Globally-Asynchronous Locally-Synchronous Architectures for VLSI Systems

A dissertation submitted to the

SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZURICH

for the degree of
Doctor of Technical Sciences

presented by

JENS MUTTERSBACH
Dipl.-Phys., University of Cologne
born October 4th, 1968
citizen of Germany

accepted on the recommendation of

Prof. Dr. W. Fichtner, examiner
Prof. Dr. G. Tröster, co-examiner

2001
Seite Leer / Blank leaf
O beatissime lector, lave manus tuas
et sic librum adprehende,
leniter folia turna,
longe a littera digitos pone.
Quia qui nescit scribere,
putat hoc esse nullum laborem.
O quam gravis est scriptura:
oculos gravat, renes frangit,
simul et omnia membra contristat.
Tria digita scribunt, totum corpus laborat...

O fortunate reader, wash your hands
and so touch this book,
turn the pages gently,
and keep your fingers far from the letters.
Those, who cannot write,
do not believe that this is labor.
O how demanding is writing:
it blurs the eyes, crushes the kidneys,
and simultaneously tortures all limbs.
Three fingers write, the whole body suffers....

Writer at monastery of Seeon
8th century

O glücklichster Leser, wasche deine Hände
und fasse so das Buch an,
drehe die Blätter sanft,
halte die Finger weit ab von den Buchstaben.
Der, der nicht schreiben kann,
glaubt nicht, dass dies eine Arbeit sei.
O wie schwer ist das Schreiben:
Es trübt die Augen, quetscht die Nieren
und bringt zugleich allen Gliedern Qual.
Drei Finger schreiben, der ganze Körper leidet...

Schreiber Kloster Seeon
8. Jahrhundert
First of all, I want to thank Prof. Wolfgang Fichtner, who encouraged me to work in this field. His supervision and the excellent working environment at the Integrated Systems Laboratory (IIS) were essential for the success of this work. I would also like to thank Prof. Gerhard Tröster for reading and co-examining my thesis.

Special thanks goes to Norbert Felber and Hubert Kaeslin for their technical guidance, their creative ideas and all their questions triggering fruitful discussions.

I also want to thank all members of the digital design group at IIS for the colleagueship and fun over the last years. Some of them contributed to this work in a special way: Ron Tschalär taught me the art of VLSI design, Thomas Villiger was my partner on the GALS project and carried out great parts of the MARILYN design, Robert Reutemann was a constant source of ideas and hints regarding design tools, Marc Oberholzer and David Studer performed the design of the MERLIN chip.

Thanks to the collaboration with Philips Semiconductors Zürich and especially the discussions with Frank Kesel and Daniel Müller I gained many insights into industrial design practice and its impact on the applicability of academic concepts. The financial support from Philips Semiconductors Zürich, Infineon Technologies Germany, and the Swiss Comission for Technology and Innovation (KTI) is gratefully acknowledged.

Last but most important, thanks and love goes to my wife Kerstin for just everything that she gives and means to me.
Contents

Acknowledgements vii

Abstract xiii

Zusammenfassung xv

1 Introduction 1
   1.1 Motivation ................................. 1
   1.2 Outline .................................. 2

2 Background 5
   2.1 Asynchronous vs. Synchronous Design Techniques .......... 6
      2.1.1 The Synchronous Design Paradigm .................... 7
      2.1.2 Asynchronous Design Techniques ..................... 8
   2.2 Synchronizers .............................. 14
   2.3 Existing Multisynchronous Approaches ................... 17

3 Modular Asynchronous Wrappers 23
   3.1 Architecture of Asynchronous Wrappers .................. 23
   3.2 Pausable Clock Generation ........................... 25
      3.2.1 Programmable Delay Line ......................... 25
      3.2.2 Arbitration ................................. 27
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.3</td>
<td>Implementation</td>
<td>28</td>
</tr>
<tr>
<td>3.3</td>
<td>Port Controllers</td>
<td>32</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Demand-Type Input Controller</td>
<td>35</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Demand-Type Output Controller</td>
<td>44</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Poll-Type Input Controller</td>
<td>47</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Poll-Type Output Controller</td>
<td>49</td>
</tr>
<tr>
<td>3.4</td>
<td>Library of Wrapper Components</td>
<td>51</td>
</tr>
<tr>
<td>4</td>
<td>Data Transfers between Blocks</td>
<td>55</td>
</tr>
<tr>
<td>4.1</td>
<td>Transfer Channels between Locally-Synchronous Blocks</td>
<td>56</td>
</tr>
<tr>
<td>4.2</td>
<td>Memory Accesses</td>
<td>59</td>
</tr>
<tr>
<td>4.3</td>
<td>Performance Evaluation</td>
<td>61</td>
</tr>
<tr>
<td>5</td>
<td>System Partitioning</td>
<td>67</td>
</tr>
<tr>
<td>5.1</td>
<td>Block Sizes and Boundaries</td>
<td>67</td>
</tr>
<tr>
<td>5.2</td>
<td>Communication Configurations</td>
<td>70</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Fork Channels</td>
<td>70</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Deadlock Analysis</td>
<td>71</td>
</tr>
<tr>
<td>6</td>
<td>Realization of a GALS System</td>
<td>75</td>
</tr>
<tr>
<td>6.1</td>
<td>The SAFER Crypto-Algorithm</td>
<td>75</td>
</tr>
<tr>
<td>6.2</td>
<td>GALS Partitioning</td>
<td>82</td>
</tr>
<tr>
<td>6.3</td>
<td>Testing Issues</td>
<td>87</td>
</tr>
<tr>
<td>6.4</td>
<td>Implementation and Current Design-Flow</td>
<td>88</td>
</tr>
<tr>
<td>6.5</td>
<td>A GALS-Compliant Tool-Flow</td>
<td>94</td>
</tr>
<tr>
<td>6.6</td>
<td>Performance and Power Measurements</td>
<td>95</td>
</tr>
<tr>
<td>6.6.1</td>
<td>GALS Implementation (Marilyn)</td>
<td>95</td>
</tr>
<tr>
<td>6.6.2</td>
<td>Synchronous Reference (Merlin)</td>
<td>102</td>
</tr>
<tr>
<td>7</td>
<td>Conclusions</td>
<td>107</td>
</tr>
<tr>
<td>7.1</td>
<td>Results</td>
<td>107</td>
</tr>
</tbody>
</table>
Seite Leer / Blank leaf
Abstract

This thesis describes the specification and implementation of a design methodology for globally-asynchronous locally-synchronous (GALS) architectures. Such architectures are a novel approach to the design of complex VLSI systems which are suffering from clocking problems and high power consumption.

In a GALS architecture the system gets partitioned into several independently clocked modules which are communicating in a self-timed manner. Thus the functionality of each locally-synchronous (LS) module can be described and synthesized along well established synchronous design flows and clocking problems are eased by confining them to a moderately sized subsystem. The circuitry necessary to coordinate clock-driven with self-timed operation is contained in an asynchronous wrapper that surrounds each LS module.

These wrappers have a modular internal structure that can be assembled from a library of six basic elements: four different port controllers for input and output ports, respectively, a memory access controller, and a clock generation unit. To prevent metastability while transferring data across clock boundaries, the concept employs a pausable clocking scheme, i.e. whenever data changes and sampling clock edge occur dangerously close to another, either the clock or the data get delayed. The pausable clock is provided by an on-chip clock generator, which does properly arbitrate between numerous concurrent requests for pausing and active clock edges. The frequency of the clock is programmable for a large range of values.

Data channels between LS modules, which are established with the mentioned basic elements, are able to safely transfer over $300 \times 10^6$ data words of
arbitrary width per second\textsuperscript{1}. The channels have a latency of far less than one clock period and transmissions in subsequent local clock cycles are possible. These two features and the prevention of metastability distinguish the proposed method from all previous concepts.

By realizing a cryptosystem in GALS architecture the methodology is validated on silicon. The ASIC encrypts and decrypts data with the SAFER SK-128 algorithm in all standardized block cipher modes. The system contains 58,000 gate equivalents and scan-testability of synchronous circuitry is fully sustained. This work is the first to demonstrate GALS operation for a system of substantial complexity.

A second ASIC implements the same cryptosystem in a conventional globally-synchronous architecture and serves as a reference for comparisons. Both chips have been manufactured on the same wafer. The measured maximum data throughput of the synchronous design is about 30\% higher than that of the GALS counterpart. On the other hand the synchronous chip, which already employs clockgating to save power, dissipates 30\% more energy per Mbit of data than the GALS implementation.

\textsuperscript{1}All measured numbers refer to a 0.25\textmu m CMOS technology.
Zusammenfassung


\(^2\)Alle Zahlenangaben beziehen sich auf eine 0.25μm CMOS Technologie
Chapter 1

Introduction

1.1 Motivation

Most digital VLSI systems of the past and present are designed in a synchronous fashion, i.e. they rely on the use of a chip-wide clock signal. The associated design strategy bases on the assumption that transitions on the global clock signal do occur (almost) simultaneously everywhere in a particular integrated circuit. There are good reasons that made synchronous styles so popular and the most important is their simplicity during design and verification. But the progress in CMOS technologies has made the clock the seed for some crucial challenges and problems:

- Faster combinatorial logic asks for shorter clock periods, but the increase in available die size means that this high frequency clock has to be distributed over larger distances.

- The allowed clock skew margin decreases for faster logic gates, but the growing relative influence of interconnect delays tends to make clock skew larger.

- To distribute a clock signal despite the aforementioned obstacles accounts for a considerable fraction of the system’s power consumption (typically one quarter to one third of total power).
As soon as the dawning crisis of synchronous design was perceived the interest in fundamentally different approaches began to rise – the most radical being asynchronous design. Fine grained asynchronous design techniques completely eliminate the global synchronization signal and the related problems. It gets substituted by distributed control circuitry that ensures coordinated and safe operation. Although purely self-timed methods are elegant and appealing from an academic point of view, the area overhead and power consumption created by the larger control circuitry often outweighs its advantages.

This thesis deals with the development of globally-asynchronous locally-synchronous (GALS) architectures to facilitate the design of complex VLSI systems. It amalgamates the best of both design methodologies: Inside individual clock domains the conventional synchronous design methodology gets applied. Between any two clock domains data exchanges follow asynchronous guidelines. This approach eliminates most of the control circuitry of fine-grained asynchronous designs and the related power consumption as well as removing the need to provide a precisely balanced global clock.

The major goal of the work described in this thesis is to prove the feasibility of GALS architectures by developing the necessary methodology and by implementing a GALS system on silicon. The new methodology shall be compatible with industrial VLSI design practice. This implies that the implementation shall be based on the use of conventional standard cell libraries and the system designer should not be bothered with asynchronous design techniques, that he is not familiar with.

1.2 Outline

In order to mediate between the synchronous circuitry inside the locally-synchronous (LS) modules and the surrounding asynchronous network, each LS module gets equipped with an asynchronous wrapper (fig. 1.1). The circuitry in these wrappers defines protocols to both sides and makes sure that data are exchanged safely across the boundaries of clock domains. This necessitates the ability to pause the local clock and for this reason the clock generation is also situated inside the asynchronous wrapper. The implementation of wrappers should be as simple as possible for the system de-
1.2. Outline

Figure 1.1: Globally-asynchronous locally-synchronous (GALS) architecture using asynchronous wrappers

signer, therefore they are assembled from a small set of predesigned modules, which are provided as netlist macros.

This thesis will describe the construction of complete GALS systems and thus needs to travel through all levels of its hierarchical organization. To prevent confusion we will adhere to some naming conventions: The locally-synchronous islands are called (LS) modules, while the constituents of the asynchronous wrapper are named units. On system level the entity formed by an LS module and the corresponding wrapper is a block.

Furthermore, all numbers describing timing behaviour, area or power consumption in this thesis refer to a 0.25μm, 5-metal interconnect CMOS process in which the circuits were designed and fabricated.

The thesis is structured as follows:

After a brief introduction in this chapter, chapter 2 describes the background needed for the understanding and evaluation of this work. It defines certain terms, outlines the current state of asynchronous controller design and presents some existing work on multisynchronous architectures.

Chapter 3 describes the specification and implementation of all required wrapper constituents, which are then assembled to data transmission channels between blocks in chapter 4. We do also explain how the safety of data transfers across clock boundaries gets ensured and determine the performance of the channels.
Some guidelines for transferring a conventional synchronous system into a GALS architecture are given in chapter 5, including notes on additional communication configurations and deadlock safety.

Chapter 6 describes details about the design and performance of a GALS system implementing the SAFER cryptoalgorithm. To provide fair comparisons another design team implemented the same functionality in a conventional globally-synchronous design style.

This thesis is completed with conclusions and an outlook to future developments in chapter 7.
Chapter 2

Background

This chapter shall provide a common starting point for the understanding of this thesis. This is of special importance as subsequent chapters do extensively use asynchronous design methodologies the average reader might not be familiar with. To provide a complete picture a brief definition of the fundamental concepts and limitations of synchronous design is also included.

Synchronous circuits or (sub)systems inevitably have to interface to a non-synchronized or asynchronous environment. Unknown timing relations at this interface give rise to possibly non-deterministic behaviour of the synchronous circuit, if data and sampling clock events occur in an indistinguishable manner. Two major approaches exist to cope with this metastability problem (see table 2.1): The first one is to tackle the conflict from

<table>
<thead>
<tr>
<th></th>
<th>recovery from metastability</th>
<th>prevention of metastability</th>
</tr>
</thead>
<tbody>
<tr>
<td>acts on data lines</td>
<td>classical synchronizers</td>
<td>adaptive synchronization</td>
</tr>
<tr>
<td>clock stretching</td>
<td>metastab. detection, Q-modules</td>
<td>GALS</td>
</tr>
</tbody>
</table>

Table 2.1: Methods for synchronization
the data signal's side, which is known as synchronization (in a traditional sense). In the second category the clock gets manipulated in order to assure data integrity. Both classes can be further subdivided by whether they try to recover after metastability has occurred or they prevent metastability before it occurs.

GALS architectures belong to the class of concepts which prevent metastability by manipulating the clock signal in an appropriate way. We will give an overview over previous work in the section on multisynchronous approaches.

2.1 Asynchronous vs. Synchronous Design Techniques

Although there was little distinction between synchronous and asynchronous circuits during the early days of digital electronics, the main-stream became focussed on the synchronous style since the 1960's. This was mainly due to the fact that synchronous circuits are easier to design and understand. Decades of research and experience in this field result in a large number of academic and commercial design tools for synchronous VLSI. Therefore the reader is expected to be familiar with the methodology and the coverage of synchronous terms and concepts will be limited to some basic definitions.

The situation for asynchronous design methods is fundamentally different. Although it is an inherently larger class of circuits, little development effort was spent on it. Only after progress in CMOS technology made the limitations of the synchronous paradigm apparent, research in asynchronous styles blossoms since the late 1980's. Most of recent research is focussed more on theory than practice. Nonetheless some practical circuits are designed to prove the advantages of asynchronous styles. Though this thesis profits from numerous previous works in the field, we do not intend to cover all aspects of asynchronous research and developments. Only the most relevant concepts and design methods will be described. The more interested reader is referred to the well written overviews, namely by Hauck [Hau95], Davis [DN97], Berkel [BJN99] or Josephs [JNvB99].
2.1.1 The Synchronous Design Paradigm

This section describes the essential terms and concepts about synchronous design. Some definitions might slightly differ from the mainstream to adopt to the special situations we encounter in globally-asynchronous locally-synchronous systems.

A system is said to operate synchronously if all memory operations are triggered on a common timing grid which is defined by the status of a global signal referred to as clock. This clock signal must solely be used for the control of memory cells. The remainder of this thesis will only deal with synchronous systems where the timing grid is defined by rising edge events of the clock (positive edge triggered systems).

A property making synchronous systems easy to design and verify is, that all (local) timings are related to a common global clock and delay properties are only subject to one-sided constraints. To ensure timing correctness, all propagation delays between two memory elements shall be shorter than the nominal cycle time \( T_{\text{cycle}} \), which is defined as the minimum time span between two consecutive active clock events. It should be pointed out that these definitions do not constrain the clock to be strictly periodically and thus two active clock events may be further apart than the nominal cycle time. Leaving this freedom on the other hand excludes the use of some design styles, for example dynamic logic.

All definitions so far assumed a clock signal that fulfills an isochronic fork condition, i.e. all events arrive simultaneously at each node of the net. This is an unrealistic simplification that contradicts the physics of signal propagation, but for a long time this discrepancy was not relevant. That changed with shrinking feature sizes and rising speed of CMOS circuits. As soon as insertion delay and skew of the clock signal become a significant fraction of the cycle time, much of the simplicity of synchronous design methodologies gets lost. This leads to a partitioning into clock domains, which are defined as parts of the system that can be designed according to the basic definitions of synchronous circuitry. A measure for the maximum tolerable clock skew \( t_{sk} \) inside a clock domain is given by

\[
\max|t_{sk}| \leq \min[ t_{cd}(\text{FF}) - t_{ho}(\text{FF}) ]
\]

with \( t_{cd}(\text{FF}) \) the contamination delay and \( t_{ho}(\text{FF}) \) the hold time of the used flipflops. If this requirement is violated, hold time violations in shift regis-
ters, for example in the scan chain, are likely to occur. It is worth mentioning that the maximum tolerable clock skew only depends on properties of the used standard cell implementations and thus an ASIC designer usually has no control over it.

Data transfers across the boundaries of clock domains require special attention as the assumption, that data are settled and valid before an active clock edge arrives, is not necessarily valid any more. Fundamentals of such synchronizer circuits will be explained in section 2.2

2.1.2 Asynchronous Design Techniques

In asynchronous design the concept of discrete time steps, which is the root of the synchronous paradigm, is given up and has to be substituted by some other data-valid signaling schemes. Proponents of asynchronous design styles claim some major advantages:

- Easing of global timing issues and no clock skew problems
- Average case instead of worst case performance
- Lower power consumption
- Automatic adaption to physical properties (temperature, supply etc.)
- Better technology migration potential
- Modularity of designs

As only asynchronous controller design is of interest for GALS architectures, we will focus on the issues relevant for this and neglect the field of asynchronous datapath synthesis.

