Die approbierte Originalversion dieser Dissertation ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Asynchronous Logic in Real-Time Systems

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der technischen Wissenschaften

by

Dipl.-Ing. Markus Ferringer
Registration Number 0025578

to the Faculty of Informatics
at the Vienna University of Technology

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

The dissertation has been reviewed by:

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

(Prof. Dipl.-Ing. Dr. Gerhard Fohler)

Wien, 15.12.2011

(Dipl.-Ing. Markus Ferringer)

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Markus Ferringer
Bäuerlegasse 3/10, 1200 Wien

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

(Ort, Datum)

(Unterschrift Verfasser)

Kurzfassung

Es ist mittlerweile unbestritten, dass asynchrone Logik zahlreiche Vorteile im Vergleich zur herkömmlichen synchronen Logik hat. Ein zentrales Problem jedoch ist die schwierige Vorhersagbarkeit der zeitlichen Abläufe eines asynchronen Designs. Aufgrund eines fehlenden (hochpräzisen) Schwingquarzes hängt die tatsächliche Ausführungsgeschwindigkeit maßgeblich von Faktoren wie Umbgebungstemperatur und Versorgungsspannung ab, wobei bereits minimale Fluktuationen messbare Auswirkungen auf die Geschwindigkeit haben können. Klarerweise werden asynchrone Schaltungen daher als gänzlich ungeeignet für den Einsatz in Echtzeitsystemen angesehen. Diesem Umstand soll mit dem Projekt ARTS[1] (Asynchronous Logic in Real-Time Systems) entgegengewirkt werden, indem die genauen zeitlichen Charakteristika von ungetakteten Digitalschaltungen auf ihre Tauglichkeit für Echtzeitsysteme (und zwar speziell für das zeitgesteuerte Protokoll TTP) untersucht werden. Zu diesem Zwecke wird in dieser Arbeit ein geeignetes Zeitmodel entwickelt, welches neben deterministischen auch probabilistische Signallaufzeitvariationen modellieren kann. Darauf aufbauend wird ein sich automatisch auf den TTP Datenstrom kalibrierendes System entwickelt, welches eine geeignete (asynchrone) Zeitbasis für einen asynchronen TTP-Kontroller zur Verfügung stellt. Wie sich heraus stellt, sind unter allen Designalternativen jene mit linear rückgekoppelten Schieberegistern (LFSR) am besten für unsere Anforderungen geeignet. Um die Funktionsfähigkeit und Robustheit der vorgestellten Lösung zu demonstrieren, unterziehen wir das Design verschieden empirischen Tests, wie zum Beispiel Temperatur- und Spannungstests, und untersuchen die jeweiligen Auswirkungen auf Jitter und Frequenzstabilität.

In Verbindung mit den theoretischen Untersuchungen können einige sehr interessante Erkenntnisse im Zusammenhang mit zeitlicher Vorhersagbarkeit von asynchronen Schaltungen gemacht werden: Trotz der speziellen Eigenschaften des verwendeten Design-Stiles gibt es erheblichen datenabhängigen Signaljitter. Weiters wurde festgestellt, dass Herstellungsvariationen gravierenden Einfluss auf die Geschwindigkeit und Jittercharakteristika haben. Nichtsdestotrotz wirken sich diese Einflüsse nicht negativ auf die automatische Kalibrierung aus. Untersuchungen am fertigen und funktionierenden asynchronen TTP-Kontroller zeigen deutlich, dass es grundsätzlich möglich ist, asynchrone Logik für Echtzeitanwendungen — mit gewissen Einschränkungen — einzusetzen.

---

[1] Das Projekt ARTS mit der Projektnummer 813578 wird vom FIT-IT Programm des Österreichischen Ministeriums für Verkehr, Innovation und Technology (bm:vit, http://www.bmvit.gv.at/) finanziert.

Abstract

While asynchronous logic has many potential advantages compared to traditional synchronous designs, one of the major drawbacks is its unpredictability with respect to temporal behavior. Having no high-precision oscillator, a self-timed circuit's execution speed is heavily dependent on temperature and supply voltage. Small fluctuations of these parameters already result in noticeable changes of the design's throughput and performance. Without further provisions this jitter makes the use of asynchronous logic hardly feasible for real-time applications. In this work, which is part of project ARTS[2] (Asynchronous Logic in Real-Time Systems), we investigate the temporal characteristics of self-timed circuits regarding their usage in real-time systems, especially the Time-Triggered Protocol. We propose a timing model capable of dealing with deterministic as well as probabilistic timings caused — besides others — by PVT (process, voltage, temperature) variations, and elaborate self-adapting circuits which shall derive a suitable notion of time for an asynchronous TTP controller. Out of the proposed variants we find the simple LFSR (linear feedback shift register) implementation with rate correction most promising for our purposes. We further introduce and analyze the jitter compensation concept, which is a three-fold mechanism to keep the asynchronous circuit's notion of time tightly synchronized to the remaining communication participants. To demonstrate the robustness of our solution, we perform different tests and investigate their impact on jitter and frequency stability. These tests include, e.g., varying operating temperature, changing core supply voltage, and process variations among several devices of the same type.

The experiments in combination with the theoretical analysis reveal some interesting insights for the temporal behavior of self-timed circuits: Even though the used design style is strongly indicating, considerable data-dependent jitter effects can be identified. It also turns out that process variations significantly influence the jitter characteristics and performance of asynchronous circuits. Nevertheless, the proposed self-adaptive time reference generation circuit is capable of tolerating different temporal conditions. Measurements with the fully functional asynchronous TTP controller reveal that it is indeed possible to use asynchronous logic in real-time systems. However, there are some major limitations (especially for actively sending messages in a time-triggered system) that must be considered.

---

[2] The ARTS project received funding from the FIT-IT program of the Austrian Federal Ministry of Transport, Innovation and Technology (bm:vit, http://www.bmvit.gv.at/), project no. 813578.

For Daniela

Contents

1 Preface **1**
- 1.1 Motivation . . . 1
- 1.2 Design Methodologies . . . 3
  - 1.2.1 Synchronous Paradigm . . . 4
  - 1.2.2 Globally-Asynchronous Locally-Synchronous . . . 6
  - 1.2.3 Asynchronous Paradigm . . . 8
- 1.3 Asynchronous Circuit Design . . . 10
  - 1.3.1 Delay Models . . . 10
  - 1.3.2 Micropipelines . . . 14
  - 1.3.3 Two-Phase Dual-Rail . . . 15
  - 1.3.4 Application Examples . . . 18
- 1.4 Asynchronous Design Flow . . . 20
- 1.5 The Time-Triggered Protocol . . . 22
- 1.6 ARTS - Aims and Contributions . . . 25
  - 1.6.1 Contribution . . . 27
- 1.7 Chapter Organization . . . 28

2 Interfacing Asynchronous Circuits **31**
- 2.1 Related Work . . . 32
  - 2.1.1 Synchronization Techniques . . . 32
    - 2.1.1.1 Loosely-coupled . . . 32
    - 2.1.1.2 Strongly-coupled . . . 33
  - 2.1.2 Conversion Techniques . . . 34
- 2.2 Lockstep Conversion . . . 35
  - 2.2.1 LEDR to NCL . . . 35
  - 2.2.2 NCL to LEDR . . . 38
  - 2.2.3 Bundled Data as Consumer . . . 40
  - 2.2.4 Bundled Data as Producer . . . 40
- 2.3 Free Conversion . . . 42
  - 2.3.1 Implementation Alternative . . . 44
- 2.4 Experimental Results . . . 46
- 2.5 Chapter Notes . . . 48

3 Temporal Characteristics 51
- 3.1 Related Work . . . 52
  - 3.1.1 Jitter in Synchronous Circuits . . . 52
  - 3.1.2 Timing Analysis . . . 56
- 3.2 Jitter in QDI Circuits . . . 57
- 3.3 Circuit Timing . . . 60
  - 3.3.1 Data-Dependent Execution Jitter . . . 61
  - 3.3.2 Timing Variations . . . 62
    - 3.3.2.1 Interconnect Delay . . . 63
    - 3.3.2.2 Gate Delays . . . 64
- 3.4 Case Studies . . . 65
  - 3.4.1 Measurement Setup . . . 66
  - 3.4.2 Voltage-Temperature Characteristics . . . 67
  - 3.4.3 Data-Dependent Execution Jitter . . . 68
    - 3.4.3.1 4-bit Counter . . . 68
  - 3.4.4 16-bit Counter . . . 71
- 3.5 Chapter Notes . . . 72

4 Asynchronous Reference Time 73
- 4.1 Related Work . . . 74
  - 4.1.1 Design Options . . . 74
    - 4.1.1.1 Distributed Clocks . . . 75
    - 4.1.1.2 Self-Timed Oscillator Rings . . . 76
  - 4.1.2 Allan Variance . . . 78
  - 4.1.3 PVT Variations . . . 79
- 4.2 Implementation Concept . . . 80
  - 4.2.1 Requirements . . . 81
  - 4.2.2 Temporal Properties . . . 83
- 4.3 Implementation Details . . . 87
- 4.4 Experimental Results . . . 91
  - 4.4.1 Measurement Setup . . . 91
  - 4.4.2 Comparison . . . 92
  - 4.4.3 Temperature Tests . . . 96
  - 4.4.4 Supply Voltage Tests . . . 97
  - 4.4.5 Fabrication Variations . . . 99
    - 4.4.5.1 Simulation . . . 102
  - 4.4.6 Frequency Stability . . . 103
- 4.5 Chapter Notes . . . 104

5 Asynchronous TTP Controller 105
- 5.1 Implementation Details . . . 106
  - 5.1.1 Hardware . . . 106
  - 5.1.2 Software . . . 109

5.1.3 TTP Cluster 111
5.2 Limitations and Restrictions 114
5.3 Experimental Results 115
5.3.1 Passive Communication 115
5.3.2 Active Communication 118
5.3.3 Discussion 121
5.4 Chapter Notes 124

6 Conclusion and Outlook **127**

A Software Implementation **131**
A.1 Register Definitions 131
A.1.1 Status Register 131
A.1.2 Configuration Register 132
A.1.3 Timer Register 133
A.1.4 Control Register 133
A.1.5 LFSR Register 135
A.1.6 Asynchronous Timer Register 135
A.1.7 Timestamp Register 136
A.2 Message Descriptor List 137
A.3 Message Data Structure 137
A.4 TTP Application 138
A.5 Index to LFSR conversion 139

List of Figures

1.1 General flow control (handshaking) of logic circuits [20]. 3
1.2 The synchronous design paradigm. 4
1.3 Exemplary GALS system structure (Source: [83]). 7
1.4 GALS taxonomy (Source: [83]). 7
1.5 Example circuit to illustrate different delay models (Source: [81]). 10
1.6 Bundled Data (a) and Dual-Rail approach (b) (Source: [81]). 11
1.7 Four-Phase (a) and Two-Phase bundled data protocol (b) (Source: [81]). . 12
1.8 4-Phase Dual-Rail example waveform (a) and encoding (b) (Source: [81]). . 12
1.9 Micropipeline control structure without (a) and with data processing (b) (Source: [82]). 14
1.10 LEDR coding scheme (a) and exemplary LEDR waveform (b). 16
1.11 Example LEDR circuit structure. 17
1.12 ASYN design flow. 21
1.13 LEDR AND gate block diagram [19] (a) and technology mapping for Altera Cyclone II FPGA (b). 22
1.14 TTP-system structure [48]. 23
1.15 Time Division Multiple Access scheme. 23
1.16 ARTS system setup. 25
1.17 TTP-node block diagram. 26

2.1 Loosely coupled: 2-stage synchronizer (a), and stoppable clock (b). 33
2.2 Strongly coupled: Request/Data driven (a), and Globally Synchronous Locally Asynchronous (b). 34
2.3 Conversion interface for LEDR to NCL (a), and Petri-Net representation (b). 35
2.4 Synthesized conversion circuit. 37
2.5 NCL to LEDR block diagram (a) and single bit conversion circuit (b). . . . 38
2.6 NCL to LEDR Petri Net representation. 38
2.7 Conversion circuit for bundled data to NCL (a) and LEDR (b). 41
2.8 Converting single bits to LEDR without back-pressure. 42
2.9 Alternative circuit implementation (b). 44
2.10 Example waveform of synchronizer. 45
2.11 Simulation of LEDR to NCL conversion. 46

2.12 Simulation of NCL to LEDR conversion. 47
2.13 Simulation of NCL/LEDR to bundled data conversion. 47
2.14 Simulation of self-triggered single-bit to LEDR conversion. 47

3.1 Jitter classification scheme (Source: [85]). 53
3.2 Jitter manifestations: General (a), Timing jitter (b), Period jitter (c), and Cycle-to-Cycle jitter (d) (Source: [94]). 55
3.3 Timing model, example circuit. 61
3.4 Voltage, temperature, delay relationship for Cyclone II FPGA (a) and voltage dependent temperature coefficient Γ (b). 68
3.5 Post-layout simulation (left), FPGA measurements and Monte Carlo simulation (middle), and cycle-to-cycle jitter (right) of a 4-bit asynchronous counter's `cDone` signal. 69
3.6 Measurement vs. Monte Carlo simulations for non-optimized (a) and optimized (b) 16-bit counter. 71

4.1 Charlie Diagram (Source: [93]) (a) and Drafting Effect (Source: [85]) (b). . 77
4.2 TTP-specific Manchester encoding of frames. 81
4.3 Manchester code with sampling points (a), TTP-slots, resynchronization (b). 83
4.4 Illustration of all characteristic temporal system properties. 86
4.5 Basic structure of the time-reference generation circuit. 87
4.6 LFSR counting at half speed due to shadow registers S_i. 89
4.7 Optimized structure of the time-reference generation circuit with continuous rate correction. 90
4.8 Synchronization with continuous rate correction. 90
4.9 Exemplary histograms (a) and cycle-to-cycle jitter (b) for signal `cDone`. . . 93
4.10 Histograms for design without (a) and with (b) continuous rate correction for signal `ref-time`. 94
4.11 Allan Variance of signal `cDone` for counter and LFSR implementation with continuous rate correction (a) and signal `ref-time` for two exemplary baudrates (b). 95
4.12 Temperature vs. normalized execution speed (a) and LFSR-index cnt_{ref} vs. duration τ_{rep} of `ref-time` (b). 96
4.13 Cycle-to-Cycle jitter (left) and jitter histograms of signal `cDone` (right) under varying supply voltage. 98
4.14 Allan Variance of `ref-time` under varying supply voltage. 98
4.15 Box-and-whiskers plot of `cDone` signal for all 17 boards. 99
4.16 Voltage-speed relationship (a) with detailed measurements around 1.2V (b). 100
4.17 Jitter histograms for two exemplary boards. 101
4.18 Simulation histograms for three different speed-grades 102
4.19 Reference time histogram (left) and Allan Variance with and without resynchronization (right). 103

5.1 Block diagram of simplified asynchronous TTP controller. 106

5.2 Flow chart of the main controlling application software. 110
5.3 TTP cluster architecture, symbolic photo (with the courtesy of TTTech Computertechnik AG, source: http://www.tttech.com/products/ttp/). . . 112
5.4 TTP schedule (nodes, TDMA slots, TTP-round). 112
5.5 Bit-fields of X-frames for our configuration. 113
5.6 cnt_{ref} values for different baudrates with upper and lower bounds for correct message reception. 116
5.7 Relative error in percent of one bittime with 1 LSB upper/lower bounds. . 117
5.8 Exemplary jitter histogram for $275k$ for two cnt_{ref} values. 118
5.9 Scope screenshot of running TTP cluster (a) and jitter of SOF transition of synchronous and asynchronous node (b). 119
5.10 Absolute error made for $cnt_{ref} = 47$ (large slope) and $cnt_{ref} = 48$ (low slope) 120

List of Tables

3.1 Characteristic figures of example circuits. 66

4.1 Comparison of the implementation alternatives for the time reference generator. 92

5.1 Overview of the different function blocks (cf. Figure 5.1). 107
5.2 Semantics of the 16-bit data field for the host application. 113

A.1 Register overview for TTP extension module. 131
A.2 Bit-definitions of `status-reg`. 132
A.3 Bit-definitions of `config-reg`. 132
A.4 Bit-definitions of `timer-reg`. 133
A.5 Bit-definitions of `control-reg`. 134
A.6 Bit-definitions of `lfsr-reg`. 135
A.7 Bit-definitions of `asyn-timer-reg`. 136
A.8 Bit-definitions of `timestamp-reg`. 136

Chapter 1

Preface

> *In the beginning the Universe was created. This has made a lot of people very angry and been widely regarded as a bad move.*
>
> Douglas Adams

1.1 Motivation

Asynchronous circuit design techniques can provide economic solutions in cases where the traditional synchronous design is facing its limitations [89]. Still, however, industry in general is reluctant to consider asynchronous design a viable alternative in these cases. There are several reasons for this, one prominent being the common belief that asynchronous logic is not suitable for real-time applications due to its apparently unpredictable temporal behavior.

One of the most frequently cited statements in conjunction with asynchronous designs is that they operate as fast as they can, thereby achieving average case performance. Although this property clearly is one of the biggest advantages of asynchronous circuits, it also manifests as a major drawback when it comes to temporal predictability. Furthermore, asynchronous circuits' ability to adapt their operating speed to the respective environmental conditions directly translates into a variation of the hardware execution time for a given task. The origin is the inherent closed-loop flow control that allows for automatic adaptation of the operating speed, which stands in contrast to the rigid temporal (open-loop) control in synchronous systems. The seemingly undetermined behavior is due to a complex interaction of several factors — starting from the switching speed of each single transistor to the handshake interaction of entire functional blocks. All these aspects have to be considered when determining an asynchronous circuit's speed of operation. However, in a synchronous design we have to deal with all these effects as well. The main difference is that the synchronous paradigm divides the task of system design into two separate parts [20]: The timing analysis deals with low level aspects of digital circuits and provides the clock period as a result. The logic and functional design considers the behavior of the system at a high level of abstraction and uses the clock period as basic time unit. In this way synchronous systems have a clear separation/interface between those domains, which allows considering each part independently from the other — thereby simplifying the modeling of the entire system with respect to functionality and timing. However, this separation has its price, as it results in a waste of performance and reduced robustness. As a matter of fact, the temporal behavior of asynchronous circuits is by no means more undetermined than the synchronous one. After all, the underlying technology is the same. However, it is more complex to model, since asynchronous circuits do not have a clock signal that would enable a separation between logic design and timing analysis. At the same time this clearly leverages a higher potential for optimization and robustness.

Nowadays, to allow for minimal clock skew throughout the die and of course for maximum performance, lots of effort is invested in designing the clock distribution network properly [13, 35, 46, 58, 76]. Furthermore, highly sophisticated power saving mechanisms are implemented: Reducing the clock frequency in case of low CPU load, lowering the chip's supply/core voltage, and clock gating are only some examples. Again, asynchronous circuits have great potential to overcome at least some of the limiting issues of their synchronous counterparts. Regarding the enormous power dissipation of cutting-edge technology, one of the most promising properties of asynchronous designs might be that the underlying functional blocks are only running if there is indeed work to be done. Clearly, such power-saving behavior can only be achieved by a thorough system design, which is — generally speaking — also one of the major drawbacks of asynchronous circuits: The entire design process is far more complex compared to synchronous designs. This situation is further worsened by the fact that almost all available development tools are optimized for synchronous logic only. Also rapid prototyping with Field Programmable Gate Arrays (FPGA) still is difficult, as estimates on performance, chip size, and power dissipation are hardly representative for asynchronous systems. Nevertheless, a lot of academic research is conducted in this area, and various impressing asynchronous designs have already been realized (e.g., asynchronous processors [38, 92], cryptographic applications [84], or clockless crossbar switches [16], also refer to Section 1.3.4).

Simultaneously, while industry still seems to avoid this promising design alternative in their products, it is important to increase the reputation and acceptance of the asynchronous design paradigm, especially with respect to reliability, determinism, and predictability. In the course of this work we want to make a step in this direction by combining real-time systems and asynchronous logic. Our goal is to design an asynchronous communication controller for the Time-Triggered Protocol (TTP) that is able to communicate with a set of synchronous counterparts even under changing environmental conditions (such as voltage and temperature variations). In this context we elaborate a model for determining the timing behavior of a suitable asynchronous logic design style. As there is no high-precision reference clock available, we need to exploit the strict determinism and predictability of TTP to derive a feasible notion of time for the low-level

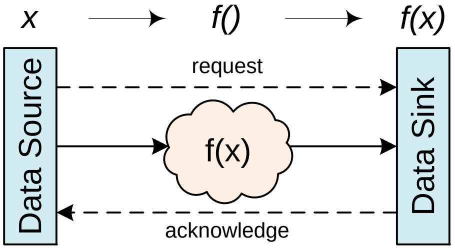


Figure 1.1: General flow control (handshaking) of logic circuits [20].

system services. Without a dedicated clock, however, the jitter of logic circuits becomes a major issue: If the execution cycles of the designs jitter too much, synchronization with TTP cannot be maintained. To this end, high-frequency jitter (as induced by classical jitter sources such as simultaneous switching noise or cross-talk) as well as low-frequency jitter (e.g., temperature drift or voltage fluctuations) must be addressed and compensated accordingly.

Before we start with a detailed description of our research project *Asynchronous Logic in Real-Time Systems (ARTS)*, the following sections provide a comprehensive summary of several important topics that are directly related to this work. In the next section we will introduce different logic design styles and point out their major benefits and drawbacks. Section 1.3 then describes delay-insensitive asynchronous design methodologies which we use for implementing our circuits. Before finally taking a closer look at the aims and contributions of the project ARTS in Section 1.6, Section 1.5 explains the basic concepts of the Time-Triggered Protocol.

1.2 Design Methodologies

When considering (real-world) logical circuits in general, one might notice that functional blocks almost always follow the simple data flow model [20] of Figure 1.1: There is a data source, which provides the input data x, a (boolean) function f which implements the desired functionality and maps x to the respective result $y = f(x)$, and a data sink which finally stores the computation result y. Although this basically is the definition of a simple mathematical function $f : x \mapsto f(x)$, there are some major issues to consider for practical electrical/digital systems, which severely complicate things compared to the strict mathematical formulation above. For the sake of simplicity let us assume that a data word x is represented as a vector of n bits $x_i, 0 \leq i < n$. The bit-widths of input x and output y do not necessarily have to be the same.

- *Acknowledgement:* The source must hold its current value until all processing has finished and the sink has successfully stored the result. Only then new data can safely be assigned to the source's output[1].

---

[1] For this overview we assume the combinational logic to obey the inertial delay model rather than, e.g., the transport delay model as supported by VHDL [91].

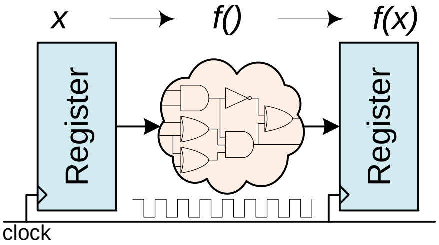


Figure 1.2: The synchronous design paradigm.

- *Request:* Furthermore, there must be a way to signal the arrival of new data to the sink — even if the result $f(x)$ does not change for two successive and probably different input words x. This is realized by means of *completion detection*: Assuming input vector x to be stable and consistent[2] at the source, the calculation of the result $y = f(x)$ needs some (variable) time Δ to finish. To be more precise, each bit $y_i, 0 \leq i < m$ of the resulting data vector y may need some slightly different amount of time until the final and correct value is reached. Consequently, the sink must have some means to determine a point in time when it is safe to store the entire calculation result, i.e., when all bits are stable and consistent. We will present different methodologies for implementing completion detection in Sections 1.2.1 and 1.3.3.

As one can imagine, there are many different ways to solve the above issues on the logical level [20]. The following paragraphs give an overview to existing and well-established digital design methodologies. We will see that the applied solutions are indeed quite diverse, and cannot easily be mixed or exchanged.

1.2.1 Synchronous Paradigm

In industry, synchronous designs are by far the most common ones. In a synchronous logic circuit, there is a periodic, globally available clock signal. This signal not only defines the speed of operation, but also solves the basic issues described above (illustrated in Figure 1.2). The desired functionality is implemented using "ordinary" boolean logic (which is considered "formally incomplete" [25] as it does not provide any means for expressing validity, consistency, or temporal relationships), whereas the data sinks and data sources are realized as edge-triggered registers. All registers of an isochronic region [3] are connected to a common, periodic clock signal, whose (usually positive) edges mark points in time where new data is assigned to all registers. The basic assumption for this mechanism to work properly is that the clock's signal transitions occur almost simultaneously all over the chip. In this context, the clock signal is a very simple way to solve all fundamental issues:

---

[2] Informally speaking, consistency means that all logical values represented by the single bits x_i of a data vector x belong the the *same* data word.

- In synchronous designs, there is no need for an explicit *acknowledgement* from the sink to the source, as the mere progression of time (i.e., a positive clock transition) guarantees proper storage of data at the sink.

- Likewise, an explicit *request* signal from the source to the sink can also be omitted as each positive clock transition is automatically interpreted as new request. *Completion detection* is moved to the time domain by setting the period of the clock signal to an appropriate duration for the logic function f to safely complete its operation and produce stable output. The data sink can thus be sure by design that its input is consistent and stable whenever a positive clock edge occurs.

All implementation and technology specific timing properties and constraints (as defined in [20], e.g.) are directly or indirectly masked by the concept of a global clock in the time domain. To put it in other words, the synchronous design methodology solves all “synchronization issues” between source and sink in the time domain rather than in the signal/information domain. This clearly has some remarkable advantages. To mention just a few, the resulting circuitry is very simple and efficient. Designers can focus on implementing the desired functionality instead of thinking about control flow, and on the logical level there is hardly any overhead. Furthermore, as the clock is usually generated with high-precision crystal oscillators, an accurate notion of time can be derived without additional effort. All in all, the synchronous design paradigm has proven itself very useful and efficient in many terms.

However, with every upside comes a downside as well. Especially the last few years have shown that it becomes more and more difficult to further increase the clock frequencies. With ever increasing performance requirements and technology scaling, not only the rapidly increasing power consumption, but also the design of the clock distribution network [35] itself become limiting factors. Without sophisticated compensation mechanisms a multi-GHz chip can no longer be considered isochronous, as clock skew becomes too large throughout the complex clock tree [35, 46]. Another issue related to the synchronous design style is reduced robustness against process variability and changing operating conditions, since they directly influence the execution speed of the underlying hardware. Consequently, appropriate safety margins need to be foreseen when specifying a system’s clock frequency. After all, the overall clock frequency is determined by a static worst case timing analysis. Yet another drawback comes at system or module interfaces: Systems running with different clocks (even if their nominal frequencies match) cannot easily be connected. Special synchronizer circuits must be implemented [40], thereby increasing not only complexity but possibly also reducing performance. While considerably simplifying the design process, decoupling the control flow from the data flow by means of a global time reference negatively impacts on performance, flexibility, and power consumption. It is not easily possible to switch off idle modules (i.e., zero-skew clock gating [13, 76]), or to combine modules with different performance.

1.2.2 Globally-Asynchronous Locally-Synchronous

A possible solution to the clock skew and distribution problems is the so called Globally-Asynchronous Locally-Synchronous (GALS) approach [14, 51, 83]: A complex design is built up of several independent functional units, each of which has its own clock signal. The relatively small blocks can be implemented using the conventional synchronous design approach with all its advantages (and without some of the disadvantages coming with technology scaling and increased system complexity). The communication among the synchronous blocks has, because of the independent clock sources, to take place asynchronously. Figure 1.3 illustrates a possible structure of a GALS design. The asynchronous interconnect is responsible for data transfer between the different synchronous blocks. As indicated in the figure, these blocks generally do not have the same operating frequency. Depending on the exact system setup, the following taxonomy of timing relationships between the different clock domains is commonly used in literature [61, 83]. This classification directly impacts the actual implementation of the asynchronous interconnect.

- A *mesochronous* relationship between two synchronous blocks means that both communication participants operate at exactly the same frequency. However, there is a stable but unknown phase difference between the two blocks. For example, blocks 1, 2 and 3 in Figure 1.3 share the same clock source. If the frequency multipliers are set to 1, these three nodes are said to be mesochronous.

- Assume that both clock sources in the figure have the same nominal frequency. Now, blocks 2 and 4 are said to be *plesiochronous.* While having the same nominal operating speed, the two independent clock sources are subject to drifting phases (e.g., due to minor frequency deviations in the range of a few parts per million).

- Finally, when a sender and receiver operate at totally different clock frequencies, the blocks are considered *heterochronous.* Assuming that both clock sources and frequency multipliers are set to different values, all four blocks are heterochronous with respect to each other.

The latter class of heterochronous systems can further be subdivided into *ratiochronous* and *nonratiochronous* designs. Again consider Figure 1.3: Blocks 1, 2 and 3 share the same clock source, but nodes 1 and 3 have frequency multipliers and additional delays in their clock path. As the frequency multipliers are also fed by Clock Source 1, there is a predictable and periodic relationship between the different clocks' phases. This relationship is called ratiochronous (as the clocks are exact rational multiples of each other) and can be exploited when designing the asynchronous interconnect.

According to Figure 1.4, all the above "synchrony characteristics" belong to the superior class of *loosely synchronous* GALS designs. However, a more distinctive classification can be made due to the different possible GALS design styles.

- *Loosely Synchronous.* This design style exploits the known relationships between the operating frequencies of communicating blocks in order to optimize throughput

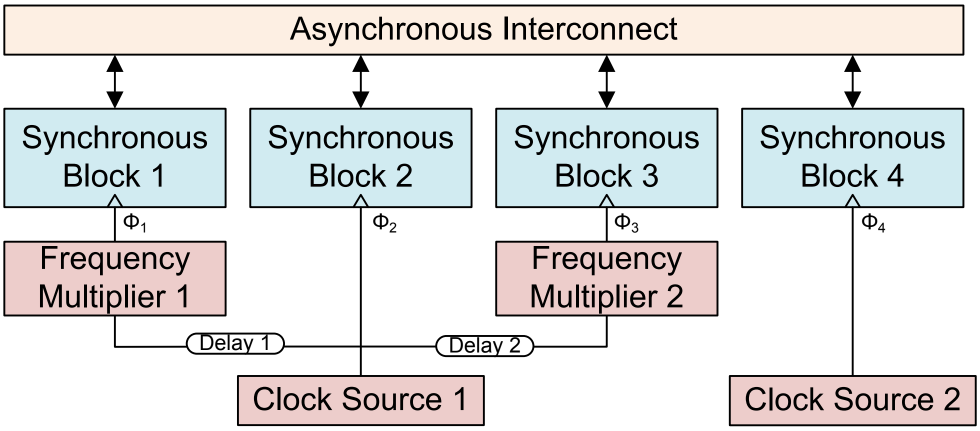


Figure 1.3: Exemplary GALS system structure (Source: [83]).

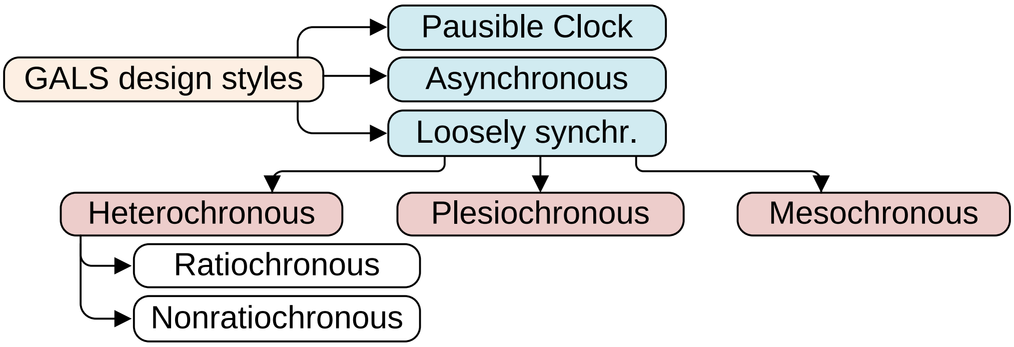


Figure 1.4: GALS taxonomy (Source: [83]).

and latency, as well as to ensure that all timing constraints are met. While allowing for high efficiency (for both area consumption and performance), this style suffers flexibility as changes in the clock frequencies cannot be handled without a redesign.

- *Asynchronous.* This style represents the most flexible way of communication between different clock domains. Usually, explicit *request* and *acknowledge* signals are used for implementing flow control in asynchronous systems (see Section 1.3 for more details). It is not necessary to make any assumptions on the relationships of the respective clock domains. However, this advantage comes with the severe drawback of reduced throughput and relatively high latency.
- *Pausible Clock.* In this design style the clocks are usually generated locally by means of (pausible and stretchable) ring oscillators. While data transmission is active, the clock can be paused, thus totally avoiding potential metastability. While pausing the clock introduces performance penalties on the respective synchronous block, robustness and power consumption are the main benefits of this approach (no dynamic power is consumed while the clock is paused).

While the idea of pausible clocks sounds reasonable and efficient, the main problem with industrial designs are the ring oscillators. As they are strongly sensitive to operating temperature, supply voltage and process variations, they need careful (and thus expensive) calibration. A general issue with all GALS approaches is that the asynchronous interconnect will degrade system performance due to the overhand introduced by the handshaking protocols. On the other hand, GALS designs have great potential when it comes to power reduction, as it is easily possible to set the supply voltage and operating frequency of each synchronous block independently. Furthermore, the single blocks may also save some power due to the reduced complexity of the clock distribution network. Another important advantage especially for industrial designs originates in reduced electromagnetic interference. The different (possibly uncorrelated) clock domains may significantly reduce signal noise because register switching is distributed in time, thus leading to a flatter power spectrum. This can also weaken the requirements on the power supply network and the needed number of power pads. Interestingly, a flatter power spectrum (caused by different uncorrelated clocks) may also increase immunity against power analysis attacks, thus rendering GALS designs useful especially for cryptographic applications [42].

To summarize one can say that GALS designs offer an interesting alterative to the ordinary, strictly synchronous design approach. They combine some possible advantages of asynchronous designs (e.g., power consumption, modularity, etc.) with the comfort of synchronous circuit design. As system integration and technology scaling further advance, one can expect GALS design style — besides other alternative digital design methodologies — to gain more importance in future.

1.2.3 Asynchronous Paradigm

Today's mostly synchronous logic is based upon two assumptions: First, signals are limited to binary values, and second, it is assumed that time is discrete. The former allows for the use of simple boolean logic, whereas the latter facilitates that hazards and feedbacks can mostly be ignored [43]. For asynchronous circuits, these central assumptions are basically removed: On the one hand, replacing the global clock with local handshaking protocols allows events to happen at any time (they are not triggered by clock-edges any more). On the other hand, multi-rail encodings are often used to represent data in different code-sets and allow for efficient completion detection. Appropriate examples for such encodings are presented in Section 1.3. By changing the logic design paradigm, some interesting potential benefits can be identified [43, 81, 89]:

- Since there is no (global) clock signal, *clock skew* and all related problems (e.g., clock distribution) can be ignored by definition.
- Without a clock signal, unused modules are not clocked and therefore need less power. The overall *power consumption* can benefit from an asynchronous design, without implementing dedicated power saving mechanisms.
- The clock frequency in synchronous designs results from the critical path's propagation delay, which is the worst case performance. Even worse, steadily increasing process and fabrication variations result in a very pessimistic estimation of the worst case timing. In contrast, asynchronous systems generally do not need to wait for any clock transition if a task is finished. This *average case performance* may lead to considerable improvements in speed.
- Not having a system clock, asynchronous circuits need alternative means for completion detection. These circuits often have the benefit of automatically adapting themselves to changing physical and environmental conditions (e.g., temperature drift, supply voltage fluctuations, fabrication process variations).
- The often problematic *electromagnetic emissions* generated by thousands of almost simultaneously switching registers are significantly reduced in asynchronous designs. The reason for this behavior is based on the fact that signal transitions tend to occur without synchronization to a global clock.
- The simple handshake interfaces allow for better *composability and modularity.*

Unfortunately, many asynchronous design styles have a significant overhead in area consumption. This is mainly caused by the — compared to synchronous logic — relatively complex control structures (e.g., completion detection, handshaking, etc.), and may also negatively influence system performance. From an engineer's point of view, the lack of sophisticated CAD tools that actually support asynchronous design methodologies is one of the most severe drawbacks. Another major issue concerns testability, as asynchronous circuits cannot simply be stopped and started by means of clock transitions. Design for testability is thus far more complex and thereby increases the overall costs dramatically.

It is undoubted that asynchronous circuit design has great potential and offers a lot of new possibilities to hardware engineers. It is important to notice that the existing

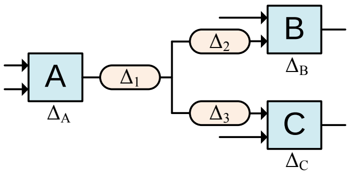


Figure 1.5: Example circuit to illustrate different delay models (Source: [81]).

asynchronous design styles (some examples are presented later in this chapter) all have strengths and weaknesses of their own. The challenge is to find solutions that combine most of the above properties, and simultaneously keep the design and resource overheads low.

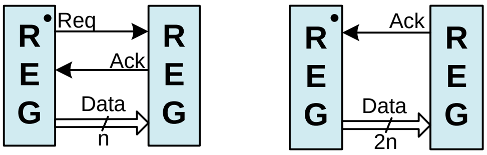
1.3 Asynchronous Circuit Design

In the last sections we have discussed some abstract design methodologies for digital circuits. We have seen that asynchronous logic offers interesting properties and opportunities to design engineers. In this section, we now take a closer look to asynchronous logic design and its underlying principles. We present the most common delay models, exemplary asynchronous design techniques, as well as various asynchronous application examples. Furthermore, we discusses the design style used for our implementations in more detail.

1.3.1 Delay Models

There are four major delay models to classify asynchronous circuits [43, 65, 81]. These models form the very basis of each design (asynchronous as well as synchronous) because they make fundamental assumptions about signal, gate, and wire delays. Clearly, these assumptions have to be based upon the physical properties of the target technology, and if they are violated the circuit will most likely not work at all. To better illustrate the different delay models, Figure 1.5 shows a simple circuit consisting of three gates A, B, C with associated propagation delays $\Delta_{A,B,C}$ and the corresponding interconnect with delays $\Delta_{1,2,3}$, respectively.

- *Self-Timed.* Also often called bounded delay model, the timing characteristics of digital circuits are modeled on an elaborate engineering level. The propagation delays of wires and gates are defined to be bounded *and known.* This model therefore is the basis for all synchronous designs, as a sophisticated critical path timing analysis is only possible with detailed knowledge of the technology's timing properties. For the example in the figure this means that all delays have well-defined values. The worst case propagation delay can therefore be easily expressed as $\Delta_A + \Delta_1 + \max(\Delta_2 + \Delta_B, \Delta_3 + \Delta_C)$.



(a) (b)

Figure 1.6: Bundled Data (a) and Dual-Rail approach (b) (Source: [81]).

- *Speed-Independent (SI).* These types of asynchronous logic are also known as Muller circuits. The delay model assumes that while gate delays are bounded but unknown, wire delays are negligible, i.e., zero. For Figure 1.5 this requires all three Δ_A, Δ_B, and Δ_C to arbitrary (but greater than zero), and $\Delta_1 = \Delta_2 = \Delta_3 = 0$.

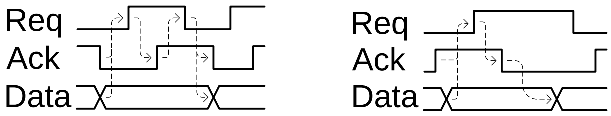
- *Quasi Delay-Insensitive (QDI).* By going one step further and allowing also arbitrary Δ_1 and Δ_2, but at the same time requiring $\Delta_2 = \Delta_3$, we obtain the so called Quasi Delay-Insensitive (QDI) model. Since wire delays can not be considered negligible (in fact they already dominate gate delays significantly), the QDI model makes more realistic assumptions on wire delays (compared to SI). By introducing isochronic forks, which have the property that signal transitions reach all end-points at the same time, the gap between SI and DI (see below) circuits is closed.

- *Delay-Insensitive (DI).* An extremely robust delay model is obtained by allowing all delays for gates and wires to be unbounded, finite, and unknown. For Figure 1.5 this leads to arbitrary (positive) values for all the delays $\Delta_{A,B,C,1,2,3}$. As the designer cannot make any predictions on the actual delays, validity of data must be implemented in the value domain rather than in the time domain (as it is done in synchronous designs, e.g.). The weak assumptions made by this design model result in a very limited number of truly DI circuits, because only C-elements [75] and inverters may be used as building blocks [59].

As we will see in Section 1.3.3, delay-insensitive circuits built out of usual boolean gates (such as AND or XOR) are, strictly speaking, only quasi delay-insensitive: The basic building blocks (e.g., delay-insensitive boolean gates) are modeled on a low abstraction level, where the designer can directly control the wire delays. All necessary timing assumption are hidden inside these basic low-level blocks. A complex circuit composed of such gates can indeed be considered delay-insensitive, because all timing dependencies are masked at a higher level of abstraction.

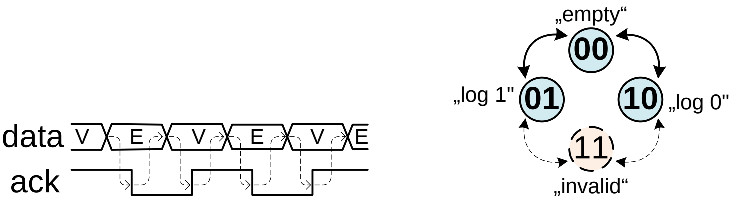
The following few paragraphs will provide an overview over the most common types of asynchronous (handshake) protocols [81]. In Figure 1.6, the initiator (sender) is marked with a black dot.

Bundled Data. In this single-railed approach data values are encoded as usual boolean values. All bits of an entire data-vector are bundled, and associated with a pair of `request` and `acknowledge` signals. These control signals are used to communicate



(a) (b)

Figure 1.7: Four-Phase (a) and Two-Phase bundled data protocol (b) (Source: [81]).



(a) (b)

Figure 1.8: 4-Phase Dual-Rail example waveform (a) and encoding (b) (Source: [81]).

with handshaking when new data is available (`request`) at the sender, and when data has been captured (`acknowledge`) by the receiver (cf. Figure 1.6(a)). In order to exchange n bits of data between sender and receiver, $n+2$ signal wires are necessary. There are two possible implementation alternatives for the actual handshaking protocol. As illustrated in Figure 1.7, the *4-phase* protocol works in *return-to-zero (RTZ)* fashion using state signaling: The sender prepares the data and sets its `request`. The receiver captures the data and asserts is `acknowledge` line. In response, the sender de-asserts `request`, followed by resetting `acknowledge` to its initial value. Obviously, the last two transitions (both `request` and `acknowledge` return to zero) cost unnecessary time and energy. This can be overcome using the *two-phase protocol* depicted in Figure 1.7(b), which is based on transition signaling and therefore also called *non-return-to-zero (NRZ)* protocol. Again, the sender prepares the data which shall be transmitted and generates a transition on the `request` line. After the receiver has captured the data, it sends a transition back on `acknowledge`, which finalizes the data transfer. Although this method in principle saves time and energy, logic responding to signal events (rather than signal states) is considerably more complex. Both these handshaking protocols have in common that they rely on the correct order of events (at both sender and receiver[3]), as the data-vector needs to be stable before the receiver starts capturing. Clearly, neither the SI nor (Q)DI delay model can guarantee this prerequisite, leaving self-timed the only option for the bundled data

Dual-Rail. In contrast to single-rail encoding, a logical value is represented by two physical signal lines in dual-rail circuits. As four different states can be encoded with two binary signals, it is possible to move part of the handshaking (and of the associated

---

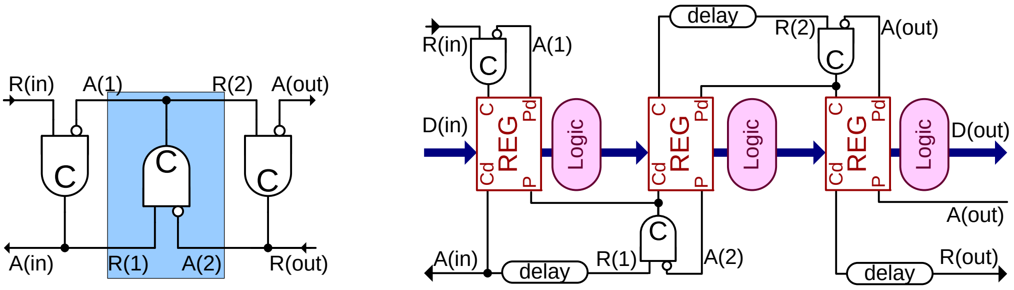
[3] Different lengths in signal wires may lead to an inconsistent view of the order of events at sender and receiver. Therefore, special care must be taken when placing and routing these components.

timing constraints) to the value domain. Depending on the exact coding scheme, one can again distinguish between two-phase and four-phase protocols. While the former is explained in greater detail in Section 1.3.3, we will focus on the latter in this paragraph. Both methodologies have in common that they need $2n + 1$ wires to transmit n bits of data, as indicated in Figure 1.6(b). The four-phase protocol, however, uses a different coding scheme (cf. Figure 1.8(b)) to represent logical values than the two-phase protocol. According to the state diagram in the figure, logical values are *one-hot* encoded, and any two consecutive logical values are always separated by the empty state (the fourth state marked as "invalid" is not used). This leads to the exemplary waveform shown in Figure 1.8(a): As a response to valid data (V) (which is basically a request encoded in the data bits themselves) the `acknowledge` line is asserted, signaling that data has been captured. The source then assigns the "empty" data word (E) as a spacer, which is followed by de-asserting `acknowledge` to its initial value. This powerful encoding is fully delay-insensitive, as wire delays do not matter any more: A bit is considered valid as soon as one of its signal lines is 1, regardless of the (routing) delays that might occur. It is of course also possible to bundle n dual-rail data bits to a data vector, as completion detection (i.e., a check whether all data bits are valid) comes down to a simple test of any bits still being"empty". Speaking more formally, there are three disjoint (and practically easy to identify) sets of codewords [90]: (i) The *empty codeword* contains n "empty"-only data bits. (ii) The *intermediate codewords* contain some "empty" and some "valid" data bits. (iii) The *valid codeword* contains n "valid"-only data bits. Generally, the transition from (i) to (iii) and vice versa also includes a period of time where intermediate codewords (ii) can be observed. The subsequent circuitry "just" has to wait until the intermediate state is left.

