, we only have a single register for the sender and the receiver side. The modulo counter operates on the negated clock to ensure a stable output at the time a register in the buffer is accessed. The buffer consists of two buffer cells that store the full/empty flags. The design of a buffer cell that can be set to 1 by one clock domain and reset to 0 by another clock domain is given in Figure 9.6. The design uses a flip-flop for each clock domain that forwards its output to a `xor` which computes the output. If the sender flip-flop is enabled it copies the negation of the

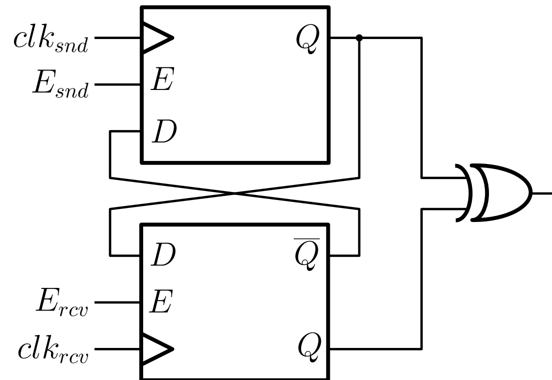

Figure 9.6: Implementation of a buffer cell that can only be set by the sender and only be reset by the receiver.

receiver state. For differing states, the `xor` will output a 1. If the receiver enables its flip-flop the state of the sender is copied. Hence, the output of the buffer cell is reset to 0.

We can optimize the controller from Figure 9.4, as we already compute the write address of the sender. We remove the flip-flop 'ffa' and read the address from the SND address logic. The `MUX` in CTRL is connected such that we sample from the buffer cell that is currently not written by the sender. The timing diagram in Figure 9.7 shows the behavior of CTRL.

Recall that we require the sender and receiver oscillators to be well-behaved even when control bits are unstable. Specifically, we require that (i) oscillator frequencies are always within $[s^-, f^+]$, and (ii) frequency mode changes occur within T_{osc} time ((C2) to (C5)). This is why we resorted to starved-inverter ring oscillators that guarantee such behavior [2]; we designed the sender and receiver starved inverter rings at transistor level following [91]. Note that we do not need the full control logic overhead typically required to drive the starved inverter cells, since we only need two speeds: slow and fast. Hence, the control logic of the oscillator takes a single bit and adjusts the delay of the starved inverters according to fast or slow mode. As the receiver oscillator $\mathrm{OSC}_{\mathrm{rcv}}$ additionally drives the control logic CTRL its load is higher than the load driven by the sender oscillator. The effect is that $\mathrm{OSC}_{\mathrm{rcv}}$ has slightly slower fast and slow modes. The difference does not matter as long as the oscillator speeds lie within their theoretical bounds. One can keep the imbalance very small by decoupling the oscillators from the load with buffers.

Signals $\mathrm{md}_{\mathrm{rcv}}$ and $\mathrm{md}_{\mathrm{snd}}$ are used as control signals of the rings, which run at roughly 2 GHz and 2.3 GHz for input 0 and 1, respectively.

Extracting delay and frequency parameters from the standard cell library we get $\Delta = 1$ in Corollary 9.8, i.e., ClockedTh is provably correct for $N \geq 2$. This fits to the bounds given in Table 9.1.

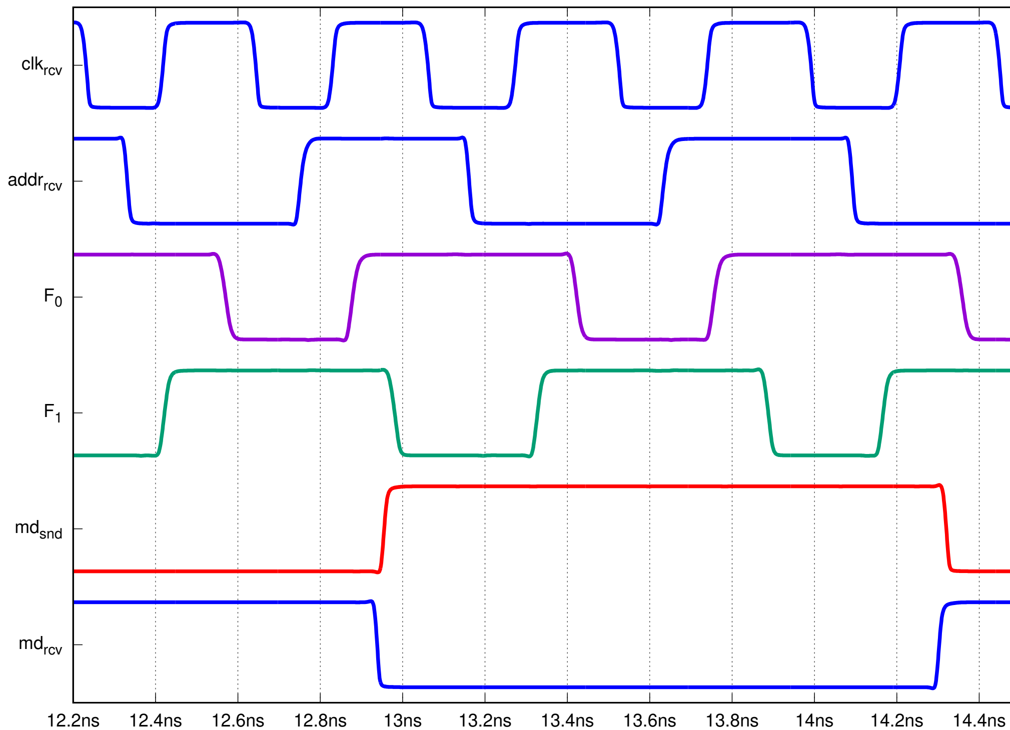

Figure 9.7: Timing diagram of the controller CTRL. The address $addr_{rcv}$ decides which full flag is sampled into the register of the controller at a rising clock transition.

9.4.2 Frequency Stability of Tunable Oscillators

Typically, the accuracy of oscillator frequencies is stated as a two-sided error, i.e., if the nominal frequency of the oscillator is f and it has a relative frequency error of at most r, then at any time its momentary frequency is between $(1-r)f$ and $(1+r)f$.

Recall from (C5) that we require that the fast oscillator mode is always faster than the slow oscillator mode. For a 2 to 2.3 GHz clock we must therefore tune the clock within an error r that satisfies the condition $2 \cdot (1+r)^2/(1-r)^2 \leq 2.3$, i.e., $r \leq 3.49\%$ is a sufficient bound on the frequency error. In case these error margins are too restrictive, we could choose a clock with a larger gap between fast and slow modes, e.g., 2 to 2.5 GHz. Depending on the outcome of the timing analysis (see also Corollary 9.8), this may require a larger buffer size N.

For comparison, the accuracy requirements for the oscillators used in [30] are as follows. If both the sender and receiver oscillator run at (roughly) the same nominal frequency, $\Delta p < g/S$ is proven to be sufficient for the correctness of the design, where Δp is the relative phase change per clock cycle, $S = 4$ the number of synchronizer stages, and $g = 0.1$ the guard band. However, the proof assumes a perfectly stable

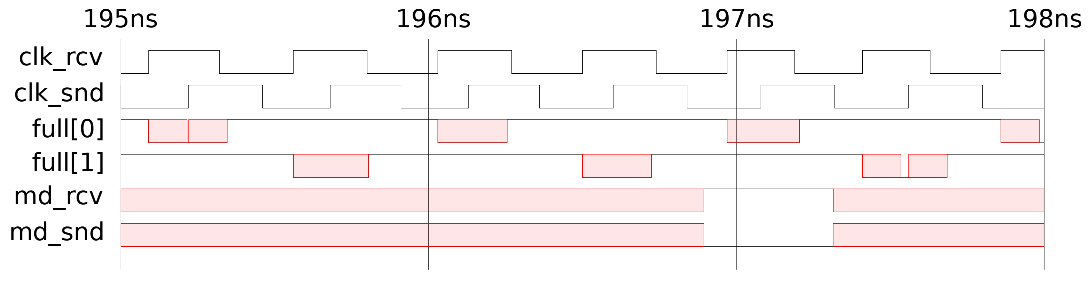

Figure 9.8: Gate-level simulations for link with ClockedTh.

receiver clock. If the receiver and sender oscillator may drift, the above inequality becomes $2\Delta p(1 + \Delta p) < g/S$. This is equivalent to a frequency error of less than 1.24%.

9.4.3 Gate level and SPICE Simulations

VHSIC hardware description language (VHDL)

We first ran gate-level VHDL simulations of designs of our ClockedTh controller with delay and setup/hold parameters from the ASIC design. The starved-inverter rings were simulated by forward Euler integration of a first-order ordinary differential equation (ODE) model, where current clock rates are independently uniformly distributed in each integration step to account for drift. The high respectively low frequency of the starved inverter rings were set to 2.3 GHz respectively 2 GHz. Potential meta-/instability of signals was simulated by X in a worst-case manner; this includes flip-flops with setup/hold violations, full/empty flags, and oscillator mode signals. Simulated traces were 5 ms (10^7 clock cycles) long and all in accordance with the proven correctness results. We stress that signals $\mathrm{md}_{\mathrm{rcv}}$ and $\mathrm{md}_{\mathrm{snd}}$ were unstable (X) almost all the time due to the conservative gate model assumptions, yet no buffer over-/underrun is encountered; cf. Figure 9.8.

We then ran Spice simulations for the ClockedTh design: The design was implemented in Spice using standard cells and parameters of the UMC 65 nm library combined with an implementation of a tunable ring oscillator. The oscillator runs at speed 2.09 GHz in slow mode and 2.42 GHz in fast mode. Taking into account timing constraints and propagation delays of the elements we can use a ring buffer of size two, according to Corollary 9.8.

When simulating the design for 500 ns (about 1100 clock cycles) no faulty behavior could be detected. However, the simulation confirms what we stressed previously. In almost 50% of the cases, the setup time of 'ffs' (see Figure 9.4) is violated due to late transitions of the full flags. Still, the controller behaves correctly and the two oscillators run synchronously.

Figure 9.9(a) shows the full flags of the buffer. Sender and receiver alternatingly access cell 0 and cell 1. Figure 9.9(b) shows the clock signals produced by the sender

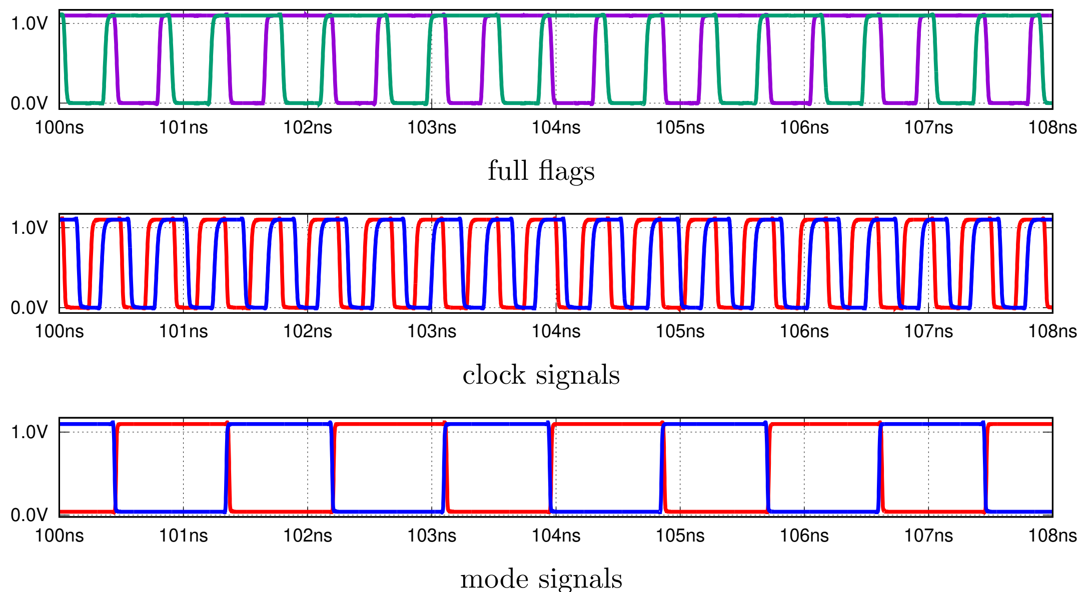

Figure 9.9: Ring buffer with two cells. (a) Rising and falling full flags of cell 0 (purple) or 1 (green) show write and read access to the respective cell. (b) Clock signals of the sender (red) and receiver (blue) oscillator. When stabilized both run at 2.28 GHz on average. (c) The mode signals for the sender (red) and receiver (blue) sides alternate between fast (2.42 GHz) and slow (2.09 GHz) mode.

and receiver oscillators. When stabilized, the sender is ahead by slightly more than a clock cycle. Both run on average with a frequency of roughly 2.28 GHz. Figure 9.9(c) shows the mode signals of the sender and receiver which are computed by ClockedTh.

9.4.4 Increasing Initialization Slack

If a sufficiently small δ (i.e., initial clock offset) cannot be guaranteed, the address pointers may collide. However, if the pointers move apart sufficiently far, the link will resume operating as intended. Note that the pointers colliding and moving at the same speed (i.e., the clocks running at the same speed) is an unstable equilibrium state, as the control logic aims at pushing them apart. Accordingly, this is a metastable state of the link, which can be expected to resolve fairly quickly.

We used a variation of the Spice simulation that allows us to initialize sender and receiver clocks to a specific offset (due to the machinery, the simulation does not start exactly at 0 ns). Together with a suitable initialization of the full/empty flags, this simulates one of the clocks being started earlier.

We simulated the link with small initial offsets of the continuous pointers, i.e., $p_s(0) - p_r(0) = C_s(0) - C_r(0) + N/2 \approx 0$, with the goal of finding a good tradeoff

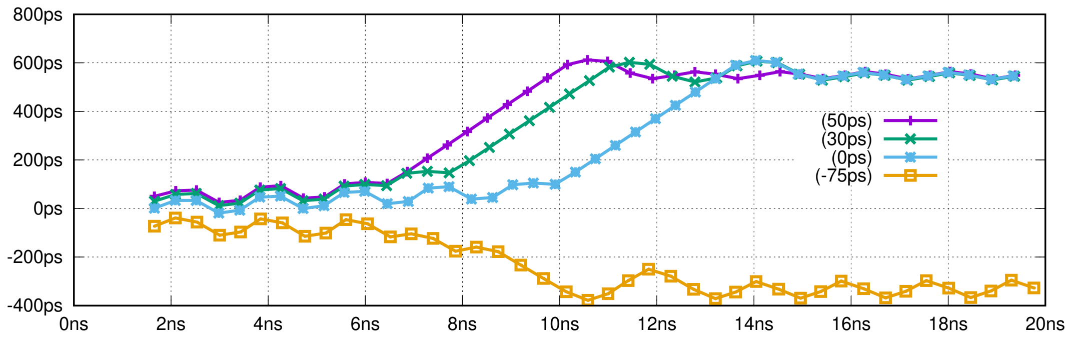

Figure 9.10: Offset of the sender and receiver pointers over simulation time. When initializing the pointer offset to 0 ps, 30 ps, 50 ps and −75 ps, we observe different times to stabilization. According to the analysis, setup, and hold times cannot be violated once the link is stabilized.

between resolution time and precision of the initialization. Figure 9.10 shows the pointer offset of the sender and the receiver clock $(p_s(t) - p_r(t) - N/2)$ over time t for different initializations. We see that simulations with an initial offset of 0 ps, 30 ps and 50 ps stay in the metastable state until eventually, the sender advances by one clock cycle relative to the receiver and the simulation stabilizes. Similarly, a simulation with an initial offset of −75 ps stays in the metastable state until the receiver advances by one clock cycle relative to the sender and the simulation reaches the corresponding stable state. Simulations with 30 ps resp. −75 ps offsets resolve after 11 ns resp. 10 ns. Hence, if the designer is willing to wait 11 ns after initialization, it is sufficient to guarantee to avoid this window of 105 ps during initialization. At the given clock speed, this corresponds to a much larger $\delta = 1/f^+ - 0.5 * 105$ ps, which in our setting is roughly 360 ps. In general, waiting for a couple of clock cycles after initialization increases the slack δ to being close to a full clock cycle.

