Multi-point Interconnects for Globally-Asynchronous Locally-Synchronous Systems

Author(s): Villiger, Thomas

Publication Date: 2005

Permanent Link: https://doi.org/10.3929/ethz-a-004948731

Rights / License: In Copyright - Non-Commercial Use Permitted
Multi-point Interconnects for Globally-Asynchronous Locally-Synchronous Systems

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZURICH
for the degree of
Doctor of Technical Sciences
presented by
THOMAS VILLIGER
Dipl. El.-Ing. ETH, Federal Institute of Technology Zurich
born July 31th, 1971
citizen of Sins AG
accepted on the recommendation of
Prof. Dr. W. Fichtner, examiner
Dr. Chr. Heer, co-examiner
2005
Acknowledgements

First of all, I would like to thank my supervisor, Prof. Wolfgang Fichtner for his confidence in me and my work and for providing such an excellent working environment at the Integrated Systems Laboratory. I also thank Dr. Christoph Heer from Infineon for reading and co-examining this thesis.

My special thanks go to Norbert Felber and Hubert Kaeslin for their commitment to our GALS project, for many fruitful discussions, and for the constructive proofreading and commenting on my thesis.

This work has been carried out in collaboration with Infineon. In this regard I am thankful to Dr. Christioph Heer and Dr. Vassilios Gerousis for the generous financial support and their willingness to join the two projects we had both with Infineon and Philips Semiconductors Zürich. I am also grateful for the support from Philips and the Swiss Commission for Technology and Innovation (KTI).

I also want to express my gratitude to all colleagues at the IIS who contributed in any way to this work. In particular, I wish to thank Frank Gürkaynak and Stephan Oetiker who carried out great parts of the test chip “Shir Khan”. They were very patient with me adding new features all the time and contributed an unbelievable effort during the tough design phases. Without Franks Perl scripts such a complex integration would not have been possible. I am also thankful to my former project partner Jens Muttersbach who taught me the art of asynchronous VLSI design, to Robert Reutemann and Matthias Brändli for their excellent tool support, and to Thomas Roewer, Markus Thalmann and Manfred Stadler who significantly contributed to a good working environment.

Special thanks go to Christine Haller and the team at the secre-
tariat for their unbureaucratic way of taking care of the administration, to our technicians Hanspeter Mathys and Hansjörg Gisler for their kind help in all practical things, as well as to Christoph Wicki and Anja Böhm for providing an outstanding computer support.

Last but not least, I like to express my heartfelt gratitude to Tina and our two sons Leo and Peter for their appreciation and loving support during the past few years I was working on this project. Special thanks also go to my parents for their support I gratefully got during my life.
Contents

Acknowledgements

Abstract vii

Zusammenfassung ix

1 Introduction 1
  1.1 Motivation ............................... 1
  1.2 Outline ................................. 4

2 Background 5
  2.1 Synchronous Design Paradigm ............... 5
  2.2 Asynchronous Design Methodologies .......... 7
    2.2.1 Self-timed Signalling ................. 10
    2.2.2 Handshaking Alternatives ............. 12
    2.2.3 Classes of Asynchronous Circuits ...... 14
    2.2.4 The Muller C-element .................. 15
    2.2.5 Metastability .......................... 17
    2.2.6 Arbitration ........................... 18
  2.3 Multi-synchronous Systems .................. 19
    2.3.1 Cascaded Synchronisers ............... 20
    2.3.2 Adaptive Data Delay Synchronisation ... 21
    2.3.3 Adaptive Clock Synchronisation - the Road to
          GALS .................................. 22
Contents

3 GALS Design Method

3.1 Self-timed Wrapper

3.1.1 Local Clock Generation

3.1.2 GALS Ports

3.2 Data Transfer Channels

3.3 GALS SAFER-SK128 Implementation

4 System-level Interconnects

4.1 Requirements

4.2 Interconnection Topologies

4.2.1 Shared Bus

4.2.2 Ring

4.2.3 Star, Central Switch

4.2.4 Mesh, Torus

4.3 Arbitration and Access Mechanisms

4.3.1 Arbitration Schemes

4.3.2 Access Mechanism

4.3.3 Bidirectional Data Transfer Channel

4.4 Protocols

4.5 Synchronous On-chip Buses

4.5.1 Advanced Microcontroller Bus Architecture

4.5.2 CoreConnect

4.5.3 Peripheral Interconnect

4.6 Asynchronous Interconnects

4.6.1 MARBLE

4.6.2 CHAIN

4.6.3 Self-timed Ring

4.7 Socket Interface Approaches

4.7.1 Virtual Component Interface

4.7.2 Open Core Protocol

5 Multi-point Interconnects for GALS Systems

5.1 MOGLI

5.1.1 Data Transfer on the Bus

5.1.2 Arbiter

5.1.3 Address Decoder

5.1.4 Demand-type Initiator Port

5.1.5 Poll-type Initiator Port
B.2.1 Instruction Memory
B.2.2 Communication Ports
B.2.3 Instruction & Port Memory Initialisation
B.2.4 Execution Unit
B.3 Instructions

Bibliography

Curriculum Vitae
Abstract

This thesis briefly outlines the development of a novel design methodology for Globally-Asynchronous Locally-Synchronous (GALS) architectures that was done in the first phase of the GALS project and then concentrates on the specification and implementation of appropriate on-chip interconnection structures for complexer GALS systems.

GALS is an approach to VLSI system design that holds the promise of combining the advantages of both synchronous and asynchronous design methodologies. A GALS system employs a self-timed communication scheme between independently clocked circuit blocks, termed locally-synchronous islands. This islands are designed in accordance to proven synchronous clocking disciplines and along a well established design-flow. Any asynchronous circuitry necessary for coordinating the clock-driven with the self-timed operation is confined to self-timed wrappers arranged around each locally-synchronous island. A synchronous island together with the surrounding wrapper then forms a GALS module. To avoid synchronisation failure, all data inputs from other GALS modules have to be synchronised to the local clock. To achieve this, the local clock is paused when data and sampling clock edges occur too close to each other.

The decreasing feature size of modern CMOS technologies allows for integrating more and more functionality on a single chip, referred to as System-on-a-Chip (SoC). This complexity is only manageable if independent subsystems are developed individually before being brought together in the final stages of development. As technology scales down, system-wide communication becomes one of the key compo-
ponents of modern ultra-deep submicron SoC designs, as it entails the main design constraints in terms of system performance, power consumption, robustness, and cost.

So far, all GALS approaches were restricted to point-to-point data transfer channels. Multi-point data exchange has become key necessity of a modern SoC design as point-to-point links alone do not provide the necessary modularity and extendibility. The lack of proven mechanisms for transferring data between multiple synchronous islands has been a major impediment for applying the GALS techniques to SoC design. The GALS point-to-point data exchange channels have therefore been extended towards more versatile multi-point interconnection schemes while preserving the modular approach and the self-timed operation.

Three interconnection structures with distinct topologies were developed, implemented, and compared:

- MOdular GaLs Interconnect (MOGLI), a shared bus solution with both central arbitration and address decoding.
- A ring topology, named Self-Timed Ring for Gals (STRING). With STRING, several GALS modules are connected with a circular path. Dedicated self-timed ring transceivers free the local islands from managing en route traffic.
- SWItching Network for Gals (SWING), a switch matrix built from self-timed crossbar elements.

The three interconnection solutions together with variants thereof have been validated by implementing a complex multiprocessor array on silicon. While their performance measure up to the commercial synchronous counterparts, their main advantage is clearly the modularity they offer and the timing closure that is inherently guaranteed at their interfaces.
Zusammenfassung

Diese Arbeit fasst kurz die Entwurfsmethodik für global-asynchrone lokal-synchrone (GALS) Architekturen zusammen, welche in einer ersten Phase des GALS Projektes entwickelt wurde, und konzentriert sich dann auf die Spezifikation und Implementierung von geeigneten Kommunikationsstrukturen für komplexere GALS Systeme.


Die immer kleiner werdenden Strukturgrössen moderner CMOS Technologien erlauben es, immer mehr Funktionalität auf einen einzelnen Integrierten Schaltkreis zu packen. Diese Systeme werden als System-on-a-Chip (SoC) bezeichnet. Deren Komplexität lässt sich nur noch durch Einsatz von vorgefertigten Schaltungsteilen beherrschen, welche in den letzten Entwicklungs schritten zusammengesetzt werden. Der Einsatz immer fortschrittlicher Technologien lässt die systemweite

Drei Verbindungsstrukturen mit unterschiedlichen Netzwerktopologien wurden entwickelt, auf einem Testchip implementiert und miteinander verglichen:

- **MOdular GaLs Interconnect (MOGLI)**, eine Busstruktur, basierend auf einem gemeinsamen physikalischen Übertragungsmedium. Sowohl Zugriffskontrolle als auch die Dekodierung der Adressen erfolgt zentral.


- **SWItching Network for Gals (SWING)** ist eine Matrix, welche aus kleineren Schaltelementen aufgebaut ist.

Ein komplexes Multiprozessor-System wurde auf Silizium implementiert, um damit die verschiedenen Varianten der drei Netzwerkstrukturen zu verifizieren. Während die GALS Alternativen in Bezug auf Übertragungsrate mit den kommerziellen Lösungen mithalten können, bieten sie den entscheidenden Vorteil, dass sie die modulare Struktur von GALS beibehalten und sich selbst synchronisieren.
Chapter 1

Introduction

1.1 Motivation

Already in 1965 Gordon Moore [Moo65, Moo98] predicted an exponential growth in the number of transistors per integrated circuit made possible by frequent feature size shrinks in CMOS technologies and increasing die areas. The doubling of transistors roughly every 18 months has been maintained, and still holds true down to the present day. Although there is a controversy when physical limits are going to be hit, many experts expect that this trend will continue for at least the next decade.

As a result of these rapid advances in semiconductor technology, the increasingly large number of transistors per chip raises the possibility of integrating more and more functionality on a single chip. System parts that used to be on the PCB board find their way onto one single chip. This so called Systems-on-a-Chip (SoC) greatly reduce the overall costs of a product.

Due to its simple timing model with a global clock that simultaneously triggers all the events in the system, current circuit design is still dominated by synchronous design styles. Nevertheless, technology scaling entails a variety of challenges designers have to meet.

In today’s submicron technologies wire delays are dominating over device delays, making a proper distribution of low-skew clock signals...
reaching the gigahertz range difficult. To minimise skew of a clock distribution network significant costs in terms of die area, power dissipation and design effort. Furthermore, in large chips data can no longer be transferred from one end of the die to the other within one single clock cycle.

In order to minimise costs and meet short times to market, independent pre-designed macro cells, so called intellectual property (IP) modules, are reused and brought together in the final stages of development. Since these IP modules often come from different companies, a multitude of modules adhering to different interface conventions and running at different clock frequencies and clocking styles are to be integrated on a common die, therefore complicating global on-chip communication. In strictly synchronous designs, synchronisers are used between the clock domains to reduce the possibility of metastability or data corruption at the price of higher latencies.

The growing market for battery-operated mobile electronic devices ask for integrated circuits with very low power dissipation. Unfortunately, average power consumption is still rising mainly due to the growing die sizes and number of switching transistors. In combination with the decrease in minimum feature size, this leads to a tremendous increase in power density, making power dissipation a crucial problem not only for portable but also for stationary, high-performance systems.

Once the limitations of globally synchronous designs became highly visible, interest in fundamentally different methods has begun to revive. In asynchronous designs the clock gets replaced by distributed control mechanisms to ensure correct operation. If designed properly, asynchronous systems are robust, have the potential to consume less power, and may even be faster than synchronous counterparts. The price to pay is a widespread control overhead which often increases the die size significantly and eats away the mentioned advantages. Even worse, the well established synchronous design flows can’t be used without large modifications and additional tools that support asynchronous design.

\(^1\)the voltage supply scales slower than the device size scales down
1.1. MOTIVATION

The Globally-Asynchronous Locally-Synchronous (GALS) architecture tries to combine synchronous and asynchronous paradigms to get the benefits from both. The basic idea is to partition a system into several independently clocked modules that are communicating in self-timed fashion as shown in figure 1.1. The functionality of each subsystem is still described and synthesised along well established synchronous design flows, while only the communication between locally-synchronous modules requires specialised asynchronous components.

The intention of the GALS project at our lab was to develop a design methodology based on Cheung and Bormanns work [BC97] and to make it applicable to industrial VLSI design techniques. The results are briefly addressed in chapter 3 and elaborately described in Jens Muttersbach’s PhD thesis [Mut01].

Our first GALS realisations only supported point-to-point connections. Yet, some form of multi-point data exchange is a key necessity of a modern SoC design as it offers the desired modularity and extendibility. This thesis investigates various interconnection schemes and explores their applicability to GALS systems. Three different approaches have been investigated and feasibility has been proven by implementing a GALS multiprocessor array connected by a variety of different on-chip networks.

Figure 1.1: Globally-asynchronous locally-synchronous architecture, showing two synchronous islands connected by a point-to-point link
1.2 Outline

This thesis is structured as follows:

Chapter 2 shall familiarise the inexperienced reader with the fundamentals of asynchronous circuit design. Multi-synchronous systems are described, and related work in the field of GALS is reviewed.

Chapter 3 introduces the GALS approach developed at Integrated Systems Laboratory and describes the main building blocks used for constructing GALS systems.

In chapter 4 the basic principles behind on-chip networks are discussed. Protocol alternatives and the most common academic and commercial on-chip network solutions are addressed.

From a variety of possible topologies, three suitable approaches are chosen for further development. Chapter 5 concentrates on the implementation details of a bus structure, a ring topology, and a switching network and compares their main characteristics.

Our test implementation called "Shir Khan" is addressed in chapter 6. To be able to design a GALS system of a complexity of nearly 3 million transistors, a design flow had to be developed to fill the gap between a standard synchronous flow and the additional needs for handling our asynchronous components. After explaining the design, measurement results are presented.

Finally, chapter 7 concludes this work and gives an outlook on future work.
Chapter 2

Background

The purpose of this chapter is to compare synchronous versus asynchronous circuits, and to give an introduction to asynchronous VLSI circuit design. An overview of previous work done in the field of globally-asynchronous locally-synchronous systems should provide the reader with the necessary background for understanding this thesis. Details on asynchronous design can be found in excellent overviews by Davis [DN97], Hauck [Hau95], Berkel [BJN99] and Josephs [JNvB99].

Circuit design styles are usually classified into two major categories: synchronous and asynchronous. Neither is actually completely independent of the other and many designs have been manufactured using some hybrid form that mixes aspects of both design methods. An example are multi-synchronous systems where Globally-Asynchronous Locally-Synchronous (GALS) designs belong to. Major contributions in the field of GALS are briefly reviewed towards the end of this chapter.

2.1 Synchronous Design Paradigm

Synchronous circuits may be simply defined as circuits which are controlled by one or more globally distributed periodic timing signals called clocks. Transitions of this clock signal indicate points of time
at which data signals are stable and can safely be sampled by the receiving registers. The fixed clock period $T_{clk}$ is usually determined by the application and its worst-case timing analysis.

![Timing path model for synchronous circuits](image)

**Figure 2.1: Timing path model for synchronous circuits**

As shown in fig. 2.1, the minimal clock period is limited by the sum of of the time required for data to leave the originating flip-flop ($t_{pd \, ff}$), the time to propagate through combinational logic and interconnects ($t_{pd \, c}$), the setup time ($t_{su}$) of the receiving register\(^1\), and the margin for clock skew ($t_{sk}$). Clock skew is the maximum difference in the clock arrival time as seen by all clocked elements in the circuit $\max|t_{sk}| = \max(t_{di}) - \min(t_{di})$ with $t_{di}$ denoting the clock distribution delays from clock source to the clocked elements. Only $t_{pd \, c}$ actually contributes to functional computation, the rest is often referred to as timing overhead. The clock is usually not adaptive, such designs do not take advantage of average-case delays. An additional margin to account for manufacturing process, temperature and supply voltage (PTV) variations is required ($t_{PTV}$).

$$T_{clk} > \max(t_{pd \, ff} + t_{pd \, c} + t_{su \, ff}) + \max|t_{sk}| + t_{PTV} \quad (2.1)$$

This simple abstraction of the reality works with 2-valued signals and discrete time steps (the clock). This makes a synchronous system relatively easy to design and verify. Decades of research and experience in synchronous VLSI design has not only created a large number of commercial design tools but has also spawn experienced designers.

Frequent progress in CMOS technologies and the increase in available die size allow for an integration of large systems on a single die.

---

\(^1\)To be safely sampled, all data must be stable within a small timing window around the sampling clock edge, namely a setup time $t_{su}$ before and a hold time $t_{ho}$ after the active clock edge.
Asynchronous Design Methodologies

Asynchronous circuits also work with 2-valued signals but remove the assumption of discrete time steps. Instead they use some form of signalling scheme to indicate data validity. This leads to several potential benefits but also to some drawbacks:

Benefits:

Average instead of worst-case performance. Even if situations dominated by the worst-case delay path rarely occur, synchronous circuits must be clocked in a manner that accommodates the worst-case conditions. Asynchronous circuits in contrast adapt to the situation at hand.

Automatic adaption to physical properties. Circuit delays depend on variations in the fabrication process, temperature, and supply voltage (PTV). Synchronous circuits have to cope with worst-case situations, asynchronous ones can automatically adapt to PTV variations and run as quickly as current physical properties allow.

Low power. Synchronous circuits toggle their clock lines no matter whether a certain part of the system is unused in the current computation. Although clock gating partly eliminates this
deficiency, asynchronous circuits are solely data-driven, so precluding any unnecessary signal transitions when nothing is to execute.

**Simplified global timing.** Asynchronous circuits need no globally distributed clock. So, circuit area, power consumption and the effort for clock tree design can be saved.

**Design reuse.** The inherent ease of asynchronous module composition is due to the fact that properly designed asynchronous circuits export their timing requirements at their interfaces via clean signalling protocols. Modules can be replaced by new components with different delays without altering the rest of the system except for a different overall system speed.

**Reduced electromagnetic interference.** Clocks synchronise activity across the chip. This causes peaks in power-supply noise and concentrates all energy in narrow spectral bands at harmonics of the clock frequency. In asynchronous designs, the activity in different parts of the chip is uncorrelated, so averaging current transients. The spectral energy is spread over a wide band and not concentrated in narrow peaks.

**Drawbacks:**

Why do synchronous designs predominate despite all the potential advantages of asynchronous circuits? The adaptivity of asynchronous circuits not necessarily implies that they achieve higher performance in general. This is due to the fact, that asynchronous circuits have additional constraints compared to their synchronous counterparts:

**Area overhead due to hazard prevention.** When signals change from one voltage level to another, this may happen in a non-monotonic way by first changing from one state to another before settling in a final state. This is termed *hazard* or *glitch*. Since there is no clock to tell when outputs are stable and safe to sample, asynchronous designs must prevent any hazards on their interfaces. In order to make asynchronous circuits hazard-free, these circuits often contain more gates than actually needed for
the pure functionality. Therefore, asynchronous designs are often larger than synchronous circuits. Larger circuits have longer connections with higher capacitances. This slows down the circuit.

**Overhead of the handshaking.** In order to achieve the adaptivity, asynchronous circuits must generate some form of sequence control signals, so called handshake signals. The generation of these explicit control signals further increases the complexity of asynchronous circuits and may lead to performance degradation. The overhead in area can easily reach a factor of two or three [KL02] thus tremendously reducing production yield and increasing costs.

**Lack of commercial design tools.** Asynchronous design has been studied since the early years of computing around the 1950's, but then the synchronous paradigm became popular and there was little research activity on asynchronous designs in the following years, mainly concentrating on theoretical issues. Once the limitations of synchronous design styles became manifest in the late 1980's, research in asynchronous design has been resumed. More structured approaches are now tackled than the asynchronous ad hoc design of the early days. Several methodologies and CAD tools have been developed for asynchronous design, though they usually remain less refined than commercially available tools for synchronous designs and often badly fit into the design flow used by semiconductor companies. A comprehensive list can be found on the asynchronous homepage [Gar].

**Verification and testability.** Important obstacles for a wider distribution of asynchronous methodologies are verification and testability issues. Although addressed by many research groups [MH91, BM91, HBB95, KB95, RB96, Bee03], testing and verification remains a difficult task due to redundancies required for hazard-suppression and problems with including scan paths.

The following subsections briefly explain the relevant issues for asynchronous design, show different classes of asynchronous circuits and important asynchronous elements.
2.2.1 Self-timed Signalling

Asynchronous designs require clean signalling methods to ensure that each data value is correctly passed from one sub-circuit to another. A signalling protocol is used between sender and receiver to control the data transfer. A request signal indicates valid data and the corresponding acknowledge signal informs on acceptance by the receiver. These two control signals called *handshake signals* provide the necessary sequence control for any events in the system. This method is usually referred as self-timed. Throughout this thesis, we stick to the term self-timed wherever data is passed under control of such a handshaking protocol.

Handshake signals may be transmitted on dedicated signalling wires or can be an implicit part of the data-encoding. They act independent of any global time grid. They only depend on a temporal relationship between two subsystems exchanging information with each other. The group of wires including data and handshake information is generally known as *data transfer channel* or *channel* in short. The flow of information can be controlled in two different ways. In a *push* channel the sender initiates the transfer by issuing a request as soon as data is available, the receiver answers with an acknowledge after it was ready to react. The information thus flows in the same direction as the request signal. The opposite is true for the *pull* channel where the receiver requests data and the data flows together with the acknowledge signal from sender to receiver (fig. 2.2).

![Push versus pull channel](image)

Figure 2.2: Push versus pull channel

It seems to be a good idea to distinguish between sender, receiver, initiator, and target as strictly realized by John Bainbridge in his publications on the asynchronous bus called MARBLE [BF98, Bai00]. In
this definition, a sender is the origin of the data and the receiver acts as data sink. The initiator always issues the request and thus initiates any data transfer, the target as the passive part answers with the corresponding acknowledge.

2-Phase Signalling. 2-phase signalling, also called 2-cycle protocol, transition signalling, or non-return to zero (NRZ) signalling is depicted in figure 2.3(a) (the example uses a push channel). Both the rising and falling transitions on the request line indicate a new request. The same holds for the acknowledge signal. The arrows denote the required sequence of events which has strictly to be followed. 2-phase signalling is good from both a power and performance point of view, since every transition represents a meaningful event. No time is needed and no power is consumed for an additional transition back to zero that actually carries no useful information. While in principle this is true, the tide turns as soon as it comes to implementation. Most 2-phase interface circuits require more logic than their 4-phase counterparts. This increased complexity often consumes more power than is saved by the reduced transitions. Most asynchronous designers therefore prefer 4-phase signalling, while 2-phase signalling is widely used in fast micropipelines [Sut89].

![Diagram of 2-phase and 4-phase signalling](image.png)

(a) 2-phase  
(b) 4-phase

Figure 2.3: 2-phase versus 4-phase signalling in the example of a push channel
**4-Phase Signalling.** Figure 2.3(b) shows a 4-phase signalling protocol, which is also known under the following terms: 4-cycle protocol, level-signalling, return to zero (RZ) signalling. It uses the signal level to indicate the validity of data. A complete cycle in this protocol involves four transitions. Interfaces for this protocol are typically smaller than for 2-phase signalling and the falling transitions on the handshake lines often do not degrade performance as they can be performed during time intervals of other actions in the circuit.