This clever encoding in combination with the properties (i)-(iii) allows for a rather efficient implementation scheme for delay-insensitive circuits. It is called Null Convention Logic (NCL) and was proposed (and also used in numerous applications) by Theseus Logic, Inc., in 1996 [25] (refer to Section 1.3.4 for application examples). The main advantages of NCL are its great flexibility, high modularity, and inherent robustness. Especially when it comes to changing environmental conditions (e.g., huge temperature and supply voltage fluctuations) and fabrication process variability, delay-insensitive circuits can show their full potential.

Others. The implementation schemes described so far are only a subset of the various possibilities that exist for implementing asynchronous logic. For instance, the bundled data protocols can be adapted from acting as *push channel* (i.e., actively offering data to the receiver) to a *pull channel*, where data is requested explicitly by the receiver. Clearly, the protocol needs to be changed such that data is valid when the receiver actually wants to capture it. It is also possible to extend the protocol for exchanging data in both directions, e.g., by bundling data of node A with `request` and data of node B with the `acknowledge` signal.

In this section we introduced some of the most important and most often used methodologies to implement asynchronous logic. Especially the delay-insensitive alternatives



(a)

(b)

Figure 1.9: Micropipeline control structure without (a) and with data processing (b) (Source: [82]).

(2-phase and 4-phase dual-rail) are directly or indirectly based on micropipelines. Furthermore, the structure of micropipelines forms a powerful and relatively simple control structure for asynchronous circuits in general. Thus they are worth explaining in the following section.

1.3.2 Micropipelines

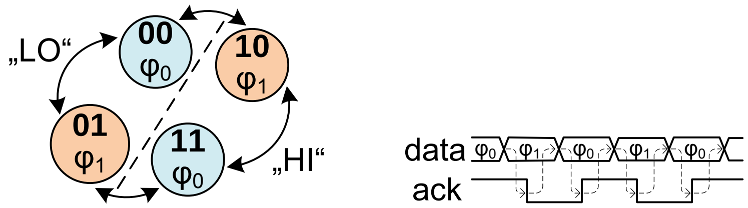
In 1989, Ivan E. Sutherland first introduced the amazing and extremely powerful — yet relatively simple — concept of micropipelines [82]. Generally speaking, if no processing logic is inserted into any pipeline, it just acts like a series of storage elements through which data can pass. Pipelines can be categorized into clocked or event-driven and elastic or inelastic. As the names suggest, clocked pipelines depend on a globally distributed clock signal whereas event-driven pipelines are controlled by locally generated events. A pipeline is said to be inelastic if the amount of data in it is fixed, which implies that the input and output data rates must match. On the other hand, an elastic pipeline may contain a varying amount of data because of internal buffering. Without any processing logic, inelastic and elastic pipelines can be compared to shift registers and FIFOs, respectively. Sutherland's micropipelines are event-driven and elastic. This solution totally avoids setup- and hold-time violations normally induced by different clock sources at the FIFO's inputs and outputs.

Control Circuit. Figure 1.9(a) shows the control circuitry of a three-stage micropipeline. Each of these stages i has a `request` input (coming from the previous stage $i-1$) and output (going to the next stage $i+1$) as well as an `acknowledge` output (to confirm the request to its predecessor $i-1$) and input (to get a confirmation from its successor). The composability of micropipelines benefits from this interface since single stages (which may run at different speeds) can easily be connected together. The behavior of the micropipeline can be explained by looking at a single stage (i.e., a C-element with one inverted input): "if the predecessor and the successor differ in state then copy predecessor's state else hold present state" [82]. Assuming that all stages are in the same state (which corresponds to an empty pipe), an input transition on R_{in} propagates through the pipeline stage by stage. Without an acknowledgement on A_{out}, all but the last stages are then in the same state (now, the pipe is partially filled). Further input transitions on R_{in} will eventually cause the pipe to be full, which is characterized by opposite states of adjacent stages. An acknowledgement on A_{out} will remove the rightmost transition from the pipe, thereby producing an empty stage. This empty slot now moves backwards, stage by stage, until it reaches the pipe's beginning, which is signaled via A_{in} to the environment — a new input transition on R_{in} is allowed to occur. This concept can abstractly be seen as "bubbles" and "tokens" moving back and forth in the pipe. As we will see in Section 1.3.3, the method of dual-rail two-phase asynchronous circuits is quite similar.

Micropipeline with Processing.. A micropipeline that contains both storage elements and combinational logic is presented in Figure 1.9(b), and can be seen as yet another way to realize asynchronous circuits (in addition to the methods presented in the previous section). To realize such a FIFO, special event-driven storage elements must be used. As indicated in the figure, each register *REG* has two control inputs (Capture and Pass) and two control outputs (Capture done and Pass done). Whenever the values of C and P match, data can directly pass through the register (thus not acting as storage element). Otherwise, i.e., when $P \neq C$, the current value is preserved. The control outputs Pd and Cd are slightly delayed copies of the corresponding control inputs and serve as acknowledgement signals (they are applied as soon as the register has finished its work). The delay elements explicitly included in the drawing are of fundamental importance: The control signals must be delayed long enough for the combinational logic to finish its computations, in order to guarantee stable data when the next stage is activated (matched delays, refer to the self-timed delay model in the previous section). In comparison with the presented delay models, Micropipelines can be seen as combination of the bundled-data and bounded-delay models.

1.3.3 Two-Phase Dual-Rail

We have already learned about Null Convention Logic and its encoding and properties in Section 1.3. Recalling the dual-rail coding scheme of Figure 1.8(b), the question arises if there is another, from a mathematical point of view maybe more efficient, way to represent the logic values. Not surprisingly, the answer to this question is yes. As illustrated in Figure 1.10(a), the coding differs from its four-phase counterpart in that it uses all available states as valid codewords. Thereby, the logical values 0 and 1 are both represented by two codewords each. Hence, in addition to the logical value itself, the codewords also contain a *phase*, which is either φ_0 or φ_1. This is also evident from Figure 1.10(b), where an exemplary waveform is illustrated. Each *transition* of the `acknowledge` line (in combination with a transition of exactly one of the two signal rails per bit) signals a new data wave, making an explicit "empty" state superfluous and thereby potentially increasing performance. State transitions are only allowed between codewords of alternate phase, i.e., two successive states always differ in at least their phase (and maybe



(a) (b)

Figure 1.10: LEDR coding scheme (a) and exemplary LEDR waveform (b).

their logical value). The coding was chosen in a way that from one state to another, exactly one physical signal needs to change its value. In literature, this encoding scheme is usually referred to as Level-Encoded (Two-Phase) Dual-Rail (LEDR) [18,60]. In [56], *phased logic*, an entire new design methodology based upon LEDR, was proposed. It allows to design delay-insensitive, asynchronous circuits while conceptually still supporting the well-known synchronous design paradigm.

When directly comparing LEDR to NCL, one might wonder which of these two design alternatives is better suited for implementing delay-insensitive circuits. To answer this question, one can use a code's characteristic *rate* R and *redundancy* r [90], defined as follows:

$$R = \frac{\log M}{n} \tag{1.1}$$

$$r = n - \log M \tag{1.2}$$

In these equations, n is the code's length (number of bits per word), and M is the code's size (number of valid codewords). For the two-phase code we have $M_{2-phase} = 4$, as all possible codewords are valid "messages". In contrast, $M_{4-phase} = 2$, because one codeword is unused, and the empty codeword does not encode actual data. Clearly, both codes are dual-rail and hence share $n = 2$. While $R_{2-phase} = 1$ and $r_{2-phase} = 0$ is the optimum that can be reached, the four-phase scheme having $R_{4-phase} = 0.5$ and $r_{4-phase} = 1$ is by far not optimal from a strictly mathematical point of view. However, it is also important how efficiently a code can be ciphered and deciphered by the communication participants. As it turns out, actual hardware implementations are significantly more efficient (in terms of area consumption and as a consequence also performance) for the four-phase methodology. We have already mentioned earlier that transition signaling (e.g., LEDR) is more complex to handle than state signaling (e.g., NCL).

For all asynchronous implementations in our project we use LEDR, as they exhibit more pronounced "asynchronous" properties than bounded delay circuits, and we have already gained some practical experience with it [19,45]. The target technology are Field Programmable Gate Arrays (FPGAs), so obviously the actual implementations follow the QDI delay model. However, on LEDR-gate abstraction level (e.g., 2-input dual-rail logic gates) the circuits can be considered delay-insensitive. In this approach, completion

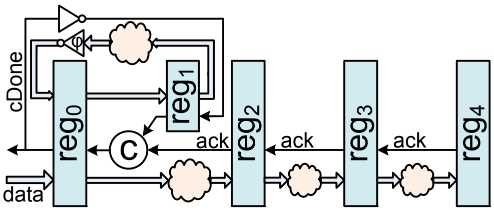


Figure 1.11: Example LEDR circuit structure.

detection comes down to a check whether the phases of all associated signals have changed and match. As usual, handshaking between register/pipeline stages is performed by virtue of a *capture done* signal (and the request implicitly coded in the data rails).

Figure 1.11 shows an exemplary LEDR circuit with two sequential registers ($reg_{0\to 2}$) and a feedback loop ($reg_{0\to 1\to 0}$). Direct feedback (i.e., without a shadow register like reg_1) is not possible, as race conditions and deadlocks may occur when a register issues its own acknowledges and requests. Also notice the phase inverter in the feedback path. The `acknowledge` line is a combination of acknowledges from all n directly succeeding stages, combined by an n-input C-gate. The structure resembled in the figure (with the exception of reg_1) has some similarities to micropipelines. The inherent handshaking between the register stages leads to virtual "tokens" and "bubbles" moving back and forth in the pipeline, depending on the "fill state" of the pipeline. Notice that in contrast to synchronous logic, the single stages are only active if new data is assigned to their inputs, and the succeeding stages have already issued their acknowledge. In principle, the single logic stages can perform their operations concurrently. Similar to synchronous designs, where fast stages have to idly wait on the next clock transition, fast LEDR stages need to wait for new data (or the respective `acknowledge`) from neighboring slower stages. Consider a simple LEDR pipeline with n stages, each of which having a unique stage delay $\Delta_i, 1 \leq i \leq n$. Now the following observations concerning the total propagation delay in case of a full (τ_f) and empty (τ_e) pipeline can be made (compared to the synchronous case τ_s with a clock period Δ_{clk}):

$$\tau_f = n \max(\Delta_i) \tag{1.3}$$

$$\tau_e = \sum_{i=1}^{n} \Delta_i \tag{1.4}$$

$$\tau_s = n\Delta_{clk} \geq n \max \Delta_i \tag{1.5}$$

Consequently, when looking at LEDR circuits at pipeline abstraction level, average case performance is generally possible. However, when directly considering the function blocks themselves (e.g., combinational logic), one can observe that they are *strongly indicating* [81] and therefore always exhibit worst case latency. Strongly indicating means that a function block does not compute *any* output until *all* its inputs are valid and consistent. Obviously, both the LEDR and NCL design methodologies as we have presented them fulfill this property.

So far, we have seen various possibilities to realize asynchronous hardware. Especially NCL and LEDR seem to be very promising as they resemble the synchronous design paradigm in may ways. In the next section we present some real-world examples of asynchronous designs.

1.3.4 Application Examples

There are numerous asynchronous designs available today. While being interesting and promising research objects for academia, also industry is increasingly using asynchronous circuits. For example, Infineon Technologies, Intel, Sun Microsystems (led by Ivan Sutherland), Boeing, and many others have current research activities in the field of asynchronous logic. Products using asynchronous technology are for example offered by Philips Semiconductors, Myricom, and Sharp Corporation [22]. In this section we introduce some impressive asynchronous designs that have been developed over the last few decades. We will see from these examples that it is immanent to combine different asynchronous design methodologies and exploit the specific circuit properties in order to obtain highly optimized, economically competitive systems.

AMULET. Fabricated in 2000, AMULET3 is the third version of the 32-bit ARM-based asynchronous processor core developed at the Department of Computer Science at the University of Manchester (U.K.) [37,39]. The overall architecture is based on Sutherland's Micropipelines in order to allow for low power and area consumption. The pipeline of AMULET does not consume dynamic power if there is no actual work to be done. To optimize silicon area, the designers decided to use ordinary 4-phase latches with 6 transistors each, which considerably saves area but at the same time requires more complex control structures for 2-phase to 4-phase conversion. An interesting optimization has been applied to the ALU in order to achieve average case performance: A special circuit determines the longest carry propagate path and dynamically adjusts the corresponding delay of the pipeline stage accordingly. Compared to its synchronous counterpart, performance, area and power-consumption are almost similar, while power dissipation in idle mode and electromagnetic emissions are considerable better. This proves that also rather complex asynchronous designs are competitive.

AsyncRFID. In 2006, Caucheteux et al. presented a fully asynchronous implementation for contactless systems [12]. An asynchronous solution seems well suited for this kind of application, as low power consumption and adaptability to varying operating conditions (especially supply voltage) are of central concern. The strictly data-driven activation methodology of QDI circuits in combination with reduced electromagnetic emissions (and consequently lower noise on the supply voltage lines) reduces power consumption and considerably eases power management. AsyncRFID uses the 3-state 4-phase asynchronous protocol (NCL). However, purely asynchronous logic also requires the communication link to be remodeled, because without a fixed operating speed, serial communication cannot be established. To this end, an asynchronous event-based communication scheme has been developed. Similar to the dual-rail encoding shown in Figure 1.6(b), the "phase information" is encoded in the data stream, making the communication insensitive to timing delays. The authors also compare their new design to other existing solutions and find that while having the lowest downlink data rate, the power consumption is far less (about $100\mu A$ instead of $150\mu A$ and $400\mu A$, respectively). It is also evident from the measurements that increasing the distance from the transmitter from 0 to 6cm, the power consumptions decreases hand in hand with the data rate.

Atmel AVR. For the use in Wireless Sensor Networks, a low cost processor with extremely high energy efficiency was needed. In [68] an asynchronous counterpart to a well-known 8-bit AVR has been developed. For this project the focus has been set on short design time and low power consumption, rather than on high performance. This is the reason why the developer used so called desynchronization (see Section 1.4) to transform a synchronous design into an asynchronous design. D-flipflops have been replaced by latches, and combinational logic has been supplemented with matched delays (thus, no performance gain can be expected, as these delays must reflect worst-case timing assumptions). The process of desynchronization added a 5% overhead in area consumption and did not have any impact on the system's speed. However, the asynchronous version can be operated at $0.5V$ (instead of nominally $1.2V$). Compared to the synchronous design, a gain in energy efficiency of a factor of 5 for $1.2V$, and a factor of 10 for $0.5V$ could be achieved, respectively.

Clockless Crossbar Switch. Fulcrum Mircosystems presented a 16-port clockless crossbar switch in 2004 called PivotPoint [16]. This device is capable of routing 1.6 terabits of data per second with an effective operating frequency of $1.4GHz$ at nominal voltage and temperature. The design itself is a mixture of synchronous and asynchronous modules, depending on which design style brings more benefits for a given functionality. The design uses dual-rail four-phase handshaking protocols for implementing the asynchronous modules. In order to improve performance, *"Fulcrum's circuit technology combines the four-phase handshake with domino logic in a method called integrated pipelining"* [16]. While domino logic can achieve very high speeds, low latencies and low power consumption, it at the same time suffers from higher delay variability and reduced noise immunity.

As we have seen so far, there are plenty of relatively diverse applications of asynchronous circuits. Having in mind the bad support by professional software tools, one might wonder how such complex designs can actually be implemented. Unfortunately, the answer to that is not that simple, as there is a huge number of modeling and synthesis techniques. To stay within the scope of this work, we will only present the automated design flow used at our department for implementing asynchronous circuits in the following section.

1.4 Asynchronous Design Flow

For efficiently implementing complex asynchronous circuits it is of upmost importance that sophisticated software tool support exists. We have already seen in the previous sections that asynchronous logic design has many quite diverse facets. The unique properties of various asynchronous design styles need to be reflected by the used software tools. The overall complexity and diversity (and the associated lack of acceptance) compared to synchronous circuit design lead to a relatively low support by (commercial) tools.

However, the special area of automated synchronous-to-asynchronous converters gains more and more interest. This procedure, which is also called *desynchronization*, has the major advantage that designers need not care about the fallacies and pitfalls of asynchronous design. The systems can be described in a synchronous fashion, and an (optimally fully) automated tool chain converts it accordingly. A good overview to existing design flows using automated circuit conversion has recently been presented in [79], while e.g. [8] goes into more detail and focuses especially on handshake protocols. The approach taken at our department [55] is also a form of desynchronization, and is explained in more detail in the next few paragraphs.

The general idea behind automated circuit conversion is to use a suitable circuit representation (one which designers are familiar with), and let a software tool convert the circuit into an asynchronous representation. In the process, the tool needs to identify concurrency, as well as temporal and causal dependencies, and must of course guarantee functional and temporal[4] equivalence between the input and output circuits.

The design flow used throughout this work is illustrated in Figure 1.12. Although in general the design flow is not limited to systems described in VHDL (Very High speed Integrated Circuit Hardware Description Language), we limit this description to pure VHDL projects. An ordinary synchronous and synthesizable circuit description is compiled with the Synopsys Design Compiler (or any other suitable synthesis tool) against the `ASYN library`. This library only contains D-flipflops and all combinational gates supported by the asynchronous library (i.e., 2-input `AND`, `OR`, `XOR`, `NAND`, and the single-input inverter `INV`). The result of this first step is a gate-level netlist containing only instances of supported asynchronous elements (i.e., only elements of the `ASYN library`).

This very netlist now serves as input for the tool developed at our department [55]. The PC-based application analyzes the netlist, generates a graph representation out of it, identifies the registers, and detects all data dependencies between the registers. All information obtained in these steps is combined to build a register topology graph, which in turn is used to decide about the structural extensions necessary for building the final asynchronous circuit. As also indicated in the figure, the software has to perform several central steps before the final netlist can be generated:

1. Replacing single-rail signals with dual-rail signals.
2. Replacing the gates from the netlist with their dual-rail pendents.

---

[4] This just addresses the order of events, not their exact timing. The latter must obviously be expected to change when the clock is removed.

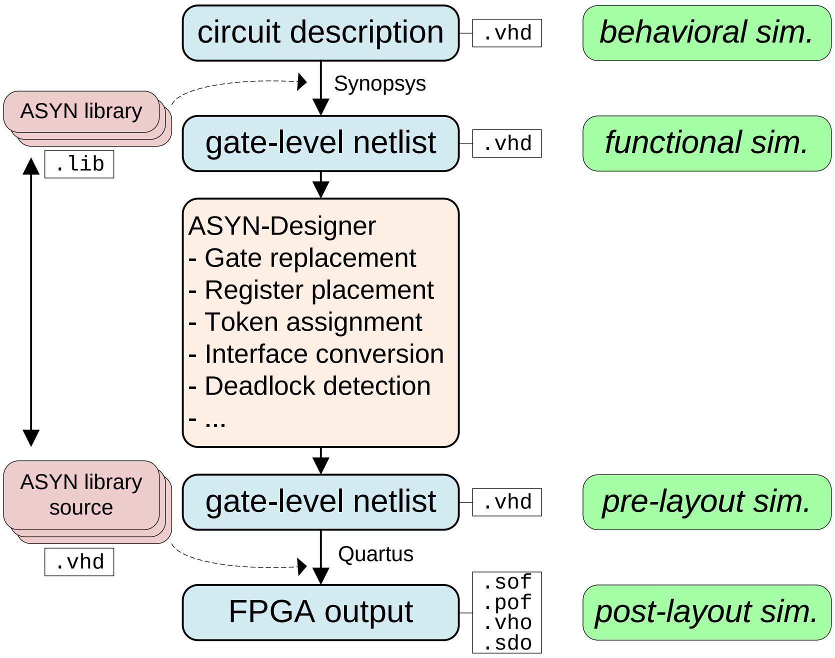
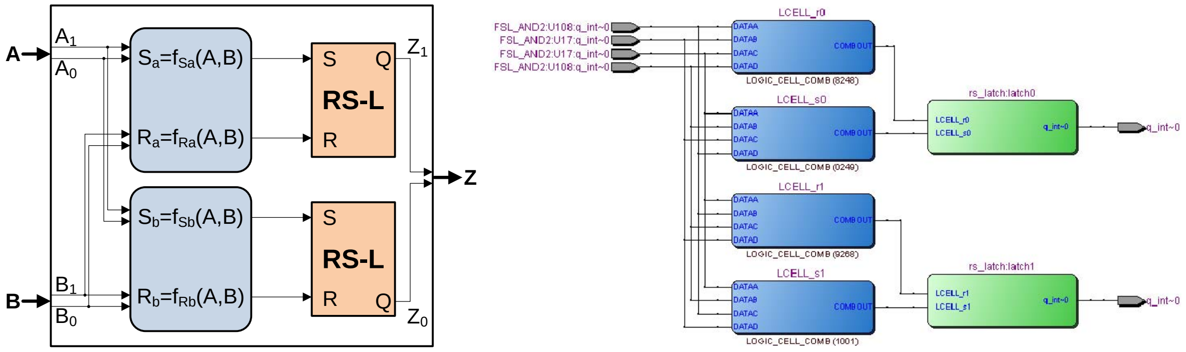


Figure 1.12: ASYN design flow.

3. Replacing the synchronous registers with latch-based asynchronous registers.
4. Adding handshaking circuitry between the registers according to the topology graph.
5. Adding handshaking signals to the design's entity.
6. Converting the entity's interface according to the designer's requirements.
7. Removing obsolete clock signals from the entity and the netlist.
8. Generating a valid token assignment (can be controlled by designer).
9. Check for deadlocks in the initial token assignment (if it has been performed manually).

As a result of this conversion, another gate-level netlist is obtained. While the *input* netlist describes the original synchronous circuit, the *output* netlist now represents the respective asynchronous design. In combination with the (technology dependent) VHDL source code of the `ASYN library`, the final design can now easily be generated using practically any suitable commercial tool. In our case, we use the Altera Quartus software design suite to generate the programming and simulation output files for the target FPGA devices. Obviously, the `ASYN library` needs to be adapted manually for each target technology, because different hardware platforms require different implementations to guarantee the internal timing constraints of the basic LEDR gates (this is especially true for FPGAs).



(a) (b)

Figure 1.13: LEDR AND gate block diagram [19] (a) and technology mapping for Altera Cyclone II FPGA (b).

The presented design flow is almost fully automated, and there are just a few manual steps included. It also offers us different levels of abstraction for simulation (*behavioral* and *functional* of the synchronous design, *pre-layout* of the generated gate-level netlist, and finally *post-layout* simulation of the resulting programming files, see Figure 1.12), and at the same time integrates relatively seamlessly into existing tool chains. It should be clear that synchronous-to-asynchronous conversion is a complex procedure. For instance, the initial phase and token assignment algorithms are topics of their own. Another rather interesting issue is the construction and layout of dual-rail LEDR gates and registers, and their mapping to a specific target technology. Figure 1.13(a) shows the latch-based structure of an LEDR AND gate [19]: The dual-rail inputs `A` and `B` are both fed to combinational functions which calculate set and reset signals for the output latches of Z_0 and Z_1, respectively. The inputs' current phases and logical values thereby determine whether a latch has to hold or toggle its value — only if both `A` and `B` are in the same phase, the output `Z` is allowed to change. The structure can easily be mapped to any LUT-based FPGA architecture. The result of technology mapping for an Altera Cyclone II device is shown in Figure 1.13(b). One can clearly identify the output latches with their set and reset inputs, as well as the combinational functions which generate these signals. Implementation details for registers, phase detectors, phase inverters, and other basic LEDR gates can be found in the respective literature [8, 19, 45, 55, 79].

1.5 The Time-Triggered Protocol

The Time-Triggered Protocol (TTP) has been developed for the demanding requirements of distributed (hard) real-time systems [48, 49]. It provides several sophisticated means to incorporate fault tolerance and at the same time keep the communication overhead low. TTP uses extensive knowledge of the distributed system to implement its services in a very efficient and flexible way.

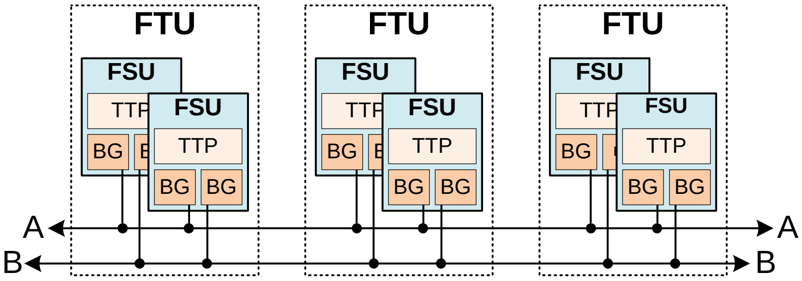


Figure 1.14: TTP-system structure [48].

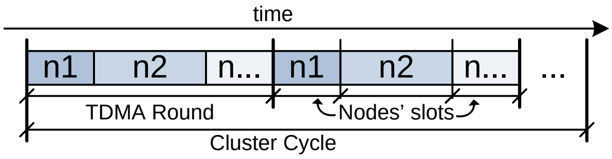


Figure 1.15: Time Division Multiple Access scheme.

This basic system architecture is illustrated in Figure 1.14. A TTP system generally consists of a set of fail-silent units (FSUs), all of which have access to two duplicated broadcast communication channels (labeled `A` and `B` in the figure). Two replicated FSUs are usually further grouped into a Fault-Tolerant Unit (FTU), whereby both units are guaranteed to perform the same state changes at about the same time (within some known, system-dependent precision Π). A fundamental concept of TTP is to trigger all activities by the progression of time. The required global notion of time is established by a clock synchronization service that is part of the protocol. According to the time-triggered philosophy, the access to the communication channel is performed in a TDMA (Time Division Multiple Access) fashion: Communication is organized in periodic TDMA rounds (typically with a duration in the order of milliseconds), which are further subdivided into the single sending slots for the communicating nodes (Figure 1.15). Each node has *statically* assigned sending slots, thus the entire schedule as well as all message delivery times are already known at design-time. Since each node knows exactly when other nodes are expected to access the bus, collision avoidance, membership service, clock synchronization, and fault detection can be handled without considerable communication overhead. Explicit Bus Guardian units (BG) are used to limit bus access to the node's respective time-slots even in the case of faults. The physical communication layer is not part of the TTP specification [86], but usually 1 up to 4 Mbit/s Manchester coded data streams with open collector bus drivers are used. Notice, however, that modern applications usually use a star architecture rather than this linear structure for reasons of efficiency and dependability. In that case the bus guardian units are located at the centralized star-coupler unit rather than locally at each FSU.

The following list provides an (incomplete) overview over the most important system properties and services that the Time-Triggered-Architecture offers:

- **Communication latency:** The static nature of TTP makes it easy to guarantee timeliness of messages and to calculate upper bounds on communication latency.

- **Bus access**: Since each node has predefined time-slots in each TDMA-round, there is no need for collision detection. Collisions are avoided by design. The so called MEDL (Message Descriptor List) holds the exact schedule of the entire system (which data is transferred, in which slot is it transmitted, which tasks must be executed, etc.).

- **Acknowledgement:** All (correct) messages are explicitly acknowledged in the message of the subsequent time-slot. This way, both sender and receiver are able to determine whether a message has been transmitted successfully to all (correct) nodes or not.

- **Controller-State (C-state):** For TTP to work properly, all (correct) nodes must agree upon their internal state (i.e., mode, time and membership). State changes amongst duplicated FSUs must be guaranteed to happen consistently and approximately simultaneously (within the precision Π of course).

- **Clock Synchronization:** This is the main prerequisite for all services to work properly. It is performed once per TDMA round, using a fault-tolerant distributed convergence-averaging algorithm. In this algorithm each node compares the arrival times of the messages received from the other nodes with the respective expected values (according to the static schedule) and corrects its local clock according to the perceived differences. This periodic state correction compensates the imperfections of the local clock sources. The attainable precision Π typically is in the order of microseconds.

- **Membership Service:** All nodes sustain a membership vector which indicates the enabled and disabled (i.e., faulty) nodes. All correct nodes must have a consistent view of the membership vector. To save communication bandwidth, the membership vector is not explicitly transmitted over the bus in each message, but is part of the CRC calculation. Thus an invalid checksum indicates either a faulty message or an inconsistent membership vector (or possibly both).

- **Sparse timebase:** In TTP, events (e.g., sending/receiving messages) are restricted to occur at a globally synchronized lattice of action points in time. Thus, the time-line is not continuous but sparse, and can therefore be handled much more efficiently. This is especially important to maintain a consistent order of events at the distributed nodes.

- **Fault hypothesis:** For TTP, it is assumed that faulty nodes behave in a fail-silent manner. Since receivers can detect faulty messages and inconsistent C-states, such messages can be ignored, which equals a fail-silent behavior. However, notice that designing fail-silent nodes actually is a quite complex task.

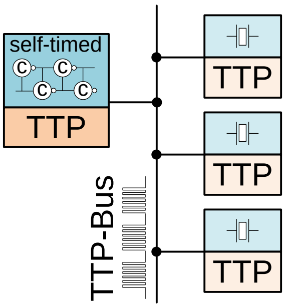


Figure 1.16: ARTS system setup.

In the next section we will introduce our research project ARTS. The deterministic nature of TTP, in combination with its strict timeliness of events (as we have seen in this section), should make it quite obvious that a self-timed asynchronous logic design is not well suited as basis for real-time applications. The project overview shall point out the basic ideas to overcome these intrinsic obstacles.

1.6 ARTS - Aims and Contributions

The aim of the ARTS (Asynchronous Logic in Real-Time Systems) research project is to investigate the temporal predictability and stability of asynchronous (quasi) delay-insensitive hardware designs. More specifically, we want to compile models for the timing uncertainties of hardware execution times and extend these to make quantitative and qualitative statements on the timing behavior of self-timed circuits. The theoretical and experimental analyses and considerations shall also provide indications for improving the temporal stability of self-timed circuits. As introduced in previous sections, our design flow supports the generation of (Q)DI asynchronous logic. By its definition, however, any form of delay-insensitivity strictly contradicts the requirements of time-triggered systems. As no delay assumptions can be made, predicting the execution time of a DI block is simply not possible. In contrast to the properties of the *model*, real hardware does not behave in a delay-insensitive manner: Depending on various factors, propagation delays have upper and lower bounds (and are subject to jitter). Thus, while operating with circuits that are in principle (quasi) delay-insensitive, we consider all logic elements to follow the more realistic bounded delay model — and these upper/lower bounds of an asynchronous block are the very properties we are interested in.

The tangible project result shall be an asynchronous implementation of a TTP controller operating in an ensemble of conventional, synchronous controllers as illustrated in Figure 1.16. It is evident from the explanation in Section 1.5 that TTP's static bus access schedule as well as its clock synchronization and data transmission protocol rely on the stability of the constituent nodes' local clock sources. Therefore it seems quite daring to implement the controller logic in an asynchronous design style. However, moving such a deeply synchronous application to an asynchronous implementation is an interesting

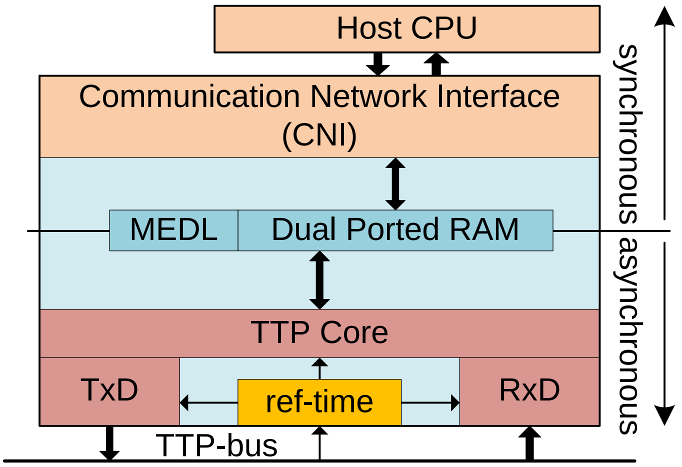


Figure 1.17: TTP-node block diagram.

and informative challenge of its own, and a very convincing case study for demonstrating the temporal predictability of asynchronous logic. In this context we need to solve two fundamental problems, whereby it is mandatory to derive quantitative boundaries for the attained properties in both cases:

1. We have to make our design operate stable enough to meet TTP's stringent requirements on execution times and jitter. Conceptually this issue has to do with the fact that control flow in self-timed circuits is flexible and not strictly time driven, as in the synchronous paradigm.

2. We have to provide a stable local time reference for bit timing and bus access. This issue originates from the fact that in synchronous systems the clock sources can also be used as time reference — which is missing in a self-timed approach.

Figure 1.17 shows the internal structure of the envisioned asynchronous TTP controller. It is very similar to the existing TTP communication chip from TTTech Computertechnik AG [87], our project partner. The interface to the controlling host CPU is called Communication Network Interface (CNI) and will remain synchronous, thus existing soft- and hardware-solutions for TTP can be used without any modifications. In our setup, the host CPU is a Motorola MPC555 microprocessor, which executes the node's application software. The CNI is realized as a standard parallel memory interface with address, data, and some control signals. In the synchronous solution, both the host CPU and the communication controller share the same clock source, thus defining a tight phase relationship between the two chips. However, the "central" parts (`TxD`, `RxD`, `ref-time`: bus access, receiver-unit, transmitter-unit, etc.) of the controller will be replaced by fully asynchronous implementations. The higher level services and the overall protocol management units (Clock synchronization, membership, consistency checks, etc.) will at least partially be realized as software stack executed on a suitable microprocessor core. A dual-ported RAM separates the CNI from the `TTP-core`, thereby forming not only a temporal firewall [48], but also dividing the synchronous from the asynchronous parts[5]. The entire controller will be synthesized for FPGA technology only, as ASIC (Application Specific Integrated Circuit) designs are too costly. It should be noted that the asynchronous controller implementation is intended solely as an academic case study and not as a prototype for an industrial design. Consequently, we will not implement all features defined in the TTP specification or supported by existing solutions, but only a feasible subset that allows us to demonstrate the project goals.

1.6.1 Contribution

- Since in fact the temporal behavior of asynchronous logic is not unpredictable but just more difficult to model than in the synchronous case, we will elaborate a timing model for asynchronous circuits that allows us to predict its execution time for a given hardware task.
    - Besides a worst case timing analysis (which is also performed for synchronous systems), asynchronous logic also requires a statistical investigation of the timing behavior, as deterministic and random jitter sources significantly influence the overall temporal behavior of a circuit.
    - Another very important part of the timing analysis and prediction addresses varying environmental conditions such as fluctuations in supply voltage, operating temperature, or fabrication-related parameter variations. These fluctuations clearly affect the circuit's speed, and they are usually masked with additional margins of a synchronous systems' clock period. Obviously, asynchronous designs need to actively deal with changing operating speeds.
- In real-time applications a time reference is needed for communication, scheduling, timers etc. While this reference comes for free with the clock in a synchronous design, explicit measures are needed for this purpose in an asynchronous design. We shall propose and investigate appropriate solutions here as well.
- In order to guide our investigation by the actual needs of a practical application we use a TTP controller as our showcase. The controller for a time-triggered communication system is commonly accepted as a demanding real-time design, therefore it is a tough benchmark for our approach that makes our results convincing.
- The design and implementation process of an asynchronous TTP controller is, from an engineering point of view, an interesting task of its own. It will highlight many fallacies and pitfalls associated with asynchronous design, especially when embedding an asynchronous module into an existing (synchronous) environment.

---

[5] Both host-CPU and TTP controller share a global time-base, thus concurrent access to the dual-ported RAM can be controlled to avoid collisions.

- Another contribution concerns asynchronous QDI modules interfacing with the "outside" world (which is obviously the case for a TTP controller). First of all, input signals are mostly neither delay-insensitive nor dual-rail. Suitable conversion blocks and adequate timing constraints need to be defined. Furthermore, especially for FPGAs, some hardware components (e.g., internal memory blocks) just have synchronous interfaces. In order to use them as well in asynchronous blocks, special wrapper modules need to be constructed.

1.7 Chapter Organization

In this chapter we have presented the basic goals and ideas behind the research project ARTS. Besides the detailed overview to asynchronous circuit design in combination with delay models, coding schemes, and automated tools support, we also introduced the concept of the Time-Triggered Protocol and presented the almost entirely automated design flow for implementing asynchronous hardware.

However, as we have seen in Section 1.6, focusing solely on asynchronous logic will not be sufficient for our demonstrator design. One can conclude from the block diagram in Figure 1.17 that we need some interfaces to the outside world. This outside world shows many different facets, which are not obviously and directly compatible to the self-timed (and theoretically delay-insensitive) design approach. For one, there is the host interface. Although we intend to keep this interface synchronous for reasons of compatibility (and direct reuse of the existing hardware platform), the synchronous and asynchronous parts interact with each other via the dual-ported RAM. Another issue is the bus interface itself. It is, from the asynchronous receiver's point of view, a single-rail signal, thus not capable of performing any handshaking interactions. A third obvious interface is needed to use an FPGA's built-in memory. Some devices (such as the one used in our case) only support registered memory access, thus necessitating an appropriate interface. All these interfacing issues are discussed in detail in Chapter 2.

Afterwards, in Chapter 3, we take a closer look on the temporal characteristics of self-timed logic. To this end, we investigate signal jitter as defined for synchronous systems and adapt these definitions to our (asynchronous) purposes. We also elaborate a suitable timing model in order to better understand the temporal behavior of asynchronous logic. Based upon this very model we present some case studies which show how deterministic and predictable asynchronous logic actually is.

Having a solid theoretical timing model, Chapter 4 puts all timing relevant topics together and shows actual implementation strategies for time reference generation units. In this chapter we elaborate a detailed list of requirements in preparation for the final TTP controller implementation, and we show how these requirements can efficiently be implemented. Clearly we also include some experimental results to demonstrate the correct functionality of the proposed solution.

The resulting implementation of a functional TTP controller is finally explained in Chapter 5. Here we show how to integrate the time reference generation unit into the remaining system. This is a major challenge as we have to deal with different design methodologies here: Not only are there synchronous and asynchronous modules, also memory interfaces and temporally independently running self-timed units need to be integrated accordingly. A software stack must be developed which actually implements TTP's protocol stack. The inadequacies of the time reference (compared to a crystal oscillator) must be compensated, and full compatibility to the host CPU needs to be guaranteed. We further present the experimental results of our test setup in this chapter. We perform different tests to investigate the robustness and capability of our design to adapt itself to changing operating conditions.

The work is concluded in Chapter 6, where we — besides giving a summary of the key contributions made in this work — critically discuss the obtained results. This discussion includes, e.g., remarks on usability in industrial designs, the impact on fault-tolerance of TTP when using asynchronous hardware, and the major benefits and drawback when using self-timed logic for time critical applications. Clearly, we also present some outlook and explain possible future projects and work to be done.

Chapter 2

Interfacing Asynchronous Circuits

A picture is worth a thousand words. An interface is worth a thousand pictures.
Ben Shneiderman

In the previous chapter several commonly used design methodologies for digital systems have been presented. In order to obtain a highly optimized circuit it is often necessary to combine different implementation paradigms in one design. For example, one style might be very efficient in terms of area consumption, while another one greatly reduces signal transitions and therefore power consumption. Yet another style might offer good performance at the cost of reduced modularity. While it is up to designer to create a modular system architecture by efficiently interchanging the supported design alternatives, the interfaces between these modules shall be as simple, fast, and transparent as possible.

As we will see in later chapters, for the implementation of our TTP controller we will also need efficient interfacing and conversion techniques. Not only do we need to synchronize the physical bus-signal to our internal timing, there are also independently running asynchronous modules (which can potentially be realized using different design styles) in combination with a microprocessor core. In order to allow data exchange and manage control flow among these units, proper interfacing and conversion techniques must be elaborated. Therefore, this chapter introduces design methodologies and concrete implementation strategies for such building blocks. We consider two major interface types: (i) Interfaces based on handshaking protocols which convert one code into another one (e.g., NCL, LEDR, two-/four-phase bundled data), and (ii) interfaces between different timing domains without the possibility of back-pressure in the control path [26, 29, 30].

As our main implementation platform are common (synchronous) FPGAs, we try to restrict ourselves to basic gates or elements which can be synthesized in most FPGA technologies. We further consider gate level circuits only as they can be ported to different

FPGA families without considerable effort, and they allow us to study the properties and pitfalls on a relatively high level.

2.1 Related Work

2.1.1 Synchronization Techniques

Synchronizing external signals with the internal clock domain in case of “ordinary” synchronous logic is an interesting topic that has already been discussed in great detail in literature [17,40,47]. When also considering asynchronous logic, synchronous/asynchronous interfaces can be classified by the degree of concurrency allowed for the interconnected circuit parts. Thus, we distinguish between *loosely-coupled* and *strongly-coupled* methods for interfacing synchronous and asynchronous modules. Loosely-coupled components operate fully parallel most of the time. Asynchronous and synchronous modules perform computations concurrently and are only stopped and synchronized in case there is data to be exchanged. Strongly-coupled circuits, on the other hand, run in lockstep mode. All modules are synchronized with each computation step. This is typically used in pipelined circuit structures where data is passed on from one stage to another after each cycle.

The discussion in this section will be focused on asynchronous producers and synchronous consumers. Passing data in the other direction is straight-forward since a synchronous producer can issue a new request at any time as long as the asynchronous receiver is ready. This readiness is signaled by the acknowledge signal which indicates that the previous data request has been processed. Since the acknowledge signal is an asynchronous input to the synchronous producer, it needs to be synchronized using one of the methods presented below.

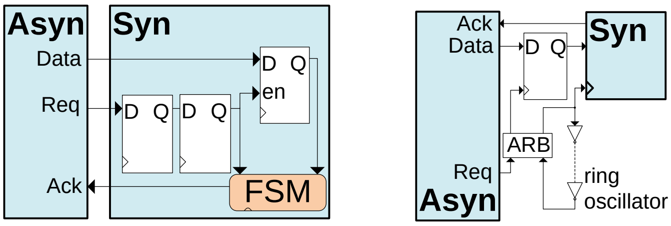
2.1.1.1 Loosely-coupled

Since both the synchronous and the asynchronous module are allowed to run freely, the asynchronous module may initiate a data transfer at any point in time. Consider a typical 4-phase bundled data interface with a dedicated request signal indicating the availability of new data. The most simple solution for reading is to latch the asynchronous request signal with a commonly used 2-stage synchronizer (cf. Figure 2.1(a)). This synchronizer reduces the probability of metastability to an acceptable level[1]. Once the synchronized request signal changes to high, a data register can safely latch the input data word. The downside of this methodology is the increased latency when passing on data from an asynchronous producer to a synchronous consumer, even when there is no metastable upset.

Another approach for completely preventing synchronization failures at the interfaces is based on the idea of stoppable clocks [14, 74], which is very common in GALS systems (recall Section 1.2.2). A local clock generator provides a periodic clock signal for the

---

[1] It is assumed that in future the resolution time for metastable states will increase relative to the clock period and therefore three or more synchronizer stages will be required [6].



(a) (b)

Figure 2.1: Loosely coupled: 2-stage synchronizer (a), and stoppable clock (b).

synchronous component, which is thus able to perform its computation steps without any impediment as long as no asynchronous inputs need to be processed. However, in case a request occurs, the clock of the synchronous circuit is halted. Stoppable clocks are typically built from gated ring oscillators since an external crystal oscillator could not be stopped or restarted fast enough. As can be seen in Figure 2.1(b) an arbiter is used to decide whether an asynchronous request is processed or another clock tick is issued to the synchronous system. If the asynchronous request is granted, the input register is clocked and the system clock of the synchronous module is halted until the asynchronous data input has been latched and it is safe to continue computation. Obviously the arbiter can become metastable itself if a transition of the request signal occurs close to the next clock edge generated by the ring oscillator. In this case the whole circuit may be delayed as long as the arbiter tries to resolve the upset. Thus, synchronization failures on the data interface are effectively mitigated.

2.1.1.2 Strongly-coupled

Data-driven a.k.a. *request-driven* clocks can also be used to synchronize asynchronous and synchronous modules in GALS systems. The key idea is to exploit the request signal of the asynchronous circuit to derive a clock signal for the synchronous receiver [64]. Figure 2.2(a) illustrates how a simple ring oscillator can be extended to a request-driven clock generator. By adding a Muller C-element the oscillator requires a transition on the request input and an event at the output of the delay-element before the next clock edge can be generated. This signal can be used to clock the synchronous circuit parts. Asynchronous data inputs can safely be latched without any additional synchronizers. The delay element of the ring oscillators defines the minimum clock period. Obviously, this delay needs to be adjusted to the operating frequency of the synchronous circuit. Using a data-driven clocking scheme has the big advantage that the synchronous module is only active when new data is available. No superfluous computations or signal transitions are performed. Thus, this interfacing method can be beneficial for low-power applications. In some cases, however, multiple clock cycles might be necessary for completing a computation. Therefore several clock ticks need to be generated upon a single input request of the asynchronous module. In [52] a solution is proposed, which combines request-driven

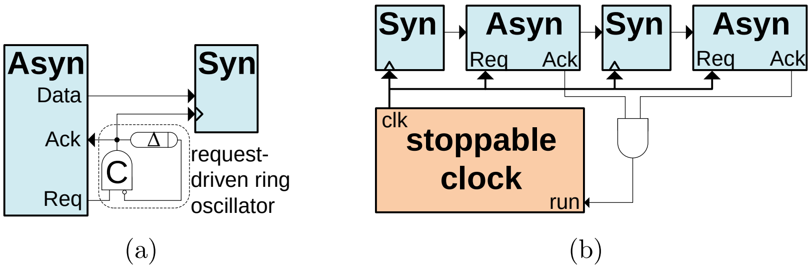


Figure 2.2: Strongly coupled: Request/Data driven (a), and Globally Synchronous Locally Asynchronous (b).

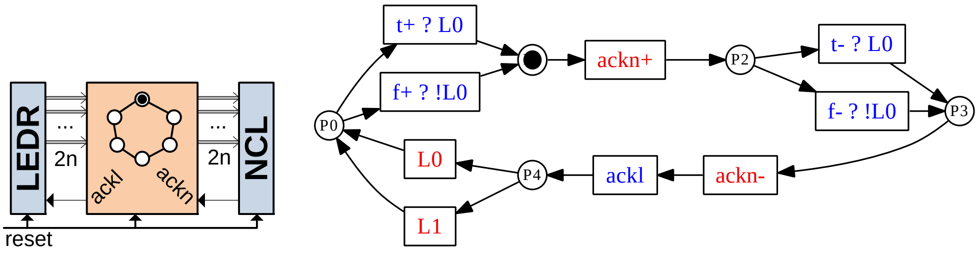
clocking with free-running local clock generators in order to flush the data out of the synchronous module.