Terms and Concepts

Signaling Scheme To mark the validity of data or control signals without a global clock dual-rail encoding uses two bitlines to encode a single bit of information. Any two values have to be separated by a specified 'null'. Despite some performance advantages implementations using this scheme usually have to face area overhead, considerably larger power dissipation and are vulnerable to hazards [Fur97].
2.1. Asynchronous vs. Synchronous Design Techniques

A **bundled data protocol** employs two separate wires to transmit the validity of a whole data vector. This approach reduces the cost of status transmission (for vectors containing more than 2 bits) but requires careful matching of delays between data and control lines. Mostly **handshake protocols** are used for status interaction between the sender and receiver of data. Interpreting transitions as events on the request and acknowledge lines constitutes the basic idea of **2-phase or transition signaling**. As soon as it comes to implementation details transition signaling gets tricky. That’s why most asynchronous designers prefer **4-phase or level signaling**. A complete cycle in a 4-phase handshake involves four (sequential) events\(^1\): \( \text{Req}+, \text{Ack}+ \), \( \text{Req}-, \text{Ack}- \). Some interpretations can be distinguished that differ in the interval in which data are required to remain valid (Fig.2.1)[Pee96]. On a **push channel** the sender of data drives the request line and the acknowledgment is asserted by the data receiver. The other way round would constitute a **pull channel**, where the receiver initiates the transmission.

**Delay Assumptions and Robustness** In the absence of a global clock, event ordering is critical for asynchronous circuits. Thus implementations are often classified by their robustness against delay variations, that might disturb event ordering and cause circuit malfunction.

In **bounded-delay** (BD) designs both gate and wire delays are assumed to be known. This is rather close to the assumptions used in synchronous

---

\(^1\) ‘+’ denotes a rising transition of a signal while ‘-’ indicates a falling edge.
design and is the least robust model with respect to variations induced by different PTV\textsuperscript{2} conditions for example. Careful technology mapping, placing and routing are necessary to meet the delay constraints.

A speed-independent (SI) circuit is an asynchronous circuit which works correctly under the assumption, that all gate delays are unbounded, while wire delays are negligible (less than the minimum gate delay)[Hau95]. A closely related class are quasi-delay-insensitive (QDI) circuits: they adopt the speed-independent assumptions and supplement it with the isochronic fork, i.e. that differences in delays between several destinations of a particular net are negligible. For most practical purposes SI and QDI circuits are identical. Both classes require careful layout and routing, as the neglect of different wire delays is hardly valid in modern technologies.

Delay-insensitive (DI) circuits are required to work correctly for any gate and wire delay. Both are arbitrary but finite. This set of assumptions is obviously the most robust but also the hardest to meet. Therefore the class of DI circuits consisting of basic gates is almost empty [CK97]. DI implementations mostly contain macromodules which are designed in a way such that wire delay variations outside the macromodules do not cause malfunction of the circuit. The internal structure of these macromodules is not strictly delay-insensitive. There are numerous theoretical and practical works concerning DI circuits [Hau95].

**Structured Design Approaches**

As interest in asynchronous design became stronger during the last years, several approaches to structured design styles have been investigated. We can roughly distinguish two classes among them, compilation from a high-level language and graph-based methods, which are both mainly different interfaces to the designer.

**High-Level Languages**

Compilation from a high-level description requires a language which is able to model the concurrency of asynchronous behavior. The synthesis typically consists of a chain of transformations to generate SI or DI circuits and

\textsuperscript{2}Process, Temperature, Voltage
map them onto hardware. Ebergen [Ebe91] derived his language from trace theory, while the works of Martin [Mar86] and Brunvand [BS89] are based on Hoare’s communicating sequential processes (CSP) [Hoa85]. This approach also forms the foundation for the Tangram and Balsa projects, that will be discussed later.

The main advantage of a high-level description is that complex asynchronous systems can be described, verified and modified without concerns about low-level timing issues. But the implementations derived by synthesis are mostly non-optimal, as global optimization techniques are still a major problem.

To prove the feasibility of asynchronous design on industry-scale circuits, Philips Natlab and the TUEindhoven started the Tangram project in the early 1990’s. Berkel et al. [Ber93] developed a language (Tangram) that can be synthesized to quasi-delay-insensitive circuits and is based on the ideas of communicating sequential processes (CSP). In Tangram processes are communicating via channels where one participant is denoted to be the initiator of handshaking.

The code is amenable to both direct simulation and compilation into handshake circuits. Thus the designer gets the possibility to simulate system behavior (functionality, performance, power consumption) prior to any synthesis steps. The Tangram team tries to transform VLSI design into a programming activity, where the designer/programmer does not even need to think about circuit implementations.

The compilation into handshake circuits is kept transparent, i.e. the processes instantiated in the program are directly transferred into communicating components. Only simple peephole optimizations are performed. This approach transfers all the burden of optimization to the designer, who has to modify the Tangram high level code himself. Synthesis results are thus easy to control but often non-optimal.

In the Tangram approach details of handshake protocols are decoupled from the language itself, thus it is possible to realize internal implementation changes by switching to a different library during compilation. In the beginning dual-rail coding was used which resulted in large area overhead. Peeters developed single-rail implementations which reduced area and

---

3Peephole optimization is a local optimization technique utilizing pattern recognition to find structures that can be substituted with optimized implementations.
power consumption while offering better throughput [Pee96]. The whole library needed for Tangram contains about 30 elements including control and data handshake circuits. Such a small library can also easily be transferred between target technologies.

In the last years effort was spent on the development and integration of test tools into the design flow. Berkel et al. claim to reach (stuck-at) fault coverages of more than 98% percent which makes it possible to use Tangram for commercial applications. Exploiting the performance optimum of asynchronous circuits in synchronous environments, e.g. on boards, remains a major problem.

The Amulet design team from Manchester University started to write their own tool based on a core of Tangram ideas and algorithms. Their language and tool (Balsa) is – in contrast to Philips' Tangram – publicly available but not yet as mature as the Philips tool suite.

What distinguishes these two projects from many other asynchronous design examples is that their methodology allows to design complex asynchronous systems (controller and datapath) with short turnaround times and thus bridges the gap between academic research and industrial needs. This is proven by a couple of successful asynchronous products [BBK+94, GBvB+98, KKdB+00].

**Graph-Based Methods**

Most graph-based methods rely on Petri nets to describe concurrent behavior of a system. There are numerous modifications regarding the syntax and interpretation of Petri nets. Over the last years the problems occurring during graph transformation, state coding and technology mapping have been solved to a great part [Chu87, Men91, CKLY97] and some tools for this style are available.

Graph-based specifications are strong in expressing concurrency in processes. That they need a very detailed specification by the designer is both advantage and drawback. They offer lots of optimization possibilities to experienced designers but are harder to apply in the beginning. Modifications to a system possibly require a thorough redesign. Methods and tools for verification and testing are still a matter of research.
2.1. Asynchronous vs. Synchronous Design Techniques

Extended Burst Mode Specifications Most design methods for asynchronous circuits are totally event-driven, i.e. the current state of the system is completely determined by the polarity of signal transitions and their order. There is no capability to model level-sensitivity at inputs. This gap between synchronous and asynchronous styles is bridged by the extended-burst-mode design style.

Burst-mode design was developed by Nowick, Yun and Dill [NYD92, NDDH93] based on earlier work at HP labs [DCS93]. It is a graph-based method where each arc is labeled with a non-empty set of input transitions (an input burst) and a set of output transitions (an output burst). All transitions in an input burst are allowed to occur in arbitrary temporal order and the machine does not react until the entire burst has occurred. The machine then fires the specified output burst and enters the next state. New inputs are allowed only after the system has settled in response to the previous input burst (fundamental mode assumption). Input bursts leaving one state are required to be unambiguous, i.e. no input burst can be a subset of another input burst leaving the same state.

The extended-burst-mode specifications [Yun94] add another two important features: level-sensitive conditional inputs and directed don’t cares. Fig. 2.2 gives an example. Signals not enclosed in angle brackets and ending with + or - are usual edge sensitive, terminating signals. The signals enclosed in angle brackets are conditionals, which are level signals the values of which are sampled when all of the terminating edges associated with them have occurred. A signal ending with an asterisk is a directed don’t care. If a state transition is labeled with a*, the following state transitions must be either labeled with a* or with a+ or a- (the terminating edge for the directed don’t care). Signal a is allowed to change monotonically anywhere between the first a* and the terminating edge.

Due to these supplements extended-burst-mode specifications can be used to model everything from edge-sensitive asynchronous behavior to synchronous design. Yun developed algorithms and a synthesis tool (3D) to automatically derive logic equations out of the graph description. Assuming unbounded wire delays in combinational circuits and bounded wire delays for feedbacks from outputs to inputs, Yun is able to generate results with hazard-free state assignment and a state encoding free from critical races. Dynamic hazards can be suppressed by synthesis algorithms.
The extended-burst-mode design style suits for interfacing synchronous and asynchronous domains as one can model the event-oriented asynchronous domain as well as the burst-mode fashion of synchronous circuits. If a state machine does not get too complex the fundamental mode assumption is easily assured. These are the reasons why the asynchronous controllers needed for GALS implementations will be designed using extended-burst-mode and the 3D tool.

2.2 Synchronizers

Wherever a signal enters a synchronous domain from the outside – this can be an asynchronous environment or an independent clock domain – the signal’s status might change anytime as its source has no fixed timing relation to the sampling clock. Occasionally the signal will change undistinguishably close to an active clock event, a situation called marginal triggering. This drives the sampling bistable into an undecided state. This phenomenon is basically due to the fact that latch circuits have three stable operating points: one at logic high and low and one at some mid-level. In practice this mid-level operating point is metastable and gets resolved after some time.
Metastability was first observed in the 1960's [Gra63, CM73] and the bipolar devices of this era did actually maintain a logically undefined mid-level for reasonable time. In modern CMOS flipflops metastability yields an overly long propagation delay of the cell. The basic result is the same in both cases: subsequent circuitry may sample wrong or inconsistent values and the behaviour of the system may become undetermined. As this has to be prevented, synchronizer circuits are inserted at the critical interface. Such a synchronizer usually consists of a cascade edge-triggered bistable which samples the incoming signal.

The probability that the synchronizer's output is incorrect some time $t$ after the sampling clock edge is given by [Vee80, Rab96]

$$P(t) = f_d f_{clk} w e^{-\frac{t}{\tau}}$$  \hspace{1cm} (2.1)

with $f_{clk}$ the clock frequency and $f_d$ the average toggle frequency of the data input. The resolution constant $\tau$ mainly reflects the internal RC constants of the flipflop. The parameter $w$ can be described as the probability that the incoming signal is in an undefined state at the moment of sampling [Rab96]. It thus contains information about both the switching behaviour of the synchronizer and the slope of the input changes. In practice both parameters ($w$ and $\tau$) are determined by fitting the equation to measurements. The inverse of the integrated probability is known as the meantime between error (MTBE) of a synchronizer. Recently the theory of synchronization has been extended slightly by the inclusion of thermal noise effects [DB99].

Because the more critical parameter $\tau$ depends on the internal design of the synchronizer, standard flipflops are usually unsuited for synchronization purposes. Dedicated synchronization cells, like simple jamb structures, can outperform standard flipflops with respect to metastability resolution by orders of magnitude [DB99].
Two basic properties can be derived from equation 2.1: Firstly the probability of malfunction is reduced if the clock or data signals toggle less often. Although this sounds simple it has far reaching implications especially for the clock design. Secondly the chance for errors reduces exponentially if we give the synchronizer more time to settle.

This observation is the motivation for cascaded synchronizer configurations, as depicted in fig. 2.3. The settling time for a cascade with $N$ stages is then $N \cdot T_{clk}$ and the probability of synchronization failure can be traded against latency by inserting (or removing) stages. There are numerous related synchronization configurations exploring the possible design space. All have in common that they reduce the probability of synchronization failure but cannot exclude it completely.

One basic assumption in deriving equation 2.1 is that the changes of the input signal are uniformly distributed over time. If we have a signal traversing the boundary between two clock domains, the clocks may skew slowly with respect to one another. This jeopardizes the statistical distribution of signals and makes the situation for synchronization much worse. If marginal triggering occurs it is likely to recur.

For situations where two communicating clock domains are supplied from a common clock source, Kol and Ginosar [GK98] propose to employ adaptive synchronization. Relative clock and data delays between the two domains can be assumed stationary for some time. The phase relations are measured and delays in the data lines are adjusted to assure data correctness and avoid metastability (fig. 2.4). The delay adjustments can be repeated in special training periods during operation.
2.3. Existing Multisynchronous Approaches

Adaptive synchronization can substantially reduce the probability of metastability in the particular situation required for its application [Kol97]. But implementing it in a real system requires considerable effort to be paid to delay engineering. As the delay of the data ready signal is used as an indicator for the delay of all data lines in the corresponding vector, the delays of the whole data bundle ($\delta_D$ and $\delta_R$) need to match closely, which is a two-sided constraint. Nevertheless the idea of confining metastability detection and resolution to a single signal of a data vector, in this case $DataRdy$, is worth being kept in mind.

2.3 Existing Multisynchronous Approaches

A second class of solutions for synchronization does not try to tackle the problem on the data lines, but shifts the (local) clock in a way that ensures safe and correct sampling of incoming data. This requires that the designer is willing to give up the exact clock frequencies and phases usually derived from crystal oscillators. As the clock has become somehow sacrosanct over the last decades the hesitations toward this change of paradigms should not be underestimated. This may explain why there were only few proposals for such solutions until recently.

A systematic approach was presented by Chapiro [Cha84]. In a first proposal he introduces unsynchronous systems, which are GALS systems using flipflops with completion detection. If a metastable state at the flipflop's outputs is detected, the local clock period of the corresponding LS module is stretched until the outputs have settled. This excludes synchronization failures but a module might theoretically be halted for an infinite amount of time. Chapiro elaborates the possible performance vs. reliability trade-off for this scheme and proposes a technique to limit the duration of clock stretching.

The idea to apply clockstretching after metastability has been detected is further elaborated by Rosenberger et al. [RMCF88] in their so called Q-modules. Awad and Smith [AS91] try to automate the transformation of a synchronous system into a GALS architecture, but their interface circuits are hazard-prone and thus not compatible with modern CMOS technologies. The performance of systems based on metastability detection is further analyzed by Afghahi and Svensson [AS92].
In Chapiro’s second concept, *escapement organizations*, he tries to prevent synchronization failure instead of coping with it. He uses handshaking protocols for communication between modules and derives a stretch signal for the local clock from the request- and acknowledge-signals by simple combinatorial circuits (see fig. 2.5). If data arrive too close to an active clock edge, the clock is stretched to provide safe latching. His method relies mainly on hand-editing of schematics, which makes it unsuitable for larger systems, and he does not address hazard or arbitration problems at all. To be valid, his approach requires the explicit use of non-overlapping two-phase clocks and separate access to both clocks. Nevertheless his basic ideas comprise the foundation for the recent works of Yun [YD96] and Bormann [BC97].

In the work of Traver [Tra88] Chapiro’s escapement concept is further elaborated and analyzed. She derives a more structured design methodology and pays much attention to the testability of the system. Nonetheless until here no working implementation of any of the concepts has been reported.

The work from Yun et al. [YD96] develops a modern approach to synthesis of GALS systems by using the extended-burst-mode specifications. Each LS block is attached a pausable clocking control (PCC) module that contains a simple ring oscillator for clock generation (fig. 2.6). The incoming request line $R_{cl}$ gets sampled by a mutual exclusion (ME) element,
2.3. Existing Multisynchronous Approaches

Figure 2.6: (a) Pausable clocking control (PCC) for bidirectional communication (b) two synchronous modules communicating via an asynchronous FIFO channel [YD96]

which has two inputs, that may rise concurrently, but its outputs are restricted to be mutually exclusive [Sei80]. Thus if $R_\alpha$ is high during a rising clock edge the local clock is stretched until $R_\alpha$ is falling again. By latching the acknowledge signal $A_\rho$ they ensure that data are valid during the active clock edge. To solve the arbitration problems arising if several AFSMs try to stop the clock generation simultaneously, Yun introduces arbiter trees. These become large and impractical for increasing fan-ins and fan-outs of modules. Additionally arbiter trees allow at most one port per clock cycle to be active.

Yun et al. validate the PCC concept by realizing a small example on silicon. They insert FIFOs in the communication channels to smoothen bursty data transfer between modules. Their method requires a permanently running clock, which seems rather disadvantageous regarding power dissipation.
In an early publication Bormann proposes to connect synchronous and asynchronous modules using a quasi-delay-insensitive bus architecture [BMC96]. He makes use of the Tangram/Balsa concepts and mainly addresses the problem of large interconnect numbers in asynchronous designs. The asynchronous circuitry is purely event-driven and thus interfacing to synchronous domains remains pretty much a problem.

In a more mature concept Bormann develops an asynchronous wrapper as a standardized frame to embed synchronous blocks into an asynchronous network [BC97]. He uses the extended-burst-mode specifications to develop the asynchronous controllers for input and output ports. A particularly interesting feature is, that the local clock is only running when data transfers are to be performed. This avoids clock power consumption, if there is nothing to compute for the LS part. To implement this concept the LS block is needed to deliver a signal which shows at which in- or output port a data transfer is enabled.

Bormann elaborates detailed specifications for all possible port configurations (push or pull channels, input or output) and the wrapper concept can easily be scaled for large numbers of interconnects. One of the major insufficiencies of this work is, that arbitration of concurrent requests is not properly addressed. This may result in hazards on the clock network.