### 2.2.2 Handshaking Alternatives

There are different schemes for encoding data validity information. The common ones are depicted in figure 2.4.

![Handshake Encoding Diagram](image)

Figure 2.4: Different alternatives for encoding data validity

**Single-rail Encoding**

In single-rail data encoding schemes, one wire represents a single data bit [Pee96].

**Bundled data.** With this approach, the validity information is passed on two additional wires per bundle of data wires. This wires are called handshake lines. This encoding scheme contains a timing condition in that the propagation delay of the control lines are
2.2. ASYNCHRONOUS DESIGN METHODOLOGIES

at least as fast as the data lines. Bundled-data is widely used in asynchronous design, mainly because its area requirement is similar to those of comparable synchronous circuits.

The handshake protocol defines the period when the data is kept stable and valid. Figure 2.5 shows four possibilities for a 4-phase protocol.

![Figure 2.5: Data validity interpretations [Pee96]](image)

Multi-level logic. This scheme uses a special logic level on the data wires to separate consecutive data items. At least three logic levels are needed for that purpose. Multi-level logic is rarely used, as noise immunity is reduced compared to 2-level logic.

Multi-Rail Encoding

This scheme encodes the validity information into the data value. Multi-rail communication is very robust and offers performance advantages because no delay matching between data and handshake lines is necessary. Unfortunately, the wiring and any logic for encoding and decoding leads to a considerably area overhead.

Dual-rail encoding. Dual-rail or 1-of-2 encoding uses two wires to represent a single bit. Using 4-phase signalling, 01 stands for a logic0, 10 for a logic1, 00 is a spacer to separate consecutive data values, and 11 is never used. Null Convention Logic (NCL) introduced by Theseus Logic Inc. [FB96] works with this scheme.

One-hot encoding. One-hot or 1-of-M codes extend the dual-rail scheme to a 1-of-M encoding, where only one wire of a bundle
of wires carries a one, all others stay at zero. $2^n$ wires are thus needed to transmit $n$ bits of information.

**General N-of-M encodings.** Both dual-rail and one-hot schemes can be generalised to N-of-M codes with arbitrary $N$ and $M$ (as long as $N < M$).

### 2.2.3 Classes of Asynchronous Circuits

Asynchronous circuits can be classified by their robustness against delay variations on their gates and wires.

**Delay Insensitive (DI) circuits.** DI circuits are the most robust asynchronous circuits. They work correctly regardless of the delays on their gates and wires. Unfortunately, this is hard to accomplish and the class of DI circuits built from basic gates is therefore very limited. Only circuits composed of C-elements and inverters can be realized delay-insensitive [Mar90, BE92].

**Quasi Delay Insensitive (QDI) circuits.** This is the minimal deviation from DI circuits to allow any useful circuits. Only the so called isochronic fork constraint is added, that restricts two branches of a single wire to deliver an event at the same time or with negligible difference [Ber92]. It mostly suffices to guarantee that one branch is faster or equally fast to the other under all circumstances. While this seems rather easy to meet, the isochronic fork constraint is often problematic to satisfy because transitions are not instantaneous, but take finite time. Due to variable threshold levels, two different gates on an isochronic fork may see the same transition at different times even though the delays are equal (fig. 2.6).

![Figure 2.6: Signal transition detection by different receivers](image_url)
2.2. ASYNCHRONOUS DESIGN METHODOLOGIES

Speed Independent (SI) circuits. SI circuits work with arbitrary gate delays, but ignore wire delays. For most practical purposes, QDI and SI circuits can be considered identical. While quasi delay-insensitive and speed-independent models allow more implementation alternatives than pure delay insensitive circuits, they require delay assumptions that can be difficult to realise in practice, especially for today's submicron technologies.

Bounded Delay (BD) circuits. This type of circuit has to meet certain timing constraints in order to work properly. Synchronous circuits belong to that class with their flip-flops and latches where setup and hold-times must be met. Careful but simple verification is necessary to ensure correctness after placement and routing.

2.2.4 The Müller C-element

The C-element is commonly used in asynchronous designs and is nearly as fundamental in asynchronous as the NAND gate in synchronous circuits. It raises its output when both inputs become high and lowers the output when both inputs are low, but keeps the old value as long as the inputs have different polarities as illustrated in figure 2.7. The C-element is widely used to synchronise events as it effectively merges two requests to a single request.

Figure 2.7: Behaviour of a Müller C-element

Figure 2.8 depicts the symbol of the symmetric 2-input C-element, together with its logic function and a possible implementation with standard logic gates. Due to the feedback paths from the output back to the AND-gates it acts as a state holding element. To work properly, all internal feedback paths must not be slower than any external
return path back to its inputs. A pure delay-insensitive version can be obtained with the pseudo-static CMOS implementation shown in figure 2.9.

\[ Z = A \cdot B + Z \cdot (A + B) \]

(a) symbol  
(b) logic function  
(c) implementation

Figure 2.8: Muller C-element

Figure 2.9: Pseudo-static implementation of a C-element

Possible variations are asymmetric C-elements with some input signals only affecting either the rising or falling transition\(^2\) of the output signal \(Z\). Figure 2.10 depicts two basic examples.

\(^{2+1}\) inputs only affect the rising transition, \(-\) inputs only the falling one
2.2. ASYNCHRONOUS DESIGN METHODOLOGIES

2.2.5 Metastability

If data at the input of a bistable, such as a flip-flop or a latch, change in a small window\(^3\) around the active clock edge, this causes a setup or hold time violation. Such a violation is also said to produce marginal triggering. The flip-flop may enter a state with unpredictable output that is balancing between its stable logic0 or logic1 condition, as shown in figure 2.11.

This situation, called metastable state, was first observed in the 1960's [Gra63, CM73]. It is a quasi-stable equilibrium (top of the hill) and will eventually settle to a normal steady-state condition, be it logic0 or logic1. How long it takes to settle down largely depends on the technology of the flip-flop, but the delay can be excessive in comparison to the common propagation delay \(t_{pd\,ff}\) [KW76] (fig. 2.12).

\(^3\)called aperture window
2.2.6 Arbitration

If several input signals can arrive in unrestricted sequence and without fixed timing relation, their concurrency must be controlled by a form of arbitration in order to prevent nondeterministic behaviour of the receiving sub-circuit. Consider a circuit to react one way if a transition on signal A occurs and differently if a transition on signal B is detected. The reaction on a concurrent arrival of events on both signal lines can not be defined properly, the circuit would most likely respond in a nondeterministic way and may produce hazards at its outputs.

An arbitration circuitry takes care of the ordering of two asynchronous inputs that occur almost simultaneously. Simple latches and flip-flops are inapplicable for that purpose due to their inherent danger of entering a metastable state as explained in section 2.2.5. All arbiters are therefore realised based on a mutual exclusion element, called MUTEX or ME element, that resolves a possible race at its in-
puts. The MUTEX is basically a bistable element consisting of a pair of inverting gates connected with a positive feedback loop, followed by a filter to defer the response until any metastability has resolved. A CMOS version of Seitz' NMOS implementation [Sei80] is shown in figure 2.13.

![MUTEX Diagram](image)

**Figure 2.13: MUTEX**

If the two input events are sufficiently separated, the faster simply wins the arbitration. However, if both inputs change within a device-specific time slot, the circuit with the feedback loop goes metastable and the filter prevents any change at the outputs as long as the metastability condition is not resolved. The closer the arrival times of the rising transitions at the two inputs are, the longer it probably takes to resolve the metastability. Giving an upper bound for the time needed to make a reliable decision is fundamentally impossible [CM73, KW76]. However, it has been experimentally confirmed that the metastability resolution is extremely fast in general. Arbitration with a MUTEX is “failure-free” if rarely occurring long delays can be accepted [KBY02].

### 2.3 Multi-synchronous Systems

Hybrid designs mix aspects of both synchronous and asynchronous design methods. An example are multi-synchronous systems.

In order to minimise cost and time to market, a SoC device is often composed of heterogeneous parts, so called IP modules. Removing the constraint that all parts of a circuit must run at the same clock rate increases flexibility and modularity. But synchronisation
problems make system integration a major challenge to the emerging SoC industry.

This synchronisation issues are addressed in the next subsections, while the remainder of this chapter reviews major contributions in the field of Globally-Asynchronous Locally-Synchronous (GALS) design methodology, a smart way to synchronise multi-synchronous systems.

2.3.1 Cascaded Synchronisers

The cascaded synchroniser is probably the best known approach for sampling asynchronous input signals or for bridging independent clock domains. Figure 2.14 shows a synchroniser with two flip-flops connected in series. The input signal is sampled by the first register. Unknown timing relations at the interface may lead to metastability, if data and sampling clock edges occur too close to each other.

A statistical model for the probability of system failure [KW76, Vee80, DB99] can be given. A flip-flop with a clock of frequency $f_{clk}$ and an input signal with an average edge rate of $f_d$ has a Mean Time Between Failure (MTBF) of

$$MTBF = \frac{e^{\frac{T}{f_d \cdot f_{clk} \cdot T_{\omega}}}}{f_d \cdot f_{clk} \cdot T_{\omega}}$$

(2.2)

when a time period $T$ is allowed for metastability resolution. For the cascaded flip-flop synchroniser shown in figure 2.14, $T$ is equal to the clock period as the second flip-flop samples the incoming signal one clock cycle later, thereby giving metastability a chance to resolve in the meantime.
Both parameters $T_w$ and $\tau$ are technology dependent and in practice determined by fitting the equation to measurements. $\tau$ is the settling time constant of the flip-flop and $T_w$ is related to the setup/hold window. With cascaded synchronisers using more flip-flops in series, higher MTBF is attained at the cost of additional latency. While the probability of synchronisation failure can be reduced by giving the synchroniser more time to settle, it cannot be anticipated completely.

Equation (2.2) assumes that the input signal changes are uniformly distributed over time. For signals crossing clock domains this is not necessarily the case, though. The clocks of the different domains are often related to each other, resulting in a slowly shifting or even constant phase relation. This can be the case for two communicating blocks that run at different frequencies but derive their clocks from a common source. When data and sampling clock edges occur too close to each other, this will most likely also be the case in the next few clock cycles. This leads to a highly decreased MTBF!

### 2.3.2 Adaptive Data Delay Synchronisation

In systems with derived clocks, adaptive synchronisation can be applied instead. Kol and Ginosar [GK98, GK00] proposed a circuit that measures phase relations between data\(^4\) and clock for adjusting delays in the data path in order to reduce the probability of metastability. This is possible, when phase differences can be considered stationary over large time windows. The delays are readjusted during operation in special training periods.

\[\text{Figure 2.15: Adaptive Synchronisation [GK00]}\]

---

\(^4\)or a corresponding valid or request signal respectively
2.3.3 Adaptive Clock Synchronisation - the Road to GALS

As an alternative to synchronising the incoming data to the local clock, methodologies based on pausable or stretchable clocks are used to synchronise the local clock to the received data.

Related Work on GALS Systems

While stoppable clocks were already mentioned by Pečhouček [Pec76] and Seitz [Sei80], Chapiro proposed what he called unsynchronous systems in 1984 [Cha84]. His approach based on flip-flops with completion detection. When metastability is detected at the outputs of the receiving flip-flops, the local clock of the corresponding synchronous circuit is stopped until the signals have properly settled. Although his ideas are difficult to apply to complex designs, they are the basis of most succeeding work.

Rosenberger et al. [RMCF88] took up the idea of stretching the clock when a metastable state has been detected. Their Q-modules rely exclusively on polling external signals and cannot respond directly to handshake requests. They require up to a full clock cycle to detect a new request and their busy-wait behaviour consumes unnecessary power as the internal clock is running even if no new data is available.

Chapiro's second approach called escapement organisation no longer handled metastable situations but tried to prevent them. Communication was controlled by a handshaking protocol and the local clock was stretched to provide safe sampling when data changes and sampling clock edge occurred too close to each other. For proper operation, the approach uses two non-overlapping two-phase clocks. The concept widely relays on manual design methods and is therefore difficult to apply to larger systems.

Traver [Tra88] further elaborated Chapiro's approach and developed a more methodical design style, thereby primarily concentrating on testability issues.
2.3. MULTI-SYNCHRONOUS SYSTEMS

In 1996 Yun et al. [YD96, YD99b] invented Pausable Clocking Control (PCC) circuits to manage data transfers between independently clocked modules. A PCC circuit generates stretchable clocks and is responsible for processing the external handshake. Yun was the first to address the issue of proper arbitration in clock stretching, but needs to stretch the local clock for each handshake transition. A four-phase handshake protocol therefore requires two clock stretches and a minimum of two full clock cycles. At most one PCC per module can be active at a time, as arbiter trees are used to control access to the local clock. So, the module cannot respond to several requests from different channels concurrently. This arbiter blocks become large and impractical for substantial fan-ins and fan-outs. The approach requires permanently running clocks, thus giving away the chance of reducing the power dissipation.

In 1994, Teich and Thiele [TSTM94, TTSM97] analysed performance and timing behaviour of GALS systems. They presented a graph model called MASS that is basically a timed marked graph extended with additional schedule constraints. It consists of asynchronous as well as synchronous nodes. Whereas the firing rule for asynchronous nodes is similar to nodes in marked graphs, a synchronous node can only react at a tick of its local clock. To keep the model manageable, it is restricted to a common clock period for all nodes, at least with arbitrary but stationary phase relation. Unfortunately, the model is therefore unable to predict the effects of frequency optimisation at the synchronous nodes, as well as handshake protocol variations, and implementation alternatives.

Bormann used a quasi-delay-insensitive dual-rail bus architecture based on the Tangram/Balsa concept of handshake circuits [Ber93, Pee96] to connect both synchronous and asynchronous modules. Previous work from Molina et al. [MCB96] was extended to accommodate stretchable clock modules. Later, Bormann [BC97] introduced asynchronous wrappers to surround locally-synchronous circuits and allow them to communicate asynchronously with the environment (fig. 2.16). Extended burst-mode specification [Yun94] is used to develop all asynchronous controllers that handle the data transfers at the input and output ports. To reduce power dissipation, the local clock is only run-
ning when data is to be processed or transferred. This mature concept covers a variety of possible port configurations (push and pull channels, both for input and output ports) and can easily be scaled for large number of data transfer channels. Unfortunately, it still lacks proper arbitration between concurrent requests to the clock generator, and data exchange is possible every second clock cycle only.

![Bormann's asynchronous wrapper](image)

Figure 2.16: Bormann's asynchronous wrapper [BC97]

Jou and Chuang compared a 8-bit array multiplier in GALS style against a synchronous design and measured the impact on the current and power distribution [JC97]. Not very surprisingly, the GALS version shows a more evenly distributed current pattern and also lower power consumption. While at 100% workload, the power consumption of the GALS multiplier is comparable to the synchronous counterpart, the GALS version consumes only 75% of the power at 50% workload.

Meineke and Hemani [MHK+99, HMK+99] investigated the effects of GALS architectures on clock power consumption. For a circuit of complexity between 1 to 3 million gates, they calculated power savings of up to 70%, only concerning clock distribution. This corresponds to roughly 30% percent overall power dissipation reduction compared to conventional globally synchronous designs. They did not present any design solution and their calculations have so far not been proven by an implementation on silicon, but rely on simulation. All the same, they demonstrated the possible benefits of GALS architectures concerning power reduction.
They also described a strategy for partitioning large synchronous designs into GALS systems. Due to the missing experience with implementations, they make numerous simplifying assumptions and the methodology remains rather vague.

In 2000, Moore et al. [MTC+00, MTMR02] proposed interfaces for connecting synchronous and asynchronous modules that are also based on stretchable clocks. The interfaces are kept small and simple, but communication on their point-to-point channels needs FIFO buffering. As with many other solutions, this concept needs two clock cycles per data transfer.

Quite different from the GALS methods relying on stretchable clocks, Chelcea and Nowick use asynchronous dual-port FIFOs to synchronise modules. No special interface circuitry is needed [CN01]. This is a convenient way to build heterogeneous systems without the need of affecting any local clocks. To reduce the high latencies involved in such a configuration, they did not use pipeline-based “ripple-through” FIFOs but developed an asynchronous FIFO using a token ring structure [CN00] as depicted in fig. 2.17. The main advantage of the FIFO approaches with distinct read and write clocks is, that only the FIFO’s status signals have to be synchronised to the local clocks of either the sender or receiver.

![Mixed-clock FIFO interfaces](image)

Figure 2.17: mixed-clock FIFO interfaces [CN01]

Sjogren et al. [SM00] replace a standard synchronous pipeline within a processor with a mixed pipeline consisting of both synchronous and
asynchronous modules. To prevent potential metastability failures, they incorporate pausable clocks. Their handshake control circuits are designed on transistor level, in order to achieve high throughput. As they have no implementation on silicon, all the given figures are solely based on simulation results. Compared to other GALS approaches, their systems are of considerable finer grain and therefore not directly comparable in performance.

Liljeberg et al. [LPI00, LPI01] employ so called master interface units that connect to a number of server units. They also use a FIFO approach combined with stoppable clocks to synchronise different clock domains. They need a dual-port RAM as FIFO queue plus both an asynchronous and synchronous controller. The former takes care of the external interface towards other units, while the latter controls internal communication with the synchronous circuit. Each controller has a related counter to track the FIFO memory addresses. As this counter must be accessible by both the asynchronous and synchronous controller, an arbiter is required. All this additional circuitry leads to significant area overhead.

Nølstad [NTS+01] and Damhaug [DN02] present a GALS socket interface that is widely based on our solution (explained in chapter 3) with some modifications in the actual implementation of the communication controllers and clock generation unit. While at first sight, the architecture looks rather different, the basic functionality is nearly identical. They evaluate the advantages of voltage reduction and intend to add dynamic voltage scaling but haven’t shown an implementation thereof up to now.

The Electronic Systems Laboratory at University of Linköping recently started activity in multi-synchronous designs. Inspired by our work, Carlsson et al. [CLN+02] and Zhuang et al. [ZLC+02] propose an asynchronous wrapper with low-swing bus drivers for data communication in GALS systems. They only use demand-type ports\(^5\) and a simplified clock stretching mechanism without a complete handshake and no arbitration. While their simple clock generation unit may work

\(^5\)explained in the next chapter
with well behaved demand ports, it would definitely not be sufficient for a second class of ports we developed, the so called poll ports explained in the next chapter. Both works specify the ports with signal transition graphs (STG) [CLW85]. While Carlsson implements the port controllers as hand-designed circuits working with pulsed inputs and week feedback inverters, Zhuang employs ports built upon Muller C-elements.

In Manbo's master thesis [Man02], he employs our port controllers and wrapper design. By using transition signalling on the handshake lines and omitting a full handshake towards the clock generator, he simplifies the controller. Although no figures are given, this most likely leads to improved performance but at the cost of adding many timing constraints to the environment in order to make the system reliable.

In another master thesis, Bart Blaauwendraad [Bla02] focuses on a test strategy for GALS systems to perform manufacturing tests.

The STARI technique uses a self-timed FIFO to compensate for clock-skew between the sender and receiver [Gre95]. Based on that work, Chakraborty [CG03] presents an interface circuit for crossing clock domains that derives a trigger signal from both the sender and receiver clocks. This trigger signal clocks a flip-flop sitting in the data channel. The point of time is chosen in a way, that the vulnerable time window around the active clock edge is hit neither at the flip-flop nor at the receiver side. The circuit primarily only copes with slowly varying phase relations. This is the case, when the sender and receiver clocks are correlated. The approach is therefore suited for streaming data applications with modules clocked by local clocks that are derived from a common clock (mesochronous or rational clock frequency multiples). For plesiochronous designs, where the clocks are independent but closely match in frequency, they cope with the slow drift in the relative timing of clock edges by skipping a clock edge on either the sender or receiver side and inserting "stuff bytes". Synchronisers and additional effort are required for systems where clocks can be arbitrary and uncorrelated. In contrast to GALS systems with stretchable clocks, all clocks are continuously running.
A GALS communication structure using single-track (ST) asynchronous handshaking [BB96] is proposed by De Clercq and Negulescu [CN02]. The ST protocol operates in the following way. The sender pulls down the single-track line to signal a request while the receiver pulls the line high again to signal the acknowledge. So, only one wire for both request and acknowledge is required. A keeper, a positive feedback loop of weak inverters, maintains the voltage level constant when the line is not driven. They achieve high throughput by pipelining the data channel, but their results rely on simulations only.

Iyer and Marculescu use a cycle-accurate simulation environment to study the impact of asynchronous architectures [IM02]. They show for a 5 clock-domain microprocessor that transforming a strictly synchronous system into a GALS structure causes a drop in performance between 5-15%, while power consumption is reduced by only 10% in the average. They claim that the elimination of the global clock does not lead to drastic power reductions, but agree that the flexibility offered by the independently controllable local clocks enables the effective use of other energy-conservation techniques like dynamic voltage scaling. By applying multiple-clock multiple-voltage modules they achieved power savings of up to 21%. As they use Chelcea’s FIFOs to bridge locally-synchronous blocks without clock stretching, it is unclear to what extent their results are applicable to approaches working with clock synchronisation.

Girault and Ménier claim to have a method for automatically deriving a GALS system from an ordinary synchronous circuit on a high level of abstraction. The tool named screp [GM02] is mainly an implementation of an algorithm that takes as input a program whose control structure is a synchronous sequential circuit, and some distribution specifications given by the user. The output is a set of distributed programs that communicate with each other through asynchronous FIFOs. This output is further processed by the Esterel compiler. Esterel is both a programming language, dedicated to programming synchronous reactive systems, and a compiler which translates Esterel programs into finite-state machines [BG92].
Chapter 3

GALS Design Method

This chapter provides an introduction to the GALS design style developed by our group at the Integrated Systems Laboratory as a prerequisite for the understanding of the following chapters. A comprehensive introduction can be found in Muttersbach’s PhD thesis [Mut01].

While the basic concepts of most GALS approaches rely on similar schemes, the various approaches mainly differ in the way they synchronise the different clocks to ensure safe data transfers between synchronous circuit blocks. Our GALS technique [Mut01, MVF00] is based on self-timed wrappers similar to those introduced by Bormann and Cheung [BC97]. As an extension to previous work, where a data transfer can occur every second clock cycle at most, our circuits allow a faster arbitration between concurrent requests to the clock generator, and transfer data in every clock cycle.

![Drawing conventions](image)

Figure 3.1: Drawing conventions
To make things as clear as possible, this thesis adheres to an easy nomenclature. Synchronous subsystems are always drawn with a rectangular box, while asynchronous or self-timed components are depicted with a rounded box (fig. 3.1).

GALS operation employs a self-timed communication scheme on a coarse-grained block level. In GALS, a system is divided into several independently clocked circuit blocks, we name them synchronous islands. They are developed in accordance to proven industry-standard synchronous clocking disciplines and designed along a well established synchronous design-flow. To adapt the synchronous islands to their self-timed environment, they are equipped with so-called self-timed wrappers. A synchronous island together with the surrounding wrapper forms a GALS module. Each data vector entering or leaving the module is accompanied by a request-acknowledge pair of handshake signals and each data exchange strictly follows a 4-phase handshake protocol.

There is no need to time-align the operation of all modules within the framework of a common base clock period. Instead, each synchronous island is driven from a local clock generator located in its self-timed wrapper and is allowed to run at its individual natural clock frequency without regard to other circuit blocks. To avoid synchronisation failure, all data inputs from other GALS modules have to be synchronised to the respective local clock. GALS port controllers can pause the local clock generator to prevent synchronisation failure at the data interfaces. The problem of metastability is thus addressed by the ability to pause the local clock when data and sampling clock edges occur too close to each other. By this method, metastability is prevented and not resolved, as it is done with synchronisers (section 2.3.1). No extra latency for special synchroniser flip-flops or FIFOs is introduced.