Another interesting approach for interfacing synchronous and asynchronous components is presented in [80]. The proposed solution is based on a *globally synchronous locally asynchronous* architecture. Asynchronous components are embedded into a conventional synchronous pipeline (see Figure 2.2(b)). A stoppable clock generates clock ticks and request signals for the synchronous and the asynchronous stages. Thus, the asynchronous modules operate in lockstep with the synchronous ones. After a clock tick has been issued, the clock generator is stopped until all asynchronous stages have indicated completion of their computations. This method allows to increase average-case performance of a synchronous pipeline by replacing the slowest stages with asynchronous modules. The synchronous stages do not need to be redesigned.

2.1.2 Conversion Techniques

When it comes to converting different asynchronous design styles and communication protocols into each other, several interesting papers are worth mentioning. For instance, [71] proposes an interface called SCAFFI for GALS systems. The interface uses clock stretchers to avoid metastability issues in the synchronous sender and receiver circuits. The stretchers are controlled by special output and input ports, respectively. These ports use two-phase protocols to communicate with the synchronous designs, while actually transmitting data from one module to another employing a four-phase handshaking protocol with the bundled data approach. For large communication lines where signal delays cannot be predicted reliably any more, SCAFFI also provides the possibility to use delay-insensitive dual-rail encoding (NCL) for data transmission.

[63] proposes efficient two-phase to four-phase converters specifically designed for LEDR and NCL. While this is similar to what we want to achieve in the following sections, the problem with this solution is that it is optimized for off-chip communication only. In contrast, we focus on on-chip protocol conversion, thus area requirements (and performance) are of central importance. The need for explicit phase detectors and output latches introduces considerable area overhead, which we can avoid with our solution.



(a) (b)

Figure 2.3: Conversion interface for LEDR to NCL (a), and Petri-Net representation (b).

Yet another interesting alternative is described in [54]. The authors implement a CMOS transistor-level two-phase to four-phase conversion circuit for the use in a specially designed asynchronous FPGA. The solution is for one bit only, and without modifications the circuit scales relatively bad for large bit-widths, because a “consistency detector” for the NCL part is needed. Furthermore, transistor-level circuits are not well-suited for our purposes as FPGA devices do not allow such a fine-grained modeling.

2.2 Lockstep Conversion

2.2.1 LEDR to NCL

The first case we consider is the conversion from LEDR design style to NCL. As both of these styles are considered delay-insensitive, the conversion circuit must be built in a way to maintain this property. When considering Figure 2.3(a) one can see that the interfaces on both sides of the conversion block are basically the same: There are $2n$ signal lines for parallel transmission of n logical bits, and an explicit acknowledge line back to the producer. Clearly, and despite the similarities, the semantics are completely different and must be converted correctly inside the block. In order to maintain delay-insensitivity, the producer’s (LEDR) acknowledge `ackl` must not be asserted until *after* the consumer (NCL) has successfully received and stored the data and the empty word. Both of these events are indicated by asserting (and de-asserting) the signal `ackn` by the consumer.

Figure 2.3(b) shows the Petri-Net representation of a simplified conversion block for only one (dual-rail) data bit. The physical signals from LEDR’s dual-rails data input are labeled L_0 and L_1, and the NCL outputs are called t and f. In addition, $ackn$ and $ackl$ are the (single-rail) acknowledge signals as shown in Figure 2.3(a). For this example, we consider the following scenario (point (1) one from the list below): NCL issues the empty word, LEDR the first data word (in phase φ_0) to convert. Initially (P_1) we set $ackn = ackl = 0$. In addition, and depending on the data input L_0, one of the NCL outputs t or f is preset accordingly. The block then waits for the NCL register to store and acknowledge the new data word by generating a positive edge on $ackn$. We finish the 4-phase cycle by clearing the NCL data lines ($P_2 \rightarrow P_3$) and waiting for the respective negative $ackn$ transition. Now that the NCL part is completed, we can safely assert (i.e., toggle) *ackl* to the waiting LEDR circuit. We are at place P_4 now, where we wait for any of the signals of the dual-rail input to toggle (which indicates a change of phase and thus new data). As soon as this happens, a new cycle starts by asserting the respective NCL output after P_0.

In principle, this is a fundamental mode Huffman circuit, because only one input changes at a time ($ackn+ \rightarrow ackn- \rightarrow L_{0|1}$, where either L_0 or L_1 changes, but not both). Although this property is not required by our design, it simplifies logic synthesis and helps to avoid hazards. To achieve maximum concurrency and increase performance, one could insert the assertion of *ackl* directly after transition $ackn+$ (or anywhere else between $ackn+$ and $ackn-$). However, this would complicate the Petri-Net and possibly introduce new states as the LEDR block might produce new data while the converter is still waiting on $ackn-$. Without applying timing restrictions on the environment (which clearly would violate the property of delay-insensitivity) or implementing additional logic, such "optimizations" are difficult and troublesome to apply.

The major problem that we face here is to change from a two-phase to a four-phase protocol, as we need to insert an empty word followed by the respective value word for each data wave of the LEDR part. This directly implies the conversion block to contain a (simple) state-machine, thus making the circuit more complex than one would expect in the first place. As direct consequence of the non-stateless structure, the block itself needs to be reset in accordance to the producer's and consumer's initial states (inconsistencies in the initial state will result in immediate deadlock). To define a suitable initial state of our conversion block, there are several possibilities:

1. An obvious situation is that the producer presents the first data word during reset already, while the consumer is set to the empty word. Therefore it can receive the first data vector immediately after the reset is de-asserted. Consequently, the conversion block must not validate `ackl` until it receives the corresponding `ackn` from the NCL circuit (place P_1 in Figure 2.3(b)).

2. Another reasonable possibility is to reset both LEDR and NCL with the same data vector. The first conversion would then be obsolete and would not consume any time after reset. In this case the conversion block can initially assert `ackl`, as `ackn` will also be asserted on reset (P_4).

3. Depending on the application, it might also be possible to store the second data vector (if known at design time already) in the LEDR part and initialize the NCL circuit with the first data vector. In contrast to the previous situation, `ackl` must be de-asserted now, as LEDR's data vector is not yet consumed by NCL (P_3).

In the example of Figure 2.3(b) we chose place P_1 to be our initial place. However, we could also choose, e.g., place P_4 to be our reset state. This would be the scenario where NCL issues the empty word, and the converter is waiting for the LEDR block to send *new* data (the reset data is already acknowledged during reset). It is important to notice that all signals (inputs as well as outputs) must in either case be reset to a value that

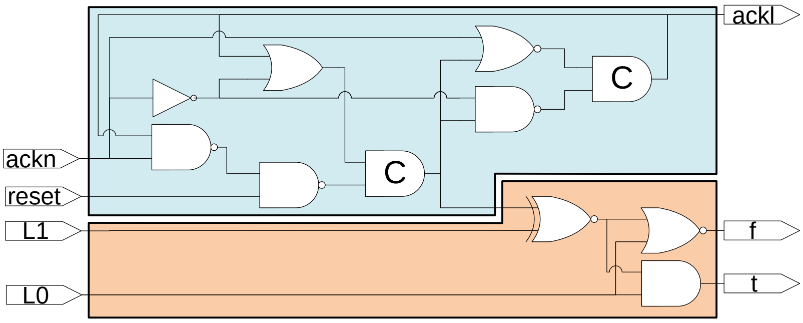
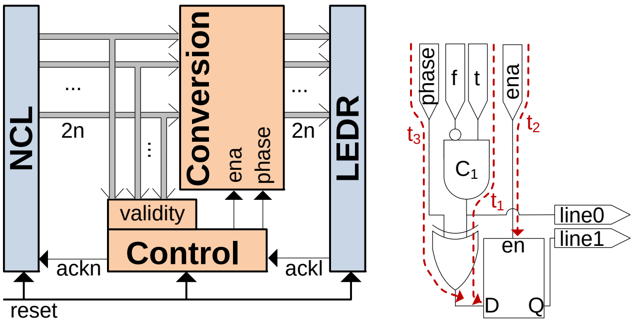


Figure 2.4: Synthesized conversion circuit.

is consistent with the initial state, otherwise the state-machine will not work properly. Based on the presented Petri Net representation and the considerations mentioned above, we can now synthesize a conversion circuit for arbitrary bit length.

A very quick and simple method to derive a working circuit description out of the Petri Net representation is to use the free tool *petrify* [15]. The tool performs multiple checks on the input description, such as deadlock detection, inconsistency checks, and unstable initial state detection. It automatically generates a state transition graph and inserts — if necessary — additional states to resolve ambiguities. However, manual review of the circuit is necessary, especially to identify and separate control logic from bit-conversion logic. This step is needed to generalize the design for arbitrary input/output bit-widths. We can also apply some optimizations because the environment (i.e., the LEDR and NCL blocks) behaves totally deterministically and strictly follows the steps defined in Figure 2.3(b). The result of the described implementation procedure is depicted in Figure 2.4. While the control circuitry is relatively complex and contains feedbacks and Muller C-gates for state preservation, the actual bit-conversion logic is quite simple. The conversion block's area (e.g., gate or transistor count) therefore scales quite well with increasing bit width, as only three additional gates are needed for each dual-rail bit. It is important to notice that the conversion circuit itself does not use latches (or other sequential elements) to store the outputs for `t` and `f` (which is usually necessary for delay-insensitive circuits). This can be justified because the LEDR block does not change its data word until the respective acknowledge has been issued. In other words, we just use the LEDR block's output latches and simply feed through the values to NCL in order to keep the area requirements low. The small bit-conversion part also exploits the fact that for each cycle, either `L0` or `L1` can change, but not both. If correctly initialized, the control circuitry always keeps track of the current phase the LEDR circuit is in (this is obvious, as it must generate the correct `ackl` signal), and can therefore use the `XNOR` gate as a kind of enable signal for data feed-through. The fundamental mode assumption can be extended to multi-bit input vectors, as all data lines can be considered independently from each other.

The main advantage of this solution compared to the implementations presented in the related work section is the fact that it scales good for large bit widths. This is because



(a) (b)

Figure 2.5: NCL to LEDR block diagram (a) and single bit conversion circuit (b).

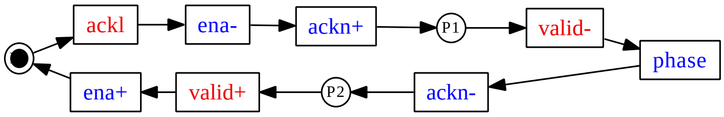


Figure 2.6: NCL to LEDR Petri Net representation.

the LEDR part does not need an explicit phase detector (which needs $2n-1$ gates[2], n being the number of dual-rail bits). Similarly, there is no need for an explicit consistency detector for the NCL side as well — all the information is implicitly provided by the respective acknowledge signals `ackl` and `ackn`. The circuit can be kept quite simple as we exploit the restricted properties of the surrounding blocks. Thus, no additional buffers are needed (except for the control logic). On the downside stands the need for correct initialization. The presented circuit needs the LEDR module to be initialized in phase φ_0 to work properly. For other initial states, the conversion circuit needs to be reset differently. As we do not implement a dedicated phase detector, and also spare output latches, we decrease system robustness. However, the presented solution targets in-chip protocol conversion only, and the physical proximity of the blocks to the conversion circuits relativizes this drawback.

2.2.2 NCL to LEDR

A quite common use-case is the conversion from four-phase NCL to two-phase LEDR. The former is often used to realize combinatorial circuits, while the latter is well suited for interfacing with other modules over probably long distances. What we need to take into account for this case is the fact that the NCL part does not hold its signal values for the entire computation cycle. This return-to-zero property stands in contrast to LEDR, where all signals are stable until a new computation phase starts. Figure 2.5(a) shows the basic block diagram of such a conversion circuit: From NCL's point of view, the converter is the data sink, therefore it is necessary to perform a validity check on the data inputs.

---

[2] One needs n XORs and $n-1$ C-gates if only two-input gates are available.

This checker module shall return logic 1 if *all* dual-rail data lines show valid data, and 0 if *all* data lines are zero. For all other (intermediate) states, the module shall implement a hysteresis and hold its old value. In order to reduce the gate count and increase the scaling capabilities for large bit widths as much as possible, we again separate the control logic from the local (per-bit) conversion circuit.

The control block itself has two inputs (`ackl` as capture-done coming from LEDR, and `valid` as "phase" signal coming from NCL's validity detector), and three outputs (`ena` to set the output latches transparent, `phase` to indicate LEDR's output phase, and `ackn` to acknowledge data transfers to NCL). Figure 2.5(b) shows the conversion circuit for a single bit cell. According to Figure 2.5(a) each bit has the `t` and `f` data lines from the NCL block as input, and the LEDR data lines `line0` and `line1` as output. From the control block there are two additional inputs `phase` (indicating the desired phase at the output) and `ena` (the enable signal for the output-latch of `line1`). Data conversion from the one-hot encoded NCL to the LEDR scheme can be achieved easily by Muller C-gate C_1. Generating the data-dependent phase signal `line1` can, for instance, be implemented using a simple latch and an XOR gate.

Having this general structure in mind, it is straight forward to describe the desired conversion functionality of the control block in terms of a Petri Net (see Figure 2.6). Starting in P_0 (LEDR output is valid and ready to be acknowledged), we first need to wait until signal `ackl` is issued. By applying `ena-` we then we set the output-latches of the LEDR-side to "hold" and acknowledge the capturing of all data with `ackn+`. After waiting for NCL to show the empty word (`valid-`), we assign the new LEDR output-phase to an internal variable `phase`[3] and acknowledge the reception of the empty word with `ackn-`. The timing constraint that `ena-` needs to reach all latches *before* transition `phase` does can be denoted[4] as $t_2^- < t_3$ according to Figure 2.5(b). Eventually, NCL assigns new data and forces `valid+`. Before actually enabling the output latches with `ena+`, another timing constraint needs to be fulfilled: The data signals coming from NCL must reach the latch's input `D` before `ena+` reaches the latches in order to avoid hazards ($t_1 < t_2^+$). The Petri-Net representation can be transformed into a hazard-free circuit using the tool petrify [15], which has already been presented in the previous section.

There exist several possibilities for circuit initialization. In the examples shown above we assume state P_0 to be the initial state (latches set transparent, data word of NCL input is assigned to LEDR output in phase φ_0). However, it is also possible to reset the circuit in states P_1 (latches disabled, NCL starts with the empty word) or P_2 (same as P_0 with latches disabled). The main disadvantage of selecting one of the alternatives on reset is that in those cases the latch and also gate C_1 must be reset explicitly. In the presented setup, however, they are initialized implicitly by their input signals.

---

[3] While this variable holds the same information as input signal `ackl`, it is assigned at a time when the latches are not transparent any more.

[4] We denote the point in time when rising/falling/any transitions reach a specific location as $t^+/t^-/t$, respectively.

2.2.3 Bundled Data as Consumer

As there exist both two-phase and four-phase handshaking solutions for the bundled data interface, NCL as well as LEDR can be converted with moderate effort. The request signal, which is needed to indicate the arrival of new data to the receiver, can easily be extracted directly out of the dual-rail data lines using a phase or validity detector.

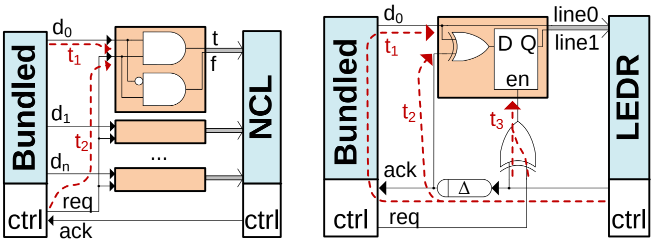
In case of NCL a validity detector as mentioned previously is needed. It can be directly seen by comparing Figure 1.7(a) and Figure 1.8(a) that the validity detector resembles the bundled data's request signal. Furthermore, the acknowledge signal can also directly be connected to NCL's `ack` line. Similar to the other solutions it is important to notice that there are timing constraints that must be met. The data inputs from the bundled data modules must be stable before the validity detector produces a positive transition and thereby triggers capturing of the data.

On the other hand, also LEDR can be converted quite easily by using the two-phase handshaking technique for bundled data. Again, the request signal can be obtained by evaluating the phase of the input data vector. As above, this phase detector outputs the current phase of the applied data in case all bits are consistent, and holds its value otherwise. Also similar to above, the acknowledge output can directly be connected to LEDR's *capture done* input. Care must be taken only for the initial state, as the polarities of request/acknowledge must match the reset state of the LEDR block. In case of a mismatch, the circuit does not work at all due to deadlock.

2.2.4 Bundled Data as Producer

Let us again start with the four-phase protocols, as they are relatively easy to handle because of the favorable return-to-zero property. One timing constraint that comes with bundled data is that all data needs to be stable *before* the respective request signal arrives at the consumer. Baring this prerequisite in mind, it is possible to use the request signal directly to generate valid and empty words as needed for NCL. The acknowledge signal can, similar to the previous section, directly be connected to the corresponding bundled data's acknowledge line. This is also illustrated in Figure 2.7(a): The request signal is used to switch the `and` gates transparent (for a valid data word) and force them to zero (for the empty word). No buffers or other conversion logic is required. The only timing requirement is that all data lines are stable at the converter gates before the enabling/disabling request transition occurs ($t_1 < t_2$ in Figure 2.7(a)).

The second case is slightly more complicated because we need to generate the correct phase signal for each bit. As this signal not only depends on the current phase, but also on the current data value, several critical timing constraints must be fulfilled in order to obtain a glitch-free, correctly working circuit. There are many concrete implementation alternatives to realize such a conversion circuit, and we will present three variants in the following paragraphs which seem to be most promising considering complexity, performance and area requirements. All the proposed solutions have in common that they



(a) (b)

Figure 2.7: Conversion circuit for bundled data to NCL (a) and LEDR (b).

rely on a reliable way of generating the enable signal for the (`line1`-)output latches of Figure 2.7(b):

- As shown in the Figure one can exploit the delay between the request and the acknowledgment. As soon as request toggles, an `XOR` gate generates a high level until the corresponding acknowledge is issued. During this phase the latches of all `line1` signals are transparent. The advantage of this solution is that the duration of the high-pulse is automatically adapted such that all latches actually reach a stable state (LEDR does not acknowledge until all data is in correct phase — and the enable pulse stays high accordingly). The case where only data lines `line0` change may result in an extremely short enable pulse (the phase of each bit changes without the need of any `line1` signal). But as both input and output of the latches show the same value, there is no risk of metastability or inconsistency. The timing constraints for this solution are easy to identify: The inputs of the latches must be stable as soon as `en` goes high ($max(t_1, t_2) < t_3^+$), and need to stay stable until it returns to zero again ($t_3^- < min(t_1, t_2)$).

- Another possibility to create the latches' enable signal is an `XOR` gate with a delayed input (cf. edge generation circuit in Figure 2.8). A pulse with the duration of the delay element in the signal path is generated for each transition of `req`. The duration must be chosen with respect to the latches' timing requirements, but it must also be chosen short enough to be disabled before the next data wave arrives at the converter. The disadvantage compared to the previous solution is to find a proper delay for optimum performance and reliability. The delay element now moved out of the critical signal path and performance can be increased assuming the LEDR module is not too fast.

- A third alternative is to use a four-phase instead of a two-phase bundled data interface. This way, it is possible — similar to the NCL conversion shown at the beginning of this section — to use the request line directly as enable signal for the output latches. However, the converter now needs to translate two-phase to four-phase handshaking protocols (and vice versa) which increases circuit complexity.

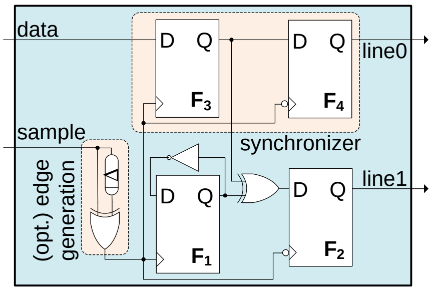


Figure 2.8: Converting single bits to LEDR without back-pressure.

2.3 Free Conversion

In the previous sections we exclusively dealt with interlocked conversion mechanisms, where the consumer is able to signal successful data reception to the producer. The latter is therefore able to wait upon this acknowledgment before sending the next data vector. Both sender and receiver hence operate in a lockstep way, which means that their execution speeds (at least at their interfaces) directly depend upon each other — one step at the producer (new data to send) means one step at the receiver (reception of that very data).

In contrast to the already presented conversion methodologies there might also be a situation where lockstep operation is not possible or desired. For example, state signals (interrupt requests, digital inputs, ...) or serial interfaces (UART, SPI, CAN, I^2C, ...) do not provide the necessary means for applying back-pressure on the sender to achieve an interlocked operation. Often this would not make sense at all, especially when the progression of time itself is part of a signal's or communication protocol's semantics. Using such inputs in asynchronous (QDI) circuits directly implies that the asynchronous modules must meet certain timing requirements[5]. The operating speed needs to be fast enough to reliably sample the respective input signals, and to allow the design to reach a stable state between any two changes of the input signal. Furthermore, for serial communication interfaces, the asynchronous designs must incorporate some notion of time to restore the original serialized bitstream (this issue is addressed in the remaining chapters of this work). Without a dedicated back-channel there is also no known relation between the consumer's execution cycles and the sender's signal transitions. The receiver must therefore treat the input as asynchronous signal, and effectively avoid metastability issues by adequately synchronizing it to the internal timing.

Let us consider Figure 2.8. The circuit shows a design which converts a single-rail input signal to dual-rail LEDR without providing an acknowledgement back to the sender. For this solution we deliberately use edge-driven flipflops as they have well-known properties with respect to metastability and signal synchronization. As we can see, there is an ordinary two-stage synchronizer consisting of sequential gates F_3 and F_4, which simply clock the data signal in and finally provide the data line `line0` for the LEDR output.

---

[5] In that cases we speak of self-timed circuits, as delay-insensitivity contradicts any timing deadlines.

Flipflop F_2 is used to generate the respective phase signal `line1` by XORing the current data signal with the expected phase (stored in F_1). As the phase changes after each execution cycle, F_1 simply alternates it value continuously. Notice that F_1 must be reset according to LEDR's initial phase to prevent deadlocks. There are four possibilities which signal shall be used for clocking (`sample`):

1. **Data Clock:** For synchronous serial interfaces there is a data-clock available (e.g., SPI) which may serve as input to `sample`. In most cases this data clock can be used directly without the need of the edge generation unit (marked as optional in the figure). There is also no need for the 2-stage synchronizer because the data-clock only ticks when data is guaranteed to be stable — thus one flipflop driven by the data clock is sufficient in this case. Then, however, the remaining circuit needs to be adapted as well: F_2 must trigger on the rising rather than on the falling edge, and the XOR gate's input must directly be connected to `data` rather than F_3's output. The subsequent LEDR circuit performs exactly one execution step for each positive data-clock transition.

2. **External Sampling:** Another feasible way is to use a separate signal for `sample`. Wherever this signal comes from[6], each positive transition triggers one execution cycle of the LEDR module. This saves power as the asynchronous block is only active when a sample is taken. Again, the edge generation circuit is not needed.

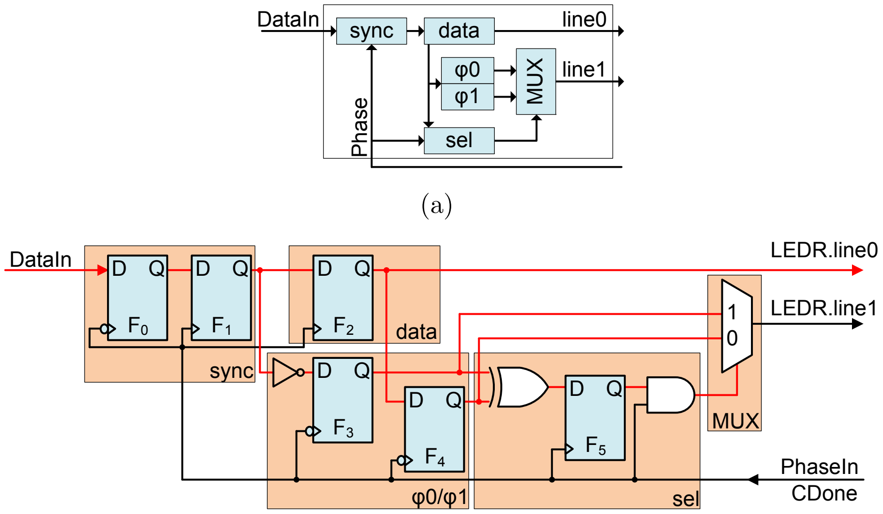
3. **Capture-Done:** Using the LEDR's (or NCL's) capture done signal is the most obvious possibility. However, as LEDR is a two-phase protocol, we need a separate edge generation unit which produces a rising edge for *any* transition of signal `sample`. An XOR gate with one delayed input can be used, but choosing an appropriate Δ is critical for correct operation — alternatively, double-edge flipflops could be used. Notice that in this case the receiver runs at full speed (non-blocking).

4. **Data Line:** For state signals (e.g., switch or button states, interrupt requests, etc.) it is further possible to connect the `data` signal itself to `sample`. Each change of data then again triggers one execution step. For this solution, however, an additional delay needs to be inserted directly after the edge generator to guarantee an appropriate setup time for `data` at the flipflops. Furthermore, there is no need for the 2-stage synchronizer because the timing relationship between data and sample is fixed by the implemented delays in the edge generator. Similar to case (1), one flipflop is sufficient here.

While in general forks should be avoided in synchronizer circuits in order to avoid glitches and race conditions, splitting F_3's output is not a problem as long as it is guaranteed that the `sample` signal's rising and falling transitions do not occur too close to each other so that the flipflops always capture stable (and thus consistent) data. The

---

[6] E.g., a synchronous timer unit which toggles `sample` at the desired bit-rate. This also solves the problem of deriving a suitable notion of time.



(a)

(b)

Figure 2.9: Alternative circuit implementation (b).

XOR in the synchronizer circuit reduces the duration of the available resolution time in case of metastability, which either decreases the circuit's MTBF (Mean Time Between Failure [40, 47]) or the maximum clock frequency. Choosing a suitable Δ for the edge generator has a central influence on the correct functionality of the synchronizer. In particular, it must be large enough to guarantee (up to the required probability) any metastable states to be resolved before the falling edge triggers $F_{2,4}$. As the latter two flipflops trigger on the falling edge, a new data value is already available at the outputs after one asynchronous execution cycle, thus decreasing total latency. Notice that using both rising and falling edges as trigger for synchronizer flipflops is rather unusual. However, in LEDR *each* transition of signal `cDone` indicates an *entire* execution cycle and is thus comparable to one clock period. Another important constraint is the fact that the sampling rate must be slower than or equal to the asynchronous logic's execution speed.

When concatenating more of these modules to synchronize different inputs with the same LEDR block, flipflop F_1 can be used as provider of the next phase for all bits. It is also possible to replace F_1 with LEDR's capture done signal, but this introduces another race condition especially for case (3).

The same basic structure can also be used so synchronize a signal to NCL. A conversion circuit similar to the one in Figure 2.7(a) can be implemented following the two-stage synchronizer. F_1 and F_2 are not needed, also the edge generator can be omitted due to the four-phase protocol of NCL.

2.3.1 Implementation Alternative

The main disadvantage of the free running conversion circuit is the need for an explicit edge generation unit. While such a unit is relatively easy to implement for ASIC de-

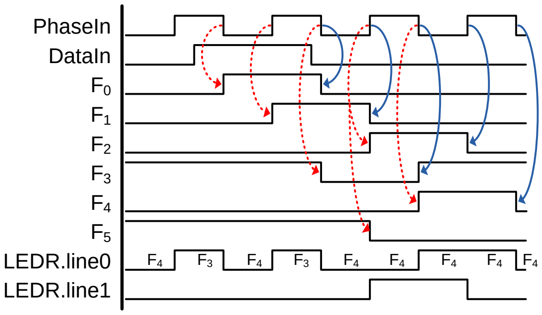


Figure 2.10: Example waveform of synchronizer.

signs, FPGAs are a bit more difficult to control when it comes to explicit delay lines. Especially portability between different FPGA families becomes troublesome, as most tools have their own way of specifying delays. Consequently, we are looking for an alternative implementation strategy with a more general approach. The result is shown in Figure 2.9(b), where logical submodules from Figure 2.9(a) are grouped together by rectangles to simplify the following explanation.

The `sync` block just consists of two cascaded D-flip-flops and forms a 2-stage synchronizer. Similar to the previous section, both rising and falling transitions are used to capture data. The `MUX` block is a simple 2-way multiplexer, which selects between φ_0 and φ_1, respectively[7]. φ_0 is generated directly out of the `data` block, which by itself is only another D-flip-flop F_4 that stores the data-signal for `line1`. Likewise, block φ_1 is created by inverting the incoming data signal in F_3. Finally, block `sel` chooses whether to use φ_0 or φ_1. If `data` is stable, `sel` must alternately select between φ_0 and φ_1, otherwise the `LEDR.line1` needs to maintain its value.

Notice that the *internal* timing constraints of this design are derived from common synchronous design methodologies. As already mentioned above, the solution uses rising and falling edges of the trigger signal `PhaseIn` to interleave all signal assignments and generate a clean LEDR output. Figure 2.10 contains a waveform which illustrates the timing of this circuit. The arcs (dashed and solid) illustrate the dependencies between the input phase and the internal flip-flops. The dashed lines show changes due to the *first positive* transition of `DataIn`, whereas the solid arcs indicate changes caused by the *second negative* transition. For the output `LEDR.line0`, the figure further shows which of the phase blocks is actually selected by `MUX` to generate the output. Element F_2 assigns new data with the rising edge only. Consequently, it makes sense to precalculate φ_0 (F_4) and φ_1 (F_3) with the falling edge of `PhaseIn`. As long as `DataIn` is stable, register F_5 stores constantly logical one ($F_{3,4}$ are stable as well), thereby making the subsequent `AND`-gate transparent. The incoming phase signal toggles the multiplexer and selects the corresponding output. As all input signals of `MUX` are constant and unequal, no glitches occur. On the other hand, as soon as F_1 gets a new value, F_3 changes with the next

---

[7] A standard 2-way multiplexor produces static-one hazards if both data-inputs are 1 and the select-input toggles. Inserting redundant logic can eliminate this behavior.

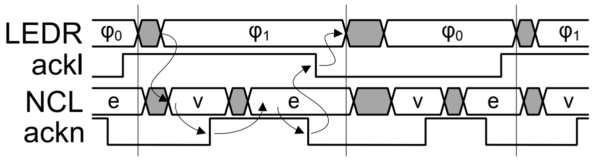


Figure 2.11: Simulation of LEDR to NCL conversion.

falling edge. Now, F_5 will store a logical 0 with the same edge that triggers F_2. At this time, $F_3 = F_4$, so changing the `MUX` selection does not do any harm - the LEDR output is stable. There are two valid situations right now: Either F_1 still has its old value, or F_1 has changed again (at the same time as F_2 and F_5 have changed). The former case is trivial, as $F_3 \neq F_4$ with the next falling edge. This state is similar to the initial state, simply with all flip-flops having inverted values. For the latter case, i.e., F_1 has toggled again, $F_3 = F_4$ also holds for the next cycle and prevents `MUX` from switching. Again, stable output is guaranteed.

2.4 Experimental Results

So far, we introduced various different techniques for converting asynchronous design styles into each other. In this section we present simulation results of the described circuits and discuss the specific properties of these simulations. To this end we generate gate-level representations of the proposed modules (using standard combinatorial gates, latches and Muller-C elements). We assign static propagation delays to all gates and interconnects (of course in a way that all necessary timing constraints are met). In addition, to achieve realistic results and at the same time simulate fabrication variations, all gates and interconnects are subjected to Gaussian jitter. Concrete performance evaluation does not make sense at this abstraction level as no specific target technology is specified. Our toolchain consists of Synopsys Design Compiler as synthesis tool and Modelsim as simulator. Notice that we manually redrew most of the simulator screenshots in order to improve readability. In the presented waveforms the grayed areas indicate durations where data vectors actually change, while non-grayed signals mark stable states.

First we want to consider the conversion from LEDR to NCL. Figure 2.11 shows a waveform obtained by simulating the circuit of Figure 2.4. Both data source and sink are implemented to need arbitrary (but bounded) time for "processing" and `ack` generation. The gray areas between any two consistent data vectors indicate the durations when the respective dual-rail bits are inconsistent (i.e., a mixture of different phases or valid/empty bits, respectively). The sequence of states according to Figure 2.3(b) is indicated by the arrows, starting with state P_4 while waiting for LEDR to apply new data. After the NCL output is consistent (P_1), it will eventually be acknowledged (`ackn+`, P_2), directly followed by applying the empty word (P_3) and the waiting for the respective acknowledge (`ackn-`). Finally, the LEDR part is informed that all data has been stored to start a new transaction (toggle `ackl`, P_4).

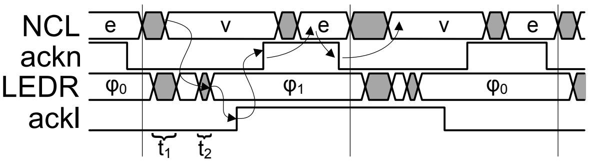


Figure 2.12: Simulation of NCL to LEDR conversion.

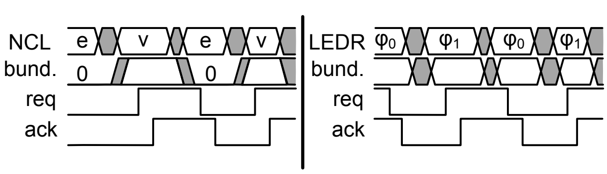


Figure 2.13: Simulation of NCL/LEDR to bundled data conversion.

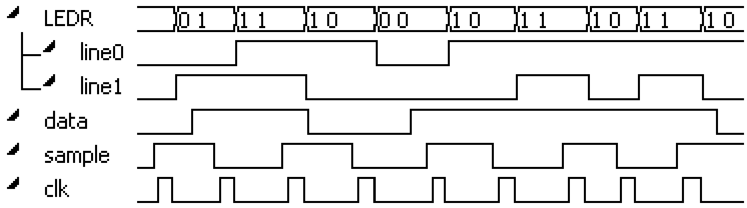


Figure 2.14: Simulation of self-triggered single-bit to LEDR conversion.

The next case we consider is the conversion of NCL signals into LEDR. As illustrated in Figure 2.12, the transitions from empty to valid data bits are directly propagated to the LEDR part (t_1). However, as long as not all bits are valid, the *validity detector* does not issue the `ena` signal for `line1`'s output latch (recall Figure 2.5(a)). As a result, only those bits that actually changed their logical value since the last cycle will toggle during t_1. Only after all NCL bits are valid, `ena` is asserted and the remaining LEDR bits can change accordingly (t_2, we chose a long delay for signal `ena` to better illustrate this behavior). Eventually, all LEDR bits are in φ_1 and `ackl` will be asserted. This finishes the cycle by acknowledging NCL and issuing the empty word.

The next two cases we consider are shown in Figure 2.13. As mentioned in Section 2.2, conversion from any dual-rail design style to bundled data is quite straight forward, as both two-phase and four-phase bundled data protocols exist. In case of LEDR (right part of the figure), a phase detector is used to determine data consistency and generate signal `req`. Each change of phase triggers a new data transmission request, which is consequently acknowledged by the consumer. The phase detector assures that all data bits are stable before the request is asserted. While the overall procedure is similar for NCL (left part of the figure), it is important to notice that the inputs to the bundled data module are all zero in case the empty word is issued. However, as data is captured during the *valid* phase only, this behavior does not incur any restrictions.

Finally, we want to take a closer look on the circuit presented in Figure 2.8. In the simulation waveform shown in Figure 2.14 we connect the *capture done* signal to the sample input (case (3) of the list in Section 2.3), thus the circuit continuously samples the data line in a non-blocking way. The figure also shows a signal named `clk`, which is the output of the edge generation circuit. The delay element has been given a value high enough for good visualization, the exact margins strongly depend on the technology and the achievable synchronization reliability. We can see from the waveform that each falling `clk` edge generates a new LEDR word. By using both rising and falling edge of `clk` it is possible to achieve minimum latency of only one execution cycle.

2.5 Chapter Notes

In this chapter we demonstrated how to construct different asynchronous conversion circuits for two-phase dual-rail, four-phase dual-rail, and bundled data asynchronous communication protocols. Generally speaking, conversion from LEDR as a producer has the advantage that all applied signals are stable for the entire execution cycle, thus simplifying the conversion logic in that no latches (or other sequential elements) are needed. On the other hand, the four-phase protocols are simpler to design (i.e., their area requirements are more beneficial), because state signaling can be handled more efficiently than transition signaling. Compared to related solutions (as presented in Section 2.1) our strategies are optimized for on-chip communication. By exploiting the specific protocol properties we are able to achieve solutions that scale well for large bit-widths and allow good performance with relatively few timing constraints to consider (this also simplifies concrete implementations considerably).

Another aspect we discussed in this chapter focused on interfacing serial communication interfaces without a dedicated back-channel (cf. Figure 2.8). We identified four major alternatives for sampling the incoming data line, each of which with its own advantages and disadvantages. Depending on the application, the generic interface block can be operated in blocking (exactly one asynchronous execution step is performed for each sample) or non-blocking mode (the asynchronous circuit runs freely and continuously samples the data line). An edge generation circuit allows us to achieve very low synchronization latencies of only one asynchronous execution cycle by exploiting both rising and falling signal edges for the flip-flops. As we trigger a new conversion for *each* alternation of the subsequent LEDR-block's phase, the average sampling error introduced is also limited by one execution cycle. On the downside of this solution stands the edge generation unit with the delay element. The delay must be chosen long enough to allow any metastabilities to settle (with the desired probability, also considering setup- and hold-time requirements), but also short enough to not negatively influence system performance. Clearly, the entire block must reach a stable state before the `sample` input toggles again. Especially for FPGAs it is sometimes troublesome to realize a precise delay element. By cascading various buffer elements, the delay can vary considerably from compilation to compilation depending on the exact placement and routing of the entire circuit. Furthermore, adaption of the delay might be necessary in case the frequency of signal `sample` changes. Although the solution is very useful and efficient in terms of performance and area usage, it is not well suited for prototyping circuits that still undergo lots of changes.

In contrast to the generic interface block, the design alternative presented in Section 2.3.1 allows a straight forward, easily portable implementation of a non-blocking single-rail to LEDR converter, which uses only standard gates that are available for virtually all FPGA platforms. The module itself can be constructed using synchronous design techniques while interpreting signal `cdone` as local clock source. On the downside, however, the alternative solution uses considerably more resources. Furthermore, it only provides new data with the rising edge of signal `cdone`, which means that the latency is much higher compared to the generic solution (three execution cycles). As new conversions are triggered by falling `cdone` edges only, the average sampling error is two execution cycles at most.

We will see in later chapters that the presented conversion and interfacing blocks come in handy for the implementation of the TTP controller. Not only the independently running submodules need to be synchronized to each other, also the asynchronous bit stream on the TTP bus needs to be sampled accordingly. Furthermore, control and status signals must be handled using some of the above converters. As mentioned earlier, interchanging different asynchronous design styles also necessitates the existence of suitable converters.

Chapter 3

Temporal Characteristics

> *Gosh, that takes me back. Or forward. That's the trouble with time travel; you can never remember.*
>
> The Doctor, Dr. Who

So far, we have introduced different widely-used asynchronous design styles and described in detail how conversion and interfacing with these implementation methodologies can be achieved. While this knowledge forms important background information especially for Chapters 4 and 5, we now want to take a closer look on the temporal characteristics and timing predicability of asynchronous logic.

We have already pointed out earlier that asynchronous circuits may elegantly overcome some of the limiting issues of their synchronous counterparts. Two prominent potential advantages of asynchronous logic are reduced power consumption and inherent robustness against changing environmental conditions. Especially the latter is important as recent silicon technology suffers from high parameter variations and high susceptibility to transient faults [62]. A substantial part of this robustness originates in the ability of asynchronous (QDI) circuits to adapt their speed of operation to the actual propagation delays of the underlying hardware structures, which is accomplished by the feedback signals for completion detection and handshaking. However, while asynchronous circuits' adaptive speed is hence a desirable feature with respect to robustness, it becomes a problem in real-time applications that are based on a stable clock and a fixed (worst-case) execution time. Therefore asynchronous logic is commonly considered inappropriate for such real-time applications, which excludes its use in an important share of fault-tolerant applications that would benefit from its robustness.

It is consequently reasonable to take a closer look at the actual stability and predictability of asynchronous logic's temporal behavior. After all, synchronous designs operate on exactly the same technology, but hide their imperfections with respect to timing behind a strictly time driven control flow that is based on worst-case timing analysis (which is the root of the current substantial problems with parameter variations). This masking provides a convenient, stable abstraction for higher layers. Asynchronous designs, on the other hand, simply allow the variations to happen and propagate them to higher layers. There is no fundamental obstacle to handle these variations on higher layers (although this may create considerable efforts), thus the interesting question is: Which character and magnitude do these variations have?

In this chapter we want to elaborate a convenient model for describing the temporal characteristics of asynchronous circuits on a qualitative level. We start by exploring jitter in synchronous systems and adapt the classifications for asynchronous designs accordingly [28, 32]. We also perform some experimental case studies in order to validate the proposed model. Based on the results from this chapter we then continue with the actual implementation of an asynchronous time reference generator unit in Chapter 4.

3.1 Related Work

When in comes to temporal behavior of logic circuits, plenty of literature can be found in the field of jitter. Especially for high-speed communication systems, jitter is one of the major factors limiting the maximum transmission speeds by making the “eye” (i.e., the valid area of an eye-diagram [77]) even narrower[1]. For our investigation, we want to start by examining the concepts of signal jitter for synchronous systems, which shall form the foundation for our asynchronous jitter definitions in Section 3.2.

3.1.1 Jitter in Synchronous Circuits

In synchronous systems we have the abstraction of an equally spaced time grid to which all transitions are aligned, and all deviations from this ideal behavior are commonly subsumed under the term jitter. Often jitter is associated with a synchronous clock source like a crystal oscillator, where it is obviously an undesired effect. Consequently, attempts have been made to identify the different sources and effects of jitter in order to mitigate the most relevant ones.

Literature generally distinguishes deterministic and random (indeterministic) jitter, as illustrated in Figure 3.1. The term *random* thereby refers to the statistical and thus random characteristics of jitter, and by that the corresponding magnitude is in principle unbounded. In contrast, *deterministic* jitter sources have well-defined origins, are always bounded in magnitude, can basically be predicted, and are thus reproducible[2]. The following list shortly explains the most common sources of jitter [78, 85, 95]. Notice that the single items are *not* mutually exclusive – even worse, measurements mostly indicate a combination of several if not all of these types.

---

[1] In combination with relatively shallow slopes, tight voltage ranges, and very high bit rates.

[2] Notice that random effects may also have well-defined origins and be reproducible, but this only applies for their statistical parameters.

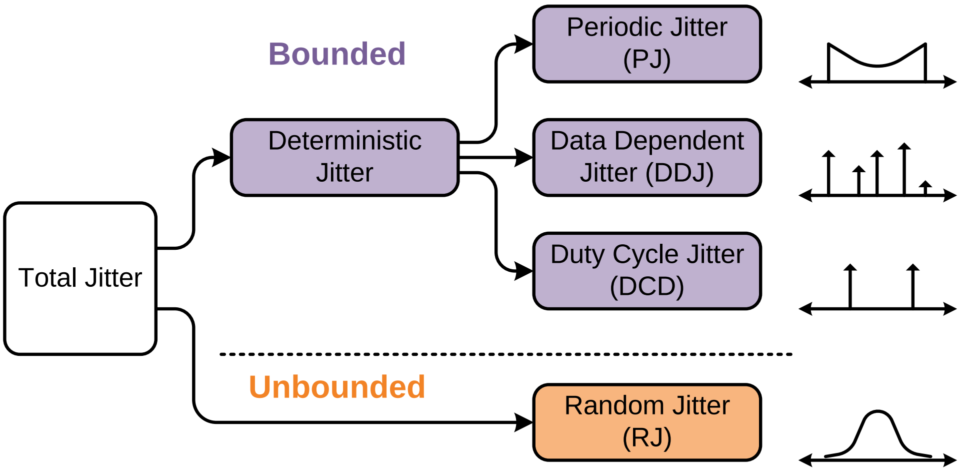


Figure 3.1: Jitter classification scheme (Source: [85]).

- *Data Dependent Jitter (DDJ)* is added to a signal according to the sequence of processed data values. Intersymbol interference (ISI), simultaneous switching noise (SSN), etc. are common sources of DDJ. Furthermore, data-dependencies can be observed in cases where the voltage level does not reach its absolute end value during one bit time. Then, if two equal bits are sent in succession, the voltage fully settles and the next opposing transition will take longer to reach its respective threshold value (cf. Figure 4.1(b)). In a jitter histogram, DDJ can often be identified as multiple separated peaks, caused by the concrete influence of the actual data values on the physical signals.

- *Bounded Uncorrelated Jitter (BUJ)* (which is not shown in the figure) is used to model crosstalk effects from *other* transmission lines. Consequently, the resulting jitter is uncorrelated to a communication channel's *own* data stream [53].

- *Duty Cycle Dependent Jitter (DCD)* has its origin in differences of the slopes of rising and falling signal transitions. High and low pulses of periodic signals appear to have different lengths, which manifests as two distinct peaks in the jitter histogram (as illustrated in Figure 3.1). A similar effect can be observed (even in case of matching slopes) if the decision threshold for binary values is not at 50% of the supply voltage V_{DD}.

- *Periodic Jitter (PJ)* is induced by periodic external events, such as switching power supply noise or strong local RF (Radio Frequency) carriers, and is per definition uncorrelated to any data-streams. Due to its periodicity, pronounced peaks in the corresponding FFT (Fast Fourier Transformation) plots can be identified, for which reason it is also called sinusoidal jitter. In jitter histograms the characteristic curve of PJ often looks like a bathtub when jitter continuously changes between two periods.

- *Random Jitter (RJ)* can be seen as the (statistical) sum of multiple uncorrelated random effects (e.g., thermal or supply voltage noise). Although in theory any probability distribution is possible, it is usually assumed that RJ has Gaussian-like characteristics for modeling[3]. Due to its random nature RJ is not predictable and in principle unbounded in magnitude.
- *Process and fabrication variations* introduce significantly different timings and delay characteristics among different devices. In that sense, they are not directly related to the other jitter types of the above list, because they can only be observed when comparing timing characteristics of *different chips* to each other.

Again notice that in practical circuits we typically observe superpositions of different types of jitter. It is therefore an intricate task to distinguish them in a measurement, even though powerful support by special jitter oscilloscopes is available. With the above abstract classification in mind, we can now derive concrete definitions of jitter [78,94,95,97] for periodic signals.