Recently Hemani and Meincke [HMK+99, MHK+99] evaluated the effects of GALS architectures on system-wide clock power consumption.
They calculate power savings of up to 70% in the clock net. Taking into account that in modern high-performance designs the clock consumes more than one third of the energy, this translates into 20% reduction in overall dissipation (compared to conventional globally-synchronous designs). They do not present a design solution for the asynchronous interconnects and thus some parameters of their calculation remain unproven. Nevertheless they do demonstrate the benefit of GALS architectures for reducing power consumption.

Hemani et al. also outline a strategy for partitioning of large synchronous systems into locally-synchronous modules. Due to missing experience with implementations of GALS systems, they have to make numerous simplifying assumptions and the argumentation necessarily remains rather vague.

An analysis method for performance and timing behaviour of GALS system is presented by Teich and Thiele [TSTM94, TTSM97]. They develop a model using extensions to timed marked graphs and analyze its behaviour, in particular periodicity and throughput rate. To keep unfolding and modeling manageable they have to assume, that all synchronous nodes use the same clock period. Thus the model is unable to predict the effects of frequency optimization in LS modules. Also the influence due to different handshake and wrapper implementations are not addressed.

Jou et al. [JC97b, JC97a] use the term 'globally-asynchronous locally-synchronous' rather different and we briefly describe their approach to prevent confusion: Their work focuses on minimizing control overhead in pipeline stages of fine-grained asynchronous datapaths. In a first step it implements the functionality with gates from a 'synchronous' standard cell library and extracts the longest path. Only this longest path is then re-implemented using self-timed cells and the completion signal of this path is triggering the next pipeline register. This technique resembles the construction of micropipelines [Sut89] and can only be applied to a subset of designs. Furthermore it looses some important advantages of asynchronous design. Regarding power consumption, which was the main intention for their work, their design example consumes even more power than a conventional synchronous counterpart.
Seite Leer / Blank leaf
Chapter 3

Modular Asynchronous Wrappers

This chapter provides detailed information about the circuitry needed to implement a GALS system. After explaining the basic concept of modular asynchronous wrappers, the required constituents are explained with respect to their particular concept and implementation.

3.1 Architecture of Asynchronous Wrappers

The basic purpose of the asynchronous wrapper is to mediate between the strictly synchronous domain of the locally-synchronous (LS) module and its asynchronous environment in a way that ensures that data are correctly transferred across their respective boundaries.

Fig. 3.1 depicts a block level schematic of the internal organization of an asynchronous wrapper. Each data port is equipped with a port controller, that handles the handshake protocol. To achieve data consistency and prevent metastability at the interfaces, each port controller has access to a pausable, on-chip clock-generation unit and can thus ensure that active clock edge and data line transitions never occur dangerously close to one another. The operation of a particular port controller is initiated by the LS module
through a transition on the enable line $En$. That a transfer has taken place is posted back through the transfer acknowledge $Ta$. The asynchronous wrapper in fig. 3.1 is drawn with one input and one output port only, but any number of ports is possible.

Defining asynchronous wrappers in modular way yields a number of advantages:

- The wrapper’s behaviour can be specified from a rather abstract view of the system and is then assembled accordingly.

- The wrapper constituents are available as a small library of pre-designed blocks, described in HDL. Equipped with this library, designers unfamiliar with the intricacies of asynchronous finite-state machines can easily construct a wrapper.

- Transferring a particular wrapper from one CMOS technology to another, does only require to update a small number of library macros.

- Adhering to simple, speed-independent protocols for communication between wrapper components facilitates functional and timing verification for the units.

- If a wrapper component also has a fixed layout representation, its internal timing needs to be verified only once.
3.2 Pausable Clock Generation

The wrapper concept for GALS architectures requires pausable clock generation, which is best done on-chip. The nominal clock frequency shall be tunable over a certain range to exploit the optimum performance of the particular LS module. The tuning is so far performed from the outside.

The clock generation used in this work is based on a ring-oscillator consisting of a delay line and an inverter (fig. 3.2). To make the generated local clock $lclk$ pausable, an arbitration block is placed in parallel to the delay line. Both the request clock $rclk$ and requests for clock stretching $Ri$ are inputs to this block and it asserts $clkallowed$ only if $Ri$ is low or rises after $rclk^+$. The Muller C [MB59] will withhold the rising of $lclk$ until both of its inputs have risen; thus the active clock edge gets shifted if $Ri$ persists.

Fig. 3.2 depicts the outline for the construction of pausable on-chip clock generation units. The following two sections will detail the implementation of the adjustable delay line and the mechanism of arbitration for more than one incoming $Ri$ signal.

3.2.1 Programmable Delay Line

The propagation delay of the delay line has to be consistently longer than half the critical path inside the corresponding LS module, to meet the synchronous design paradigm. Additionally rising and falling delay should be about the same to generate a clock with a 50% duty cycle. As the drift due
to PTV variations shall closely match the drift inside the LS module, CMOS implementations should employ gates which are more complex than simple inverters and thus do have internal nodes.

To make assembly of the delaylines simple they are divided into a chain of identical slices [TMWR00]. Each slice contains a small delay element which is traversed in forward direction (fig. 3.3) and a second delay that gets traversed by transitions on their way back. The shunt allows for controlling how many stages of the entire line shall be active. In the first inactive slice the switch from fin toward bout gets closed and the forward arc is interrupted. By this we ensure that unused slices don’t consume any switching power. The delay denoted as emergency margin is only active if all delayslices are open in forward direction and it creates a considerable safety margin against totally unexpected variations in timing behaviour of the LS module.

The delayslice dslA (fig. 3.4) implemented according to Taylor et al. [TMWR00] employs an AND-OR gate in the forward arc. By setting the fine-control signal fc the load at the complex gate’s internal node changes significantly and thus the delay is varied. The amount of this effect differs for rising and falling transitions and only slices activated by the coarse-control cc contribute. Transistor level simulations for dslA yield coarse steps of 210ps additional delay per element and fine steps of 8ps and 32ps for rising and falling transitions respectively (fig. 3.4).

Implementing clock generation units for the frequency range between 250MHz and 500MHz with slices of type dslA results in less than 10 slices active. This would lead to gaps in the tunability range (fig. 3.10), because we need at least 11 active slices to cover the whole spectrum with fine steps. The storage and distribution effort needed for the fine control signals makes the implementation dslA even more unattractive.
The implementations dslB through dslD in fig. 3.4 use only simple gates and are controlled by a single signal (cc). The best coverage of the desired speed range is achieved with dslC, which exhibits a stepsize of 125ps, which is even smaller than the gaps in the tunability of a dslA-based implementation. Therefore we chose dslC for the final implementation of our GALS system.

The delayline for the emergency margin can be implemented as simple and area efficient as possible. We opted to do it with a simple chain of two-input AND-gates.

3.2.2 Arbitration

The arbitration block of the clockgeneration units is required to safely arbitrate between incoming requests for clock stretching Ri and rising edges on rclk. This is realized with a row of mutual exclusion (ME) elements, where one input to a particular ME is connected to rclk and the second input
Table 3.1: Stepsizes obtained with different delay slice implementations from fig. 3.4 (Simulated with Cadence Spectre at transistor level)

<table>
<thead>
<tr>
<th>slice type</th>
<th>delaysteps coarse</th>
<th>delaysteps fine (rise/fall)</th>
<th>area per slice</th>
</tr>
</thead>
<tbody>
<tr>
<td>dslA</td>
<td>200ps</td>
<td>8ps / 32ps</td>
<td>144 ( \mu \text{m}^2 )</td>
</tr>
<tr>
<td>dslB</td>
<td>185ps</td>
<td>–</td>
<td>99 ( \mu \text{m}^2 )</td>
</tr>
<tr>
<td>dslC</td>
<td>125ps</td>
<td>–</td>
<td>90 ( \mu \text{m}^2 )</td>
</tr>
<tr>
<td>dslD</td>
<td>155ps</td>
<td>–</td>
<td>90 ( \mu \text{m}^2 )</td>
</tr>
</tbody>
</table>

To the \( Ri \) of the corresponding port controller (fig. 3.5, [TMWR00]). Only after \( rclk+ \) is granted by all ME elements the merging AND-gate issues the clockallowed signal.

As the ME elements are organized in a parallel fashion the given implementation can easily be scaled according to the number of port controllers inside a particular wrapper. This is a considerable advantage with respect to the approach of Yun et al. [YD96], who use an arbiter tree configuration. This tree becomes large and slow for blocks having more than two ports, thus making their approach unsuitable for complex systems.

Additionally the parallel configuration can grant several stretching request simultaneously, thereby allowing an arbitrary number of ports to be transmitting in a single clock cycle. A drawback of the implementation from fig. 3.5 is that it might postpone a stretching request longer than actually needed. To illustrate this assume that one request, e.g. \( Ri1 \), has already stopped the local clock (i.e. \( rclk \) and \( Ai1 \) are high). If \( Ri2+ \) arrives in this situation it will not be granted until the next active clock edge has occurred. This minor performance penalty (at most one cycle) is compensated by the easy scalability of the presented arbiter configuration.

### 3.2.3 Implementation

To employ our clockgeneration unit in a GALS wrapper we have to take some measures for initialization, configuration and observability.

During reset the whole clockgeneration needs to be properly initialized. This may be realized by introducing a two input AND-gate into the
ring-oscillator and connect one input to an active low initialization signal \( \text{ClkinitxRB} \). As the additional delay enlarges the delay offset of the whole clock generation, we merged the AND and the inverter to a NOR-gate with one inverted input (fig. 3.8). For safe initialization of the delay line \( \text{ClkinitxRB} \) should be kept low for the time a transition would need to travel through the whole delay line and the emergency margin delay. (For safety reasons one should refrain from ’tuning’ the initialization time to the actual chosen delay line programming.)

To adjust or measure the nominal frequency of the clock generation unit we implement a free-running mode that is initiated by an external signal \( \text{ClkfreexS} \). The inserted multiplexer bypasses the arbitration block and thus the clock cannot be stretched or stopped. In most cases the resulting free-running clock signal will not be observable outside the chip, due to the frequency limitations imposed by the output pads. Therefore we insert a clock divider which creates a transferable clock signal with 50% duty cycle. Using an asynchronous counter implementation for this purpose keeps the load on the generated clock signal low and allows to safely divide a clock signal which might have a cycle faster than the counter’s settling time. In our case dividing by a factor of 8 suffices to bridge the gap between the fastest possible clock and the minimum pulse width transferable via standard output pads.

The implementation of the clock generation unit uses two building blocks which are common in asynchronous designs but unavailable in commercial standard cell libraries: the mutual-exclusion (ME) element and the

---

**Figure 3.5:** *Arbitration block implementation (shown for a 3 port wrapper)*
Muller-C. Because the ME is the one and only point of arbitration between incoming data and the local clock, its correct behaviour is crucial for the entire system. As the timing behaviour of the ME is critical and should not depend on its placement, we decided to realize it as a standard cell newly designed on layout level using the circuitry proposed by Seitz [Sei80] (fig. 3.6).

Opposed to this, the Muller-C is not exposed to any critical timings in its environment and can easily be implemented using a two-level AND-OR circuit as shown in fig. 3.7. The vulnerable interval during which the feedback outputs are still unsettled is certainly smaller than any feedback paths through delay line or arbitration block. Thus achieving a possibly smaller propagation delay does not outweigh the pain of designing another new standard cell.

For a couple of reasons we want to be able to run a GALS system in a globally synchronous mode controlled by an external clock: Firstly this makes scan test of the entire system possible, and secondly it opens the opportunity to verify the functionality of the locally synchronous blocks.
3.2. **Pausable Clock Generation**

![Diagram of clock generation unit](image)

**Figure 3.8:** Complete circuitry of a clock generation unit for a three-port wrapper

![Waveform diagram](image)

**Figure 3.9:** Waveform of clock generation unit (drawn for a two-port configuration). Bars mark intervals where the delayline is active.
alone. This external clock is inserted with a multiplexer just in front of the \( lclk \) output.

By raising \( Ai \) the clockgeneration unit signals the respective port controller that no active (rising) clock edge will occur on \( lclk \) before \( Ai \) has fallen again. To ensure this relationship despite the delay of the circuitry between the ME element and the \( lclk \) output, (small) delays are inserted just before the \( Ai \) output. They model the delaypath from the ME to \( lclk \), but this constraint is not very strict. We used two 2-input AND gates for this purpose.

The waveforms in fig. 3.9 are drawn to yield detailed insight into the behaviour of the clockgeneration unit. The first request for clock stretching (\( Ri2+ \)) arrives slightly before the local clock is set and gets acknowledged soon after \( rclk \) falls. The request \( Ri2 \) is withdrawn rather fast and thus there is no need to stretch the local clock. The second request persists longer and the active clock-edge is shifted accordingly. The third phase illustrates a case where more than one port needs to be served simultaneously. All requests for clock stretching arriving before \( rclk+ \) and get acknowledge in the same cycle of the local clock. Requests arriving later than \( rclk+ \) will be granted in the following cycle.

The important aspects to keep in mind during specification of port controllers and transfer protocols are:

- \( Ai+ \) can only occur, when \( rclk \) is low.
  
  (Possible conflicts are resolved by the ME element.)

- \( Ai- \) will occur before \( lclk+ \)

Thus \( Ai \) will be in the center of our synchronization mechanism as it transmits information about the status of the local clock signal to the port controllers.

Performing transistor-level simulations of the clockgeneration unit yields an overview over the range of achievable nominal clock frequencies (fig. 3.10).

### 3.3 Port Controllers

A port controller is responsible for managing all data transfers on a particular port of a block in a GALS system. It gets enabled by the LS module
3.3. Port Controllers

Figure 3.10: Clockperiods generated with different adjustable delay line implementations. All units contain 20 delay slices and 20 2-input AND gates as emergency margin. Simulation is performed on transistor level without interconnect backannotation. Grey bars mark the range of fine-tunability of a dslA-based implementation.

and has to synchronize data transmission and local clock phases. The enable line employs a two-phase protocol (transition signalling) while the links between controller and clock-generation and between controller and corresponding module both employ a complete four-phase handshaking (level signalling). Thus it is the controller’s task to translate transition signalling into two coordinated four-phase protocols.

The transfer acknowledge line is the only signal from a port controller entering the locally-synchronous domain. It requires special treatment, that will be explained for each controller implementation separately.

In order to transmit data fast and efficiently, the port controllers need
to act independent from the local clock signal. This is achieved by implementing them as asynchronous finite state machines, using modern synthesis tools for asynchronous control circuitry.

To cover the diverse needs for intermodule communication we defined two families of port controllers:

- **A Poll-type** (P-type) port issues the request for clock stretching exclusively to prevent metastability and thus ensures data correctness. The clock is influenced as scarce as possible.

- **Demand-type** (D-type) ports also ensure data integrity on the transfer channel but add a feature similar to clock gating: As soon as they are enabled they stop the local clock and release it as soon as the required transfer has taken place.

Therefore a D-port is used where a data transfer is immediately needed because the LS module could not carry out any useful computations without the data item being requested. While awaiting the pending exchange, the D-port suspends the local clock, thereby effectively preventing any dynamic power dissipation in the LS module. As soon as a new data item becomes available, the LS module resumes operation directly in phase with the handshake.

A P-port is more appropriate wherever a data transfer is possible but does not necessarily need to happen immediately. The LS module continues to operate normally while the P-port handles the data transfer.

Though most GALS systems might be implemented from D- or P-ports alone, including both families makes it possible to choose between a "sleep while waiting" and a "proceed while waiting" paradigm as mandated by the situation at hand.

The remaining part of this chapter provides a detailed description of the specification, behaviour and implementation of the available port controller library.
3.3. Port Controllers

3.3.1 Demand-Type Input Controller

Asynchronous Finite State Machine

The extended-burst-mode description for the asynchronous controller for a D-input is given in fig. 3.11. After reset the machine is in state 0. A rising edge on the enable line Den triggers the transition to state 1, regardless of the state of the incoming request line, which is defined as a directed don’t care (Rp*). During the state transition the request for clock stretching gets set (Ri+). Proceeding further to state 2 requires the clock stretching to be acknowledged (Ai+) and that Rp+ has occurred. In this case the acknowledge of the port handshake is set (Ap+) and the system finally reaches state 2. As soon as Rp- arrives the local clock can be released again (Ri-). The transfer cycle ends with the falling transitions of both acknowledge lines (Ai-, Ap-).

When arriving in state 4, the FSM is idle again. To adapt for the transition signalling on Den, the states 4 through 0 duplicate the first transfer cycle (states 0 to 4) with only the direction of the Den transition inverted. Although this yields a slightly larger state machine, this effort is compensated by the gain in the date transfer rate.

As mentioned earlier the 3D tool set [YD99] allows a direct synthesis from an extended-burst-mode description to a hazard-free two-level AND-
OR circuit. For the D-input controller this yields:

\[
R_i = R_p R_i + \overline{\text{Den}} Z_0 + \text{Den} \overline{\text{Ap}} Z_0 \\
A_p = R_p A_i + A_i A_p \\
Z_0 = \overline{R_p} Z_0 + \overline{A_i} Z_0 + \text{Den} \overline{R_p} A_p
\]

This set of equations includes the logic for a state variable \(Z_0\) which is automatically assigned by the synthesis algorithm. In this case \(Z_0\) rises concurrently with a toggle on the \(R_i\) output (transition 2 \(\rightarrow\) 3). The external delay path from \(R_i\) to \(A_i\) traverses only a single ME element in the clock generation unit and thus accounts in best case for only about 120ps.

To avoid that the resulting transition creates conflicts with the settling of the machine (i.e. if \(Z_0\) has not settled before \(A_i+\) arrives), we move switching of internal state variables to cycles where \(R_i\) remains stable. This is achieved by adding a (virtual) output \(H_i\) (fig. 3.12) to the extended-burst-mode specification which then replaces \(Z_0\) during state coding. This results in

\[
R_i = R_p R_i + \overline{\text{Den}} H_i + \text{Den} \overline{\text{Ap}} H_i \\
A_p = R_p A_i + A_i A_p \\
H_i = \text{Den} H_i + \overline{R_p} H_i + \overline{A_i} H_i + \text{Den} R_p A_i
\]
3.3. Port Controllers

In this particular case the modified implementation is slightly larger than the original one, but for many asynchronous FSMs hand-coding of state variable can even yield smaller circuitry. This is due to the fact that the 3D tools produce a valid implementation for extended-burst-mode specifications but do not search for an optimal one.