Network Synchronization 10

In this chapter, we present and extend the results published at the ASYNC 2020 conference [13]. Here we bring everything together, we use hazard-free circuits to implement the GCS algorithm. In [13] we propose a hardware implementation, which we discuss in this chapter. Furthermore, we show full proof of the implementation. Extensive simulations show the advantages of our design over state-of-the-art clock generation grids.

Outline. Starting in Section 10.1 we give an introduction. Section 10.1 also discusses related work. We evaluate on the hardware modules in Section 10.2 and present their implementation in Section 10.3. Finally, Section 10.4 presents Spice simulations of the implementation and compares our implementation to related solutions. We briefly discuss follow-up questions in Section 10.5.

10.1 Introduction

By using a distributed clock synchronization algorithm, we essentially create a single, system-wide clock domain without needing to spread a clock signal from a single dedicated source with a small skew. In our setting, local clock regions correspond to nodes in a graph, they are connected by an edge if they directly communicate (i.e. exchange data). Thus, nodes of the clock synchronization algorithm communicate only if the respective regions exchange data for computational functionality. This leads to an easy integration of our algorithm into the existing communication infrastructure.

In contrast to Chapter 9, we present synchronization of more than two nodes. As the synchronization algorithm is integrated into an existing communication structure, we do not incorporate data buffers or similar into the implementation. Estimates of the offsets are made solely by measuring the clocks of neighboring nodes. Modules for measurement and control, thus, also differ from the previous implementation.

We measure the offset between two nodes by measuring the time difference in their clock pulses. This is achieved by an approach similar to classical TDC circuits [43]. The control module evaluates the measurements and sets the oscillator speed accordingly. The measurement is forwarded to the control module without the use of synchronizers. Hence, the control module may face meta-/unstable inputs. We show that the implementation of the control module is hazard-free. Thus, the mode signal is stable when the GCS algorithm requires it.

Asymptotic Guarantees. Recall that the GCS algorithm guarantees a local skew of $\mathcal{O}(\delta \log_{\mu/\rho} D)$ and a global skew of $\mathcal{O}(\delta D)$ (cf. Section 6.4). Parameter ρ is the oscillator drift, μ is the speedup factor, and δ is the measurement error.

We can control the base of the logarithm in the local skew bound by choosing μ. Picking, e.g., $\mu = 100\rho$ means that $\log_{\mu/\rho} D \leq 1$ for any $D \leq 100$. The constants hidden in the $\mathcal{O}$-notation matter, but they are reasonably small. Concretely, for a grid network of 32×32 nodes in the 15 nm FinFET-based Nangate open cell library (OCL) [75], 2 GHz clock sources with an assumed drift of $\rho = 10^{-5}$, and $\mu = 10^{-3}$, our simple sample implementation guarantees that $\delta \leq 5$ ps in the worst case. The resulting local skew is 30 ps, well below a clock cycle. We stress that this enables much faster communication than for handshake-based solutions incurring synchronizer delay.

In order to show that the asymptotic behavior is relevant already to current systems and with our (pessimistic) ρ, we compare the above results to skews obtained by clock trees in the same grid networks in Section 10.4.2.

10.2 Hardware Modules

For a hardware implementation of the OffsetGCS algorithm, we break down the distributed algorithm into modules as we did in Chapter 9. Here we are concerned with a network that comprises an arbitrary number of nodes connected by (more than one) link. Data buffers are not included, the offset between nodes is estimated by sampling the clock.

We distinguish between the implementation of a node and the implementation of a link. Per node, we have a local clock and a controller. Per link, we have two offset measurement modules, one for each node connected by the link. We introduce the modules in Section 8.2. In this section, we specify parts of the offset measurement and the control module that are left open. Further, we relate the delay $T_{\max}$ from Section 6.4 to the module delays.

10.2.1 Local Clock

The clock signal of node v is derived from a tunable oscillator. We present the specification of a tunable oscillator in Section 8.2. The oscillator follows requirements (C1) to (C5). It has input md_v, the mode signal (given by the control module, described in Section 10.2.3), and output clk_v, the clock signal. The mode signal md_v is used to tune the frequency of the oscillator within a factor of $1 + \mu$. An oscillator responds within time $T_{\mathrm{osc}} \geq 0$, i.e., switching between the two frequency modes takes at most T_{osc} time.

10.2.2 Time Offset Measurement

In order to check whether the FT conditions are met, a node v needs to measure the current phase offset $L_w(t) - L_v(t)$ to each of its neighbors w. This is achieved by a time offset measurement module between v and each neighbor w. Node v has no direct access to $L_w(t)$ as propagation delays are prone to uncertainty. Hence, a node can only estimate the offset to w, where the offset estimate is denoted by $\widehat{O}_w(t)$.

Inputs to the offset measurement are signals clk_v and clk_w. The output $\widehat{O}_w(t)$ is encoded by a unary encoding of length 2ℓ, for $\ell > 0$. We evaluate further on the output in the following paragraph.

Thresholds. The algorithm does not require full access to the function $\widehat{O}_w(t)$, but only to the knowledge of whether $\widehat{O}_w(t)$ has reached one of the thresholds defined by (FT-1) and (FT-2). FT defines infinitely many thresholds, i.e., the algorithm checks for each $s \in \mathbb{N}$ whether (FT-1) or (FT-2) is satisfied. However, practically the system can only measure finitely many thresholds. The maximum skew between two neighbors is bounded by the local skew. Hence, the local skew is also a bound on the largest threshold that can be reached.

We encode $\widehat{O}_w(t)$ by an unary encoding of the reachable thresholds. Let $\ell \in \mathbb{N}$ be the largest number such that $(2\ell + 1)\kappa + \delta < \mathcal{L}$, where $\mathcal{L}$ is the upper bound on the local skew. Then $\widehat{O}_w(t)$ is defined as a binary word of length 2ℓ. The bits are denoted (from left to right) by $Q_w^{\ell}, \ldots, Q_w^{1}, Q_w^{-1}, \ldots, Q_w^{-\ell}$. For example, a module with $\ell = 2$ has 4 outputs Q_w^2, Q_w^1, Q_w^{-1}, and Q_w^{-2} corresponding to thresholds $-3\kappa - \delta$, $-\kappa - \delta$, $\kappa - \delta$, and $3\kappa - \delta$. Each signal $Q_w^{\ell}, \ldots, Q_w^{1}, Q_w^{-1}, \ldots, Q_w^{-\ell}$ is a function of time. For better readability, we omit the function parameter t when it is clear from context. For $i \in \{1, \ldots, \ell\}$ each output bit $Q_w^{\pm i}(t)$ denotes whether $\widehat{O}_w(t)$ has reached the corresponding threshold. All possible offsets will always be in the measurement range because we encode each offset that is smaller than $\mathcal{L}$.

Decision Separator. Any realistic hardware implementation of the offset measurement will have to account for the setup/hold times of registers used for the output. We dedicate the decision separator ε to account for (small) additional setup/hold times.

We require that signal $Q_w^{\pm i}(t)$ is 1 at time t if the offset exceeds the ith threshold and we require that signal $Q_w^{\pm i}(t)$ is 0 at time t if the offset does not exceed the ith threshold. When the offset is close to the threshold (within ε), then we allow that $Q_w^{\pm i}(t)$ is unconstrained, i.e., $Q_w^{\pm i}(t) \in \{0, \mathfrak{u}, 1\}$.

Definition 10.1 (decision seperator). *Let ε be a (small) time span with $\kappa \gg \varepsilon > 0$. At time t, we require the following constraints for all $i \in \{1, \ldots, \ell\}$. Signal $Q_w^{\pm i}(t)$ is set to 1 if the offset estimate is larger than $\mp(2i-1)\kappa - \delta$.*

$$\begin{aligned} \widehat{O}_w(t) \geq -(2i-1)\kappa - \delta \;\Rightarrow\; Q_w^{i}(t) = 1 \\ \widehat{O}_w(t) \geq (2i-1)\kappa - \delta \;\Rightarrow\; Q_w^{-i}(t) = 1 \end{aligned} \tag{M1}$$

Signal $Q_w^{\pm i}(t)$ is set to 0 if the offset measurement is smaller than $\mp(2i-1)\kappa - \delta - \varepsilon$.

$$\begin{aligned} \widehat{O}_w(t) \leq -(2i-1)\kappa - \delta - \varepsilon \;\Rightarrow\; Q_w^{i}(t) = 0 \\ \widehat{O}_w(t) \leq (2i-1)\kappa - \delta - \varepsilon \;\Rightarrow\; Q_w^{-i}(t) = 0 \end{aligned} \tag{M2}$$

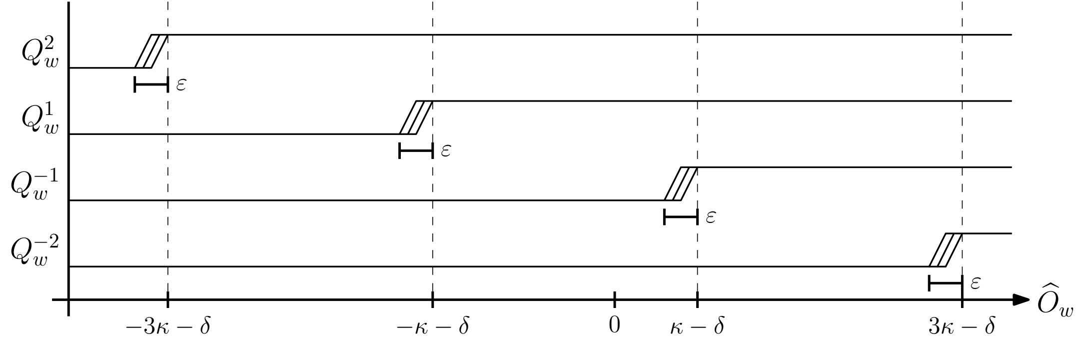

Figure 10.1: Signal transitions of the output bits $Q_w^{\pm i}$ relative to $\widehat{O}_w(t)$.

Otherwise, $Q_w^{\pm i}(t)$ is unconstrained.

$$\begin{aligned} \widehat{O}_w(t) \in -(2i-1)\kappa - \delta - (0, \varepsilon) \;\Rightarrow\; Q_w^i(t) \in \{0, \mathfrak{u}, 1\} \\ \widehat{O}_w(t) \in (2i-1)\kappa - \delta - (0, \varepsilon) \;\Rightarrow\; Q_w^{-i}(t) \in \{0, \mathfrak{u}, 1\} \end{aligned} \tag{M3}$$

Figures 10.1 to 10.3 show the timing of signals $Q_w^{-1}(t)$, $Q_w^1(t)$, and $Q_w^2(t)$ in relation to the clock of neighbor w. When clk_v transitions to 1 the measurement module takes a snapshot of the outputs $Q_w^{\pm i}$. In Figure 10.3 we show two examples.

Figure 10.1 depicts transitions of the signals $Q_w^{\pm i}(t)$. Along the x-axis the figure shows increasing $\widehat{O}_w$. As $\widehat{O}_w$ increases bits $Q_w^{\pm i}(t)$ flip to 1. The decision separator ε is small enough that no two bits can flip at the same time. If $\widehat{O}_w(t) = 0$ all bits $Q_w^i(t) = 1$ and $Q_w^{-i}(t) = 0$.

Figure 10.2 also depicts transitions of the signals $Q_w^{\pm i}(t)$, but along the x-axis $L_v(t)$ increases while L_w is fixed. We mark time $\mathcal{L}_w$ at which $L_v(t) = L_w$. A digital implementation is only able to measure the offset on a clock event, e.g., a rising clock transition. Hence, $\mathcal{L}_w$ will be the time where clk_w rises. When $L_v(t) = L_w$, we have that $\widehat{O}_w(t) = 0$, such that all bits $Q_w^i(t) = 1$ and $Q_w^{-i}(t) = 0$. As $L_v(t)$ increases $\widehat{O}_w(t)$ decreases. Hence, the signal transitions look like a mirror image of Figure 10.1.

Example 10.2. *Regarding Figure 10.3, a measurement module with $\ell = 2$ can have output $Q_u(t) = 1100$ if $\kappa - \delta - \varepsilon \geq \widehat{O}_u(t) \geq -\kappa - \delta$. The output may become $Q_w(t) = 11\mathfrak{u}0$ if $\kappa - \delta > \widehat{O}_w(t) > \kappa - \delta - \varepsilon$.*

In general, closely synchronized clocks have output $Q_w^{\pm i}(t) = 1^\ell 0^\ell$. If the clock of v is ahead of ws clock, the measurement $Q_w^{\pm i}(t)$ contain more 0s than 1s. Similarly, if vs clock is behind the clock of w, the outputs contain more 1s than 0s.

Furthermore, we can bind the number of unstable bits by a single bit, i.e., at most one output is $\mathfrak{u}$ at a time:

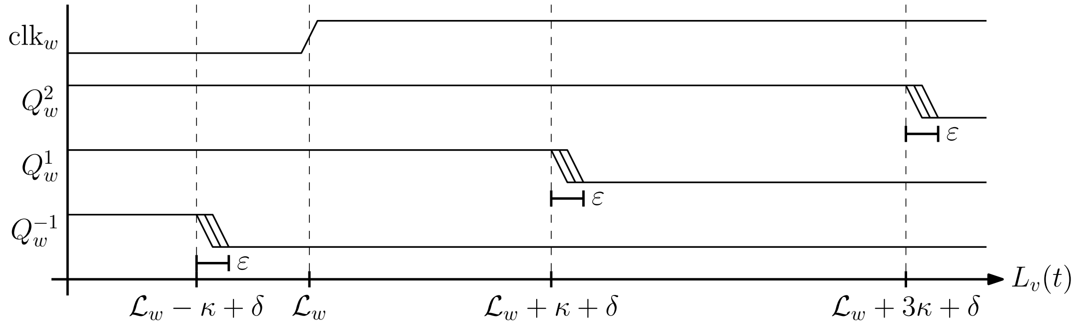

Figure 10.2: Signal transitions of the output bits $Q_w^{\pm i}(t)$ relative to clk_w, assuming that $L_w(t)$ is known to v ($\widehat{O}_w(t) = L_w(t) - L_v(t)$).

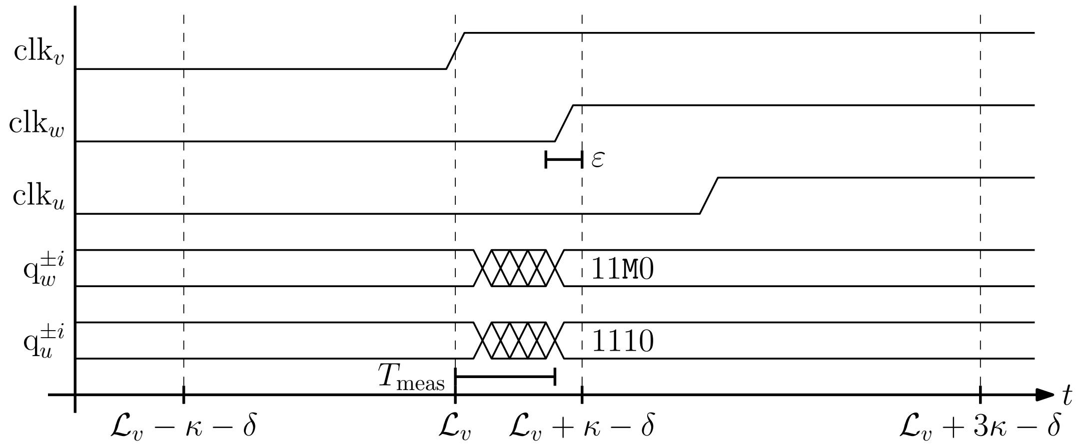

Figure 10.3: Example measurements from node v to nodes u and w.

Lemma 10.3. *At every time t there is at most a single $i \in \{1, \dots \ell\}$ such that $Q_w^{\pm i}$ is unconstrained.*

Proof. Assume for $i > 0$, that $Q_w^i(t)$ is unconstrained. Then we have that

$$-(2i-1)\kappa - \delta - \varepsilon < \widehat{O}_w(t) < -(2i-1)\kappa - \delta\,.$$

Hence, for all $i' < i$ it holds that $\widehat{O}_w(t) < -(2i'-1)\kappa - \delta$, such that, by Definition 10.1, $Q_w^{i'} = 0$ and $Q_w^{-j} = 0$ for all j. For all $i' > i$ we obtain $\widehat{O}_w(t) > -(2i'-1)\kappa - \delta - \varepsilon$, as $\kappa > \varepsilon$. Thus, by Definition 10.1, $Q_w^{i'} = 1$. An analogous argument shows that there is only one bit unconstrained if $Q_w^{-i}(t)$ is unconstrained. □

Remark. In essence the proof of Lemma 10.3 shows that the ε-regions in Figures 10.1 and 10.2 do not overlap.