- *Timing or Absolute Jitter* is the deviation of a signal transition from its ideal position. For a clock signal this means that the nominal values are integral multiples of the clock period T, and the measured (absolute) deviation for each transition is called timing jitter. This type of jitter accumulates over time and always specifies the absolute deviation from the nominal value.
- *Period or Cycle Jitter* is the deviation of a signal's period from its nominal value. Period jitter is determined separately for each cycle. In other words, it is not accumulated over multiple cycles.
- *Long Term (Accumulated) Jitter*, on the other hand, is defined as deviation of the measured *multi-cycle* time-interval from the respective nominal value. Especially random jitter accumulates over time, and thus its absolute value increases when observing long time intervals. Considering the interval error for multiple cycles is a generalization of period jitter, which only accounts for a single cycle.
- *Cycle-to-Cycle Period Jitter* is the variation in the deviations of cycle-periods of *adjacent* cycles compared to their nominal cycle times. In other words, cycle-to-cycle jitter is the second order difference of the measured cycle periods [95].

Figure 3.2 graphically illustrates the different types of jitter mentioned above. According to Figure 3.2(a) we assume a periodic signal with nominal period T, whereby the actual transitions jitter around the average period T. Consequently, timing jitter (Figure 3.2(b)) can be identified as the difference from the nominal to the actual period, as shown in the following equations. In these equations, the timing jitter of the n-th period is denoted as ΔT_n, while nT defines the reference time after n periods, and T_n the

---

[3] Thermal noise follows a Normal distribution, and the composition of many (uncorrelated) noise sources also approaches a Gaussian distribution [85].

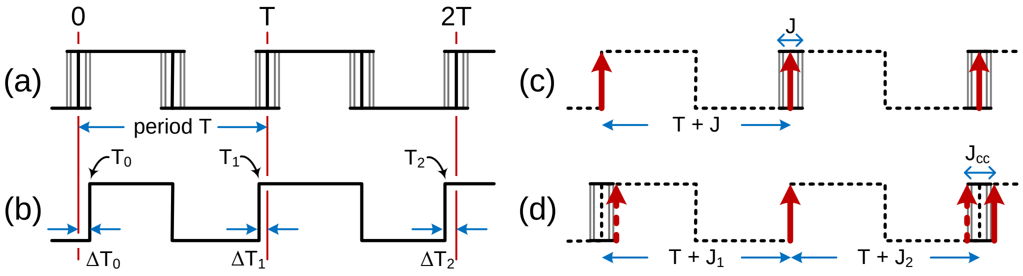


Figure 3.2: Jitter manifestations: General (a), Timing jitter (b), Period jitter (c), and Cycle-to-Cycle jitter (d) (Source: [94]).

respective *actual* time. Notice that timing jitter defines the *absolute* deviation of the n-th transition of a square-wave signal from its nominal value.

$$\Delta T_n = T_n - nT \tag{3.1}$$

Likewise, Figure 3.2(c) shows the case for period jitter J, which is defined as the instantaneous period minus the nominal period T (Equation 3.2) [94]. The *instantaneous period* for cycle n can be expressed using the (instantanous) angular frequency $\omega[n] = 2\pi f[n]$, and shall highlight the continuously changing nature of $J[n]$.

$$J[n] = \frac{2\pi}{\omega[n]} - T \tag{3.2}$$

Finally, Figure 3.2(d) illustrates cycle-to-cycle jitter, which is defined by Equation 3.3 and directly follows from the difference of period jitter of adjacent cycles [94].

$$J_{cc}[n] = J[n+1] - J[n] = \frac{2\pi}{\omega[n+1]} - \frac{2\pi}{\omega[n]} \tag{3.3}$$

On the basis of these definitions, long term accumulated jitter can either be defined similar to timing jitter (but over a long period m and under the assumption $\Delta T_0 = 0$), or using period jitter J, which is expressed in Equation 3.4 for the observation/reference interval $(0, mT)$.

$$J_{lt} = \sum_{i=1}^{m} J[i] = \Delta T_m \tag{3.4}$$

Although both timing and period jitter can be used to express long term accumulated jitter, it is important to notice the subtle difference between these two types. For instance, ΔT_n gives no indication about the actual duration of the respective periods (while J does). Furthermore, period jitter is only applied to one single period, while timing jitter is an absolute measure of error, even over multiple cycles.

3.1.2 Timing Analysis

Although the timing model we want to elaborate is not intended as tool in the same sense as industry uses *timing analysis* to predict the (worst-case) performance of a given circuit, there are some significant similarities with respect to our needs after all. In this section we therefore give a concise overview to the complex topics of static-timing analysis (STA) and statistical static-timing analysis (SSTA), based on the work presented in [7,57,66,67]. Especially [7] provides basic information in combination with more detailed insight to both STA and SSTA, while the other references mostly focus on specific problems associated with SSTA. For timing analysis of digital circuits STA has so far been an efficient and effective way of determining a circuit's (worst-case) delays and retrieve the achievable performance. Four major factors can be identified which account for the widespread use of STA in industry [7]: (i) Basically, STA scales linearly with circuit size, allowing for complex circuits to be analyzed efficiently. (ii) Analysis results can be considered conservative, which means that the estimated performance is guaranteed under certain environmental conditions. (iii) The STA algorithms themselves are relatively sophisticated and address many different timing issues. (iv) Last but not least, the process of deriving delay characteristics for given cell libraries is well defined.

The deterministic result of STA has many years been a sufficient measure of the chip's performance. However, in order to address manufacturing variations (which become increasingly significant for deep sub-micron devices), STA usually uses so-called *corner files*, which contain timing characteristics of gates under specific process conditions (e.g., varying gate-widths). By repeatedly executing the STA algorithms on a given design using different corner files, the expected performance under changing process parameters can be estimated. Notice, however, that corner files only simulate *die-to-die* variations, as all gates of the chip are modeled with the same parameters. Therefore, *within-die* variations cannot be handled by traditional STA. It has been shown in [7] that this fact may — for certain circuit topologies — lead to either over- or underestimations of propagation delay, which mainly depends on the correlation assumptions made between different paths[4]. SSTA on the other hand tries to addresses the increasing process variability by including statistical analysis in the algorithms. Three major approaches have been developed over the past few years [7]:

- *Numerical-Integration Method:* This approach uses numerical integration over the entire process parameter space in order to derive the circuit delay for critical paths under all conditions. While this is a very general and flexible method, numeric integration over the entire (feasible) parameter space is an extremely time-consuming task, especially for balanced circuits with many potential critical paths.

- *Monte Carlo Method:* This approach is based on traditional STA. Process parameters are (randomly) chosen according to their statistical distribution, and deterministic STA is performed for each fixed set of parameters. After a sufficient number

---

[4] This is also true for SSTA when unrealistic correlation assumption are made to simplify the calculation models.

of "samples" the probability distribution of critical paths can be estimated and the timing yield can be found. Similar to before, the main drawback is the associated computation time (due to the huge number of runs). On the upside, existing and sophisticated timing analysis tools can be used for simulations, and the technique is completely general in the sense that arbitrary process parameter distributions can be used. As we will see in the next sections, our solution is also based on the Monte Carlo Method.

- *Probabilistic Analysis Method:* In contrast to the methods above, where the entire sample-space is enumerated, this technique models *gate delays* and *signal arrival times* as random variables. By applying statistical sum and maximum operations, arrival times (i.e, their probability distribution functions) are propagated through the timing graph and a probability distribution function is obtained for each critical path, allowing to estimate the achievable timing yield. However, implementing the statistical operators is not trivial and execution speed rather time consuming. The approach also looses some generality as often normal distributions are assumed in order to simplify statistical calculations.

3.2 Jitter in QDI Circuits

Before we actually start with the elaboration of a suitable timing model for LEDR circuits, it is necessary to take a closer look at jitter in asynchronous systems. Compared to the synchronous case, measuring jitter effects in asynchronous circuits is somewhat different. Taking into consideration the above definitions of jitter manifestations, we see that (except for cycle-to-cycle jitter in Equation 3.3) the nominal period T is present in all equations. In this sense, however, the question arises: What is the nominal period for asynchronous systems? (Quasi) delay insensitive circuits in general, and LEDR designs in particular do not have a predefined operating speed, consequently there is no nominal point in time where the transitions are *supposed* to occur. The property of delay insensitivity makes it impossible to define such ideal behavior even on a conceptual level, which renders the above definitions more or less useless for our purposes. However, also in our asynchronous design there are some signals that must have a predefined frequency for the system to work properly (e.g., reference time, bit-timing, macrotick generation, etc.). For these signals nominal properties (period/frequency, phase, etc.) exist and the common synchronous jitter definitions can be applied accordingly. The remainder of this section deals with the case where no reference signals can be found (i.e., the free-running asynchronous modules).

The first and most central point is that we do not operate in the multi Gigahertz communication domain. In contrast, our purpose is to establish an accurate-enough notion of time to allow for relatively low baudrates (in the range of 100kHz to 1MHz), distributed clock synchronization and macrotick generation (both in the range of milliseconds). Instead of tweaking communication channels towards their physical limits, what we need is to generate a stable time reference. Jitter itself is, on a high level of abstraction and within certain bounds, not the central problem at all — the *average* frequency, on the other hand, needs to be stable after all. While some of the presented jitter sources contribute only very little jitter (in the picosecond domain, e.g., crosstalk induced jitter, SSN, etc.) and thus play a minor role for our timing model, new sources and types of jitter must be defined in order to accommodate for the specific properties of asynchronous logic. The absence of the clock signal has severe influence on jitter: Most importantly, clock related jitter sources such as duty cycle distortion or uncertainties introduced by the clock distribution network need not be considered any more. On the downside, asynchronous systems need to deal with all the timing characteristics and jitter sources that are usually indirectly eliminated by the edge-triggered flip flops and high-precision oscillators (refer to item "Data Dependent Execution Jitter" in the list below for a more detailed explanation).

Considering once again the exemplary LEDR circuit of Figure 1.11 on page 17, there is yet another more fundamental question to answer: When there is no clock signal, how can we measure and classify the execution speed of our system? The phase of any register, represented by the respective `cDone` signal, is a suitable measure of execution speed, as it changes exactly once per execution cycle. The inherent handshaking guarantees the *average* rate of execution cycles for all coupled registers to be the same. However, due to the fact that LEDR circuits are "elastic", there may be substantial differences in the execution speeds of adjacent pipeline stages for consecutive cycles. We can now define *execution jitter* to be the deviation in the durations of execution steps. Notice that there is no dedicated (ideal) reference period T, and simply defining $T = \mu_T = \frac{1}{n}\sum T_i$ to be the average over all sampled periods $T_i, i = 1 \ldots n$ is dangerous for large n. The reason is that μ_T is not stable over time, as the operating speed of a logical device continuously drifts (e.g., temperature changes due to warm-up or external cooling, supply voltage drops due to additional consumers, etc.). The corresponding variance (i.e., the signal jitter) would then be considered much higher than expected due to the drifting mean value. In practice one should therefore either use reasonably short measurement periods, or consider cycle-to-cycle jitter instead. The first order difference[5] practically eliminates even drifting constant components. The following list gives a detailed overview to the jitter types and sources we want to investigate separately in this work. In order to distinguish the new definitions from the original ones, we generally label them as *execution* jitter to indicate its asynchronous nature.

- **Data-Dependent Execution Jitter (DDEJ):** Similar to the common definition of DDJ, this kind deals with cases where the actual data values induce jitter on a signal. However, and quite in contrast to synchronous systems, DDEJ has different origins and has a much more pronounced influence on signals. The reason for this is the missing clock signal, which usually synchronizes all internal signals to a defined transition. For example, consider the most significant bit of a simple ripple-carry adder. Depending on the current state of the adder, the MSB assumes

---

[5] In the asynchronous case, cycle-to-cycle jitter is only a first order difference as there is no predefined signal period T.

its final value after different durations. If the MSB changes due to propagation of the carry signal through all preceding bits, the calculation will take considerably longer compared to the case where the bit is set directly (without carry). However, in a synchronous design, the final value is not made public before the next clock transition — independently from the actual completion time of the calculation. In LEDR designs, on the other hand, all operations are performed asynchronously and can thus finish at arbitrary times. This naturally generates large amounts of jitter on a signal. Notice that our definition of DDEJ also subsumes all remaining data dependent jitter effects (such as inter symbol interference), even if their magnitude can be expected to be marginally small in comparison. We will further discuss this important type in Section 3.3, especially with respect to the specific properties of LEDR circuits.

- **Bounded Uncorrelated Execution Jitter (BUEJ):** A typical example of this kind of jitter is, similar to the already presented definitions, crosstalk. However, the magnitude of BUEJ can be expected to be negligibly small four our considerations.

- **Random Execution Jitter (REJ):** We define random execution jitter to subsume all *random* jitter effects, such as local thermal or voltage noise. In that sense the definition is not different from the original one. It is just important to realize that the magnitude of REJ can be expected to be substantially higher compared to synchronous circuits. The reason is again the missing clock signal, which at least partly hides accumulated (random) jitter effects behind edge driven flip flops. In LEDR circuits, however, a signal propagates through different stages and thereby accumulates large amounts of random jitter, which are directly visible at the respective endpoint/output.

- **Process and fabrication variations:** For our investigation, these properties are of central interest, because they severely influence a chip's overall operating speed (refer to Section 4.4.5 for details on fabrication variations and their influence on execution speed). However, fabrication variations for a specific device are fixed (neglecting aging effects, which are so slow that we can safely ignore them), and can only be observed when comparing different devices to each other[6]. We will discuss this issue again when we investigate the impact of process variations, supply voltage and operating temperature on asynchronous designs.

From an abstract point of view, we can categorize jitter into two major groups. On the one hand, *systematic jitter* (DDEJ, global voltage and temperature change, e.g.) describes all effects that can be reproduced by our system setup. Consequently, for a given circuit, if we apply the same input transitions in the same state under the same operating conditions, we may expect the delay to be the same as well. If this is not the case, *random jitter* (BUEJ, REJ: local voltage and temperature fluctuations, noise, ageing, e.g.) has been

---

[6] In that context we do not treat process variations as *jitter*, but as static influence on delays. This is also expressed in the timing model of Section 3.3.2.1.

experienced. Obviously, the latter cannot be controlled by the system setup. Recalling the different types of jitter from Section 3.1.1, we consider DDJ and PJ to be systematic, while BUJ, DCD and RJ are considered random. Although BUJ and DCD seem to be systematic after all (they show very little dynamics, which are almost constant over a chip's lifetime), it is hardly possible to influence them by means of system setup. The classification as "random" therefore seems adequate; it can also be expected that they are not a major source of frequency instabilities for QDI circuits.

Notice that the above definition of data-dependent execution jitter does not directly fit the commonly used definition: As will be shown below, data-dependencies are induced at gate-level, mostly because signals take different paths through the combinational logic depending on the current state and thus need different amounts of time to complete. Another reason is that the actual value and the respective signal transitions inside the gates result in different gate-propagation delays (e.g., falling edges may have sharper slopes than rising edges, or zeros may propagate more quickly through a gate than ones). The original definition refers to effects at the physical/electrical level caused by data-dependencies, rather than on gate-level effects (which are usually masked out by the global clock signal in synchronous systems).

3.3 Circuit Timing

Keeping the above classification of jitter sources in mind, we now examine sources of data-dependent and random jitter from a logic designer's point of view. It is important to realize that the main goal of our approach is to better understand the temporal characteristics of asynchronous circuits. We want to investigate the influence of PVT variations on a specific design, rather than build another tool for detailed timing analysis — STA and SSTA already provide adequate techniques for this purpose. Our model is intended to operate on a relatively high level of abstraction (i.e., LEDR-gate level): Figure 3.3 shows an example circuit with two gates A, B, five interconnect delays $\Delta I_{1,2,3,4,z}$, and two gate delays $\Delta G_{A,B}$. For reasons of simplicity the gates are single-rail only, but due to the modular approach generalization to dual-rail is relatively simple. Let us further define the arrival times of input and output transitions at the ports a, b, c, z as t_a, t_b, t_c, t_z, respectively. Likewise, t_1, t_2, t_3, t_4 shall denote the signal arrival times at the gates' inputs, and t_A, t_B represent the points in time when a transition actually reaches a gate's output. Under the not necessarily valid assumption that the gates' input-to-output delays ΔG_A and ΔG_B are the same for both input ports and any combination of input values, deterministic static timing analysis evaluates the maximum delay ΔP_{max} for the given circuit as:

$$\begin{aligned}\Delta P_{max} = \max(&\Delta I_1 + \Delta G_B + \Delta I_z,\\ &\Delta I_2 + \Delta G_A + \Delta I_4 + \Delta G_B + \Delta I_z,\\ &\Delta I_3 + \Delta G_A + \Delta I_4 + \Delta G_B + \Delta I_z)\end{aligned} \tag{3.5}$$

This equation is especially valid for the synchronous case where ports a, b, c are directly driven by clocked registers ($t_a = t_b = t_c = 0$). Otherwise, analysis must be extended to

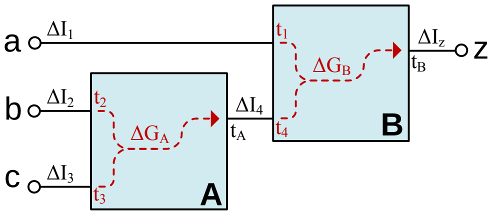


Figure 3.3: Timing model, example circuit.

also include concrete signal arrival times at the circuit's input ports. In the case of STA all necessary propagation delays are well known as soon as a concrete circuit layout has been created for the desired technology, and the analysis returns the critical path of the circuit. The overall delay ΔP_{max} must not exceed the duration of one clock period, already including margins for timing uncertainties and setup-/hold-times of the flip flops. The case would be more complex for SSTA, especially if all three paths have approximately the same delay. Statistical analysis must then be performed for all three paths instead of only one, and the results must be evaluated against the desired timing yield. While modern STA is far more complex compared to this basic model, this deliberately simple view is sufficient for now in order to demonstrate some important properties that must be considered for asynchronous circuits. Later on, we will enhance the model to account for more complex gate implementations, random timing variations (jitter) and process variations.

3.3.1 Data-Dependent Execution Jitter

In this section we demonstrate the sources of data-dependent execution jitter using a simple example. We first consider the case of ordinary synchronous logic before extending the explanation to LEDR circuits. Again consider Figure 3.3, and assume both gates to be OR gates, all $\Delta I = 2$ time units and all $\Delta G = 1$. For simplicity, we denote as $\Delta x = t_z - t_x$ for $x \in \{a, b, c\}$ the propagation delays from the input ports to the output z, which results in delays of $5, 8, 8$ time units for $\Delta a, \Delta b, \Delta c$ in this special case, respectively. Having all inputs set to 0 and simultaneously changing a, b to 1 results in $z = 1$ after a delay of $\Delta a = 5$ time units. Gate B being an OR gate makes Δa the dominating path as the gate does not need to wait for the second input. On the other hand, if after some time both a and b are (simultaneously) reset to zero, the output transition at node z is now delayed by $\Delta b = 8$ time units. A similar situation can be observed if only c changes to 1 in the initial state, which results in a positive output transition after time $\Delta c = 8$. So even for this extremely simple case with just two gates and matching delay values we can observe significantly different delays. One can imagine more complex circuits with even more complex timing assumptions to have a quite pronounced distribution of possible propagation delays. In synchronous circuits all those variations are masked by the global clock, and the only delay of interest is that of the critical path. Although it is also evident from the presented example, it is important to notice that the observed variations solely depend on the circuit structure, the associated logic function and the current *state* of the system (i.e., the actual data values) — hence the name *data-dependent execution jitter*. Based on these facts, a transition from some specific state $S_1 \rightarrow S_2$ always leads to the same propagation delays. Out of this analysis it is easy to see that DDEJ is systematic, although difficult to predict for complex paths, especially under the presence of significant process variations.

Since we are operating at gate-level (with relatively large delays), transistor-level effects such as the Charlie- or Drafting-effects [96], as well as SSN (Simultaneous Switching Noise [11]) can be neglected. The latter is covered indirectly by DDEJ anyway. However, there is a comparable behavior for the Charlie-effect at gate-level as well (which directly follows from the delay assumptions we made): We need to look at the relative arrival times of input transitions at gates. Assume in Figure 3.3 that the falling edges of inputs a, b arrive simultaneously, which results in the output changing to zero after $\Delta_b = 8$ time units. However, if input b arrives 3 time units before a (i.e., $t_a = 0, t_b = -3$), we evaluate $t_1 = 2$ and $t_4 = 2$. Consequently, output z already changes after 5 time units *after the arrival of the last input signal*. To put it in other words, a larger separation between different input events might lead to a bounded decrease in total propagation time. This behavior is especially true for asynchronous logic, as signals feeding combinational logic are relatively loosely coupled rather than changing simultaneously.

Extending the model to LEDR circuits seems quite straightforward because it can be applied hierarchically by simply replacing the 2-input gates with more complex (dual-rail) LEDR gates. Although this is basically true, the problem with LEDR circuits is their property to be *strongly indicating* [81]. This means that the realized logic functions always exhibit worst case performance, because each LEDR gate needs to wait until *all* of its input are present in the same phase before actually changing the output. For the example from above this means that even if $t_1 < t_4$, B holds its current output until the second input arrives at time t_4 and only then changes the output accordingly (resulting in a worst case delay of $\max(\Delta a, \Delta b, \Delta c)$). However, also LEDR gates show considerable data-dependent jitter because their internals are based on ordinary single-rail components (and consequently the very same effects as described above can be observed). Making the delay assumptions of a single gate more complex allows us to model the actual behavior of LEDR gates more accurately — thereby maintaining the high level of abstraction and allowing to better understand the temporal characteristics of LEDR circuits on LEDR-gate level. Most importantly we need to deal with dual-rail inputs (and outputs), delay dependencies with respect to the current execution phase, varying delays due to different arrival times of inputs, and many more.

3.3.2 Timing Variations

According to Figure 3.3 we just distinguish two kinds of delays: *Interconnect* and *gate* delays. Both types are subject to process, voltage, temperature (PVT) variations and can therefore not be considered constant. Adequate modeling of these dependencies is necessary to gain further insight to the temporal behavior of asynchronous circuits. As we operate on FPGA devices for our experiments, it is worth noticing that interconnect delays also include the delays introduced by FPGAs' interconnect switches — we do not model them separately. The resulting model is in turn used as basis to perform Monte Carlo simulations (recall Section 3.1.2). Given a LEDR circuit, we define interconnect and gate delay parameters for all LEDR gates and assign static and random delays to the single components. After performing a reasonable number of simulation runs the results reflect the expected circuit behavior under the given probabilistic conditions. The main advantage compared to traditional deterministic timing analysis and simulation is that we operate on a relatively high level of abstraction (LEDR-gates). This allows us to include jitter, voltage and temperature effects in the simulations while still keeping the simulation time low. For post layout simulations of FPGA devices it is — depending on the tools — not always possible to include indeterministic timing effects in the simulations on a low level (e.g., on signal level), because neither the timing files nor the simulation tools support it. However, the listed advantages come at the cost of reduced accuracy as all LEDR gates are modeled in the same way, while timing variations due to different placement and routing parameters are not considered.

3.3.2.1 Interconnect Delay

We first consider interconnect delays which are labeled ΔI in Figure 3.3. Basically, the resulting delay can be written as

$$\Delta I_x = i(p(\Delta_x, P_x), V, T, J_x) \tag{3.6}$$

where ΔI_x denotes the interconnect delay of a specific segment x, $i()$ is a function returning the actual delay depending on the specified parameters. Δ_x defines the static delay, and P_x, V, T, J_x the process variation, voltage, temperature, and random jitter component for interconnect instance x, respectively. For simplicity we assume both voltage V and temperature T to be the same for all instances over the entire chip (thus omitting the subscript). The static delay Δ_x is the nominal delay according to the length, width, thickness, and spacing of the wire segment. It is subject to process variations P_x, which leads to an overall static delay given by function $p(\Delta_x, P_x)$. Neglecting aging effects this latter term is considered constant for each segment over a chip's lifetime. The random jitter component is modeled as normally distributed probability function $J_x = N(0, \sigma)$ with $\mu = 0$ and a standard deviation $\sigma \propto p(\Delta_x, P_x)$ being proportional to the static delay of segment x. On the other hand, process variations are modeled to be uniformly distributed. Function p randomly varies the nominal Δ_x with a factor of interval $[1 - P_x, 1 + P_x], 0 \leq P_x < 1$.

Given the fact that $p(\Delta_x, P_x)$ is only evaluated once per segment and then kept constant, only V, T, J_x remain as variable sources for delay. Further assuming V, T to be constant for an entire simulation run, J_x is the only source for timing uncertainties and will consequently always lead to a Gaussian distribution. If necessary, the model can further be extended to include thermal and power supply noise as well. In that case, V and T must not be considered constant for all instances, but need to be defined as random variables themselves (e.g., $V = N(V_{nom}, \sigma_V)$, $T = N(T_{nom}, \sigma_T)$). One is not restricted to Normal distributions for noise effects, any reasonable probability function can be used accordingly. While the random jitter component J_x is considered independent (and to follow a Gaussian distribution) for all instances by definition, V and T can also be used to model correlations among adjacent segments. Due to their physical proximity, adjacent segments tend to have correlated temperature and/or supply voltage. Sharing the same random variables for V, T among several instances covers this correlation to a certain extent (the same is true for process variations). However, this requires placement information, which is usually not available at early design stages.

The physical parameters affecting delay of an interconnect segment are resistance R and capacitance C. Besides the exact geometry, these parameters depend on wire length, thickness, width, and material [69]. However, capacitance is independent of temperature, while the resistance changes significantly with temperature. This relationship is also shown in Equation 3.7, where k_T is the material and temperature dependent scaling factor [69].

$$\Delta I \propto R(T)\ C = k_T\ R\ C \tag{3.7}$$

3.3.2.2 Gate Delays

From the theoretical point of view, gate delay modeling is quite similar to the already presented interconnect delays. One main difference is the introduction of data-dependent delays. As already mentioned, not only the logic function itself, but also the relative arrival times of inputs have a significant effect on the gates' overall propagation delays. All other characteristics are basically the same as above, with the difference that some of them have another impact on gates than on interconnect wires.

$$\Delta G_y = g(p(\Delta_y, P_y), V, T, J_y, d_y) \tag{3.8}$$

In this equation, function $g()$ maps the specified parameters to the respective gate delay of gate instance y. As one can see, the parameters are basically the same as in the previous section. There is just one new argument d_y, which defines the actual data values of a gate's inputs. This parameter is of crucial importance to model the data dependent effects mentioned in Section 3.3.1, and holds all important information of a gate's inputs: The old and new data values of each signal, the corresponding phase, and the actual arrival times at the gates' ports. Out of this information, the resulting delay ΔG_y is computed. Notice that this scheme only works for strongly indicating circuits, because calling function $g()$ implies that the arrival times of all input signals are already known — early propagation of signals cannot be modeled directly. However, LEDR circuits preserve their output until all inputs have performed a change of phase, thus calling $g()$ not before *all* inputs are updated is legitimate (the early input will not be propagated anyway).

Another significant difference compared to interconnects is related to the impact of voltage and temperature on a gate's performance. The delay τ of a given gate depends on the transistor's resistance R and the respective load capacitance C_L [69, 70], as shown in Equation 3.9. The resistance R can be substituted by its respective voltage-to-current ratio, whereby V_{dd} is the transistor's supply voltage and I_d is a complex combination of technology dependent constants α and β, the supply voltage V_{dd} and the threshold voltage V_{th} (see Equations 3.10 and 3.11). Without going into much detail, the *velocity saturation index* α is determined empirically. The capacitance per unit area of gate oxide is described by parameter C_{ox}, the carrier mobility by symbol μ, and the gate's width and length by W and L, respectively [69]. Combining Equations 3.9 to 3.11 and substituting all constants with symbol K finally leads to the expression for gate delay [70] shown in Equation 3.12.

$$\tau = C_L \; R = C_L \frac{V_{dd}}{I_d} \tag{3.9}$$

$$I_d = \beta \; (V_{dd} - V_{th})^{\alpha} \tag{3.10}$$

$$\beta = \mu \; C_{ox} \; \frac{W}{L} \tag{3.11}$$

$$\tau = K \; \frac{V_{dd}}{(V_{dd} - V_{th})^{\alpha}} \; \mu^{-1} \tag{3.12}$$

Equation 3.12 is of central importance as it illustrates the relationship between gate delay τ, supply voltage V_{dd}, threshold voltage V_{th}, and carrier mobility μ. However, neither carrier mobility nor threshold voltage are constant, but depend on the operating temperature T:

$$\mu(T) = \mu(T_r)(\frac{T}{T_r})^{-k_{\mu}} \tag{3.13}$$

$$V_{th}(T) = V_{th}(T_r) - k_{vt}(T - T_r) \tag{3.14}$$

In these equations, T_r defines the reference temperature, and k_{μ} and k_{vt} are empirical constants usually in the range of $(1.2, 2.0)$ and $(0.5, 3.0) mV/K$, respectively [69]. Carrier mobility decreases with higher temperature and thus leads to increasing gate delay τ. On the other hand, the threshold voltage gets smaller and therefore *slightly* compensates temperature degradation. To summarize, one can find the following relationship between gate delay τ, voltage V_{dd} and temperature T:

$$\tau(V, T) = K \; \frac{V}{(V - V_{th}(T))^{\alpha}} \; \mu(T)^{-1} \tag{3.15}$$

3.4 Case Studies

We now present a couple of elementary circuits and study their characteristics according to the properties elaborated in the previous section. All designs consist of a free-running, closed-loop LEDR circuit with two registers (one of which being a shadow register with a phase inverter, recall Section 1.3.3 stating that direct feedback is not possible). The first two examples are a 4-bit and a 16-bit counter, respectively. Both counters are realized as

	CNT 4bit	CNT 16-bit	CNT 16-bit opt.	CNT 64bit
LEDR-Gates (+inv.)	$5 + 1$	$29 + 13$	$109 + 47$	$125 + 61$
LEDR-Registers	$2 * 4$	$2 * 16$	$2 * 16$	$2 * 64$
Logic Depth	3	15	7	63
Avg. Speed	$16ns$	$25ns$	$16ns$	$94ns$
Avg. Speed, sim.	$22ns$	$36ns$	$22ns$	$134ns$
Counting Period	$262ns$	$1.66ms$	$1.02ms$	n/a
Execution Jitter	$1.19ns$	$400ps$	$800ps$	$3.08ns$
Counting Jitter	$1.13ns$	$46ns$	$31ns$	n/a

Table 3.1: Characteristic figures of example circuits.

ripple-carry adders, thus the logic depth increases linearly with the widths of the counting registers. The third design again is a 16-bit counter, but this time with optimizations turned on. While this significantly decreases logic depth, it at the same time increases the necessary number of LEDR gates and produces a relatively balanced circuit. The final example (for the measurements in Section 3.4.2) use a 64-bit counter.

The characteristic figures of each design are summarized in Table 3.1. Row “LEDR-Gates” and “LEDR-Registers” specify how many dedicated combinational LEDR-gates and registers are used, respectively. Notice that all registers are duplicated (shadow registers), which is indicated by the multiplication factor of 2. “Logic Depth” defines the maximum number of LEDR-gates connected in series (excluding registers), and “Avg. Speed” and “Avg. Speed, sim.” give the *average* delay of asynchronous execution steps for actual hardware and ordinary post-layout simulation, respectively. “Counter Period” gives the time it takes the circuit for an entire counting cycle (i.e., 2^n cycles for the counters). Finally, “Execution Jitter” and “Counting Jitter” show the standard deviation of jitter of single execution steps and entire counting cycles (measured on the device), respectively.

3.4.1 Measurement Setup

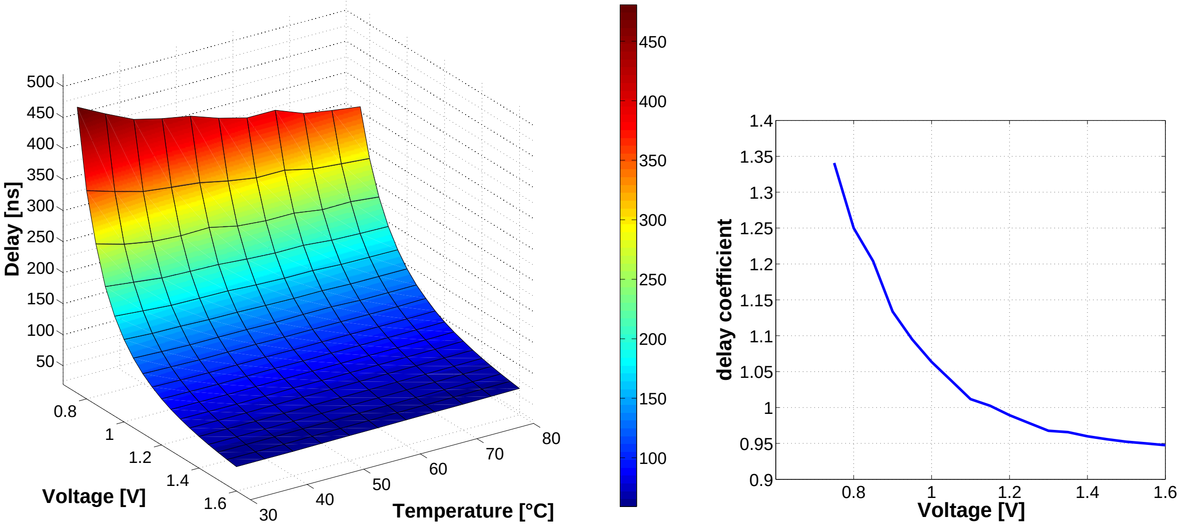
The experiments conducted in this section are threefold: First of all, post-layout simulation is performed on all evaluated circuits. Static timing analysis is used as reference to compare the real-world measurements and the enhanced simulations according to our model against. For static timing simulation, the timing outputs obtained by the EDA tools are used without modifications. As these tools do not include signal jitter or any other indeterministic effects, the results are strictly discrete. In a second simulation run we simulate the circuits according to the model we presented earlier in this chapter. This is done my means of Monte Carlo simulations over the specified process space, so that well-established and powerful EDA tools can be used. Certainly, we are also interested in the real-world behavior of the investigated circuits, thus we also download them to an FPGA board and measure their execution speed and jitter characteristics.

As a target platform we use *Altera Cyclone II EP2C35F484C6* devices ($90nm$ technology, approx. $35k$ logic elements) mounted on an *Hpe mini* evaluation board. We monitor the LEDR circuit's acknowledge signal `cDone` as it is a direct measure of the design's execution speed (it toggles for each execution cycle of the asynchronous system). We also monitor the counter values in order to find correlations between the current state and the respective execution speed. All real-world measurements are performed on the same board with the same power-supply at room temperature. Neither temperature nor supply voltage are subject to (deliberate) changes. The designs have been implemented according to the design-flow described in Section 1.4, and are synthesized with *Altera Quartus II* ($V10.1$) software toolkit. As simulation tool we use *Mentor Graphics Modelsim* ($V6.5$) in combination with the timing output files produced by *Quartus*.

3.4.2 Voltage-Temperature Characteristics

For a more detailed characterization of the relationship between voltage, temperature and delay, a parameter sweep for both voltage and temperature has been performed on one of the available *Hpe mini* evaluation boards. As design under test an asynchronous 64-bit counter has been synthesized and its respective execution speed has been monitored continuously. The evaluation board has been modified such that the core supply voltage could be controlled by an external laboratory power supply, and the entire system was put into a laboratory oven in order to regulate the environmental temperature accordingly. The measurements cover a temperature range starting at $30°C$ up to $80°C$, in steps of $5°C$. For each temperature step, the system was given sufficient time to acquire the environmental temperature. Afterwards, the core supply voltage was modified from $0.75V$ to $1.6V$ in steps of $50mV$, while the ambient temperature was kept at a constant level.

Figure 3.4(a) illustrates the measurement results in a 3D colormap diagram. The well-known dependence of circuit delay on supply voltage is clearly visible in the figure: Core supply voltages considerably lower than the nominal voltage ($1.2V$) lead to severe performance penalties due to this relationship. On the other hand, increasing the voltage results in performance gain, but this benefit is much less pronounced due to the decreasing slope of the curve. This relationship also is in accordance with Equation 3.15. However, the temperature dependent behavior does not meet the expectations over the entire observed range. While for voltages around $1.1V$ or higher the circuit indeed looses performance with rising temperature (due to the high dynamic range of the 3D diagram this is hardly visible in the figure), the effect decreases with falling supply voltage until it almost diminishes at $1.1V$ and even reverses for lower voltages. This means that at low supply voltages the asynchronous counter runs even faster if the temperature is high. Figure 3.4(b) shows how the temperature's influence on the speed changes with voltage: We define parameter $\Gamma(V) = \Delta P(30°C, V)/\Delta P(80°C, V)$ to be the voltage dependent *temperature coefficient* by dividing the circuit delay ΔP at lowest temperature by the delay at highest temperature (for a given supply voltage). One can clearly see the non-linear relationship, and the fact that for low supply voltages, high operating temperatures considerably speed up circuit execution.



(a) (b)

Figure 3.4: Voltage, temperature, delay relationship for Cyclone II FPGA (a) and voltage dependent temperature coefficient Γ (b).

A possible explanation for the unexpected temperature-delay relationship can be found by looking at Equation 3.15: While the operating temperature increases, carrier mobility and threshold voltage V_th decrease. For low V_{DD}, lowering V_th might indeed result in a significant performance gain, because $V_{DD} - V_th$ is relatively small and electrical signals approach V_{DD} asymptotically — when the signal's slope is approaching 0, even a small change of V_th can lead to a major decrease in delay. On the other hand, for high V_{DD} even a major change of V_th does not notably change the overall timing of a transistor because V_th is then in the region where the signal's slope is steepest. It is important to notice that considering just the characteristics of a single transistor might not be sufficient to fully explain the observed effects. After all, the measurements are performed on an evaluation board with plenty of external components. Furthermore, FPGAs themselves are extremely complex devices, thus reducing the findings to a single transistor certainly is inadequate.

3.4.3 Data-Dependent Execution Jitter

3.4.3.1 4-bit Counter

The 4-bit asynchronous counter shall be considered in this section. This circuit has a total of 16 different states (one for each counter value), and the relationship between value and associated LEDR phase is always the same for each value (i.e., all even numbers are always associated with φ_0, and all odd numbers with φ_1, respectively). If we had an odd number of counting states, the association would alternate for any two consecutive counting periods — the number of states to consider would then consequently be twice the

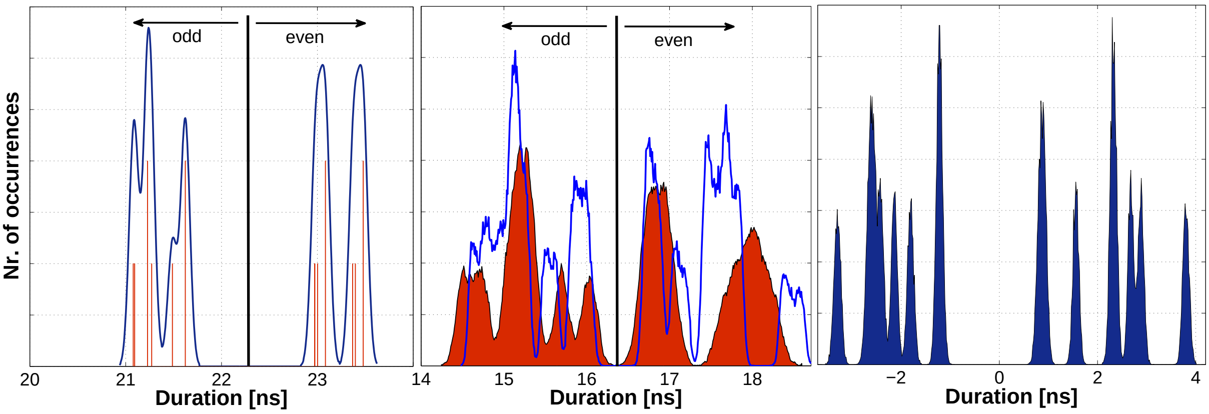


Figure 3.5: Post-layout simulation (left), FPGA measurements and Monte Carlo simulation (middle), and cycle-to-cycle jitter (right) of a 4-bit asynchronous counter's `cDone` signal.

number of counting states. For illustration purposes the original design[7] has been altered by manually distributing the LEDR gates all over the entire available chip area. This way, the interconnects between the gates become considerably larger and the resulting jitter histogram becomes more widespread and shows several distinct peaks instead of just one accumulated hump.

The jitter histogram of signal `cDone` for an ordinary post-layout simulation is shown in Figure 3.5(left). Notice that the simulation results are of course discrete (the red vertical bars), but we added some random jitter by means of convolution in order to get the easier-to-compare dark blue graph[8]. The figure clearly reveals two facts: (i) Each state has a deterministic and discrete execution duration (for some states, this duration is almost the same, thus leading to higher bars in the jitter histogram). (ii) There is a clear separation of the execution speeds of different phases: Odd values (i.e., odd execution phases) are significantly faster compared to even values (i.e., even execution phases).

Measurements on the actual FPGA device have been taken and the results are summarized in Figure 3.5 (middle) in the non-shaded blue graph. Like in the post-layout simulation, there is a significant separation between different execution phases, and there are several distinct peaks caused by the different states of the counter (data-dependent execution jitter). In contrast to before, however, the peaks now have a real jitter component: Instead of being completely discrete, the peaks now naturally "spread" because of accumulated random jitter. Another interesting observation can be made: The results obtained by post-layout simulation are about 30% slower compared to the circuit running on actual hardware, which obviously demonstrates the huge safety margins included in the

---

[7] "Original" means that placement and routing is not altered manually but remains as chosen by the fitting application.

[8] This jitter composition has nothing to do with real jitter, it is just used to obtain a more demonstrative looking shape for the simulation results

timing specification of FPGA devices[9]. Figure 3.5 (middle) also shows the jitter histogram obtained by performing Monte Carlo simulations based upon the model of Section 3.3 in the red-shaded graph. Again, a distinct separation between alternating execution phases is evident. It can be observed that the manually increased interconnect delays between LEDR gates also result in several distinct peaks instead of one accumulated hump in the jitter histogram (compared to measurements performed on the original circuit). Following the model presented earlier this is no surprise: It can be expected that each counting state has an associated characteristic propagation delay. With only 16 states and relatively large interconnects, these delays become even more separated from each other. On the other hand, as we will see for the 16-bit counter, having much more different states results in a superposition of a huge number of almost similar "characteristic" peaks in the jitter histogram.

Finally, Figure 3.5(right) illustrates the cycle-to-cycle (C2C) jitter for the real-world measurements of the middle figure. One can see the usefulness of cycle-to-cycle jitter for asynchronous systems (especially for a simple design like the current one): C2C jitter histograms clearly highlight the significantly different execution durations of adjacent stages. Furthermore, the random component can directly be determined by the width of the single peaks (still, some peaks are superpositions). With the constant component eliminated, the graph provides an appropriate illustration of data-dependent as well as random jitter.

The discrepancies between the shapes of the simulated (shaded) and measured histograms in Figures 3.5 (middle) have several fundamental reasons:

- Some of the empirical data needed for our model are directly extracted from the post-layout timing files. We have already seen that these files contain worst-case information rather that realistic values.

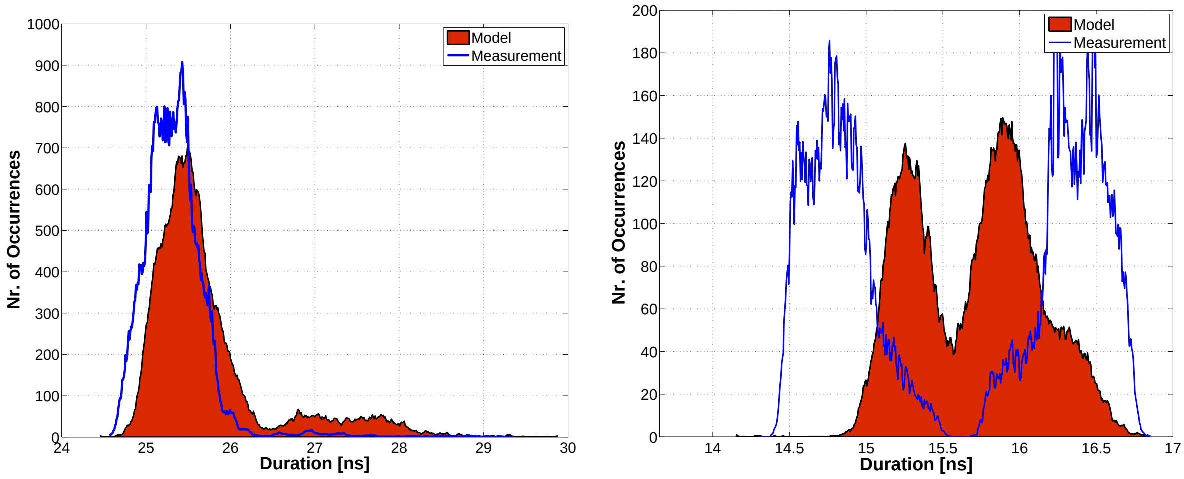
- Our model makes some significant simplifications compared to a fully placed and routed netlist simulation. We model all LEDR gates the same way, thus neglecting timing variations due to different routing and placement. The same is true for interconnect delays, thus routing information does not directly reflect the specific layout of the netlist.

- As we will see in more detail in Section 4.4.5, fabrication variations significantly influence the resulting characteristics of jitter histograms. Even if the Monte Carlo simulation was tweaked to produce outputs similar to the blue graph in Figure 3.5 (middle), the results would not be valid any more if another FPGA device was used.

Besides the obvious differences, there are some important similarities as well. First of all (and most importantly), the *jitter characteristics* of our model closely match the measured ones. Not only are the standard deviations of `cDone` almost the same ($1.18ns$ vs. $1.21ns$ for measurement and Monte Carlo simulation, respectively), also the jitter components

---

[9] This is not only true for the simple 4-bit counter, but for all designs we investigated throughout this work.



(a) (b)

Figure 3.6: Measurement vs. Monte Carlo simulations for non-optimized (a) and optimized (b) 16-bit counter.

of single peaks (associated with a specific counter value) are closely related. Taking into consideration that back annotated timing information from the place and route tool is not available at the considered level of abstraction, the Monte Carlo simulation results are quite useful indeed for estimating jitter effects and data-dependent timing characteristics of a given circuit.