Transfer Acknowledge

The asynchronous port controller provides no information for the LS module whether and when a data transfer has taken place. Therefore a transfer acknowledge signal $T_a$ is needed which indicates a data transfer and can be safely sampled by the local clock. In an earlier GALS proposal [MVK+99] we used a dedicated asynchronous controller, which was tailored for a simpler version of clock generation units and introduced a large overhead. We learned the lesson, that the inclusion of a clock signal into extended-burst-mode specifications is at least painful if possible at all. In most cases a clock signal creates either additional timing constraint for the machine or violates the direction requirement of the don’t cares.

Thus it is advisable to derive the transfer acknowledge from an existing signal, which indicates by its level that data are transmitted. Good candidates for this are $A_p$ and $A_i$. $A_p+$ happens to be in the center of all common validity interpretations of four-phase handshake protocols [Pee96], while $A_i+$ is a prerequisite for $A_p+$ to happen. For it is unknown when the local clock will resume after a transfer, we cannot use these signals directly but need to store their rising transitions in a bistable.
Because of its tight coupling to the clock generation mechanism we use $Ai$ as a starting point and fig. 3.13 depicts an implementation. The $Ta$ flag gets asynchronously set by the $Ai$ pulse and is erased by shifting in '0' with the next rising event of the local clock. As $Ai$- does always occur long before $lclk+$, set removal violations are not an issue.

**Implementation**

The synthesis results for the asynchronous finite state machine are directly transferred to an implementation with standard AND and OR gates. The inverted literals can be generated by inverters, but using gates with inverted inputs speeds up the machine. In cases where this is not possible for all required literals, the fed-back outputs should be preferred in order to minimize the vulnerable interval of the FSM. We did not use complex AO-type gates, because most libraries do not cover all needs and this would result in a heavily imbalanced timing of logic trees inside a machine.

The extended-burst-mode description of the asynchronous controller does not contain any reset mechanism, because we would run into trouble with the unique entry conditions. Since all in- and outputs and internal state variables are defined to be '0' after reset, we can initialize the machine by adding an active-low initialization signal to the AND-plane. I.e. a product with $n$ literals gets implemented by an $n + 1$-input AND, where the additional input is connected to $InitxRB$. The initialization input has to be kept...
active (low), longer than the machine needs to settle in worst case.

Combining the asynchronous controller and the flipflop generating $Ta$ would suffice to control a demand-type port in an asynchronous wrapper. To enhance functional verification and automatic pattern generation for scan test we wanted to include a completely synchronous port controller, substituting the asynchronous FSM during test and verification of the LS blocks. The synchronous input controller uses exactly the same protocols toward the outside and the LS module like the original port, but needs at least three clock cycles for transferring a data item.

The outputs of the synchronous and asynchronous port controllers are multiplexed under control of $ExtClkEnxS$ as sketched in fig. 3.15. Thus by setting $ExtClkEnxS$ a complete block is switched into synchronous mode: the LS module is controlled by the external clock and the interfaces use a strictly synchronous handshake protocol.

The complete implementation of a D-input unit occupies an area of $2340\, \mu m^2$ which equals 86 gate equivalents\(^1\). These numbers include the asynchronous port which accounts for an area of $30\, \mu m^2$ only.

---

\(^1\)Gate equivalents are calculated using the cell area. One gate equivalent equals the area of a NAND gate with standard drive strength. The underlying process has five metal interconnect layers and thus routing does not introduce any area overhead.
Timing Verification

Validating that the implementation of an asynchronous finite state machine meets all timing constraints is a major issue and requires special attention, as it is not supported by current tools. Verification is done in two steps: First all timing constraints for internal correctness of the machine need to be checked. At this stage the environment is assumed to be extremely slow. In a second step the behaviour of the environment must be checked to comply with the fundamental mode requirements of the machine.

In the case of extended-burst-mode machines in an AND-OR implementation the constraints for internal correctness are well defined by Yun et al. [YD99]. They derive two conditions for safe operation:

\[
\begin{align*}
  t_{\text{in} \rightarrow \text{out}} + t_{\text{out} \rightarrow \text{outf}} & > T_{\text{in} \rightarrow \text{lit}} \\
  t_{\text{in} \rightarrow \text{out}} + t_{\text{out} \rightarrow \text{outf}} + t_{\text{outf} \rightarrow \text{prod}} & > T_{\text{in} \rightarrow \text{prod}}
\end{align*}
\]

where \( t_{x \rightarrow y} \) denotes the minimum delay from a transition of type \( x \) to a transition of type \( y \), while \( T_{x \rightarrow y} \) denotes the maximum delay (see fig. 3.16).
Equations 3.1 and 3.2 express that no transition triggered by an input change is allowed to overtake any other transition due to the same input burst until it has passed the AND-plane.

As we do not insert buffers in output feedbacks and assume that isochronic fork conditions are met, we can set $t_{out} \rightarrow t_{out f} = 0$. If a machine can be implemented without explicitly inverting literals we also achieve $T_{in \rightarrow prod} = 0$. In cases where the above timing constraints are not met the machine can be made correct by inserting buffers in some of the feedbacks paths.

The requirements for compliance of the machine with its environment are based on the fundamental mode constraint: As long as the machine is not settled after an input event, no further input events are allowed to occur. Using the notation from above we can express the (worst) cycle time as

$$T_{cycle} = T_{in \rightarrow out} + T_{out \rightarrow out f} + T_{out f \rightarrow prod} = 390ps + 0ps + 220ps$$

This is the time needed by the machine to settle if all maximum timings are triggered and is thus a very conservative estimation. Safe operation of the machine would then require that any feedback path external to the AFSM needs to be slower than the difference between the minimum latency ($t_{in \rightarrow out} = 210ps$) and the cycle time $T_{cycle}$, i.e. 400ps for the D-input controller. To meet such a conservative constraint many of the external feedback paths would require additional delays, e.g. the ME element only has a minimum propagation delay of 110ps. We would sacrifice system performance to meet an unrealistic constraint.

Hence we opted to analyze timing constraints for each individual state transition. Because each transition only triggers a subset of timing paths such an analysis yields less stringent constraints. This strategy complies with Yun's definition of cycle time as 'the delay required to avoid circuit malfunction from the last terminating input burst to the first compulsory input edge of the next input burst' [Yun94]. This implies that it is not mandatory that the whole machine is stable when the next input burst arrives.

The results of the thorough timing analysis from fig. 3.17 are based on gate-level simulations using non-backannotated timing descriptions (pre-layout sdf-files). The difference between minimum latency and cycle time
Figure 3.17: Timing analysis for asynchronous D-input controller

yields the lower bound of the delay of external paths:

$$\Delta = T_{cycle} - t_{in\rightarrow out}$$

For this machine we derive constraints for the external path from $Ap$ to $Rp$ which must not be shorter than 194ps. We need to check this constraint in the analysis of the all types of outputs controllers that may be connected with the D-input port. The delay of the multiplexers at the AFSM’s outputs (fig. 3.15) creates an additional safety margin.

Synchronous Input Controller

To make the description of the D-input unit complete, a brief description of the synchronous input controller is included. The synchronous port is only useful for debugging and testing purposes and does substitute the asynchronous controller if required. Therefore performance issues are rather insignificant during its design and specification.

As the synchronous port controller has no access to the pausable clock generation unit, the difference between D-type and P-type vanishes. We can use one and the same synchronous port for both D- and P-type input units and the enable signal will be labeled with $En$ instead of $Den$ or $Pen$. 
3.3. Port Controllers

The state graph in fig. 3.18 sketches the specification of one half of the synchronous FSM. The second half is obtained by inverting the specification of the enable signal. Again this is due to the use of transition coding on the En line. By using a Medwedjew model for the description the outputs are forced to be glitch free and can thus be safely used in an asynchronous network. In the VHDL coding this property is sustained by performing the state coding by hand.

The controller remains in state idle0 until it samples a rising edge of En. The state of the handshake’s request then determines whether it directly enters fire0 or changes to hot0 where it is waiting for the rising of Rp. In fire0 the acknowledge is set and only after request has gone low the machine enters return0 and sets the transfer acknowledge Ta. By leaving return0 unconditional it is ensured the Ta is sampled high only once per transfer. Data transfers under the control of a synchronous input port consume at least three clock cycles each.

Figure 3.18: Partial specification of synchronous input controller
3.3.2 Demand-Type Output Controller

Asynchronous Finite State Machine

The main differences between a demand-type input and output port are due to the fact, that push-type handshakes get initiated by the sending module. Hence we can specify the asynchronous D-output controller without using directed-don’t cares (3.19). The basic mechanism in translating a two-phase protocol on the enable line to two interwoven four-phase protocols is the same.

Being a demand-type port the controller issues $Ai+$ immediately after being enabled and only after this has been acknowledged the request gets set ($Rp+$). Transitions from state 2 through 4 are the return phase of the first transfer cycle.

The last input transition of this return phase ($Ai-$) is defined as concurrent to the subsequent enabling transition. These two events have different sources (clock generation unit and LS module) and joining them in one input burst definition avoids creating a race condition between them. As a consequence thereof we need to define a separate reset state ($0$) to meet the unique entry condition for state 1.
Figure 3.20: Schematic waveforms and timing analysis of asynchronous D-output controller

Synthesis with 3D yields

\[
\begin{align*}
Rp & = Den \overline{Ai} Ap Ri Z1 + Den \overline{Ai} Ap Ri Z1 \\
Ri & = Ap + Den \overline{Ai} Z0 + Den \overline{Ai} Z0 + Den Ri Z1 + Den Ri Z1 \\
Z0 & = Den Ap + Den Z0 + \overline{Ap} Z0 \\
Z1 & = \overline{Den} Ap + \overline{Den} Z1 + \overline{Ap} Z1 + Den \overline{Ap} Z0
\end{align*}
\]

This contains two additional state variables which are inevitable as nine states need to be encoded and the specification contains only two outputs that can be used for this. The switching of the state variables does not interfere with events on Ri during the main cycle and thus no need to shift them by means of virtual outputs exists.
Transfer Acknowledge

The transfer acknowledge signal is generated using similar circuitry as for the D-input port (fig. 3.13), but the flipflop gets set by $\overline{Ap}$ instead of $\overline{Ai}$. In the D-output specification the $Ap$ exactly fits our needs as it is set only during an interval where the local clock is sure to be inactive, e.g. $Ai = 1$. A second signal that would meet all demands is $Rp$.

Implementation

The transfer from synthesis results to a gate netlist follows the lines explained during the discussion of the D-input asynchronous finite state machine. A slight difference is that AND-gates with two or three inverted inputs were not available and explicit inverters had to be used instead. The imbalance in the $Ri$-tree due to the direct inclusion of the input $Ap$ in the sum was compensated by the AND gate inserted for initialization.

The resulting timing behaviour of the asynchronous finite state machine is sketched in fig. 3.20. The only relevant timing constraint for the environment is, that the arc from $Rp$- to $Ap$- must not be shorter than 123ps. This is easily fulfilled by the minimum latency of all input port controllers.

Like the asynchronous input controller, the output controller is also tied together with a synchronous substitute for verification and test purposes. All inputs are shared and outputs are multiplexed under the control of the $ExtClkEnxS$ signal. Details on the synchronous output port will be given in the next subsection. The complete D-output unit occupies 2448$\mu$m$^2$ (91 gate equivalents) while the asynchronous controller does only account for half of this (1260$\mu$m$^2$, 47 GE).

Synchronous Output Controller

This controller is a rather simple synchronous finite state machine. Again it is described as a Medwedjew-type machine to make the outputs glitch free and thus compatible with a possibly asynchronous environment. Due to the transition signalling on the enable line the FSM specification contains eight states for two consecutive transfer cycles. Each synchronous transfer of a data item takes four cycles of the local clock (fig. 3.21).
3.3.3 Poll-Type Input Controller

Asynchronous Finite State Machine

As mentioned before the main difference between D- and P-type port controller is how clock-stretching is applied. The poll-type ports use their control over clock generation only to ensure data correctness on the transfer channel. This is reflected in the extended-burst-mode description of the P-input controller (fig. 3.22): \( R_i^+ \) is issued only after the port has been enabled and a handshake cycle is pending. Marking the \( R_p \) as directed don’t care avoids a possible race between \( A_i \) and \( R_p \) although a malfunction would have been rather improbable. The remainder of the specification is very similar to the D-type correspondent.

Naming the enable signal as \( P_e n \) and \( D_e n \) for P- and D-type ports respectively yields an additional consistency check. During wrapper assembly the designer can see if he inserts the port types which were assumed during the design of the locally-synchronous module.

Synthesizing a P-input controller according to fig. 3.22, yields

\[
\begin{align*}
    Ap &= R_p A_i R_i \\
    R_i &= R_p R_i + R_p \overline{P_e n} H_i + R_p P_e n \overline{H_i} \\
    H_i &= A_i P_e n + \overline{A_i} H_i + \overline{R_i} H_i
\end{align*}
\]
Switching of internal state variables was again manually moved to uncritical cycles; the resulting implementation is smaller than the original one. This is due to the lack of an optimization step during 3D synthesis.

**Transfer Acknowledge**

The $Ta$ signal is generated with the same circuitry as described for the transfer acknowledge of the D-output port. Using $Ai$ to set the flipflop creates the largest possible margin towards the sampling clock edge.

**Implementation**

Implementing the asynchronous controller with nine standard gates is straightforward and in our case even the missing OR in the $Ap$ logic does not need to be compensated to make the machine’s behaviour correct and safe.

Because synchronous port controllers do not have any access to the clock generation unit, the difference between P-type and D-type behaviour vanishes for them. We can use the same synchronous input controller for the P-input unit as we did for the D-input (see 3.3.1). This results in a P-input unit occupying $2187\mu m^2$ (81 GE) including an asynchronous controller with an area of $648\mu m^2$ (24 GE).
3.3. Port Controllers

3.3.4 Poll-Type Output Controller

Asynchronous Finite State Machine

To give a detailed and complete picture of the implemented set of port units, we include the description of the P-type output controller, although it does not yield actually new insights. In the extended burst mode description (fig. 3.24) the only difference to the D-output is that $Ri+$ and $Rp+$ were interchanged. This reflects the difference between D- and P-type behaviour.

Synthesis with 3D yields:

\[
\begin{align*}
Rp & = \overline{Ai} \overline{Pen} H2 + \overline{Ai} Pen H2 \\
Ri & = Ap \\
H1 & = Ai H1 + \overline{Pen} H1 + Ai \overline{Pen} Ri \\
H2 & = \overline{Ai} H2 + \overline{Ri} H2 + Ai Pen H1
\end{align*}
\]

**Figure 3.23:** Waveforms and timing analysis for asynchronous P-input controller
Figure 3.24: Specifications of asynchronous P-output controller

Again it was necessary to manually shift switching of internal state variables to uncritical cycles to ease the constraints for the environment. The direct path $Ap\rightarrow Ri$ is inevitable and will require some delay balancing when being mapped to a gate netlist.

Transfer Acknowledge

To generate the transfer acknowledge signal $Ta$ the same circuitry as for the D-input controller (fig. 3.13) is attached to the asynchronous finite state machine.

Implementation

As mentioned above we need to insert delay elements to fake the missing logic tree for the $Ap$ signal. Otherwise the timing would be incorrect and the constraints for the environment too stringent. Two AND gate delays are sufficient to meet both requirements.

Again the synchronous controller inside the P-output unit is the same as in the corresponding D-type port (see 3.3.2). The implementation of the entire P-output unit covers a cell area of $2079\mu m^2$ (77 GE) where the asynchronous subblock accounts for far less than half of it ($855\mu m^2$, 32 GE).
Timing analysis of the asynchronous controller part results in a single relevant constraint for its environment: The delay $R_{p+} \rightarrow A_{p+}$ must not be less than 189ns.

### 3.4 Library of Wrapper Components

All port units, the clock generation and some minor additions are gathered in a library of wrapper components – the wrapperlib. Each unit is represented by a structural VHDL description. While the description of the ports is fixed, the clock generation unit is parametrized using VHDL generics. It can thus be easily adjusted to the number of ports inside the wrapper, the desired number of delay slices and elements in the emergency margin.
Table 3.2: Area and timing numbers for port controllers

<table>
<thead>
<tr>
<th></th>
<th>AFSM area [μm²]</th>
<th>average</th>
<th></th>
<th>area [μm²]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$t_{in\rightarrow out}$ [ps]</td>
<td>$T_{cycle}$ [ps]</td>
<td>synch.</td>
</tr>
<tr>
<td>D-input</td>
<td>801</td>
<td>306</td>
<td>350</td>
<td>1710</td>
</tr>
<tr>
<td>D-output</td>
<td>1260</td>
<td>310</td>
<td>342</td>
<td>1368</td>
</tr>
<tr>
<td>P-input</td>
<td>648</td>
<td>198</td>
<td>246</td>
<td>1710</td>
</tr>
<tr>
<td>P-output</td>
<td>855</td>
<td>256</td>
<td>276</td>
<td>1368</td>
</tr>
<tr>
<td>average</td>
<td>891</td>
<td>268</td>
<td>303</td>
<td>1539</td>
</tr>
</tbody>
</table>

To make the wrapperlib complete two macros are added (The need for both will be explained in the subsequent chapter): The first is a parametrized description of a latch register, which gets mainly included for convenience as the wrapper itself is also assembled by writing structural VHDL code. By making the latch register available in the wrapperlib we obtain smaller and better readable code. The second macro is a multi-input Muller-C element. This is required for forked transfer channels.