No hardware implementation of the measurement can have instantaneous transitions of the output. Although we account for setup/hold times, we also have to account for signal transition and gate delays. Let T_{meas} denote the maximum end-to-end latency of the measurement module, i.e., an upper bound on the elapsed time from when $Q_w^{\pm i}(t)$ is set to when the measurements are available at the output. More precisely, if $Q_w^{\pm i}(t')$ is set to $x \in \{0, 1\}$ for all t' in $[t - T_{\text{meas}}, t]$, then the corresponding output $\mathrm{q}_w^{\pm i}(t)$ is x as depicted in Figure 10.3.

10.2.3 Control Module

Each node v is equipped with a control module. Its input is the (unary encoded) time measurement, i.e., bits $\mathrm{q}_w^{\pm i}(t)$, for each of v's neighbors. The control module controls the tunable oscillator, i.e., it outputs the mode signal $\mathrm{md}_v(t)$.

Algorithm OffsetGCS pushes v to fast mode if FT is satisfied, otherwise the algorithm defaults to slow mode. Intuitively, FT triggers when there is an offset that crosses threshold i and no other offset is below threshold $-i$ for some $i \in \{1, \dots, \ell\}$. Hence, we select the maximum and minimum of the offsets $Q_w^{\pm i}$ to all neighbors w. The maximum offset indicates for each i that there is an offset that crosses the threshold. Similarly, the minimum offset indicates for each i that all offsets crossed the corresponding threshold.

Since the network also includes self-loops (cf. Section 6.1), each node, conceptually, measures the offset to itself. The offset to self is always 0. In practice, that means that the maximum only needs to consider neighbors that are ahead and the minimum only needs to consider neighbors that are behind. For $i \in \{1, \dots, \ell\}$, signals $Q_w^{-i}(t)$ indicate whether node w is ahead and similar bits $Q_w^i(t)$ indicate whether w is behind. Thus, the ℓ-bit encodings of maximum ($Q_{max}^{-i}(t)$) and minimum ($Q_{min}^i(t)$) are computed as

$$Q_{max}^{-i}(t) := \bigvee \{Q_w^{-i}(t) \mid w \text{ is neighbor of } v\}\,,$$

$$Q_{min}^{i}(t) := \bigwedge \{Q_w^{i}(t) \mid w \text{ is neighbor of } v\}\,.$$

As FT is satisfied if $Q^{-i}_{max}(t)$ and $Q^{i}_{min}(t)$ are both 1 for any i in $\{1,\dots,\ell\}$. The $\mathrm{md}_v(t)$ signal can easily be computed by

$$\mathrm{md}_v(t) \coloneqq \bigvee\{Q^{-i}_{max} \wedge Q^{i}_{min} \mid i \in \{1,\dots,\ell\}\}\,.$$

A discussion of instability in the controller is deferred to Section 10.3.2. In Figure 10.4 we visualize the metastable region together with the parameters we introduced. It shows how to compute the $\mathrm{md}_v(t)$ signal. If the slow condition is satisfied then $\mathrm{md}_v(t) = 0$ and if the fast condition is satisfied then $\mathrm{md}_v(t) = 1$.

Although the computation can be performed by simple combinational logic, a hardware implementation needs to account for propagation delays. Denote by T_{ctr} the maximum end-to-end delay of the controller circuit, i.e., the delay between its inputs (the measurement offset outputs) and its output $\mathrm{md}_v(t)$. The formal specification of the control module is given in Section 8.2.

10.2.4 ClockedGCS Algorithm

The module specifications above, together, specify a realization of the OffsetGCS algorithm in hardware that we name ClockedGCS. In the following lines, we describe how to relate the measurement error to the hardware modules. We state the ClockedGCS algorithm and prove that it implements the OffsetGCS algorithm.

We denote the maximum end-to-end latency of the computation by $T_{\max}$, it combines the delays of the three modules, i.e., $T_{\max} = T_{\mathrm{meas}} + T_{\mathrm{ctr}} + T_{\mathrm{osc}}$. For a simple implementation, $T_{\max}$ naturally becomes a lower bound on the clock period. $T_{\max}$ denotes the time it takes from a rising clock edge until the oscillator guarantees a stable rate. Designs with a clock period beyond $T_{\max}$ are possible when buffering measurements and mode signals.

Example 10.4. *A timing diagram with the output signals of the modules and the clock rate is given in Figure 10.5. The offset measurement switches from* 1100 *(close to synchronous) to* 1110 *(v lagging behind w) and causes the oscillator to go to fast mode.*

Measurement Error. We relate the module delays to δ. Recall that δ describes the quality of the offset estimate. We split δ into two parts, the *propagation delay uncertainty* and the *maximum end-to-end latency.* The propagation delay uncertainty accounts for variations in the time a signal takes to propagate from a node's oscillator to the measurement module of its neighbors. Suppose clock signals arrive at the measurement module with a larger or smaller delay than expected (usually due to variation in the fabrication process or environmental influences), then the module may measure larger or smaller offsets. We denote the propagation delay uncertainty by δ_0.

The second source of error is the drift of the clocks when not measuring. The offset is measured once per clock cycle, it is used until the next measurement is made.

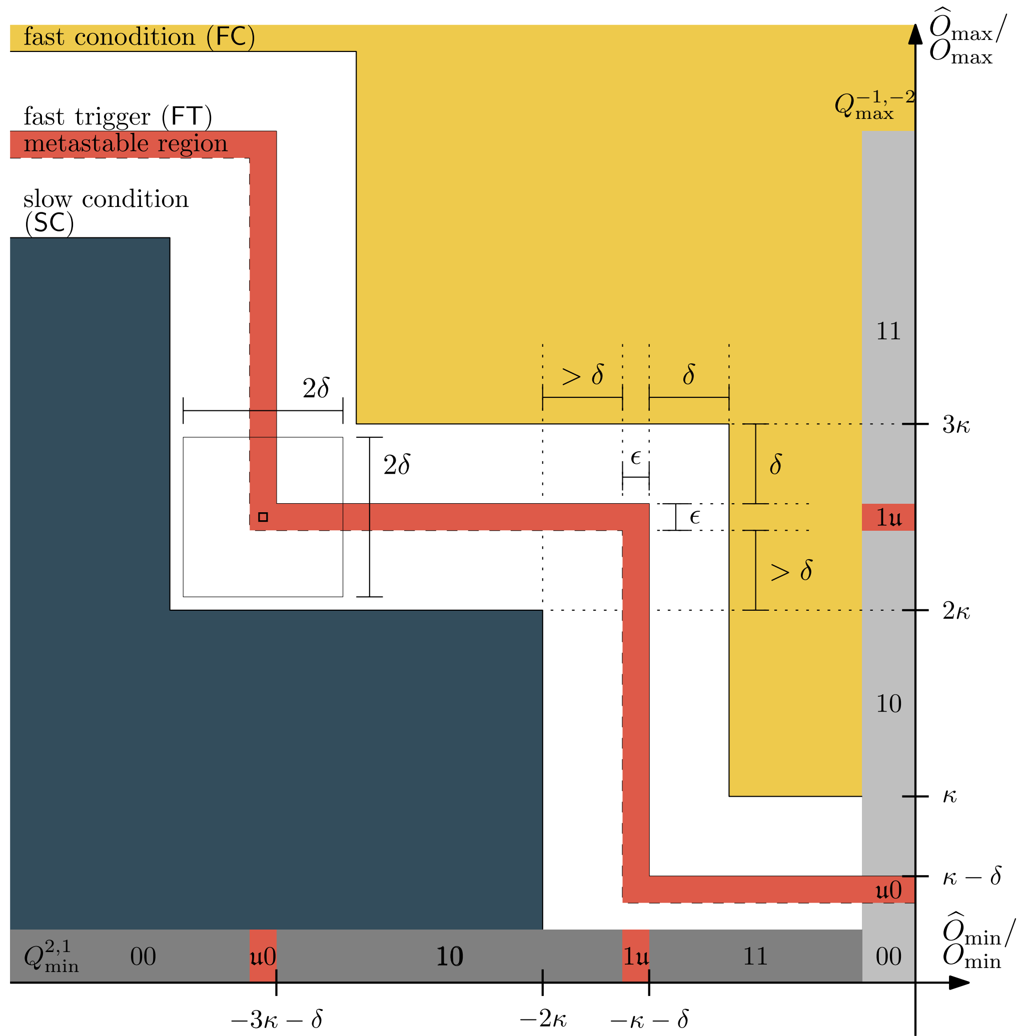

Figure 10.4: As before we visualize FC and SC with respect to $O_{\min}$ and $O_{\max}$. FT and the metastable region are visualized with respect to $\widehat{O}_{\min}$ and $\widehat{O}_{\max}$. Along the axis we show the encodings $Q_{\min}^{2,1}$ and $Q_{\max}^{-1,-2}$, marking also the effect of instability.

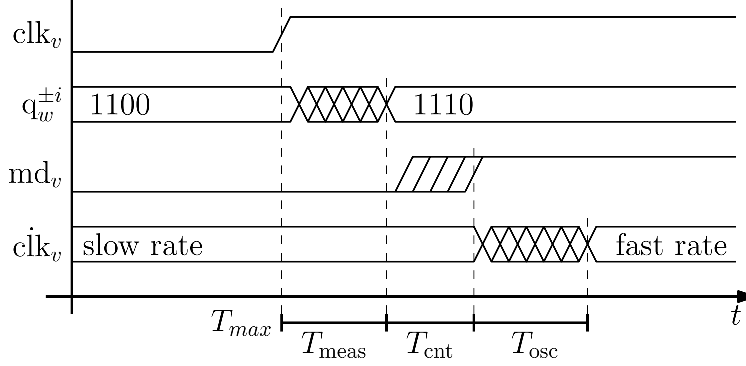

Figure 10.5: Timing diagram for the maximum end-to-end delay of the computation.

During this time the actual offset may change due to different modes and drift of oscillators. We denote the duration of a clock cycle (in slow mode with no drift) by T_{clk}. The maximum difference in rate between any two logical clocks is bounded by $(1+\rho)(1+\mu)-1 = \rho+\mu+\rho\mu$. Thus, the maximum change of the offset during a clock cycle is at most $(\rho+\mu+\rho\mu)(T_{\mathrm{clk}}+T_{\mathrm{max}})$. This describes the second contribution to the uncertainty of the measurement. Summing up both contributions the measurement error becomes

$$\delta = \delta_0 + (\rho + \mu + \rho\mu) \cdot (T_{\mathrm{clk}} + T_{\mathrm{max}})\,.$$

Clocked Algorithm. An essential difference of our implementation to the continuous time algorithm OffsetGCS is that measurements are performed only at clock ticks, i.e., at discrete time steps. The discrete-time algorithm ClockedGCS as carried out by the modules described above is given in Algorithm 6.

Each module definition comprises a maximum propagation delay (e.g. T_{ctr}). For the measurement module, we defined a possible metastable assignment using the decision separator ε. We denote the possibly metastable assignment that also accounts for the propagation delay by $\leftarrow_{\mathfrak{u}}$. Once suitable values of δ_0 and T_{max} are determined, δ can be computed easily.

Lemma 10.5. *With $\delta = \delta_0 + (\rho + \mu + \rho\mu) \cdot T_{clk}$, ClockedGCS satisfies Inequality* (6.9) *at all times t.*

Proof. The algorithm measures the offset $\widehat{O}_w(t)$ at each clock tick. Hence, we show that between two clock ticks the uncertainty never grows beyond δ. Let t_{clk} and t'_{clk} be two consecutive clock ticks at node v. By the specification above, the measurement at time t_{clk} has precision δ_0, such that

$$\left|\widehat{O}_w(t_{clk}) - (L_w(t_{clk}) - L_v(t_{clk}))\right| \le \delta_0\,.$$

Algorithm 6 Clocked algorithm ClockedGCS. The assignment $\leftarrow_{\mathfrak{u}}$ denotes a possibly unstable assignment.

1: **at** each clock tick, at time t_{clk} **do**
2: **for** each $i \in \{1, \ldots, \ell\}$ **do**
3: **for** each w is neighbor of v **do**
4: $Q_w^i \leftarrow_{\mathfrak{u}} \widehat{O}_w(t_{clk}) \geq -(2i-1)\kappa - \delta$
5: $Q_w^{-i} \leftarrow_{\mathfrak{u}} \widehat{O}_w(t_{clk}) \geq (2i-1)\kappa - \delta$

6: **at** each time t **do**
7: $Q_{min}^i \leftarrow_{\mathfrak{u}} \bigwedge\{Q_w^i(t) \mid w \text{ is neighbor of } v\}$
8: $Q_{max}^{-i} \leftarrow_{\mathfrak{u}} \bigvee\{Q_w^{-i}(t) \mid w \text{ is neighbor of } v\}$
9: $\mathrm{md}_v \leftarrow_{\mathfrak{u}} \bigvee\{Q_{min}^i(t) \wedge Q_{max}^{-i}(t) \mid i \in \{1, \ldots, \ell\}\}$

During time interval $[t_{clk}, t'_{clk}) \leq T_{\mathrm{ctr}}$ the clock rates may be different for neighbors. The difference between logical clocks grows at most by

$$(1+\rho)(1+\mu) \cdot T_{\mathrm{clk}} - T_{\mathrm{clk}} = (\rho + \mu + \rho\mu) \cdot T_{\mathrm{clk}}\,,$$

such that for $t \in [t_{clk}, t'_{clk})$,

$$\begin{aligned} &\left|\widehat{O}_w(t) - (L_w(t) - L_v(t))\right| \\ &\leq \left|\widehat{O}_w(t_{clk}) - (L_w(t_{clk}) - L_v(t_{clk}))\right| + (\rho + \mu + \rho\mu) \cdot T_{\mathrm{clk}} \\ &\leq \delta_0 + (\rho + \mu + \rho\mu) \cdot T_{\mathrm{clk}}\,. \end{aligned}$$

Hence, at every time the error is at most δ, such that Inequality (6.9) is satisfied. □

The skew bounds known from the GCS algorithm apply to ClockedGCS, as ClockedGCS implements the OffsetGCS algorithm for suitable choice of ℓ, δ_0, ρ, μ and κ. The constraints of Theorem 6.10 are met by the following choices:

Equation (I1)	is satisfied when choosing $\mu > 2\rho$,
Equation (6.9)	is satisfied according to Lemma 10.5, and
Constraint $\kappa > 2\delta$	is satisfied by choosing κ accordingly.

Corollary 10.6. *For suitable choices of ℓ, δ_0, ρ, μ and κ, ClockedGCS maintains the local skew of Theorem 6.8.*

10.3 Hardware Implementation

In this section, we discuss the gate-level design and its performance measures of the modules from Section 10.2. Target technology is the 15 nm FinFET-based Nangate OCL [75]. The gate-level design is laid out and routed with Cadence Encounter,

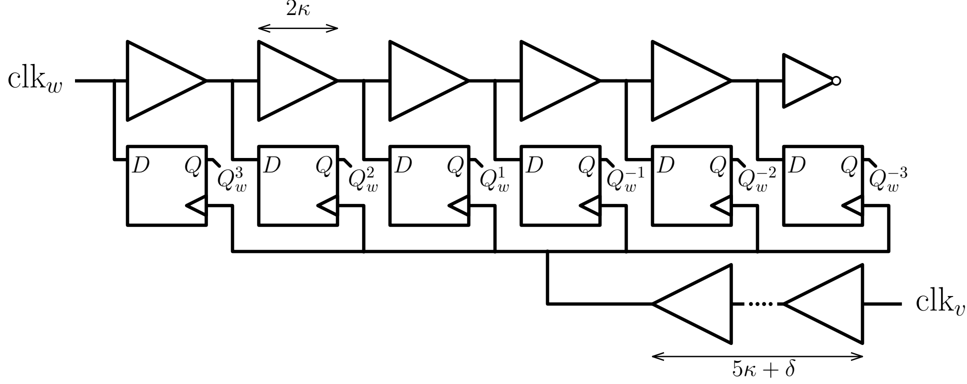

Figure 10.6: Schematics of the time offset measurement module for $\ell = 3$.

which is also used for the extraction of parasitics and timing. Local clocks run at a frequency of approximately 2 GHz, controllable within a factor of $1 + \mu \approx 1 + 10^{-4}$. We use $\mu = 10^{-4}$ here to make the interplay of ρ and μ better visible in traces. We compile two systems of 4 respectively 7 nodes connected in a line. To resemble a realistically sparse spacing of clocks, we placed nodes at distances of 200 µm.