3.4.4 16-bit Counter

In this section a slightly more complex circuit is considered. An asynchronous 16-bit counter is synthesized in two different ways: (i) as unoptimized ripple-carry adder, and (ii) with optimizations turned on. While optimizing the circuit reduces the logic depth (cf. Table 3.1) it at the same time introduces a considerable increase of the LEDR gate count. When comparing the speed of the 4-bit counter to the 16-bit counters, one must take into consideration that the 4-bit counter has been altered manually is thus slower than one would expect due to lower logic depth. Without any modifications the 4-bit counter achieves an execution period of approximately $9ns$, and produces a significantly narrower jitter histogram.

Figures 3.6(a) and 3.6(b) show a comparison of the measured jitter histograms (blue graphs) versus the simulated ones (red-shaded graphs) for both the non-optimized and the optimized circuits, respectively. In contrast to the previous section the circuit now has 2^{16} different states, thus identifying separate peaks for each distinct state is not possible any more. In the non-optimized case all the states superimpose each other and result in only one remaining hump. On the other hand, due to the reduced logic depth and a more balanced logical structure, the histogram in Figure 3.6(b) clearly shows two peaks — one for each associated execution phase $\varphi_{0,1}$.

3.5 Chapter Notes

In this chapter a detailed discussion on asynchronous execution speed and jitter characteristics have been given. After providing an overview to existing jitter definitions and sources, adequate redefinitions and extensions have been introduced to fit the properties of asynchronous circuits. Not only the deterministic and systematic data-dependent execution jitter (DDEJ) plays an important role when characterizing a given circuit's temporal behavior, but also random execution jitter (REJ) considerably influences a signal's timing.

The main focus in this chapter has been to provide an adequate model for describing LEDR circuits in the temporal domain. Starting with a simple hierarchical model based on two gates, major properties of interconnect and gate delays have been identified. Most importantly, it has been shown that even though LEDR designs are strongly indicating, they exhibit considerable data-dependent variations of their propagation delay. In order to obtain a suitable model, supply voltage V, operating temperature T, process variations P, and random jitter J are considered in the presented calculations. Furthermore, experimental results showed that especially gate delays also depend on the actual data value and the implemented logic function (amongst other things because of DDEJ). The combined model is applied to example circuits using Monte Carlo simulations, which has the main advantage of being extremely flexible (with respect to the complexity of modeling), and is based on well-established and well-supported deterministic static timing analysis (thus there is no need for separate tools). All in all our model is able to consider PVT variations and jitter at a relatively high level of abstraction (at LEDR gate level), without the need for back-annotated placement and routing information. It has also been demonstrated (and we will again discuss this topic in Section 4.4.5) that "ordinary" post-layout simulations are only of limited use for asynchronous circuits.

Recalling Section 3.4.2 an interesting finding was made: While digital circuits are usually expected to slow down with increasing temperature, this seems only to be true for high supply voltages. As the core voltage decreases significantly below the nominal value, the impact of temperature on reducing the threshold voltage V_th seems to be dominating and in turn lower the observed execution delays. However, for voltages around and above the nominal voltage, temperature indeed decreases performance as expected.

The last part of this chapter presented some simple case studies in order to check the predictability and usefulness of the proposed model. While our model does not include placement and routing information, obtaining the same results as on a real device is hardly possible. Furthermore, as we will see in the next chapter, jitter characteristics are strongly influenced by process variations — the same design may produce completely different jitter histograms on different devices (of the same type).

Chapter 4

Asynchronous Reference Time

> *How do we know the age of a fossil? How do we know the age of the Earth? How, for that matter, do we know the age of the universe? We need clocks, and clocks are the subject of the next chapter.*
>
> Richard Dawkins

We have already mentioned in the introduction the main problem of project ARTS being the fact of a missing time reference. While in synchronous systems a crystal oscillator usually provides a feasible notion of time, asynchronous circuits do not have any sufficiently accurate reference. The inherent handshaking protocols between asynchronous modules result in considerably varying signal delays: According to the previous chapter, gate and wire delays are not only a function of the (fixed) technology parameters, but also of the actual operating temperature, the circuit's supply voltage, and (submicron) inter- as well as intra-die parameter variations. We have also seen that static timing analysis is only of very limited interest for asynchronous circuits, as it does not comprehend all these uncertainties and variations. Consequently, we need to find another way for establishing absolute time within an asynchronous node, which shall be the central topic of this chapter.

Before we start with actual implementation strategies, a short review on existing clocking techniques for logic circuits is provided (especially focused on self-timed ring oscillators and distributed clocks), followed by an introduction to Allan Variance, which is often used to classify the frequency stability of a given clock. After a detailed discussion of possible circuit implementations, a series of experiments is conducted to test the precision and suitability of the proposed designs under the aspect of changing operating conditions (temperature, supply voltage and fabrication variations) [27, 28, 31, 32]. In Chapter 5 the most promising implementation of all proposed designs is finally integrated into the TTP controller itself, and further adapted to meet the stringent requirements of the time-triggered architecture.

4.1 Related Work

4.1.1 Design Options

As already mentioned, the focus of this chapter will be on how to attain a stable time reference in the context of self-timed logic. From an abstract point of view this problem results from our attempt to insert an asynchronous TTP node into an ensemble of otherwise fully synchronous nodes. Let us first review the most common options for building time references in synchronous systems:

1. *Crystal Oscillators:* This approach exploits mechanical vibrations paired with the piezo-electric effect, which attains highest precision at high frequencies. One of the severe drawbacks of crystal oscillators is their incompatibility with standard process technology. They need to be attached externally, which is area consuming, costly, and unreliable (as soldering contacts may break in harsh environments). Another drawback is the relatively long startup time of crystal oscillators (in the range of one to ten milliseconds). Furthermore, susceptibility to mechanical vibrations, humidity and shock are considerably higher compared to alternative solutions.

2. *RC Oscillators [5]:* Here the time constant associated with charging a capacitance over a resistor is used to define the time reference. While resistors and capacitors are very cheap components and can be integrated on silicon, they suffer from high fabrication variations as well as relatively large temperature and supply voltage dependencies. RC oscillators provide a good alternative to external crystal resonators, as long as high frequency and high precision are not major concerns.

3. *Integrated Silicon (Ring-)Oscillators:* In this approach the oscillations produced by a negative digital feedback loop, usually a ring spanning an odd number of inverters, are exploited [93]. The implementation is fully compatible with the CMOS fabrication process, but the produced frequency is determined by the delay path through the closed loop and hence heavily dependent on fabrication variations, supply voltage, and temperature. Different circuit structures are conceivable, from a simple chain of inverters to more complex solutions, as for example a free running self-timed circuit based on micropipelines [23], which we will discuss later.

4. *Distributed Clock Generation [33]:* For the use in embedded systems, distributed algorithms can be implemented to generate clock signals in a fault-tolerant distributed way. Each node can have its own clock source (i.e., an instance of the distributed algorithm) that remains in synchrony with the others within some known precision bounds. This approach can be viewed as a complex distributed silicon ring oscillator, inheriting the properties of the method above, but being more complex and robust due to the desired fault tolerance.

From the list above, items (3) and (4) have interesting properties with respect to asynchronous circuits: They have in common with asynchronous logic that their timing is solely determined by their propagation delays rather than a (probably external) reference clock. In addition, these solutions can fully be realized in the digital domain, no external or analog modules are necessary. Consequently, it seems adequate to further look into these techniques in the following sections.

4.1.1.1 Distributed Clocks

In contrast to the strictly synchronous design paradigm, which has been presented in Chapter 1.2.1 and has mainly dealt with *phase synchronization*, there are also high-level solutions to the clock synchronization problem. Notice, however, that these methodologies do not solve the on-chip clock distribution problems: The goal is rather to ensure that two distributed clocks do not drift apart indefinitely, but stay within a predefined maximum precision. In this context, a hardware clock (e.g., an oscillator) in combination with a counter register is used to establish a local concept of time. Each positive transition of the clock signal increments the counter by one. Consequently, the counter value can be interpreted as the node's local time (this is called a logical clock). Clock synchronization now means that, at any time, the difference of the counter values of two synchronized nodes is not greater than a predefined precision π [48]. Usually, not the clock-cycles themselves are synchronized, but the logical clocks are adapted by means of rate or state correction to guarantee synchrony of *time*. Basically, it is not possible to achieve clock synchronization by means of high precision oscillators only. After a sufficient amount of time, even small differences in their frequencies would lead to unacceptable variations in the corresponding counter values. As a consequence, correctable clocks such as, e.g., VCXOs (Voltage Controlled Crystal Oscillators) are usually needed to implement proper clock calibration.

There are three major clock synchronization principles, each of which assuming that every node has a separate clock generator which increments a local counter. Furthermore, every node executes the same algorithm and is able to send messages to all other nodes. A very good overview to (fault-tolerant) clock synchronization techniques in distributed systems is provided in [72]:

- *Convergence-based* [72] methods send one message containing the local counter value to all other nodes in each round of synchronization. After receiving enough messages, the convergence algorithm (e.g., averaging) calculates the new time value and changes the local counter accordingly. The algorithm assumes that all local clocks are synchronized at start-up, that each message can uniquely be assigned to its sender and that the message delay is bounded. The upper bound of the message delay has central influence on the algorithm's precision. This technique is used in TTP to establish a global notion of time among all non-faulty, distributed nodes.

- The *consistency-based* [72] principle works similar to the one above. All nodes broadcast a message, but this time each processor additionally forwards the received counter values (which increases network traffic). Consequently, all non-faulty processors have a consistent snapshot of the system. The new time is calculated using the median of the received messages. Again, the algorithm assumes that the message delay is bounded and that messages can be associated with their senders.

- In general, *probabilistic* [4, 72] algorithms receive and store the data of other nodes and as soon as enough data is collected, statistical analysis is performed to derive a new time. It is not necessary to uniquely identify a message's origin. The drawbacks of this principle are that collecting data can take very long, synchronization is only reached with a probability of less than 1 and hardware implementations are — because of the statistical tests — infeasible.

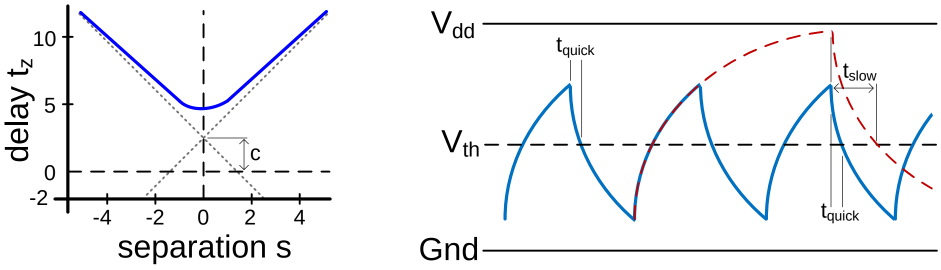
A different approach is to actually generate a distributed clock signal, rather than to synchronize independent hardware clocks to each other. The authors in [36] present a simple fault-tolerant tick generation algorithm (based on Srikanth and Toueg's consistent broadcast primitive), prove its correctness, and also propose an appropriate asynchronous VLSI implementation. Rather than performing state or rate correction on logical clocks locally at each node, the algorithm directly generates distributed ticks which are guaranteed to be synchronous within some precision π. The main disadvantage of this solution is the fact that a fully-connected 1-bit network is required for the algorithm to work.

4.1.1.2 Self-Timed Oscillator Rings

While TTP uses a convergence-average-based algorithm for global time synchronization on a high level, distributed clock synchronization algorithms (on a hardware level) seem to be overly complex for our purposes. Moreover, distributed solutions have the disadvantage that we would also need to modify the synchronous reference nodes to participate in the synchronization procedure – this clearly contradicts the goal to integrate a single asynchronous node in an otherwise untouched synchronous TTP cluster.

We therefore want to take a closer look at self-timed oscillator rings in this section, as they seem to offer very promising solutions to our problems and form an important alternative for generating precise time references. Compared to simple inverter rings, which consist of a chain of cascaded inverters only, self-timed rings are based on Sutherland's micropipelines [82]. A lot of research has been conducted on self-timed oscillator rings. For example, [24] proposes a methodology for using self-timed circuitry for global clocking. The same authors also use basic asynchronous FIFO stages to generate multiple phase-shifted clock signals for high precision timing in [23]. Furthermore, it has been found that event spacing in self-timed oscillator rings can be controlled [93, 96]. The Charlie- and the drafting-effects have thereby been identified as major forces controlling event spacing in self-timed rings [21, 23]:

- Given a two-input gate (with inputs a, b and output z, and respective transition times t_a, t_b and t_z), such as an ordinary AND gate or a more complex Muller-C Element, the *Charlie Effect* describes the gate's delay from the average input arrival time $m = (t_a + t_b)/2$ to the output transition $t_z = m + Charlie(s) + c$ as a function of the input separation time $s = (t_a - t_b)/2$ and some constant gate delay



(a)

(b)

Figure 4.1: Charlie Diagram (Source: [93]) (a) and Drafting Effect (Source: [85]) (b).

c. As it turns out, the delay through a gate increases when the input transitions are close to each other. Explanations can be obtained by looking at the gate's respective transistor level circuit: For largely separated inputs, the respective transistors are already saturated when the other transition occurs. Consequently, the output is reached quicker. In contrast, for (almost) simultaneous transitions, transistors in series need to switch at the same time, thereby increasing the overall delay. A typical *Charlie Diagram* [93] is shown in Figure 4.1(a): We can see that around $s = 0$ the overall delay is greater than just the gate's static delay. For large s, on the other hand, the delay approaches the asymptotes ($t_z \approx m + c$).

- The *Drafting Effect*, on the other hand, targets the separation of output events. Load capacitances force electrical signals to approach their V_{dd} or GND asymptotically rather than instantaneously. Considering Figure 4.1(b), in the case of fast output transitions the final voltage level has not been reached completely, allowing the next transition to cross the threshold V_{th} earlier (t_{quick}). On the other hand, for the dashed red signal $t_{slow} > t_{quick}$ because the lower transition frequency allows the signal to almost reach V_{dd} — the subsequent falling transition consequently needs more time to cross V_{th} again. Notice that this is one of the reasons for *data dependent jitter* as introduced in Section 3.1.1.

Self-timed rings have the very same structure as shown in Figure 1.9(a), however, the first and last micropipeline stages are connected to each other in order to obtain a self-oscillating closed loop. Fairbanks and Moore even proposed a special C-element consisting of six inverters only for the use in these ring structures [23]. Depending on the concrete initialization of a micropipeline ring, one can adjust the number of "tokens" and "bubbles" actively shifting through the ring, thereby regulating the resulting oscillation frequency. Using the outputs of different pipeline stages provides several clock signals, all of which have the same frequency and a *fixed* phase-relationship to each other.

Although we are not using self-timed oscillator rings directly as described above in our implementation, it is worth noticing that both the NCL as well as the LEDR design style are based (when seen from an abstract point of view) on the very same control structures — as long as the system is free running, with no blocking inputs from the environment. To summarize, self-timed oscillator rings have the advantage of being fully compatible with the digital domain, are almost directly applicable to the LEDR asynchronous design style, and are — just as LEDR circuits themselves — subject to timing deviations caused by PVT variations. Especially the latter is of major importance, because the investigation of timing variations due to PVT fluctuations is the central topic of this work.

4.1.2 Allan Variance

So far in this chapter we investigated various methodologies to generate clock signals. Now the immediate question arises, how to quantify the accuracy and stability of these time references. One can expect that self-timed rings or distributed synchronization algorithms do not achieve the same precision and frequency stability compared to ordinary crystal oscillators. A well established statistical technique for characterizing the *frequency stability* of clocks (crystal oscillators or atomic clocks, e.g.) is called *Allan variance* or *two-sample variance* [1, 2, 44].

Due to the fact that the observed random noise of clock signals consists not only of white amplitude noise, but also of white frequency noise as well as flicker frequency noise [2], traditional statistical measures such as the standard deviation do not converge and are thus only of limited use for the examination of frequency stability (it turns out that for special types of noise they become a function of data length rather than converging to a specific value [1]). The Allan variance, named after David Allan, overcomes these issues. It is the special case of the M-sample variance with $M = 2$ and defined as follows:

$$\sigma_y^2(\tau) = \frac{1}{2}\langle(\Delta y)^2\rangle \tag{4.1}$$

$$= \frac{1}{2}\langle(y_{n+1} - y_n)^2\rangle \tag{4.2}$$

In Equation 4.2, the angle brackets indicate the statistical expectation value over the entire observation window n. Furthermore, τ is the duration of the observation window (also called averaging window), and y_n describes the normalized frequency-deviation, which can also be written in two alternative forms:

$$y_n = \langle\frac{\delta f}{f}\rangle = \frac{1}{\tau}(x_{n+1} - x_n) \tag{4.3}$$

$$x_n = x_0 + \tau \sum_{i=0}^{n-1} y_i \tag{4.4}$$

In Equation 4.3, δf is the frequency deviation and f the nominal frequency, again averaged over an entire averaging window of duration τ. An alternative way of defining y_n is to use *time deviation* x_n instead of frequency deviation, which is shown in Equation 4.4. The time deviation x_n of period n is simply the sum of all preceding frequency deviations $y_i, 0 \leq i < n$. This last expression is especially useful for our purposes as oscilloscopes (or logic analyzers) usually return period- instead of frequency-measurements (even though the transformation would be trivial). Alternatively, we can now write the Allan variance depending on the time deviations x_n rather than frequency deviations y_n, as shown in Equation 4.5:

$$\sigma_y^2(\tau) = \frac{1}{2\tau^2}\langle (x_{n+2} - 2x_{n+1} + x_n)^2 \rangle \tag{4.5}$$

In addition to the already mentioned benefits, Allan variance also offers another important feature. Instead of a single number, Allan deviation is usually displayed as (log-log) graph for gradually increasing durations τ of the averaging window. It therefore combines measures for both short (e.g., execution steps) and long (e.g., generated ticks for bit-timing) term stability in a single plot. Later in this chapter we will see some Allan variance plots and discuss them in detail.

4.1.3 PVT Variations

To start with, a very comprehensive overview to the topic of PVT variations in general and process variations in particular is presented in [88] and [9]. The former presents a classification of parameter variations (process related or environmental), and then further details sources and possible compensation techniques for process, voltage and temperature variations. More formal approaches to model PVT variations are presented, e.g, in [10] and [73]. The latter paper proposes an intuitive model for process variations and timing errors caused by parameter variations. On the other hand, the former work investigates the impact of die-to-die as well as intra-die variations on the maximum operating frequency. In the context of statistical static-timing analysis, [7] focuses on process variations and their related physical origins. Most notably is the authors' classification of physical parameter variations in *systematic* (detailed analysis allows deterministic and premanufacturing modeling of layout-dependent variations) and *random* (only statistical parameters are known at design time, must be modeled as random variables) sources. Especially for process variations, the following distinction can be made [7]:

- *Die-to-die variations*, which are sometimes also called inter-die or global variations, affect certain parameters of a die in the same way. For example, for a specific die all the gate widths may be slightly higher than nominal, whereas for another die all widths are slightly lower. As source for these variations [7] states that *"die-to-die variations are the result of shifts in the process that occur from lot to lot, wafer to wafer, reticle to reticle, and across a reticle if the reticle contains more than one copy of a chip layout"*.

- *Within-die variations* are also sometimes called local or intra-die variations, and have different influence on gates even within a single die. As a consequence, nominally equal gates have slightly different actual parameters after manufacturing. According to [7] *"within-die variations are only caused by across-reticle variations within the confines of a single chip layout"*.

In another interesting work, Krishnamurthy et al. propose a method for dynamic calibration of critical delay paths under temperature variations [50]. By reconfiguring critical paths as ring oscillators the maximum operating frequency can be computed (for different temperatures). This technique is robust against process variations, and achieves precise estimations at little design overhead. While this last approach actively adapts the speed of operation to the current physical conditions, we try to use the opposite technique: We just let the variations happen, which results in a different operating speed of our asynchronous circuit. At a higher level of abstraction we then try to derive the magnitude of these fluctuations and to compensate them accordingly in order to obtain a stable time reference.

4.2 Implementation Concept

For both bit synchronization as well as for the alignment of the sending slots to the global schedule, the resulting asynchronous controller must have a precise notion of time. As there is no accurate reference time available in the case of asynchronous logic, we design a circuit that uses the TTP communication stream to derive a suitable and sufficiently stable timebase. We construct an adjustable tick-generator and periodically synchronize it to *incoming* message bits. In our specific configuration, the bitstream of TTP uses Manchester coding [34] (thus there is at least one signal transition for each bit), which we can potentially use for calibration. The Manchester encoding is a line code which represents the logical values 1 and 0 as falling and rising transitions, respectively. Consequently, each bit is transmitted using two successive symbols, and the necessary communication bandwidth is double the actual bitrate. This encoding scheme has the advantage of being self-clocking, which means that the clock signal can be recovered directly from the bitstream (encoding is done by XORing the data bits with the virtual Manchester-clock). From an electrical point of view, Manchester codes provide a DC-free (direct current) physical interface.

We define for the rest of this work that the idle state of the bus is represented by a high voltage level. Furthermore, falling edges (at 50% of the bit time) shall represent logical 1s being transmitted, and a rising edges shall be interpreted as logical 0s, respectively. These properties also match the settings of the used TTP cluster. The illustration in Figure 4.2 shows an exemplary frame as it is encoded by existing TTP controllers: The first bit (SOF) always is a logical 1, followed by a bunch of data bits ($1-0-0-1$ in this example). Each frame ends with a special sequence of three 0 half-bits followed by three 1 half-bits (EOF or *postamble* in TTP notation). Notice that this is specific to TTP and has nothing to do with Manchester coding. We will learn about the exact fields of TTP messages in the next chapter, for now the semantics of single bits are not important.

Before we start to compile the requirements of the time reference generation circuit there is one important issue to discuss. As mentioned before, the circuit will be implemented using the (asynchronous) LEDR design style. LEDR is, however, considered delay-insensitive, which clearly contradicts the goal of designing a circuit which produces a stable reference time. In the context of delay-insensitivity it is always possible that a signal is delayed arbitrarily, consequently rendering any measures of (worst-case) propa-

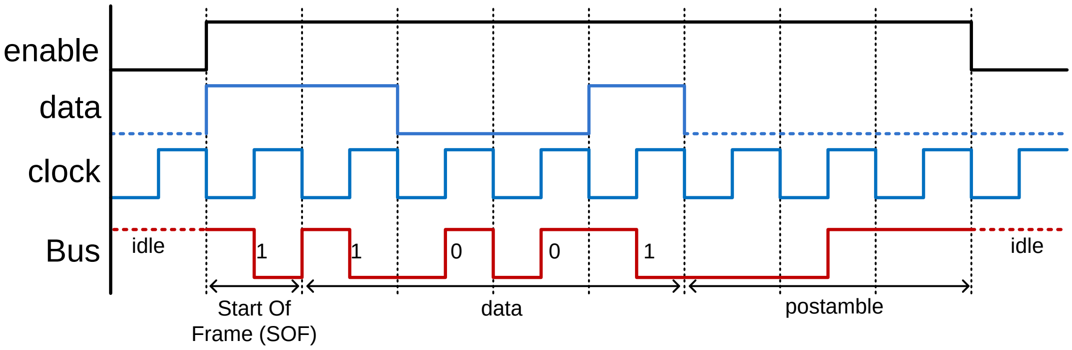


Figure 4.2: TTP-specific Manchester encoding of frames.

gation delay useless. For the course of the project it is therefore necessary to apply the bounded delay (or self-timed) timing model rather than strict delay-insensitivity. From a practical point of view this model is more realistic anyway, because real hardware has (assuming constant operating conditions) upper and lower bounds for the respective propagation delays. From a theoretical point of view, any delay-insensitive circuit also works under the bounded delay assumption by definition.

4.2.1 Requirements

In order to get a profound understanding of the requirements for the time reference generation circuit let us first consider the red part at the top of Figure 4.3(a), where a sequence of three Manchester coded bits is shown. As already mentioned, Manchester coding uses two symbols to transmit a single bit, thus the "feature-size" τ_{ref} of the communication stream is half the actual bit-time τ_{bit}. In order to correctly receive messages, the sampling points need to be located at 25% and 75% of τ_{bit}, respectively. We intend to achieve this quarter-bit-alignment by doubling the generated tick-frequency ($\tau_{gen} = \frac{\tau_{ref}}{2}$). Although there are alternative ways to accomplish this, we could not identify significant advantages of other solutions over the chosen one. The intended reference time signal `ref-time` is marked in blue color at the bottom of Figure 4.3(a). Obviously, each rising edge of signal *ref-time* defines an optimal sampling point. As our circuit is implemented asynchronously, the generated reference signal can be expected to jitter considerably. Furthermore, temperature and voltage fluctuations will also change the reference's signal period τ_{gen}. It is therefore necessary to make the circuit self-adaptive to changing operating conditions.

The basic idea for the envisioned time reference generation circuit is to use a free running, asynchronous counter and "measure" the duration τ_{ref} in terms of the attained counter value cnt_{ref}. We can then approximate the original duration τ_{ref} by continuously counting to the reference value cnt_{ref}. We will discuss advantages, disadvantages as well as fallacies and pitfalls in detail in this chapter. Let us now review the list of basic requirements that a potential solution must fulfill:

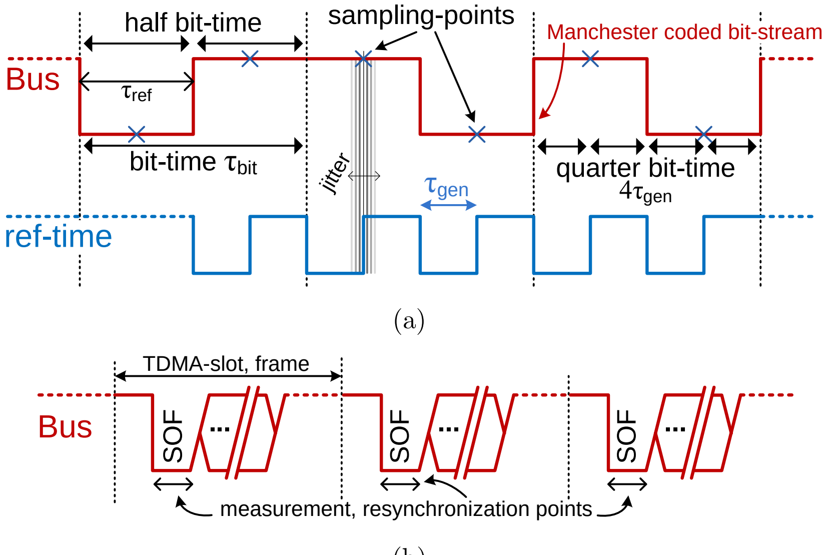
1. As each logical bit on the bus is represented by either a falling or a rising edge at 50% of the bit time, it is necessary to sample (at least) twice per τ_{bit}, ideally at 25% and 75% of the bit time.
2. While the nominal bit length at a specific transmission rate is τ_{bit}, the Manchester coding exhibits a "feature size" of only $0.5\tau_{bit}$ (see Figure 4.3(a)). The quarter-bit-alignment necessary for the optimum sampling points might further half the resulting period τ_{gen}. For high bit-rates it is therefore important that the underlying reference circuitry is fast enough in order to achieve an acceptable precision.
3. The time reference needs to remain synchronized with the other local references in the system. Recall that this requires periodic re-synchronization even in the synchronous case. It is therefore mandatory to have a reference whose timing can precisely be adjusted.
4. In the synchronous case the resolution of the adjustment is determined by the local clock generator. With a typical clock frequency of 40MHz we have a resolution R of some 25ns. It seems reasonable to strive for a similar resolution in our case.
5. Re-synchronization in TTP is usually performed once every TDMA round in the synchronous case. As we expect the asynchronous reference to be considerably less stable than a crystal clock, we have to perform re-synchronization more often (e.g., for each message or even each bit).
6. For the purpose of our study we want to consider all provisions to compensate for the non-ideal behavior of our reference. Among these are the elimination of long-term effects by virtue of periodic re-calibration, masking of random effects by means of averaging, and avoidance of systematic effects by means of design measures. These points are discussed in more detail in Section 4.2.2.

Requirements (2), (3) and (4) spoil our hope to use the operation cycles of the asynchronous controller as a time reference – due to the complexity, these will neither be fast enough nor adjustable. Therefore, we decided to use a separate, small circuit as a time reference generator that is not dependent on the remaining controller's control flow. A counter suggests itself here to count up to a threshold cnt_{ref}, whose adjustment already implements the rate correction desired in (5). The price for this decoupling is the need for an explicit synchronization of the operation cycles of the remaining controller logic to this reference, which can be achieved by one of the methods already presented in Chapter 2.

We will exploit the deterministic nature of TTP and use features of known length in the periodic data stream provided by the other (synchronous) communication participants as a reference to periodically adjust our local timing[1]. According to (3) we have to adjust the reference as often as possible. We can take advantage of the start of frame (SOF) sequence (HI followed by LO with a length of $0.5\tau_{bit}$ each) for our measurement. More

---

[1] We are well aware that this may become a circular argument in case of all nodes in the system being implemented asynchronously. This is, however, not our intention in this work.



(a)

(b)

Figure 4.3: Manchester code with sampling points (a), TTP-slots, resynchronization (b).

specifically we use the first LO as our "reference half-bit" (see Figure 4.3(b)). Measuring just a half-bit cell instead of a much longer interval clearly increases the quantization error. However, longer intervals tend to become dependent on the system configuration (number of involved nodes, configured message length, etc.), thereby considerably complicating the measurement circuitry because more control logic is necessary. This increased complexity not only downgrades performance (which in turn increases the quantization error), but also introduces more jitter and makes timing analysis/predictions substantially harder.

To summarize shortly, the proposed procedure comprises two phases: (i) A *measurement phase* m during which the reference counter's threshold cnt_{ref} is determined by starting at 0 at the beginning of the reference half-bit and simply stopping the counter at the observed half-bit's end. (ii) A *reproduction phase* r during which the observed half-bit length is periodically reproduced by having the counter wrap around to 0 as soon as it reaches the threshold determined above (with a proper initial alignment of $0.25\tau_{bit}$ according to (1)). Implementation details, especially design alternatives and optimizations, will be discussed in Section 4.3. For now, however, we restrict ourselves to the basic concept and analyze it in more detail.

4.2.2 Temporal Properties

The above procedure implies *absolute rate correction* of the local time, as the internal time is corrected upon completion of the start of frame field. In addition, *continuous rate correction* can be achieved by adjusting the threshold value cnt_{ref} for every bit. The latter allows for a very tight matching between the current sender's actual bit length and the period of our reference counter (which is subject to variations caused by changing operating conditions). True *random* jitter effects are automatically averaged by counting to the (same) measured threshold value over and over again. The temporal proximity of measurement and associated reproduction phases is beneficial, as it facilitates an effective compensation of long term variations (long with respect to the frame length). In other words, the disturbing impact of environmental fluctuations is automatically compensated over time, because the periodic re-synchronization events will lead to different cnt_{ref} values depending on the actual speed of the free running asynchronous counter circuit.

The obvious questions that arise are, which properties does our solution have with respect to frequency stability, and how can changing environmental conditions be dealt with. The following list summarizes all effects that need to be taken into account and discusses their impact on our design. To this end, we need to define some parameters for a simple quantification. τ_{bit} has already been introduced as the duration of one Manchester coded bit on the bus. We further define $\tau_{step,m}$ and $\tau_{step,r}$ to be the average durations of single execution cycles in *measurement phase* and *reproduction phase*, respectively. τ_{step} is used if the difference between these two phases is not important. Finally, τ_{ref} denotes the duration of the reference signal to be measured. Consequently, $cnt_{ref} = \lfloor \frac{\tau_{ref}}{\tau_{step,m}} \rfloor$ is the average number of execution steps (i.e., the counter threshold) for the measured pulse of length τ_{ref}.

1. The *quantization error* clearly depends on the type of synchronizer used. For instance, the version presented in Section 2.3.1 provides new data in phase 1 only, resulting in $|err_{quant}| \leq 2\tau_{step,m}$. On the other hand, the more generic synchronizer of Section 2.3 (which we actually use) is able to sample data with each change of phase, thereby halving $|err_{quant}| \leq \tau_{step,m}$ compared to before. For the measured counter value this means $cnt_{ref} = cnt_{opt} \pm 1$, with cnt_{opt} being the optimal or nominal counter value for the current execution speed $\tau_{step,m}$. By keeping $\tau_{step,m}$ as low as possible, err_{quant} can be improved accordingly. In addition, with increasing cnt_{ref}, the relative error introduced by a single tick decreases.

2. As a direct consequence from the timing model presented in Chapter 3 we identify *systematic errors* to be introduced by data dependent jitter. Two major cases need to be distinguished for our design:

   (a) The single execution steps while counting up to the measured threshold value show considerable DDJ with respect to each other. A detailed discussion of this effect has already been presented earlier (recall the case study with a 4-bit counter of Section 3.4.3.1, e.g.). Considering entire counting cycles, however, the timing variations are exactly the same for each run (assuming matching cnt_{ref}), thus the “error” is compensated automatically.

   (b) As *measurement* and *reproduction phase* are different states with slightly different register/input values, their average execution speeds typically do not match exactly, i.e., $\tau_{step,m} \neq \tau_{step,r}$ (because different signal paths are enabled, recall the previous chapter). The relative deviation of the generated time reference from its measured value can be expressed as factor $f_{dev} = \frac{\tau_{step,r}}{\tau_{step,m}}$ and should optimally be $f_{dev} = 1$ (which in practice, it is not). These timing errors can at least *partially* be compensated by clever circuit design and complex correction measures at logic level. In contrast to above, the *absolute* error in case of $f \neq 1$ can be kept low only by keeping cnt_{ref} low.

3. *Systematic long term effects* are mainly caused by slow changes in temperature or supply voltage. Given these fluctuations are slow enough (compared to one TDMA slot), they are compensated automatically at each resynchronization point (cf. Figure 4.3(b)), because depending on the current speed of the counter circuit there will be a different reference value cnt_{ref}. Systematic in this case means "non-random", e.g., when the circuit is heating up after startup.

4. *Random effects* cannot be compensated at all. However, when averaging over long periods, statistical outliers become less important and frequency stability improves (this is also evident in the Allan plots from the experiments in Section 4.4). For our design, averaging occurs automatically due to the periodic counting-cycles. Notice that not only the *number* of counting cycles is important, but also the associated circuit *state* for each cycle — otherwise, DDEJ is not compensated adequately. Consequently, as for quantization errors, large values of cnt_{ref} are desirable (in contrast to systematic errors, where lower threshold values are preferable in case $f_{dev} \neq 1$ in order to keep the *absolute* error low).

5. *Short term effects* (either random or systematic) that occur faster than a TDMA slot can only be compensated on the bit-transmission level while actually *receiving* messages. This is, however, insufficient for active message transmission, because TTP requires frames to hit a very narrow *action window*, which can only be calculated and adjusted once for every *receiving* TDMA slot.

It is important to realize the severity of the systematic error f_{dev} with respect to the achievable accuracy. During the measurement phase, we obtain a suitable reference counter value cnt_{ref} by resolving the following expression (neglecting random jitter):

$$\sum_{i=1}^{cnt_{ref}} \tau_{step,m}(i) = \tau_{ref} - err_{quant} \tag{4.6}$$

Following the theoretical model of the previous section, $\tau_{step,m}(i)$ is not constant but depends on the current state, in this case the current counter value i. However, we have already seen that this state-dependency is always the same for matching i, thus we can replace $\tau_{step,m}(i)$ with the *average* duration $\tau_{step,m}$ of a single step. This results in the following equation for cnt_{ref}:

$$cnt_{ref} = \lfloor \frac{\tau_{ref}}{\tau_{step,m}} \rfloor + \lceil \frac{err_{quant}}{\tau_{step,m}} \rceil \tag{4.7}$$

The reproduction phase now performs basically the same task as described by Equation 4.6, but in the opposite direction. The goal is to achieve a reproduced period τ_{rep}

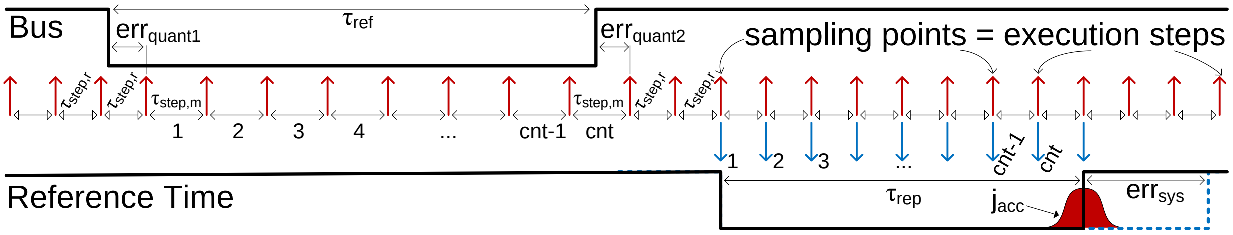


Figure 4.4: Illustration of all characteristic temporal system properties.

that approximates τ_{ref} as good as possible[2]. The optimum case is indicated in Equation 4.8, and uses the original executions speed $\tau_{step,m}$ of the measurement phase as well as the quantization error err_{quant}. In reality, however, the situation looks like Equation 4.9: During reproduction, the execution speed is determined by another circuit state and must be replaced with $\tau_{step,r}$. Furthermore, the quantization error is unknown by the system, and cannot be compensated for.

$$\tau_{ref} = \tau_{rep} = \sum_{i=1}^{cnt_{ref}} \tau_{step,m}(i) + err_{quant} \approx cnt_{ref}\tau_{step,m} + err_{quant} \tag{4.8}$$

$$\tau_{ref} \approx \tau_{rep} = \sum_{i=1}^{cnt_{ref}} \tau_{step,r}(i) \approx cnt_{ref}\tau_{step,r} = cnt_{ref}\tau_{step,m}f_{dev} \tag{4.9}$$

Ignoring the quantization error (e.g., for large cnt_{ref}), consider the case where $f_{dev} = 1.01$, which means that the average execution cycle in the reproduction phase is 1% longer than in the measurement phase. While this may seem to be acceptable at a first glance, a typical (short) TTP message consists of approximately 200 bits. An error of 1% means that we are off by two entire bit-times for a single frame, which is a significant deviation that might not be tolerable.

The resulting τ_{rep} may substantially differ from the optimum τ_{ref}. The overall systematic error err_{sys} can be written as shown below. While f_{dev} is almost constant for a given circuit (under specific operating conditions), err_{quant} can be different for each measurement, thus err_{sys} is variable as well.

$$err_{sys} = err_{quant} + (1 - f_{dev})cnt_{ref}\tau_{step,m} = err_{quant} + cnt_{ref}(\tau_{step,m} - \tau_{step,r}) \tag{4.10}$$

Finally, long term accumulated jitter (by definition indeterministic and unbounded) causes additional inaccuracies during the reproduction phase. Assuming a normal distribution for random jitter induced in every execution step, the accumulated jitter can be approximated as $j_{acc} = N(0, \sigma_{acc}^2)$ with $\sigma_{acc}^2 = cnt_{ref}\sigma_{step}^2$ (where σ_{step}^2 is the variance of a single execution step).

All the introduced characteristic figures are summarized in Figure 4.4. The top signal labeled `Bus` represents the bus-line, where only one low-pulse with duration τ_{ref} is shown.

---

[2] We actually want to achieve $\tau_{gen} = \tau_{ref}/2$ by halving cnt_{ref}. For now, we neglect frequency doubling and use just with τ_{ref} and $\tau_{rep} = 2\tau_{gen}$ as direct result of the reproduction phase.

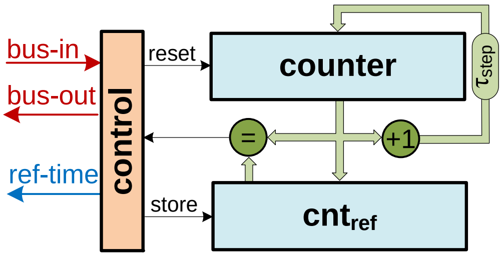


Figure 4.5: Basic structure of the time-reference generation circuit.

On the other hand, the bottom signal with label `Reference Time` symbolizes the output generated by our time reference generator during reproduction phase, also with just one pulse shown. The red (and blue) arrows in between indicate the sampling points, which are equivalent to the execution steps of the asynchronous circuit (we sample the bus in each cycle). Notice the extremely high value of $f_{dev} = 0.75$ which was chosen for illustration purposes. The resulting pulse of duration $\tau_{rep} + j_{acc}$ is, in this example, considerably shorter than the original pulse because of the severe deviation of $\tau_{step,r}$ and $\tau_{step,m}$. As can be seen in the figure, err_{quant1} reduces the length of the measured pulse-width, while err_{quant2} prolongs it. Therefore, $err_{quant} = err_{quant2} - err_{quant1}$ is (according to Equation 4.10) part of err_{sys}. Notice that we use τ_{rep} instead of τ_{gen} in Figure 4.4. The reason is that the period we actually want to reproduce is half that of $\tau_{rep} = 2\tau_{gen}$.

4.3 Implementation Details

According to the requirements of the previous section, we want to design a circuit which uses the TTP-communication stream to derive a suitable, stable time-base (the known baudrate of the communication bus is the only external “time reference” available). Our idea is to construct an adjustable tick-generator and periodically synchronize it to *incoming* message-bits. We will now present a feasible strategy for implementing an adaptive time-reference generation circuit. We start with a straight-forward implementation and will include optimizations and enhancements afterwards. This way, we are able to classify the effectiveness of the incorporated optimizations and cross-check them with the predictions made by the corresponding theoretical models.

Let us now take a closer look at the proposed designs of the time reference generator circuit, the basic structure of which is shown in Figure 4.5. As one can see, the interface of the design is quite simple:

- `bus-in`: This signal simply is the single-bit receive-line of the TTP-bus, directly coming from the bus transceiver.

- `ref-time`: This signal is the asynchronously generated time reference with known period τ_{rep}.

- `bus-out` is the "synchronized" signal `bus-in`. We need this signal for the higher levels (which actually interpret the received bitstream). To avoid race conditions between the reference time generator and the higher level, `bus-out` is sampled by the low-level modules and passed on to the remaining system with a short (but constant) delay.

The control block continuously monitors `bus-in`. If it detects the Start Of Frame sequence (i.e., a falling edge on the bus after a long period of idle-time, cf. Figure 4.3(a)), it resets the free running counter-unit to zero. This asynchronous counter periodically increments its own value at a certain rate $\tau_{step}(counter)$ depending on the current value $counter$ and is on average $\tau_{step,m}$ or $\tau_{step,r}$ in measurement or reproduction phase, respectively. The exact value of τ_{step} mainly depends on the circuit structure, placement and routing, the circuit's state, and on environmental conditions such as supply voltage and operating temperature. After time τ_{ref}, the corresponding rising edge of the SOF sequence will eventually be detected by the control-block. As a consequence, the current counter value is preserved in a separate register cnt_{ref} and `counter` is restarted. The controller is now able to reproduce the measured low-period τ_{ref} by periodically counting from zero to $cnt_{ref} - 1$, and generating a signal transition on line `ref-time` on every compare match. However, so far the generated reference signal does not provide the required 25%/75% alignment for the optimal sampling points. In order to achieve this we double the output frequency by simply dividing cnt_{ref} by two (i.e., shifting, and possibly loosing one LSB precision for odd cnt_{ref}).

It is obvious from this description that we have exactly one resynchronization point for each TTP-message (the SOF sequence). Consequently, slow changes in the system's execution speed are automatically compensated: If, for example, increasing supply voltage levels speed up the circuit, a higher cnt_{ref} counteracts the expected increase of the output frequency. In order to study the properties and characteristics of the structure presented in Figure 4.5, we realize the following five implementation alternatives. We choose a uniform width of 16 bits for the respective counter registers for all measurements.

- *CntRef*: Our reference implementation is a one-to-one mapping of Figure 4.5, with all optimizations turned off. Consequently, the incrementer circuit will be synthesized as simple ripple-carry adder.

- *CntManual*: We apply manual optimizations to reduce the propagation delay. The most important one being that the 16-bit counter unit is replaced by four pipelined 4-bit incrementers. Also the comparator is pipelined to reduce maximum logic depth and thus propagation delay.

- *LFSRRef*: This alternative is based on *CntRef*, the only difference being that the counter-unit is replaced by a much simpler Linear Feedback Shift Register (LFSR). The rationale is to positively change execution speed and area consumption, but LFSRs have the severe drawback that halving cnt_{ref} to achieve quarter-bit alignment is not possible. Instead we integrate a second LFSR circuit, which counts at the

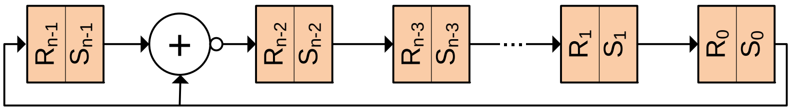

R_{n-1} S_{n-1}
+
R_{n-2} S_{n-2}
R_{n-3} S_{n-3}
R₁ S₁
R₀ S₀

Figure 4.6: LFSR counting at half speed due to shadow registers S_i.

half speed of the original one. A more detailed discussion of this issue is presented below.

- *CntRate*: Based on *CntManual*, we now also apply continuous rate correction for each bit of a message to further increase precision. The SOF signature is again measured absolutely, however, cnt_{ref} is additionally incremented or decremented by one for each bit, depending on whether the actual bus-transitions occur later or earlier than expected. This issue is also discussed in more detail below.

- *LFSRRate*: This alternative is based on *CntRate*, but again with all increment and decrement circuits replaced by simple LFSRs. As LFSR-values cannot easily be divided by two, we now just perform bitwise rate correction, so the SOF sequence of duration τ_{ref} is not measured explicitly any more (this simplifies the control logic, absolute rate correction is disabled). However, without an absolute measurement of τ_{ref}, it is necessary for this solution to work properly that the initial cnt_{ref} corresponds to a time period $\tau_{gen} \approx \frac{\tau_{ref}}{4}$ in the range of $\frac{\tau_{ref}}{5} < \tau_{gen} < \frac{\tau_{ref}}{3}$ (refer to Equations 5.1 and 5.2 on page 117). This initial value can either be obtained by measurement, post-layout simulation, or a rough estimation. Our synchronization strategy allows the initial value of τ_{gen} to be off its optimum according to the above relation. Once these limits are exceeded, the circuit will *not* be able to synchronize itself correctly to the incoming bit-stream.