GALS holds the promise for solving or avoiding a variety of problems that are bound to become more important with very deep submicron technologies and in view of the virtual component business.

1. Clock domains are confined to manageable sizes thereby alleviating the clock skew problem. For the same reason, the number
of critical paths that must be trimmed at one single moment of time to allow for a specific target clock frequency is greatly reduced. This feature should prove particularly beneficial when timing critically depends on place-and-route because of interconnect delays in submicron designs dominate over gate delays.

2. Exchanging a module against a faster or a more economic alternative becomes possible without having to redesign the rest of the system. This is because of the self-timed interaction and the mutually independent clocks.

3. Assembling systems from pre-designed modules asks for safe and standardized interfaces. GALS addresses the issues of flawless timing and of low-level protocols by imposing a single and well-defined sequence of events at all clock boundaries.

4. If the clock period is locally optimised for every synchronous island, power consumption will be lower, as every subsystem works at the lowest frequency possible to accomplish its tasks. Self-timed operation provides hooks for additional low-power circuit techniques. Due to their inherent adaptability, self-timed interfaces can greatly simplify dynamic voltage scaling techniques.

5. A global clock causes peaks in the spectral band at harmonics of the clock frequency. GALS spreads the spectral energy over a wide band, so reducing electromagnetic emission.

3.1 Self-timed Wrapper

Figure 3.2 depicts a block-level schematic of a GALS module with its self-timed wrapper surrounding the locally-synchronous island. The wrapper contains an arbitrary number of GALS ports, a local pausable clock generator, and test structures. Each GALS port is equipped with a port controller that handles the self-timed communication. To ensure data consistency and prevent metastability at the interfaces, each port controller has access to the local clock generator. Due to the possible disadvantages of asynchronous circuits, the asynchronous parts are kept as small and as simple as possible.

The self-timed wrapper is constructed from a small library of pre-defined elements, thereby making wrapper assembly fast and safe.
This unburdens the system designer from the need of having detailed knowledge of low level asynchronous circuit design. We have developed a sufficient library of partly parameterised wrapper elements described in the technology-independent VHDL hardware description language.

3.1.1 Local Clock Generation

Each wrapper contains a pausable local clock generator based on a ring oscillator structure with a tunable delay line. A coarse block diagram is shown in figure 3.3.

To stretch the local clock $\text{lclk}$, an arbitration block is placed in parallel to the delay line. As requests to pause the clock come from some GALS ports, each incoming request for a clock pause is connected to a mutual exclusion (MUTEX) element that decides whether to grant the request or to permit the next clock pulse. So, the MUTEX is the central element to synchronise data transfers and local clock phases by resolving any metastability within the element itself. As typical standard cell libraries do not include MUTEX elements, it had to be added. Since the MUTEX elements are organised in parallel, the given implementation can easily be scaled to virtually any number of port controllers. This is a considerable advantage compared
to the approaches of Yun[YD96] and others, that use an arbiter tree configuration which becomes large and slow for three or more ports. Additionally, the parallel configuration can grant several stretching requests simultaneously, thereby allowing an arbitrary number of ports to be transmitting within a single clock cycle.

The \textsl{ClkGrant} signal gets active only if all MUTEX elements agree to enable the \textsl{rclk} request. The Muller C-element withholds the rising of \textsl{clk} until both its inputs have become high. Therefore, the active clock edge gets delayed for as long as at least one clock stretching request \textsl{R}_i persist. The reset input \textsl{ClkInitxRB} to the C-element sets the gate to a proper starting state and permits to externally stop the local clock.

\textbf{Adjustable Delay Line}  The delay line in the oscillator determines the clock frequency. To fulfil the synchronous paradigm, its delay has to be consistently larger than half the critical path inside the locally-synchronous island (as it is traversed twice for a full clock cycle, once from rising edge to falling edge, once from falling edge to the rising one). The nominal clock frequency shall be tunable over a wide range to allow adjustments to the actual performance requirements or to
process variations. For our first implementation, we used a simple inverter chain. It provided a high maximum frequency and fine delay steps. However, the simple inverter chain did not closely match the delay drift in the functional circuits over a wide range of PTV variations. This is mainly due to the missing internal nodes compared to slightly complexer elements like AND or OR gates. The capacitances of the internal nodes change with the operating conditions and the gate’s propagation delay drift is different from the one of a simple inverter. For subsequent implementations, the delay line was replaced by a little bit more complex chain built from identical slices of standard gates [TMWR00]. This delay line now closely matches the drift inside the synchronous islands.

![Figure 3.4: Improved delay line](image)

Each transition on \( rclk \) traverses the delay line shown in fig. 3.4(a) both in forward and reverse direction. Each delay slice is built from three NAND gates as depicted in fig. 3.4(b). The control signal \( cc \) steers how many stages of the entire line shall be active. If \( cc \) is low, the signal transitions are travelling through the slice from \( fin \) to \( fout \), through subsequent slices, and on the way back from \( bin \) to \( bout \). In the last active slice \( cc \) is set to one, so the path from \( fin \) towards \( bout \) is enabled and the forward branch gets interrupted. This mechanism ensures that unused slices don’t propagate any signal transitions thereby not consuming switching power. In both cases, the slice’s delay equals the propagation delays of two NANDS.

For a 0.25 \( \mu \) CMOS process technology, we obtain a minimal step size of 320 ps. While this is sufficient for slow circuits, this small delay
step already counts for a considerably large gap in the frequency range for high speed clocks. Let us assume that our synchronous island runs at a maximum clock rate of 600 MHz. If we select 6 delay slices to be active, we get a frequency of $f_{clk} = \frac{1}{6 \times 320 \text{ps}} \approx 513 \text{ MHz}$. If we disable one slice, we get $f_{clk} = \frac{1}{5 \times 320 \text{ps}} = 625 \text{ MHz}$ which is slightly too fast. Thus, we have to stick to 513 MHz facing a drop in performance of 14.5%.

When we want to exploit the synchronous island's maximum possible performance, a high resolution clock generator is required. In our recent implementations we replace the basic clock generator by an enhanced version in terms of maximum achievable clock rates and smaller delay steps for finer tuning. The delay line gets divided into two parts, one for coarse frequency selection, and one for fine tuning. The former is built identical to the architecture described above, the latter is a custom cell with two inverters in series. The first inverter drives a capacitive load, that can be digitally controlled by hooking up one to 24 small loads that provide delay increments of 20 ps each. Two additional diminutive loads even deliver delay increments as small as 12 ps which is sufficient for high demands in clock tuning. We also employ a self-calibrating version based on Taylor et al. [TMWR00] but with enhanced control algorithm. By using a slow external reference clock \(^1\), it automatically finds the appropriate control settings for the desired frequency at power-up and it adjusts the frequency during operation to compensate for changing operating conditions. Details thereof can be found in [OVG+02].

\(^{1}\text{32 kHz in our case}\)
Figure 3.6 shows the whole clock generation unit with all additional signals necessary for configuration, test modes, and enhanced observability. *ClkInitxRB* sets the C-element to proper starting state, *freerun* disables the arbitration block to measure the resulting clock frequency without disruption from the locally-synchronous island, while *ClkSel* together with the *clock divider* allows to use a divided clock. This helps to increase observability, as the fast on-chip clocks can not be observed directly from outside the chip. Slowing down the clock is also useful for low speed operation phases when the circuit is in a nearly sleeping state but still has some watchdog functions to fulfil. *CfgClkSel* selects between local clock and external configuration clock *CfgClk*, which is used to synchronise the functional circuit to the external hardware during configuration or test. While depicted as a simple Mux, the implementation behind is slightly complexer to prevent glitches on the clock lines under any circumstances. Finally, the signal vector *delay control* is responsible for turning on the appropriate number of slices in the delay line.
3.1. SELF-TIMED WRAPPER

3.1.2 GALS Ports

The port controller is responsible for managing all data transfers on a particular port in a GALS system. It consists of an asynchronous finite state machine (AFSM) and a flip-flop for signalling that a transfer has taken place (fig. 3.7). Input ports additionally include a latch register bank, its purpose is explained later in section 3.2.

![Figure 3.7: GALS ports](image)

The enable signal triggered by the locally-synchronous island uses transition signalling while the links between controller and clock generation and between two controllers both employ a complete 4-phase handshaking. The controller therefore basically translates and coordinates the protocols on the three interfaces. Although they get enabled by the synchronous island, the port controllers need to act independently from the local clock signal, in order to transmit data fast and efficiently. This is best achieved by implementing them as asynchronous finite state machines. We describe the behaviour of the controller using extended-burst-mode description [Yun94].

Burst-mode specification was first introduced by Davis et al. [DCS93] and formalised by Nowick, Dill and Yun [ND91, NYD92, Now93]. It is a graph-based finite-state machine specification that consists of a
number of states, a set of arcs, and a unique starting state. Each arc is labelled with a non-empty set of input transitions (an input burst) followed by a set of output transitions (an output burst). Input and output bursts are separated by a slash "/", a rising transition is indicated by "+", a falling transition by "-". Transitions of one set are allowed to occur in arbitrary temporal order. The entire input burst has to occur, before the AFMS fires the specified output burst and enters the next state. The states are held by combinational feedback loops. Burst-mode circuits operate under fundamental mode [Ung69] constraints. That means that new inputs are allowed only after the system has settled into a new stable state in response to the previous input burst. No input burst can be a subset of another burst leaving the same state. This is essential for the finite state machine to determine when the input burst is complete.

As most other asynchronous design methods, burst-mode is totally event-driven, i.e. the order of signal transitions completely determines the current state of the circuit. Level-sensitive inputs as used in synchronous designs, where the clock is the only signal to determine when to trigger a state transition, can not be modelled. The extended-burst-mode (XBM) specifications [Yun94] fills the gap between asynchronous and synchronous styles by adding level-sensitive conditional inputs and directed don’t cares. This allows to model virtual everything from delay-insensitive circuits to synchronous Moore automata. This makes XBM especially useful to model interface circuits between synchronous and asynchronous domains.

Figure 3.8 gives an example of a XBM specification. Signals ending with "+" or "-" are common edge sensitive signals named terminating signals. Conditional or level-sensitive signals are enclosed in angle brackets. Their values are sampled when all of the terminating edges associated with them have occurred. \(<\text{cntgt1}+>\) denotes “if \(\text{cntgt1}\) is high”, \(<\text{cntgt1}->\) stands for “if \(\text{cntgt1}\) is low”. The accordant signal transition only occurs if the conditions are true and all the terminating edges have appeared. Conditional signals must be stable around the sampling point, thus having to fulfil setup and hold time constraints like in a synchronous circuit.

A signal ending with an asterisk is a directed don’t care, which allows a signal to either keep the same value or change exactly once. A directed don’t care (\(\text{fain}^*\)) must be followed by either another directed
3.1. SELF-TIMED WRAPPER

don't care (*fain*) or by an ordinary transition on that particular signal (fain+ or fain-), the so called terminating edge. Directed don't cares are monotonic signals, they are allowed to change at most once during the whole sequence of state transitions they label. If they haven't changed during this sequence, they have to change during the state transition labelled by its terminating edge.

One of the major advantages of the extended-burst-mode description is that it can directly be synthesised into a hazard-free implementation using the 3D synthesis tools for asynchronous control circuitry [YD99a]. Figure 3.9 shows our basic flow to synthesise the asynchronous port controllers. Starting from a XBM specification in graphical form, a 3D description file is written manually and is fed to the 3D tool set. 3D synthesis results in a set of equations, one for each output and one for each additional internal state variable. These equations represent a two-level AND-OR implementation and can therefore easily be transformed into an electrical circuit. For our first implementations, this was a manual task, now this is done by a Perl script developed by Gürkaynak [GOV+03] at our lab. By including technology library information, it translates the equations given by the 3D tool set into a gate netlist written in structural VHDL, and provides basic optimisation for term sharing.
To cover the diverse needs for intermodule communication we have defined two families of port controllers:

**Poll-type port.** A poll port issues requests for clock stretching exclusively to prevent metastability and so ensures data correctness. The clock is influenced as infrequently as possible. A P-type port is appropriate wherever a data transfer is possible, but does not necessarily need to happen immediately. The synchronous island continues to operate normally, while the P-port takes care of the data transfer.

**Demand-type port.** This type of port also ensures data integrity on the transfer channel but adds a feature similar to clock gating. As soon as it is enabled, it stops the local clock and does not release it until the required transfer has safely taken place. A demand-type port is used when the locally-synchronous island can no longer carry out any useful computations without new data. While awaiting the pending exchange, the D-type port
suspends the local clock, thereby effectively preventing any dynamic power dissipation. As soon as a new data item becomes available, the synchronous island resumes operation directly in phase with the incoming data.

The four types of GALS ports are briefly described. Detailed descriptions can be found in Muttersbach’s thesis [Mut01], chapter 3.3.

**Demand-type Input Port.** The asynchronous port controller for the demand-type input port is specified by the extended-burst-mode description given in figure 3.10. The corresponding signal waveforms are given in fig. 3.11.

![Extended-burst-mode specification for a demand-type input port controller AFSM](image)

Figure 3.10: Extended-burst-mode specification for a demand-type input port controller AFSM

After power-up, the asynchronous finite state machine (AFSM) starts in state0. Triggered by a rising edge on the port enable signal Pen, it moves to state1 and concurrently requests for clock stretching (Ri+). The incoming request from the communication partner is defined as directed don’t care Rp*, a transition on that signal may occur by that time but has no influence yet. As soon as the clock is stopped (indicated by Ai+) and a rising edge on the request signal (Rp+) has occurred by now, the machine proceeds to state2 and rises the acknowledge line (Ap+). After having received Rp-, it releases the clock by lowering Ri and state3 is reached. The transfer ends with
receiving \(Ai\)- and releasing the acknowledge signal \((Ap-)\). The AFSM arrives in state 4, where it remains idle waiting for another port enable \((Pen-)\).

To adapt for the transition signalling on \(Pen\), the states 4 through 0 duplicate the first transfer cycle with an inverted direction of the \(Pen\) transition. Although this results in a slightly larger state machine, the area penalty is more than compensated by achieving double the date transfer rate compared to a solution where the synchronous part first has to lower the enable signal before reasserting it in a subsequent clock cycle.

\[
\begin{align*}
Ri &= Rp Ri + Pen Hi + Pen Hi \\
Ap &= Rp Ai + Ai Ap \\
ZO &= Pen ZO + Rp \overline{ZO} + MZO + Pen Rp Ai
\end{align*}
\]

Figure 3.11: Diagrammatic waveforms of a D-type input controller

Synthesis of the extended-burst-mode description yields a hazard-free two-level AND-OR circuit represented by a set of equations, one for each output, and an additional one for the internal state variable \(Z0\) that is assigned by the synthesis algorithm.
As the XBM description of the asynchronous controller does not contain any reset mechanism, the AFSM has to be properly initialised by adding an active low reset signal to all of the gates of the first AND-plane. This calls for an additional input at every gate of this particular plane. Our first implementations described in [Mut01] were fabricated in a technology with equalized gate delays of inverting and noninverting gates. In the technology used for our recent implementations this is no longer the case. Due to the unnecessary inverter stage at the output of the CMOS gates NAND an NOR gates are faster. The AND-OR netlist is therefore transformed to a NAND-NAND structure. The resulting circuit is depicted in fig. 3.12.

![Figure 3.12: Gate-level netlist of the D-type input controller AFSM](image)

**Demand-type Output Port.** Figure 3.13 presents the XBM description of a demand-out port, while fig. 3.14 shows the signal flow. The demand-out port controller behaves similar to the input port, but it plays the active part in the data transfer and rules the sequence of signal transitions. Directed don’t cares ("may come during a particular transition or may not") are therefore unnecessary. On reception of the port enable signal $Pen$, the machine performs a state transition from starting state0 to state1. The clock is stopped, and as soon as
this is confirmed, the request line is raised ($Rp+$). After a full 4-phase handshake on the handshake lines ($Rp+ \rightarrow Ap+ \rightarrow Rp- \rightarrow Ap-$) the clock is released and the AFSM waits in state 4 for a new data transfer request. The remaining states do exactly the same, just with the negative edge on $Pen$.

Figure 3.14: Diagrammatic waveforms of a D-type output controller
3.1. SELF-TIMED WRAPPER

3D synthesis generates the following equations:

\[
R_p = Pen \overline{A_i} \overline{A_p} R_i Z_1 + \overline{Pen} \overline{A_i} \overline{A_p} R_i \overline{Z_1}
\]
\[
R_i = A_p + \overline{Pen} \overline{A_i} Z_0 + \overline{Pen} \overline{A_i} \overline{Z_0} + Pen R_i Z_1 + \overline{Pen} R_i \overline{Z_1}
\]
\[
Z_0 = Pen A_p + Pen Z_0 + \overline{A_p} Z_0
\]
\[
Z_1 = \overline{Pen} A_p + \overline{Pen} Z_1 + \overline{A_p} Z_1 + Pen \overline{A_p} \overline{Z_0}
\]

Poll-type Input Port. The poll-type ports differ from their demand-type counterparts in the way they stop the local clock. While the latter stops its clock after a port enable as soon as possible to perform a "sleep while waiting" scheme, poll ports stretch the clock only to synchronise the transfer and influence internal computation as scarce as possible.

![Poll-type Input Port AFSM](image)

Figure 3.15: Specification of a poll-type input port controller AFSM

The XBM specification is shown if fig. 3.15, the resulting waveforms in fig. 3.16. Different from a demand-in port, the state transition from state 0 to state 1 and clock stretching \( R_i + \) does not occur before both the port enable \( (Pen+) \) and the request from the sending port \( (R_p+) \) have arrived. Then, a normal handshake cycle is performed. Note that \( R_p^* \) is defined as directed don’t care in transition from state 3 to 4 and from state 7 to 0. This is needed since a new request may come in reaction to \( A_p^- \) while the local clock is still about to be restarted. Implementation details can be found in [Mut01].
Poll-type Output Port. The poll-type output port works similar to the other ports mentioned above, its specification (fig. 3.17) and waveforms (fig. 3.18) are simply shown for completeness.

As it is an output port, it determines the handshake sequence, so it does not have to cope with concurrency. Directed don’t cares are therefore unnecessary.
Area and Timing. Table 3.1 gives an overview of the main port controllers. For each type, the area in gate equivalents (GE) is given as well as timing information. \( t_{in\rightarrow out} \) specifies the propagation delay from input to output. The cycle time \( T_{cycle} \) is the time needed by the asynchronous FSM to settle. In order to meet the fundamental mode constraints, no further input events are allowed to occur during this settling time.

<table>
<thead>
<tr>
<th></th>
<th>area [GE]</th>
<th>( t_{in\rightarrow out} )[ps]</th>
<th>( T_{cycle} )[ps]</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-input</td>
<td>20.5</td>
<td>202</td>
<td>227</td>
</tr>
<tr>
<td>D-output</td>
<td>44.5</td>
<td>342</td>
<td>363</td>
</tr>
<tr>
<td>P-input</td>
<td>17.0</td>
<td>187</td>
<td>232</td>
</tr>
<tr>
<td>P-output</td>
<td>21.5</td>
<td>181</td>
<td>194</td>
</tr>
<tr>
<td>average</td>
<td>26</td>
<td>228</td>
<td>254</td>
</tr>
</tbody>
</table>

Table 3.1: Area and timing figures for the port controllers (for a typical 0.25\( \mu \)m CMOS technology)
The asynchronous port controllers are apparently very small and fast due to their two-level logic structure. They achieve cycle times of less than 350ps for a 0.25μm technology. Obtaining the same performance with a synchronous implementation would require clock frequencies well above 2.8 GHz.

The numbers differ from the ones given in [Mut01] because he relied entirely on synthesis with the 3D tool set that does not find the optimal solution. In the meantime, we use our "eqn2gate" script to fill the gap between equations and gate-level netlist. This script starts with the equations provided by the 3D tool set, but additionally tries to share common terms in the AND-plain whenever possible. This, and the transformation into the NAND-NAND structure reduce the area requirement and slightly speed up the circuit.

3.2 Data Transfer Channels.

![Figure 3.19: Unidirectional GALS data transfer channel](image)

Data is passed between GALS modules using an output port, an input port, and a group of wires. This is called data transfer channel, or channel in short. These channels are usually unidirectional point-to-point interconnections and consist of an arbitrary number of
3.2. DATA TRANSFER CHANNELS.

data wires and two additional handshake lines for signalling request and acknowledge. The GALS data transfer channels all employ a 4-phase handshake protocol and broad data validity scheme (figure 2.5 describes possible data validity options). A channel works with a rendezvous scheme, that means, a transfer can only take place at a point in time when both sender and receiver are ready and have enabled their port controllers.

Figure 3.19 shows a typical GALS data transfer channel, in this particular example from a demand-out to a poll-in port, to gain a brief inside into both types of operation. The corresponding signal flow is depicted in fig. 3.20.

Figure 3.20: GALS data transfer mechanism
The demand-out port immediately rises $R_i (B)$ on reception of a port enable (A). As soon this is confirmed by the local clock generator (C), a request is issued on $R_p (D)$. By this time, the receiving port has not been enabled yet. Only when the poll-in port got both an enable (E) and a rising edge on $R_p$, it tries to stop its local clock (F). When this was successful (G), it replays with a logic1 on the acknowledge line $A_p (I)$. $A_{i2+}$ also sets the transfer acknowledge $T_{a2} (H)$. In reaction to the acknowledge, the sender signals $T_{a+}$ to the synchronous island (J) and withdraws the request (K), then the receiver port lowers the acknowledge (N) and clock stopping request (L) concurrently. Upon receiving $A_p-$, also the sender releases its clock (O) and both GALS modules restart their local clocks ((M) $\rightarrow$ (Q) and (P) $\rightarrow$ (Q)) and resume normal operation. Both clocks reset the transfer acknowledge signals of the corresponding GALS module (R).

The latch bank of the input port is controlled by the acknowledge signal $A_p$. The grey area in fig. 3.20 indicates, when the latches are transparent. The latches are needed only to decouple the timing of the sender from the receiver side. Let us assume, that after sucessfull completion of the transfer, the receiving module would still be blocked by a pending transfer on another port. As it could not release its local clock and would be unable to store the incomming data into its input registers. When working without latches, it couldn’t signal completion of the handshake cycle ($A_p-$) and this would block the sender as long as the receiver module is waiting for any other transfers to complete.

### 3.3 GALS SAFER-SK128 Implementation

To verify the correctness and feasibility of the GALS methodology, we have implemented a small system in a 0.25$\mu$m process and compared it to a synchronous version fabricated on the same lot [VMK+01]. A secret-key iterated block-cipher algorithm called SAFER SK-128 was chosen as application. SAFER stands for Secure And Fast Encryption Routine, an algorithm invented by James Massey [Mas94]. This algorithm is attractive for system studies and hardware implementations in general, because the datapath contains a number of well defined sub-functions. Only the data flow among them has to be adapted to the required operating conditions. Thus, this design poses interest-
ing and challenging constraints to the structure of the communication network and is a good example for vivid interaction between modules performing different tasks. The GALS system consisted of 9 modules (5 clock domains, 3 memories, and an asynchronous FIFO) containing a total of 25 port controllers. Table 3.2 gives a short summary of the results obtained. All communication within this system is based on unidirectional point-to-point links.