10.3.1 Gate-level Implementation

Offset Measurement. We depict schematics of the time offset measurement in Figure 10.6. The figure shows a linear TDC-based circuitry for the module which measures the time offsets between nodes v and w. Buffers are used as delay elements for incoming clock pulses. The offset is measured in steps of 2κ, hence, buffers in the upper delay line have a delay of 2κ. The delay line is tapped after each buffer for corresponding $Q_w^{\pm i}$. A chain of flip-flops takes a snapshot of the delay line by sampling the taps. We require $Q_w^{-i} = 0$ and $Q_w^{i} = 1$ when $\widehat{O}_w \geq -\kappa - \delta$ and $\widehat{O}_w \leq \kappa - \delta - \varepsilon$ (cf. Figure 10.1). Thus, we delay clk_v by $5\kappa + \delta + \varepsilon$.

time to digital converter (TDC)

Example 10.7. *If both clocks are perfectly synchronized, i.e., $L_v = L_w$, then the state of the flip-flops will be $Q_w^3 Q_w^2 Q_w^1 Q_w^{-1} Q_w^{-2} Q_w^{-3} = 111000$ after a rising transition of clk_v. Now, assume that clock w is ahead of clock v, say by a small $\varepsilon > 0$ more than $\kappa + \delta$, i.e., $L_w = L_v + \kappa - \delta + \varepsilon$. For the moment assuming that we do not make a measurement error, we get $\widehat{O}_w = L_w - L_v = \kappa - \delta + \varepsilon$. From the delays in Figure 10.6 one verifies that in this case, the flip-flops are clocked before clock w has reached the second flip-flop with output Q^1, resulting in a snapshot of 110000. Likewise, an offset of $\widehat{O}_w = L_w - L_v = 3\kappa - \delta + \varepsilon$ results in a snapshot of 100000, etc.*

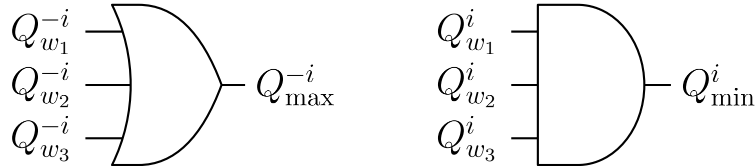

Figure 10.7: Schematics of the controller part that, for $i \in \{1, 2, 3\}$, computes the minimum and maximum offset to neighbors w_1, w_2, and w_3.

Control Module. Given node v's time offsets to its neighbors in unary encoding the control module computes the minimal and maximal threshold levels that have been reached. The circuits in Figures 10.7 and 10.8 implement the control module for 3 neighbors w_1, w_2, and w_3. As described in Section 10.2.3 we only need to compute the maximal value of bits Q^{-i}_w and the minimal value of bits Q^i_w. They can be easily computed by a `or` resp. `and` over all w as shown in Figure 10.7.

Given the maximal and minimal values, the circuit in Figure 10.8 computes if FT conditions are satisfied, i.e., if there is an $i \in \{1, 2, 3\}$ that satisfies both FT-1 and FT-2. If FT is satisfied, then md_v is set to 1.

Tunable Oscillator. As a local clock source, we use a ring oscillator. Some of the inverters in the ring oscillator are starved-inverters which allows us to control the frequency by a voltage input. The control voltage is used to switch between fast mode and slow mode. Nominal frequency is around 2 GHz, controllable by a factor $1 + \mu \approx 1 + 10^{-4}$ via the md_v signal. We choose $\rho \approx \mu/10 \approx 10^{-5}$, assuming a moderately stable oscillator. This is below drifts achievable with uncontrolled ring oscillators. We defer a discussion of the issue to Section 10.5.

Timing Parameters. We next discuss how the modules' timing parameters relate to the extracted physical timing of the above design.

The time required for switching between oscillator modes T_{osc} is about the delay of the ring oscillator, which in our case is about $1/(2 \cdot 2\,\mathrm{GHz}) = 250\,\mathrm{ps}$. An upper bound on the measurement latency T_{meas} plus the controller latency T_{ctr} is given by a clock cycle (500 ps) plus the delay (25 ps) of the circuitry in Figure 10.7 and Figure 10.8. In our case, delay extraction of the circuit yields $T_{\mathrm{meas}} + T_{\mathrm{ctr}} < 500\,\mathrm{ps} + 25\,\mathrm{ps}$. We thus have, $T_{\max} < T_{\mathrm{meas}} + T_{\mathrm{ctr}} + T_{\mathrm{osc}} = 775\,\mathrm{ps}$.
The uncertainty in measuring if $\widehat{O}_w$ has reached a certain threshold, δ_0, is given by the uncertainties in latency of the upper delay chain plus the lower delay chain in Figure 10.7. For the described naive implementation using an uncalibrated delay line, this would be problematic. With an uncertainty of $\pm 5\%$ for gate delays, and starting with moderately sized κ, extraction of minimum and maximum delays showed

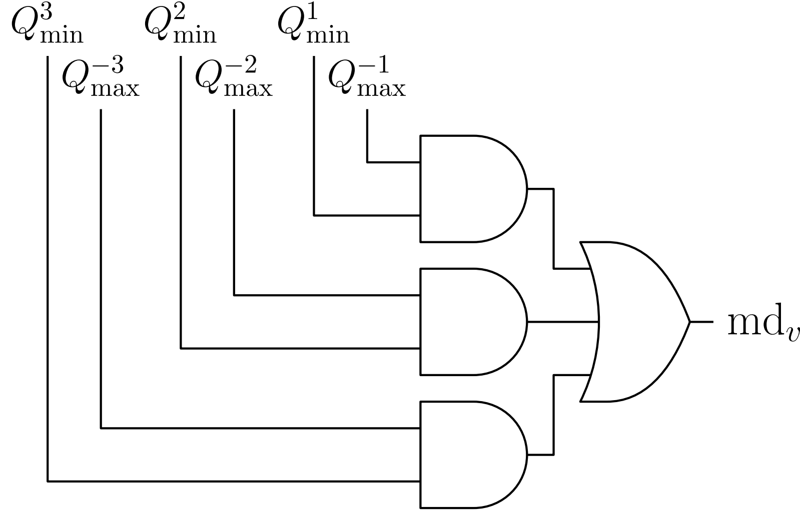

Figure 10.8: Schematics of the controller part that computes the mode signal.

that the constraints for δ and κ from Theorem 6.10 were not met. Successive cycles of increasing δ and κ do not converge due to the linear dependency of δ and κ on the uncertainty δ_0 with a too large factor. Rather, delay variations (of the entire system) have to be less than $\pm 1\%$ for the linear offset measurement circuit, depicted in Figure 10.6, to fulfill requirements of Theorem 6.10.

Improved Offset Measurement. Figure 10.9 shows an improved TDC-type offset measurement circuit that does not suffer from the problem above. Conceptually the TDC of node v that measures offsets w.r.t. node w is integrated into the local ring oscillator of neighboring node w. If w has several neighbors, e.g., up to 4 in a grid, they share the taps but have their own flip-flops within node w.

Figure 10.9 presents a design for $\ell = 2$ with 4 taps and a single neighbor v. In our hardware implementation we set $\ell = 2$, as even for $\mu/\rho = 10$ this is sufficient for networks of diameter up to around 80 hops (see how to choose this set of thresholds in the specification of this module in Section 10.2). The gray buffers at the offset measurement taps decouple the load of the remaining circuitry. An odd number of starved inverters is used to set slow or fast mode for node w (at the bottom of the oscillator ring). The delay elements at the top are inverters instead of buffers to achieve a latency of $\kappa = 10\,\mathrm{ps}$. We inverted the clock output to account for the negated signal at the tap of clock w at the top.

Integration of the TDC into w's local ring oscillator greatly reduces uncertainties at both ends: (i) the uncertainty at the remote clock port (of node w) is removed to a

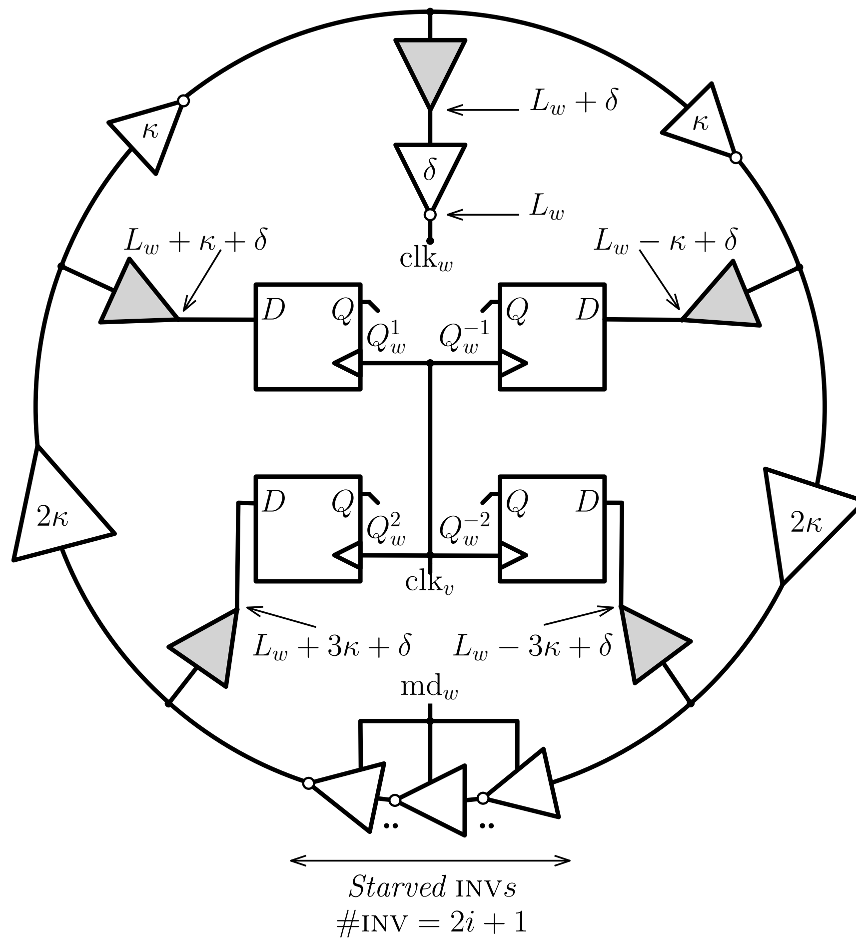

Figure 10.9: Schematic of the improved offset measurement implementation. The v's offset measurement is integrated into w's ring oscillator. Labels of the delay elements denote their delay. We also annotate the phase offset that we measure at each tap.

large extent, since the delay elements which are used for the offset measurements are part of w's oscillator, and (ii) the uncertainty at the local clock port is greatly reduced by removing the delay line of length $5\kappa + \delta$. The remaining timing uncertainties are the latency from taps to the D-ports of the flip-flops and from clock v to the clk_v-ports of the flip-flop. Timing extraction yielded $\delta_0 < 4\,\text{ps}$ in presence of $\pm 5\%$ gate delay variations.

From Theorem 6.10, we thus readily obtain $\kappa \approx 10\,\text{ps}$ and $\delta \approx 5\,\text{ps}$ which matched the previously chosen latencies of the delay elements. Applying Theorem 6.8 finally yields the bounds:

$$\mathcal{G}(t) \leq 1.223\kappa D = 12.23D\,\text{ps and}$$
$$\mathcal{L}(t) \leq (2\lceil \log_{10}(1.223D) \rceil + 1)\kappa\,.$$

For our design with diameter $D = 3$ this makes a maximum global skew of 36.69 ps and a maximum local skew of $3\kappa = 30\,\text{ps}$. Also, for our design with diameter $D = 6$ this makes a maximum global skew of 73.38 ps and a maximum local skew of $3\kappa = 30\,\text{ps}$.

Remark. Considerably larger systems, e.g., a grid with side length of $W = 32$ nodes and diameter $D = 2W - 2 = 62$, still are guaranteed to have a maximum local skew of $3\kappa = 30\,\text{ps}$. If we choose $\mu = 10^{-3}$, the base of the logarithm in the skew bound increases from 10 to 100.

10.3.2 Hazard-Free Control Module

It remains to show that the control module is hazard-free. We can split the task by showing first that the circuit in Figure 10.7 is hazard-free, i.e., that the following equations hold:

$$1 \dots 1 Q^{-1}_{max} \dots Q^{-\ell}_{max} = (\widehat{O}_{\text{max}})_{\mathfrak{u}}\,,$$
$$Q^{\ell}_{min} \dots Q^{1}_{min} 0 \dots 0 = (\widehat{O}_{\text{min}})_{\mathfrak{u}}\,.$$

Then it is left to show that the circuit in Figure 10.8 computes $\gamma_v(t)$ as defined in Section 6.4.1.

Lemma 10.8. *The implementation of the control module, given in Figures 10.7 and 10.8, is hazard-free, i.e., it computes* $\text{md}_v(t) = (\gamma_v(t))_{\mathfrak{u}}$.

Proof. Signal Q^i_{min} is computed by `and` over all Q^i_w. The `and` will only propagate an $\mathfrak{u}$ if there is (at least) one $Q^i_w = \mathfrak{u}$ and all other $Q^i_{w'} = 1$. If for the smallest measurement $Q^i_w = 0$, then this will mask all $\mathfrak{u}$'s of larger measurements. Only the (single) $\mathfrak{u}$ of the smallest measurement can propagate to the Q_{min} output. Thus, the output has a single $\mathfrak{u}$ at most. Similar, for `or` over all Q^{-i}_w, $\mathfrak{u}$ only propagates when all $Q^i_{w'} = 0$. The 1 part of the largest measurement will mask all $\mathfrak{u}$'s of smaller measurements. Hence, Q_{max} has at most a single $\mathfrak{u}$ in the position the largest measurement has an $\mathfrak{u}$. □

Remark. Alternatively, we can show that the control module is hazard-free by the results of Ikenmeyer et al. [52]. Both, the circuit in Figure 10.7 and the circuit in

Figure 10.8, are monotone, i.e., both use `and` and `or` gates only. By [52] any monotone circuit is hazard-free.

By Lemma 10.3 and constraints (M1) to (M3) the unary encoding of measurement $\widehat{O}_w$ has the form 1^*0^* or $1^*\mathtt{u}0^*$. In other words, the output of the measurement module has a single u (at most) that is conveniently located between 1's on the left and 0's on the right.

10.4 Simulation and Comparison

simulation program with integrated circuit emphasis (SPICE)

We ran SPICE simulations of the post-layout extracted design with Cadence Spectre. Simulated systems had 4 and 7 nodes arranged in a line, as described in Section 10.3. Nodes are labeled 0 to 3 (respectively 6). For the simulations, we set $\mu = 10\rho$ (instead of 100ρ), resulting in slower decrease of skew, to better observe how skew is removed.

10.4.1 SPICE Simulations on a 4 Node Topology

Scenarios. We present three simulation scenarios with different initial skews. By starting with a significant initial skew we can simulate best the behavior of the algorithm. Simulation time of all scenarios is 1000 ns ($\approx$ 2000 clock cycles). We simulated the following three scenarios:

AHEAD Node 1 is initialized with an offset of 40 ps ahead of all other nodes.

BEHIND Node 1 is initialized with an offset of 40 ps behind all other nodes.

GRADIENT Nodes are initialized with small skews on each edge, that sum up to a large 105 ps global skew.

Remark. All three scenarios break the assumption that the initial skew maintains the local skew bound of 30 ps. Due to its self-stabilizing features an execution of the algorithm still converges to a small skew. In our implementation, the initial skew should not exceed half a clock cycle ($\approx$ 250 ps), as otherwise, the algorithm tends to synchronize clock pulses that belong to different clock cycles.