The above list introduced several new cases that we now discuss in more detail. The first significant optimization we apply to the basic circuit is to use Linear Feedback Shift Registers (LFSR) instead of full adders. LFSR do not count in a regular “ordered” fashion, but rather generate pseudo random numbers while executing. However, the generated sequence of numbers is deterministic. In our case we do not need a strict ordering of the counting events, it is sufficient if the sequence of states is unique and reproducible. There exist different implementation alternatives, but we use so called Galois LFSRs [41] as they have a logic depth of only one gate equivalent. Using LFSRs one can achieve maximum counting periods for almost arbitrary bit widths with just one XNOR gate inside the shift registers (for some widths, three gates need to be deployed). By reversing the shift direction and changing the positions of the feedback tabs, LFSRs are also able to “count backwards”, i.e., reverse the sequence of states. Compared to ordinary counter implementations, LFSRs are extremely efficient in terms of area consumption and performance, at the expense of no structured ordering of states.

This latter property is a major drawback, because we need to divide the measured counter value cnt_{ref} by two, which is a simple shift operation in the case of an ordinary

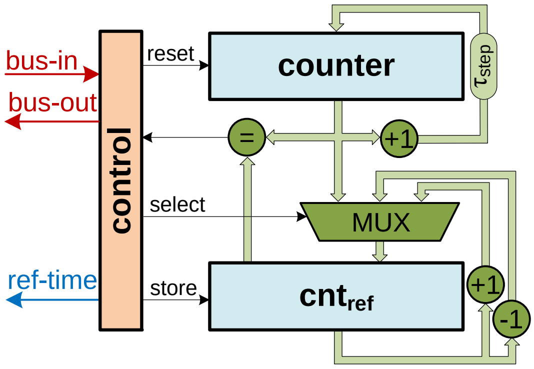


Figure 4.7: Optimized structure of the time-reference generation circuit with continuous rate correction.

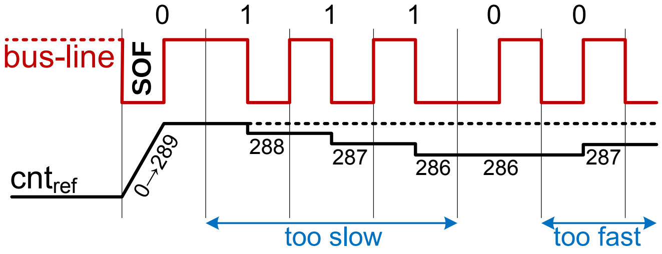


Figure 4.8: Synchronization with continuous rate correction.

incrementer. This is not possible with LFSRs, thus we need to add a second LFSR which changes states at exactly the half rate compared to the first one. This can be achieved by inserting shadows registers S_i after each ordinary register R_i, as shown in Figure 4.6. Consequently, an effective change of state is only performed every second execution step.

In the above list we also introduced a new feature called *continuous rate correction*. Due to the properties of Manchester coding, it is possible to slightly adjust the internal timing *for each bit* by means of rate correction. The SOF sequence is again measured at the beginning of a frame (absolute rate correction), but with each bit we further check whether the bus-transitions occur before or after we actually expected them. As we are assuming the incoming messages to be correct in terms of timing, "late transitions" mean that the internal time-base is too fast (thus we increment cnt_{ref} by one for compensation). On the other hand, "early transitions" indicate that the asynchronous controller is too slow, making a decrease of cnt_{ref} necessary. The resulting block diagram is shown in Figure 4.7, and Figure 4.8 shows an example on how continuous rate correction actually works. Using the described technique we can increase accuracy compared to performing the absolute measurement of τ_{ref} only. Another important advantage is that we have significantly more resynchronization points (each bit rather than each message), thus allowing better adjustment in case of changing execution speeds.

4.4 Experimental Results

After discussing the measurement setup and comparing the five design alternatives in the next two sections, we will take a detailed look at measurements with different operating temperature (Section 4.4.3), varying core power supply (Section 4.4.4), and with different FPGA devices of the same type (Section 4.4.5).

4.4.1 Measurement Setup

Unless otherwise stated, all measurements have been taken on an Altera FPGA-Evaluation board at room temperature $25°C$. The precise FPGA device used is an *EP2C35F484C6N*, which is a Cyclone II device from Altera's low-cost FPGA family. A laboratory power supply is used instead of the shipped power supplies, as the latter usually have very bad stability characteristics. Both the I/O voltage ($3.3V$) as well as FPGA core supply voltage ($1.2V$) are directly connected to the lab power supply using the available test pins. The mounted power-ICs and some related components have been removed from the evaluation board in order to obtain a high quality power supply and remove some potential sources of noise. It is important to notice that we used the same evaluation board for all measurements (the only exception being Section 4.4.5). This is a necessary prerequisite because fabrication and parameter variations (both affecting the FPGA chip itself as well as the external electrical components) considerably change the characteristics of various system parameters (e.g., FPGA timing, jitter, supply ripple/noise, etc.). In order to obtain expressive measurements we only investigate one system parameter at a time (while holding all remaining conditions as constant as possible).

As our primary concern is to gain accurate and reproducible results, we thoroughly prepared the system setup by assuring equal conditions during all measurements. Especially for Section 4.4.5 setting up the trigger condition is crucial in order to guarantee equal preconditions for all measurements, and also the operating temperature including the warm-up phase shall be equal among all runs.

Besides the measurement conditions described above, we use a second FPGA board to emulate the TTP bus. At this point we are not yet interested in global clock synchronization and data exchange (or any other TTP related issues), thus a synchronous FPGA design just periodically generates (random) TTP messages which the asynchronous circuit can use for calibration. While the proposed circuits differ in their specific internal implementation, there are two signals which need to be monitored for all designs:

- `cDone`: The capture-done signal from the time reference generation unit directly represents the low-level circuit's execution speed. This signal toggles once for each execution step performed by the synchronous logic (τ_{step}).
- `ref-time`: This signal toggles once for each compare match of the current counter value and the reference cnt_{ref}. It is the adjusted time-base which will be used by higher-level logic to read/write from/to the bus and perform other time-related tasks (τ_{rep}).

	CntRef*	CntRef	CntManual	LFSRRef	CntRate	LFSRRate
LEDR Gates	134+16	237+46	200+64	156+86	432+81	212+42
LEDR Registers	39	39	43	72	52	42
Logic Depth	16+1	10+0	4+2	4+3	10+1	6+0
Avg. Performance	32ns	25ns	21ns	20ns	25ns	17ns
$f = \tau_{step,r}/\tau_{step,m}$	0.99710	0.99650	0.99397	0.99683	1.00010	1.00029

Table 4.1: Comparison of the implementation alternatives for the time reference generator.

The measurements are executed using a 4 Gigasamples digital oscilloscope. Using all four channels, a single shot can store four million data samples per channel with a resolution of $500ps$. Assuming `cDone` has an average period of $30ns$ (which is worst case), a single shot can store at least ≈ 65000 signal transitions. For example, a confidence interval of $\alpha = 1\%$ for the mean value (assuming $n = 65000$) results in $[\bar{x} \pm t(1-\alpha/2, n-1)\frac{s}{\sqrt{n}}] \approx [\bar{x} \pm 0.01s]$, with s being the sample deviation, $\bar{x}$ the sample mean, and t being the Student-t distribution. In other words, the relatively large number of samples allows us to provide good confidence in the estimated statistical figures.

4.4.2 Comparison

Table 4.1 summarizes the most representative properties of all proposed alternatives. The row "Avg. Performance" shows the respective mean execution speed τ_{step}, averaged over approximately 60000 to 115000 samples, depending on the actual speed of the circuit under test. A second interesting property is shown in row "Logic Depth", which displays the number of (two-input) combinational LEDR gates in the critical path. Inverter stages are explicitly written after the plus symbol. Even for LEDR, inverter stages are just inverters, but they are realized as explicit components. Consequently, they introduce additional interconnect delays. In the table we also point out parameter f (recall Section 4.2.2) to demonstrate that there is a small yet observable difference between $\tau_{step,r}$ and $\tau_{step,m}$. Notice the additional column labeled *CntRef** is the very same design as *CntRef*, but with all synthesis and compile optimizations turned off. This design serves as reference for comparisons.

It is evident from the table that parameter f usually deviates less than 1% from 1 for the chosen implementations. Especially for the designs with continuous rate correction, f is even closer to its optimum value, as we explicitly tried to balance measurement and reproduction phases for these designs (the performed tasks in both states need to be as similar as possible). Although LFSRs are efficient implementation alternatives to ordinary counters, including a second LFSR which counts at the half rate increases complexity (control logic) and area consumption (more registers) in the case of *LFSRRef*. Consequently, we removed SOF-measurement entirely in *LSFRRate*, and just perform continuous rate correction there. Comparing *CntRate* and *LFSRRate* also proves the effectiveness of LFSRs both in terms of performance and area consumption. We can save almost 50% of the logic gates as well as 20% of the registers and at the same time even

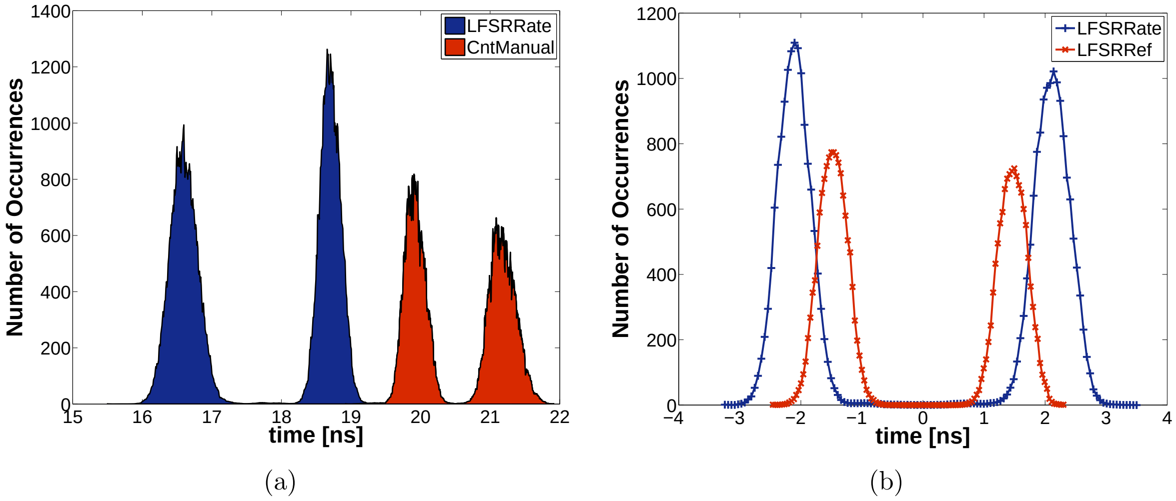
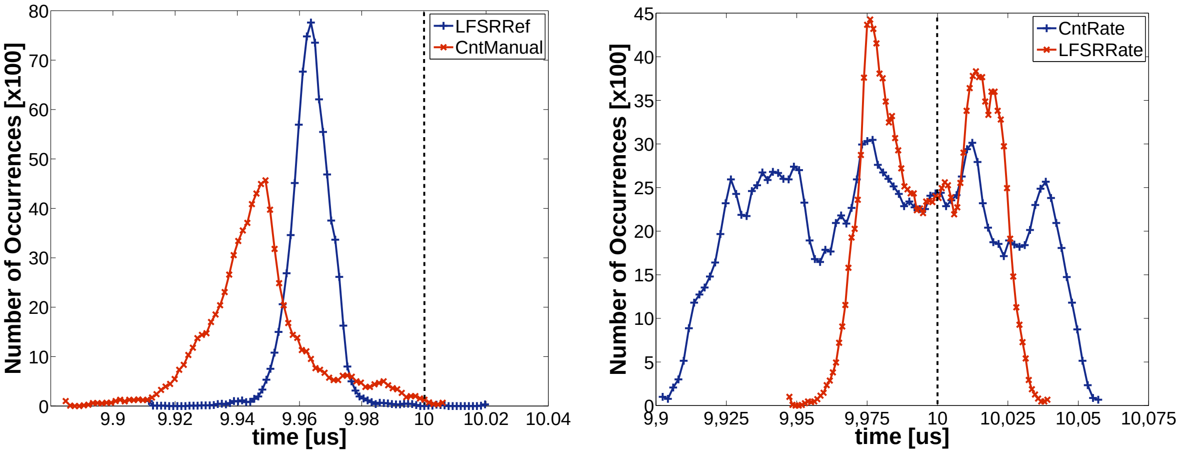


Figure 4.9: Exemplary histograms (a) and cycle-to-cycle jitter (b) for signal `cDone`.

decrease the logic depth by four gate equivalents (40%). Even compared to *LFSRRef*, *LFSRRate* is superior as we save 30 Registers (42%) by removing the half-speed LFSR, and at the same time adding the continuous rate correction functionality.

Let us now take a closer look at the single execution steps of the asynchronous counter. Figure 4.9(a) shows the histograms for designs *LFSRRate* and *CntManual*. Notice that all other implementations show similar characteristics — of course with different scaling on the x-axis according to the circuit's speed, so the discussion can be generalized accordingly. Similar to the experimental results of Section 3.4 we observe two distinct peaks, which are caused by execution steps performed in alternate phases (in other words, one peak corresponds to steps of φ_0, the other peak to steps of φ_1 — the separation is caused by data-dependent execution jitter). This fact is important when it comes to the average execution speed: The mean value of `cDone` for *LFSRRate* is approximately $17.5ns$. However, the circuit only performs execution steps with a duration of approximately $16.5ns$ or $18.7ns$, respectively. In the reproduction phase, this introduces some error depending on whether there is an odd or even number of execution steps per period (in our model, we only consider the *average* duration τ_{step}). Random execution jitter accumulated during signal propagation manifests itself as Gaussian-like shape of the single peaks. To better compare jitter characteristics of different implementation alternatives with different speeds, Figure 4.9(b) shows the cycle-to-cycle jitter (first order difference) of designs *LFSRRate* and *LFSRRef*. Cycle-to-cycle jitter effectively removes the constant component and just leaves "real jitter". The figure confirms the significant separation of execution steps performed in different phases and the quite similar jitter characteristics of the two designs.

In Figures 4.10(a) and 4.10(b) we show jitter histograms of the generated time reference signal `ref-time`. Notice that the histograms in Figure 4.10 have been obtained during reproduction phase only, i.e., the results shown are for fixed cnt_{ref} and thus exhibit mostly *random* execution jitter (in case of cnt_{ref} being odd, there is some additional data

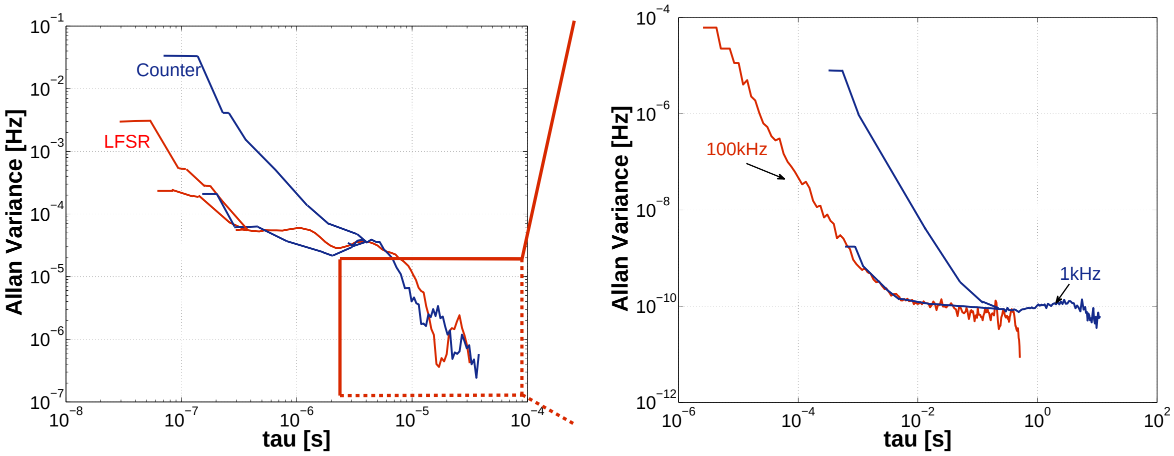


(a) (b)

Figure 4.10: Histograms for design without (a) and with (b) continuous rate correction for signal `ref-time`.

dependent jitter because the relationship between counting states and associated phase changes after each entire period). The figures are separated for designs with (right side) and without (left side) continuous rate correction capability. In this setup, we choose a low reference duration $\tau_{ref} = 10\mu s$ to keep the relative quantization error legibly low. We can see in the figures that the LFSR implementations have a considerably narrower jitter histogram compared to the counter implementations, which clearly is a result of the reduced complexity due to the replacement of the full adders with simple shift registers. It is also evident from the figure that continuous rate correction considerably increases accuracy: Both designs that incorporate continuous rate correction have their mean values quite close to the optimum at $10\mu s$. In contrast, the simple versions deviate significantly from this optimum, which is at least partly caused by the quantization error and parameter f being less than one. Since both *LFSRRef* and *CntManual* are very flat designs with a logic depth of only four gate equivalents, the resulting jitter histograms of Figure 4.10(a) are tighter than the more complex designs in Figure 4.10(b). For our needs, however, it is more important to keep the average execution speed as constant as possible rather than limiting signal jitter.

In order to classify the frequency stability of the different designs we use Allan Variance (*avar*). These plots show the frequency stability of a given signal (y-axis) over steadily increasing averaging windows τ (x-axis) (recall Section 4.1.2). Figure 4.11(a) shows the Allan variance of single execution steps (`cDone`) for both the LFSR and the counter implementation with continuous rate correction capability. As expected, the LFSR design is one to two orders of magnitudes better than the counter (for small τ). This can be explained by the simple structure of LFSRs and the corresponding decreased data-dependencies. For $\tau \geq 2 * 10^{-6}s$, however, both plots are approximately the same. This behavior also satisfies the expectations as for larger τ, data-dependent execution jitter becomes less and less significant, and random jitter dominates. According to the theoretical considerations



(a) (b)

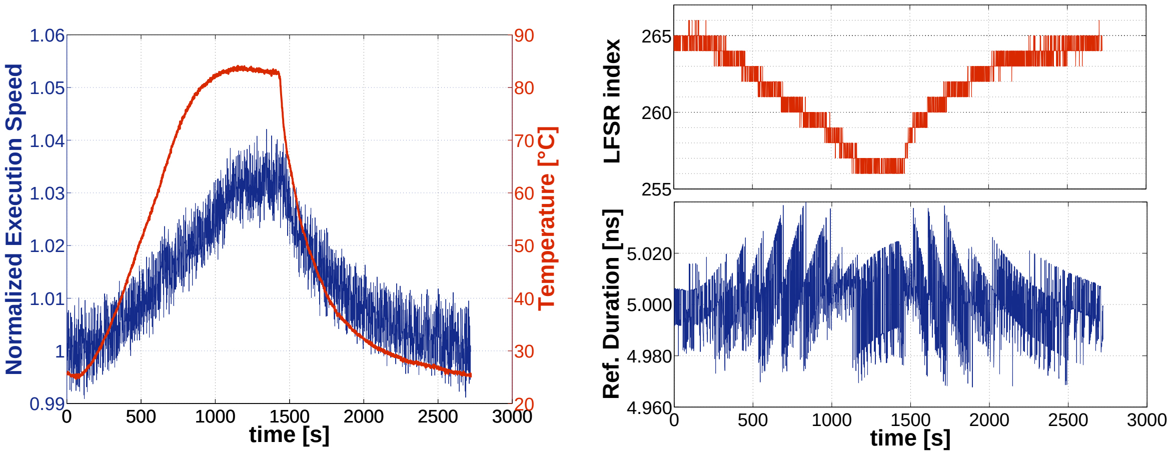
Figure 4.11: Allan Variance of signal `cDone` for counter and LFSR implementation with continuous rate correction (a) and signal `ref-time` for two exemplary baudrates (b).

of the previous chapter, data-dependent jitter is systematic. By periodically counting to cnt_{ref}, all data-dependent jitter effects are the same for each *entire* counting period. In other words, longer observation periods directly result in less DDEJ (relative to the increasing observation period τ). On the other hand, random execution jitter accumulates over time without any bounds. Increasing τ therefore also results in a considerable increase of REJ, making it the dominating jitter component for long observation windows.

Another interesting observation can be made for small τ in this Allan plot: There are two “branches” that converge at about $2 * 10^{-7}s$ and $2 * 10^{-6}s$, respectively. These lines represent the different delays for phases φ_0 and φ_1, because they introduce instabilities for short observation periods. The longer the observation window gets, the less influence these differences have on overall frequency stability.

Even more interesting is the plot shown in Figure 4.11(b). It illustrates the Allan variances for the generated time reference `ref-time` with a nominal bitrate τ_{ref} of 100k and 1k (thus the generated signals have frequencies of about 400kHz and 4kHz, respectively). Figure 4.11(b) is representative for both the counter and LFSR design alternatives, as our measurements did not reveal major differences for the two implementations. Although thorough analyzing of the data reveals that the counter design is slightly inferior to the LFSR implementation, the deviations would not clearly be visible in the figure. This fact is also in accordance with the already mentioned influence of DDEJ and REJ on the frequency stability for different observation periods. For both designs, DDEJ gets insignificant for increasing τ, while only random execution jitter remains. As the counter implementation is more complex, more REJ accumulates over time, thus degrading frequency stability.

As can be seen in Figure 4.11(b), the baudrate itself has only minor influence on frequency stability. At low bitrates, however, an unapparent effect can be observed: There are differences in the low- and high-widths of the generated time-signal, indicated



(a) (b)

Figure 4.12: Temperature vs. normalized execution speed (a) and LFSR-index cnt_{ref} vs. duration τ_{rep} of `ref-time` (b).

by two branches of the 1kHz Allan plot that converge at approximately $\tau = 2 * 10^{-1}s$. The reason for this difference can be explained as follows: If there is an odd number of counting steps, the phase - value alignment toggles for each period (i.e., the counter alternately restarts in phase φ_0 and φ_1). This causes minor changes in the respective propagation delays which accumulate over time, hence the effect can only be observed for low frequencies where there is more time for accumulation. It gradually disappears as the frequencies increase. It can also be observed that the Allan variance of the generated time-signal is up to five orders of magnitude more stable than the underlying execution steps themselves (because the data-dependent jitter is eliminated for matching counting-periods). Suppose we drew the Allan Variance of both `cDone` and `ref-time` in a single plot, we could observe a smooth transition from $avar(cDone)$ to $avar(ref\text{–}time)$, because the latter is directly created out of a sequence of the former — their variances are thus strongly correlated. This is also indicated in Figure 4.11, where the right diagram basically is a detailed view of the marked area of the left diagram (notice the scales of both axis). Clearly, the measurements in Figure 4.11(b) have been taken with an adapted sample period to allow for the prolonged observation duration.

4.4.3 Temperature Tests

One of the main advantages of our solution is the automatic adaption to changing operating conditions. To this end we will now exemplarily investigate the behavior of *LFSRRate* under changing operating temperatures. Comparing all measured data reveals that the results can — on a qualitative basis — also be applied to the alternative designs. In Figure 4.12(a) we show the normalized execution speed (normalized with respect to the "nominal" speed at room temperature) in combination with the temperature measured on the evaluation board of design *LFSRRate*. We plot the normalized execution speed for intervals of one second. As our design is fully asynchronous, changing the operating temperature directly impacts on performance (recall that τ_{step} is, besides others, a function of temperature). In our case the performance loss is, at peak-to-peak temperatures, about 3.5%. Considering the high temperature difference of approximately $55°C$ this might not seem to be dramatic, but it certainly is a showstopper for reliable TTP-communication. Notice that all measurements have been taken during reproduction phase in order to obtain more accurate results.

To demonstrate the effectiveness of the proposed solution, Figure 4.12(b) compares the LFSR-index of the corresponding cnt_{ref} to the signal period of `ref-time`. While the ambient temperature increases, the LFSR-index steadily decreases because the circuit slows down. The period of `ref-time` makes an approximate step of τ_{step} ($\approx 19ns$ in this example, the duration of a single execution step) each time the LFSR-index changes. During the periods where the changes in execution speed cannot be compensated by adapting cnt_{ref} (because they are too small), `ref-time` slowly drifts away from the nominal value of $5\mu s$. Without any compensation measures the duration of `ref-time` would be about $5.180\mu s$ at the maximum temperature, instead of being in the range of about $5\mu s \pm 38ns$ ($\pm 0.76\%$), no matter what temperature ($\pm 38ns$ equals the duration of two execution steps). We are well aware that the presented results can only be seen as snapshot for our specific setup and technology. Changing the execution platform will certainly change the outcomes of our measurements, as jitter and the corresponding frequency instabilities mainly depend on the circuit structure and the used technology. However, from a qualitative point of view, our results are of course valid for other platforms and technologies as well, even if concrete measurements must be taken for a quantitative evaluation.

4.4.4 Supply Voltage Tests

Far more pronounced delay variations compared to the previous section can be obtained by changing the core supply voltage. For the following measurements, the core supply voltage was increased in steps of $20mV$ from $0.8V$ to $1.68V$. The execution speed of the self-timed circuit increases from about $\tau_{step} \approx 80ns$ per step at lowest to $15ns$ for the highest supply voltage, as shown in Figure 4.13(right). This plot illustrates the durations of `cDone` versus the FPGA's core supply voltage on the x-axis. Thereby, the densities of the histogram are coded in gray-scale (dark lines indicate dense distributions). This illustration shows other interesting facts (magnified for better illustration in the close-up): For one, almost all voltages have *at least* two separate humps in their histograms. These are caused by data-dependencies that originate in the different phases $\varphi_{0,1}$. Furthermore, for low voltages, additional peaks appear in the histograms and the separations between the phases increase as well. It seems as though the histograms get "streched", which can be explained as data-dependent effects caused by different delays through logic stages are magnified while the circuit slows down. This characteristic is better illustrated in Figure 4.13(left), where cycle-to-cycle execution jitter is plotted over the supply voltage. The graph appears almost symmetrically along the x-axis, which is caused by the continuous alternation of phases. As one can see, the gap between execution steps performed in different phases is

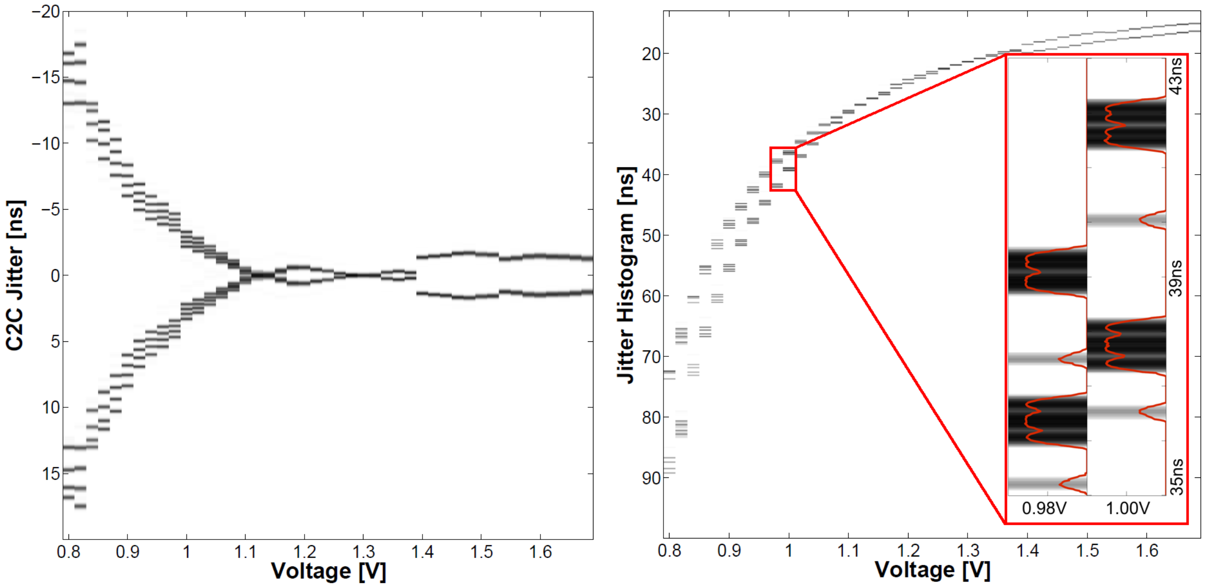


Figure 4.13: Cycle-to-Cycle jitter (left) and jitter histograms of signal `cDone` (right) under varying supply voltage.

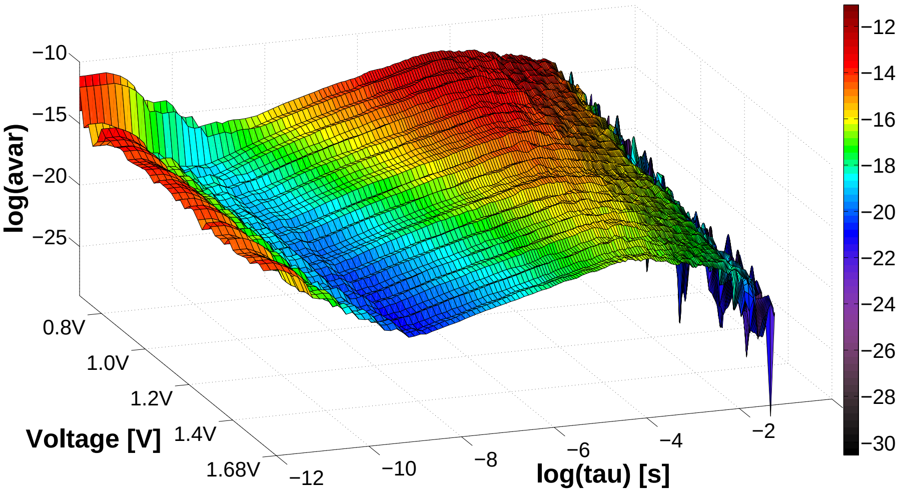


Figure 4.14: Allan Variance of `ref-time` under varying supply voltage.

as high as $15ns$ and rapidly decreases with increasing voltage. We also see that there is an optimum voltage level for which the overall jitter is minimal. For this example it is around $1.3V$. Unfortunately, this cannot be generalized as jitter depends on the specific circuit structure and device parameters — it has to be found for each design and device individually, and is thus only of very limited use for optimizations.

Let us now take a look at the frequency stability of the generated time signals `ref-time`. For each setting of the core supply voltage we performed a measurement of signal `ref-time` (as usual in reproduction phase only to avoid resynchronization during data capturing)

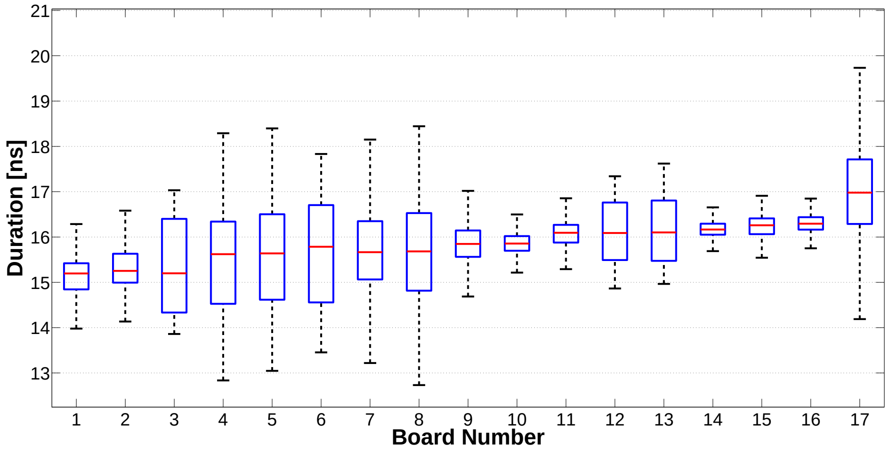


Figure 4.15: Box-and-whiskers plot of `cDone` signal for all 17 boards.

and plotted its respective Allan Variance in Figure 4.14. Notice that two axis are shown in logarithmic scale to make the Allan plot more expressive. On the x-axis the figure shows the averaging window $\log \tau$, while the vertical axis depicts $\log(avar(\texttt{ref-time}))$. The color-bar therefore represents the exponent of the actual Allan variance. The remaining axis finally shows the corresponding supply voltage levels. As we can see in the figure, lowering the system's supply voltage significantly degrades the achievable accuracy (by slightly more than two orders of magnitude from the best to worst case). Clearly this is directly correlated to the spreading of the jitter histograms we have seen before, which results in a more significant accumulation of jitter throughout the logic stages. However, decreasing circuit speed also negatively impacts on the achievable resolution: With steadily falling cnt_{ref} the quantization error increases and so does err_{sys}.

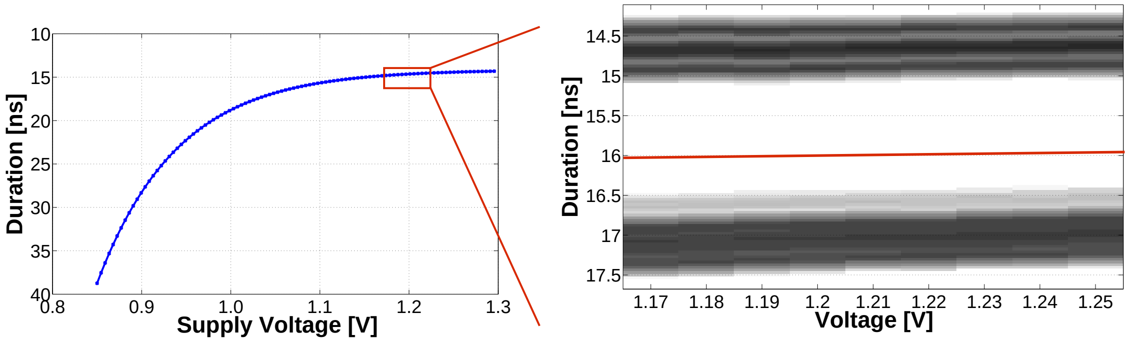
We conclude that varying operating conditions not only affect the speed of asynchronous circuits, but also the respective jitter characteristics. In this perspective, slower circuits tend to have higher jitter, which is further magnified by increased quantization errors due to the low sampling rate. Depending on the specific circuit and device, it might be possible to find an optimum operating point (with respect to jitter) by adjusting the core supply voltage accordingly.

4.4.5 Fabrication Variations

In this section, we want to investigate how process variations affect the execution speed of asynchronous circuits. To this end we synthesize *LFSRRate* for an Altera Cyclone IV (EP4CE115F29C7) device and download this one programming file to 17 identical FPGA development boards[3]. For each board, we perform a series of measurements to determine

---

[3] Unfortunately we do not have as many Cyclone II board, thus we needed to change the device family to Cyclone IV. Furthermore, it was not possible to control the core supply voltage without applying risky modifications on the development boards, so we need to address this issue separately.



(a) (b)

Figure 4.16: Voltage-speed relationship (a) with detailed measurements around 1.2V (b).

the specific speed of the time reference generation circuit. An investigation of achievable execution speeds and jitter characteristics among several "identical" devices will certainly reveal interesting insights about process variations. The properties we investigate are the execution speeds and jitter characteristics of the low-level reference time generator. Figure 4.15 summarizes the results taken for signal `cDone` as box-and-whisker diagram. These diagrams represent statistical data in a very concise form and are thus well suited to visually compare different data sets to each other: The solid box in the middle marks the upper and lower quartiles, while the line inside this box defines the median. The whiskers (dotted lines starting from upper and lower quartiles, respectively) indicate the total range of data (excluding outliers — those are sometimes included as separate circles or dots beyond the whiskers). The results in the figure have been sorted by the statistical mean values in ascending order. The x-axis defines the FPGA board number, and the y-axis shows the duration of execution cycles in nanoseconds. Analyzing the data reveals that the execution cycles have an average duration varying from approximately $15.12ns$ to $16.95ns$. While this does not seem to be significant at a first glance, the relative change (compared to the slowest board) is as high as 12%. Taking into consideration that the device manufacturer already performs a rather strict pre-selection of the devices according to their specific speed-grade, and the fact that a (random) sample of only 17 devices does most probably not cover extreme outliers, the results are remarkable indeed.

In our setup we have taken care to establish equal and reproducible conditions. Therefore, the variations in execution speeds shown in Figure 4.15 are mainly caused by process and fabrication variations among the different FPGA devices. However, also the other components on the development boards are subject to fabrication uncertainties. In other words, the measured data does not only reflect process variations of the FPGA device, it subsumes all variations of the entire development board (most importantly of the board's supply voltage, see Section 4.4.4). In practice, this is a fact that cannot be changed anyway, because different units will always have different electrical characteristics. However, we want to take a closer look on how critical the voltage fluctuations are for our target platform. We exemplarily performed a series of voltage measurements on one of the boards, minimally varying the supply voltage around the nominal value of $1.2V$ (we

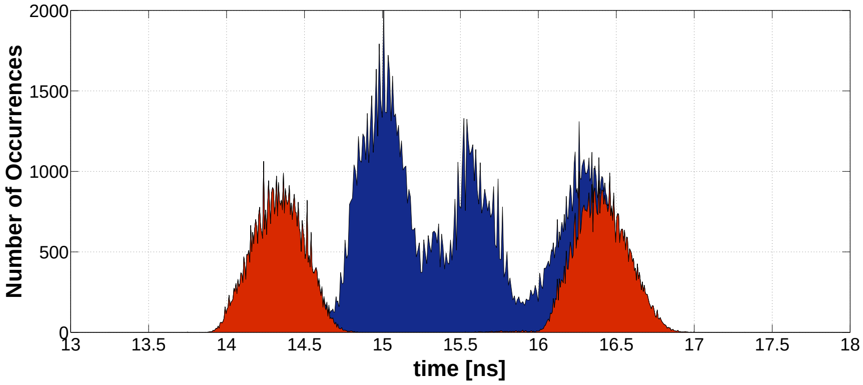


Figure 4.17: Jitter histograms for two exemplary boards.

changed the voltage from $1.17V$ to $1.25V$ in steps of $10mV$, as these are the minimum and maximum voltages measured throughout all boards). The results of these measurements are illustrated in Figure 4.16(b). The color-map shows the histogram of the monitored `cDone` signal. Like before, dark areas indicate higher densities for the corresponding durations, and there again are two separated peaks for phases $\varphi_{0,1}$. The line in the center highlights the mean values of the data series and shows only a very decent voltage dependency of less than 1% for the depicted range. Consequently, it can be assumed that it does not notably affect the delay and jitter characteristics. However, following the non-linear trend in Figure 4.13(right) one can suspect that the speed/voltage relationship would be considerably more pronounced for, e.g., the $1V$ devices of the Cyclone IV family.

Another observation can be made: When comparing the jitter histograms measured on different boards, the specific characteristics are often completely different from each other. Figure 4.17 shows two jitter histograms taken from different devices. While one has a gap of almost $2ns$ between two separated peaks, the other one is considerably more concentrated around its mean value at $15.5ns$. This observation can be explained by *within-die* or *intra-die* [7, 50] process variations that do not affect the entire chip in the same way. Adjacent islands with approximately matching properties can be identified [73], but from a global point of view, a systematic influence is difficult to identify. In contrast, *die-to-die* or *inter-die* variations are of a more systematic nature and apply to all timing paths in a statistically similar way (in our case within a range of $\approx 12\%$). As the asynchronous part of our design is rather complex and area-consuming, it uses a wide-spread portion of the available die. Consequently, some parts (interconnects, combinational logic, etc.) are faster, some other parts are slower specifically for each device. During propagation of a signal from a location A to another location B it passes different “speed regions” on the chip and accumulates chip-specific jitter on its way, thus resulting in completely different jitter histograms.

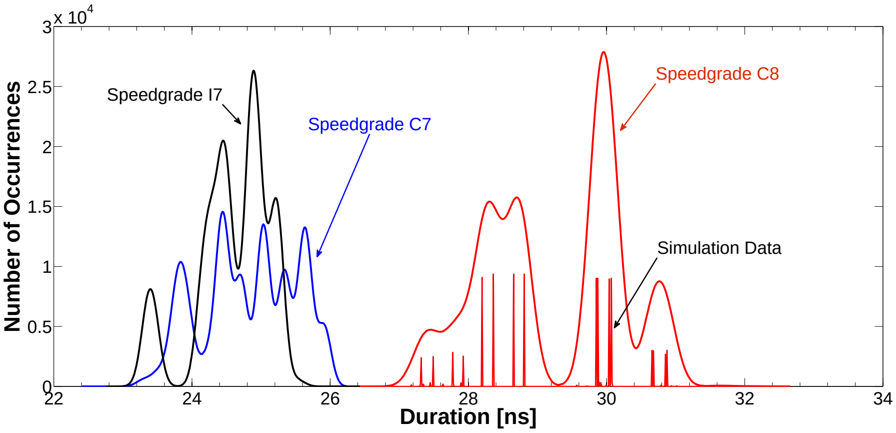


Figure 4.18: Simulation histograms for three different speed-grades

4.4.5.1 Simulation

One remaining question is whether computer simulations also show fabrication variations in the execution speeds of asynchronous designs. We therefore synthesize our design with Altera Quartus II and perform post layout / post fitting simulations with Modelsim VHDL simulator (using static timing models only). The advantage compared to real-world measurements is that we now do not have any temperature, voltage or process variations that influence the results. Furthermore, signal propagation delays are deterministic, as they are not subject to any signal jitter. Figure 4.18 shows three different simulation histograms for the available $1.2V$ speed-grades C7, C8 and I7[4]. The spikes under C8's graph are the original discrete simulation results (scaled vertically to improve readability), which are difficult to read/interpret. To obtain a more illustrative curve, we apply Gaussian jitter to the simulated values by means of convolution. It is important to notice that this does not resemble the actual behavior of real circuits, because we apply jitter to the *final* signal transitions only, rather than for each logic primitive or interconnect along the signal's propagation path.

We can see that also the computer simulations result in relatively wide-spread "jitter" histograms. All delay variations have their origin in data-dependencies inside the asynchronous design — different states possibly enable different signal paths, thus leading to different propagation delays. Considering curve C7 we have a spread from $\approx 23ns$ to $\approx 26ns$, or 12% compared to the average duration. The deviation of the simulation results is thus in the same range as the variations observed among the different development boards, consequently both effects significantly influence the overall signal characteristics of the asynchronous design. Also notice the discrepancies between simulated and measured execution speeds (approx. $24ns$ vs. $16ns$ for the very same design), which allows rough estimations on the safety margins included in static timing analysis. When comparing the simulation results of different speed-grades to each other, we see that while C7 and

---

[4] The devices just differ in their speed grade, the design is the same for all three devices. C and I speed-grades specify commercial and industrial devices, respectively.

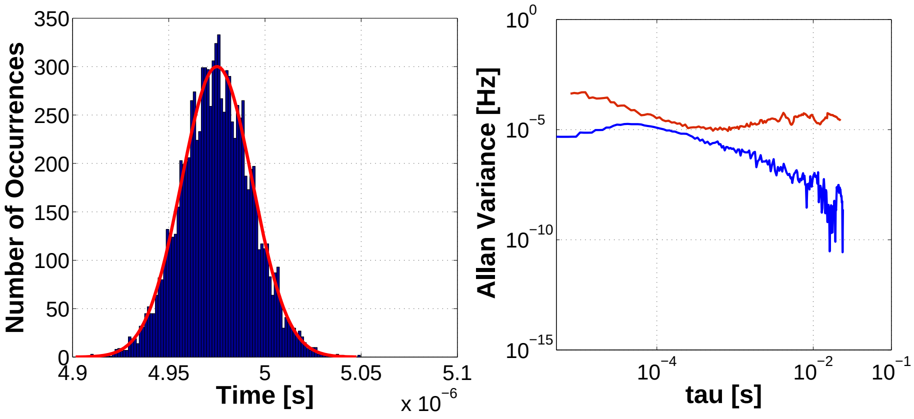


Figure 4.19: Reference time histogram (left) and Allan Variance with and without resynchronization (right).

I7 operate basically at the same speed, C8 reaches only about 85% of C7's performance. Even for the simulations the shapes of the histograms do not match, which further proves the critical impact of inter-chip timing variations on the overall signal characteristics. Taking a closer look at the timing characteristics of different speed grades one notices that not all elements of a chip are affected similarly: For example, lookup tables might have a different relative "speedup" than memory cells, and global/local interconnection speed most probably scales differently than that of multiplier blocks (recall inter-die variations). Consequently, switching to another speed grade does not simply "squeeze" or "stretch" the jitter histogram, but leads to a totally new curve.

4.4.6 Frequency Stability

In this section we focus on the quality of the generated time reference, which is the central component of our TTP controller. In Figure 4.19(left) we measure the duration of the generated time reference while no communication is active on the bus. This is necessary in order to prevent the module from resynchronizing itself to the bit-stream, which would lead to considerable changes in the signal's period and thus invalidate any investigations on frequency stability. We can see in the graph that the signal is slightly off the optimum value (which is $5\mu s$ for the given setup) having a mean value of only $4.975\mu s$. The Gaussian-like shape of the histogram is a result of accumulated random jitter, and has a standard deviation of $\sigma = 18.4ns$. Interestingly, the relatively distinct shapes of jitter histograms presented earlier (e.g., in Figure 4.17) are not observable any more — statistical averaging over thousands of execution cycles (thus eliminating data-dependent jitter effects) leaves a relatively clean Gaussian-like characteristic, which is also predicted by the model in Chapter 3. We further want to consider Figure 4.19(right) which shows two Allan Variance plots of the same time reference signal (both measured on the same board). However, one time bus communication was active (the red one at the top, measurement phases occur periodically), while for the other graph no communication was allowed (the bottom blue one, reproduction phase only). Not surprisingly, resynchronization negatively impacts frequency stability because the reference counter-values are continuously updated. For our application, however, it is important that the frequency of the reference signal stays constant if *no* communication is active, because in that case the previous measurements are the only information available to keep the internal time accurate. The bottom blue graph represents this latter case and clearly shows that frequency stability is one to two orders of magnitudes better compared to the other case. This behavior also reflects our expectations.

4.5 Chapter Notes

In this chapter we proposed a method to generate a time-reference for our envisioned asynchronous controller out of the bit-stream provided by TTP. We presented five concrete implementations with different levels of optimizations, and compared them to each other with respect to performance, frequency stability, and other important properties. Although there are no significant differences in frequency stability, the designs with continuous rate correction are more accurate as they better compensate quantization errors and systematic delay-variations. We also used these designs under changing operating conditions and performed systematic measurements:

- The *temperature tests* show that when heating up the system from about $25°C$ to $85°C$ a performance loss of approximately 3.5% can be observed. However, the proposed circuits are capable of adapting themselves to changing operating temperatures while maintaining a relatively stable reference duration (with approximately $\pm 2\tau_{step}$).

- A far more pronounced impact on the execution speeds of the asynchronous designs is observed when varying the *core supply voltage*. Here the execution period of the design significantly increases from just $15ns$ at $1.68V$ to almost $80ns$ at $0.8V$, which is a factor of 5.3. We have also seen that the jitter histograms spread considerably with lower supply voltage — in combination with the reduced sampling rate this results in loss of accuracy for the generated time reference.