Throughput is slightly slower in the GALS design compared to the synchronous version. This is only due to a minor design flaw, which prohibited to tune the clock frequency of the datapath module to its optimal operation range. Simulations showed that the GALS approach could actually run slightly faster than the synchronous design.

<table>
<thead>
<tr>
<th>Figure of merit</th>
<th>Clocking</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GALS</td>
</tr>
<tr>
<td>Fabrication process</td>
<td>0.25µm 5LM</td>
</tr>
<tr>
<td>Supply voltage [V]</td>
<td>2.5</td>
</tr>
<tr>
<td>Clock domains</td>
<td>5</td>
</tr>
<tr>
<td>Total cell area [mm²]</td>
<td>1.56</td>
</tr>
<tr>
<td>GALS area overhead without test structures [%]</td>
<td>9</td>
</tr>
<tr>
<td>Max. throughput [Mbit/s]</td>
<td>227</td>
</tr>
<tr>
<td>@ 10 rounds CBC</td>
<td>577</td>
</tr>
</tbody>
</table>

Table 3.2: GALS versus synchronous implementation
Seite Leer / Blank leaf
Chapter 4

System-level Interconnects

Advances in silicon technologies allow for the integration of huge functionality on one single silicon chip. Peripherals formerly found at the board level are now integrated onto the same die together with the main circuit, e.g. a microprocessor core. Such systems are commonly called Systems-on-Chip (SoC). SoCs typically contain numerous functional blocks and consist of a few million gates. To meet requirements in terms of cost and short time-to-market, a macro-based approach is necessary. It provides numerous benefits during development and verification, but the ability to reuse pre-developed circuit blocks or third party modules is often considered the most significant. Such Intellectual Property (IP) modules must be interconnected in a convenient way to ensure effective communication among them. Interconnectivity of modules is usually provided by buses or more general on-chip interconnect structures.

This chapter introduces the basic concepts of multi-point interconnects for on-chip communication. To compete with today’s synchronous systems, our point-to-point GALS data transfer channels have to be extended towards more versatile multi-point interconnects. Key requirements for these new interconnect structures are found by reinves-
tigating existing on-chip solutions in the context of a self-timed implementation. When evaluating or developing on-chip interconnects, the designer has a broad range of alternatives: He can chose from a variety of topologies, arbitration schemes, protocols, and – for more complex structures – routing algorithms. While a self-timed environment calls for special precautions in terms of timing issues, most decision criterions remain similar to the synchronous case.

4.1 Requirements

In order to evaluate possible solutions, the designer must have appropriate performance metrics. The key performance requirements for interconnects are:

**Throughput.** Throughput often also called bandwidth is an important parameter to describe the performance of an interconnect solution. It measures the capacity of the network and is defined as the total amount of data transferred per second. It is often expressed in units of bits per second (bps), calculated as the product of the number of bits that can be transmitted in parallel in any transaction by the number of transactions that occur per second.

**Latency.** Communication latency is the amount of time it takes from issuing a command till the associated response is completely received by the recipient. This determines for example how long a processor will have to wait when it fetches an instruction from memory.

**Efficiency.** The routing of data traffic should utilise the network resources as efficient as possible to avoid congestion, which would dramatically reduce throughput.

Throughput and latency compete with each other. Higher throughput can usually be gained by accepting higher latency, e.g. by inserting pipeline registers.
4.2 Interconnection Topologies

The topology of a network is the way the devices are physically or logically connected together. Depending on the specific communication requirements and the available resources, a suitable topology is chosen from a variety of different structures. Figures 4.1 and 4.2 show possible network topologies. The selection is based on the US Federal Standard 1037C [NCSTSD96].

Figure 4.1: Network topologies (I)
The point-to-point link and the linear structure (fig. 4.1(a) and 4.1(b)) are not addressed within this chapter since they can be built with our standard GALS point-to-point transfer channels. The tree (fig. 4.1(f)) is uncommon for on-chip interconnects and is also not discussed. Hybrid structures are formed by combinations of different topologies, their characteristics are mainly determined by the employed topologies.

4.2.1 Shared Bus

A bus connects all nodes by one single data transfer channel through which all data transfers take place (fig. 4.1(c)). Each node intending to transmit data, has to arbitrate for the shared resources before it is allowed to access the bus.
4.2. INTERCONNECTION TOPOLOGIES

Advantages

Cost. As the interconnecting wires are shared among all the nodes, the area requirement is low. This leads to reduced die sizes and lower costs.

Simplicity. A bus network is simple.

Modularity. It is easy to attach new nodes to the bus. For on-chip buses this has to be done at design time.

Disadvantages

Performance. As the resources are shared, the available bandwidth is divided among all active nodes.

Scalability. Due to a limited overall bandwidth, additional devices have an influence on the available bandwidth for the other modules.

Reliability. The system relies on a single backbone. A failure there would crash the whole system.

4.2.2 Ring

A ring consists of a number of nodes that are linked in a circular fashion to form a closed loop (fig. 4.1(d)). Adjacent pairs of nodes are directly connected by unidirectional point-to-point links.

Advantages

Performance. The number of ring segments, data links from one node to its neighbour in direction of the data flow, scale linearly with the number of attached nodes. So, the throughput is less limited by bottleneck problems compared to the shared bus. However, the communication latencies get worse with the number of network nodes that have to be traversed on the way from sender to receiver.
Fairness. A token ring passes a token around. Only the node holding the token is allowed to access the network. If it has nothing to send it simply passes the token on to its neighbour. Another form of ring communication is called slotted ring. Here, a fixed number of data slots are travelling round the ring. When a node wants to transmit data, it waits for the next empty slot and inserts its data packet. These systems treat every node equally fair.

Simplicity. Construction of a ring is rather simple. All nodes can be equal, no difference is made between nodes that can actively access the ring and others that can just respond to requests.

Disadvantages

Reliability. Unfortunately, the ring is error-prone. A failure of any node or data link would block the whole ring. This can be avoided by providing switches at every node, that bypass this particular node or link, if it is broken. Hardware failures are not an issue for on-chip rings but system misbehaviour may still lead to communication deadlocks. When system reliability is of critical concern, an alternative network like a mesh proves superior to a ring.

4.2.3 Star, Central Switch

In a star topology as shown in fig. 4.1(e), all nodes are connected to a single central node or switch that performs all routing.

Advantages

Performance. The number of data links scales with the number of attached nodes. The bandwidth of the links therefore scales well. This is a bit different for the central node. Only if the switch fabric provides sufficient bandwidth to cope with the data streams coming from the nodes, performance degradation can be avoided. This usually requires a high number of concurrent data paths within the switch.

1Defective chips have been sorted out during production test.
Modularity. New devices can easily be attached to the star, at least as long, as the maximum number of provided interfaces at the switch is not exceeded.

Reliability. A star is rather safe, as far as only the data links are considered. If a node or a data link fails, just this particular node is unreachable, none of the other links or nodes will be affected. However, the central switch itself is a vulnerable element. If it fails, the whole network breaks down.

Disadvantages

Cost. All the dedicated lines from central switch to all the nodes account for an increased area requirement compared to a bus solution.

Scalability. Although the scalability of the link’s bandwidth is excellent, a central switch can become the bottleneck.

4.2.4 Mesh, Torus

In the partial mesh topology, some but not all of the nodes are directly connected. The nodes are often arranged on a regular two-dimensional (or even n-dimensional) grid as depicted in fig. 4.2(a). A variation of a mesh with the links wrapped around at the edges of the grid is called torus. It can be seen as a combination of ring and a mesh. A special case is a fully connected mesh that provides dedicated links between every pair of nodes (fig. 4.2(b)).

Advantages

Performance. Due to the high number of links between nodes, the network provides an extremely high bandwidth. Communication latencies are higher than with a shared bus, because data has to pass several nodes on the way from sender to receiver. At least, the latency is usually lower compared to a ring topology of equal number of nodes since the higher number of connections allows shorter end-to-end paths.
Reliability. Thanks to a high redundancy, the reliability of this topology is very high. Even if some links fail, there are many other possible routes between any two nodes.

Scalability. A mesh scales very well in terms of performance, but the area requirement grows rapidly, especially for highly connected meshes.

Disadvantages

Cost. The main drawback of the mesh is its expense, because of the large number of connections.

Latency. The end-to-end latency from sender to receiver is usually higher than for a bus structure.

Complexity. Intermediate nodes traversed on the way to the destination, actively route any data packets on a path towards the receiver. If required, this can be done in a highly sophisticated way. The router could try to find an optimal path to avoid congestion, reduce communication latency, or circumvent spots of failure. So, the routing circuitry can get pretty complex.

4.3 Arbitration and Access Mechanisms

Access to the interconnection fabric must be properly handled. In this context, one has to distinguish between the right to take over control and initiate a data transfer, and the right to drive a logic value on shared data wires:

Arbitration. A device attached to a network that can initiate a data transfer is called an initiator or master. A device that is just responding to requests is called a target or slave. If more than one initiator is present, some sort of arbitration is necessary to grant exclusive access to one single initiator.

Bus access. Different from the control point of view, a sender is the origin of data and the receiver acts as data sink. If many devices are connected to the same group of wires, only one device is
allowed to drive the line at a time to prevent shorts. This is handled by proper access schemes described below.

### 4.3.1 Arbitration Schemes

This subsection provides a list of common arbitration schemes. All known on-chip bus connections either rely on central arbitration or on a token ring approach. Distributed arbitration schemes based on collision detection are nearly out of use today for computer networks, and have never been used for on-chip systems due to the high current flow during data collision when multiple driver access the shared resources.

**Central arbitration:** One central arbiter decides which initiator is allowed to take over control.

**Handshake.** All initiators are connected to the central arbiter and ask for permission to control the bus. Mostly, a request-grant handshake is implemented for that purpose. A variety of priority schemes are applicable (fixed priority, first-come first-serve, round robin, adaptable priority). Drawbacks are the number of necessary wires.

**Polling.** It works similar to the handshaking approach, but with one single request signal that combines all requests. Upon receiving a request, the arbiter has to interrogate every initiator to find out which one has sent the request.

**Daisy chain.** One request signal goes to the arbiter. The grant signal is passed from initiator to initiator. Only if a initiator asked for the bus, it does not pass the grant signal to its neighbour. Therefore the priority is fixed by the physical location on the daisy chain. The device closest to the arbiter wins. This scheme could also be classified as distributed arbitration.

**Distributed arbitration:** The initiators decide themselves which one is allowed to start a bus transaction. Advantages compared to the centralised approach are the enhanced flexibility and fault tolerance. On the other hand, more control logic generally slows down the system.
CSMA/CD. Using Carrier Sense Multiple Access / Collision Detection, each initiator accesses the bus without preceding arbitration. Before sending, it listens to the bus to check if it is empty. If so, it starts transmitting. Then it checks the data on the bus lines. If there is a difference between the data packet sent to the bus and the one read back, there was a collision with data from another initiator. In this case, all involved initiators withdraw from the bus and wait for a random time interval before retrying to access the shared resources.

**Token ring.** A token is passed around. Only the interface holding the token is allowed to send on the bus.

### 4.3.2 Access Mechanism

Figure 4.3 depicts the three main approaches to prevent multiple drivers from shorting shared wires.

![Access mechanisms for shared wires](image)

Figure 4.3: Access mechanisms for shared wires (for simplicity reasons, drawn for unidirectional data transfer only)

**Tristate driver.** Access to the shared bus lines is provided by tristate buffers. Only one source at a time may actively drive the bus, while all others must be high impedance. The bus protocol is responsible to ensure this under any circumstances.
4.3. ARBITRATION AND ACCESS MECHANISMS

**Multiplexed.** A multiplexer is used to switch between the different sources of the signal. Only the output from one source module is routed to the destinations.

**OR connection.** The tristate buffers are replaced by AND gates. Each of the source modules must drive the output low, unless it is the active source of the signal, in which case it drives the desired value on the output. The output of all source modules is then ORed together before being routed to all the destinations. As this method basically implements a multiplexer it is often named “distributed multiplexing”.

In order to avoid driver conflict, floating bus signals, and difficulties with testing, industry tends to avoid the use of tristate on-chip buses in favour of an OR connection or a multiplexed approach.

### 4.3.3 Bidirectional Data Transfer Channel

**True bidirectional.** Data is usually transferred between initiator and target in both directions. In the early days, this was often done on a bidirectional transfer channel, using a bundle of physical wires, that were accessed by both the initiator and target (fig. 4.4(a)). Tristate drivers must be used to access the shared wires. Because of the problems associated with the tristate approach, it is hardly used for on-chip buses.

**Pseudo-bidirectional.** The bidirectional data transfer channels commonly used in on-chip buses are composed of two unidirectional bundles of wires running in different directions as depicted in figure 4.4(b). Only one arbiter and address decoder is necessary, since there is only one single transfer channel. Access control to the wires however must still be handled at both the initiator and target side.

**Dual-channel.** As an alternative, two distinct transfer channels can be used to handle data flow in both directions. Separate arbitration, address decoding, and access control is needed for the back channel, called response channel (fig. 4.4(c)).
(a) true bidirectional transfer channel with bidirectional wires

(b) bidirectional transfer channel built from two unidirectional bundles of physical wires

(c) separate transfer channel for each direction

Figure 4.4: Different ways to transfer data bidirectionally
4.4 Protocols

A protocol is a set of rules determining the format and transmission of data on the interconnection structure. While this section concentrates on bus protocols, they also apply to other network structures.

**Bus primitives.** For a synchronous bus, one bus clock period is called *bus cycle*. A *bus transfer* is a read or write operation of a data packet, which may take one or more bus cycles. Communication on a bus is usually divided into discrete *transactions*, each of which consist of several bus transfers. An example are burst transfers explained below.

In order to initiate a transaction, an initiator has to gain control of the bus. If this was successful, it uses a communication protocol to control the transfer. In an asynchronous protocol the transfer can begin at any time, whereas in a synchronous protocol, transfers are controlled by a global clock and only start at discrete points in time. A bus transaction usually contains a *command/address phase* and one or multiple *data phases*. In the command/address phase, the initiator accesses the appropriate target and requests to perform a certain task. The target address, possible target internal addresses, and a command word (including all necessary control bits) are transferred in this phase. Finally, one or several data values are transmitted.

Address/command information and data values can either be transmitted in time multiplexed fashion on the same or on separate physical wires.

**Multiplexed address/data lines.** The transaction phases share the bus wires they need. First the command/address phase is transmitted, then the data phases on the same lines. This scheme is common on board level or for backplane buses with their limited number of available connectors but not for on-chip use.

**Separate address and data lines.** Pipelining of address and data

\[\text{\footnotesize Self-timed solutions do not rely on a clock. The most basic primitive there is a handshake cycle that normally controls a bus transfer.}\]
phases as shown in fig. 4.5 allow for higher throughput and lower latencies. This is only possible with separate lines for the command/address and the data part. On-chip bus solutions prefer separate address and data lines, as they are less limited in wiring resources.

Figure 4.5: Pipelining of command/address and data phases on a shared bus

An interconnect solution generally supports at least part of the following features:

**Hidden arbitration.** Hiding the latency of the arbiter by doing the arbitration for the next bus cycle concurrently with the ongoing one is called hidden arbitration. The answer from the arbiter is then an "early-grant", indicating which device is allowed to take bus ownership when the shared resources next become idle.

**Burst transfers.** A single arbitration is followed by several successive and related data transfers. Burst transfers usually allow a higher throughput than multiple distinct transfers because of the reduced number of arbitrations that have to be performed.

**Atomic sequences.** An initiator is allowed to make a number of consecutive bus transfers without other devices accessing that resource in the meantime (e.g. read-modify-write).

**Interlocked or decoupled transfers.** In interlocked buses, the address/command phase and the data phase of a transfer are tightly coupled. An alternative approach is to dissociate the two phases, allowing the address/command and response/data phases to be separated by other bus activities. A separate arbitration for
each part of the transfer is required in this case. The advantage of using a decoupled protocol is that it gives a greater bus availability, allowing transactions to be interleaved on a phase-by-phase instead of a transfer-by-transfer basis.

**Split transactions.** Slow devices may accept the address/command part and then disconnect from the bus in order to reconnect later to perform the data action of the bus transfer. Command and response phases are treated as separate packets to be transferred. This improves bus availability since the bus is free for other devices in the meantime. Split transactions can be implemented on top of either an interlocked or a decoupled protocol and may require two transfers per transaction. One to pass the command and address and one to return status and data.

**Bus deferral.** Deadlock situations can occur when bridges exist between two buses: one initiator wants to talk to a target on the other bus, but here is also a initiator addressing a target on the first bus. When the bridge can not handle both transfers concurrently, the situation ends up in a deadlock. Thus, one device has to cancel its transfer. The bus deferral feature is therefore a prerequisite for bus bridging.

### 4.5 Synchronous On-chip Buses

As design reuse is one of the most important advantages in SoC designs, a standard interconnect fabric is highly desirable. There was some effort to standardise on-chip interconnects. However, despite the rapid growth of providers of IP modules, industry did not manage to agree on one single solution so far. A few solutions coexist, each of which originates from a single or a group of companies.

Three widely accepted synchronous solutions and three asynchronous ones are described below. Except for CHAIN and a self-timed ring, they are all shared bus approaches with central arbitration. Table 4.1 provides links to available on-chip interconnects. The large number of approaches points out how difficult it is to find a common standard. Three widely used solutions are briefly reviewed below, table 4.2 abstracts their basic properties.
<table>
<thead>
<tr>
<th>Bus Name</th>
<th>Originator</th>
<th>Hot link</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMBA</td>
<td>ARM Limited</td>
<td><a href="http://www.arm.com">www.arm.com</a></td>
</tr>
<tr>
<td>Avalon</td>
<td>Altera Corporation</td>
<td><a href="http://www.altera.com">www.altera.com</a></td>
</tr>
<tr>
<td>CHAIN</td>
<td>University of Manchester</td>
<td><a href="http://www.cs.man.ac.uk/apt">www.cs.man.ac.uk/apt</a></td>
</tr>
<tr>
<td>CoreConnect</td>
<td>IBM</td>
<td><a href="http://www.ibm.com">www.ibm.com</a></td>
</tr>
<tr>
<td>CoreFrame</td>
<td>Palmchip Corporation</td>
<td><a href="http://www.palmchip.com">www.palmchip.com</a></td>
</tr>
<tr>
<td>IPBus</td>
<td>Integrated Device Technology (IDT)</td>
<td><a href="http://www.idt.com">www.idt.com</a></td>
</tr>
<tr>
<td>IP Interface</td>
<td>Motorola Inc.</td>
<td><a href="http://www.motorola.com">www.motorola.com</a></td>
</tr>
<tr>
<td>MARBLE</td>
<td>University of Manchester</td>
<td><a href="http://www.cs.man.ac.uk/apt">www.cs.man.ac.uk/apt</a></td>
</tr>
<tr>
<td>OCP</td>
<td>OCP-IP</td>
<td><a href="http://www.ocpip.org">www.ocpip.org</a></td>
</tr>
<tr>
<td>PI-Bus</td>
<td>Open Microprocessor Systems Initiative (OMI)</td>
<td><a href="http://www.sussex.ac.uk/Units/vlsi/projects/pibus">www.sussex.ac.uk/Units/vlsi/projects/pibus</a></td>
</tr>
<tr>
<td>SiliconBackplane</td>
<td>Sonics Inc.</td>
<td><a href="http://www.sonicsinc.com">www.sonicsinc.com</a></td>
</tr>
<tr>
<td>SoC-it</td>
<td>MIPS</td>
<td><a href="http://www.mips.com">www.mips.com</a></td>
</tr>
<tr>
<td>VSIA on-chip bus</td>
<td>Virtual Socket Interface Alliance (VSIA)</td>
<td><a href="http://www.vsia.com">www.vsia.com</a></td>
</tr>
<tr>
<td>Wishbone</td>
<td>created by Silicore Corp., transferred to OpenCores.</td>
<td><a href="http://www.opencores.net">www.opencores.net</a></td>
</tr>
</tbody>
</table>

Table 4.1: Links to available on-chip multi-point interconnect sources [Sil]

### 4.5.1 Advanced Microcontroller Bus Architecture

Advanced Microcontroller Bus Architecture (AMBA) is an open-standard on-chip bus specification provided by ARM Ltd. that defines a collection of distinct on-chip buses for SoC systems [AMB94].

**AMBA-AHB.** The AMBA Advanced High-performance Bus is a synchronous system backbone.
4.5. SYNCHRONOUS ON-CHIP BUSES

**AMBA-ASB.** The Advanced System Bus is a multi-master bus too, but acts as a general-purpose bus. It can be seen as the low-performance version of AHB and has become obsolete recently.

**AMBA-APB.** The Advanced Peripheral Bus is used to connect many peripheral functions to one single master, which also acts as bridge to the AHB.

Due to the wide use of ARM IP modules, AMBA has been widely adopted throughout the industry. As a consequence, there is support for the development of AMBA based systems from a growing number of companies.

### 4.5.2 CoreConnect

CoreConnect from IBM is a family of three synchronous buses that cover a wide range of performance criteria [IBM99c, IBM99b, IBM99a].

**CoreConnect PLB.** The CoreConnect Processor Local Bus is the high-performance member of this bus family.

**CoreConnect OPB.** The simpler On-Chip Peripheral Bus connects low speed peripheral devices.

**CoreConnect DCR.** Besides their connection to the main bus, all devices are attached to the Device Control Register bus. A DCR serves as a separate low-bandwidth bus to access configuration registers. This allows slow and infrequent configuration operations to proceed without disturbing activity on the other buses.

### 4.5.3 Peripheral Interconnect

The Peripheral Interconnect Bus (PI-Bus) was developed by five of Europe’s major semiconductor companies within the European Union ESPRIT Open Microprocessor Initiative (OMI) project in order to standardise on-chip interconnects [OMI94]. A low-overhead protocol guarantees short response times for time-critical applications.
### Table 4.2: Properties of three different on-chip-bus collections

<table>
<thead>
<tr>
<th>Purpose</th>
<th>AMBA</th>
<th>CoreConnect</th>
<th>PI-Bus</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AHB</td>
<td>ASB</td>
<td>APB</td>
</tr>
<tr>
<td>typical bus freq. [MHz]</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>multimaster capability</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>arbitration</td>
<td>central</td>
<td>central</td>
<td>n/a</td>
</tr>
<tr>
<td>hidden arbitration</td>
<td>yes</td>
<td>yes</td>
<td>n/a</td>
</tr>
<tr>
<td>privilege level</td>
<td>fixed</td>
<td>fixed</td>
<td>n/a</td>
</tr>
<tr>
<td>data width [bits]</td>
<td>32/64/128/256</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>addr. width</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>single cycle transfers</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>separate read/write bus</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>address/data pipelined</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>split transaction</td>
<td>yes</td>
<td>†</td>
<td>no</td>
</tr>
<tr>
<td>burst transfers</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>DMA</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>atomic transactions</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>bus deferral</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>broadcast</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>physical structure</td>
<td>MUX</td>
<td>TRI ‡</td>
<td>any</td>
</tr>
</tbody>
</table>

**Glossary:**
- **BB:** backbone bus; **SB:** system bus; **PB:** peripheral bus; **CB:** configuration bus.
- **n/d:** not defined; **n/a:** not applicable; †: only by retracting from the bus transaction; ‡: others possible.
4.6 Asynchronous Interconnects

4.6.1 MARBLE

MARBLE, short for Manchester Asynchronous Bus for Low Energy, was developed by Bainbridge [Bai00, BF00, BF98] as a backbone for the asynchronous AMULET3i microprocessor [GBB+00]. MARBLE is a self-timed dual-channel bus with centralised arbitration and address decoding. It uses split transaction by default in every transaction. On each channel, bundled data transfers controlled by a 4-phase handshake protocol are performed. Physically, there are two bidirectional tristate-buses. Other forms of access control are possible.