Table 3.2 wraps up some key figures of the implemented set of port units. All units are quite small (less than 2600μm²) and on average less than 40% of the area is covered by the asynchronous controller. This is mainly due to the enormous area consumption of flipflops compared to simple logic gates. The synchronous controller and the multiplexors can be regarded as overhead introduced to ease test and verification.

Furthermore the asynchronous controllers do exhibit an excellent speed. Without spending any effort on speed optimization (and without having tools for this purpose) we achieve cycle times of less than 350ps corresponding to frequencies above 2.5GHz. In comparing asynchronous with synchronous implementations one should bear in mind that asynchronous circuits react immediately upon changes at their inputs, while a synchronous finite state machine always loses at least fractions of a clock cycle until input changes get recognized. Hence achieving the same performance in a synchronous implementation would require clock frequencies above 2.5GHz.

Taking into account the numbers on area and performance given above and considering the advantages with respect to synchronization, the deci-
sion to implement the port controllers as asynchronous finite state machines seems well justified. This decision is further backed by the progress in the research on asynchronous circuitry and corresponding tools, making synthesis and implementation easy and reliable.
Chapter 4

Data Transfers between Blocks

This chapter will explain how data transfer channels are assembled from wrapperlib components and how these channels are instrumental in transmitting data across clock boundaries without metastability.

The transfer channels presented in this work do all employ a rendezvous scheme, i.e. the transfer takes place at one point in time when both the sender and receiver of data do enable the transmission. This implies that sender and receiver have some means to exchange information about their status and do ensure, that no data item is lost. The extreme opposite of the rendezvous technique would be a broadcast communication with a 'send and forget' scheme.

It was already mentioned that only push channels are used, i.e. the data lines and the signal, which initiates the handshake (here $Rp$), are driven by the same block ('sender'). One can further distinguish between unidirectional and bidirectional channels (fig. 4.1), where the notion of direction corresponds to the dataflow. In unidirectional transfer channels data are transmitted from the sender to the receiver and for a push channel this means that data lines have the same orientation as the request signal. A bidirectional channel consists of two data vectors of opposite direction which are controlled by a common handshake pair. Such bidirectional channels are
Figure 4.1: Illustration of unidirectional and bidirectional transfer channels

e.g. beneficial to implement connections to memories. Assume that block D in fig. 4.1 is a RAM or ROM. The data vector C→D provides the address while D→C delivers the corresponding memory content. In such a case the terms sender/receiver become ambiguous. Block C should better be called the initiator while D has a responding role.1

Informations about the concepts and the detailed implementation of transfer channels will be given in the two subsequent sections, which are named according to the main use of uni- or bidirectional channels. The last section of this chapter will then discuss the performance of transfer channels and compare them with globally-synchronous implementations.

4.1 Transfer Channels between Locally-Synchronous Blocks

A block diagram of an unidirectional channel between two locally synchronous modules, LS1 and LS2, is depicted in fig. 4.2. The lower half contains the port controllers which are connected to the LS modules via the enable and transfer acknowledge signals. Both ports have access to the clock generation of the respective block and receive the local clock to generate the transfer acknowledge. The connection between the port controllers is established via the handshake signals, Ap and Rp. So far everything was already mentioned in the previous chapter.

An addendum are the latches in the data lines that are controlled by the

1Throughout this thesis we confine the term 'bidirectional channel' to this concept.
4.1. Transfer Channels between Locally-Synchronous Blocks

Figure 4.2: Schematic of a unidirectional channel connecting two locally-synchronous modules

handshake acknowledge. They are inserted to decouple the communicating modules as much as possible. In order to transfer data safely across clock domain boundaries, we need a well specified point in time at which the inputs to the receiver are allowed to switch. Because this point in time has to be defined with respect to lclk2, the latches are operated by a signal generated by the receiving port controller.

Data are transmitted and consumed by synchronous circuitry, where they are stored and sampled in clocked elements, i.e. flip-flops. Adding memory to the transfer channel allows the sender to resume operation although the receiving clock has not yet sampled the data. Hence a receiving module whose clock remains stopped for an infinite amount of time will not block the sending one. This alleviates the danger of system deadlocks.

Why is it safe to use Ap as latch control? On the sending side the data are required to remain stable all the time from the port enabling event until the transfer acknowledge gets sampled. The transparent phase of the latches entirely falls into this interval for both P- and D-output ports. On the receiving side the clock must be known inactive while the data lines toggle. Again, this is ensured by the specifications of the port controllers.

Because the data latches introduce a considerable area overhead it is worth discussing possible alternatives:
One might keep the sender's clock paused until data are successfully sampled by lclk2. This sounds easy but requires to broaden the handshake protocol, introduce enable flipflops at the receiver, and diminish the performance of LS1. Since the receiver's clock might remain stopped by an other D-port in its wrapper it can also end up in completely blocking LS1 or even deadlock the system.

Alternatively lclk1 might resume as soon as possible but the transfer acknowledge is retained until data are read. This creates difficulties and overhead for the synchronization of $TaI^+$: To transfer this signal safely, the port controller would have to perform a second synchronization cycle including accesses to the local clock generation unit.

Both alternatives are equally unappealing and the area overhead for the latches seems justified.

The waveforms in fig. 4.3 illustrate a data transfer from a D-output to a P-input port (only a subset of signals is depicted). In the beginning the D-output gets enabled, stops its clock and issues $Rp^+$. At this time the receiving port has not yet been enabled. As soon as this happens it detects the pending handshake, stops its clock and acknowledges the handshake. After
the external handshake has been processed, both ports and their corresponding LS modules may resume their operation.

The grey shaded area marks the transparent phase of the data latches ($Ap = 1$). At the time the latch opens the receiving clock is inactive ($Ai2 = 1$) and remains inactive far longer than the propagation delay of the latch. This ensures that events on the data lines arrive at the receiving flip-flops safely and no metastability can occur. Keeping the sending clock stopped ($Ai1 = 1$) assures that $data1$ do remain stable while the latches are transparent.

4.2 Memory Accesses

In a bidirectional transfer channel one port controller unit manages both the forward and backward data transfer as sketched in fig. 4.4. The asynchronous part of the port unit is a normal D-output controller. This controller is chosen because it keeps the clock of the LS module stopped during the whole handshake cycle (see fig. 3.20).

The port controller used during globally-synchronous operation does not receive the acknowledge signal $Ap$, as this might lead to metastabilities. Accordingly the controller specification differs from the normal synchronous
output port (see page 46) by missing all dependencies on \( Ap \) and skipping the return states. A transfer cycle therefore always consumes two local clock cycles and one cycle has to be longer than the delay of the memory.

Our concept of bidirectional channels assumes that the responding module delivers a level-coded completion signal. This is true for most of today's RAM and ROM macros. In cases where a memory block does not offer any completion detection, e.g. simple combinational look-up tables, a delay model of its longest path can be inserted between \( Rp \) and \( Ap \). We will present an elaborate implementation in chapter 6.

A data exchange on a bidirectional channel is illustrated with the waveforms in fig. 4.5. After being enabled the D-output stops the clock and issues \( Rp^+ \). This passes the address data to the memory. As soon as the memory sends the corresponding data back and signals its completion (\( Ap^+ \)), the port controller initiates the recovery phases of the handshake.

This concept for bidirectional data channels implies that the locally-synchronous module is inactive during memory access and the data are always received with the active clock edge following the \( Ai^+ \) event. If a memory is slow and data are not required in the subsequent clock cycle, one might think of an access scheme with the LS module remaining active during the operation. This can easily be achieved by defining a special memory port controller for this purpose. An alternative solution would substitute the D-type with a P-type output port and change the data input register of the LS module to an enable type (enabled by \( Ta \)). However in this work we had no use for such operation and only implemented the scheme described above.
4.3 Performance Evaluation

To gain insight in the behaviour and performance of the transfer channels numerous configurations were simulated at transistor level. Some illustrating examples are presented in this section. All simulations are performed with data for a 0.25μm CMOS process under typical conditions (2.5V, 27°C).

In fig. 4.6 a unidirectional channel between two D-type ports runs a burst transfer, i.e. the ports are enabled immediately after the previous transfer has

---

**Figure 4.6**: Behaviour of a transfer channel between two D-type ports (Gray shaded areas mark transparency of the data latches)
been acknowledged. Both local clocks have a frequency of 305MHz and are initially out of phase. The first data transfer synchronizes the clocks and their phases remain locked as long as the transfer burst persists. Due to the transition signalling on the port enable line, data items can be transmitted in subsequent cycles. 305 Mi/s is the highest achievable throughput, which is mainly limited by the speed of the sending D-port and the delay between ME-element and the output of the clock generation unit. Usually channel throughput does not limit the performance of a GALS system and burst transfers should be rare (see guidelines for system partitioning in section 5.1). If in a particular situation throughput does create a bottleneck, one could either make the transmitted data word wider and thus reduce the number of transfers or optimize the port controllers for speed. Currently the wrapper concept is more focused on safety and ease of assembly.

In the second example (fig. 4.7) two LS modules with different clock frequencies – 305 and 530 MHz respectively – are connected with two P-ports. Again both modules enable their ports in a burst fashion. Very soon, a recurring pattern on the channel builds up: 4 cycles of the faster receiver’s clock are matched with 3 cycles of the sender’s clock and two transfers are performed during such a period. It is a typical observation that the P-output port does not create any observable clock stretching, because the time span during which these ports require $A_i$ to be high is short and they remain in the initiating role during the center of the transfer. The clock stretching of the receiver has been made more visible by choosing a relatively fast clock for this side.

Another observation is that the duration of the handshake on $R_p$ and $A_p$ does vary considerably. It can be as fast as 1.3ns, but can also last for a couple of nanoseconds. In this example even active clock edges of the sender’s clock ($lclkI$) occur while the latch is transparent. This is no danger, because the transmitted data are required to remain stable until the transfer acknowledge is sampled (see pg. 57) and, if this constraint is met, the transfers are correct.

In our third example (fig. 4.8) a connection from a P-output to a D-input port is shown. The receiving port gets enabled every third local clock cycle while the sender does not enable its port every third and fifth cycle (pattern length of six). Again the P-output controller generates hardly any clock stretching. But the receiver’s clock is frequently stopped. It resumes activity as soon as the incoming data are safely latched in the channel.
These three examples show a large variety of transfer patterns that heavily depend on the speed of the participating modules and how often the ports get enabled. If multi-port wrappers are involved, the resulting pattern will become more complex and will also be influenced by the transactions taking place on the other channels. In general the behaviour of complete globally-asynchronous communication networks might become complex and hard to predict. The implementation in chapter 6 will demonstrate how a well planned token flow scheme creates a deterministic behaviour of the system.
Compared to classical synchronizers the presented transfer channels have the advantage that, although they can connect modules with arbitrary clock frequencies, they introduce no latency to the channel. If data arrive too close to a sampling edge, the clock gets shifted by fractions of a cycle and resumes in phase with incoming data. This mechanism optimizes the relation between clock and data communication in a way, that is not accessible with conventional fixed clocking schemes.

Our concept of GALS communication removes one of the major insufficiencies of the works from Yun [YD96] and Bormann [BC97]. They always
need at least two local clock cycles to transmit a data item. By control-
ling the complete handshake with asynchronous ports and using transition 
signalling for port enabling, the channels presented in this work can transfer 
data items in subsequent clock cycles and thus achieve a significantly higher 
peak throughput.
Seite Leer / Blank leaf
Chapter 5

System Partitioning

This chapter will provide answers to the question 'How shall a globally-synchronous system be transformed into a GALS architecture?'. In a first step the designer has to cut the system in partitions which will become the locally-synchronous modules. Some guidelines for this process will be described in the first section.

The second section is concerned with the organization and operational safety of the asynchronous global communication network. Special attention is given to the issue of possible system deadlocks due to the communication network. A procedure for the analysis of system safety is derived.

5.1 Block Sizes and Boundaries

Partitioning of a system is a creative process, which so far resisted all attempts to automation for general cases. Thus it seems impossible to derive a straightforward algorithm for the transfer of a globally-synchronous system into a GALS architecture. Nevertheless each decision made during partitioning has manifold consequences for performance, area and complexity of the GALS implementation and we will describe and summarize these effects.

The goal of partitioning is to derive a description of the system’s func-
tionality on block level, where each block is as much self contained as possible. This shall reduce communication between blocks to an essential minimum.

**Block Sizes**

Of course the inter-block communication overhead is zero, if the entire system gets squeezed into a single, large block – the conventional, globally-synchronous approach. However this is often impossible due to some upper limits on block size:

- Clock skew does constrain the size of synchronous domains. This can be seen as the ultimate upper bound to block size, as explained in chapter 2.
- If the possible or required clock frequencies inside a block are too far apart, it should be split up. For example if a block has to serve slow off-chip interconnects and also contains performance-critical parts of on-chip data handling, these two issues should be put into separate blocks. Each of these blocks will then be assigned a clock frequency that fits its particular needs.

An excessive fragmentation of the system is disadvantageous due to the overhead introduced by the asynchronous wrappers:

- The area occupied by an asynchronous wrapper can be described as a function of the number of ports $n$:
  \[
  A_{\text{wrapper}} = n \cdot A_{\text{port}} + A_{\text{clkgen}}
  \]
  with
  \[
  A_{\text{port}} = A_{\text{control}} + \#\text{bits} \cdot A_{\text{latch}}
  \]
  \[
  A_{\text{clkgen}} = \#\text{slices} \cdot A_{\text{slice}} + A_{\text{margin}}
  \]

  The parameters values for our implementation are given in tab. 5.1. All equations contain a considerable offset and thus the wrapper overhead increases with number of blocks.

- According to Rent’s rule [LR71, Bak90] and its application to heterogeneous systems [ZDLM00] the number of interconnects grows exponentially with the number of modules. Thus an overly fine granularity of the partitioning is even less attractive.
5.1. Block Sizes and Boundaries

\[ A_{\text{control}} = 84 \text{ GE} \text{ for complete port} \]
\[ = 33 \text{ GE} \text{ for async. port only} \]
\[ A_{\text{latch}} = 4 \text{ GE} \]
\[ A_{\text{slice}} = 3 \text{ GE} \]
\[ A_{\text{margin}} = 40 \text{ GE} \]

*Table 5.1: Parameter values for wrapper area*

**Block Boundaries**

Continuously scalable block sizes are of course an unrealistic assumption. Each system contains some inherent functional granularity. Examples are controllers, datapaths, I/O-interfaces, memories. Cutting such blocks into even smaller modules does often not make sense as the numbers of block interconnects would rise dramatically.

Therefore the designer should position the boundaries in a way that enables a good bundling of block interconnects. As each transfer channel requires a separate port controller, such vectorization pays off in reducing overhead.

Reducing the number of transfer channels in a GALS architecture has an important implication on the organization of datapath controllers. In a globally-synchronous implementation a central control unit for organization of system-wide activity is common practice. Implicit knowledge about the status of each functional block in a particular clock cycle is fundamental in designing such controllers efficiently. This knowledge is lost, as soon as the system is partitioned into separate modules, each one having an individual clock generation.

Hence a distributed control approach is fundamental to achieve an efficient GALS implementation. Each locally-synchronous block is equipped with a controller that organizes its internal operation (fig. 5.1). A well-organized control partitioning will require as few transfer channels as possible for transmitting control information. Practical hints for this are:

- Central control should be transformed into a token passing scheme. This can mean that a data package contains not only the data word but also information about what to do with the data. Sometimes even the
occurrence of a data item suffices as control information.

- The FSMs of central controllers are usually divided into smaller FSMs to reduce complexity and alleviate debugging. These cut lines often yield natural block boundaries.

- If activity patterns of particular parts of the system differ substantially from one another, these parts should be put into separate modules. For example, some functionality might only be used during the start-up of the system and is idle during the remaining operation. Containing it in a separate module offers the chance to put this module into a sleep mode during main operation and thus to reduce dynamic power consumption.

5.2 Communication Configurations

5.2.1 Fork Channels

In chapter 4 unidirectional one-to-one transfer channels were introduced. Although these are sufficient for organizing system-wide information transfers, their use might result in an overly large number of channels. Information transfer via an asynchronous, GALS-compatible bus (multiple senders,
5.2. Communication Configurations

We introduce fork channels as a first step into this direction. A fork channel has a single, fixed source and a fixed set of receivers (fig. 5.2). The implementation\(^1\) is based on the hardware developed for unidirectional channels, supplemented with a Muller-C. The data and request lines are distributed to each receiver. The corresponding acknowledges are bundled by the Muller-C which issues the Ap only after all receivers have granted the transfer. This implies that if one receiver does not react, the whole fork channel remains blocked.

![Figure 5.2: Principle of fork channel organization](image)

5.2.2 Deadlock Analysis

Transferring a system into a GALS implementation should of course not impair operational safety of the system. As the GALS methodology is concerned with global communication in the system, the main danger is malfunction of this asynchronous network. As we use only hardware-defined block connections, the issue of misled data items is excluded. The remaining danger is that the system deadlocks during operation.

Traditionally, deadlocks are a matter of operating system studies and are commonly described in terms of resource allocation [PS83]: a deadlock occurs when each in a set of processes has allocated some resources and is waiting for access to further resources that are held by other processes in

\(^{1}\)It should be mentioned that the simplicity of implementation is mainly due to the use of push channels.
the same set. In general, a circle in the corresponding dependency graph is a necessary condition for a resource allocation deadlock and, as shown by Carlsson [CM97], this is also true for process communication deadlocks. Therefore dependency cycles appear to be a better characteristic than any particular description in terms of communication or resource allocation.

Deadlocks in operating systems are usually detected *dynamically*, i.e. during operation. In the case of process communication, dynamic detection can only serve for discovering the possible existence of a deadlock. Also the description in terms of resources is not suitable for this case, because signals or messages are parts of the computation performed by a process. If process $A$ suspends while trying to receive a message or data item from process $B$, it is not because $B$ currently 'owns' the message (and can possibly be forced to release it), but the message or data have not been computed yet. Nothing else than completing that computation could give process $A$ access to it. If $B$ would not resume until it receives a message from $A$, these two processes are caught in a deadlock. There is no recovery from such a disaster without loosing data or information.