- Finally we have investigated the impact of *fabrication variations* on our design. While we observed a difference of almost 12% regarding the execution speeds of the asynchronous circuit, reproduction accuracy was relatively stable among all tested boards. We further found that process variations significantly influence the jitter and delay characteristics of signals propagating though the chip, which results in totally different jitter histograms for the very same design running on other devices.

Chapter 5

Asynchronous TTP Controller

> *There is a theory which states that if ever anyone discovers exactly what the Universe is for and why it is here, it will instantly disappear and be replaced by something even more bizarre and inexplicable. There is another theory which states that this has already happened.*
>
> Douglas Adams

In this chapter we finally integrate the previously presented time reference generation unit into a fully operating TTP cluster. While we have already discussed the advantages and disadvantages of the various implementation alternatives in Chapter 4, only the most promising design (i.e., design *LFSRRate*, the LFSR implementation with continuous rate-correction and without absolute SOF measurement) is considered. Recalling Table 4.1 of the previous chapter, *LFSRRate* is superior compared to the other alternatives (and especially the counter designs):

- The gate count is relatively low as LFSR are very simple counting constructs.
- The logic depth is very low. Actually, control logic dominates the logic depth while the counting logic itself has a depth of only one gate equivalent.
- There is no more overhead in registers as SOF measurement has been removed (recall the LFSR counting at half speed using additional shadow registers).
- The corresponding jitter histogram is relatively narrow, especially compared to design *CntRate* because of the reduced complexity.

Also for measurements on physical devices, *LFSRRate* produces slightly better results than any of the other designs while using less area and achieving better performance.

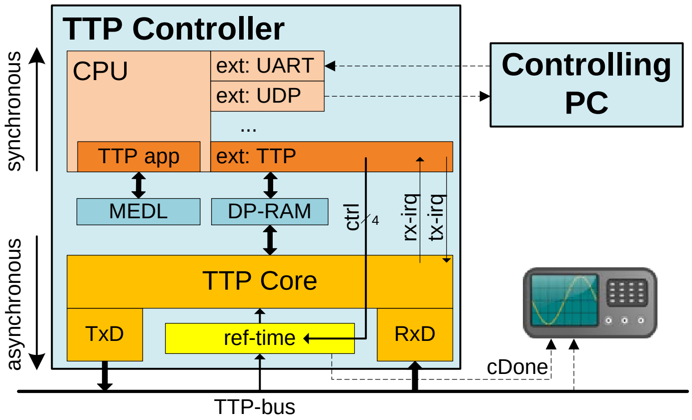


Figure 5.1: Block diagram of simplified asynchronous TTP controller.

The remainder of this chapter describes in detail the implementation structure of envisioned TTP controller, and the system architecture of the TTP cluster. As we will also see in this chapter, it is necessary to make some modifications to the time reference generator design in order to allow for convenient interaction with the controlling software. Before explaining the experimental results, we also review the expected limitations of the proposed setup. Experiments are concerned with both passive (listening mode only) and active (sending, hitting sending slots, synchronization) TTP communication.

5.1 Implementation Details

This section describes the architecture of the asynchronous TTP controller's hard- and software, while the cluster configuration is finally presented in Section 5.1.3. Table 5.1 summarizes all function blocks from Figure 5.1 (see also Sections 5.1.1 for details on hardware blocks, and 5.1.2 for information on software components) and shortly describes their respective tasks.

5.1.1 Hardware

Figure 1.17 on page 26 illustrates the basic blocks of the existing synchronous TTP controller solution. To shortly summarize, there is an external host-CPU (usually a Motorola *MPC555* microprocessor), which uses the *Communication Network Interface* to communicate with the TTP controller. Communication is performed in a memory mapped manner, which means that the host just read/writes from/to specific memory addresses inside the TTP controller via a default memory interface with data-, address-, and some control-signals. Clearly, the block marked “ref-time” is not present for synchronous TTP controllers, as crystal oscillators are available for generating a time reference. Inside the TTP controller, there is a dual-ported RAM which temporally separates the CNI from the TTP core. Following a modular concept, bus access is achieved using external physical

Block	**Description**
CPU	Synchronous CPU core: – Executes “TTP app” with each receive interrupt request `rx-irq`
ext: UART	Extension module: – Receives code for CPU via serial interface from controlling PC;
ext: UDP	Extension module: – Sends high-speed data via ethernet interface to the controlling PC;
TTP app	Software that runs on CPU: – Evaluates and interprets incoming messages – Assembles new messages – Calculates checksums, time correction factor, etc.
ext: TTP	Extension module: – Provides access to DP-RAM – Provides access to `ctrl`, `tx-irq` and TTP control register
MEDL	Necessary data of the TTP schedule – Message lengths – Slot-assignments – Message-type, etc.
DP-RAM	Dual-ported RAM: – Temporal firewall for synchronous and asynchronous modules – Data exchange between “ext: TTP” and “TTP core”
TTP core	Managing hardware layer: – Read and write access to DP-RAM – Execution steps triggered by “ref-time” – Generates TTP timestamps – Controls sending and receiving messages
Txd	Transmitter unit: – Generates TTP specific Manchester code – Generation of preamble and postamble – Bit-encoding
Rxd	Receiver unit: – Decodes incoming bit-stream – Detection of preamble and postable
ref-time	Generates reference time for “TTP core”: – Measures bit-durations, adapts cnt_{ref} – Allows manual correction of cnt_{ref} with control signals `ctrl` – Automatic adaption to changing operating speed – Periodic time reference generation if bus is idle
Controlling PC	Personal Computer: – Compile and download “TTP app” – Receive debug data via ethernet interface – Store, evaluate and illustrate collected data

Table 5.1: Overview of the different function blocks (cf. Figure 5.1).

layer boards for all kinds of transmission and encoding standards. From the controller's point of view, there is just one input signal for reading, and one output signal for writing.

Based on this structure we develope the asynchronous version of the controller. The result is shown in Figure 5.1, and has some significant differences compared to the original design (the implications of which are described in detail in Section 5.2). As can be seen in the picture, a (synchronous) processor core is integrated in the otherwise asynchronous design (the orange and yellow blocks are fully asynchronous). This core manages all communication with the controlling development computer, i.e., application download using the serial UART interface, and measurement/debug data transmission using UDP packages on the ethernet interface[1]. It is important to notice that this processor core does not perform any time-critical operations, nor does it use any synchronous timers. It would in principle be possible to replace the synchronous core with an asynchronous version, as its functionality is limited to managing/debugging tasks which are neither time- nor performance critical. However, for reasons of simplicity, a synchronous core is used. As indicated in the figure, the processor can be extended with memory-mapped "extension modules". The TTP controller itself is also integrated in the scope of such an extension module. Data exchange between the synchronous and asynchronous parts are handled using a dual ported RAM: The asynchronous receiver stores the raw data (in combination with an asynchronously generated timestamp) of received messages in the DP-RAM, and signals the processor core the arrival of new data with a dedicated interrupt signal `rx-irq`. The only timing assumption made is that the processor must read these data before the next message is received, otherwise the old data is lost (this assumption is easily fulfilled because of the known operating and transmission speeds). Similar to receiving messages, the transmitter (in our case the CPU) can store messages with associated timestamps in the DP-RAM and assert the transmit-request to the "TTP-Core" unit with another dedicated interrupt line `tx-irq`. As soon as the specified time is reached, the message is transmitted automatically. Generating the reference time is completely managed in the lower-level asynchronous block labeled "ref-time" (i.e., design variant *LFSRRate* from Chapter 4). This module runs independently from the remaining designs, continuously monitors the TTP bus, and generates the reference time signal `ref-time` out of the received bit-stream. This is achieved by measuring (and in turn reproducing) the durations of single bits with an asynchronous, free-running counter (cf. Chapter 4). Clearly, however, due to process, voltage and temperature variations, it is necessary to periodically adjust the measured counter value to the bitstream as changing environmental conditions lead to variations in the operating speed. During the experiments it turned out that additional means for controlling block "ref-time" need to be included in the design. Consequently, four additional signals (labeled `ctrl` in the figure) are introduced to perform the following actions:

---

[1] The SPEAR (Scalable Processor for Embedded Applications in Real-time environments) processor was developed at our department, is easy to integrate into the controller design, has relatively low resource requirements, and features a working *gcc* toolchain. Although the processor was developed for real-time applications, none of the respective features are needed for the current work.

- `ref-inc`: Each *transition* on this signal increases the current counter value cnt_{ref} by one. This way, the software can directly control the duration of the time reference and circumvent automatic resynchronization (in combination with `ref-freeze`).
- `ref-dec`: Each *transition* on this signal decreases the current counter value cnt_{ref} by one. This way, the software can directly control the duration of the time reference and circumvent automatic resynchronization (in combination with `ref-freeze`).
- `ref-freeze`: When *set*, the current value of cnt_{ref} does not change during resynchronization. This feature is useful to test the robustness of the design when resynchronization is disabled. When *cleared*, each received bit is used to resynchronize according to the current operating speed, thereby eventually adjusting cnt_{ref}.
- `mask-bus`: When this signal is *active*, module "ref-time" is disconnected from the TTP bus (i.e., it receives the idle state, no matter what the bus currently transmits). This feature is useful to deliberately hide (entire or parts of) messages.

We also see in the figure that the interface to the outside world is relatively simple. There are two extension modules for the SPEAR processor core: The UART module for downloading new software using the serial interface, and the UDP module for transmitting high-speed debug data from the TTP core (i.e., received messages with timestamps, data, checksums, and other useful information needed for evaluation). Besides the physical bus interface, which just consist of two signals for bus-read and bus-write access, there is only one additional debug signal `cDone` in order to monitor the low-level asynchronous execution speed. Most notably, there is no dedicated Communication Network Interface for interaction with the host CPU (in contrast to Figure 1.17): In order to keep the focus on the relevant effects, we decided to execute the simple TTP application ("TTP app", cf. Figure 5.2) directly within the already existing CPU core. This does not in any way influence the functionality or change the temporal characteristics of the real-time application.

5.1.2 Software

The main CPU, i.e., the SPEAR processor core, executes the TTP application. However, this task is only a very small part of the software that is actually executed. Figure 5.2 shows a flow-chart of the entire software stack run by the CPU. As one can see, the host application itself, labeled "TTP task" in the figure, is only one of many different blocks. A detailed description of all TTP hardware configuration registers, the most important data structures, as well as the host application are provided in Appendix A. While the software is idle most of the time, it waits for the arrival of a receive interrupt by means of signal `rx-irq`. After reading out the message from DP-RAM, checking both the C-state's and the message's CRC, extracting the C-state and retrieving the data, the routine takes

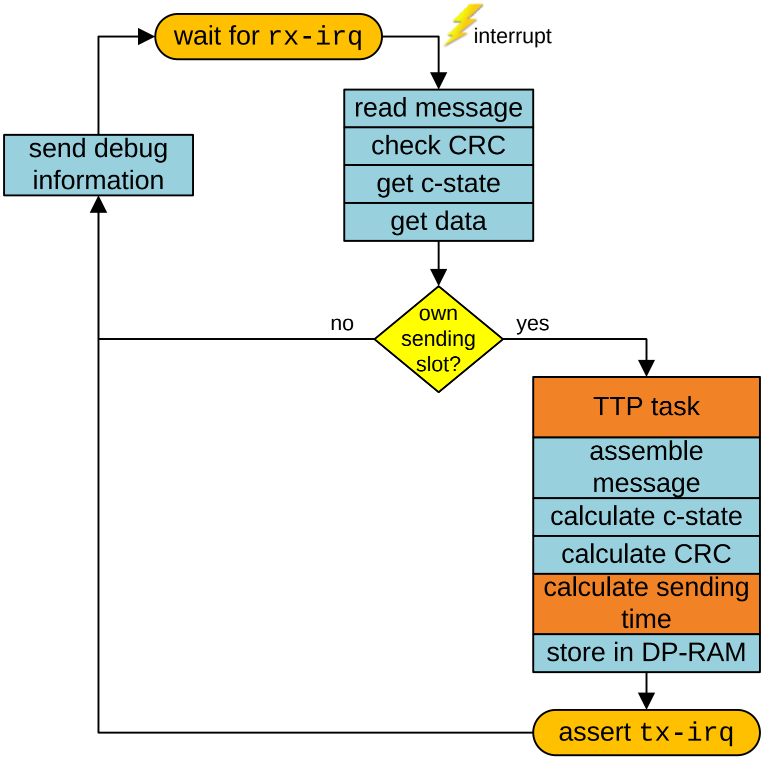


Figure 5.2: Flow chart of the main controlling application software.

two different branches depending on wether the own sending slot is active or not[2]. In any case, before leaving the subroutine and waiting for another `rx-irq`, debug information (i.e., message data, C-state, synchronous and asynchronous timestamps, global time, etc.) is sent to the controlling PC via the ethernet interface.

In case the currently active TDMA slot is the node's own slot, a couple of additional tasks must be executed. First of all, the host application "TTP task" must run in order to evaluate all received data messages and store the node's own data in the respective data structure. We will return the to host application in the next section. After assembling the final message, calculating the corresponding C-state and determining the message's CRC values, the critical task of evaluating the actual sending time is performed (how this is done is explained later in this section). Finally, the complete message including message length and pre-determined sending time are stored in DP-RAM and the transmit interrupt request is asserted with signal `tx-irq`. Only now the asynchronous controller accesses the DP-RAM and retrieves the timestamp and message data for transmission.

Calculating the sending time is a central task performed by the software. It does of course not use any synchronous timestamps to calculate the respective sending point. In

---

[2] The `rx-irq` marks the *end* of a specific TDMA round, which means that upon execution of the interrupt service routine, the current slot is the one succeeding the one whose message has just been received.

contrast, only the timestamps generated by the asynchronous TTP core are used for evaluation. As the message schedule is deterministic and all information is available at compile time already, it is easily possible to compare the actual arrival times to the expected ones. Due to the predefined baudrate, we know exactly how long one asynchronous execution step *is supposed* to last (according to Section 4.2.2 this ideal duration was named τ_{ref}). Let further ω_i, $\Sigma_{nom,i}$, $\Sigma_{act,i}$, t_i, and δ_i be the nominal duration, the nominal number of asynchronous execution steps, the actual number of performed execution steps, the asynchronous timestamp, and the difference between actual and nominal execution steps of slot i, respectively. Obviously, $\Sigma_{nom,i} = \frac{\omega_i}{\tau_{ref}}$, $\Sigma_{act,i} = t_i - t_{i-1}$, and $\delta_i = \Sigma_{act,i} - \Sigma_{nom,i}$. Generally we can expect $\delta_i \neq 0$ and even $|\delta_i| > 1$ because of several reasons already mentioned in Section 4.2.2 (quantization error, $\tau_{step,m} \neq \tau_{step,r}$, etc.). By deriving a respective correction factor $f_{corr} = \frac{\Sigma_{act,i}}{\Sigma_{nom,i}}$ we can now easily evaluate the expected sending time of our own message: $t_{i+1} = t_i + f_{corr} * \Sigma_{nom,i}$. For the special case where all sending slots have equal duration (which is true for our setup), i.e., $\Sigma_{nom,i} = \Sigma_{nom}\ \forall i$, this can also be written in terms of the actually observed execution steps, thus no floating point operations are necessary: $t_{i+1} = t_i + \Sigma_{act,i}$.

Together with this feature, the proposed TTP controller allows for up to three levels of resynchronization:

- *Absolute rate correction*: Performing an absolute measurement of the duration of the first half-bit of each transmitted message. Recall that this feature has been removed for the used design *LFSRRate* in order to improve performance and reduce complexity. An adequate initial value must be provided by software in this case.

- *Continuous rate correction*: Each received bit allows to adjust the internal reference value cnt_{ref}. This is possible due to the fact that Manchester code provides at least one transition per bit which can be used as resynchronization point.

- *High Level correction*: The controlling software knows about the nominal and actual slot durations and can calculate a respective correction factor out of these numbers and change the respective message sending time according to the current speed of operation of the asynchronous logic.

5.1.3 TTP Cluster

Figure 5.3 presents a photograph of the TTP cluster architecture used for all experiments. There is the actual TTP cluster (on the very left), which contains four synchronous nodes. These nodes are labeled *nodeA* for the leftmost throughout *nodeD* for the rightmost device. Furthermore, in the middle of the figure, there is a so called Monitoring-Node, which passively monitors TTP communication and provides all received data in combination with debug and timing information to a visual interface on the respective personal computer or notebook using the ethernet interface. The Monitoring-Node also provides services for application software and schedule updates. The bus itself consists of a series of twisted-pair cables, forming a line topology where all four nodes and the Monitoring-Node

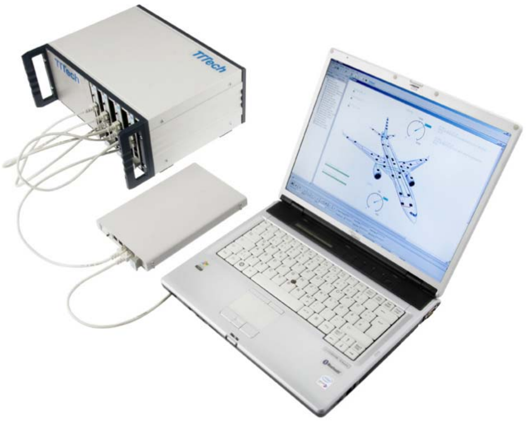


Figure 5.3: TTP cluster architecture, symbolic photo (with the courtesy of TTTech Computertechnik AG, source: http://www.tttech.com/products/ttp/).

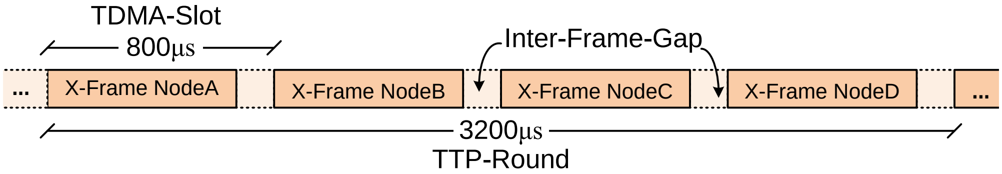


Figure 5.4: TTP schedule (nodes, TDMA slots, TTP-round).

are connected to. Notice that only *nodeD* will be replaced by an asynchronous controller design — the other three nodes remain synchronous in any case.

The central property besides the hardware configuration for a TTP system is of course its schedule. Figure 5.4 illustrates the schedule for the previously described cluster. For simplicity, all nodes send the same frame types, have the same net data length, and therefore have similar slot durations. Notice, however, that this is not a restriction of the asynchronous node. Irregular schedules with different message types and slot lengths are of course supported by our design. Nevertheless, using a more complex configuration does not provide any additional insights, so we stick to the regular system configuration. One TTP round is configured to last $3200\mu s$, which gives each TDMA-slot a total amount of $800\mu s$ for data transmission and is sufficiently long for the lowest valid baudrate (including inter-frame gaps and spacings). Keeping these values constant, the actual ratio of data transmission time to idle time obviously depends on the configured baudrate. A higher transmission speed for the same amount of data and the same duration of the sending slot automatically increases idle times. As shown in the figure, the four slots per round are statically assigned to the respective nodes starting with *nodeA* through *nodeD*. Consequently, the asynchronous *nodeD* always transmits last in each round.

TTP distinguishes (besides others) two basic types of frames: *I-frames* and *X-frames*. The former only contain a 4-bit header, followed by the user-data field and a 24-bit checksum. The C-state is not explicitly contained in I-frames, but it is included in the

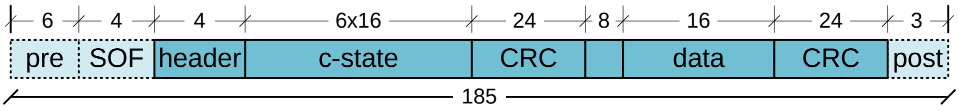


Figure 5.5: Bit-fields of X-frames for our configuration.

Name	**Bits**	**Description**
`ID`	[15...12]	One-hot encoded ID. Node A has 0001, node D has 1000.
`Sum`	[11...8]	Sum of *Cnt* fields of all *other* nodes.
`CntD`	[7...4]	Last received *Cnt* from node D (asynchronous node)
`Cnt`	[3...0]	Node-specific, independent counter. Increments once each round.

Table 5.2: Semantics of the 16-bit data field for the host application.

CRC calculation to allow the receiving nodes to find out whether their local C-state are consistent with the remaining cluster. On the other hand, X-frames contain the C-state explicitly, which is the reason why all nodes in our system are configured to send X-frames only. This makes message allocation and traffic monitoring must easier, but clearly introduces some overheads in message length. Figure 5.5 shows the single bit-fields of an X-frame including the respective bit-width for each field. Fields shaded in light-blue are not part of the actual message, but are added/removed by the transceiver to/from each message automatically. These fields are *pre* (preamble, fixed sequence 1010), *SOF* (start of frame sequence, fixed sequence 1011), and *post* (postamble, fixed sequence violating Manchester encoding[3]). The actual message consists of a 4-bit header containing the type of frame (i.e., I-frame or X-frame), directly followed by the $6 * 16$-bit wide C-state (containing global time in macro-ticks, slot count, and the 64-bit membership vector). Both header and C-state are validated by a 24-bit CRC. After a dummy byte the actual data section starts. In our very simple case, this data section is only one word wide, but TTP supports up to 127 words for the data section. Finally, the *entire* message (i.e., also the header, the C-state, and the respective CRC) are secured by another checksum. To summarize, all messages sent by any node of the described system have a fixed length of 185 bits and always send the C-state as part of the message.

Finally, Table 5.2 shows the semantics of the application specific 16-bit user data field (also refer to A.4 for a more detailed explanation of the host application). All nodes basically execute the same application[4], thus the table is valid for all nodes. Clearly, the node-specific counter *Cnt* must be substituted accordingly for each node (i.e., *CntA*, *CntB*, etc.).

---

[3] The half-bit sequence 000111 is sent as postamble. As Manchester encoding does not allow more than two consecutive half-bits of same polarity, this is a violation.

[4] Clearly, this is not necessary, but we want to keep the complexity low and maximize debug capabilities.

5.2 Limitations and Restrictions

In this section we want to take a closer look on limitations and restrictions that are introduced by the use of our asynchronous TTP controller. Some of the limitations are only of minor interest for the reliable functionality of TTP, and are rather concerned with maintainability and compatibility. However, having a totally independent asynchronous node in the system also leads to some disadvantages concerning reliability, precision, and fault-tolerance of the remaining system. Notice that the following list restricts itself to limitations that directly follow from our implementation and the characteristics and requirements of TTP. Findings during the experimental evaluation are discussed later.

- While the controller itself is fully compatible with the existing hardware platform, the host interface (CNI, see Figure 1.17) is not implemented. Therefore, using an external host CPU is not supported. This also implies that software (e.g., the host application) and schedule updates cannot be performed using the well established tools provided by TTTech. Instead, software must be downloaded manually via the serial interface.

- The data structure of the TTP schedule is not compatible with the existing one. Our implementation only accounts for features that are essential for our purposes. As a consequence, changes in the schedule must be adopted manually for the asynchronous node. All necessary information must be extracted manually from TTTech's tools (e.g., baudrate, slot-durations, message length, frame-type, schedule-ID, etc.). This is also the main reason why a regular schedule is beneficial in our case.

- Due to the fact that FPGAs are not well suited for asynchronous logic design, performance is a limiting issue as well. The main problem is achievable resolution and counting speed of module "ref-time", which limits the number of counting events (cnt_{ref}) per bit-time and therefore the maximum achievable baudrate.

- The asynchronous TTP core runs at a speed directly proportional to the baudrate. Consequently, the achievable granularity (e.g., for macrotick generation, global time, hitting the correct transmission window) directly depends on the baudrate. Given the implementation from the previous chapter, the maximum achievable granularity is $1/(2f_{baud})$.

- The previous item is especially true for macrotick generation. It is not possible to achieve all configurable macrotick durations as integral multiple of the above mentioned granularity. Consequently, the macrotick must be chosen according to the baudrate and cannot be configured freely.

- Up to now, only Manchester encoding is supported. Other techniques (such as MII) cannot be used.

- TTP offers a lot of features for sophisticated fault-tolerant applications. The existing software tools allow a great variation of settings and optimizations. However, as the goal of project ARTS is not to rebuild an already existing controller in all its details for industrial use, these features have not been implemented. Anyway, no additional insights regarding the use of asynchronous logic in real-time systems can be expected.
- For the asynchronous controller to be able to integrate itself into the communication scheme, it is important that the remaining synchronous nodes are up and running, because it needs an already existing communication stream to adjust its internal timing.
- As the asynchronous controller has no separate crystal oscillator but relies on the correct timing of the remaining nodes, system properties like fault-tolerance, reliability, and synchronization precision can be expected to significantly degrade compared to a fully synchronous setup.

5.3 Experimental Results

In this section we experimentally evaluate some important properties of the asynchronous TTP controller when running in an otherwise synchronous system. We distinguish the following two cases:

- *Passive Communication:* The controller is not allowed to write to the bus, but only to monitor it. While having all *four* synchronous nodes running, we can check how good the asynchronous node would perform in case bus access was granted. Thus direct comparison with the non-modified cluster is possible.
- *Active Communication:* In this case *nodeD* is replaced by the asynchronous controller. During the communication slots of the remaining nodes it must be able to correctly receive all messages, extract the data, calculate timing correction terms, and determine the estimated transmission time of its own sending slot. During data transmission, the bitrate must be as stable as possible to avoid receive errors at the other nodes.

5.3.1 Passive Communication

As mentioned above, in passive communication mode the asynchronous TTP controller only has read access to the bus. The TTP cluster consists of four synchronous TTP nodes. The (fifth) asynchronous node is granted read access to the TTP bus, while write access is prohibited — thus it is impossible for our controller to jeopardize TTP communication or negatively influence global clock synchronization of the other nodes in any way. In order to check the robustness of the asynchronous controller, the cluster is configured with various different bitrates: Starting at the lowest possible speed ($275kb/s$) the bitrate is increased

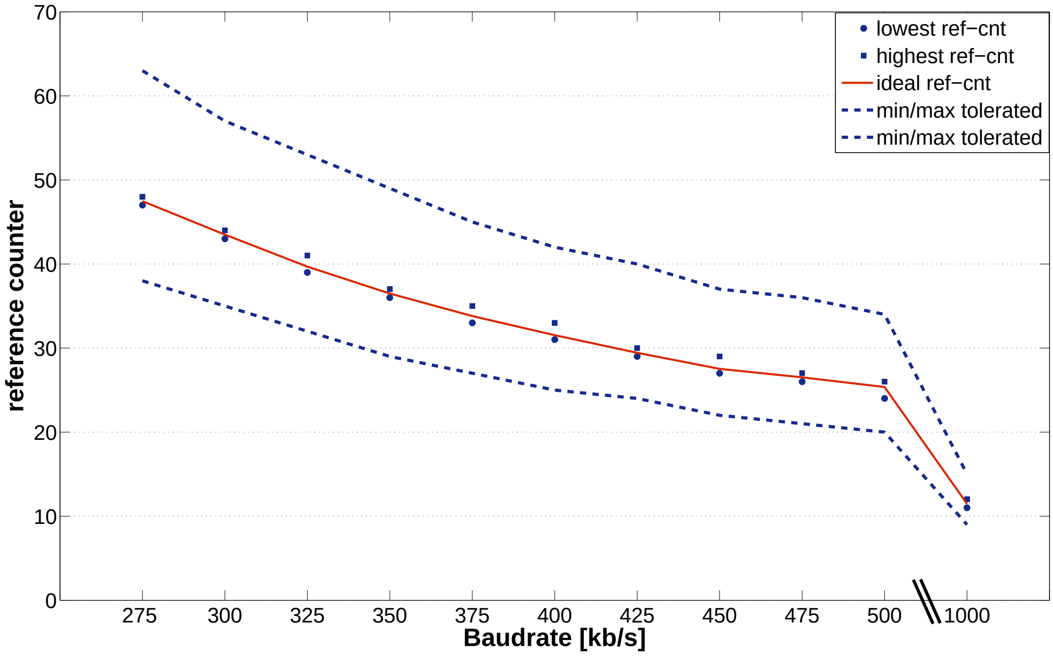


Figure 5.6: cnt_{ref} values for different baudrates with upper and lower bounds for correct message reception.

in steps of $25kb/s$ until $500kb/s$, which seems to be the highest reasonable communication speed given our setup and the FPGA's performance constraints. However, for reasons of completeness, also $1Mb/s$ is added to all measurements, because it is a commonly used speed. As mentioned before, we just modify the baudrate. All other parameters (i.e., slot duration, round duration, etc.) remain unchanged.

The first property we investigate are the measured reference values cnt_{ref} for the respective baudrates. Figure 5.6 summarizes the results, whereby the x-axis shows the baudrate[5] and the y-axis the respective reference counter values. The highest and lowest measured values of cnt_{ref} are indicated as squares and circles in the figure, respectively. The red line in between these upper and lower resynchronization bounds indicates the *ideal* value of cnt_{ref}, which can usually not be reached as it is not an integral value. It is calculated as average over all measured cnt_{ref} for each baudrate. Due to sampling and quantization errors, cnt_{ref} always "jitters" for at least 1 LSB (in case supply voltage and operating temperature are constant), sometimes even for 2 LSB — this issue is also evident in Figure 5.7 and is further discussed below. Figure 5.6 also shows the upper and lower bounds of cnt_{ref}-values (as dotted blue lines) for which receiving messages still works reliably (i.e., without CRC or synchronization errors). In other words, manually setting cnt_{ref} beyond these bounds will eventually lead to decoding errors of the bitstream. Not surprisingly, these upper and lower bounds directly follow from the ideal cnt_{ref}-value by multiplication with a factor of 4/3 or 4/5, respectively, which is also in accordance with

---

[5] Notice the kump of the scale as the bitrate changes from $500kb/s$ to $1Mb/s$.

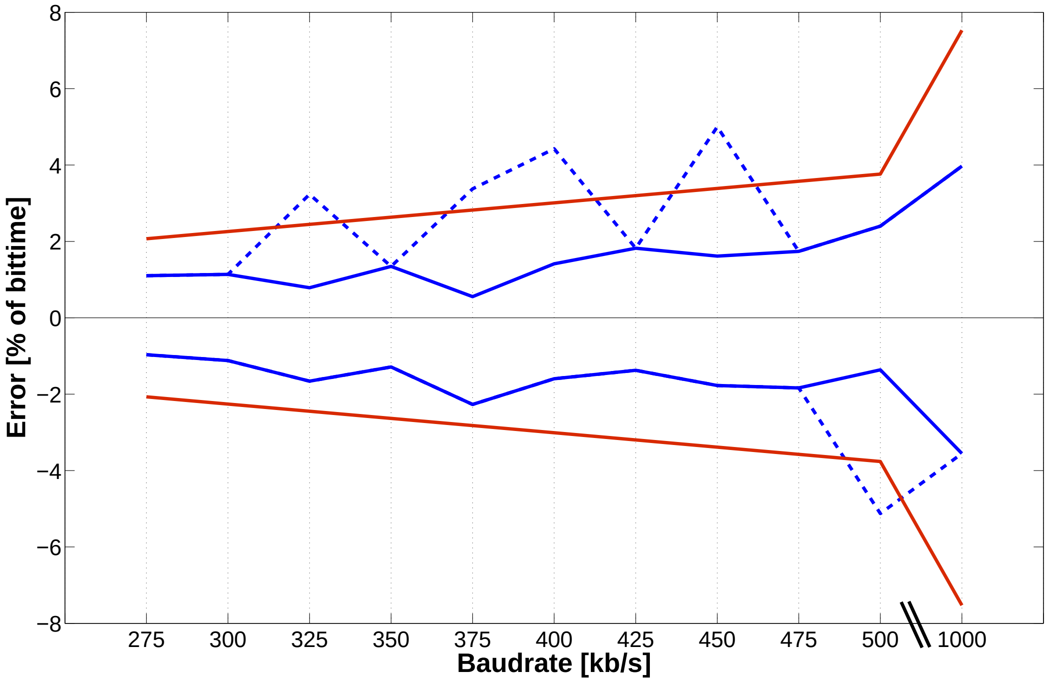


Figure 5.7: Relative error in percent of one bittime with 1 LSB upper/lower bounds.

the theoretical considerations made earlier (cf. Figure 4.3 on page 83): For a Manchester code the bus signal does not change its polarity for *at most* one bit-time τ_{bit}, which equals $4\tau_{gen}$ in the ideal case. After at most one bit-time, a transition is guaranteed to happen, thus resynchronization points are separated at most by τ_{bit}. For the lower bound of τ_{gen} it is important that no more than four transitions of `ref-time` occur during one τ_{bit}. Likewise, for the upper bound there must be *at least* three transitions per τ_{bit} to keep up synchrony. If these constraints are violated, the phase-relationship between sampling signal `ref-time` and the actual bus-signal is lost. The following two equations summarize this simple relationship (using the definition $4\tau_{gen,nom} = \tau_{bit}$, and f being the unknown factor such that $(f\,\tau_{gen,nom}) = \tau_{gen}$):

$$5\,(f\,\tau_{gen,nom}) > 4\tau_{gen,nom} \quad \Rightarrow \quad \frac{4}{5} < f \tag{5.1}$$

$$3\,(f\,\tau_{gen,nom}) < 4\tau_{gen,nom} \quad \Rightarrow \quad f < \frac{4}{3} \tag{5.2}$$

Another interesting aspect is presented in Figure 5.7. While the x-axis again represents the configured bitrates, the y-axis defines the relative error of signal `ref-time` in percent of one nominal bit-time. The red lines represent the relative errors of ±1 LSB: As the counting period τ_{step} is independent of the baudrate, the relative error introduced by one counting step linearly increases with rising communication speed. Obviously, this is also the case for the measured relative errors, because 1 LSB becomes more and more significant as the ideal cnt_{ref} decreases with speed. Resynchronization usually jitters for

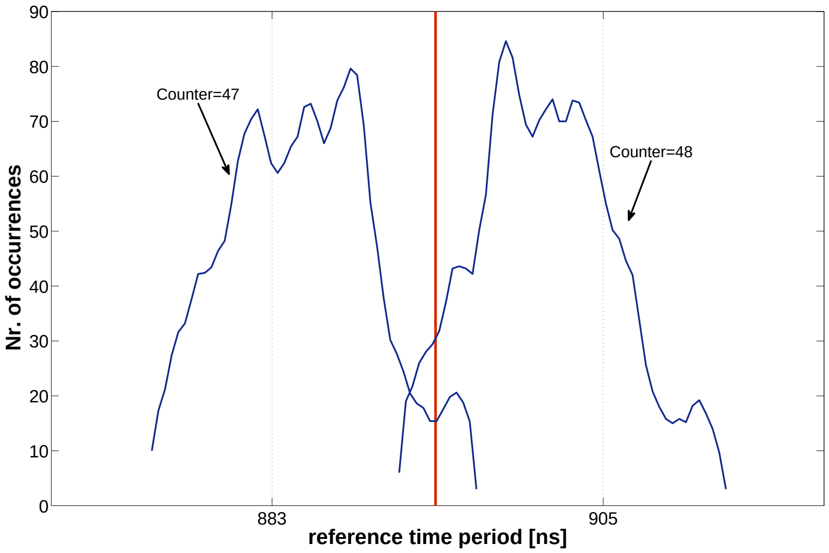


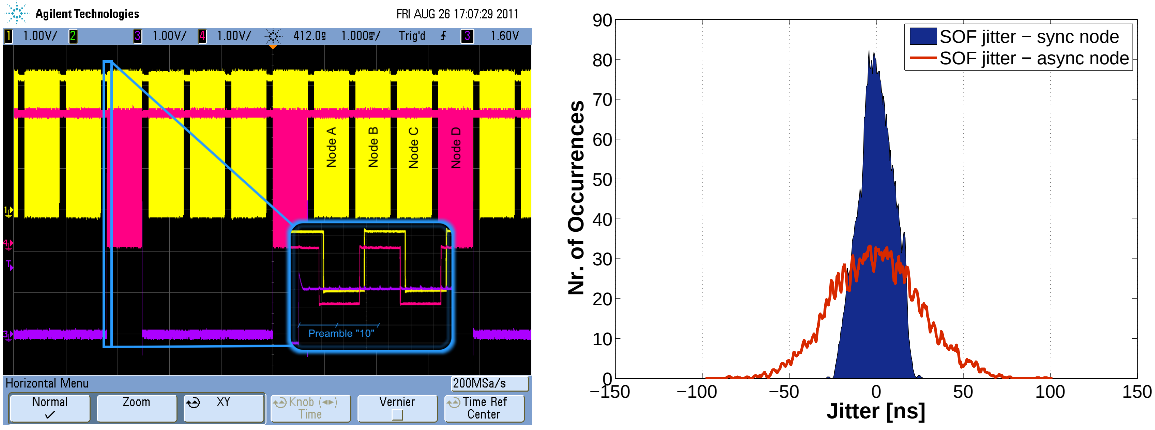
Figure 5.8: Exemplary jitter histogram for $275k$ for two cnt_{ref} values.

about 1 LSB, only in less than 5% of all cases even for 2 LSBs (given that environmental conditions are kept constant as good as possible). In the figure, the former case is indicated by the solid blue lines, and the rare latter case by the dashed blue lines. Notice that local voltage and temperature fluctuations inside the chip can in general not be controlled, therefore deviations larger than 2 LSBs can be expected to be observed eventually — this is, however, the very nature of our design and must not be considered as unintended behavior.

Finally Figure 5.8 shows the jitter histogram of signal `ref-time` for $275kb/s$: In accordance to the previous figures, cnt_{ref} jitters between the values 47 and 48. Analyzing the data reveals that the optimum value is approximately 47.467, which is indicated by the vertical red line. In the figure, the x-axis shows the period of `ref-time` in nanoseconds, while the left histogram corresponds to a $cnt_{ref} = 47$ and the right one to $cnt_{ref} = 48$, respectively. The widths of the histograms are caused by the random jitter components which accumulate during repeated execution of the counting procedure. One can see that, depending on the magnitude of the jitter, the periods of `ref-time` for consecutive counter values are not necessarily clearly separated from each other (they overlap slightly in the middle). Consequently, speaking of an *ideal* value of cnt_{ref} has only theoretical meaning to better understand the basic effects, but is only of limited interest for practical purposes when jitter must also be considered.

5.3.2 Active Communication

The system setup for active communication is quite similar to above. There is again the cluster with four synchronous nodes. In addition, however, the asynchronous controller takes the role of *nodeD*. Clearly, either the synchronous or the asynchronous *nodeD* are



(a) (b)

Figure 5.9: Scope screenshot of running TTP cluster (a) and jitter of SOF transition of synchronous and asynchronous node (b).

allowed to write to the bus, but never both of them. This setup has the advantage the we can compare the temporal behavior of the asynchronous node to the respective synchronous one in real-time. Which node is actually granted write access to the bus can easily be configured by setting hardware jumpers. Let us first consider Figure 5.9(a): The image presents a screenshot from an Agilent oscilloscope showing the actual bus line in yellow, the message sent in *slot 4* by the asynchronous controller in purple, and the `transmit-enable` signal in violet. The timebase is configured at $1ms$ per division in order to see two entire TTP rounds (i.e., one cluster cycle). The assignment from slot to node is shown in the image. According to the cluster's configuration, one slot has a duration of $800\mu s$. Given the bitrate[6] of $273,973\ b/s$ and a message length of 185 bits, we evaluate the transmission time per slot to be $675.25\mu s$ or 84.4%. Consequently, the idle time or inter-frame gap has a duration of $124.75\mu s$ or 15.6%. The magnified area in Figure 5.9(a) shows the start of transmission for *slot 4* of *nodeD*: The yellow channel represents the transitions generated by the synchronous node, which serves as a reference. The purple channel, on the other hand, shows the bit-pattern sent by the asynchronous controller. We can see that we hit the action window very well, as we are only approximately $500ns$ off the synchronous reference transition (each message starts with three preamble sequences, the first of which is highlighted in the figure). Notice, however, that this relationship changes with the bitrate, the actual speed of the hardware, the message and slot length, and the current value of cnt_{ref}. In general terms, we can achieve an error which is bounded by the following inequality, where $bits_{IFG}$ defines the number of bit-times in the inter-frame gap (which need not necessarily be an integral value):

$$|error| < (4 * \tau_{step,r}\, cnt_{ref} - \tau_{bit})\, bits_{IFG} = (4\tau_{gen} - \tau_{bit})\, bits_{IFG} \tag{5.3}$$

---

[6] The nominal bitrate is of course $275kb/s$, but as the cluster operates at $40MHz$ this baudrate cannot be precisely achieved. Instead of the correct prescaler of $145.\overline{45}$, the value is ceiled to the next integer by the provided software tools. This leads to an actual rate of $273,973\ b/s$.

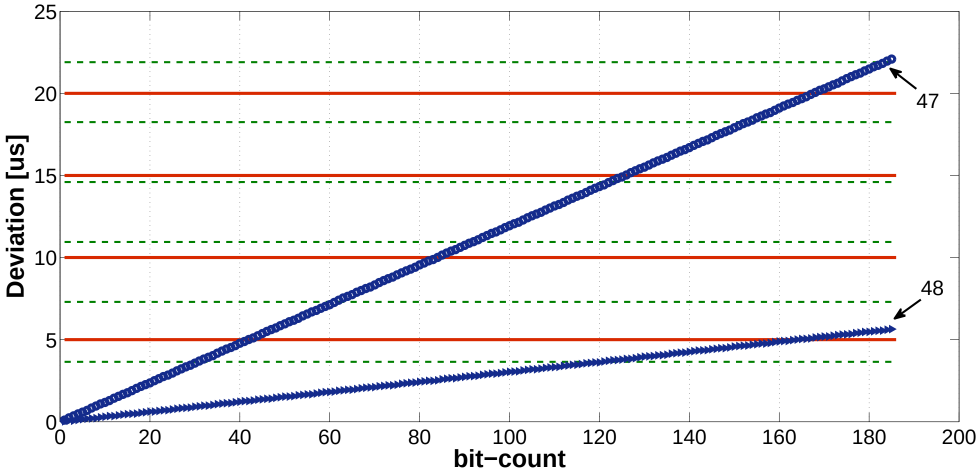


Figure 5.10: Absolute error made for $cnt_{ref} = 47$ (large slope) and $cnt_{ref} = 48$ (low slope) .

Notice that τ_{gen} is subject to jitter. For our given setup and hardware configuration we find those values to be $\tau_{step,r} \approx 18.11ns$, $\tau_{bit} = 3.65\mu s$, $bits_{IFG} \approx 34.2$, and $cnt_{ref} \in \{47, 48\}$. This leads to an error of no more than approximately $3.6\mu s$ and $1\mu s$ for the two possible values of cnt_{ref}, respectively.

Another interesting aspect is illustrated in Figure 5.9(b). The magnitude of the jitter of the very first transition of a message is compared for the synchronous *nodeD* and for the asynchronous target. In this setup, the scope is triggered at the last transition of *slot 3* by synchronous *nodeC*. Since the targets do not share a common oscillator, even the synchronous devices introduce some jitter to the start of message transmission. For reasons of comparability, the histograms in the figure are centered around zero — any static deviation has been removed to better compare the significance of jitter. The area shaded in dark blue jitters with a magnitude of at most $\pm 25ns$, which equals exactly ± 1 clock cycles at 40MHz. In contrast, the asynchronous node produces considerably more jitter. The maximum deviation is almost three times that of the synchronous reference. As before, the actual width of the jitter histogram depends on many different factors, thus the figure only represents a snapshot of the concrete setup we used for these measurements.

The central question which is still not answered is the following: Can the asynchronous controller actively take part in TTP communication? The short answer is *yes*, but with some major limitations, which leads to the long answer. It is indeed possible to replace the synchronous *nodeD* with its asynchronous pendent without jeopardizing the correct operation of the cluster. Our controller is not only able to correctly receive all messages sent on the bus, it also is capable of calculating the correct transmission time of its own slot (based on the asynchronous timestamps of the received messages) and actually send its message such that the other nodes accept it without any error (i.e., correct action window, correct C-state, correct timing, etc.). The problem is, however, that the synchronous nodes seem to be very sensitive with respect to the actual bit-timing.

For a given hardware operating speed (and at the baudrate of $275kb/s$, which is the lowest speed supported), there is only *one* value of cnt_{ref} which is actually "accepted" by the synchronous nodes for message transmission. This problem is also illustrated in Figure 5.10: The chart shows the accumulated error made for a given cnt_{ref} depending on the bit-count (x-axis) over an entire message consisting of 185 bits. The dashed green lines show the durations of single bit-times, while the red solid lines indicate macrotick durations. In other words, if for example the blue graph crosses the first green line it means that the accumulated error has reached the duration of an entire bit-time. It is obvious from the figure that $cnt_{ref} = 48$ performs much better than $cnt_{ref} = 47$. While the former accumulates an error of slightly more than one macrotick or 1.5 bit-times, the latter is off the nominal timing by more than four macroticks or six bit-times. It is thus no surprise that the receivers consider the latter messages as incorrect. Without having the source code and VLSI files of the synchronous TTP nodes available, it is difficult to say what exactly causes message transmission to fail. The following three options seem reasonable:

1. *Bittiming:* It is possible that the receivers at the synchronous nodes do not tolerate inaccurate bit-times, even though Manchester code would allow relatively large deviations due to the implicitly encoded clock signal.

2. *Relative deviation:* It is also possible that the accumulated error must stay within certain bounds relative to the baudrate (e.g., ±1 bittime). Even if single bits are allowed to significantly deviate from their nominal durations, the accumulated error over an entire message must be bounded. Too large deviations result in receive errors due to misaligned signal transitions.

3. *Absolute deviation:* A third possibility is that the actual time of the end of frame is compared to the expected time. This is in some sense similar to the previous option, as the absolute deviation of the last bit is too large to be still tolerated as correct (e.g., ±1 macrotick). However, this option allows the message itself to be received correctly, while the receive timestamp forces the message to be dropped as erroneous. Furthermore, the macrotick duration does not change with the baudrate.

Clearly, TTP assumes synchronous nodes only, thus it is convenient to allow only very tight margins. Too large deviations might indicate a broken PLL or crystal oscillator, thus the observed behavior helps to improve error detection and fault containment, but at the same time makes the integration of our asynchronous controller far more difficult.

5.3.3 Discussion

Regarding the software and hardware implementation of the asynchronous TTP controller, especially considering the resynchronization techniques presented in Chapter 4, there are some open questions to discuss.