Initiators are bus interfaces that initiate transfers, targets respond via the second channel. The transfers on this second channel occur independently of the ones on the first one, initiated by the target when the answer is available. This decoupled transfer protocol offers a fine-grained interleaving of bus transactions and a better bus availability than the interlocked-transfer technique. Each logical device may either be initiator, target, or both. Arbitration, address and data cycles are pipelined. MARBLE supports deferred transfers for bus bridging, bursts, atomic transfer sequences, and error signalling, but no broadcasts.

Figure 4.6: MARBLE principle [Bai00]
4.6.2 CHAIN

While MARBLE uses single-rail signalling, Bainbridge [BF01] moved towards a self-timed on-chip network topology using 1-of-4 encoded point-to-point channels switched through multiplexers. The use of a 1-of-4 encoding provides delay insensitive signalling and thus guaranteed timing closure that avoids the need for extensive timing verification on system level. CHAIN uses pipelining, narrow links, and low-cost switches. It allows a wide range of network topologies like ring, star, multiplexed, or hierarchical structures. Connections between modules can use one or more CHAIN links, allowing a trade-off between cost and performance.

In a joint project with Cambridge University, Bainbridge’s group is now looking at ways of deploying CHAIN based on-chip networks for GALS design.

4.6.3 Self-timed Ring

Yakovlev et al. developed a self-timed ring [YVMS95] mainly intended for multicomputers. They implemented many of the features offered by IBM's Token Ring (standardised as IEEE 802.5 [Gro85]) but in a self-timed fashion by using delay-insensitive 3-of-6 codes. The ring uses token passing, so only the ring transceiver (they call it ring adapter) holding the token can insert data into the ring. Only one data frame from any of the ring nodes may be in the whole ring at any time. Once a frame has been sent, the sender must wait for a response to this frame before sending the next frame.

4.7 Socket Interface Approaches

For an IP core to be truly reusable, it must be possible to include it in a new system without any changes to the core. Its interfaces must be well defined and ideally be independent from a specific interconnect implementation, because an interconnect-specific (bus-centric) interface limits the market into which an IP core can subsequently be utilised or sold.

The solution to maximise an IP core’s potential reuse is to adopt a core-centric rather than bus-centric protocol as the IP’s native in-
4.7. SOCKET INTERFACE APPROACHES

terface. A core-centric interface, also called socket, enables unconstrained bridging to any interconnect structure that is also equipped with an appropriate socket interface. A comprehensive and scalable interface specification between IP cores and on-chip communication structure allows IP core developers to focus on core generation without any knowledge beyond the interface, allowing complete independence of IP modules. The system designer is also free to choose the on-chip interconnect that best suits the requirements of the application.

4.7.1 Virtual Component Interface

The *Virtual Socket Interface Alliance* (VSIA) [VSI] was formed with the goal of establishing technical standards required to connect virtual components (VC) from multiple sources. VC is basically just another name for an IP module. VSIA claims, no single bus will meet the need of all SoCs. So, they provide a *Virtual Component Interface* (VCI) standard that defines a generic cycle-based address-mapped point-to-point communication protocol. This interface will also connect to a bus, but mandates the use of a bridge, also called bus wrapper. This simple bus wrapper can be designed for almost any standard or proprietary on-chip bus to make it VCI compatible. VCI incurs performance and area overhead though. According to VSIA, the standard will eliminate the need to modify any VCI-compliant VCs when connecting to different VCI-compatible buses.

![VCI socket interface](image)

*Figure 4.7: VCI socket interface*
4.7.2 Open Core Protocol

The open-licensed Open Core Protocol (OCP) is a bus-independent, core-centric point-to-point socket interface that allows functional cores to communicate with each other while maintaining functional independence from one another as well as from the communication structure. In order not to restrict inherent core capabilities, it is scalable and configurable to match different communication requirements associated with different designs. The standard protocol was developed by Sonic and is now promoted and supported by the Open Core Protocol International Partnership Association, Inc. (OCP-IP).

Both OCP and VCI are socket specifications and rather similar in capability and nature. VSIA's Virtual Component Interface is limited to simple data flows though. Therefore, it is up to the designer to deal with the remaining inter-core communications requirements (such as flow control, interrupts, error or test signals) by connecting them in an ad hoc fashion. OCP is a superset of VCI in that it also handles control and test signals in addition to the data flow.

In October 2003, VSIA and OCP-IP agreed on a strategic alliance. While VSIA endorses the OCP interface, OCP-IP becomes the first VSIA adoption group.
A system bus or another form of multi-point data exchange is an important component of a modern SoC design. It provides the necessary modularity to interconnect subsystems.

Synchronous SoC designs most often use shared buses with central arbitration. Clock-skew across the chip, and different timing domains with different clocking requirements are sources of severe problems concerning placement & routing and strongly limits throughput. As described in chapter 3, the GALS method helps to alleviate this problems. Our first GALS implementations only supported point-to-point links. In order to handle large SoCs with a multitude of different IP modules, GALS requires also versatile multi-point on-chip interconnect structures.

To find suitable interconnection alternatives, existing on-chip bus solutions are re-investigated in the context of a GALS implementation. Several options and topologies are taken into consideration. The right decision for a particular topology or protocol is not obvious. It is always a tradeoff between throughput and latency or between throughput and area requirement. In order to cover the variety of cases, three
rather different topologies have been developed:

**MOdular GaLs Interconnect (MOGLI).** MOGLI is a shared bus. Access to the communication medium is handled by a set of GALS ports supporting proper access of the shared media. It works with central arbitration and address decoding. This approach is modular, easy to build, and should offer a self-timed alternative to established synchronous on-chip bus solutions. As the shared bus lines can easily become the bottleneck of the system, this approach is suitable for systems with only moderate communication demands.

**Self-Timed RING for Gals (STRING).** Several GALS modules are connected with a circular path. Dedicated self-timed ring transceivers free the GALS modules from managing en route traffic. A ring topology is a modular, easily scalable solution. It is especially suitable for systems that distribute computation to different subsystems and pass data from one computational block to the next in a regular fashion.

**Switching Network for Gals (SWING).** This is basically a matrix of small self-timed crossbar elements that form the interconnection fabric between GALS modules. It offers concurrent data transfer channels, so achieving a high total throughput. The higher the number of attached modules, the higher the number of provided channels. It is therefore easy scalable in throughput, but always at the cost of increased latency and area.

**Handshaking.** In order to keep the control overhead in tight bounds, all our GALS multi-point communication schemes employ single-rail bundled data encoding with a 4-phase handshake protocol (explained in section 2.2.2) as already used for point-to-point links.

Dual-rail or other delay-insensitive data encoding styles are possible. Unfortunately, the costs of a dual-rail system for wide parallel interconnects often outweighs its benefits. Quasi-delay-insensitive encodings [BTEF03] could become a valuable alternative in the future for long interconnects with high delay variations between single bits.
5.1 MOGLI

Modular GALS Interconnect (MOGLI) is a shared bus approach. It has the lowest complexity of all the presented interconnection solutions and also the lowest area requirement. It can be constructed from a common data transfer channel based on an extension of the GALS point-to-point channels. Figure 5.1 shows a coarse block level diagram of MOGLI, while figure 5.2 depicts the structure of the transfer channel.

![Figure 5.1: Single-channel MOGLI](image)

While the data transfer channel is shared by all the initiators, only one initiator is active at a time. Data is transferred bidirectionally between this single initiator and a single target (the addressed one). The information is always transferred over parallel wires, controlled by a single handshake pair (bundled data). This bundle of data wires together with the handshake lines form the GALS data transfer channel. In order to avoid driving conflicts associated with tristate approaches (see sec. 4.3.2), only unidirectional wires are used on the physical level. Data information from initiator to target and vice versa flow on different wires, but are part of the same transfer channel as shown in figure 5.2. This corresponds to the approach in figure 4.4(b) in section 4.3.3. Access to the shared bus lines is controlled by distributed multiplexers (fig. 4.3).
The information transferred from initiator to target consists of a target address field, eventually an internal sub-address, a control field, and the data payload. The target delivers status information and data back to the initiator. Although this information is always transferred on parallel wires and not serially, it is called data packet and looks as follows:

<table>
<thead>
<tr>
<th>initiator → target</th>
<th>target → initiator</th>
</tr>
</thead>
<tbody>
<tr>
<td>target address</td>
<td>sub-address</td>
</tr>
</tbody>
</table>

The target address is decoded in the central address decoder, the optional field sub-address allows for addressing internal memory locations within the target. The control field encodes the command to the target, in the simplest case just one bit to flag a read or write. A status field allows for a detailed response back to the initiator and the data fields carry the actual data values. All the fields can be adapted in width in some reasonable bounds to accommodate for the application at hand.

Figure 5.3 provides a more detailed view of MOGLI including the GALS ports, bus access, and the handshake lines.
5.1. MOGLI

Figure 5.3: Block-level diagram of a single-channel MOGLI

Initiator and target modules control the data transfer on the bus by means of asynchronous port controllers. The point-to-point GALS output ports (explained in chapter 3) lack the ability to arbitrate for shared resources. They must be extended to be able to act as initiator ports.

On the target side a *not acknowledge* signal is required in order to allow the target to respond even if it is temporary unable to receive data. This reduces bus contention. Details are explained below, when the target ports are described.

5.1.1 Data Transfer on the Bus

With the basic version of MOGLI, each transaction includes only one bus transfer (see sec. 4.4). Address and control information is thus needed for each single data packet. This makes address decoding and forwarding of the handshaking to the appropriate target easy and straightforward.

Figure 5.4 illustrates two bus transactions on MOGLI. Each transaction starts with an arbitration phase, in order to gain control of the
bus. In the example, two initiators try to gain access almost simultaneously. The arbiter first grants access to initiator1 which addresses target1 by rising the transfer request line Rp. The address decoder selects target1, and the request is passed on to the target port controller (SRp1). Target1 is ready to accept data, the port answers on Ap. The initiator restarts its local clock and withdraws BusReq as soon as the data transfer is terminated. The first bus transaction is completed. Now, initiator2 is allowed to use the shared bus lines and addresses target2. Target2 is unable to accept data and answers on the not-acknowledge line NAp. All the same, initiator2 completes the transactions to free the bus and its synchronous island will retry the transmission later.
5.1. MOGLI

The next subsections describe the implementation details of the new GALS elements necessary to build a GALS system communicating on the MOGLI bus. New initiator and target ports in both demand and poll versions are added to the small library of predefined wrapper elements. A central arbiter and an address decoder are also provided as predefined elements to ease the design of complex GALS systems.

5.1.2 Arbiter

MOGLI uses a central arbiter. This is efficient in power consumption and needs little area. All initiator ports are connected to the central arbitration block via dedicated request and grant handshaking lines. A 4-phase protocol is used for this purpose. An initiator which requires access to the bus, rises his request line. The arbiter then grants the bus to only one initiator at a time.

The arbiter consists of a tree of Tree Arbiter Elements (TAEs). Figure 5.5(a) shows an example of an arbiter supporting up to eight initiators. Josephs and Yantchev proposed a speed-independent custom TAE with very low latencies [JY96]. The upper graph of figure 5.5(b) shows a similar tree arbiter element built around a MUTEX [Mar86]. It is slightly slower and not speed-independent but can be mapped to a standard library. The simple circuit around the MUTEX guarantees that the grant signal \( G_1 \) or \( G_2 \) is not released until both the corresponding request signal \( R_1 \) or \( R_2 \) and the grant input \( G \) from the subsequent TAEs is lowered. This grants compliance with a proper full handshake protocol. The TAE version used in MOGLI is shown on the bottom of fig. 5.5(b). It is an extension of the ordinary TAE and provides forward paths from \( R_1 \) to \( R \) and \( R_2 \) to \( R \) in order to bypass the MUTEX. A long decision time of the MUTEX, that occurs when both \( R_1 \) and \( R_2 \) arrive close to each other (addressed in sec. 2.2.6), can so be hidden. By the time the grant signal \( G \) arrives, the MUTEX has most likely settled and thereby enabled the path to the appropriate grant output. The root node of the arbiter tree is reduced to a MUTEX.

The arbitration delay depends on the depth of the tree. In an arbiter build for a maximum number of \( n \) initiators, the path from bus
request to bus grant passes in TAEs where $m = \lceil \log_2 n \rceil - 1$ and the MUTEX, resulting in a total arbitration delay of $t_{pda} = t_{pdmute} + t_{pdTAE} \cdot (\lfloor \log_2 n \rfloor - 1)$.

For the 0.25µm process we used for our implementation, the delay of the MUTEX is $t_{pdmute} = 200$ ps and a TAE accounts for $t_{pdTAE} = 390$ ps.

**Hidden Arbitration.** Arbitration and bus access are sequential. The bus is idle during arbitration for the next bus access. Adding hidden arbitration where the arbitration phase occurs concurrently with the ongoing data transfer of another initiator is straightforward. A slight modification in the extended-burst-mode description of the initiator ports allows for releasing the bus request before concluding
the data transfer. Another initiator can then be announced to gain bus ownership as soon as the ongoing bus activity is completed. To do so, the port has to observe the handshake signals while waiting to access the bus. For our GALS systems with a reasonably small number of initiators, the reduction in latency is too small to compensate for the extra effort necessary in the GALS port controllers. MOGLI does therefore not support hidden arbitration.

Static Priorisation. The arbiter in fig. 5.5 is balanced and therefore treats all initiators equally fair. Asymmetric tree structures as shown in fig. 5.6 allow to assign higher priorities to selected initiators. The closer an initiator is attached to the root, the higher priority it gets. The available throughput can so be distributed as required by the system. An initiator being attached to the Req5/Gnt5 input/output pair in fig. 5.6 would have 4 times higher priority than the others connected to the other four interfaces.

![Figure 5.6: Asymmetric tree arbiter](image)

Dynamic Priorisation. Depending on the system behaviour, the arbiter circuit increases or decreases the priority levels of the different initiators. This could be in reaction to a high priority request from the system ("give initiator xy immediate access, as it has to perform an extremely urgent task") or because an initiator have been waiting for a long time and now gets a higher priority to prevent starvation.
For synchronous buses, it's quite easy to build arbiters with dynamic prioritisation. Building arbiters with dynamic behaviour is not that simple for asynchronous designs. Different solutions have been presented [LYFS97, BKS99, BKY00] and Yang and Ravi [YR90] investigated the performance of different priority schemes. According to them, an arbitration with fixed priority in most cases outperforms complexer schemes. MOGLI therefore sets dynamic priorisation aside.

5.1.3 Address Decoder

To transfer data to a certain target, an unique address is delivered together with the data value. This address is decoded to select the appropriate target. Decoding is generally performed in one of the following ways:

Pre-decoding. Every initiator is equipped with an address decoder. The address gets decoded before being fed to the communication channel. Therefore, a target select line is necessary from all initiators to each target. An advantage is, that the address decoding occurs concurrently with the bus arbitration. A drawback is the high area requirement due to multiple decoders and wiring.

Central decoding. Only one decoder is used. When a valid address is applied the central decoder selects a target by setting the appropriate select line.

Post-decoding. Each target has its own decoder to detect its address. This is a common approach. The area overhead for decoding is not necessarily much higher than with a central decoder because the wiring for target select signals is not needed with this approach.

MOGLI employs a central address decoder. This to avoid both the area overhead of the pre-decoder solution and the difficulties that arise with targets having to observe all transitions of the request signals on the bus\(^1\)

---

\(^1\)by the time they receive a request, they do not know, whether they are addressed. They have to take part in the handshake, receive the address, and decode it to decide on the appropriate reaction.
Decoding is done conventionally with a combinational decoding circuit that is not hazard-free. As depicted in figure 5.7, an event on the request signal $Rp'$ on the timing path ① must not occur at the inputs of the AND gates ③ before the decoder’s target select line $Sel1$ on path ② has properly settled. This constraint has to be carefully checked during timing analysis.

### 5.1.4 Demand-type Initiator Port

Like the GALS point-to-point output ports, an initiator port consists of an Asynchronous Finite State Machine (AFSM) to control the transfer and a flip-flop to flag completion as shown in figure 5.8.
The block diagram of a demand-type initiator port AFSM together with its burst-mode specification is shown in fig. 5.9.

Figure 5.9: A demand-type initiator port controller. Symbol and burst-mode specification

Upon activation by a switching event on the port enable signal \( Pen^+ \) it first requests bus ownership by rising \( BusReq^+ \), suspends its local clock with \( Ri^- \), and changes to state1. The central arbiter answers with \( BusGrant^+ \) when the bus is available. After the local clock stretching is also confirmed by \( Ai^+ \) \( Rp \) is raised and the AFSM proceeds to state2. Upon reception of the acknowledge signal \( Ap' \), \( Rp \) is lowered and state3 is reached. The handshake cycle is terminated with \( Ap' - \), the local clock is restarted by lowering \( Ri^- \) and bus ownership is released with \( BusReq^- \). When arriving at state4 the AFSM has executed an entire data transfer cycle and is idle again. As with the point-to-point port controllers, transition signalling is used on the enable line \( Pen \) to gain high transfer rates. The rest of the states just duplicate the handshake sequence for a transfer started by a falling edge on \( Pen \).

Different to a point-to-point channel, the addressed target may answer with a not-acknowledge \( NAp \) instead of the acknowledge when it is not ready to respond. \( Ap' \) is an OR-combination of \( Ap \) and \( NAp \), as the port AFSM should react the same way in both cases. No matter whether the target is ready or not, it terminates the transfer and restarts the clock. The synchronous island however, only gets
a transfer completion flag \( Ta \), if the target has been able to receive the data packet and has set \( Ap \). If this is not the case, the island re-initiates the transfer with the same data packet.

**Transfer Acknowledge.** None of the asynchronous port controller AFSMs provides direct information for the locally-synchronous island if a GALS data transfer was successful or not. A transfer acknowledge flag \( Ta \) was introduced for that purpose. For the new bus port controllers, the same flip-flop is used as in the point-to-point ports (shown in fig. 5.10).

![Figure 5.10: Generation of the transfer acknowledge signal](image)

The \( Ta \) flag gets asynchronously set by the \( Ap \) pulse and is erased by shifting in '0' with the next rising event of the local clock. As \( Ap \)- does always occur long before \( lclk^{+} \), setup time violations are not an issue. For the initiator port, \( Ta \) is only set if a positive \( Ap \) was detected. It is now up to the synchronous circuit to react depending on \( Ta \).

**Implementation.** Synthesis with the 3D tool set [YD99a] yields the following set of equations, which then get transformed by the simple Perl program “eqn2gate” mentioned in chapter 3 into a technology dependent gate-level netlist.
\[ Ri = Ap + Pen \text{ BusReq } Z_1 + \overline{Pen} \text{ BusReq } Z_1 + \]
\[ Pen \overline{Ai} \text{ BusGrant } Z_0 + Pen \overline{Ai} \text{ BusGrant } Z_0 \]

\[ BusReq = Ri \]

\[ Rp = Pen Ai \text{ BusGrant } \overline{Ap} \text{ BusGrant } Z_1 + \]
\[ Pen Ai \text{ BusGrant } \overline{Ap} \text{ BusGrant } Z_1 \]

\[ Z_0 = Pen Ap + Pen Z_0 + \overline{Ap} Z_0 \]

\[ Z_1 = \overline{Pen} Ap + \overline{Pen} Z_1 + \overline{Ap} Z_1 + Pen \overline{Ap} Z_0 \]

Figure 5.11: Circuit implementation of a demand-type initiator port controller
The burst-mode (BM) specification depicted in fig. 5.9 without the extensions offered by the extended-burst-mode (XBM) is sufficient to describe the desired behaviour (sec. 3.1.2). The port can therefore be synthesised with the Minimalist package [FNT+99] that only supports BM descriptions so far. Synthesis with Minimalist includes optimisation steps and thus often yields smaller or faster circuits than the 3D tool set. This reduces the area requirement from 47 GEs to 32 GEs. The resulting equations can be mapped directly into the circuit shown in fig. 5.11.

The reset input InitxRB is necessary to bring the AFSM into a proper starting state. As the XBM description of the asynchronous controller does not contain any reset mechanism, the signal is added to all gates of the first AND-plane.

**Timing Verification.** Timing verification of burst-mode circuits is an important issue. A *Perl* script creates command files for Cadence static timing analyser *Pearl*, and nicely fits into our design flow [GOV+03].

The AFSM circuits are implemented in a two-level NAND-NAND structure that is simply derived from the basic AND-OR implementation. For safe operation, it has to meet the following internal timing constraints [YD99a] (fig. 5.12):

\[
\begin{align*}
t_{\text{in} \rightarrow \text{out}} + t_{\text{out} \rightarrow \text{outf}} & > T_{\text{in} \rightarrow \text{lit}} & \quad (5.1) \\
t_{\text{in} \rightarrow \text{out}} + t_{\text{out} \rightarrow \text{outf}} + t_{\text{outf} \rightarrow \text{prod}} & > T_{\text{in} \rightarrow \text{prod}} & \quad (5.2)
\end{align*}
\]

where \(t_{x \rightarrow y}\) denotes the minimum delay from a transition of type \(x\) to a transition of type \(y\), while \(T_{x \rightarrow y}\) terms the maximum delay. Equations 5.1 and 5.2 express that no transition triggered by the current input change is allowed to overtake any other transition due to the same input burst until it has passed the first NAND plane. In our implementation, no buffers on the feedback path are necessary to meet the timing conditions, so \(t_{\text{out} \rightarrow \text{outf}} = 0\). When the above constraints are not met a priori, buffers can be inserted into the affected feedbacks paths.
Figure 5.12: Timing requirements for AND-OR implementation of extended-burst-mode finite state machines.

In addition to the internal constraints, the machine’s environment has to satisfy the fundamental mode constraint [YD99a]: As long as the machine has not properly settled after an input change, no further input events are allowed to occur. The cycle time of the AFSM is defined as follows:

\[ T_{cycle} = T_{in \rightarrow out} + T_{out \rightarrow outf} + T_{outf \rightarrow prod} \]  

(5.3)

This is the time needed by the machine to settle if all its slowest paths are triggered and thus a very conservative estimation. To meet the fundamental mode constraint, any feedback path external to the AFSM needs to be slower than the difference between the minimum latency \( (t_{in \rightarrow out}) \) and the cycle time \( T_{cycle} \). To meet such a conservative constraint many of the external feedback paths would require additional delays, severely reducing the system’s performance.
Better performance can be achieved by analysing the timing constraints for each individual state transition. Every single state transition only triggers a subset of timing paths. As only paths that are affected by both the state change and the new input burst have to be checked, such an analysis relaxes the strict constraints.