We therefore assert that the only safe way of handling inter-process communication deadlocks in a GALS system is *static* analysis and verification during the design phase. For such an analysis it is assumed that the network of locally-synchronous modules is *synchronous-safe*, i.e. it is insensitive to arbitrary bound latency in the transfer channels (fig. 5.3). This prerequisite ensures that the system does not deadlock due to some (often non-obvious) timing assumptions inside the synchronous controllers. This

---

**Figure 5.3:** *Model for synchronous-safety: delays in transfer channels are arbitrary but bound*
property can partially be checked by simulating the GALS system in the globally-synchronous fallback mode.

In this thesis only rendezvous channels are employed for communication. A channel of this category can be in four different states (fig. 5.4): After power-up neither the sending nor the receiving port are enabled and the channel is said to be *inactive*. If either the sender or the receiver gets enabled, the channel is in *pending* mode and enters the *active* state as soon as the second port gets enabled. Data are only transferred in the active mode and the channel falls back to inactive immediately after transmission has taken place.

The rendezvous scheme is extremely useful as it yields the opportunity to make blocks truly reactive and to introduce synchronization points in the globally-asynchronous network. But the requirement that both sender and receiver have to be enabled in order to make a transfer possible is also the seed for potential deadlocks.

The port enabling pattern of each block is controlled by its synchronous finite state machine and hence a thorough understanding of this behaviour is a prerequisite for an analysis of global communication. The port enabling behaviour can be represented by an abstract model, e.g. a Petri net. Due to the finite number of states of synchronous controllers it is always possible to derive such a model. Usually the enabling patterns are repetitive and the number of variations is small.

Interaction between the block representations is established by port enable tokens: as soon as both the sending and all receiving ports of a channel

**Figure 5.4:** Operation states of rendezvous channels (filled circles represent enabled ports)
are occupied by enable tokens, these annihilate each other and the transfer is known to have taken place.

As the states of each module are finite, the number of possible states for the complete system is also finite. Hence a token-flow simulation of system communication is complete and if successful verifies that the communication behaviour of a synchronous-safe system is deadlock free.
Chapter 6

Realization of a GALS System

To prove our GALS concepts and evaluate its influence on system performance and power we implemented two ASICs, which both implement an advanced crypto-algorithm. One chip is designed as conventional globally-synchronous system, while the second one employs a GALS architecture. Both ASICs were fabricated in the same 0.25μm process and were even in the same production lot. This does allow for a fair evaluation of the effects of GALS architectures.

This chapter will describe the SAFER cryptoalgorithm and how it was implemented both synchronously and in GALS fashion. Two sections will detail on implementation and tool-flow issues. The chapter is concluded with measurements of both design.

6.1 The SAFER Crypto-Algorithm

Secure transmission or storage of sensitive data calls for appropriate measures to prevent unauthorized access to the information content. Storing the data in a locked place or using only dedicated transmission lines is often no option due to organizational or cost reasons. Cryptography approaches the
problem from a different angle: Ciphering shall make the data useless for any unauthorized receiver. Such messages can then also be send over public networks or stored in rather unsafe places.

The continuously growing amount of sensitive electronic data, created a rising demand for cryptography algorithms, that achieve good security with reasonable computational complexity. Furthermore they should be publicly available. This was the background for the development of the SAFER (Secure And Fast Encryption Routine) algorithm by J.Massey [Mas94]. SAFER is a secret-key iterated block cipher that is completely byte oriented. That means that information is ciphered or deciphered in blocks of 8 bytes (64 bits) by applying the algorithm in up to 12 iterations (rounds). Additionally an output or input transformation gets applied (fig 6.1). If encryption and decryption are initiated by different persons, they need to exchange the secret user-key by some secure communication beforehand.

We implemented the currently strongest SAFER version, SK-128 [Mas95a, Mas95b], which employs a Strengthened Key scheduling based on a 128 bit key. The detailed encryption structure of SAFER is given in fig. 6.2. The first layer consists of xor/add operations between data and key bytes. The results are then passed into a nonlinear layer of exponential and logarithmic functions to the base of 45 (modulus 257). The third layer again consists of xor/add operations with a second subkey. The bit shuffling with
6.1. The Safer Crypto-Algorithm

Figure 6.2: Data dependency graph for SAFER encryption with $i \in [1, r]$ as round index
Figure 6.3: Data dependency graph for SAFER decryption with $i \in [1, r]$ as round index.
6.1. The SAFER Crypto-Algorithm

The SAFER algorithm uses pseudo-hadamard transformations (2-PHT) in layers four through six to create the diffusion of the SAFER algorithm. After the final round of encryption, an output transformation consisting of XOR/add operations yields the ciphertext.

Being the reverse of encryption, the decryption structure is also the exact opposite (fig. 6.3). All operations are substituted with their inversion and they are applied in the opposite order.

Ciphering and deciphering with $r$ rounds requires $2r+1$ subkeys with 64 bits each. These are derived from the 128 bit user key through a sequence of byte and bit rotations and by adding a fixed bias key. We do omit a detailed description of the key scheduling algorithm; it can be found in [Mas95b]. The subkeys are static during the actual cipher operation and should therefore be computed at the very beginning, immediately after the user key is available.

When using a block ciphering algorithm each block of plaintext is translated into a certain block of ciphertext. If the same plaintext block occurs
more than once in a message, it will always be assigned the same cipher-word. As there are numerous situations where this is disadvantageous, block ciphers are employed in four standardized operational modes [ISO97]:

**Electronic code book (ECB):** This simple mode represents the application of the plain block cipher algorithm.

**Cipher block chaining (CBC):** Each incoming block is xored with the ciphertext of the preceding block. For the generation of the first block's ciphertext a user-defined initialization vector (IV) is used.

**Cipher feedback (CFB):** The ciphering block acts as a pseudo-random number generator with initial value IV. Incoming data are simply xored with this number sequence. The encrypted block is then also fed back into the cipher engine.

**Figure 6.5: Dataflow in CFB and OFB mode**
6.1. The Safer Crypto-Algorithm

Figure 6.6: Block schematic of synchronous cryptochip MERLIN (only some important data lines depicted)

Output feedback (OFB): Again the ciphering block delivers pseudo-random numbers. In contrast to the CFB operation their sequence does only depend on the initialization vector.

Our ASIC implementations provide the SAFER SK-128 in all four configurations for both encryption and decryption (in contrast to [MVF00]). The actually required mode is configured during startup and kept constant during the cipher operation.

Globally-Synchronous Implementation

The conventional synchronous SAFER chip was implemented by Marc Oberholzer and David Studer [OS00], who developed the block diagram depicted in fig. 6.6. This design is divided in two main parts: One receives the key and modus information, prepares all the subkeys and stores them in a RAM. The other part performs the actual cipher operation, with the SAFER datapath as the dominant element.

In an initial phase only the key preparation part is active. After computing all subkeys and storing them in the RAM, this part has no further purpose. Correspondingly it is switched off by a clockgate.
As soon as the crypto part receives the signal that the subkeys are ready, it starts with ciphering or deciphering. The two mux-blocks contain the external XORs and are responsible for managing the data flow according to the chosen mode of cipher operation. The SAFER datapath has access to two memory blocks: The RAM to retrieve the subkeys and a 64 bit wide look-up-table (ROM) for the results of the exp- and log-operations. For sake of clarity only the most important data connections are depicted in fig. 6.6. Actually there are numerous control and status signals across block and part boundaries.

6.2 GALS Partitioning

The partitioning of the GALS counterpart bases on the organization of the MERLIN design (fig. 6.6), which reflects the inherent functional groups in the system. The task is thus to decide which blocks should be grouped to LS modules and how the communication network between them should look like. Both tasks are tightly coupled with the organization of the controllers in the LS modules.

We propose the scheme shown in fig. 6.7 and will explain its purpose and behaviour block by block:

The **controller** block receives user key, modus information (de- or encryption, number of rounds, and operation modus) and if necessary the initialization vector from the outside. These informations are then distributed to other blocks. The key preparation block gets the user key, while the modus information is passed to both mux blocks and the datapath. Not all blocks do receive all data lines of this fork channel, but all participate in the same handshake. The initialization vector is distributed from the controller to the asynchronous FIFO and muxl. Here the order of distribution is important: The transmission of the IV is required to be finished before the distribution of the modus information starts. This has implications for the behaviour of mux1.

After the controller has delivered all data to the respective blocks, there remains nothing useful to do. By enabling the D-port at the external input the controller halts itself during cipher operation.

Right after the reset procedure the **key preparation** block enables the
D-type input receiving the user key. It thus remains halted until it receives this information. With the user-key available, the block starts computing the subkeys and stores them in the RAM. During this procedure the necessary bias-key values are read from a look-up-table, which is connected through a (bidirectional) memory channel. The key preparation block does not receive the modus information and thus cannot tailor the number of generated subkeys to the number of rounds actually used. It thus always computes the maximum key field of 25 subkeys. As soon as all keys are successfully generated, a keys-ready signal is sent to the SAFER datapath. This keys ready channel does not contain any data signals, but merely sets a flag.

The subkey RAM consists of a layout macro provided by the foundry. It already provides an asynchronous protocol, but cannot connect to two master blocks. A simple interfacing AFSM provides multiplexers and conversion logic to address this. The keys-ready channel assures that either the key preparation block or the SAFER datapath governs the RAM and that access conflicts are excluded.

The mux1 block handles both the internal data flow organization and the external data input. In the synchronous design the I/O-operations, which are managed speed-independently by a four phase handshake, were put into sep-
arate blocks to simplify controller specification. In the GALS architecture this task is completely handled by the port controller in the asynchronous wrapper.

After the reset sequence the mux1 LS module enables the ports for both modus and IV reception and will resume operation only after the arrival of modus information. A possible initialization vector will be transmitted by the controller block prior to the modus. This makes sure that, at the time mux1 detects that it requires an IV (CBC encryption, CFB, OFB), its value is already available at the input latches. If the operational mode does not necessitate an initialization vector, the P-input port does not block further operation.\footnote{Here a design flaw sneaked in: In the actual implementation only the P-port is enabled after reset and the controller always transmits an IV value (either dummy or real). The D-port is not enabled until this item has been received} During ciphering, all D-ports that handle mandatory transfers, get enabled simultaneously and hence the LS module always resumes in phase with the transfers and consumes as little power as possible.

The actual cipher engine sits in the SAFER datapath block. It has access to the subkeys stored in the RAM and to the exp/log values of the ROM. The use of a P-port for the read connection to the RAM demands some explanation: According to the worst-case specifications of the RAM, a read access can account for up to three cycles of the datapath clock. But in our particular case the memory addresses are known far in advance (they are determined by the order of needed subkeys). By using a P-port we can issue the address and the results will then be synchronized under the control of the port controller. This scheme gives us access to the actual RAM performance and we do not need to schedule for the worst case.

Dividing the SAFER computations into separate LS modules or pipelining it internally, would yield no gain as the operational modes of blocks ciphers (especially the feedback modes) are hardly amenable to such transformations. The entire computations act on a single 64 bit register and a split of the datapath would also increase this overhead.

The mux2 block is rather similar to the mux1 in controlling the backend dataflow and the output operations.

The CBC register of the synchronous implementation is substituted by a 4x64 bit dual-input asynchronous FIFO. It serves as feedforward buffer for the CBC decryption and gets initialized with the IV value. Similar to the
Figure 6.8: Activity patterns of the SAFER crypto-system in GALS architecture (gate-level simulation with 12 rounds encryption in CBC mode)
situation at the RAM, conflicts between the two inputs are suppressed by the system-wide data flow and thus an explicit local resolution can be omitted.

All asynchronous wrappers in our implementation are dominated by D-type ports. Firstly this employs clock stretching/gating extensively and yields a low power consumption. Secondly it coordinates the clocks of communicating modules and fosters overall performance. Both effects can be observed in the activity patterns of the system. In fig. 6.8 each LS module is represented by its clock and each memory access or I/O channel by one of its handshake lines.

The very first clock pulse after reset appears in each LS module. During this cycle the appropriate ports get enabled. Afterwards the occurrence of clock pulses is a good indicator for useful computation inside the module. In our case most of the modules are halted for a great deal of the time.

In situations where two modules communicate in a burst mode fashion their clocks behave coordinate during the transfer sequence. An example is the clock of the key preparation block: During the transfer of the user key the clock is locked to the controller’s clock. Then the computation of parity bytes and the first shuffling rounds are performed at the nominal frequency of the block, which is much faster, until the bias keys are needed. During the transfer sequence with the bias ROM the key preparation clock is locked to the (slower) ROM performance.

A similar effect can be observed during each cipher-round in the datapath. The exp/log ROM is (configured to be) slower than the datapath’s clock and the particular clock cycle containing the ROM access gets stretched to adapt to this.

Keeping in mind that this is mainly an evaluation design for the GALS methodology, our partitioning yields modules which considerably differ with respect to their complexity and size. The block boundaries mainly follow functional borders derived from the synchronous implementation. To reduce the wrapper overhead, larger blocks might be advisable, but this would in turn eliminate some of the interesting tasks for the globally-asynchronous network.
6.3 Testing Issues

Testability of asynchronous circuitry is a topic of current academic and industrial research. Most of the problems in this field are due to the logic redundancy, which is intended to suppress hazards, and the necessity of fine-grained delay testing. This thesis is not intended to yield an exhaustive coverage of the testability of GALS systems. But we will show that, although large parts of the asynchronous network remain currently untested, the scan test for the synchronous modules is not hampered.

In order to make a synchronous (sub)system scan testable with an appropriate coverage, we need access to all of its memory elements (flipflops) and to all functional input and output pins. The access to the flipflops is facilitated by grouping them all into one large shift register. Beginning and end of this scan chain are connected to the outside as input and output pins.

Connecting all flipflops of an LS module to a shift register in test mode and knitting these together to one or several systemwide scan chains is straightforward. The crucial point is how we get access to all functional input and output pins of a locally-synchronous module that is surrounded by untested asynchronous circuitry. Here the implemented synchronous fall-back mode is instrumental. During globally-synchronous operation the transfer channels are controlled by the synchronous ports and the latches in the data lines are kept transparent. The data inputs to a particular module are thus driven by the output register of the corresponding LS module and
vice versa the outputs can be observed at the input register of the receiving block. By making the synchronous port controller scan controlled as well, we also have access to the signals between port and LS module (Den/Pen, Ta). A schematic of the proposed scan test organization is given in fig. 6.9.

Thus we have complete test access to all synchronous circuitry (LS modules and synchronous port controllers) and can apply standard scan test and pattern generation tools. Balancing the distribution of the external clock is needed to prevent hold time conflicts at the point where the scan chain crosses block boundaries. What remains untested so far are the asynchronous components inside the wrappers: the clock generation and the asynchronous part of the port controllers. The testability of these components is considered future research.

6.4 Implementation and Current Design-Flow

A main goal of implementing the SAFER crypto-system twice — in a conventional and in a GALS architecture — was to provide a comparison between both methodologies. To keep this contest as fair as possible we decided to use the same micro-architectures in both designs. The re-use of great parts of the VHDL code from the synchronous design also saved a considerable amount of time during the design process.

Soon after starting the design process it became clear, that both chips will be pad-limited. This gave us the freedom to focus more on performance issues than on area. As throughput during the actual cipher operation is mainly constrained by the computations in the SAFER datapath, the efficiency of this block had priority. The allocation of operations to clock cycles in a cipher round is depicted by the dashed boxes in figures 6.2 and 6.3: One round is performed in six clock cycles and the datapath contains eight 8 bit add/sub-units in parallel. The consequences for the external memory blocks are also noticeable: The exp/log ROM block has an address and data width of 64 bit each. Concerning the RAM we might get into trouble if we expect a 64 bit wide block to deliver data in at most 3 clock cycles. Thus the sub-keys are stored in pairs, which are then 128 bit long and only one memory access per cipher round is required.

For both designs we use the same RAM, provided by the foundry as a layout macro. The so-called ROMs are implemented with synthesized look-
6.4. Implementation and Current Design-Flow

up tables. This opens the chance to trade their speed and area to meet the actual requirements during synthesis. In the GALS architecture the ROMs are separate blocks that communicate via bidirectional transfer channels. To provide a handshake interface, the propagation delay inside the combinatorial logic is modeled with an adjustable delay line between request \( R_p \) and acknowledge signal \( A_p \) (see fig. 6.10). Because the sender’s clock is known to be stopped until it detects \( A_p \), the delay \( t_{ad} \) can be as small as half of the propagation delay inside the look-up table. The latches drawn in fig. 6.10 belong to the construct of the bidirectional channel. Equipped with this simple interface circuitry the look-up table perfectly fits into its asynchronous environment.

**Design Data**

Both ASICs are implemented in a 0.25\( \mu \)m, 5-metal interconnect CMOS process. Because the effective core utilization is well below 90%, there is no routing overhead and the area numbers specify the area covered by standard cells.

In the synchronous MERLIN chip the crypto-system has a size of 1 232 590\( \mu \)m\(^2\). The whole system consists of one clock domain and post-layout timing analysis yields a minimum cycle time of 3.6ns (280MHz). Speed is limited by a path through one of the datapath adders.

The MARILYN design contains the same crypto-engine in GALS architecture. The core covers an area of 1 560 100\( \mu \)m\(^2\) and an overview over speed and area of each block is given in tab. 6.1. All LS modules together are 12% larger than the synchronous core, which is mainly due to tighter synthesis constraints for some of the modules.
Because the same micro-architectures are used in both design, the longest path of the synchronous design is encountered in the datapath block. Therefore significantly better performance would require a complete re-scheduling of the datapath computation. Nonetheless all other blocks were optimized for speed to demonstrate their performance potential.