Global and Local Skews. Figure 10.10 depicts the local and global skews of all scenarios. Observe that, from the beginning, all local skews decrease until they reach less than 9 ps. The local skew then remains in a stable oscillatory state. This is well below our worst-case bound of 30 ps on the local skew. We observe that the global skew slightly increases at the beginning of scenario AHEAD and after roughly 500 ns in scenario BEHIND.

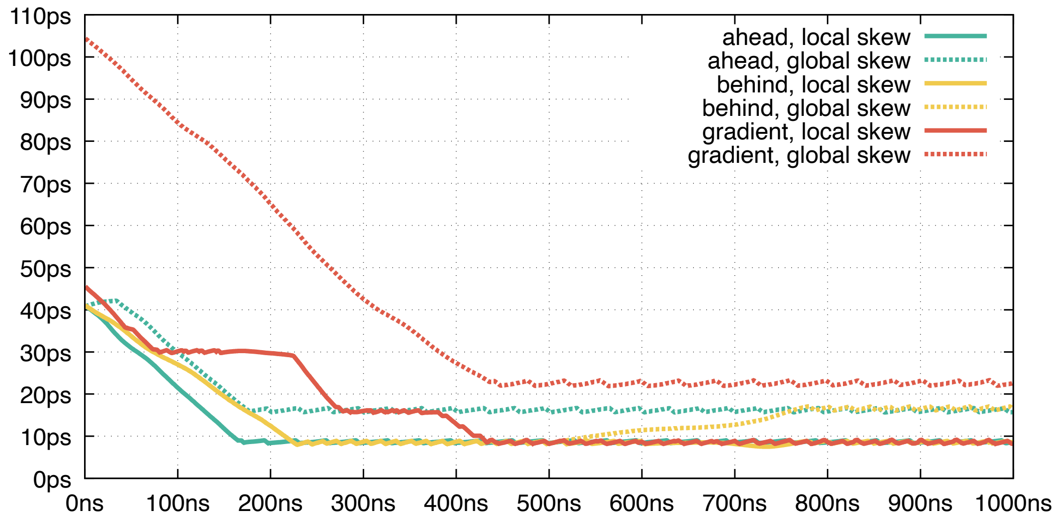

Figure 10.10: Maximum local skew (solid) and global skew (dotted) for scenarios AHEAD (green), BEHIND (yellow), and gradient (red).

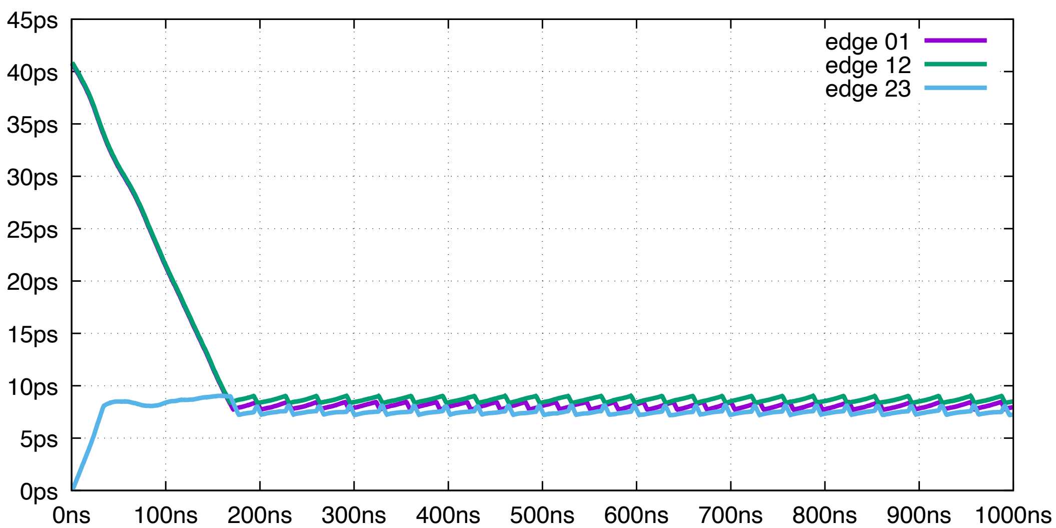

Figure 10.11: Skews in simulation of scenario AHEAD.

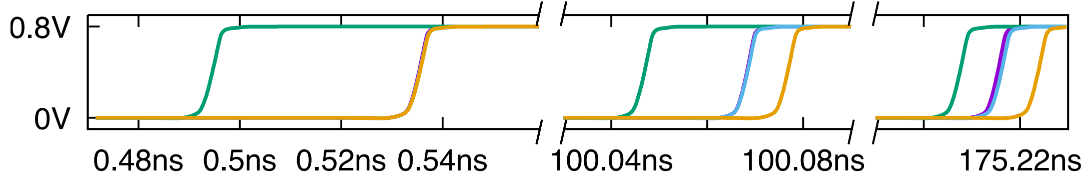

Figure 10.12: Excerpt of scenario AHEAD. Clock signals of node 0 (purple), 1 (green), 2 (blue), and 3 (yellow). Nodes from left to right: (i) 1 before 0, 2, 3, (ii) 1 before 0, 2 before 3, (iii) 1 before 0, 2 before 3.

One Node Ahead. Figure 10.12 shows the clock signals of nodes 0 to 3 at three points in time for the first scenario AHEAD: (i) shortly after the initialization, (ii) around 100 ns, and (iii) after 175 ns. The skews on the edges $(0, 1)$, $(1, 2)$, and $(2, 3)$ are depicted in Figure 10.11.

For the mode signals, in the first scenario, we observe the following: Since node 1 is ahead of nodes 0 and 2, node 1's mode signal is correctly set to 0 (slow mode) while node 0 and 2's mode signals are set to 1 (fast mode). Node 3 is unaware that node 1 is ahead since it only monitors node 2. By default, its mode signal is set to slow mode. Node 2 then advances its clock faster than node 3. When the gap between 2 and 3 is large enough node 3 switches to fast mode. This configuration remains until nodes 0 and 2 catch up to 1, where they switch to slow mode, to not overtake node 1. Again node 3 sees only node 2 which is still ahead and switches to slow mode only after it catches up to 2.

One Node Behind. For scenario BEHIND the skews on the edges $(0, 1)$, $(1, 2)$, and $(2, 3)$ are depicted in Figure 10.13. We plot the absolute value of the skew, e.g., at roughly 500 ns node 1 overtakes node 0. The simulation shows that the algorithm immediately reduces the local skew. After the system reaches a small local skew after 200 ns we observe that nodes drift relative to each other, e.g., node 2 drifts ahead of node 3 and node 1 overtakes node 0. We point out that the local skew remains in the stable (oscillatory) state after 200 ns and does not increase significantly.

Gradient Skew. Simulation scenario GRADIENT outlines how the OffsetGCS algorithm works internally. It reduces the local skew in steps of (odd multiples of) κ, as seen in the plot in Figure 10.10, which looks like a staircase. The algorithm reduces skew on one edge at a time until it reaches the next plateau.

Figures 10.14 and 10.15 vividly show the behavior of the algorithm. Figure 10.15 is a direct reference to the depiction in Figure 6.2, it shows the behavior of a real simulation. OffsetGCS reduces the skew on edge $(1, 2)$ until it reaches a plateau with $(0, 1)$ and $(2, 3)$. One by one it then reduces skew on edges $(0, 1)$, $(1, 2)$ to $(2, 3)$ until

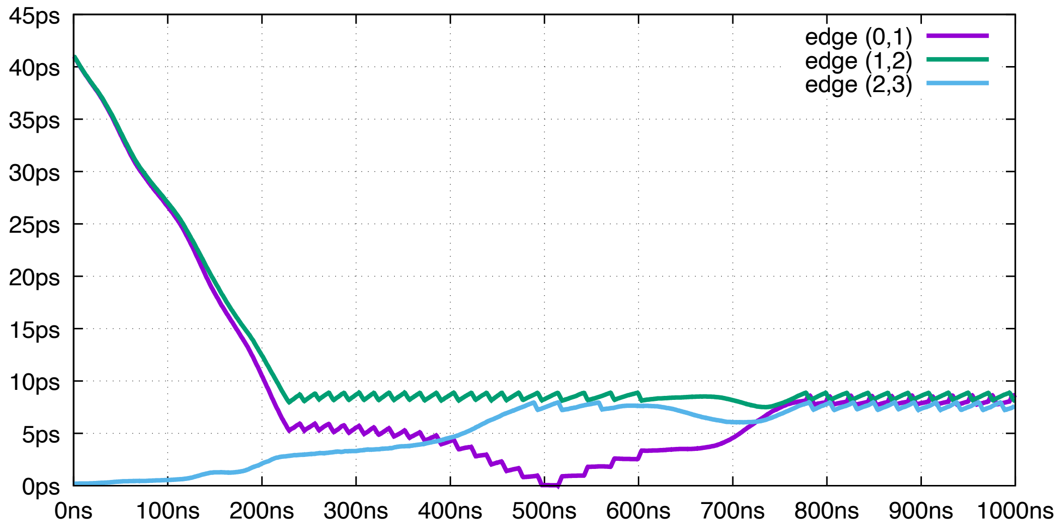

Figure 10.13: Skews in simulation of scenario BEHIND.

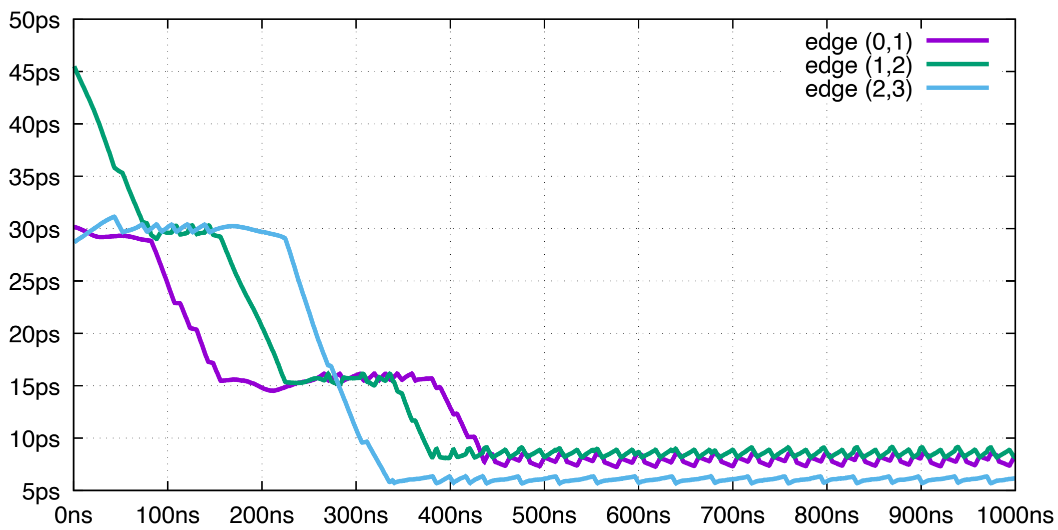

Figure 10.14: Skews in simulation of scenario GRADIENT.

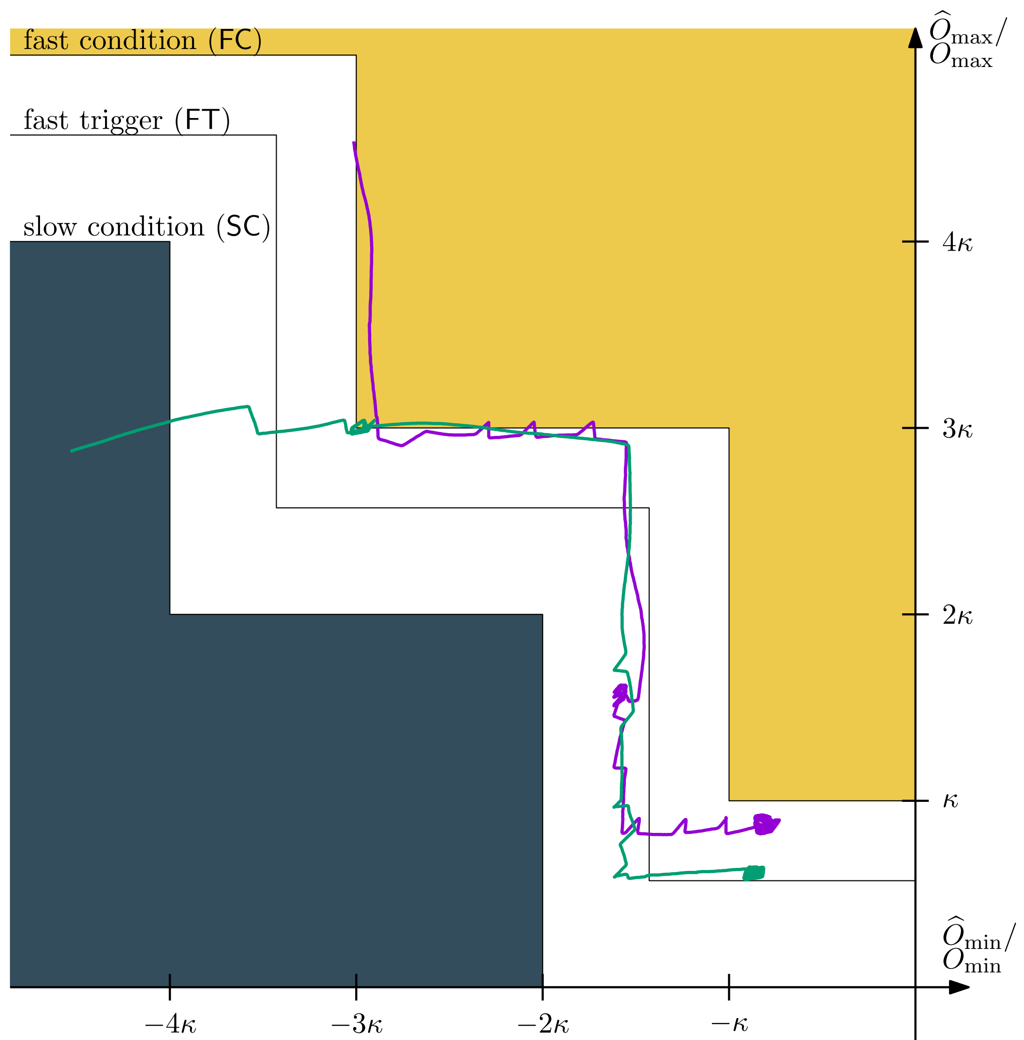

Figure 10.15: Trajectory of $\widehat{O}_{\max}$ and $\widehat{O}_{\min}$ of node 1 (purple) and node 2 (green).

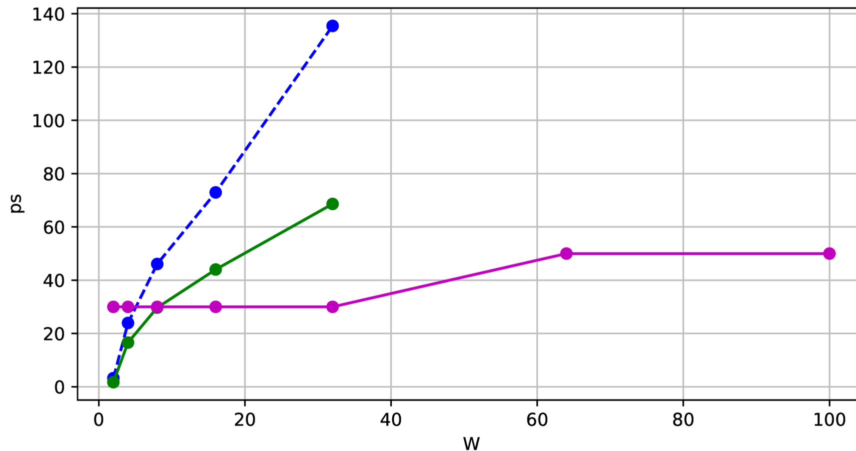

Figure 10.16: Local skew (ps) between neighboring flip-flops in the $W \times W$ grid. Clock tree with ±5% delay variation (solid green) and our algorithm with ±5% delay variation (solid magenta). The dotted line shows the clock tree with ±10% delay variation, demonstrating linear growth of the skew also in a different setting. Clock trees are shown up to $W = 32$ after which Encounter ran out of memory.

they reach the next plateau. Finally, it reduces the skews on the edges one by one (in reverse order) until it reaches a stable (oscillatory) state.