Resynchronization. First of all, let us consider passive communication only. While reading messages from the bus, the resynchronization circuit is enabled. Consequently, each incoming message (to be more precise, each bit received) allows the asynchronous controller to readapt its internal timebase to the remaining TTP nodes. As long as it is not transmitting for itself, the exact counter value cnt_{ref} is not that important — given of course that it stays within the relatively wide theoretical bounds presented in Equations 5.1 and 5.2. In contrast, just before the controller's own sending slot starts, cnt_{ref} is set to the "correct" value by software. Hereby, the term "correct" means the reference value which leads to a timing that is accepted by the remaining nodes of the TTP cluster. As explained in Appendix A, the software is able to deactivate automatic resynchronization, as well as to read and write the reference counter value manually. We have already seen that the reference counter usually only deviates for approximately ±1 LSB from its ideal value, thus software correction is mostly restricted to a correction of just 1 LSB to allow for correct message transmission.

Correct counter value. From the previous paragraph we know that software sets cnt_{ref} to the "correct" value just before the controller's own sending slot starts. Obviously, this value must be evaluated beforehand. To this end, the software stack evaluates the measured cnt_{ref} while receiving messages. This value is a good starting point, as the actual value we need for sending messages is somewhere around $cnt_{ref} \pm 1$, so usually just three adjacent reference values are potential candidates. The application then tries to send messages with one of these values, and evaluates the membership vector received from the other nodes during the next TDMA round. As long as the remaining communication participants do not mark the asynchronous node as "correct" in the vector, the current reference value is regarded as incorrect, and the next one is used in the following communication round. Once active message transmission has been established successfully (as indicated by the other nodes' membership vectors), the reference value is saved and also used in the future. While this algorithm is quite straight-forward and simple, its obvious drawbacks are as follows: (i) First of all, finding the correct value in the first place might take a couple of rounds. During these rounds, the transmitted messages are considered faulty by the other nodes. Consequently, integration of the node into the TTP communication scheme takes longer than for synchronous nodes. (ii) When the operating conditions of the asynchronous controller change (e.g., due to increasing temperature), the timing characteristics change and the "correct" counter value must be adapted accordingly. During this phase, however, active message transmission is likely to fail until the new correct cnt_{ref} is found.

Lower baudrate. Based on the given implementation and the used FPGA devices, one might ask which baudrate would be suitable to allow active communication without the need for finding a "correct" reference value by software whenever the operating conditions change. In other words, how low should the baudrate be in order to use the reference values found by the automatic resynchronization feature during message reception? One precondition is that at least $cnt_{ref} \pm 1$ reference values (i.e., at least three counter values) are accepted as "correct" by the system. Only this way, transmitted messages are not considered invalid even if cnt_{ref} is slightly off its "optimum". According to the three options listed on page 121 as possible causes of the message acceptance problems at the synchronous nodes, the following three cases can be distinguished:

1. *Bittiming:* If inaccurate bittimes are not accepted by the synchronous receivers, lowering the baudrate might indeed improve the situation. Lower bitrates result in higher reference counter values, which means that one LSB has less significance with respect to the duration of one bit. Since the current configuration with $275kbit/s$ allows for exactly one "correct" cnt_{ref} only, a rough estimate would be to use just a third of this baudrate, effectively tripling cnt_{ref} and possibly allowing $cnt_{ref} \pm 1$ to be acceptable values as well. A verification of this, however, is not possible, since the synchronous TTP nodes cannot be configured for frequencies below $275kb/s$. Notice that this is an implementation-related issue, not a conceptual one.

2. *Relative deviation:* According to Figure 5.10 it might also be possible that the synchronous receivers allow a deviation of the entire message length of up to two bittimes from the nominal value. Similar to above, lower baudrates would indeed improve message transmission capability because the error introduced by a deviation of one LSB is always constant. Having an average counting speed of $\tau_{step} = 20ns$ and a message length of 185 bits, the maximum error introduced by one LSB during an entire message is $20ns * 4 * 185 \approx 15\mu s$. Consequently, the acceptance window for tolerating ±1 LSB would be $\pm 1.5 * 15\mu s = \pm 22.5\mu s$. Therefore, the resulting baudrate must fulfil the condition that two bittimes are at least $22.5\mu s$ long, which equals approximately $88.9kbit/s$. Similar to above, this is about a third of the original baudrate.

3. *Absolute deviation:* If the maximum acceptable deviation is bounded by absolute restrictions (e.g., ±1 macrotick), then lowering the baudrate has no effect at all. The point here is, as already mentioned above, that the deviation inflicted by one LSB is constant for a given message length. Lowering the baudrate changes the relative error with respect to message transmission duration, but the absolute error remains unchanged. The only ways to counteract would be to change the duration of the macrotick or to speed up hardware in order to lower τ_{step}. However, it is not possible to configure macrotick durations of arbitrary lengths, and accelerating hardware by a factor of three is not feasible as well.

Higher baudrate. In contrast to the previous paragraph, what must be done in order to allow for higher baudrates? The answer to that is comparatively simple, as increasing the counting speed τ_{step} would do the trick. A higher sampling rate not only improves accuracy, it also reduces the quantization error and the relative error done by one LSB of the counter. Increasing the operating speed would also solve case (3) from above, because the accumulated absolute error inflicted by one LSB during an entire message would decrease accordingly. For our particular case, tripling the execution speed (from $20ns$ to $\approx 7ns$ per step) would probably be sufficient to allow stable operation even for active communication. However, neither changing the device type to a more sophisticated high-end chip, nor further optimizing the design will gain a factor of three in performance. Having an ASIC implementation, on the other hand, might achieve the desired performance gain. It might be necessary to change the design style from LEDR to NCL, as the latter is better suited for combinatorial circuits. The respective dual-rail gates are mostly based on Muller C-gates, for which highly optimized transistor level implementations exist[7].

5.4 Chapter Notes

According Figure 5.1 on page 106 it is obvious that the different modules cannot simply be connected together. At the respective interfaces, the interfacing and conversion circuits from Chapter 2 are used extensively. For instance, the four control signals `ctrl` as well as the two interrupt lines `rx-irq` and `tx-irq` can properly be converted to the respective time and value domain with the “free conversion circuit” presented in Section 2.3. This is also valid for sampling the bus-signal. While module “ref-time” always runs freely (i.e., only non-blocking synchronization techniques are applied), the “TTP core” itself runs in blocking mode — after each execution step, the module waits for the next step indicated by signal `ref-time`, thus necessitating configuration 4 from the list on page 43. Another issue concerns the dual-ported RAM, which is not available as fully asynchronous RAM in Cyclone II devices. Consequently, the respective interface on the asynchronous side needs to accommodate for this circumstance and also provide a feasible interface (at least for the `write-enable` and the `memory-clock` signals, while `data` and `address` can be bundled).

The main part of this chapter focused on the integration of the asynchronous time reference generator into the asynchronous TTP module and the combination with the SPEAR processor core. While the actual bus communication (i.e., sending and receiving messages, Manchester encoding, etc.) and basic TTP services (i.e., message timestamps, etc.) are entirely realized as asynchronous hardware, some non-time-critical services (e.g., message assembly, CRC check, etc.) are simply realized as software stack running on an ordinary synchronous microprocessor. Communication between the synchronous and the asynchronous parts is implemented by means of a dual-ported RAM with two interrupt signals (`tx-irq` and `rx-irq`) for synchronization and concurrent access avoidance. It has further been shown that the software stack does not perform any time-critical tasks, but is only needed for handling the TTP application, assembling and disassembling the TTP messages, and providing debug information and monitoring data to the personal computer for offline evaluation. After discussing limitations and restrictions, experimental results with the TTP cluster revealed some interesting insights. As it turns out, passive communication (i.e., only monitoring the bus) allows a relatively wide range of bitrates (up to $1 Mb/s$) without loosing synchrony to the remaining system. On the other hand,

---

[7] We tried to implement some parts of our design using the NCL design style. While area consumption of the FPGA device was considerably less compared to LEDR, no improvement in execution speed could be observed. The problem is that in FPGAs, even Muller C-gates must be constructed with LUTs and local feedbacks, which slows down circuit speed.

actively sending messages is more critical, as the synchronous nodes seem to expect relatively accurate message transmission times. Regarding these issues, several possible causes have been discussed. For instance, an important question is whether the comparatively high baudrate is the main cause of the observed restrictions regarding the reference value cnt_{ref}. If it were possible to lower communication speed, the relative deviation of one LSB of cnt_{ref} compared to one bit-time would be less significant. Consequently it could be possible that the observed limitations would not be present any more for lower baudrates. Unfortunately, it is neither possible to further lower communication speed (not supported by the current synchronous controllers), nor to improve hardware execution speed by a factor of approximately three. Without further investigation it is therefore difficult to make any definite conclusions on this topic. Nevertheless, we have shown that TTP communication is — besides the mentioned limitations — in principle possible even with asynchronous hardware.

Chapter 6

Conclusion and Outlook

> *I don't know why I did it, I don't know why I enjoyed it, and I don't know why I will do it again.*
>
> Matt Groening

In this work we extensively studied the temporal behavior and predictability of asynchronous logic. To this end, the adaption of existing timing models towards a more suitable model for our purposes was necessary. Based on jitter definitions for the synchronous world, respective adaptions have been performed and suitable jitter characteristics have been identified especially for asynchronous circuits. Based on ordinary static timing analysis and more complex statistical timing analysis, a timing model has been elaborated which accounts for the special properties of asynchronous logic: Not only are voltage fluctuations, temperature drift, and process variations modeled accordingly, but also random jitter is taken into account for both gate- and interconnect delays. Using case studies with elementary circuits we have shown the practical usefulness of our model. It has been demonstrated that — in accordance with simulations based on the theoretical model — even strongly indicating circuits such as LEDR designs show a significant amount of data-dependent jitter. In addition, the combined temperature and voltage measurements revealed an interesting observation: As the core supply voltage decreases, increasing temperature seems to lead to a significant speedup of a circuit's performance. The expected behavior (i.e., higher temperature slows down the execution speed) is only observed around the nominal supply voltage (or for higher voltages). A possible explanation was found as the increasing temperature lowers the transistors' threshold voltages and may consequently increase performance for low supply voltages.

On the practical side we have elaborated a way of synchronizing asynchronous logic to some deterministic external events (in our case the predefined TTP communication stream). The basic idea of using a free-running counter to measure the duration of an external event has been implemented in different ways. A practical examination revealed that linear feedback shift registers are well suited for the given task as they provide a deterministic order of states and have extremely low area requirements as well as good performance. Each transition on the TTP bus is used as resynchronization point for adapting the internal reference counter and adjust the timing accordingly. Consequently, changing hardware execution times caused by PVT variations are compensated automatically over time. This has been shown in various measurements with changing temperature, core supply voltage, and even different devices of the same type. As main result of these investigations one can summarize that the proposed design is capable of tolerating and compensating all the induced variations. However, process variations tend to *severely* influence the actual jitter characteristics. It is therefore difficult to make any predictions on the temporal behavior for an entire device family by just considering one “representative” device.

Finally we have seen that both active and passive communication using our asynchronous TTP controller basically work. While reading messages is relatively robust and works reliably for a comparatively high communication speed, active communication is more error prone due to the fact that the synchronous nodes do not tolerate too large deviations of the nominal timing. Nevertheless, this proves that the presented compensation strategy in priciple works. Throughout this work we have shown that asynchronous logic shows substantial variations in hardware execution time. Consequently, without any additional measures, asynchronous circuits are basically unsuited for real-time applications. However, we have further shown that it is indeed possible to “hide” these timing uncertainties from higher levels using the presented calibration techniques. As it turns out, it is possible to devise appropriate and sufficiently effective countermeasures (with respect to timing variations) on top of the known asynchronous design styles after all.

For future work, there are still some interesting open questions to answer, and some implementation tasks yet unattended:

- *Host interface:* Implementing the entire host-interface (CNI) would allow to be fully compatible with the existing hardware interfaces and software tools, thereby eliminating the manual steps during cluster configuration. Downloading an application could then also be performed using the Monitoring Node rather than the local serial interface. Furthermore, having an external host-CPU has the advantage of offering more performance. In the current implementation, the complexity of the TTP application is limited by the little time available besides handling the actual TTP stack.

- *Macrotick generation:* A separate hardware module for generating the macrotick would be preferable. Now, the macrotick of the TTP cluster must be configured depending on the baudrate in order to allow the asynchronous node to generate it correctly. A separate module would increase flexibility and would allow the macrotick to be freely configurable.

- *Asynchronous Cluster:* Investigating the behavior of a TTP cluster containing more than just one asynchronous node might be interesting as well. Theoretically, one synchronous node could be enough to keep the timing deviations bounded. However, synchronization precision and fault tolerance clearly decrease for such a configuration. In case not a single synchronous node is present, timing can be expected to drift "indefinitely", as each node introduces a little error and no reference is present to reset or bound that error. Furthermore, investigating algorithms to bound the maximum deviations (e.g., using temperature and supply voltage sensors to estimate the operating speed) would be interesting as well.

- *Software execution jitter versus hardware jitter:* A detailed comparison between software and hardware execution jitter would be interesting. As the commercial TTP controller uses a software stack for executing the TTP protocol, one can expect a substantial amount of software jitter (because different software-branches are executed, which might have different complexity and thus need different amounts of time to finish).
- *Receive sensitivity:* We have seen that the synchronous nodes are very sensible to inaccurate timing regarding the messages sent by the asynchronous controller. It would be of central interest to further investigate the exact reason for this behavior. Without knowing the root cause of the problem, finding an appropriate solution is difficult.
- *Synchronization algorithm:* The currently used simple synchronization algorithm for finding the "correct" reference value to allow for active message transmission could be improved. A continuous statistical analysis of measured reference counter values during message reception might allow to detect immanent changes in operating speed even before message transmission actually fails. It might be possible to enable sending messages without interruptions by corrupt counter values.
- *Synchronization accuracy and fault-tolerance:* As the asynchronous controller uses solely the messages sent by other nodes to derive its local timebase, it does not contribute to increase clock synchronization precision. Consequently, it would be interesting to investigate the exact impact on timing precision (as well as fault-tolerance, which is partly correlated to a consistent and precise notion of time).

Appendix A

Software Implementation

A.1 Register Definitions

This section provides an overview to the register definitions of the SPEAR processor's *TTP extension module* which has been developed in the course of this work (refer to Figure 5.1 on page 106). The CPU can access all necessary control and debug features of the asynchronous TTP controller via memory mapped register access. Table A.1 shows an overview to all available registers. In the table, column "Flags" defines the address offset in bytes, as well as the access rights to the respective registers: R (read only), W (write only), and RW for both read and write access. The following subsection describe all those registers and the respective bit-definitions of the TTP extension module.

Register	**Flags**	**Description**
`status-reg`	+0 (R)	SPEAR-specific status register
`config-reg`	+2 (RW)	SPEAR-specific configuration register
`timer-reg`	+4 (R)	Synchronous reference timer for debug
`control-reg`	+8 (RW)	TTP control register
`lfsr-reg`	+12 (R)	TTP LFSR value register
`asyn-timer-reg`	+16 (R)	Asynchronous TTP reference timer
`timestamp-reg`	+20 (R)	Synchronous receive timestamp reference for debug

Table A.1: Register overview for TTP extension module.

A.1.1 Status Register

The status register is combination from 8 SPEAR-specific status bits, and additional 8 module-specific status bits. This register is read-only.

15	14	13	12	11	10	9	8
–	–	–	–	–	–	–	–

7	6	5	4	3	2	1	0
`LOOR`	`RESH`	`RESL`	`FSS`	`BUSY`	`ERR`	`RDY`	`INT`

Table A.2: Bit-definitions of `status-reg`.

- `INT`: If 1, an interrupt request is pending. The respective IRQ line is active.
- `RDY`: Not used. Bit is always read as 1.
- `ERR`: Not used. Bit is always read as 0.
- `BUSY`: Not used. Bit is always read as 0.
- `FSS`: Not used. Bit is always read as 0.
- `RESL`: Not used. Bit is always read as 0.
- `RESH`: Not used. Bit is always read as 0.
- `LOOR`: Not used. Returns the value last written in `config-reg.loow`.

A.1.2 Configuration Register

The configuration register is combination from 8 SPEAR-specific config bits, and additional 8 module-specific config bits. This register can only be written. Reading from this register always returns the module's version (upper 8 bits) and the module's ID (lower 8 bits).

15	14	13	12	11	10	9	8
–	–	–	–	–	–	–	–

7	6	5	4	3	2	1	0
`LOOW`	`RESH`	`RESL`	`EFSS`	`OUTD`	`SRES`	`ID`	`INTA`

Table A.3: Bit-definitions of `config-reg`.

- `INTA`: If an interrupt request is pending, and the corresponding interrupt service routine is executed, a 1 must be written to this bit in order to reset the interrupt request and clear the `status-reg.int` flag.
- `ID`: Not used. Write access has no effect.
- `SRES`: Not used. Write access has no effect.

- `OUTD`: Not used. Write access has no effect.
- `EFSS`: Not used. Write access has no effect.
- `RESL`: Not used. Write access has no effect.
- `RESH`: Not used. Write access has no effect.
- `LOOW`: Not used. Sets the value in `config-reg.loor` accordingly.

A.1.3 Timer Register

This register is a simple 32-bit (unsigned) counter register, which is clocked by the CPU's main clock source ($40MHz$). This is an ordinary synchronous counter which can be used for debug and comparison purposes. Notice that this timer is *not* used to perform any timing critical operations, or trigger any timing critical tasks. The register is read only.

31	30	29	28	27	26	25	24
`timer[31..24]`							

23	22	21	20	19	18	17	16
`timer[23..16]`							

15	14	13	12	11	10	9	8
`timer[15..8]`							

7	6	5	4	3	2	1	0
`timer[7..0]`							

Table A.4: Bit-definitions of `timer-reg`.

- `timer`: 32-bit wide synchronous timer register for debug and reference.

A.1.4 Control Register

The TTP control register allow access to the most important features of the TTP controller. Is provides means for triggering transmit interrupts and allow to read important debug and status information. This register can be read and written.

31	30	29	28	27	26	25	24
`LED2`	`LED1`	`LED0`	`MASK`	`FREEZE`	`INC`	`DEC`	-

23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-

15	14	13	12	11	10	9	8
-	-	-	`LFSR-init[10..7]`				

7	6	5	4	3	2	1	0
`LFSR-init[6..0]`						`PHASE`	`RST`

Table A.5: Bit-definitions of `control-reg`.

- `RST`: If 0, the entire asynchronous controller is in reset mode. All registers and counters are reset, and the controller is not running. the controller must be in reset state when fields `phase` and `LFSR-init` are accessed.
- `PHASE`: Defines the initial phase of the LFSR init value (see also `LFSR-init`. Usually, this bit needs not to be changed and is 0, as the asynchronous hardware is in phase 0 after reset.
- `LFSR-init`: The initial value of the reference value cnt_{ref} must be set within certain bound (which depend on the desired baudrate). As it is inconvenient to recompile the VLSI design every time the baudrate changes, this bitfield can be used to set the initial value of cnt_{ref}. Bit `rst` must be active whenever this field is accessed. Notice that this field defines the actual LFSR value, rather than the counter-equivalent (i.d., the index). Use function `ttp_getLFSR()` to get the LFSR value from a given index.
- `DEC`: Each transition of this bit causes the reference counter cnt_{ref} to decrease by one. No range checks are performed. Bit `freeze` is ignored, which means that cnt_{ref} is decremented independently from the status of `freeze`.
- `INC`: Each transition of this bit causes the reference counter cnt_{ref} to increase by one. No range checks are performed. Bit `freeze` is ignored, which means that cnt_{ref} is incremented independently from the status of `freeze`.
- `FREEZE`: If 1, the reference value cnt_{ref} is frozen and does not change (even if bus transitions for resynchronization are received). This is useful to avoid resynchronization, but at the same time allow read-access to the bus (affects only module “ref-time”, not the “TTP core” itself).
- `MASK`: If 1, the “ref-time” module is not connected to the physical bus any more. It always receives the bus idle state. This is useful to deliberately mask out entire (or parts of) messages and to avoid the resynchronization process.

- `LED0`: If 1, switches on debug LED 0.
- `LED1`: If 1, switches on debug LED 1.
- `LED2`: If 1, switches on debug LED 2.

A.1.5 LFSR Register

This register can be used to retrieve the current value of cnt_{ref}. It is important to notice that this value is *only* updated when a receive interrupt `rx-irq` occurs. The reason is that the asynchronous logic runs independently from the (synchronous) extension module. After a message was received, the bus is guaranteed to be idle for some time. During this time, however, cnt_{ref} cannot change because there is no bus-activity which can be used for resynchronization — it is thus save to update the respective register only upon a transition of `rx-irq`. This register can only be read.

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-

23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-

15	14	13	12	11	10	9	8
-	-	-	-	-	`LFSR[10..8]`		

7	6	5	4	3	2	1	0
`LFSR[7..0]`							

Table A.6: Bit-definitions of `lfsr-reg`.

- `LFSR`: Returns the current 11-bit value of the LFSR reference value (cnt_{ref}). This is the actual LFSR value rather than the corresponding counter-equivalent (LFSR index). The PC tool provides a function to get the index for a given LFSR value.

A.1.6 Asynchronous Timer Register

This register returns the timer value generated by the asynchronous module. this timer reflects the number of events generated by module "ref-time", or half-bit times in other words. As the extension module and the asynchronous logic run independently from each other, this value is only updated at the falling edge of signal `ref-time`. It is guaranteed that the asynchronous timer is stable for the entire low-period of signal `ref-time`, thus it is save to update the corresponding synchronous register at this point in time. The asynchronous timer reflects the elapsed time of the asynchronous TTP module and forms the basis for all time-critical calculations.

31	30	29	28	27	26	25	24
`asyn-timer[31..24]`							

23	22	21	20	19	18	17	16
`asyn-timer[23..16]`							

15	14	13	12	11	10	9	8
`asyn-timer[15..8]`							

7	6	5	4	3	2	1	0
`asyn-timer[7..0]`							

Table A.7: Bit-definitions of `asyn-timer-reg`.

- `asyn-timer`: 32-bit wide asynchronously generated timer value. It reflects the time as seen by the asynchronous controller and forms the basis for all time critical calculations.

A.1.7 Timestamp Register

This register represents the timestamp of the reception of the SOF field of a TTP message. The timestamp is generated synchronously (i.e., by the extension module rather than the asynchronous controller) and serves as references for evaluation of the accuracy of the asynchronous design. Each message received is associated with an asynchronous timestamp as well (cf. Section A.3. Having a synchronous reference is a simple method for measuring the achieved accuracy.

31	30	29	28	27	26	25	24
`timestamp[31..24]`							

23	22	21	20	19	18	17	16
`timestamp[23..16]`							

15	14	13	12	11	10	9	8
`timestamp[15..8]`							

7	6	5	4	3	2	1	0
`timestamp[7..0]`							

Table A.8: Bit-definitions of `timestamp-reg`.

- `timestamp`: 32-bit wide synchronously generated timestamp of the SOF field of a TTP message. Please refer to Chapter 5 for a detailed explanation of the SOF field.

A.2 Message Descriptor List

As mentioned in Chapter 5, the structure of the MEDL (message descriptor list) differs from the reference design of the synchronous nodes. For reasons of simplicity, we only included those fields which we actually need for our purposes. The severely simplified data structure is presented below.

```
struct schedule_t {
    uint16_t slots;
    uint16_t rounds;
    uint32_t schedule_id_high;
    uint32_t schedule_id_low;
    uint32_t crc_init;
    uint16_t slot_times_us[SCHEDULE_SLOTS];
    uint16_t slot_times_mt[SCHEDULE_SLOTS];
    uint16_t slot_times_ticks[SCHEDULE_SLOTS];
    uint16_t slot_times_mt_acum[SCHEDULE_SLOTS];
    uint16_t slot_times_us_acum[SCHEDULE_SLOTS];
    uint16_t message_lengths[SCHEDULE_SLOTS];
    uint16_t message_types[SCHEDULE_SLOTS];
};
```

In this listing, `SCHEDULE_SLOTS` defines the number of TDMA slots (8 in our case). There is a 64 bit long schedule-ID, which is uniquely generated for each specific schedule and stored in fields `schedule_id_high` and `schedule_id_low`. Out of this ID, the initial value for CRC calculation can be retrieved and is stored in `crc_init`. The remaining fields starting with prefix `slot_time_*` define the lengths of the respective slots in microseconds, macrotick-counts, asynchronous execution ticks, accumulated macrotick-counts, and accumulated microseconds, respectively. Finally, `message_length` and `message_types` define the message length in bytes, and the message types (i.e., X-frame, I-frame, N-frame, etc.), respectively. some of the fields are redundant and calculated at runtime for performance optimization.

A.3 Message Data Structure

The data structure which represents a TTP message (whether received or to be transmitted) is shown in the following code listing. It can be seen that each message has an associated synchronous timestamp `syn_timestamp_measured` (this field is only important for *received* messages), and an asynchronous timestamp `timestamp`. The latter defines the point in time when either a message has been received or shall be transmitted. The actual message is stored in fields `header` and `data`. The former can further be divided into the single header fields.

```
typedef struct {
    uint32_t syn_timestamp_measured;
    uint32_t timestamp;
    union {
        struct {
            uint32_t length :  8;
            uint32_t header :  4;
            uint32_t unused : 20;
        } fields;
        uint32_t all;
    } header;
    uint8_t data[TTP_MSG_SIZE];
} ttp_msg_t;
```

A.4 TTP Application

The next below code listing shows the source code of the very simple TTP application we used for the experiments. Each node just transmits one 16-bit value which is calculated as shown below. These values are named stored in `ttp_system.counterA` to `ttp_system.counterD`, for nodes 0 to 3, respectively. Function `ttp_set_LED()` and `ttp_clear_LED()` are used to turn on some debug LEDs for visual feedback. Notice that all nodes execute a very similar application: The lowest 4 bits are a node-specific counter, which is incremented once in each round. The second nibble is the last received value (lowest 4 bits) of the asynchronous controller, i.e., *nodeD* or `counterD`. While the most significant nibble holds a unique ID for each node (one-hot encoding), the third nibble contains the sum of the other three counter values. It is therefore easily possible to identify the origin of each message (using the ID), and check whether all nodes have correctly received the messages of all other nodes, especially the ones sent by the asynchronous controller. The code listing below shows the application for *nodeD* with associated value `counterD`.

```
(ttp_system.counterD & 0x0008) ?
    (ttp_set_LED(TTP_LED2)) : (ttp_clear_LED(TTP_LED2));
old_cnt = ttp_system.counterD & 0x000F;
counter = ((ttp_system.counterD & 0x000F) + 1) & 0x000F;
counter |= 0x8000;
counter |= ((old_cnt << 4) & 0x00F0);
ttp_system.counterD = counter;
```

A.5 Index to LFSR conversion

A function referenced above is shown in the next code listing. This function takes an LFSR index `_val` as input and returns the corresponding LFSR value. In other words, `_val` defines the number of shift-and-XOR operations performed — starting from the initial value 0 — in order to obtain the respective LFSR value. This function can easily be modified to also perform the opposite task: Return an index for a given LFSR value. This is, however, not shown below.

```c
uint16_t ttp_getLFSR(uint16_t _val) {
    // 11-bit: Poly: x^11 + x^9 + 1
    #define POLY_BIT0 9
    #define POLY_BIT1 11
    #define POLY_LEN POLY_BIT1
    #define POLY_MASK (1 << (POLY_BIT0-1))
    uint16_t lfsr = 0;
    uint16_t xor_mask, bit;
    while ((_val--) > 0) {
        bit = (lfsr & 0x0001);
        xor_mask = (uint16_t)(-1 + bit) & POLY_MASK;
        lfsr = (lfsr >> 1);
        lfsr ^= xor_mask;
        if (bit) {
            lfsr |= (1 << (POLY_LEN-1));
        } else {
            lfsr &= ~(1 << (POLY_LEN-1));
        }
    }
    return lfsr;
}
```

Bibliography

[1] D. Allan. Time and frequency (time-domain) characterization, estimation, and prediction of precision clocks and oscillators. *IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control,*, 34(6):647 – 654, Nov. 1987.

[2] D. W. Allan, N. Ashby, and C. C. Hodge. The Science of Timekeeping. Application Note 1289. http://www.allanstime.com/Publications/DWA/Science_Timekeeping/ TheScienceOfTimekeeping.pdf, 1997.

[3] F. Anceau. A synchronous approach for clocking vlsi systems. *IEEE Journal of Solid-State Circuits*, 17(1):51 – 56, feb 1982.

[4] K. Arvind. Probabilistic clock synchronization in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 5(5):474 –487, may 1994.

[5] F. Bala and T. Nandy. Programmable high frequency RC oscillator. In *18th International Conference on VLSI Design.*, pages 511–515, Jan. 2005.

[6] S. Beer, R. Ginosar, M. Priel, R. Dobkin, and A. Kolodny. The devolution of synchronizers. In *IEEE Symposium on Asynchronous Circuits and Systems (ASYNC), 2010.*, pages 94 – 103, May 2010.

[7] D. Blaauw, K. Chopra, A. Srivastava, and L. Scheffer. Statistical timing analysis: From basic principles to state of the art. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(4):589 –607, april 2008.

[8] I. Blunno, J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C. Sotiriou. Handshake protocols for de-synchronization. pages 149 – 158, apr. 2004.

[9] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De. Parameter variations and impact on circuits and microarchitecture. In *Design Automation Conference, 2003. Proceedings*, pages 338 – 342, 2003.

[10] K. Bowman, S. Duvall, and J. Meindl. Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration. *IEEE Journal of Solid-State Circuits,*, 37(2):183 –190, Feb. 2002.

[11] B. Butka and R. Morley. Simultaneous switching noise and safety critical airborne hardware. In *IEEE Southeastcon, 2009. SOUTHEASTCON '09.*, pages 439–442, March 2009.

[12] D. Caucheteux, E. Beigne, M. Renaudin, and E. Crochon. Asyncrfid: fully asynchronous contactless systems, providing high data rates, low power and dynamic adaptation. pages 10 pp. –97, mar. 2006.

[13] C.-M. Chang, S.-H. Huang, Y.-K. Ho, J.-Z. Lin, H.-P. Wang, and Y.-S. Lu. Type-matching clock tree for zero skew clock gating. In *45th ACM/IEEE Design Automation Conference, 2008. DAC 2008.*, pages 714 –719, 8-13 2008.

[14] D. M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems.* PhD thesis, Stanford University, Oct. 1984.

[15] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, A. Yakovlev, and N. R. England. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, 80:315–325, 1997.

[16] U. Cummings. Pivotpoint: clockless crossbar switch for high-performance embedded systems. *Micro, IEEE*, 24(2):48 – 59, mar-apr 2004.

[17] S. Dasgupta and A. Yakovlev. Comparative analysis of gals clocking schemes. *Computers Digital Techniques, IET*, 1(2):59 –69, 2007.

[18] M. E. Dean, T. E. Williams, and D. L. Dill. Efficient Self-Timing with Level-Encoded 2-Phase Dual-Rail (LEDR). In *Proceedings of the 1991 University of California/Santa Cruz conference on Advanced research in VLSI*, pages 55–70, Cambridge, MA, USA, 1991. MIT Press.

[19] M. Delvai. *Design of an Asynchronous Processor Based on Code Alternation Logic - Treatment of Non-Linear Data Paths.* PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Dec. 2004.

[20] M. Delvai and A. Steininger. Solving the fundamental problem of digital design - a systematic review of design methods. In *9th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006.*, pages 131 –138, 0-0 2006.

[21] J. Ebergen, S. Fairbanks, and I. Sutherland. Predicting performance of micropipelines using charlie diagrams. pages 238 –246, mar-2 apr 1998.

[22] T. Edwards. The status of asynchronous design in industry. Technical report, Information Society Technologies (IST) Programme, 2004. http://www.scism.lsbu.ac.uk/ccsv/ACiD-WG/AsyncIndustryStatus.pdf.

[23] S. Fairbanks and S. Moore. Analog micropipeline rings for high precision timing. In *10th International Symposium on Asynchronous Circuits and Systems, 2004.*, pages 41–50, April 2004.

[24] S. Fairbanks and S. Moore. Self-timed circuitry for global clocking. pages 86 – 96, 2005.

[25] K. Fant and S. Brandt. NULL Convention Logic: A Complete and Consistent Logic for Asynchronous Digital Circuit Synthesis. In *ASAP 96. Proceedings of International Conference on Application Specific Systems, Architectures and Processors, 1996.*, pages 261–273, Aug 1996.

[26] M. Ferringer. Coupling asynchronous signals into asynchronous logic. *Austrochip 2009, Graz, Austria.* http://www.vmars.tuwien.ac.at/php/pserver/extern/download.php?fileid=1735.

[27] M. Ferringer. Investigating self-timed circuits for the time-triggered protocol. *Reconfigurable Communication-centric Systems on Chip (ReCoSoC 2010)*, May 2010.

[28] M. Ferringer. Towards self-timed logic in the time-triggered protocol. In *2010 International Conference on Dependable Systems and Networks Workshops (DSN-W),*, pages 136 –141, 28 2010-july 1 2010.

[29] M. Ferringer. Conversion and interfacing techniques for asynchronous circuits. In *2011 IEEE 14th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS),*, pages 11 –16, april 2011.

[30] M. Ferringer. Conversion of two- to four-phase delay-insensitive asynchronous circuits. In *EUROCON - International Conference on Computer as a Tool (EUROCON), 2011 IEEE*, pages 1 –4, april 2011.

[31] M. Ferringer. Investigating the impact of process variations on an asynchronous time-triggered-protocol controller. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W),*, pages 47 –52, june 2011.

[32] M. Ferringer. On self-timed circuits in real-time systems. *International Journal of Reconfigurable Computing: Selected Papers from the International Workshop on Reconfigurable Communication-Centric Systems on Chips (ReCoSoC 2010)*, Feb. 2011.

[33] M. Ferringer, G. Fuchs, A. Steininger, and G. Kempf. VLSI Implementation of a Fault-Tolerant Distributed Clock Generation. In *21st IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2006. DFT '06.*, pages 563–571, Oct. 2006.

[34] R. Forster. Manchester encoding: opposing definitions resolved. *Engineering Science and Education Journal*, 9(6):278 –280, dec 2000.

[35] E. Friedman. Clock distribution networks in synchronous digital integrated circuits. *Proceedings of the IEEE*, 89(5):665 –692, may 2001.

[36] M. Fugger, U. Schmid, G. Fuchs, and G. Kempf. Fault-tolerant distributed clock generation in vlsi systems-on-chip. In *Sixth European Dependable Computing Conference, 2006. EDCC '06.*, pages 87 –96, oct. 2006.

[37] S. Furber, P. Day, J. Garside, N. Paver, S. Temple, and J. Woods. The design and evaluation of an asynchronous microprocessor. pages 217 –220, oct. 1994.

[38] S. Furber, D. Edwards, and J. Garside. Amulet3: a 100 mips asynchronous embedded processor. In *2000 International Conference on Computer Design, 2000. Proceedings.*, pages 329 –334, 2000.

[39] S. Furber, J. Garside, and D. Gilbert. Amulet3: a high-performance self-timed arm microprocessor. pages 247 –252, oct. 1998.

[40] R. Ginosar. Fourteen ways to fool your synchronizer. In *ASYNC '03: Proceedings of the 9th International Symposium on Asynchronous Circuits and Systems*, page 89, Washington, DC, USA, 2003. IEEE Computer Society.

[41] M. Goresky and A. Klapper. Fibonacci and galois representations of feedback-with-carry shift registers. *IEEE Transactions on Information Theory*, 48(11):2826 – 2836, nov 2002.

[42] F. Gurkaynak, S. Oetiker, H. Kaeslin, N. Felber, and W. Fichtner. Improving dpa security by using globally-asynchronous locally-synchronous systems. In *Proceedings of the 31st European Solid-State Circuits Conference, 2005. ESSCIRC 2005.*, pages 407 – 410, 12-16 2005.

[43] S. Hauck. Asynchronous Design Methodologies: An Overview. In *Proceedings of the IEEE*, volume 83, pages 69–93, Jan. 1995.

[44] D. Howe. Interpreting oscillatory frequency stability plots. In *IEEE International Frequency Control Symposium and PDA Exhibition, 2002.*, pages 725–732, 2002.

[45] W. Huber. *Design of an Asynchronous Processor Based on Code Alternation Logic - Explorations of Delay Insensitivity.* PhD thesis, Technische Universität Wien, Institut für Technische Informatik, 2005.

[46] A. Kapoor, N. Jayakumar, and S. Khatri. A novel clock distribution and dynamic deskewing methodology. In *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.*, pages 626 – 631, 7-11 2004.

[47] D. Kinniment, A. Bystrov, and A. Yakovlev. Synchronization circuit performance. *IEEE Journal of Solid-State Circuits,*, 37(2):202 –209, feb 2002.

[48] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Kluwer Academic Publishers, MA, USA, 1997.

[49] H. Kopetz and G. Grundsteidl. TTP - A Time-Triggered Protocol for Fault-Tolerant Real-Time Systems. *Symposium on Fault-Tolerant Computing, FTCS-23.*

[50] S. Krishnamurthy, S. Paul, and S. Bhunia. Adaptation to temperature-induced delay variations in logic circuits using low-overhead online delay calibration. In *8th International Symposium on Quality Electronic Design, 2007. ISQED '07.*, pages 755 –760, 2007.

[51] M. Krstic, E. Grass, F. Gurkaynak, and P. Vivet. Globally asynchronous, locally synchronous circuits: Overview and outlook. *Design Test of Computers, IEEE*, 24(5):430 –441, sept.-oct. 2007.

[52] M. Krstic, E. Grass, and C. Stahl. Request-driven gals technique for wireless communication system. In *11th IEEE International Symposium on Asynchronous Circuits and Systems, 2005. ASYNC 2005. Proceedings.*, pages 76 – 85, 2005.

[53] A. Kuo, R. Rosales, T. Farahmand, S. Tabatabaei, and A. Ivanov. Crosstalk bounded uncorrelated jitter (buj) for high-speed interconnects. *IEEE Transactions on Instrumentation and Measurement,*, 54(5):1800 – 1810, 2005.

[54] C. LaFrieda, B. Hill, and R. Manohar. An asynchronous fpga with two-phase enable-scaled routing. In *2010 IEEE Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 141 –150, May 2010.

[55] J. Lechner. Implementation of a design tool for generation of fsl circuits. Master's thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2008.

[56] D. Linder and J. Harden. Phased logic: supporting the synchronous design paradigm with delay-insensitive circuitry. *IEEE Transactions on Computers*, 45(9):1031–1044, Sep 1996.

[57] B. Liu. On vlsi statistical timing analysis and optimization. In *IEEE 8th International Conference on ASIC, 2009. ASICON '09.*, pages 718 –721, oct. 2009.

[58] H. Mahmoodi, V. Tirumalashetty, M. Cooke, and K. Roy. Ultra low-power clocking scheme using energy recovery and clock gating. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems.*, 17(1):33 –44, jan. 2009.

[59] A. J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In *Proceedings of the 6th MIT Conference on Advanced Research in VLSI*, jan 1990.

[60] A. McAuley. Four state asynchronous architectures. *IEEE Transactions on Computers*, 41(2):129–142, Feb 1992.

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

[62] N. Miskov-Zivanov and D. Marculescu. A systematic approach to modeling and analysis of transient faults in logic circuits. In *Quality of Electronic Design, 2009. ISQED 2009.*, pages 408–413, March 2009.

[63] A. Mitra, W. McLaughlin, and S. Nowick. Efficient asynchronous protocol converters for two-phase delay-insensitive global communication. In *Asynchronous Circuits and Systems, 2007. ASYNC 2007. 13th IEEE International Symposium on*, pages 186 –195, 2007.

[64] R. Mullins and S. Moore. Demystifying data-driven and pausible clocking schemes. In *13th IEEE International Symposium on Asynchronous Circuits and Systems, 2007. ASYNC 2007.*, pages 175 –185, 2007.

[65] C. J. Myers. *Asynchronous Circuit Design.* John Wiley & Sons, Inc., Wiley-Interscience, John Wiley & Sons, Inc., 605 Third Avenue, New York, N.Y. 10158-0012, 2001.

[66] F. Najm. On the need for statistical timing analysis. In *Design Automation Conference, 2005. Proceedings. 42nd*, pages 764 – 765, june 2005.

[67] A. Nardi, E. Tuncer, S. Naidu, A. Antonau, S. Gradinaru, T. Lin, and J. Song. Use of statistical timing analysis on real designs. In *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, pages 1 –6, april 2007.

[68] L. Necchi, L. Lavagno, D. Pandini, and L. Vanzago. An ultra-low energy asynchronous processor for wireless sensor networks. pages 8 pp. –85, mar. 2006.

[69] D. H. Neil H. E. Weste. *CMOS VLSI Design: A Circuit and Systems Perspective.* Addison-Wesley, 2005.

[70] B. Paul, K. Kang, H. Kufluoglu, M. Alam, and K. Roy. Impact of nbti on the temporal performance degradation of digital circuits. *IEEE Electron Device Letters*, 26(8):560 – 562, aug. 2005.

[71] J. Pontes, R. Soares, E. Carvalho, F. Moraes, and N. Calazans. Scaffi: An intrachip fpga asynchronous interface based on hard macros. In *25th International Conference on Computer Design, 2007. ICCD 2007.*, pages 541 –546, 2007.

[72] P. Ramanathan, K. Shin, and R. Butler. Fault-tolerant clock synchronization in distributed systems. *Computer*, 23(10):33 –42, oct 1990.

[73] S. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas. Varius: A model of process variation and resulting timing errors for microarchitects. *IEEE Transactions on Semiconductor Manufacturing*, 21(1):3 –13, 2008.

[74] C. L. Seitz. System timing. In C. A. Mead and L. A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.

[75] M. Shams, J. Ebergen, and M. Elmasry. A comparison of cmos implementations of an asynchronous circuits primitive: the c-element. In *International Symposium on Low Power Electronics and Design, 1996.*, pages 93 –96, aug 1996.

[76] W. Shen, Y. Cai, X. Hong, and J. Hu. An effective gated clock tree design based on activity and register aware placement. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems.*, PP(99):1 –1, 2009.

[77] R. Shi, W. Yu, Y. Zhu, C.-K. Cheng, and E. Kuh. Efficient and accurate eye diagram prediction for high speed signaling. In *IEEE/ACM International Conference on Computer-Aided Design, 2008. ICCAD 2008.*, pages 655 –661, 2008.

[78] M. Shimanouchi. An approach to consistent jitter modeling for various jitter aspects and measurement methods. In *Proceedings of IEEE International Test Conference, 2001.*, pages 848–857, 2001.

[79] M. Simlastik and V. Stopjakova. Automated synchronous-to-asynchronous circuits conversion: A survey. pages 348–358, 2009.

[80] A. Sjogren and C. Myers. Interfacing synchronous and asynchronous modules within a high-speed pipeline. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(5):573 –583, Oct. 2000.

[81] J. Sparso and S. Furber. *Principles of Asynchronous Circuit Design - A Systems perspective.* Kluwer Academic Publishers, MA, USA, 2001.

[82] I. E. Sutherland. Micropipelines. *Communications of the ACM, Turing Award*, 32(6):720–738, JUN 1989. ISSN:0001-0782.

[83] P. Teehan, M. Greenstreet, and G. Lemieux. A survey and taxonomy of gals design styles. *IEEE Design and Test of Computers*, 24(5):418 –428, sept.-oct. 2007.

[84] J. Teifel. Asynchronous cryptographic hardware design. In *Proceedings 2006 40th Annual IEEE International Carnahan Conferences Security Technology.*, pages 221 –227, oct. 2006.

[85] Tektronix. Understanding and Characterizing Timing Jitter. 2003.

[86] TTA-Group. Time-Triggered Protocol TTP/C High-Level Specification Document, Protocol Version 1.1. 2003.

[87] TTTech Computertechnik AG and www.austriamicrosystems.com. *AS8202NF - TTP-C2NF Communication Controller*, revision 2.1 edition.

[88] O. Unsal, J. Tschanz, K. Bowman, V. De, X. Vera, A. Gonzalez, and O. Ergin. Impact of parameter variations on circuits and microarchitecture. *Micro, IEEE*, 26(6):30 –39, 2006.

[89] C. Van Berkel, M. Josephs, and S. Nowick. Applications of asynchronous circuits. *Proceedings of the IEEE*, 87(2):223 –233, Feb. 1999.

[90] T. Verhoeff. Delay-insensitive codes - an overview. In *Distributed Computing*, volume 3, jan 1987.

[91] P. Walker and S. Ghosh. On the nature and inadequacies of transport timing delay constructs in vhdl descriptions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(8):894 –915, aug 1997.

[92] T. Werner and V. Akella. Asynchronous processor survey. *Computer*, 30(11):67 –76, nov 1997.

[93] A. Winstanley, A. Garivier, and M. Greenstreet. An event spacing experiment. In *Eighth International Symposium on Asynchronous Circuits and Systems, 2002. Proceedings.*, pages 47–56, April 2002.

[94] T. Yamaguchi, M. Soma, D. Halter, R. Raina, J. Nissen, and M. Ishida. A method for measuring the cycle-to-cycle period jitter of high-frequency clock signals. In *19th IEEE Proceedings on VLSI Test Symposium. VTS 2001*, 2001.

[95] I. Zamek and S. Zamek. Definitions of jitter measurement terms and relationships. In *Proceedings of IEEE International Test Conference, 2005. ITC 2005.*, pages 10 pp.–34, Nov. 2005.

[96] V. Zebilis and C. Sotiriou. Controlling event spacing in self-timed rings. In *11th IEEE International Symposium on Asynchronous Circuits and Systems, 2005. ASYNC 2005.*, pages 109–115, March 2005.

[97] C. Zhang and L. Forbes. Simulation of timing jitter in ring oscillators. In *University/Government/Industry Microelectronics Symposium, 2003. Proceedings of the 15th Biennial*, 2003.

Curriculum Vitae
Markus Ferringer

Date of Birth	March, 20th, 1981
Place of Birth	Gmunden, Austria
1996-1991	Primary school, Gmunden
1991-1999	Secondary school, Gmunden
2000-2004	Bachelor Curriculum Technical Computer Science Vienna University of Technology
2004-2006	Master Curriculum Technical Computer Science Vienna University of Technology Graduation with distinction
2007-2011	PhD Curriculum Computer Engineering Vienna University of Technology
2002-2005	Carbone Lorraine GmbH, Vienna Software and database engineer
2006-2011	TTTech Computertechnik AG, Vienna Off-Highway electronics engineer
2007-2011	Vienna University of Technology Department of Computer Engineering Project assistant
2011-today	Riedel Communications GmbH, Vienna Hardware engineer