The asynchronous wrappers introduce an area overhead of 13%. Some of this is due to overly long delay lines in all of the clockgeneration units. For safety reasons each delay line contains 30 to 40 slices and a safety margin of 30 AND gates. Reducing these numbers to the actually needed values would reduce the wrapper overhead to approximately 11% percent. The last column of table 6.1 illustrates, that the area of the synchronous port controllers accounts for 20% of wrapper area. This is the price we pay for enhanced testability.

<table>
<thead>
<tr>
<th></th>
<th>nominal cycle time</th>
<th>(sync.) module [μm²]</th>
<th>wrapper area w/ sync [μm²]</th>
<th>w/o sync [μm²]</th>
</tr>
</thead>
<tbody>
<tr>
<td>controller</td>
<td>2.1ns</td>
<td>38 457</td>
<td>18 315</td>
<td>12 501</td>
</tr>
<tr>
<td>key prep</td>
<td>3.4ns</td>
<td>250 299</td>
<td>18 531</td>
<td>13 059</td>
</tr>
<tr>
<td>bias ROM</td>
<td>1.2ns</td>
<td>32 805</td>
<td>4 104</td>
<td>4 104</td>
</tr>
<tr>
<td>datapath</td>
<td>3.3ns</td>
<td>214 956</td>
<td>37 863</td>
<td>28 629</td>
</tr>
<tr>
<td>mux1</td>
<td>2.3ns</td>
<td>150 831</td>
<td>39 204</td>
<td>29 628</td>
</tr>
<tr>
<td>mux2</td>
<td>2.2ns</td>
<td>126 747</td>
<td>34 227</td>
<td>26 361</td>
</tr>
<tr>
<td>exp/log ROM</td>
<td>1.7ns</td>
<td>291 672</td>
<td>12 708</td>
<td>12 708</td>
</tr>
<tr>
<td>subkey RAM</td>
<td>—</td>
<td>237 415</td>
<td>17 784</td>
<td>17 784</td>
</tr>
<tr>
<td>async FIFO</td>
<td>—</td>
<td>34 182</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>area</td>
<td></td>
<td>1 377 364</td>
<td>182 736</td>
<td>144 774</td>
</tr>
</tbody>
</table>

Table 6.1: Speed and area breakdown of the GALS crypto-system. The wrapper area is given for implementations with and without the synchronous port controllers.
6.4. Implementation and Current Design-Flow

**Used Design Flow**

Implementing a globally-asynchronous locally-synchronous design is a challenge, because existing commercial tools are exclusively focused on the synchronous style. While the remainder of this section will briefly describe how we arrived at an implementation despite the existing tools, the subsequent section will outline a path towards GALS-compliant tool flows.

We encountered most of the problems with the currently available tools in the backend-part of the flow:

- Asynchronous units are sensitive to changes in their timing behaviour. They need a thorough timing analysis during all stages of the design flow, but none of the (so called) timing-driven tools does understand these special constraints. To prevent changes in the well defined structure of asynchronous circuitry, we have to ensure that none of the tools might change or 'optimize' them.

- Like most of its competitors, the static timing analyzer available to us (Cadence Pearl) does not deliver the delay of specific arbitrary paths inside combinational logic. Although it has to calculate the values of each timing arc in the system internally, it only reports paths which start or end at clocked elements and I/O pins. Hence even if we could live with timing-unaware placement and routing, we would have no means for a static timing check on the prepared layout.

- Due to the relevant influence of routing on circuit behaviour, the locally-synchronous modules demand a timing-driven flow, in order to exploit their performance. The timing constraints are usually described with gcf (global constraint format) files. Although the gcf standard would allow for constraints, which are confined to a certain sub-block of the system, this feature is currently not supported by tools. This results in constraints that propagate across module boundaries into asynchronous units. Running a constraint-based rebuffering would then change the internal timing of asynchronous state machines.

With these effects in mind we decided to prepare the layout of synchronous and asynchronous parts separately and put them together as fixed macros. This is a hierarchical approach to layouting, which is also rather
Chapter 6. Realization of a GALS System

Figure 6.11: Design-flow used for MARILYN design
poorly supported by current tools. By deciding whether the block under design requires a timing-driven approach (i.e. the LS modules) or must not be changed with respect to the underlying netlist, the procedure in the first layer of the hierarchy is chosen.

Generating a layout macro for each asynchronous controller has the additional advantage, that we need to check the timing of the block only once. Performing the timing verification of the asynchronous unit stand-alone, i.e. as the topmost level of hierarchy, allows to extract the required information about particular internal timing arcs and thus to verify internal correctness.

The complete tool-flow used during the MARILYN design is sketched in fig. 6.11. Only the most important steps are depicted and most of the checks and iterations are omitted.

The design of the asynchronous units is based on the extended-burst-mode specifications, which are then synthesized with the 3D tool set. The generated logic equations are translated into a structural description of the two-level AND-OR implementation. The LS modules are described with conventional register-transfer VHDL code. Combined with a block-level description of the complete system, this VHDL code is then synthesized to a hierarchical gate-level netlist. All pre-layout functional and timing checks are performed on this netlist.

Each LS module or asynchronous unit is then layouted separately. The LS modules are equipped with balanced clock-trees and rebuffered, while the asynchronous units are merely floorplanned, placed and routed. Afterwards each block is represented by an abstract (LEF, library exchange format) and a detailed view (DEF, design exchange format). In our case we arrived at a library of 16 prelayouted macros.

Only their abstract view is then used during the generation of the top-level layout. Each macro instance needs to be placed manually and only a few remaining standard cells are inserted by the automated placement.

The top-level DEF and the block DEFs are then merged to generate a complete detailed layout. To allow for parasitic extraction and correct back-annotation this database has to be flattened. At this point we return to a normal flow that is concluded with a number of checks, chip finishing procedures and generation of mask data.

As a consequence of the particular necessities of a GALS implementation, we missed out much of the potential of current backend-tools. For
example one can easily arrive at a block level layout and routing, that obstructs any reasonable top-level routing. In a completely flat design these optimizations and iterations would be performed by the tool itself and not by error-prone designer interactions.

6.5 A GALS-Compliant Tool-Flow

During the implementation of the GALS design Marilyn we had to struggle with numerous, partially unexpected tool problems. For a proof-of-concept implementation it was worth the effort, but for mainstream designs, facing tight schedules, it might be a severe obstacle for using GALS. In this section some approaches to a GALS-compliant tool-flow will be described.

The least changes in tools are required if all asynchronous units, including the clock generation, are provided as new standard cells. The units can be optimized at layout level and are then frozen\(^2\). The crucial point will not be the generation of the layout, but the description of the timing requirements at the cell's interface. Existing formats for timing description are focussed either on purely combinatorial or on clocked memory cells. For an asynchronous unit one would need to find a way to describe timing constraints relative to other pins, which are not labelled as clock pins. We already experienced some difficulties describing the timing behaviour of the ME-element and it will need thorough investigations on timing formats and the way they are used by tools to find solutions. If such a solution is possible, the backend design can then be performed on a flat netlist, following mainstream methodologies.

A different approach relies on progress in tool development. With the emergence of complex mixed-signal SOC's (Systems On a Chip) there is wide demand for flows supporting hierarchical layout generation. As soon as such tools are really operational, one could follow the flow depicted in figure 6.11 much more easily. Successful application of timing-based optimization techniques (tree generation, rebuffering) would require that block-level constraining gets supported by the corresponding tools. This would open the complete optimization space to the algorithms, without the danger of destroying or disturbing the asynchronous units.

---

\(^2\)Most placement tools are able to handle cells with multiple row height, although the number of different heights is often limited. E.g. Silicon Ensemble allows two different heights.
We believe that a well qualified design-flow will give the industrial designer access to the whole potential of GALS methodologies. He can get rid of most of his synchronization and interface problems, which will be handled by the asynchronous wrapper. It is important that these gains do clearly outperform the possible effort needed to implement a change in the design-flow. Thus a flow simplification is important to transfer the GALS methodology to industrial use, but we assume such tool-centered issues outside the scope of this thesis and perhaps academic research in general.

6.6 Performance and Power Measurements

This section will describe measured results of the two realized ASICs, both implementing the same SAFER crypto-system. First the globally-asynchronous locally-synchronous chip MARILYN will be presented. Since the conventional globally-synchronous implementation (MERLIN) is merely used as a reference for the evaluation of the GALS architecture, we will discuss the corresponding results rather briefly.

6.6.1 GALS Implementation (MARILYN)

Figure 6.12 shows a chip photograph of MARILYN. The design has 66 pads that define the die area. In the upper left corner the subkey RAM is visible and right to it two rows of small testpads. These provide access to separate asynchronous finite state machines for test and debugging and were included after it became apparent that the overall core utilization falls below 70%. The rest of the core area is occupied by the standard cells of the SAFER crypto-engine. Virtually only structures on the top metallizations are visible in the photograph.

We received 25 packaged samples of the MARILYN design and 21 of these are operating correct under all cipher modes with arbitrary data and moderate clock speed. Since cipher-algorithms are constructed to yield a maximum diffusion among data bits, functional test patterns already result in a high fault coverage [Bon93]. Therefore extensive functional tests are an excellent indicator for correctness of both the design and the processing. Although the mentioned functional yield of 84% seems rather low, it still exceeds the yield of the synchronous counterpart, which has been processed
A next task was to find the fastest operational set of clock configurations. By adjusting only one clock at a time, while all others are kept at an overly slow frequency, we can exclude that locking effects between communicating modules are a limiting factor. For the MARILYN design we found that all delay lines, except the ones in the SAFER datapath (6 active slices) and the key preparation block (4 active slices), can be set to the minimum cycle time (all slices inactive). That the local clock speed is limited by the clock generators offset, is mainly due to the unexpected strong influence of routing parasitics.
### Table 6.2: Performance data for MARILYN

<table>
<thead>
<tr>
<th></th>
<th>min. item cycle [ns]</th>
<th>max. throughput</th>
<th>Δ to MERLIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECB enc. 0 rounds</td>
<td>82.1</td>
<td>780Mbit/s</td>
<td>+446Mbit/s</td>
</tr>
<tr>
<td>1 rounds</td>
<td>84.4</td>
<td>758</td>
<td>+133</td>
</tr>
<tr>
<td>5 rounds</td>
<td>151.1</td>
<td>423</td>
<td>-147</td>
</tr>
<tr>
<td>10 rounds</td>
<td>275.8</td>
<td>232</td>
<td>-71</td>
</tr>
<tr>
<td>12 rounds</td>
<td>329.6</td>
<td>194</td>
<td>-61</td>
</tr>
<tr>
<td>CBC dec. 0 rounds</td>
<td>83.1</td>
<td>763Mbit/s</td>
<td>+429Mbit/s</td>
</tr>
<tr>
<td>1 rounds</td>
<td>85.5</td>
<td>749</td>
<td>+124</td>
</tr>
<tr>
<td>5 rounds</td>
<td>151.1</td>
<td>423</td>
<td>-147</td>
</tr>
<tr>
<td>10 rounds</td>
<td>282.0</td>
<td>227</td>
<td>-76</td>
</tr>
<tr>
<td>12 rounds</td>
<td>334.7</td>
<td>191</td>
<td>-64</td>
</tr>
</tbody>
</table>

key preparation time: 2612.7ns

The data throughput of Marilyn is described with the minimum data cycle time, i.e. the minimum time span between two consecutive 64-bit cipher blocks leaving the system. We measured the time between the first $R_{p+}$ transition of the respective output bursts. To determine the data cycle time correctly, one has to pay attention that the data input speed is sufficiently high, so that the system-internal pipeline never stalls. Since all tests and measurements are performed with standard digital test equipment (in our case a 660MHz HP83000 tester), which is inherently synchronous, a careful adjustment of test patterns is demanded. Exploiting the optimum performance requires an adaption to the elastic timing behaviour of each individual sample.

The corresponding results for ECB encryption and CBC decryption are depicted in fig. 6.13 and tab. 6.2. Additionally they are compared to the re-

---

3 Later we could trace back this result to a violation of the synchronous-safety requirement, i.e. the FSM inside the datapath’s LS module was not implemented accordingly.
Figure 6.13: Performance of Marilyn and Merlin described by the smallest time between the release of two consecutive 64 bit cipherwords (min. item cycle time)

sults of the conventional synchronous design Merlin. The measurements of the GALS implementation show a smooth relationship between the number of rounds and the item cycle time. For very small round numbers the time needed to transmit data between internal blocks becomes dominant while the straight increase between 5 and 12 rounds directly reflects the operation of the datapath block. It is typical for asynchronous circuits that each sample exhibits a slightly different timing behaviour. But the performance measurements of a number of Marilyn samples yield results which differ by less than 2% and are therefore represented by a typical value.

Comparing the Marilyn throughput with the Merlin results, it is apparent that the Merlin design team had a different concept for implementing the modus with zero cipher rounds (feedthrough). They propagate each 16-bit input word individually to the output and thus need many cycles to complete a feedthrough of 64 bits. When it comes to the range of 10 to 12
cipher rounds, which are relevant for operation, the MARILYN throughput is 23% lower than that of MERLIN. This result can be traced back to the already mentioned too slow SAFER datapath in the GALS implementation.

The power consumption was measured with a test pattern that applies a group of 10 arbitrary data words in an infinite loop. When reaching the loop’s end the tester hardware needs some time to restart the loop. As the time required for this jump depends on the particular settings, the fraction of actual activity during power measurements should be determined individually for each setup. During the MARILYN power measurements we found that about 7% of the time was consumed by waiting for the jumps to complete.

The numbers given in table 6.3 represent the dynamic power consumption of MARILYN, i.e. the leakage current of the core (0.5mA) has already been subtracted. A fair point for comparisons is the amount of energy dissipated during ciphering or deciphering of one Mbit of data. The first two rows describe MARILYN’s energy consumption in the usual GALS mode, while the lower half of the table gives the corresponding numbers for operation in the globally-synchronous fall-back and test configuration. The GALS operation consumes only one third of energy compared to the globally-synchronous operation and most of this difference is due to the very adaptive

<table>
<thead>
<tr>
<th>globally sync.</th>
<th>current core [mA]</th>
<th>pads data throughput</th>
<th>energy per Mbit</th>
<th>Δ to MERLIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECB enc. 10 rounds</td>
<td>23.5</td>
<td>14.2</td>
<td>103 Mbit/s</td>
<td>555nJ</td>
</tr>
<tr>
<td>ECB enc. 10 rounds</td>
<td>24.4</td>
<td>13.9</td>
<td>103 Mbit/s</td>
<td>577nJ</td>
</tr>
<tr>
<td>ECB enc. 10 rounds</td>
<td>53.2</td>
<td>8.8</td>
<td>71 Mbit/s</td>
<td>1 857nJ</td>
</tr>
<tr>
<td>ECB enc. 10 rounds</td>
<td>53.7</td>
<td>8.6</td>
<td>72 Mbit/s</td>
<td>1 845nJ</td>
</tr>
</tbody>
</table>

**Table 6.3:** Data for MARILYN’s dynamic power dissipation (measured at 2.5V, 20°C). Numbers for both GALS and globally-synchronous modes are compared to the MERLIN results with and without clockgating respectively.
and selective clocking in the GALS architecture.

Comparing the numbers from the GALS operation with those from the conventional synchronous implementation (with clock-gating) the difference becomes smaller, but MERLIN still dissipates over 30% more energy than the GALS system. The comparison between MARILYN in globally-synchronous mode and MERLIN with disabled clock-gates, is just included for completeness. Both modes are not relevant for operation.

**Programmable Clockgeneration Units**

Generating clocks for synchronous circuitry on chip is at least unfamiliar to synchronous designers. Therefore we pay special attention to the characterization of the clocks generators. External measurements are made possible by the inclusion of a clock divider in each clockgeneration unit. The pulses of these divided clocks are sufficiently long to propagate through the output pads and thus can be measured accurately.
6.6. Performance and Power Measurements

The jitter measurement in fig. 6.14 gives an impression of the quality of the clock signals. Recording is triggered 21µs (1000 cycles) before the measurement window and the last 1000 curves are drawn overlayed. Despite all this an accurate clock waveform gets depicted. A peak-to-peak measurement of the 1000 periods jitter yields a value of 3.6ns at a 21.1ns cycle. Dividing this relation by the number of clock cycles that passed since the trigger event, results in a worst case cycle-to-cycle jitter of $1.7 \cdot 10^{-4}$. Keeping in mind that the delay lines are placed by standard place and route algorithms such a low jitter is astonishing and well acceptable for the design of high-speed synchronous subsystems. Conventional extraction and timing data for some on-chip ring-oscillators
verification tools already yield an uncertainty during the design phase which is orders of magnitude larger.

An important property of our clock generation units is the programmability of the ring-oscillator's delay-line. Fig. 6.15 shows the calibration curves and data for three on-chip ring-oscillators. The two larger ones (datapath and key preparation) are actually clock generation units, while the smallest delay line is part of a look-up table interface and is switched to a feedback mode only for measurement. This difference in the circuitry surrounding the delay line gets reflected in the different offsets for zero active slices.

All curves show a good linearity and a monotonous growth of cycle times versus the number of active slices over the whole tuning range. The delay per active slice is extracted from the curves and varies between 347 and 376ps/slice. The variation of these values correlates with the size of the delay line: the longer the delay line, the larger is the delay increment. This can be explained with the greater average interconnect length in larger clock generation units. The interconnect parasitics are also the major reason for the differences between expected and measured values of the delay increment per slice. The measurements yield more than twice the delay we expected from transistor level simulations (125ps/slice; see tab. 3.1).

6.6.2 Synchronous Reference (MERLIN)

At a first glance the conventional synchronous SAFER chip (fig. 6.16) looks rather similar to the GALS system. It has just 6 pads less and the core is optically dominated by the RAM in the upper left corner. The visible dots all over the die area are metal tiles which ensure surface planarity during processing. These tiles are automatically inserted during chip finishing. Until MARILYN was ready to be finished we had already learned when and how these tiles can be omitted and therefore only MERLIN has metal tiles inserted.