10.4.2 Comparison to Clock Tree

For comparison, we laid out a grid of $W \times W$ flip-flops, evenly spread in 200 µm distance in x and y direction across the chip. The data port of a flip-flop is driven by the `or` of up to four adjacent flip-flops. Clock trees were synthesized and routed with Cadence Encounter, with the target to minimize skews. Delay variations on gates and nets were set to ±5%. The results are presented in Figure 10.16. For comparison, we plotted skews guaranteed by our algorithm for the same grids with parameters extracted from the implementation described in Section 10.3. Observe the linear growth of the local clock skew measured in the simulation compared to the logarithmic growth of the analytical upper bound on the local skew in our implementation. The figure also shows the simulated skew for a clock tree with delay variations of ±10%. This comparison is relevant, as δ_0 is governed by *local* delay variations, which can be expected to be smaller than those across a large chip.

It is worth mentioning that it has been shown that no clock tree can avoid the local skew being proportional to W [40]. One can show that for *any* clock tree there are *always* two nodes in the grid that have local skew which is proportional to W. This follows from the fact that there are always two neighboring nodes in the grid

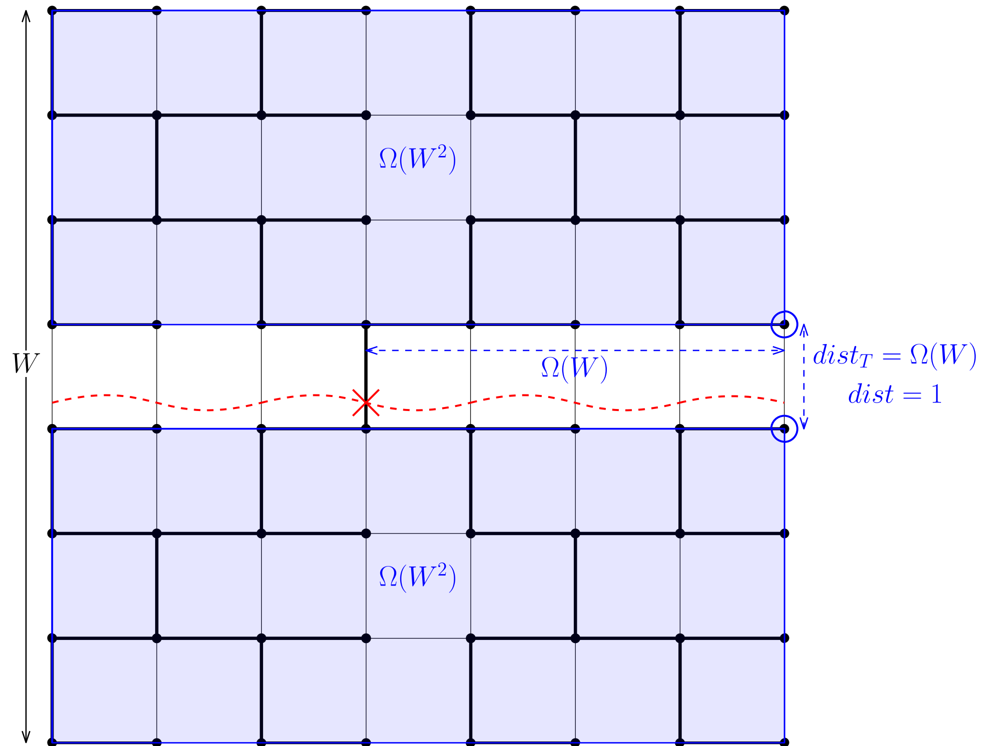

Figure 10.17: A low stretch spanning tree of an $W \times W$ ($W = 8$) grid [54]. The bold lines depict the spanning tree, i.e., our clock tree in this example. The two neighboring nodes that are of distance 13 in the tree are circled (at the middle right side of the grid).

which are in distance proportional to W from each other in the clock tree [40, 8]. Accordingly, uncertainties accumulate in a worst-case fashion to create a local skew which is proportional to W; this behavior can be observed in Figure 10.17.

In order to gain intuition on this result, note that there is always an edge that, if removed (see the edge which is marked by an X in Figure 10.17), partitions the tree into two subtrees each spanning an area of $\Omega(W^2)$ and hence having a shared perimeter of length $\Omega(W)$. Thus, there must be two adjacent nodes, one on each side of the perimeter, at distance $\Omega(W)$ in the tree.

Our algorithm, on the other hand, manages to reduce the local skew exponentially to being proportional to $\log W$.

10.4.3 Comparison to State of the Art

We compare our findings to state-of-the-art schemes for global clock generation. Simple approaches to the clock synchronization problem are *wait-for-all* and *wait-for-one*. A node produces its next clock tick once it receives a message from one (wait-for-one) or all (wait-for-all) of its neighbors. Each of the two approaches is vulnerable to non-uniform message delays. By analysis of the Fairbanks clock generation grid [38], we see that it uses a clever combination of wait-for-all and wait-for-one approaches. In the grid both approaches alternate for adjacent nodes. For a comparison, we simulated a digital abstraction of the Fairbanks clock generation grid. Based on ideas of the lower bound proof [39] we construct a simulation scenario that has a large local skew.

Fairbanks Clock Generation. We compare our system to the Fairbanks and Moore clock generation grid [38]. The clock generation grid is a self-timed analog circuit for global clocking. It is based on the *Dynamic asP* FIFO control by Molnar and Fairbanks [78].

The final version of the distributed clock generator is an analog implementation that involves rigorous transistor sizing and layout. In this work, we aim towards digital implementations as they are easier to adapt and manufacture for existing design processes. Instead of implementing the analog grid, we implement a digital abstraction of Dynamic asP as presented by the authors (cf. [38]).

Abstraction of Fairbanks. In the digital abstraction, we distinguish two types of nodes; pull-up nodes and pull-down nodes. We add a set-reset latch on every edge between two nodes. Pull-up nodes set the latch and pull-down nodes reset the latch. Pull-up nodes compute the logical `nor` of incoming edges (this is equivalent to a wait-for-all approach). Pull-down nodes compute the logical `and` of incoming edges (this is equivalent to a wait-for-one approach). We split the `and` gate into a `nand` gate and a `not` gate. An excerpt of the digital abstraction connected on a line is depicted in Figure 10.18. The clock of a node is derived by the output of the respective `nor` or `nand`. We adjust the frequency of the grid by adding a delay between each node and the set-reset latch, depicted as a buffer in the schematic.

Simulation Setup. We conducted simulations that examined the behavior at different communication delays. To simulate slower communication paths we add a small capacity (0.01 pF) to the communication channel. Both incoming and outgoing channels can be affected independently. For simplicity, we differentiate between two communication speeds; fast and slow.

Formally, the algorithm combining wait-for-all and wait-for-one approaches can have a local skew that grows asymptotically linear with the network diameter. Through our simulations we show a bad case in the Fairbanks implementation and that OffsetGCS can cope with this situation. The bad skew is achieved by a non-uniform setup of communication speeds. The setup of the simulated delays is depicted in

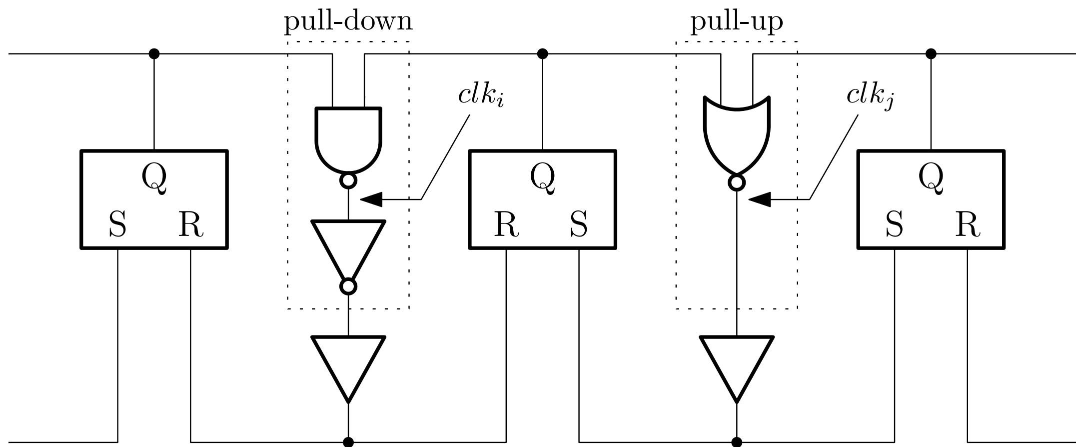

Figure 10.18: Digital abstraction of the Fairbanks clock generation scheme on a line.

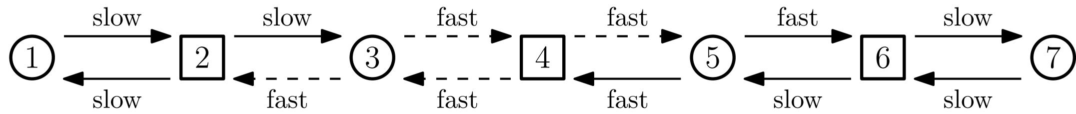

Figure 10.19: Delay setup that achieves a large local skew. Dashed edges are swapped from fast to slow communication. Round nodes denote pull-down nodes and rectangular nodes denote pull-up nodes in the Fairbanks implementation.

Figure 10.19. Only outgoing edges of nodes 3, 4, and 5 are fast and edges outgoing from 1, 2, 6, and 7 are slow.

Results for Fairbanks. The digital clock generation grid runs at a frequency of roughly 2.5 GHz. We measure that an additional capacity of 0.01 pF adds a delay of approximately 7 ps. We conducted simulations with three different delay settings. Simulation scenarios are called NOCAP, FULLCAP, and LARGE-LOCAL.

NOCAP simulates the Fairbanks grid without additional delays, i.e., with fast communication,

FULLCAP simulates the Fairbanks grid with capacity added to every edge, i.e., with slow communication,

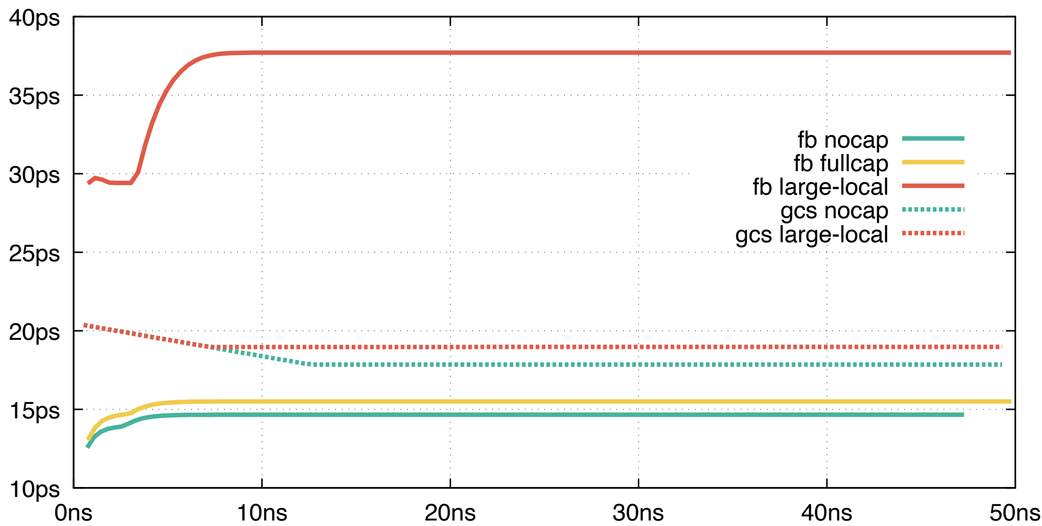

Figure 10.20: Local skews of the conducted experiments.

LARGE-LOCAL simulates the Fairbanks grid with the setting depicted in Figure 10.19, i.e., with non-uniform communication.

Local skews of simulation scenarios NOCAP, FULLCAP, and LARGE-LOCAL are shown in Figure 10.20. We observe that the Fairbanks grid achieves a very good skew if the delay is uniform on all edges. For scenario NOCAP (resp. FULLCAP) we measure a local skew of 15 ps (resp. 16 ps) and global skew of 22 ps (resp. 23 ps). Contrary, we observe that the Fairbanks grid experiences poor synchronization in case of non-uniform delays (LARGE-LOCAL). In this case, we measure a local skew of 38 ps and a global skew of 61 ps.

Remark. The setup in Figure 10.19 is tailored towards large local skews. Not every setup will achieve such a large skew. However, the setup is not purely artificial. Think of an ASIC where edges to outer nodes have higher delay due to manufacturing. Alternatively, think of a U-shape where nodes 1 and 7 are physically close together and close to an area that builds up more heat.

Results for GCS. The implementation of OffsetGCS runs at a frequency of roughly 2 GHz. We measure that the added capacity of 0.01 pF adds a delay of approximately 1 ps.

We observe that the OffsetGCS achieves a local skew of 18 ps and global skew of 20 ps if we do not add delay to the communication links. This is slightly worse than the local skew of the Fairbanks grid, as shown in Figure 10.20. However, the simulation shows that the OffsetGCS is not affected by the setting where delays in the

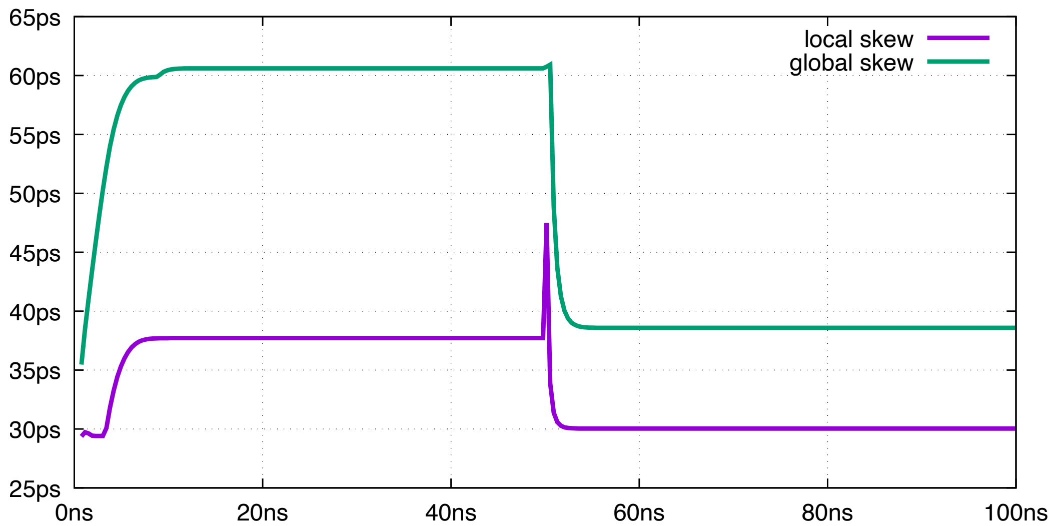

Figure 10.21: Simulation of Fairbanks with delays as described in Figure 10.19.

center are fast. We simulate our implementation also with the setting of Figure 10.19. Here we measure a local skew of 19 ps and a global skew of 20 ps.

Lower Bound Simulation. In this simulation, we apply ideas from the formal argument that shows lower bounds on wait-for-all and wait-for-one approaches. In short, the argument works by building up a large global skew in the system and then step by step changing the delay on the edges such that the global skew is pushed on one edge. The simulation in Figure 10.21 applies one of these steps. In the first part of the simulation (until 50 ns) we see that the system builds up a large local skew. At 50 ns we switch delays of edges outgoing from nodes 3 and 4 (as described in Figure 10.19). We observe that we can push even more skew on the edge with the largest local skew. We could measure a local skew of up to 47 ps.

The simulation shows that we can push (temporarily) more skew on one edge. Following the argument of the lower bound proof we claim that by repeated application of the process, we can push the global skew (temporarily) on one edge. Hence, the local skew increases significantly.

Simulation of the OffsetGCS with the same delay setup shows that switching edges from fast to slow, as for Fairbanks, has no effect on the local and global skew of the OffsetGCS implementation.