![Waveforms and timing of the d-type port controller](image)

Fig. 5.13 illustrates the port's operation. The timing figures given below the waveforms are based on pre-layout gate-level simulations. The lower bound for the delay of the external paths is given by $\Delta = T_{\text{cycle}} - t_{\text{in-out}}$. For this type of port controller, $\Delta$ is only greater than zero for the transition ending in state 7. The minimal delay of 55 ps is easily obeyed, as the next transition ($Ap$-) is triggered by the communication partner, and $Rp - \rightarrow Ap$ propagates all the way through the bus channel, through the target port controller, and back.
5.1.5 Poll-type Initiator Port

A poll-type initiator port controller AFSM is shown in Fig. 5.14. It differs from its demand-type counterpart in the way it influences the local clock. After being activated by an event on the port enable signal, a poll port does not stretch the clock until it gets the acknowledge from the target port. When the data transfer is complete, both the bus ownership is released and the local clock is restarted. The bus transfer is terminated, no matter whether the transfer was successful or not, but is repeated if the latter is the case.

![Diagram of a poll-type initiator controller]

Figure 5.14: XBM description of a poll-type initiator controller

3D synthesis generates the following set of equations:

\[
\begin{align*}
Rp &= \text{BusGrant} \overline{Ai} \text{ BusReq} \\
Ri &= Ap \\
\text{BusReq} &= Ap + \overline{Ai} \text{ BusReq} + \overline{Pen} \text{ BusGrant} \overline{Ai} \overline{Z1} + \\
&\quad \overline{Pen} \text{ BusGrant} \overline{Ai} \overline{Z1} \\
Z0 &= \overline{Pen} \ Z0 + \text{BusGrant} \ Z0 + Ai \ Z0 + \\
&\quad \overline{Pen} \ Ap \ Ai \overline{Z1} \\
Z1 &= \overline{BusGrant} \ Z1 + \overline{BusReq} \ Z1 + Pen \ Ap \ Ai \overline{Z0}
\end{align*}
\]
Together with the reset mechanism, this results in an area requirement of 32 GEs.

![Waveform diagram](image)

**Figure 5.15: Waveforms and timing of the poll-type initiator controller**

Both the ports operation and the timing checks of all the state transitions are shown in figure 5.15. Two values are given for \( t_{\text{in} \rightarrow \text{out}} \) when two output transitions occur concurrently in this particular state change. Which signal changes first, can be read out of the figure. Two state transitions pose minimal timing constraints to the environment. A minimal delay of 197 ps from a falling BusReq to a transition on BusGrant is always adhered, the arbiter is always slower than that (see section 5.1.2). 330 ps from falling Ri to falling Ai is more critical. The MUTEX within the clock generation unit is rather fast, and depending on the actual routing, an additional delay on this timing path might become necessary.
5.1.6 Burst Data Transfers

Enhanced versions of the demand-type and poll-type initiator ports are capable of performing burst transfers. Note in this context, that the term “burst” in burst transfer stands for a sequence of data transfers without releasing bus ownership in between (see section 4.4 for details), whereas an input burst in the XBM specification is a non-empty set of edges at the inputs of a burst machine.

Figure 5.16 shows the extended burst-mode specification of a d-type initiator that supports bursty data transfers. The <> symbol in the XBM specification indicates that Burst is a level-sensitive signal. It is kept high by the synchronous island as long as subsequent transfers shall be executed. It gets sampled at the time when Ap' is lowered. This always happens in the time interval the local clock is stopped (state transitions 3→4, 3→9, 7→8, or 7→11). The rising transition on Ai+ and so the last possible rising clock edge occurs sufficiently before Ap' (in transitions 1→2 and 5→6 respectively), so setup and hold time constraints are always met.

The XBM description of the poll version given in fig. 5.17 is similar, with the difference, that the clock is only stopped to synchronise the transfer.

While the support for burst transfers is easy to add to the specification, the resulting circuits get large and are therefore considerable
5.1. MOGLI

![Diagram of a poll-type initiator port with burst transfer capabilities]

Figure 5.17: XBM specification of a p-type initiator with burst transfer capabilities

slower. Table 5.1 compares demand and poll initiator ports in area and timing. A demand-type port with support for burst data transfers requires more than four times the area of the ordinary initiator, and it roughly needs twice the time to complete a data transfer. The figures look similar for the poll port: more than two and a half time higher area requirement at half the speed.

<table>
<thead>
<tr>
<th></th>
<th>area [GE]</th>
<th>average ( t_{in\rightarrow out} [\text{ps}] )</th>
<th>( T_{cycle} [\text{ps}] )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>demand</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>no burst cap.</td>
<td>32</td>
<td>493</td>
<td>499</td>
</tr>
<tr>
<td>with burst cap.</td>
<td>142</td>
<td>924</td>
<td>1226</td>
</tr>
<tr>
<td><strong>poll</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>no burst cap.</td>
<td>31</td>
<td>330</td>
<td>380</td>
</tr>
<tr>
<td>with burst cap.</td>
<td>85</td>
<td>715</td>
<td>735</td>
</tr>
</tbody>
</table>

Table 5.1: Area and timing figures for the initiator port controllers based on a 0.25\(\mu\)m CMOS process

To make sense, the time for a bus transaction of \( m \) burst data transfers must be lower than \( m \) ordinary bus transactions. In the
former case arbitration is done only once for the whole burst, in the latter case for each data transfer. The arbitration time for a 0.25µm process is $t_{pdArbitration} = 200\, ps + 390\, ps \cdot ([log_2n] - 1)$, where $n$ is the number of initiators. The time needed for an entire data transfer is 1.98 ns for the standard demand-type initiator port and 3.36 ns for the demand version with burst transfer support. The relations are similar for the poll-versions: 1.48 ns for the standard initiator port and 2.96 for the port supporting burst transfers. This leads to the following inequations:

For demand-type ports:

$$m \cdot 3.36\, ns + t_{pdArbitration} \leq m \cdot (1.98\, ns + t_{pdArbitration})$$  \hspace{1cm} (5.4)

Evaluated for $m$:

$$m \geq \frac{t_{pdArbitration}}{t_{pdArbitration} - 1.38\, ns}$$  \hspace{1cm} (5.5)

For poll-type ports:

$$m \cdot 2.96\, ns + t_{pdArbitration} \leq m \cdot (1.48 + t_{pdArbitration})$$  \hspace{1cm} (5.6)

Evaluated for $m$:

$$m \geq \frac{t_{pdArbitration}}{t_{pdArbitration} - 1.48\, ns}$$  \hspace{1cm} (5.7)

Both inequations can only deliver a reasonable $m$ for arbitration delays greater than 1.38 ns and 1.48 ns respectively. This holds true for a depth of the arbiter tree of 5 or greater including the root. Such an arbiter can already handle up to 32 initiators, which is a high number even for modern SoCs. The number of burst data transfers $m$ must then be at least 5 (demand) or 7 (poll) to gain higher throughput.

If a specific application asks for connections with very high data rates and the initiator accesses only a few targets, dedicated point-to-point connections can be applied to bypass the shared bus. If an initiator
poses high demands in bandwidth to many or all targets, a priorisation of that initiator as seen above in sec. 5.1.2 can largely reduce the arbitration time.

If burst transfers are indispensable, the arbitration sequence should be handled by a separate port. A possible arbitration port is depicted in fig. 5.18, whereas for the data port a standard point-to-point port can then be used.

**Dedicated Arbitration Port for Burst Transfers.** The specification for the arbitration port is given in fig. 5.18, and the resulting output and state equations follow below. This approach was not implemented on our test beds.

\[
\begin{align*}
R_i &= \overline{A_i} R_i + \text{ArbEn \ BusGrant} \overline{Z_0} \\
\text{BusReq} &= A_i + \text{ArbEn \ BusReq} + \text{ArbEn \ BusGrant} \overline{Z_0} \\
Z_0 &= A_i \text{ BusGrant} + \text{ArbEn} \overline{Z_0} + A_i \overline{Z_1} \\
Z_1 &= \overline{A_i} Z_1 + \text{ArbEn \ BusGrant} \overline{Z_0}
\end{align*}
\]

The signal flow is given in fig. 5.19 together with the timing behaviour. \( t_{\text{in\rightarrow out}} \) specifies input to output latency in the corresponding state, \( T_{\text{cycle}} \) denotes the cycle time of the state machine, and \( \Delta \) state the minimal time the environment must meet from output change to the next change at the inputs. A \( \Delta \) of 18 ps is no problem, already the wiring accounts for a lot more even in the fastest case.
### 5.1.7 Demand-type Target Port

The data transfer request $R_p$ and the port enable $P_{en}$ in fig. 5.20 come from different clock domains ($R_p$ from a bus initiator, $P_{en}$ from the target’s synchronous island). Extended burst-mode AFSMs can handle a moderate degree of concurrency in that all the signals within an input burst can change in any order [YDN93]. But XBM cannot model situations where the FSM should change to different states depending on the sequence of totally independent inputs\(^2\). An arbitration circuit is therefore needed to guarantee proper operation.

The demand-type target port is therefore built from an ordinary demand-type input port combined with a MUTEX that handles the arbitration. The XBM specification of the demand-in port can be found in chapter 3, page 41.

Two handshaking sequences are possible: one on $R_p/A_p$ if the target is ready to receive and one on $R_p/N_{Ap}$ if the target is unable to react on the incoming data. The port’s timing is shown in fig. 5.21.

---

\(^2\)The different input signals could change simultaneously.

---

![ArbEn, BusReq, BusGrant, Ri, Ai, Z0 waveforms](image)

<table>
<thead>
<tr>
<th>t(_{in-out}) [ps]</th>
<th>303/312</th>
<th>240</th>
<th>148</th>
<th>303/312</th>
</tr>
</thead>
<tbody>
<tr>
<td>T(_{cycle}) [ps]</td>
<td>303</td>
<td>240</td>
<td>176</td>
<td>303</td>
</tr>
<tr>
<td>(\Delta) [ps]</td>
<td></td>
<td></td>
<td></td>
<td>18</td>
</tr>
</tbody>
</table>

Figure 5.19: Arbitration port controller waveforms
The numbers in the circles denote the current state of the input port AFSM. Upon port activation by an event on Pen the AFSM asserts Ri+ to stretch the local clock. In the example given, Rp changes at the same time and arrives at the input of the MUTEX before Ri+ does. Rp+ is granted by the MUTEX on its output G1, a handshake on Rp and NAp takes place. In the meantime, the clock stretching is confirmed on Ai and the synchronous island is idle while the AFSM waits for a rising edge on RpInt. The current handshake on Rp/NAp is kept uninterrupted, RpInt can only be set after completion of the ongoing unsuccessful transfer. After Rp is lowered, Ri can pass the MUTEX and its output G2 is set. The AND-gate can assert RpInt as soon as the next transfer request indicated by a rising edge on Rp arrives (Rp can not pass the MUTEX at this point of time), and the AFSM completes the GALS data transfer with a full handshake on Rp and Ap. The timing figures in fig. 5.21 describe both the input to output timings of the complete port controller (the grey area) as well as the input to output latencies and cycle times of the AFSM alone. Due to the fast MUTEX, the delay from Rp to NAp only accounts for 200 ps.
Figure 5.21: Waveforms and timing of a demand-type target port

5.1.8 Poll-type Target Port

A poll-type target port AFSM and its XBM specification is depicted in figure 5.22 and the corresponding waveforms in fig. 5.23.

Figure 5.22: XBM specification of a poll-type target port controller
**5.1. MOGLI**

The target waits for being addressed with a request $Rp$ from an initiator port and responds with a full handshake cycle on either acknowledge $Ap$ or not-acknowledge $NAp$ signal depending on the state of the ready signal $Rdy$. Different from the demand-type target port, the port does not have to directly react on a signal from the synchronous island but always waits for a transfer request. To make the port controller as small and fast as possible, the AFSM simply stretches the local clock every time it receives a rising edge on $Rp$ and samples the ready signal $Rdy$ from the synchronous island after reception of $Ai+$. If $Rdy$ is high the transition leads to state2, if it is low to state4. $Rdy$ must be stable around the rising edge on $Ai$. This is the case, as the local clock is stopped at this point of time.

---

**Figure 5.23: Waveforms and timing of a poll-type target port**

<table>
<thead>
<tr>
<th>$t_{in-out}$ [ps]</th>
<th>195</th>
<th>703</th>
<th>120/154</th>
<th>195</th>
<th>472</th>
<th>105/154</th>
<th>195</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{cycle}$ [ps]</td>
<td>195</td>
<td>703</td>
<td>154</td>
<td>308</td>
<td>472</td>
<td>239</td>
<td>195</td>
</tr>
<tr>
<td>$\Delta$ [ps]</td>
<td>34/0</td>
<td>113</td>
<td>134/85</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Synthesis yields a rather small state machine that accounts for only 24 GEs. The output and state equations are given below:

\[
R_i = R_p \overline{A_i} + R_p R_i \\
A_p = R_p A_i R_i Z_1 \\
N_{A_p} = R_p Z_0 \\
Z_0 = R_p Z_0 + R_{dy} R_p A_i \overline{Z_1} \\
Z_1 = \overline{R_p} A_i + \overline{R_p} Z_1 + A_i Z_1 + R_{dy} A_i R_i \overline{Z_0}
\]

A synchronous island acting as a bus target keeps \( R_{dy} \) high as long it is ready to accept new data. As the clock is stretched during the transfer, the target can not operate on the received data and sends back an answer and/or status information to the corresponding initiator within the same bus transaction. A target module that both receives and sends data on the bus is therefore equipped with both a target and an initiator port.

Exceptions are memory modules attached to the bus. Simple memory interfaces for ROMs and RAMs can be as simple as shown in fig. 5.24. The can be used both for point-to-point as well as for MOGLI data transfer channels and replace the target ports. The delay must be large enough to compensate for the memory access time under all conditions in the desired range of operation conditions. If a RAM provides a ready signal, this signal can be used to generate the acknowledge back to the initiator.

Targets that do not send the acknowledge signal before they are ready to send an answer are also possible. They sample incoming data on reception of a rising edge on \( R_p \), stretch the local clock if necessary and release the clock again for working on the received data. As soon as the synchronous island reports completion to the port controller, the answer is transferred, indicated by setting \( A_p \). In this phase the clock is also stretched if necessary for synchronisation. However, with this solution the bus remains locked as long as the target circuit is busy. A better alternative is to use a second independent back channel from target to initiator as described below.
5.1. MOGLI

5.1.9 Dual Channel Implementation

MOGLI as introduced above works with a single data transfer channel. It can be extended to a dual-channel version which allows to increase throughput and improve bus availability.

![Dual-channel MOGLI with separate data transfer channels for command and response](image)

Figure 5.25: Dual-channel MOGLI with separate data transfer channels for command and response
Such a dual-channel version with separate buses for the direction initiator to target (command channel) and target to initiator (response channel) is depicted in figure 5.25. With this approach the bus initiator sends a command to the target on the command channel. When the target has completed its operation, it responds with status and data on a second channel called response channel. This corresponds to the approach in figure 4.4(c) in section 4.3.3. The whole data packet is transferred on parallel data wires as shown in figure 5.26.

Figure 5.26: Separate data transfer channels for command and response

Each command packet includes target address, internal sub-address if needed, a control field and the data value. Additionally, a tag with the initiator's address is necessary (including an internal sub-address, if needed). So, the receiving module knows where to send data and status to.
The target answers on the separate response channel with this packet:

<table>
<thead>
<tr>
<th>initiator address</th>
<th>target tag</th>
<th>sub-address</th>
<th>status</th>
<th>read data</th>
</tr>
</thead>
</table>

A separate arbiter and address decoder are necessary for the response channel, it can therefore be seen as a separate bus. Note, that the targets access the response channel with initiator ports, because they have to act as an initiator on the response channel, including arbitration. The initiators use target ports on the receiving side.

The decoupling between command and response is called split-transfer. The bus is released while the target is processing the command and is available for other data transfers. This is of special interest when accessing slow targets. Command and response phases from different transfers may overlap. Such overlapping is actually found in most parallel synchronous buses that use separate wires for address and data. In a self-timed environment the skew between an address and response cycle can be varied whereas in a synchronous system it is fixed to a multiple of the bus clock period.
5.2 Self-timed Ring

In a ring topology, all nodes are connected through a circular path. At each node local address decoders decide whether a data word is bound for itself, or if it is to be passed on to the successor node. Every node can also insert new data onto the ring.

5.2.1 Feed-through Structure

![Diagram of a self-timed ring composed of point-to-point links]

Figure 5.27: Self-timed ring composed of point-to-point links

A self-timed ring for GALS systems can be constructed from point-to-point transfer channels as depicted in figure 5.27. Every GALS module on the path from the sender to the receiver is involved in a data transfer as a repeater. It has to synchronise its local clock to the incoming data, therefore interrupting internal computations. An address decoder within the synchronous island then decides whether the data can be consumed locally or must be passed on to the successor node. If a node is idle with its local clock stopped when it receives data, it must be restarted. This approach is not energy efficient and is therefore not further elaborated.
5.2.2 Bypass Structure

A more adequate approach is to decouple the GALS modules from the ring's timing and to unburden them from time consuming re-routing tasks. Such a ring is depicted in figure 5.28.

![Diagram of Self-timed ring operating with bypass transceivers](image)

Figure 5.28: Self-timed ring operating with bypass transceivers

The ring transceivers shown in grey mainly consist of two parts: a router ① that decides where the incoming data packet has to go and an arbiter element ② that decides which request to pass (incoming request from the preceding ring transceiver, or request from the host circuitry that wants to feed a data packet into the ring). The modules are connected with two point-to-point data channels to the link, one for each direction. Such a ring transceiver with bypass capability is described in detail in the next section.
A self-timed ring introduced by Yakovlev et al. [YVMS95] was addressed in section 4.6.3. In contrast to their work, the solution presented here relies on rather basic data transfer protocols in order to leave as much functionality as possible to the synchronous part. Different to a token ring, all nodes can insert data packets into the ring whenever the involved ring segment is empty. By doing so, a ring of N nodes contains up to N data packets simultaneously travelling through the ring and not at most one as it is the case with the Token Ring. However, this implies that deadlock precautions must be taken within our synchronous islands on a higher level of network layer hierarchy.

The data packets travelling on the ring consist of different fields: target address (including a module internal sub-address if desired), source address (also with sub-address if appropriate), control & status information, as well as the actual data value. The source address provides the information in order to know where to send back the response. All the bits are transferred concurrently on parallel wires. The number of bits in the address field determines the maximum number of nodes a ring can have.

<table>
<thead>
<tr>
<th>target address</th>
<th>source address</th>
<th>control/ status</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>internal</td>
<td>node</td>
<td>internal</td>
</tr>
</tbody>
</table>

The address comparator of the ring transceiver decodes the node address to determine if the data packet is bound for the attached GALS module or not. If not so, the data packet is passed on to the next node as it is.

If the address space is not exhaustively used, special care must be taken that an access to unused addresses is properly handled. One transceiver should be foreseen to remove all ghost data packets in order to prevent data packets from travelling around the ring indefinitely. An existing ring transceiver or a dedicated one can take over this task. An alternative is to count the number of passed transceiver. The data packet is dropped once it traversed the entire ring.
5.2.3 Data Transfer

Figure 5.29 depicts an example of a possible data flow scenario. The corresponding waveforms can be found in fig. 5.30.

Three data packets are received at the input interface of the ring transceiver. The target address field of the second data packet matches the node’s address, so this packet is sent to the GALS module (**Rec-DataPacket**). For proper operation **Match** must be stable before the rising edge of **LeftReq** (indicated by the two areas checkered in light grey). **Packet1** and **Packet3** are not allotted for this particular node, they are passed on to the next node. When the GALS module wants to insert data onto the ring, it applies **Packet4** and the port controller sets **SendReq**. This is only acknowledged, if the arbiter passes the request on to the right output. So the sender has to wait till **Packet1** is safely transmitted as **ArbReq** arrived earlier than **SendReq** (hatched area in grey). The multiplexer responsible for merging the **ArbDataPacket** and **SendDataPacket** is controlled by the **MuxSelect** signal. At last, **Packet3** travelling through the transceiver is transmitted.

**Deadlock Preventions.** The pipeline stages must not be overfilled in any case in order prevent a deadlock in the ring. Once a frame has been sent, the sender must wait for that frame to be acknowledged by the receiver before sending the next frame. The number of pipeline stages can be increased, allowing a higher number of command data packets that can be inserted in the ring before the sender gets the corresponding answers (called outstanding transactions).
Broadcast. To enable broadcasting, a dedicated broadcast address can be added. Each transceiver also detects the broadcast address besides its own node address and transmits data both to the attached GALS module and to the subsequent transceivers. The sender's transmitter annihilate his broadcast message to prevent it from indefinitely circulating in the ring. Similar to the Token Ring [Gro85] approach, each ring node has an acknowledge bit in the control/status field, that is asserted when the reception of the broadcast packet was successful.
5.2. **SELF-TIMED RING**

When the sender gets the packet back, it checks, if all the acknowledge bits are set. This approach comes at the cost of n additional lines for a ring of n nodes and is not supported in our test implementation.

### 5.2.4 Ring Transceiver

Within this section the implementation details of the ring transceiver are explained. The block diagram in fig. 5.31 depict a ring transceiver (in grey) with attached GALS module.

![Self-timed ring transceiver with attached GALS module](image)

**Figure 5.31: Self-timed ring transceiver with attached GALS module**
The address comparator and the select circuitry in the transceiver decide where an incoming data packet has to be passed on. The arbiter can be found towards the bottom of the transceiver block.

The input and output ports in the GALS module are standard demand or poll-type point-to-point ports. The input port receives the data that is bound for the functional circuit within the locally-synchronous island and the output port is responsible for feeding outgoing data into the ring. Both ports act independently, therefore the module can send and receive data concurrently.

**Select circuit.** The select circuit depicted in figure 5.32 passes all incoming requests (\( \text{LeftReq}\ast \)) on to the receiving input port of the module (\( \text{RecReq} \)) or to a pipeline stage within the ring transceiver (\( \text{TransReq} \)) depending on the value on the \( \text{Match} \)-signal from the address comparator.

![Select circuit diagram](image)

Figure 5.32: Select circuit to route data packets to the appropriate destination
To ensure correct operation, it is important that the request signal does not arrive before the Match-signal has become stable. In our first implementation this is ensured by inserting a matched delay into the request line. This timing has to be carefully checked by a static timing analyser after layout and routing.

One of the Muller C-elements shown in the circuit diagram is an asymmetric variant of the standard C-element (described in sec. 2.2.4). The input labelled with a “+” sign only affects the rising output transition.

When a GALS module attached to the bus is blocked or defective for some reasons, it does not respond to a request on its input port. This can block the whole ring as the pipeline stages get filled up with data packets from the 'dead' node backwards to all the senders. To avoid such a situation under all circumstances, the input port of the GALS module and the select circuit of the transmitter are modified in such a way that the transceiver can autonomously flag the refused delivery back to the sender.

In order to detect whether the port is ready to receive, the select element must be informed on the status of that input port. Compared to the standard GALS transfer channels, where always the sender initiates the transfer, the directions of the RecReq and RecAck handshake signals are interchanged. The input port controller now sends a request RecReq as soon as it is ready to receive, the select circuit answers with the acknowledge RecAck when a data packet is available. So, this data channel from ring transceiver to locally synchronous island works with a pull instead of a push-type communication scheme (sec. 2.2.1).

In the select circuit both the RecReq and the LeftReq inputs are connected to the MUTEX that decides which signal has come first. If it was RecReq and Match is high, the handshake is acknowledged to the input port controller. If RecReq is not active by the time, TransReq is activated instead. If Match indicates a matching address, the flipflop is asynchronously set. The output of this flipflop (Refuse) is used to control the multiplexers that change certain fields in the data packet.