For MERLIN we received 25 packaged samples of which 20 do operate correctly for all cipher modes with arbitrary data at moderate clock speed. Two of the failing chips seem completely dead while the others communicate correctly, but deliver partially wrong data. Since MERLIN's main purpose is to serve as a reference for the evaluation of the GALS architecture, we resigned to perform an in-depth failure analysis.
We then determined the maximum clock frequency for which the samples do correctly compute all cipher modes. 9 of 10 tested samples do operate up to 300MHz (3.3ns clock period) and just one chip can also operate with a 3.2ns clock (312MHz). These values are about 10% faster than we could expect according to post-layout timing analysis which predicted a minimum cycle time of 3.65ns.

Measuring the throughput of Merlin is considerably easier than for the GALS version. In the globally-synchronous case one does only need

Figure 6.16: Photograph of the synchronous SAFER chip Merlin (die size 1.8 x 1.8 mm²)
Table 6.4: Performance data for Merlin

<table>
<thead>
<tr>
<th></th>
<th>min. number of cycles</th>
<th>max. throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECB enc.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 rounds</td>
<td>58</td>
<td>334Mbit/s</td>
</tr>
<tr>
<td>1 rounds</td>
<td>31</td>
<td>625</td>
</tr>
<tr>
<td>5 rounds</td>
<td>34</td>
<td>570</td>
</tr>
<tr>
<td>10 rounds</td>
<td>64</td>
<td>303</td>
</tr>
<tr>
<td>12 rounds</td>
<td>76</td>
<td>255</td>
</tr>
<tr>
<td>CBC dec.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 rounds</td>
<td>58</td>
<td>334Mbit/s</td>
</tr>
<tr>
<td>1 rounds</td>
<td>31</td>
<td>625</td>
</tr>
<tr>
<td>5 rounds</td>
<td>34</td>
<td>570</td>
</tr>
<tr>
<td>10 rounds</td>
<td>64</td>
<td>303</td>
</tr>
<tr>
<td>12 rounds</td>
<td>76</td>
<td>255</td>
</tr>
</tbody>
</table>

key preparation time: 1 336.5ns

Table 6.4: Performance data for Merlin

to count the minimum number of clock cycles between the output of two consecutive 64-bit data words, for example on a simulator. Multiplying it with the minimum clock period yields the item cycle time. The performance numbers for MERLIN are given in tab. 6.4 and have already been put into a perspective during the discussion of the MARILYN results.

The power consumption gets measured with the same procedure as described for the GALSIM implementation, but here 8% of the time are spent in loop jumps. The upper half of tab. 6.5 describes MERLIN’s dynamic energy consumption when clockgating is enabled. During the measurements presented in the lower half of the table, the clockgates were switched off and as a result the energy dissipation rose by 32%.
### Table 6.5: Data for MERLIN's dynamic power dissipation (measured @ 2.5V, 20°C)

<table>
<thead>
<tr>
<th></th>
<th>Current [mA]</th>
<th>Data throughput</th>
<th>Energy per Mbit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>w/ clk-gating</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECB enc. 10 rounds</td>
<td>44.5</td>
<td>16.6</td>
<td>151 Mbit/s</td>
</tr>
<tr>
<td>CBC dec. 10 rounds</td>
<td>44.3</td>
<td>16.5</td>
<td>151 Mbit/s</td>
</tr>
<tr>
<td><strong>w/o clk-gating</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECB enc. 10 rounds</td>
<td>58.8</td>
<td>16.6</td>
<td>151 Mbit/s</td>
</tr>
<tr>
<td>CBC dec. 10 rounds</td>
<td>58.3</td>
<td>16.5</td>
<td>151 Mbit/s</td>
</tr>
</tbody>
</table>
Seite Leer / Blank leaf
Chapter 7

Conclusions

7.1 Results

This dissertation describes a design methodology for globally-asynchronous locally-synchronous (GALS) architectures. Each locally-synchronous (LS) module gets equipped with an asynchronous wrapper and all communication between such encapsulated blocks follows an asynchronous handshake protocol.

All asynchronous circuitry necessary for coordinating clock-driven with self-timed operation is confined to the asynchronous wrappers. These are organized in a modular manner and can be assembled from a small library of predesigned elements. This modular approach makes the construction of wrappers easy and safe.

The specification and implementation of the wrapper constituents is described in detail. By using extended-burst-mode specifications and the well established 3D tool for the design of the asynchronous port controllers, the methodology is amenable to parametric timing verification tools as soon as they get available in the near future [Cha98, CDY99]. Currently the internal and external timing correctness of the used asynchronous finite state machines has to be verified by extracting all relevant path information by hand or via scripts.
This thesis makes new contributions to different aspects of multisynchronous system design. The main contributions are summarized below:

It solves the arbitration problem arising when several port controllers do request clock stretching concurrently. By placing the corresponding ME elements in parallel to each other and as bank in parallel to the delayline, the proposed arbitration concept is robust and scalable to virtually any number of ports per wrapper. Making the delayline programmable, adds a new degree of freedom to the design and tuning of GALS systems.

The performance and flexibility of our data transfer channels exceeds all previous concepts. As mentioned, the mutual exclusion elements in the clock generation unit are organized in parallel and thus an arbitrary number of ports may get allowed to transfer data in a particular cycle of the local clock. Additionally each port controller itself can transfer data items in subsequent clock cycles. These two properties distinguish our work from previous concepts, which allowed only one port to be active at a time or required at least two clock cycles per transfer. Based on simulations using a 0.25µm CMOS technology the maximum throughput of the proposed transfer channels exceeds 300Mitems/s with almost arbitrary item width.

An approach to system partitioning and safety analysis of GALS architectures is described. The partitioning process is guided with observations on control organization and overhead estimation, while the deadlock analysis is basically traced back to existing graph theories.

The feasibility and correctness of the developed methodology is proven by the design of a SAFER cryptosystem. Its GALS architecture consists of 9 blocks (5 containing LS modules and 4 memories) which are connected by 15 asynchronous transfer channels. The implementation has a complexity of 58,000 gate equivalents including the area overhead introduced to sustain the full scan-testability of embedded synchronous circuitry. Except the mentioned 3D tool for the synthesis of asynchronous finite state machines, only commercially available tools were used during the design process. As we experienced some particular difficulties with the tools, we also outline possible paths to GALS-compliant tool-flows.

To the best of our knowledge this design is the first implementation of a complex GALS architecture on silicon. Merely one of the existing approaches was verified with a small ring configuration [YD96] but its underlying methodology is not capable of handling more complex configurations. All other approaches were not verified on silicon.
Since the same cryptosystem was also implemented in a conventional synchronous architecture, this thesis also provides a fair evaluation of the effects of GALS architectures on key parameters of the system. Although the synchronous design (MERLIN) already employs clock gating for power saving it still consumes 30\% more energy per Mbit of data than the GALS implementation (MARILYN). We believe that this is mainly due to the adaptive activity patterns observed in a GALS system.

The price we have to pay for this reduced energy dissipation is the area overhead of the asynchronous wrappers. In our case the wrappers enlarge the total cell area by 13\%. Keeping in mind that these wrappers also supersede off-chip clock generation this number seems well acceptable.

In terms of maximum data throughput during cipher operations the synchronous system outperforms the GALS design by 30\%. This is mainly due to an unexpectedly slow operation of MARILYN’s datapath block.

At the bottom line we can state that GALS architectures have the potential to overcome the severe obstacles which conventional design approaches have to face in the near future. This thesis provides the necessary methodology to successfully implement GALS systems and proves the possible energy savings of GALS organization.

7.2 Future Work

Although the described work paves a way to the implementation of GALS architectures, some interesting and important issues in this field remain unanswered so far.

In order to make the organization of global communication more flexible, one might wish to employ a multi-sender multi-receiver transfer channel. A major challenge in the specification of such a bus will be to address all possible arbitration scenarios in a fully self-timed environment. To ensure safety and correctness of global communication one will need to actually perform the deadlock analysis proposed in this thesis. The single standard interface of a GALS compatible bus can also be instrumental for connecting reusable modules in a GALS fashion.

Up to now port controllers are enabled through a transition coded signal. This implies that it is impossible to disable a port unless it gets disabled by transferring data. With an explicit disabling mechanism the designer
would be able to implement more flexible enabling patterns (e.g. one out of two ports shall perform a transfer). Currently such patterns have to be implemented with P-type ports to keep the local clock running. Since a disabling technique would require arbitration between the disabling event and a possibly starting transfer cycle, we refrained from approaching this issue so far.

In order to make the GALS methodology applicable for industrial designs, the asynchronous part of the wrappers will have to be amenable to production test. To achieve this one can make use of existing concepts for testing of asynchronous circuitry and apply them to the port controllers. Far more problematic is how one shall get access to the large number of wrapper-internal nodes and how the clock generation is tested properly.

During discussion with industrial partners we feel that testability is the major obstacle for adopting the GALS methodology to industrial design practice, but lacking tool support already has the second priority. Thus, as soon as the testability issues are solved, the tool-flow for GALS implementations will need to be (re-)evaluated with the criteria given in section 6.5.

That a GALS architecture consumes less power than a conventional synchronous implementation was anticipated, but the amount and exact mechanisms are not fully understood yet. One large contribution can be ascribed to the overall reduction of switched capacitance and activity in clock nets (as calculated by Hemani et al. [HMK+99]). The influence of distributing control to numerous LS modules and the consumption of the asynchronous wrappers have not been analyzed. A power estimation theory describing all these effects would make it possible to quantify the potential power savings and will yield guidelines to system partitioning with respect to power consumption.
# Appendix A

## Acronyms and symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D</td>
<td>tool set for synthesis of extended burst mode machines</td>
</tr>
<tr>
<td>Δ</td>
<td>lower bound for contamination delay of paths, which are external to an asynchronous finite state machine (p.42)</td>
</tr>
<tr>
<td>Ack</td>
<td>acknowledge line of a (general) handshake pair</td>
</tr>
<tr>
<td>AFSM</td>
<td>asynchronous finite state machine</td>
</tr>
<tr>
<td>Ai</td>
<td>wrapper-internal acknowledge of clock stretching</td>
</tr>
<tr>
<td>AO-gate</td>
<td>complex AND-OR gate</td>
</tr>
<tr>
<td>Ap</td>
<td>acknowledge signal between two ports</td>
</tr>
<tr>
<td>ASIC</td>
<td>application specific integrated circuit</td>
</tr>
<tr>
<td>BD</td>
<td>bounded delay (insensitive); a category for the robustness of asynchronous circuits</td>
</tr>
<tr>
<td>CBC</td>
<td>cipher block chaining mode for block-cipher algorithms</td>
</tr>
<tr>
<td>cc</td>
<td>coarse-control signal for delayslice</td>
</tr>
<tr>
<td>CFB</td>
<td>cipher feedback mode for block-cipher algorithms</td>
</tr>
<tr>
<td>clkallowed</td>
<td>output signal of the clockgeneration’s arbitration block</td>
</tr>
<tr>
<td>ClkfreexS</td>
<td>forces the clockgeneration into free-running mode</td>
</tr>
<tr>
<td>ClkinitxRB</td>
<td>clock initialisation/reset signal</td>
</tr>
<tr>
<td>CMOS</td>
<td>complementary MOSFET; a design style for logic</td>
</tr>
<tr>
<td>CSP</td>
<td>communicating sequential processes</td>
</tr>
<tr>
<td>D-type</td>
<td>demand-type (port)</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>delOut</td>
<td>output signal of the delay line</td>
</tr>
<tr>
<td>Den</td>
<td>demand-port enable signal</td>
</tr>
<tr>
<td>DI</td>
<td>delay-insensitive; a category for the robustness of asynchronous circuits</td>
</tr>
<tr>
<td>dslx</td>
<td>delayslice of type x</td>
</tr>
<tr>
<td>ECB</td>
<td>electronic code book mode for block-cipher algorithms</td>
</tr>
<tr>
<td>En</td>
<td>enable signal</td>
</tr>
<tr>
<td>ExtClxC</td>
<td>external clock signal, fed into the LS modules</td>
</tr>
<tr>
<td>ExtClkEnxS</td>
<td>control signal switching the external clock mode on</td>
</tr>
<tr>
<td>fc</td>
<td>fine-control signal for delayslice</td>
</tr>
<tr>
<td>f_clk</td>
<td>clock frequency</td>
</tr>
<tr>
<td>f_d</td>
<td>average toggle frequency of data</td>
</tr>
<tr>
<td>FIFO</td>
<td>first-in first-out memory</td>
</tr>
<tr>
<td>FSM</td>
<td>finite state machine</td>
</tr>
<tr>
<td>GALS</td>
<td>globally-asynchronous locally-synchronous</td>
</tr>
<tr>
<td>GE</td>
<td>gate equivalent; a measure for the complexity of a circuit</td>
</tr>
<tr>
<td>HDL</td>
<td>hardware description language</td>
</tr>
<tr>
<td>InitxB</td>
<td>reset/initialization signal for asynchronous units</td>
</tr>
<tr>
<td>I_clk</td>
<td>local clock signal</td>
</tr>
<tr>
<td>LS</td>
<td>locally-synchronous</td>
</tr>
<tr>
<td>Marilyn</td>
<td>name of the GALS SAFER chip</td>
</tr>
<tr>
<td>ME-element</td>
<td>mutual exclusion element; an asynchronous standard cell</td>
</tr>
<tr>
<td>Merlin</td>
<td>name of the globally-synchronous SAFER chip</td>
</tr>
<tr>
<td>MOSFET</td>
<td>metal-oxide-semiconductor field effect transistor</td>
</tr>
<tr>
<td>Muller-C</td>
<td>2-input majority gate; an asynchronous standard cell</td>
</tr>
<tr>
<td>OFB</td>
<td>output feedback mode for block-cipher algorithms</td>
</tr>
<tr>
<td>P-type</td>
<td>poll-type (port)</td>
</tr>
<tr>
<td>PCC</td>
<td>pausable clocking control [YD96]</td>
</tr>
<tr>
<td>Pen</td>
<td>poll-port enable signal</td>
</tr>
<tr>
<td>PTV</td>
<td>process, temperature, voltage (variations)</td>
</tr>
<tr>
<td>QDI</td>
<td>quasi-delay-insensitive; a category for the robustness of asynchronous circuits</td>
</tr>
<tr>
<td>r</td>
<td>number of encryption rounds</td>
</tr>
<tr>
<td>RAM</td>
<td>random access memory</td>
</tr>
<tr>
<td>r_clk</td>
<td>request clock signal</td>
</tr>
<tr>
<td>Req</td>
<td>request line of a (general) handshake pair</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>$Ri$</td>
<td>wrapper-internal request for clock stretching</td>
</tr>
<tr>
<td>ROM</td>
<td>read only memory</td>
</tr>
<tr>
<td>$Rp$</td>
<td>request signal between two ports</td>
</tr>
<tr>
<td>SAFER</td>
<td>secure and fast encryption routine</td>
</tr>
<tr>
<td>SI</td>
<td>speed-independent; a category for the robustness of asynchronous circuits</td>
</tr>
<tr>
<td>SK-128</td>
<td>strengthened key algorithm with 128 bit keylength</td>
</tr>
<tr>
<td>SOC</td>
<td>system on a chip</td>
</tr>
<tr>
<td>$Ta$</td>
<td>transaction acknowledge</td>
</tr>
<tr>
<td>$t_{cd}$</td>
<td>contamination delay</td>
</tr>
<tr>
<td>$T_{cycle}$</td>
<td>nominal cycle time (in synchronous designs)</td>
</tr>
<tr>
<td>$t_{ho}$</td>
<td>hold time</td>
</tr>
<tr>
<td>$t_{sk}$</td>
<td>clock skew</td>
</tr>
<tr>
<td>$t_{x\to y}$</td>
<td>minimum delay from a transition of type $x$ to a transition of type $y$</td>
</tr>
<tr>
<td>$T_{x\to y}$</td>
<td>maximum delay from a transition of type $x$ to a transition of type $y$</td>
</tr>
<tr>
<td>VHDL</td>
<td>very high speed integrated circuits hardware description language</td>
</tr>
<tr>
<td>VLSI</td>
<td>very large scale integration</td>
</tr>
</tbody>
</table>
Seite Leer / Blank leaf
Bibliography


[BJN99] C. H. (Kees) van Berkel, Mark B. Josephs, and Steven M. Nowick. Scanning the technology: Applications of asyn-


<table>
<thead>
<tr>
<th>Reference</th>
<th>Authors</th>
<th>Title</th>
<th>Journal/Conference</th>
<th>Pages/Publication Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>[CM97]</td>
<td>Richard Carlsson and Håkan Millroth</td>
<td>On cyclic process dependencies and the verification of absence of deadlocks in reactive systems.</td>
<td>Technical report, Computing Science Department, Uppsala University, Sweden</td>
<td>February 1997</td>
</tr>
</tbody>
</table>


Seite Leer /
Blank leaf
Curriculum Vitae

Jens Muttersbach received the Diploma (M.Sc.) degree in physics from the University of Cologne, Germany, in 1996. After graduation he joined the Integrated Systems Laboratory of the Swiss Federal Institute of Technology (ETH) Zürich as a research and teaching assistant in the VLSI design group. During his time at ETH he worked on different aspects of both low-power and synchronization design methodologies and developed the concepts presented in this thesis. In January 2001 he joined Phonak AG in Stäfa, Switzerland.
Why does the sky turn red as the sun sets?

That's all the oxygen in the atmosphere catching fire.

Where does the sun go when it sets?

The sun sets in the west. In Arizona, actually, near Flagstaff.

Oh.

That's why the rocks there are so red.

Don't the people get burned up?

No, the sun goes out as it sets. That's why it's dark at night.

Doesn't the sun crush the whole state when it lands?

Ha ha, of course not. Hold a quarter up. See, the sun's just about the same size.

I thought I read that the sun was really big.

You can't believe everything you read, I'm afraid.

So how does the sun rise in the east if it lands in Arizona each night?

Well, time for bed.

I hope someday I'm as smart as Dad is.

Why, what did he tell you now?