10.5 Follow-Up Questions

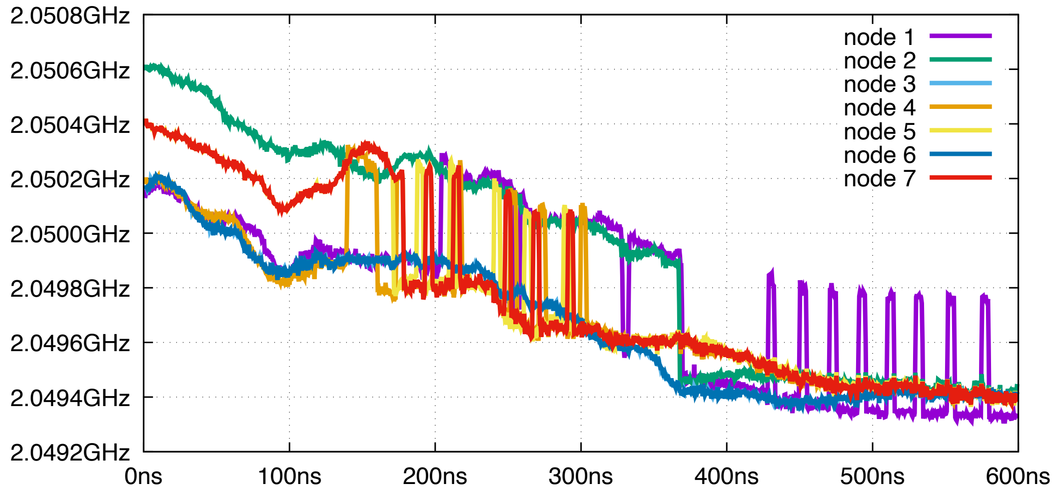

Figure 10.22: Measured oscillator frequency in a simulation of the line with 7 nodes.

Fast and Slow Mode. We conducted further simulations on the 7 node implementation. The experiment includes a higher simulation time (600 ns) and a huge initial skew (roughly 100 ps). During elaboration, we observed that the frequency of nodes drifts in a way such that fast mode becomes slower than slow mode in the beginning. We depict the measured frequency of the nodes in Figure 10.22. We observe for example when node 1 switches to fast mode (at roughly 430 ns) it has a lower frequency than in the very beginning, when it was in slow mode.

Formally, this means that the assumption $\mu > \rho$ (assumption (C5)) is not satisfied at all times. Still, we have a clear separation of fast and slow mode for small time windows. In future work, we want to adapt the formal requirements to allow for more flexible μ and ρ. If we make μ and ρ functions of time, then we can require assumption (C5) only for small time windows. This is out of the scope of this work as it requires adjusting the formal proof and analysis of the GCS algorithm. We are interested in the question of how the size of the time window affects the algorithm.

Oscillator Drift. In the implementation, we assume moderately stable oscillators with drift $\rho = \mu/10$. This is below a drift achievable with uncontrolled ring oscillators. In [28] the authors show that guardband margins that cover static and dynamic variability of an uncontrolled ring oscillator can be chosen lower than the drift of the oscillator. Process, voltage, and temperature variation affect delay lines of ring oscillators similarly to how they affect the delay of a critical path. According to the authors, the global variation dominates the local variation. We conclude that the

relative drift of oscillators on a chip is significantly smaller than the absolute drift of each oscillator.

We assume that we can achieve a better analysis when studying the relative drift of neighboring oscillators as opposed to the absolute drift of all oscillators. When regarding the drift of an oscillator as a function of time (similar as suggested above) and constraining the relative drift of neighboring oscillators by an upper bound we can adjust the model. This again requires adjusting the formal proof and analysis of the GCS algorithm.

Conclusions 11

In this chapter, we give a brief conclusion to this dissertation. First, we summarize the content highlighting the most important results and findings. Second, we recap open problems and give a vision for further work. We remind the reader, that in the introduction we stated our main research question as follows:

Can we design hazard-free circuits which implement clock synchronization algorithms?

11.1 Summary

The research question sums it up: We aim for an implementation of a clock synchronization algorithm. Every circuit on clock domain crossings, however, will have to care for hazardous signals.

Hazard-Free Circuits. It is well known that for every circuit there is a corresponding hazard-free circuit. Both implement the same Boolean function, but the hazard-free circuit also implements the extension to ternary (Kleene) logic. Unfortunately, there are circuits where the smallest hazard-free implementation is at least exponentially larger than the normal implementation. No general construction can avoid such an exponential blowup in circuit size. The general construction we present in Chapter 4 constructs a hazard-free circuit from a given finite state transducer. The size of the hazard-free circuit is parametrized by the size of the state space $|S|$, the size of the input encoding ℓ. The size of the output encoding is bounded by $\mathcal{O}(\lambda)$. A hazard-free circuit from the construction has size $\mathcal{O}((2^{3|S|} + 2^{2|S|+\ell}/\ell + 2^{\ell}\lambda)n)$ and depth $\mathcal{O}(|S| \log n + \ell)$. For small $|S|$, ℓ, and λ the construction yields a small hazard-free circuit.

Given a specific application, we might be able to find even smaller hazard-free circuits. Chapter 5 shows that we can find hazard-free circuits for the sorting primitive. Our circuit (`COMP`), which sorts two n-bit numbers, has asymptotically optimal size $\mathcal{O}(n)$ and depth $\mathcal{O}(\log n)$. The circuit can be plugged into optimal sorting networks, that are known from the literature. This yields a multi-input hazard-free sorting circuit.

The realization of the sorting primitive also demonstrates that encoding matters. In Kleene logic certain codes, like the Binary Code γ_n^{bin}, lack precision when unstable signals are involved. They lose information when applying the superposition function. In Chapter 3 we study encodings in Kleene logic. We identify favorable properties of encodings, namely k-preservation and k-recoverability, which quantify (by k) the amount of instability an encoding can handle. Intuitively, an n-bit encoding with higher recoverability has less codewords. We prove that recoverability requires redundancy, i.e., no recoverable code can have an optimal rate.

Clock Synchronization. The goal of a clock synchronization algorithm is to minimize the skew in a distributed system, where each node is associated with its own clock. We present in Chapter 6 two famous clock synchronization algorithms: the Lynch-Welch algorithm by Lundelius-Welch and Lynch, and the OffsetGCS algorithm by Lenzen, Locher, and Wattenhofer. Both have advantages and disadvantages as they pose different constraints to the network and whether faults are allowed. In Chapter 7 we show that we can allow faults in an arbitrary network, not just in cliques. If we run the Lynch-Welch algorithm on clusters of nodes, then we can run OffsetGCS on top and synchronize connected clusters. The combination of both achieves a small local skew in $\mathcal{O}(\log D)$.

The OffsetGCS algorithm alone achieves local skew in $\mathcal{O}(\delta \log_{\mu/\rho} D)$, where δ, ρ, and μ are constants and D is the diameter of the network. Constants δ and ρ are upper bounds on the message delay uncertainty and the oscillator drift, and μ is the speed up that we allow in the oscillator. Chapter 9 shows an implementation of OffsetGCS that achieves a small skew between two nodes, which process data in a producer-consumer fashion. The offset to the neighbor is estimated by the fill level of the data buffer. No explicit measurement module is needed. The clocked implementation achieves a small skew with a data buffer of size 2.

The implementation in Chapter 9 lays a foundation for the implementation of larger systems. In Chapter 10 we present the hardware implementation of OffsetGCS on arbitrary networks. It achieves synchronization of neighboring nodes in the order of a few inverter delays. The computation of a node executing the OffsetGCS algorithm can be visualized by Figure 6.1. An idealized trajectory during execution is depicted in Figure 6.2. Finally, Figure 10.15 shows the simulated behavior of two nodes in our implementation.

11.2 Vision

Network Synchronization. In this dissertation, we present an important step towards the application of clock synchronization algorithms in hardware. Chapter 10 shows a versatile implementation of the OffsetGCS algorithm on register-transfer level. We are able to compile an ASIC and simulate its physical behavior. For simulations, we ran SPICE, which is a software that uses differential equations to model the physical behavior.

Simulations were carried out for a simple system with a few nodes. They verify that the implementation behaves as predicted. The systems we simulated were graphs with four and seven nodes connected on a line. We chose the line topology, as it allows us to study the worst-case behavior of similar systems because the worst-case skew in a system appears on the path representing the diameter of the system.

Further simulations of structures closer to real-world applications would be advisable for future work. In particular, grid-like structures will be of interest for microchips, which are usually organized in rectangular grid layouts. Unfortunately, we reached the limit of our capabilities with the simulations presented. SPICE simulations of

larger systems require significantly more computational power or time. Computer simulation of a more abstract model, e.g., the register-transfer layer would increase speed significantly. But, by virtue of the abstraction, we would neglect the effects of unstable signals. It is important to simulate unstable signals and late transitions as they pose a high risk on clock domain crossings.

We propose that further simulations be carried out closer to real hardware. For example, FPGA boards are a fast and cheap way to simulate algorithms in hardware. FPGAs are integrated circuits that can be configured by a hardware description language. They are easy to set up and versatile, for example, the control module can be transferred to an FPGA with low effort. However, our implementation comprises a tunable oscillator and a measurement module, which are likely to cause problems. Depending on the FPGA board, implementation of one may break certain design constraints as they do not follow standard combinatorial or synchronous design rules. In some cases, an FPGA board may already offer a tunable oscillator. However, the oscillator can only be used if it meets the constraints in Section 8.2. Furthermore, when using multiple FPGA boards to simulate the implementation, then off-chip communication will (most probably) pose a large δ. Hence, the upper bound on the skew will become worse. An FPGA implementation has to care for many constraints. Eventually, future work has to take the effort of building an ASIC for testing. The advantages of OffsetGCS can be played best "on chip". When we have a large μ/ρ ratio and small δ, then OffsetGCS achieves a small skew.

field programmable gate array (FPGA)

Gradient Clock Synchronization. In Section 10.5 we point out that there is a mismatch of the formal assumptions and the simulated behavior of our implementation. The formal analysis assumes that at all times an oscillator in slow mode is never faster than an oscillator in fast mode. In the simulation, we observe that the frequency drifts over time, as the system stabilizes. Still, the implementation maintains a clear separation of fast and slow modes within a smaller time frame.

A natural progression of this work is to adapt the formal model to the observed behavior. This case concerns the computation of the logical clock. We propose to replace constants μ and ρ by according functions of time $\mu(t)$ and $\rho(t)$. Then the constraint $\mu > \rho$ can be replaced by $\mu(t) > \rho(t)$ for t in a (small) time frame. The proposed change allows for the observed behavior while maintaining the requirement of separating fast and slow modes. For an analysis of the algorithm in the new model, it could be helpful to also define bounds on $\mu(t)$ and $\rho(t)$, for example $\mu_{\min}$ and $\rho_{\max}$. We conjecture that the logarithm bounding the skew gets a base of the form $\mu_{\min}/\rho_{\max}$.

Hazard-Free Addition. We also discussed the Lynch-Welch algorithm which has no full mc implementation yet. We show that sorting of measurements can be done in asymptotically optimal size and time. It remains open to compute the mean of two measurements. Measurements may contain unstable signals. Hence, it is important to define a hazard-free addition and division and circuits implementing this. Discussion

of the encoding, preservation, and recoverability lays a foundation to a hazard-free arithmetic. Considerably more work will need to be done to determine recoverable codes which also allow for fast addition algorithms. We imagine a hybrid encoding consisting of BRGC and snake-in-the-box codes. Further study of error-correcting codes could be advisable as we saw similar approaches in the field. Hazard-free addition is potentially of interest for more applications, as addition is a fundamental routine in all microprocessors.

List of Acronyms

ASIC	application-specific integrated circuit
BFS	Breadth-First Search
BRGC	Binary Reflected Gray Code
CAD	computer aided design
CMOS	complementary metal-oxide-semiconductor
CMUX	metastability-containing multiplexer
DARTS	Distributed Algorithms for Robust Tick Synchronization
DNF	disjunctive normal form
FC	fast condition
FIFO	first-in first-out pipeline
FPGA	field programmable gate array
FT	fast trigger
GALS	globally asynchronous locally synchronous
GCS	gradient clock synchronization
GLIFT	gate level information flow tracking
LDL	locally delayed latching
mc	metastability-containing
MTBF	mean time between failures
MUTEX	mutual exclusion
MUX	multiplexer
OCL	open cell library
ODE	ordinary differential equation
PFD	phase frequency detector
PLL	phase locked loop
PPC	parallel prefix computation
SC	slow condition
SPICE	simulation program with integrated circuit emphasis
TDC	time to digital converter
VCO	voltage controlled oscillator
VHDL	VHSIC hardware description language

Bibliography

[1] Abadian, A., Lotfizad, M., Majd, N. E., Ghoushchi, M. B. G., and Mirzaie, H. A new low-power and low-complexity all digital PLL (ADPLL) in 180nm and 32nm. In *17th IEEE International Conference on Electronics, Circuits, and Systems, ICECS 2010, Athens, Greece, 12-15 December, 2010* (2010), IEEE, pp. 305–310.

[2] Abidi, A. A. Phase noise and jitter in CMOS ring oscillators. *IEEE J. Solid State Circuits 41*, 8 (2006), 1803–1816.

[3] Ajtai, M., Komlós, J., and Szemerédi, E. An o(n log n) sorting network. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA* (1983), D. S. Johnson, R. Fagin, M. L. Fredman, D. Harel, R. M. Karp, N. A. Lynch, C. H. Papadimitriou, R. L. Rivest, W. L. Ruzzo, and J. I. Seiferas, Eds., ACM, pp. 1–9.

[4] Bashir, A., Li, J., Ivatury, K., Khan, N., Gala, N., Familia, N., and Mohammed, Z. Fast lock scheme for phase-locked loops. In *IEEE Custom Integrated Circuits Conference, CICC 2009, San Jose, California, USA, 13-16 September, 2009, Proceedings* (2009), IEEE, pp. 319–322.

[5] Beer, S., and Ginosar, R. Eleven ways to boost your synchronizer. *IEEE Trans. Very Large Scale Integr. Syst. 23*, 6 (2015), 1040–1049.

[6] Bellman, R. On a routing problem. *Quarterly of applied mathematics 16*, 1 (1958), 87–90.

[7] Biaz, S., and Welch, J. L. Closed form bounds for clock synchronization under simple uncertainty assumptions. *Inf. Process. Lett. 80*, 3 (2001), 151–157.

[8] Boksberger, P., Kuhn, F., and Wattenhofer, R. On the approximation of the minimum maximum stretch tree problem. *Technical report/ETH, Department of Computer Science 409* (2003).

[9] Brent, R. P., and Kung, H. T. A regular layout for parallel adders. *IEEE Trans. Computers 31*, 3 (1982), 260–264.

[10] Brzozowski, J. A., Ésik, Z., and Iland, Y. Algebras for hazard detection. In *31st IEEE International Symposium on Multiple-Valued Logic, ISMVL 2001, Warsaw, Poland, May 22-24, 2001, Proceedings* (2001), IEEE Computer Society, pp. 3–14.

[11] Brzozowski, J. A., Ésik, Z., and Iland, Y. Algebras for hazard detection. In *31st IEEE International Symposium on Multiple-Valued Logic, ISMVL 2001,*

Warsaw, Poland, May 22-24, 2001, Proceedings (2001), IEEE Computer Society, pp. 3–14.

[12] Bund, J., Függer, M., Lenzen, C., and Medina, M. Synchronizer-free digital link controller. *IEEE Trans. Circuits Syst. 67-I*, 10 (2020), 3562–3573.

[13] Bund, J., Függer, M., Lenzen, C., Medina, M., and Rosenbaum, W. PALS: plesiochronous and locally synchronous systems. In *26th IEEE International Symposium on Asynchronous Circuits and Systems, ASYNC 2020, Salt Lake City, UT, USA, May 17-20, 2020* (2020), IEEE, pp. 36–43.

[14] Bund, J., Függer, M., Lenzen, C., Medina, M., and Rosenbaum, W. PALS: plesiochronous and locally synchronous systems. *CoRR abs/2003.05542* (2020).

[15] Bund, J., Lenzen, C., and Medina, M. Near-optimal metastability-containing sorting networks. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017* (2017), pp. 226–231.

[16] Bund, J., Lenzen, C., and Medina, M. Optimal metastability-containing sorting via parallel prefix computation. *IEEE Trans. Computers 69*, 2 (2020), 198–211.