Figure 5.33 depicts how the fields in the data packet are changed. Target and source addresses are exchanged in order to send the data back to the origin. The appropriate bits in the status field indicate
that the receiver is momentarily unavailable. The *Refuse* signal remains stable until the flipflop is cleared by the falling edge of *TransAck* that indicates the end of this particular handshake cycle. This ensures that the multiplexers do not switch during an active bus transaction.

**Pipeline Stage.** The pipeline stages within the ring transceiver are necessary to store data packets travelling on the ring. Two pipeline stages together store one data packet in a master-slave fashion. One latch is transparent and the other hold the values. The first pipeline stage in the transceiver of fig. 5.31, together with the final stage of the preceding node allows the data packets to move forward without blockage. The two pipeline stages in the middle of the transceiver keep the packet while waiting for arbitration. They liberate the path to the input port of the GALS module for incoming data packets when the transfer through the ring of the preceding packet is still pending.

To achieve maximum performance, the number of pipeline stages per data packet (called valid token in a pipeline) must match the dynamic wavelength [Wil90], which is given by equation (5.8),
5.2. SELF-TIMED RING

\[ W_d = \frac{P}{L_f} = \frac{2L_f + 2L_r}{L_f} \tag{5.8} \]

where \( P \) denotes the period i.e. the delay for a complete handshake cycle. The throughput \( T \) is the inverse thereof. \( L_f \) is the forward latency of the stage (delay from request in to request out) and \( L_r \) depicts the reverse latency (acknowledge in to acknowledge out). This thesis can not go into details, but there are many publications concerning timing and performance in pipelines and rings. The interested reader is referred to papers on pipeline and ring performance issues by Williams [Wil90, Wil92] or to chapter 4 of “Principles of Asynchronous Circuit Design” [SF01]. Instead of being evenly spaced, data flow in self-timed rings often show a bursty behaviour. This phenomena is investigated and remedy is addressed in [WGG02].

For our implementation, a \( L_f \) of 856 ps and a \( L_r \) of 964 ps was measured in post-layout simulation. This results in a \( W_d \) of 4.25. Thus, four pipeline stages per travelling data packet result in the maximum performance achievable.

**Arbiter.** A similar tree arbiter element (TAE) as used in MOGLI’s arbiter tree (sec. 5.1.2) is responsible for deciding which of the data packets waiting at the inputs can proceed: the packet to insert into the ring from the module or the passing data within the ring. The TAE is extended with an output \( MuxSel \) to control the multiplexer
that combines the data streams already travelling in the ring and the one from the attached GALS module.

![Arbiter to control access to the ring](image)

**Figure 5.35:** Arbiter to control access to the ring

The arbitration method is fair amongst the two inputs and results in a 'first come first served' scheme. It is therefore guaranteed that the module can insert a data packet into the ring every second handshake cycle on RightReq and RightAck, even when the ring is extremely busy. This is the case because the MUTEX element sees a logic1 at its pending input when the other ongoing data transfer is about to terminate. As soon as the request line on R2 is lowered in reaction on a rising edge on G2, R1 can pass the MUTEX and the next acknowledge goes to G1.

### 5.2.5 Dual Channel Implementation

When the receiver module sends back a response to the sender (which he knows from the source address field within the data packet), the response packet has to travel on the same ring as the command packets. Command and response packets share the total throughput of the ring. To double throughput and further decouple command and response packets, a second ring can be connected to the module for sending the response as illustrated in fig 5.36. Obviously, this doubles also the area required.
Figure 5.36: Dual channel self-timed ring
5.3 Self-timed Switch

A switch routes data between any initiator and any target (fig. 5.37). A switch based solution is preferred as interconnect, when the system has high demands in throughput. In order to offer as much throughput as possible, the switch operates in pipelined fashion and keeps the single wire segments between successive pipeline stages as short as possible. This largely reduces their capacitive load.

A self-timed switch can be built from a matrix of smaller self-timed crossbar switches as denoted in fig. 5.38. This crossbar switches contain the pipeline stages, address decoders, and arbiters to handle the data flow through the switch. Our implementation uses a matrix of small 2-input 2-output crossbar switches, but other fragmentations are possible.

In this switch matrix, every data transfer is pipelined. Like in the self-timed ring, the data packets move forward from sender to receiver module stage by stage. Obviously, this does not permit the use of bidirectional transfer channels. A dual-channel approach as introduced in section 4.3.3, fig. 4.4(c) is therefore used. Two separate switch matrices as shown in fig. 5.37 exchange data between initiators and targets and vice versa. One switch matrix routes command data...
5.3. SELF-TIMED SWITCH

![Diagram of self-timed switch](image)

Figure 5.38: Switch composed of self-timed crossbar switches (only the command channel is depicted, the grey arrows are data transfer channels including the handshake lines)

Packets to the targets, the other is responsible for the answers back to the initiators.

A whole data packet is transferred on parallel data wires as shown in figure 5.39. There are distinct command and response channels with separate handshake control. Each data packet includes target address, initiator address (in order to determine, where to send the response to), control and data field. It is easy to see in this context, that the distinction between initiator and target is actually no longer necessary. There are simply two groups of GALS modules that are connected to both the unidirectional switch matrixes. One switch provides connections from group1 to group2, the other in the opposite direction. Every module can initiate a transfer on its output GALS port and receives messages on the passive input port. The command and response data packet are therefore equal:

<table>
<thead>
<tr>
<th>target address</th>
<th>initiator address</th>
<th>control/status</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>module internal</td>
<td>module internal</td>
<td>status</td>
<td></td>
</tr>
</tbody>
</table>
A GALS module receives a packet, operates on the data while the data channels are freed for other transfers, and then sends back a response. This decoupling of command and answer is called split transaction. The split transaction scheme can easily be extended to support a number of outstanding commands. An outstanding command is one, that was not yet acknowledged with an answer packet by the target. If the initiator sends more than one command before it gets them acknowledged, the modules have to deal with several outstanding commands. To be able to assign the answers to the associated commands, a sequence number is added to the data packet:

<table>
<thead>
<tr>
<th>target address</th>
<th>source address</th>
<th>control/status</th>
<th>seq. num.</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>module</td>
<td>internal</td>
<td>module</td>
<td>internal</td>
<td>status</td>
</tr>
</tbody>
</table>

Standard demand or poll-type point-to-point GALS ports are sufficient.
for handling the data transfer at the GALS modules. The initiators use output ports on the command channel, the targets receive with input ports. On the response channel, the targets initiate the transfers with output ports and the initiators receive the answers via input ports.

5.3.1 Self-timed Crossbar Switch

Figure 5.40 shows a 2-input 2-output crossbar switch. The two pipeline stages on each input channel provide the necessary storage for one data packet in case the packet has to wait for access when the second input channel is occupying the channel. Without any storage elements, the preceding switching element could not pass on the data on its output and the whole path back to the sender would get blocked. The same arbiter and pipeline stage controllers as in the ring transceivers can be used.

![Figure 5.40: Basic self-timed crossbar switch](image)

With this structure the two inputs have to compete for the single data channel between the MUX and DEMUX in fig.5.40, even if the data flows are not crossed, i.e. LeftData0 is going to RightData0 and LeftData1 to RightData1.
The switch depicted in fig. 5.41 avoids this bottleneck. It provides two data paths with separate address decoders for each input. Arbitration is only necessary, if two data packets should both be routed to the same output interface. This enhanced concurrency decreases collisions and boost the average throughput at the cost of a higher area requirement.
Chapter 6

Implementation of a Test Chip

6.1 Test System

The GALS test chip "Shir Khan" was mainly implemented to test and compare the newly developed interconnection structures. A GALS system acting as a test bed for a variety of interconnection topologies must ideally be modular and flexible. An array of programmable elements neatly fulfils this requirement. On Shir Khan 25 GALS modules are arranged in a $4 \times 7$ grid as highlighted in fig. 6.1. Each module is composed of a clock generator, a locally-synchronous island containing a specialised test processor, all the necessary GALS ports, and a circuit to handle initialisation and programming of the processor. This modules are interconnected by a variety of different interconnections. The labels in the highlighted rectangles indicate to which interconnection architecture the module is connected to.

The remaining three rectangles shown as dashed boxes contain a fully synchronous test processor, a new local clock generator for test purposes, and standard cell rows. The rows contain all the necessary components for the interconnects, such as access control, arbiters, address decoders or ring transceivers.
Figure 6.1: Chip micrography of Shir Khan. The individual GALS modules are highlighted. The slots depicted by dashed rectangles are occupied by the interconnects (IC) and other test structures.

The chip was produced in a standard 0.25μm CMOS process, occupies an area of 25mm², and has a complexity of roughly 3 Million transistors.

The small test processor named PortProcessor acts as main test circuit and provides the stimuli used to test the GALS interconnections and additional GALS elements. A processor is flexible enough to simulate a variety of different behaviours. It has been custom designed to
6.1. **TEST SYSTEM**

keep it small and easy programmable. Its specialised instruction set is optimised for efficiently handling the GALS communication ports. The small size enables a large number of modules to be integrated, so studying a fairly complex system. The processor runs at a clock frequency up to 250 MHz. As simulations showed, this is sufficient for all performance tests of the interconnects. Details of the architecture and memory structure can be found in appendix B.

![Figure 6.2: Basic block diagram of the PortProcessor](image)

6.1.1 **Implemented Interconnection Architectures**

The top level hierarchy of the system consists of the three different multi-point interconnection solutions described in chapter 5 and versions thereof. Multiple GALS modules are connected by an interconnect architecture and together form a test system to measure performance figures of the corresponding interconnect. To save area, a few modules are connected to more than one interconnect structure. The following architectures are implemented on Shir Khan:
Single-channel MOGLI. A single-channel MOGLI connects 3 initiators to 3 targets. A coarse block diagram is given in figure 6.3.

![Diagram of the single-channel MOGLI system](image)

**Figure 6.3:** Single-channel MOGLI. Test system implementation

Figure 6.4 shows how the three initiators are connected to the central tree arbiter. Initiator2 has a higher priority than the others and claims half the total throughput.

![Diagram of the initiators connected to the arbiter](image)

**Figure 6.4:** Schematic, showing how the initiators are connected to the central arbiter
6.1. TEST SYSTEM

Dual-channel MOGLI. A double-channel version of MOGLI with 3 initiators and 3 targets\(^1\) is implemented to compare it to the single-channel version. Figure 6.5 depicts the basic structure.

\[\text{Command channel} \quad \text{Response channel}\]

\[\begin{array}{c}
\text{Arbitration} \\
\text{Addr. Decoder}
\end{array}\]

Initiator0 \quad Target0

Initiator1 \quad Target1

Initiator2 \quad Target2

Figure 6.5: Double-channel MOGLI. Test system setup

Single-channel MOGLI with AMBA AHB compliant interfaces. Shir Kahn contains a self-timed bus with AMBA AHB (see section 4.5.1) compliant bus interfaces to offer a self-timed replacement for the AMBA bus. This is important to cover the large class of IP-modules on the market with AHB interfaces. Only part of the protocol features are supported up to now.

AMBA AHB uses separate address and data lines with overlapped address and data phases as depicted in fig. 6.6. In order not to cut throughput by half, the two phases are not mapped to two self-timed data transfers. Two data channels are used instead to transmit address and data in the same interleaved fashion. A separate port handles the arbitration sequence. Small synchronous FSMs translate the AHB signals into enable signals for the GALS ports.

\(^1\)Two initiators and two targets of single and double channel versions are shared so that a total of 8 GALS modules (instead of 12) are used to realise the two buses.
CHAPTER 6. IMPLEMENTATION OF A TEST CHIP

Figure 6.6: AMBA AHB bus protocol, showing the decoupling of address and data transfer phases

STRING. A self-timed ring topology in a single-channel version is realised. It consists of 5 ring nodes.

Figure 6.7: STRING: self-timed ring topology connecting 5 nodes in circular fashion

SWING. A switch based network connecting 3 initiators and 3 targets is implemented. Two separate switches are necessary for the command and response channel. Two 4 input-4 output switch matrixes are placed on the chip. The unused inputs and outputs are connected to a special circuit that autonomously responds with an error message, if these unused addresses are accessed.
6.1. TEST SYSTEM

Figure 6.8: SWING: self-timed switch connecting a total of 6 GALS modules

6.1.2 Additional Test Structures and GALS Components

A stand-alone PortProcessor and an improved clock generator is placed on Shir Khan for test and verification purposes. In addition, Shir Khan includes a number of newly developed GALS components for initial test purposes:

Interfacing fixed-clock islands To make GALS as universal as possible, it needs means to interface with islands or external components driven by a fixed clock, e.g. at video or audio sample frequencies. This is achieved by omitting the acknowledge line while maintaining the ability to synchronise the local clock at the receiver side to the incoming data.

A schematic of a demand-type port is depicted in figure 6.9 and the poll-type version thereof in fig. 6.10. The sender on the left has no means to adapt its fixed clock. With Rp it signals the availability of new data. The receiver on the right is a GALS module with a stretchable local clock. It synchronises its local clock to the incoming data by stretching the clock if necessary. As no acknowledge line back to the sender is used, the receiver must under all circumstances be fast enough to complete its operations before the next data transfer
is requested by the sender.

Figure 6.9: Schematic of a demand-type single-sided GALS transfer channel and the XBM specification of the port controller AFSM

The demand-type port has an enable signal for a first activation\(^2\). The port can not be disabled anymore because it has to keep trace of the incoming data stream. The poll version does not need this enable feature, as its local clock is running between data transfers. A transfer acknowledge signal \(Ta\) indicates the reception of new data to the synchronous island.

In Shir Khan, two GALS modules contain one-sided port connections that are directly routed to external pins. The external tester

\(^2\) e.g. for an initialisation routine running in the receiver’s synchronous island.
6.1. TEST SYSTEM

Hardware acts as sender with fixed clock.

Buffered Transfer Channels Other GALS approaches require FIFOs for synchronisation [LPI00, MTC+00, CN01]. Our transfer channels do not, but can be equipped with FIFOs to increase the “elasticity” between modules. This is especially useful when the receiving block is heavily occupied with other tasks or pending transfers and can only cope with the incoming data stream in the average.

Pipelines can act as asynchronous FIFOs. In this case, our point-to-point data transfer scheme and the standard port controllers can be used. While handshaking and throughput are extremely fast, the drawback of such a pipeline is the high latency, especially when it is sparsely occupied because data items have to ripple through the whole pipeline.

![Schematic of a buffered GALS channel with demand-type behaviour on both the sender and receiver side](image)

Figure 6.11: Schematic of a buffered GALS channel with demand-type behaviour on both the sender and receiver side

The ring FIFO proposed by Norwick can be used to reduce latency [CN01]. To stick to the standard cell library as much as possible, an available dual-port FIFO with distinct read and write clocks was used instead. See fig 6.11 and fig. 6.12 for the schematics. This data communication scheme has the advantage, that only the (nearly) empty and full flags\(^3\) cross the clock domains and need synchronisation. As a consequence, the local clock on the sending side gets only

\(^3\)the full flag is set upon the last possible write event triggered by the write clock, but read by the reading side. The same is true for the empty flag, just with interchanged clock domains.
stretched when the FIFO is full, the clock on the receiving side when the buffer is empty.

![Diagram of a buffered GALS channel with poll-type output and input ports](image)

Figure 6.12: Schematic of a buffered GALS channel with poll-type output and input ports

6.1.3 Synchronisation with the Tester

All modules within Shir Khan, once configured, run on their own local clocks. For performance tests, it is important to synchronise the start of sequence of test operations with the tester hardware and detect the ending time. It is not necessary though to synchronise “all” processors with each other. If the starting point of a so called initiating processor can be acquired, the others can be synchronised by data transfers initiated by this processor.

![State diagram for start/stop port to synchronise operation to external tester hardware](image)

Figure 6.13: Start/stop port to synchronise operation to external tester hardware
6.1. TEST SYSTEM

A GALS port has been developed to handle this handshaking between GALS modules and Automatic Test Equipment (ATE). As the other GALS ports, it is specified in XBM (fig 6.13).

![Timing diagram for the test synchronisation](image)

Figure 6.14: Timing diagram for the test synchronisation

Fig. 6.14 illustrates a test synchronisation. The waveforms in bold represent the signals at the port processor, and the thin waveforms represent the signals seen at the ATE. The gray shaded area represents the propagation delay ($t_{pd}$) between the signals at the ATE and the PortProcessor. If the signal path for both the AckStart and Stop have similar delays the operation time can be measured accurately by the test equipment.

The ATE applies $StartATE$ signal to the chip. This signal is routed to all modules within the system ($StartProc$). For modules that need to be synchronised, a port enable command activates the StartStop port. Modules that have activated their StartStop port pause their local clocks until the $StartProc$ signal is received. Once the signal is received, this is acknowledged by setting the AckStartProc signal high and the local clock is released. When this signal arrives at the ATE (AckStartATE), the time measurement is started. As soon as the module has finished execution of its test program the StartStop port is re-enabled. On this second activation the port sends a $StopProc$ signal. The StartStop port returns to its initial state as soon as the $Start$ signal is deactivated and the ATE completes the time measurement on reception of $StopATE$.

In order to simplify the definition of the asynchronous behaviour, the AFSM specification does not cover all possible input signal se-
sequences. For example, the $StartxAI$ signal needs to be active until the $StopxAO$ signal is activated. Such timing constraints can be met easily by a proper test vector set.

6.2 Measurements of the Interconnects

The following figures summarise the figures obtained from throughput and communication latency measurements.

6.2.1 MOGLI

Some initiators are equipped with demand-type initiator ports, others with the poll-version thereof. The same is true for the target modules. This to cover all different combinations of sending initiator and receiving target port as possible: demand to demand, demand to poll, poll to demand, and poll to poll.

![Throughput Depending on Number of Initiators](image)

Figure 6.15: Throughput measurement for a single-channel MOGLI

A set of different communication patterns is applied. One initiator addresses only 1 target, then 2 targets, followed by 3 targets. The key
performance figures of the port controllers can so be measured. Addressing three targets results in a slightly higher throughput because the targets are accessed in interleaved fashion, giving them more time to cope with the incoming data. These measurements are compacted and diagrammed by the first bar in fig. 6.15.

Then, two initiators concurrently access the bus. Finally, three initiator compete for the shared bus resources. Since the data transfer channel must be shared, the available throughput is divided among the participating initiators. The PortProcessors are optimised to access the bus at a rapid rate. The saturation is therefore already reached with two initiators. More realistic system typically access the bus less frequently, and allow for a higher number of initiator before the bus goes into saturation. The curves exactly show the behaviour expected from the theoretic model. As have been indicated by fig 6.4 on page 126, Initiator2 has priority and gets half of the available throughput. The decline in throughput at 3 initiators only happens, because Initiator0 is equipped with a slower port controller. This slows down its data transfers and so also impacts the total throughput.

Figure 6.16: Communication latencies of the single-channel MOGLI
Communication latency is measured from enabling the initiator port until the corresponding response is safely received and the local clock is restarted. Figure 6.16 gives the figures for the different test setups. The latency consists of transfer latency and the time having been waiting for arbitration. A linear increase in the average latencies with the number of sending initiators can be seen, because the initiators have to wait longer to gain access depending on the number of waiting initiators.

![Throughput Depending on Number of Initiators](image)

**Figure 6.17: Throughput of the dual-channel MOGLI**

Fig. 6.17 show the throughput figures for the double-channel version and fig. 6.18 the communication latencies. Although, the dual-channel version has a separate response channel, the total throughput is not higher than with the single-channel version. Even if the response packet is controlled by a separate handshake, also the single-channel MOGLI provides data wires for the response data, just controlled together with the command. The dual-channel version has higher latency, since an arbitration is necessary for the response. In this unrealistic setup with the targets immediately responding, there is no gain in performance. The benefit of a dual-channel will become apparent when slow targets are involved.
6.2. MEASUREMENTS OF THE INTERCONNECTS

Figure 6.18: Communication latencies of the double-channel MOGLI

Figure 6.19: Comparison of the total throughput for single-channel and dual-channel architectures
To simulate a more realistic system behaviour, the targets need three clock cycles of their local clocks to prepare the response. Figure 6.19 compares the bus throughput of the two versions. Now, the advantages of the split transfer of the double-channel bus become apparent. The targets block the single-channel MOGLI, thus reducing the available throughput. The dual-channel bus is released as soon as the command has been received and is therefore not blocked.

![Average Data Transfer Latency Depending on Number of Initiators](image)

Figure 6.20: Comparison of the communication latencies of the single-channel and dual-channel version

### 6.2.2 STRING

In a ring topology, throughput and communication latency largely depend on the actual data transfer pattern. In the best case, all nodes only communicate with their next neighbour in direction of the ring transfer channel. Then all the ring segments are exclusively used by one node, the total throughput is the sum of all throughput in the ring segments (fig. 6.21). The communication latency remains low, the data packet is already consumed at the next ring transceiver (fig. 6.22).
6.2. MEASUREMENTS OF THE INTERCONNECTS

Throughput Depending on Number of Ring Nodes

![Graph showing throughput depending on number of ring nodes.]

Figure 6.21: Throughput of STRING

Average Data Transfer Latency Depending on Number of Ring Nodes

![Graph showing average data transfer latency depending on number of ring nodes.]

Figure 6.22: Communication latency of STRING
In a worst case situation, all nodes communicate with the neighbours farthest down the transfer channel. All data packets pass all ring transceivers. The throughput of the ring segments has to be shared by many packets, the total ring throughput reduces to the one of a single segment in the extreme. Worst case latency scaled linearly with the number of nodes present in the ring. All measured values nicely fits the expectations from the theoretic considerations.

If the receiver sends a response back, the whole ring has finally been traversed, when the response arrives at the sender. This leads to the highest communication latencies depicted in fig 6.22. Their values depend on the size of the ring, i.e. how many nodes are connected.

6.2.3 SWING

![Graph showing throughput depending on number of initiators]

Figure 6.23: Throughput of SWING

SWING is tested with similar test patterns, as have been used for the dual-channel MOGLI. This results in a fair comparison of the different interconnection solutions. The maximum throughput
6.2. MEASUREMENTS OF THE INTERCONNECTS

Average Data Transfer Latency Depending on Number of Initiators

Figure 6.24: Communication latency of SWING

(fig. 6.23) is achieved, when no data packets collide on their way from initiator to sender and back (Initiator0 sends to Target0, Initiator1 to Target1, and Initiator2 to Target2). In the worst case scenario, all initiators send to one single target. The receiving GALS port becomes the bottleneck. In this case, the average communication latencies also increase, because the initiators are slowed down in their effort to continuously send data packets.

Note in this context, that the latency figures displayed in fig. 6.24, only hold true for our specific test system. Although smaller switch sizes would be possible for just one or two initiators, the 4-input, 4-output switch matrix was measured. The communication latency largely depends on the number of passed crossbar switches an so on the depth of the matrix, which increases linearly with the number of attachable GALS modules.
6.2.4 Comparison

In order to compare the key performance figures of the different approaches, the throughput and latency are depicted in a single diagram in figure 6.25. It becomes obvious, that the shared bus can only satisfy moderate demands in throughput. The self-timed ring offers higher throughput but can only deliver optimal results, if the communication pattern fits the ring topology as it flows in a circular fashion. The switch outperform all other solutions, but at high area costs.