[17] Bund, J., Lenzen, C., and Medina, M. Small hazard-free transducers. In *13th Innovations in Theoretical Computer Science Conference, ITCS 2022, January 31 - February 3, 2022, Berkeley, CA, USA* (2022), M. Braverman, Ed., vol. 215 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 32:1–32:24.

[18] Bund, J., Lenzen, C., and Rosenbaum, W. Fault tolerant gradient clock synchronization. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019* (2019), P. Robinson and F. Ellen, Eds., ACM, pp. 357–365.

[19] Bund, J., Lenzen, C., and Rosenbaum, W. Fault tolerant gradient clock synchronization. *CoRR abs/1902.08042* (2019).

[20] Bundala, D., and Zavodny, J. Optimal sorting networks. In *Language and Automata Theory and Applications - 8th International Conference, LATA 2014, Madrid, Spain, March 10-14, 2014. Proceedings* (2014), A. Dediu, C. Martín-Vide, J. L. Sierra-Rodríguez, and B. Truthe, Eds., vol. 8370 of *Lecture Notes in Computer Science*, Springer, pp. 236–247.

[21] Chaney, T. J., and Molnar, C. E. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Trans. Computers 22*, 4 (1973), 421–422.

[22] Chapiro, D. M. Globally-asynchronous locally-synchronous systems. Tech. rep., Stanford Univ CA Dept of Computer Science, 1984.

[23] Chelcea, T., and Nowick, S. M. Robust interfaces for mixed-timing systems. *IEEE Trans. Very Large Scale Integr. Syst. 12*, 8 (2004), 857–873.

[24] Chung, C.-C., and Lee, C.-Y. An All-Digital Phase-Locked Loop for High-Speed Clock Generation. *IEEE Journal of Solid-State Circuits 38*, 2 (2003), 347–351.

[25] Coates, W. S., and Drost, R. J. Congestion and starvation detection in ripple fifos. In *9th International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC 2003), 12-16 May 2003, Vancouver, BC, Canada* (2003), IEEE Computer Society, pp. 36–45.

[26] Codish, M., Cruz-Filipe, L., Frank, M., and Schneider-Kamp, P. Twenty-five comparators is optimal when sorting nine inputs (and twenty-nine for ten). In *26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2014, Limassol, Cyprus, November 10-12, 2014* (2014), IEEE Computer Society, pp. 186–193.

[27] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009.

[28] Cortadella, J., Lupon, M., Moreno, A., Roca, A., and Sapatnekar, S. S. Ring oscillator clocks and margins. In *22nd IEEE International Symposium on Asynchronous Circuits and Systems, ASYNC 2016, Porto Alegre, Brazil, May 8-11, 2016* (2016), IEEE Computer Society, pp. 19–26.

[29] Daliot, A., Dolev, D., and Parnas, H. Self-stabilizing pulse synchronization inspired by biological pacemaker networks. *CoRR abs/0803.0241* (2008).

[30] Dally, W. J., and Tell, S. G. The even/odd synchronizer: A fast, all-digital, periodic synchronizer. In *16th IEEE International Symposium on Asynchronous Circuits and Systems, ASYNC 2010, Grenoble, France, 3-6 May 2010* (2010), IEEE Computer Society, pp. 75–84.

[31] Dobkin, R. R., Ginosar, R., and Sotiriou, C. P. High rate data synchronization in GALS socs. *IEEE Trans. Very Large Scale Integr. Syst. 14*, 10 (2006), 1063–1074.

[32] Dolev, D., Halpern, J. Y., and Strong, H. R. On the possibility and impossibility of achieving clock synchronization. *J. Comput. Syst. Sci. 32*, 2 (1986), 230–250.

[33] Dolev, D., Heljanko, K., Järvisalo, M., Korhonen, J. H., Lenzen, C., Rybicki, J., Suomela, J., and Wieringa, S. Synchronous counting and computational algorithm design. *J. Comput. Syst. Sci. 82*, 2 (2016), 310–332.

[34] Dolev, D., and Hoch, E. N. Byzantine self-stabilizing pulse in a bounded-delay model. In *Stabilization, Safety, and Security of Distributed Systems, 9th International Symposium, SSS 2007, Paris, France, November 14-16, 2007, Proceedings* (2007), T. Masuzawa and S. Tixeuil, Eds., vol. 4838 of *Lecture Notes in Computer Science*, Springer, pp. 234–252.

[35] Dolev, D., Lynch, N. A., Pinter, S. S., Stark, E. W., and Weihl, W. E. Reaching approximate agreement in the presence of faults. *J. ACM 33*, 3 (1986), 499–516.

[36] Dolev, S., and Welch, J. L. Self-stabilizing clock synchronization in the presence of byzantine faults. *J. ACM 51*, 5 (2004), 780–799.

[37] Eichelberger, E. B. Hazard detection in combinational and sequential switching circuits. *IBM J. Res. Dev. 9*, 2 (1965), 90–99.

[38] Fairbanks, S., and Moore, S. W. Self-timed circuitry for global clocking. In *11th International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC 2005), 14-16 March 2005, New York, NY, USA* (2005), IEEE Computer Society, pp. 86–96.

[39] Fan, R., and Lynch, N. A. Gradient clock synchronization. *Distributed Comput. 18*, 4 (2006), 255–266.

[40] Fisher, A. L., and Kung, H. T. Synchronizing large VLSI processor arrays. *IEEE Trans. Computers 34*, 8 (1985), 734–740.

[41] Friedrichs, S., Függer, M., and Lenzen, C. Metastability-containing circuits. *IEEE Trans. Computers 67*, 8 (2018), 1167–1183.

[42] Friedrichs, S., and Kinali, A. Efficient metastability-containing multiplexers. In *2017 IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2017, Bochum, Germany, July 3-5, 2017* (2017), IEEE Computer Society, pp. 332–337.

[43] Függer, M., Kinali, A., Lenzen, C., and Polzer, T. Metastability-aware memory-efficient time-to-digital converters. In *23rd IEEE International Symposium on Asynchronous Circuits and Systems, ASYNC 2017, San Diego, CA, USA, May 21-24, 2017* (2017), IEEE Computer Society, pp. 49–56.

[44] Függer, M., and Schmid, U. Reconciling fault-tolerant distributed computing and systems-on-chip. *Distributed Comput. 24*, 6 (2012), 323–355.

[45] Ghai, D., Mohanty, S. P., and Kougianos, E. Design of parasitic and process-variation aware nano-cmos RF circuits: A VCO case study. *IEEE Trans. Very Large Scale Integr. Syst. 17*, 9 (2009), 1339–1342.

[46] Goto, M. Application of logical mathematics to the theory of relay networks (in Japanese). *J. Inst. Elec. Eng. of Japan 64*, 726 (1949), 125–130.

[47] Graham, R. L., Knuth, D. E., and Patashnik, O. *Concrete mathematics - a foundation for computer science (2. ed.).* Addison-Wesley, 1994.

[48] Gray, F. Pulse code communication. us patent 2632058, 1953.

[49] Hamming, R. W. Error detecting and error correcting codes. *The Bell system technical journal 29*, 2 (1950), 147–160.

[50] Hu, W., Oberg, J., Irturk, A., Tiwari, M., Sherwood, T., Mu, D., and Kastner, R. On the complexity of generating gate level information flow tracking logic. *IEEE Trans. Inf. Forensics Secur. 7*, 3 (2012), 1067–1080.

[51] Huffman, D. A. The design and use of hazard-free switching networks. *J. ACM 4*, 1 (1957), 47–62.

[52] Ikenmeyer, C., Komarath, B., Lenzen, C., Lysikov, V., Mokhov, A., and Sreenivasaiah, K. On the complexity of hazard-free circuits. *J. ACM 66*, 4 (2019), 25:1–25:20.

[53] Jackson, S. J., and Manohar, R. Gradual synchronization. In *22nd IEEE International Symposium on Asynchronous Circuits and Systems, ASYNC 2016, Porto Alegre, Brazil, May 8-11, 2016* (2016), IEEE Computer Society, pp. 29–36.

[54] James, M. Linear solver in linear time.

[55] Jukna, S. Notes on hazard-free circuits. *SIAM J. Discret. Math. 35*, 2 (2021), 770–787.

[56] Kautz, W. H. Unit-distance error-checking codes. *IRE Transacions on Electronic Computers 7*, 2 (1958), 179–180.

[57] Khanchandani, P., and Lenzen, C. Self-stabilizing byzantine clock synchronization with optimal precision. *Theory Comput. Syst. 63*, 2 (2019), 261–305.

[58] Kleene, S. C., De Bruijn, N., de Groot, J., and Zaanen, A. C. *Introduction to metamathematics*, vol. 483. van Nostrand New York, 1952.

[59] Knuth, D. E. *The Art of Computer Programming, Volume III: Sorting and Searching.* Addison-Wesley, 1973.

[60] Kogge, P. M., and Stone, H. S. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. Computers 22*, 8 (1973), 786–793.

[61] Komarath, B., and Saurabh, N. On the complexity of detecting hazards. *Information Processing Letters 162* (2020), 105980.

[62] Kuhn, F., Lenzen, C., Locher, T., and Oshman, R. Optimal gradient clock synchronization in dynamic networks. *CoRR abs/1005.2894* (2010).

[63] Kuhn, F., Lenzen, C., Locher, T., and Oshman, R. Optimal gradient clock synchronization in dynamic networks. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010* (2010), A. W. Richa and R. Guerraoui, Eds., ACM, pp. 430–439.

[64] Kuhn, F., and Oshman, R. Gradient clock synchronization using reference broadcasts. In *Principles of Distributed Systems, 13th International Conference, OPODIS 2009, Nîmes, France, December 15-18, 2009. Proceedings* (2009), T. F. Abdelzaher, M. Raynal, and N. Santoro, Eds., vol. 5923 of *Lecture Notes in Computer Science*, Springer, pp. 204–218.

[65] Ladner, R. E., and Fischer, M. J. Parallel prefix computation. *J. ACM 27*, 4 (1980), 831–838.

[66] Lamport, L., and Melliar-Smith, P. M. Synchronizing clocks in the presence of faults. *J. ACM 32*, 1 (1985), 52–78.

[67] Lenzen, C., Locher, T., and Wattenhofer, R. Tight bounds for clock synchronization. *J. ACM 57*, 2 (2010), 8:1–8:42.

[68] Lenzen, C., and Medina, M. Efficient metastability-containing gray code 2-sort. In *22nd IEEE International Symposium on Asynchronous Circuits and Systems, ASYNC 2016, Porto Alegre, Brazil, May 8-11, 2016* (2016), IEEE Computer Society, pp. 49–56.

[69] Lenzen, C., and Rybicki, J. Self-stabilising byzantine clock synchronisation is almost as easy as consensus. *J. ACM 66*, 5 (2019), 32:1–32:56.

[70] Locher, T., and Wattenhofer, R. Oblivious gradient clock synchronization. In *Distributed Computing, 20th International Symposium, DISC 2006, Stockholm, Sweden, September 18-20, 2006, Proceedings* (2006), S. Dolev, Ed., vol. 4167 of *Lecture Notes in Computer Science*, Springer, pp. 520–533.

[71] Lundelius, J., and Lynch, N. A. A new fault-tolerant algorithm for clock synchronization. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing, Vancouver, B. C., Canada, August 27-29, 1984* (1984), T. Kameda, J. Misra, J. G. Peters, and N. Santoro, Eds., ACM, pp. 75–88.

[72] Lundelius, J., and Lynch, N. A. An upper and lower bound for clock synchronization. *Inf. Control. 62*, 2/3 (1984), 190–204.

[73] Lynch, N. A. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[74] Marino, L. R. General theory of metastable operation. *IEEE Trans. Computers 30*, 2 (1981), 107–115.

[75] Martins, M. G. A., Matos, J. M., Ribas, R. P., Reis, A. I., Schlinker, G., Rech, L., and Michelsen, J. Open cell library in 15nm freepdk technology. In *Proceedings of the 2015 Symposium on International Symposium on Physical Design, ISPD 2015, Monterey, CA, USA, March 29 - April 1, 2015* (2015), A. Davoodi and E. F. Y. Young, Eds., ACM, pp. 171–178.

[76] Máté, L. L., Das, S., and Chuang, H. Y. A logic hazard detection and elimination method. *Information and Control 26*, 4 (1974), 351–368.

[77] Mealy, G. H. A method for synthesizing sequential circuits. *The Bell System Technical Journal 34*, 5 (1955), 1045–1079.

[78] Molnar, C. E., and Fairbanks, S. M. Control structure for a high-speed asynchronous pipeline, Aug. 10 1999. US Patent 5,937,177.

[79] Moore, G. E. Gramming more components onto integrated circuits. *Electronics 38* (1965), 8.

[80] Mullins, R. D., and Moore, S. W. Demystifying data-driven and pausible clocking schemes. In *13th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC 2007), 12-14 March 2006, Berkeley, California, USA* (2007), IEEE Computer Society, pp. 175–185.

[81] Najvirt, R., and Steininger, A. How to synchronize a pausible clock to a reference. In *21st IEEE International Symposium on Asynchronous Circuits and Systems, ASYNC 2015, Mountain View, CA, USA, May 4-6, 2015* (2015), IEEE Computer Society, pp. 9–16.

[82] Pan, J., and Yoshihara, T. A Fast Lock Phase-Locked Loop Using a Continuous-Time Phase Frequency Detector. In *EDSSC* (2007), pp. 393–396.

[83] Polzer, T., Handl, T., and Steininger, A. A metastability-free multi-synchronous communication scheme for socs. In *Stabilization, Safety, and Security of Distributed Systems, 11th International Symposium, SSS 2009, Lyon, France, November 3-6, 2009. Proceedings* (2009), R. Guerraoui and F. Petit, Eds., vol. 5873 of *Lecture Notes in Computer Science*, Springer, pp. 578–592.

[84] Polzer, T., and Steininger, A. An approach for efficient metastability characterization of fpgas through the designer. In *19th IEEE International Symposium on Asynchronous Circuits and Systems, ASYNC 2013, Santa Monica, CA, USA, May 19-22, 2013* (2013), IEEE Computer Society, pp. 174–182.

[85] Rotem-Gal-Oz, A. https://arnon.me/wp-content/uploads/Files/fallacies.pdf.

[86] Sipser, M. *Introduction to the theory of computation.* PWS Publishing Company, 1997.

[87] SKLANSKY, J. Conditional-sum addition logic. *IRE Trans. Electron. Comput. 9*, 2 (1960), 226–231.

[88] SOKOLOV, D., MOKHOV, A., YAKOVLEV, A., AND LLOYD, D. Towards Asynchronous Power Management. In *FTFC* (2014), pp. 1–4.

[89] SRIKANTH, T. K., AND TOUEG, S. Optimal clock synchronization. *J. ACM 34*, 3 (1987), 626–645.

[90] STIBITZ, G. R. Binary counter. us patent 2307868, 1943.

[91] SUMAN, S., SHARMA, K., AND GHOSH, P. Analysis and design of current starved ring VCO. In *ICEEOT* (2016), pp. 3222–3227.

[92] SWARTZLANDER, E. E., AND LEMONDS, C. E., Eds. *Computer Arithmetic*, vol. I–III. World Scientific Publishing Co, 2015.

[93] TARAWNEH, G., FÜGGER, M., AND LENZEN, C. Metastability tolerant computing. In *23rd IEEE International Symposium on Asynchronous Circuits and Systems, ASYNC 2017, San Diego, CA, USA, May 21-24, 2017* (2017), IEEE Computer Society, pp. 25–32.

[94] TEEHAN, P., GREENSTREET, M. R., AND LEMIEUX, G. G. A survey and taxonomy of GALS design styles. *IEEE Des. Test Comput. 24*, 5 (2007), 418–428.

[95] WALDROP, M. M. More than moore. *Nature 530*, 7589 (2016), 144–148.

[96] YOELI, M., AND RINON, S. Application of ternary algebra to the study of static hazards. *J. ACM 11*, 1 (1964), 84–97.

[97] ZHOU, J., KINNIMENT, D., DIKE, C. E., RUSSELL, G., AND YAKOVLEV, A. On-chip measurement of deep metastability in synchronizers. *IEEE J. Solid State Circuits 43*, 2 (2008), 550–557.