![Comparison of Interconnects: Throughput](image)

Figure 6.25: Comparison of the different interconnection solutions: throughput

Communication latencies differ less than the throughput. But higher throughput is always traded against higher latency. Although SWING is pipelined, its latency is only slightly higher that that of MOGLI. This is due to the fact, that the high capacitive loads of the shared bus lines nearly outweigh the additional latencies introduced in the pipeline stages of SWING.
Figure 6.26: Comparison of the different interconnection solutions: communication latency

Figure 6.27 points out, that high performance always comes at the cost of high area requirements. While a switch based solution offers enormous throughput at moderate latencies, the required area quadratically increase with the number of initiators. The step between 3 and 4 initiators is caused by the fact, that a $2 \times 2$ matrix of 2-input 2-output crossbar switches is used for 3 as well as for 4 initiators.
6.3 System Level Aspects

6.3.1 Area Overhead Estimation

Each locally-synchronous island is surrounded by a self-timed wrapper that is composed of the GALS ports, a clock generator, and a test extension element (TEE). This wrapper adds an area overhead to the system that calculates as follows:

\[ A_{\text{wrapper}} = A_{\text{ports}} + A_{\text{clkgen}} + A_{\text{TEE}} \]

Port controllers. The number of intermodule data transfer channels define the number of necessary port controllers (\(\#_{\text{ports}}\)). This largely depends on the system partitioning and the functionality, i.e. how intensive the system components are coupled. The bit-width (\(\#_{\text{bits}}\)) of the data channels affects the number of latches.
at all ports that receive data.

\[ A_{ports} = \sum_{i=1}^{\text{all ports}} \#port \cdot A_{port \ i} + \sum_{k=1}^{\text{receive ports}} \#bits \ k \cdot A_{\text{latch}} \]

**Clock generator.** The clock generator consists of a configuration block, a regular delayline, a delayline with finer steps to enhance the frequency resolution, and the MUTEX elements for arbitration between local clock and data transfers. Via the configuration circuitry the desired frequency can be programmed and startup configuration, test modes, or clock dividers can be set. While the desired range of clock frequencies determines the number of delay slices necessary in the delayline (\#slices), the area of the fine delayline is fixed. The safety margin is built from a few buffers in series in order to get a special low frequency for test purposes. It can be left away, and the clock divider can be used instead to scale down the frequency range. The arbitration block scales with the number of port controllers accessing the clock, an extra MUTEX handles flawless switching between external configuration clock and local clock.

\[ A_{clk} = A_{\text{config}} + A_{\text{delayline}} + A_{\text{finedelay}} + A_{\text{arbitration}} \]
\[ = A_{\text{config}} + \#slices \cdot A_{\text{slice}} + A_{\text{safetymargin}} + A_{\text{finedelay}} + (\#ports + 1) \cdot A_{\text{MUTEX}} \]

**TEE.** Test extension elements test the data channels by controlling the sender port and observe the receiver port [GVO+02]. They consist of a controller part to decode test commands from a central tester, means to access the port controllers (basically a Mux) and scan flip-flops in the data paths to control and observe the data lines.

\[ A_{TEE} = A_{\text{ctrl}} + \#ports \cdot A_{\text{access}} + \sum_{i=1}^{\#ports} \#bits \ i \cdot A_{\text{scan FF}} \]

Table 6.1 gives the area figures of all these components for the used 0.25\(\mu\)m CMOS process and the Shir Khan implementation. A
TABLE 6.1: AREA REQUIREMENT

<table>
<thead>
<tr>
<th>Part</th>
<th>Component</th>
<th>area [GE]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ports</td>
<td>$A_{port}$ Demand In</td>
<td>20.5</td>
</tr>
<tr>
<td></td>
<td>$A_{port}$ Demand Out</td>
<td>44.5</td>
</tr>
<tr>
<td></td>
<td>$A_{port}$ Poll In</td>
<td>17.5</td>
</tr>
<tr>
<td></td>
<td>$A_{port}$ Poll Out</td>
<td>21.5</td>
</tr>
<tr>
<td></td>
<td>$A_{port}$ Demand Initiator</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>$A_{port}$ Poll Initiator</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>$A_{port}$ Demand Target</td>
<td>25.5</td>
</tr>
<tr>
<td></td>
<td>$A_{port}$ Poll Target</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>$A_{port}$ StartStop Port</td>
<td>17.5</td>
</tr>
<tr>
<td></td>
<td>$A_{latch}$</td>
<td>5</td>
</tr>
<tr>
<td>Clock generator</td>
<td>$A_{config}$</td>
<td>377 ($\approx 0$)</td>
</tr>
<tr>
<td></td>
<td>$A_{slice}$</td>
<td>17 (3)</td>
</tr>
<tr>
<td></td>
<td>$A_{saftymargin}$</td>
<td>140 (0)</td>
</tr>
<tr>
<td></td>
<td>$A_{finedelay}$</td>
<td>392</td>
</tr>
<tr>
<td></td>
<td>$A_{MUTEX}$</td>
<td>4 (3)</td>
</tr>
<tr>
<td>TEE</td>
<td>$A_{ctrl}$</td>
<td>$\approx 62$</td>
</tr>
<tr>
<td></td>
<td>$A_{access}$</td>
<td>2.5</td>
</tr>
<tr>
<td></td>
<td>$A_{scan,FF}$</td>
<td>9</td>
</tr>
</tbody>
</table>

TEE was actually not implemented since all the functional tests could be performed through the PortProcessor itself.

In Shir Khan the lowest area overhead of a GALS module accounts for 10.2% (Ringnode number 3), the highest for 11.4% (a AHB initiator module). In the latter case, the synchronous FSMs to translate the AMBA protocol are included in the overhead figure.

Adaptations in the Synchronous Circuit. If the datapath controllers within the synchronous island must be adapted to provide the data transfer enable signals, the area requirement for the synchronous datapath controllers is often smaller or about equal to the original ones. This is because of the simple state flow especially if only demand-type ports are involved. The controller simply requests a data transfer on the appropriate port by asserting the port enable signal.
6.3. SYSTEM LEVEL ASPECTS

When the next active clock edge (and therefore the first possible state transition) occurs, the data has already been safely transferred. The controller FSM is freed from explicitly controlling data communication, it suffices to initiate the transfers.

6.3.2 System Partitioning

Algorithms for optimal system partitioning of GALS designs in terms of performance, area, and power dissipation are still a matter of research [HMK+99, GM02]. However, implementing our ideas on silicon helped to gather some experience and to derive some rules of the thumb:

Module Boundaries. Recommendation to place system parts in different GALS modules:

- The synchronous island gets too large for a proper clock distribution.

- Subsystems running at different clock frequencies.

- There is rare interaction between the different parts and the number of interconnections is not too high.

- When a memory is not accessed frequently, it is usually advantageous to put the memory in a separate module to decouple the (slow) memory timing from the datapath accessing it.

If none of the reasons mentioned above require a splitting, whole functional units are put into one GALS module (e.g. microprocessor core, DSP core, USB interface, etc). IP-modules typically become one GALS module.

Lower Limit of the Module’s Size. The area overhead has effects on the system partitioning. Depending on the maximum overhead considered acceptable, the minimum size of the locally-synchronous island is determined. According to Rent’s rule and its application

\footnote{Data transfers every clock cycle are possible though and do not limit performance in general.}
to heterogeneous systems [Don79] the number of interconnects grows exponentially with the number of modules. An overly fine granularity of the partitioning must therefore be avoided.

**Upper Limit of the Module’s Size.** The port enable signal to initiate a data transfer is set by a flip-flop triggered by the locally-synchronous clock. The timing on the path from flip-flop through the asynchronous FSM of the port controller to the clock stretching request signal ($R_i$) as indicated in figure 6.28 is critical. A request to stretch the clock has to arrive at the MUTEX \(^5\) before the feedback of the clock in order to prevent further active clock edges.

![Timing Diagram](image)

Figure 6.28: Critical timing path from synchronous island through port controller to the local clock generator limits the possible depth of the clock tree or the maximum achievable clock frequency.

The clock tree is part of the path from clock generation unit through synchronous island and port controller back to the clock generation unit. This sets an upper limit to the depth of the clock tree and therefore also to the size of the locally-synchronous island. There

\(^5\)where the unknown timing relation between signals from different clock domains are resolved.
is a tradeoff between large but slow and fast bus small synchronous islands.

In our recent implementations, both poll-type as well as demand-type ports are equipped with a transfer acknowledge signal (Ta) to indicate a successful transfer in the preceding clock cycle. Even when a transfer is postponed to the next clock cycle because the port enable signal arrives at the MUTEX later than the rising edge of the feedback clock rclk the system remains safe. By doing so, the upper limit is only a question of performance, not safety. The synchronous controller is slightly more complex because it has to react depending on the status of Ta.

### 6.3.3 System Conversion

The self-timed wrapper can connect to the synchronous island without adaptations to the island when the synchronous circuit's interfaces provide enable signals when new data is ready. This is normally the case for data sources. Data sinks however do usually not indicate the ability to receive by a dedicated signal that could enable the receiving port controller. Adaptions to the controller FSMs are therefore necessary. This can be problematic for IP-modules.

**Converting a Fully Synchronous Design into a GALS Architecture.** Generally, it is possible to retain the datapath structure unchanged, controllers must usually be adapted. Central controllers need to be distributed over separate synchronous islands as shown in figure 6.29. Since access to global parameters is no longer possible, local copies must be held.

An alternative is to place central controllers into a dedicated GALS module and control the other modules by means of data transfer channels (fig.6.30).

GALS system do not have to implement special power down modes controlled by a central controller. This feature comes for free, because pending GALS ports suspend the local clock thereby effectively preventing any dynamic power dissipation.
The experiences made with our implementations clearly showed that modifications to a synchronous design to convert it to a GALS system takes much less effort than starting from scratch.
Figure 6.30: GALS system with central controller placed in one of the GALS modules and controlling other modules using GALS transfer channels.
Seite Leer /
Blank leaf
Chapter 7

Conclusions

7.1 Summary

In the first stage of the GALS project, a technique was developed to fit the GALS methodology within a standard VLSI design flow. The main concern was to provide a modular approach to unburden the system designer from low-level details of asynchronous circuit design. This makes it possible to take advantage of both the industry-standard synchronous design methodology within synchronous domains, and easy interfacing over self-timed interconnects across clock boundaries.

To prove the feasibility of our GALS approach, a crypto system was implemented and compared to a synchronous version. While an energy reduction of roughly 30% could be achieved using the GALS methodology, the main advantage of GALS designs is the inherent ease of composing synchronous or asynchronous functional circuit blocks into larger globally-asynchronous systems. While there are reservations about whether GALS circuits can generally achieve higher performance or lower power operation compared to fully synchronous designs, there is little doubt that GALS has an advantage with respect to modularity. This is due to the fact that the GALS port controllers acting as interface between the different modules contain both their timing and data requirements explicitly in their interfaces. This ease of composing subsystems in a late stage of the design flow and the
ability to reuse components from previous designs is a tremendous advantage to shorten time to market. In today's large SoC designs, system level interconnects more and more entail the main design constraints in terms of system performance, power consumption, and cost. So far, all known GALS approaches relied exclusively on point-to-point data transfer channels. The lack of proven mechanisms for transferring data between multiple synchronous islands has been a major impediment for applying the GALS techniques to SoC design. The scientific contribution of this thesis was to develop versatile multi-point interconnection schemes which preserve self-timed operation between module boundaries.

The following three interconnection structures have been introduced:

**MOdular GaLs Interconnect (MOGLI).** MOGLI is a shared bus. Access to the communication medium is handled by a set of GALS ports supporting proper access of the shared media. It works with central arbitration and address decoding.

The single-channel MOGLI can be extended towards a double-channel version that supports a split-transfer scheme. The decoupling of command and response makes the bus available to other initiators while the target is still operating on the received data and was not yet able to send a response. This increases throughput and improves bus availability.

This approach is modular, easy to build, and offers a self-timed alternative to established synchronous on-chip bus solutions. MOGLI is suffers from the high capacitive load of long interconnection wires. As the shared bus lines can easily become the bottleneck of the system, this approach is suitable for systems with moderate communication demands. It is questionable how long a shared bus can still meet the communication requirements of future SoCs.

MOGLI reaches a total bus throughput of 71 MegaTransfers/s for a 0.25μm process. The total transfer latency is 14.7 ns for the single-channel bus and 30.7 ns for the double-channel version. The area increases about linearly with the number of attached modules.
Self-Timed RINg for Gals (STRING). Several GALS modules are connected with a circular path. Dedicated self-timed ring transceivers route the data packets to their destination and free the GALS modules from managing the data traffic. The transceivers provide simple deadlock prevention by informing the sender in case a defective receiver is unable to accept a data item. Throughput can be doubled by using a second ring for responses.

A ring topology is a modular, easily scalable solution. It does not make any distinction between initiator and target, every node can initiate a transfer. It is especially suitable for systems that distribute computation to different subsystems and pass data from one computational block to the next in a regular fashion.

The ring provides higher throughput than a shared bus but at the price of higher end-to-end latency. The throughput accounts for 107 MegaTransfers/s per ring segment with a transfer latency of roughly 4 ns per traversed ring transceiver. The maximum total ring throughput is the sum of the throughput of all segments. Only 107 MegaTransfers/s are reached in the worst case. The area increases linearly with the number of ring nodes.

SWItching Network for Gals (SWING). This is basically a matrix of small self-timed crossbar elements that form the interconnection fabric between GALS modules. Due to the pipelined data transfer, separate transfer channels and switches are necessary for commands and responses. It offers concurrent data transfer channels, so achieving a high total throughput. It is therefore easy scalable in throughput, but always at the cost of higher communication latency and a tremendous increase in area. This should offer enough leeway for tomorrow's complex SoCs.

A throughput of 147 MegaTransfers/s per attached module can be obtained. Due to the concurrent transfer channels, the total network throughput can reach the number of modules multiplied by 147 MegaTransfers/s when no data packets collide at all. In the worst case, all initiators address one single slave which becomes the bottleneck. The total throughput then decreases to 147 MegaTransfers/s. The switch implemented in Shir Khan has a communication latency of 30.7 ns.
All communication structures can be built from a small yet sufficient library of self-timed components. It can be applied to a large number of systems including those with complex on-chip communication requirements. To prove feasibility and assure simulation results with "real-world" measurements, all the interconnects have been implemented in the test chip "Shir Khan".

<table>
<thead>
<tr>
<th></th>
<th>Transfer rate (MTransfer/s)</th>
<th>Min. latency (ns)</th>
<th>Technology (μm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOGLI (shared bus)</td>
<td>71</td>
<td>15/31</td>
<td>0.25</td>
</tr>
<tr>
<td>STRING (ring)</td>
<td>107 per segment</td>
<td>8 per node</td>
<td>0.25</td>
</tr>
<tr>
<td>SWING (switch)</td>
<td>147 per initiator</td>
<td>8 per stage</td>
<td>0.25</td>
</tr>
<tr>
<td>MARBLE</td>
<td>83</td>
<td>14</td>
<td>0.35</td>
</tr>
<tr>
<td>PI-bus</td>
<td>50</td>
<td>40</td>
<td>0.25</td>
</tr>
<tr>
<td>AMBA AHB</td>
<td>up to 150</td>
<td>14</td>
<td>&lt;0.25</td>
</tr>
<tr>
<td>CoreConnect PLB</td>
<td>100</td>
<td>20</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Table 7.1: Comparison between the three GALS interconnects, the asynchronous bus MARBLE, and synchronous solutions

Table 7.1 compares the main performance figures of the GALS approaches, their asynchronous competitor MARBLE, and the commercially available synchronous solutions. MOGLI ranges in the same region as the shared bus MARBLE. As such, they both suffer from the bottleneck of the shared resources and from the long delay between request and acknowledge. While even the complexer ring and switch structures do not exceed synchronous buses and networks, they at least measure up to them. And the GALS interconnects fulfil their main target to keep the modularity of the GALS approach. The self-timed protocol offers a timing closure that is inherently guaranteed at their interfaces. This modularity becomes indispensable to build large systems on chip.
Besides the multi-point interconnection networks, other points have been addressed during our recent GALS project. A script based design flow build upon a standard synchronous design flow was developed [GOV+03] and a functional test methodology [GVO+02] was introduced by Gürkaynak. Improved clock generators with an increased tuning range and finer frequency selection steps [OVG+02] permit higher performance.

7.2 Outlook

While several critical obstacles have been removed in order to bring the GALS methodology closer to industrial design requirements, there is still room for improvement:

The current tool flow based on design automation scripts needs to be enhanced to industrial scale requirements.

Energy dissipation can easily be decreased by dynamically adjusting the supply voltage of GALS modules depending on the performance demands of the system. Dynamic power supply scaling nicely fits into the GALS scheme, because a module can easily keep its communication partner waiting while the voltage level is adjusted. Special transient phases known from synchronous systems are unnecessary.

Automatic system partitioning for GALS will become necessary observing performance, area, and/or power dissipation.

Tools for GALS system simulations and verification need to be established.
Appendix A

Acronyms

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D</td>
<td>tool set for synthesis of asynchronous finite state machines from an extended burst mode description</td>
</tr>
<tr>
<td>AFSM</td>
<td>asynchronous finite state machine</td>
</tr>
<tr>
<td>Ai</td>
<td>acknowledge of clock stretching handshake</td>
</tr>
<tr>
<td>Ap</td>
<td>acknowledge signal between to GALS ports</td>
</tr>
<tr>
<td>ASIC</td>
<td>application specific integrated circuit</td>
</tr>
<tr>
<td>ATE</td>
<td>automatic test equipment</td>
</tr>
<tr>
<td>BM</td>
<td>burst mode</td>
</tr>
<tr>
<td>ClkInit</td>
<td>clock initialisation signal (reset)</td>
</tr>
<tr>
<td>CMOS</td>
<td>complementary metal oxide semiconductor technology</td>
</tr>
<tr>
<td>D-type</td>
<td>demand-type (GALS port)</td>
</tr>
<tr>
<td>DI</td>
<td>delay-insensitive</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 term for the complexity of a circuit</td>
</tr>
<tr>
<td>HDL</td>
<td>hardware description language</td>
</tr>
<tr>
<td>IP</td>
<td>intellectual property</td>
</tr>
<tr>
<td>lclk</td>
<td>local clock</td>
</tr>
<tr>
<td>MARILYN</td>
<td>name of the GALS SAFER chip</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>MERLIN</strong></td>
<td>name of the globally-synchronous SAFER chip</td>
</tr>
<tr>
<td><strong>MOGLI</strong></td>
<td>modular GALS interconnect</td>
</tr>
<tr>
<td><strong>MOSFET</strong></td>
<td>metal-oxide-semiconductor field effect transistor</td>
</tr>
<tr>
<td>Muller-C</td>
<td>2-input majority gate; an asynchronous standard gate</td>
</tr>
<tr>
<td><strong>MUTEX</strong></td>
<td>mutual exclusion element; an asynchronous standard element widely used as arbiter element</td>
</tr>
<tr>
<td><strong>NRZ</strong></td>
<td>non return to zero, a signalling scheme</td>
</tr>
<tr>
<td><strong>P-type</strong></td>
<td>poll-type (GALS port)</td>
</tr>
<tr>
<td><strong>PCC</strong></td>
<td>pausable clocking control [YBA96]</td>
</tr>
<tr>
<td><strong>Pen</strong></td>
<td>port enable signal</td>
</tr>
<tr>
<td><strong>PTV</strong></td>
<td>process, temperature, voltage</td>
</tr>
<tr>
<td><strong>QDI</strong></td>
<td>quasi delay insensitive</td>
</tr>
<tr>
<td><strong>RAM</strong></td>
<td>random access memory</td>
</tr>
<tr>
<td><strong>rclk</strong></td>
<td>request clock signal; feedback of lclk</td>
</tr>
<tr>
<td><strong>Req</strong></td>
<td>request line of a (general) handshake pair</td>
</tr>
<tr>
<td><strong>Ri</strong></td>
<td>request for clock stretching (wrapper-internal)</td>
</tr>
<tr>
<td><strong>ROM</strong></td>
<td>read only memory</td>
</tr>
<tr>
<td><strong>Rp</strong></td>
<td>request signal between to GALS ports</td>
</tr>
<tr>
<td><strong>RZ</strong></td>
<td>return to zero, a signalling scheme</td>
</tr>
<tr>
<td><strong>SAFER</strong></td>
<td>secure and fast encryption routine [Mas94]</td>
</tr>
<tr>
<td><strong>SI</strong></td>
<td>speed-independent</td>
</tr>
<tr>
<td><strong>SK-128</strong></td>
<td>SAFER with strengthened 128 bit key algorithm</td>
</tr>
<tr>
<td><strong>SoC</strong></td>
<td>system-on-chip</td>
</tr>
<tr>
<td><strong>ST</strong></td>
<td>single track, both request and acknowledge is signalled on one single wire</td>
</tr>
<tr>
<td><strong>STG</strong></td>
<td>signal transition graph, a graph based method to specify circuits</td>
</tr>
<tr>
<td><strong>STRING</strong></td>
<td>self-timed ring interconnect for GALS</td>
</tr>
<tr>
<td><strong>SWING</strong></td>
<td>switching network for GALS</td>
</tr>
<tr>
<td><strong>Ta</strong></td>
<td>transfer completion signal</td>
</tr>
<tr>
<td><strong>TEE</strong></td>
<td>test extension element (test access to the gals ports and transfer channels)</td>
</tr>
<tr>
<td>Verilog</td>
<td>hardware description language</td>
</tr>
<tr>
<td>VHDL</td>
<td>hardware description language</td>
</tr>
<tr>
<td><strong>VLSI</strong></td>
<td>very large scale integration</td>
</tr>
<tr>
<td><strong>XBM</strong></td>
<td>extended burst mode</td>
</tr>
</tbody>
</table>
Appendix B

Port Processor

B.1 Introduction

To test the GALS interconnection networks, a flexible and reconfigurable test bed is desirable. A simple processor core called PortProcessor has been designed to address this need.

B.2 Architecture

Figure B.1 depicts the basic architecture of the PortProcessor.

B.2.1 Instruction Memory

The instruction memory is 10 bits wide. As the port processor uses 6 bits for addressing, it supports a memory depth up to 64 words. The memory size can be adapted to different requirements. Memory depths of 8, 16, 32 or 64 instructions are selectable by generics in the hardware description language. The program memory is realized as register array and designed in a way that allows it to be shifted in and out (10-bit wide) by an external synchronous clock. This is necessary for initialization (programming) and test.
### B.2.2 Communication Ports

The PortProcessor uses 4 input and 4 output ports for communication with the other port processors. Using 4-bit data values keeps the complexity of the processor itself at a minimum. This is crucial because a fast processor is needed to test the interconnection structures to their performance limit.

All four ports (A, B, C, D) have separate 4-bit input and 4-bit output interfaces with associated PortEnable (Pen) and TransferAcknowledgment (Ta) signals. To enable fast transfers, each port is equipped with a buffer memory of up to 16 4-bit values, and an address counter. While different ports can have memories of differing sizes, the input and output parts of one single port have memories of the same size.

![Figure B.1: Basic block diagram of the port processor](image-url)
B.2. ARCHITECTURE

The ALU can use the 4 output ports as destination registers and the four input ports as source registers. Two 4-bit address counters are used to determine the actual pointer location for the input and output memories. A "Counter Control Bit" controls whether or not these counters will get incremented following a port activation command or not.

The output port memories act as a read-only memory during normal operation and can not be modified using the port processor instructions\(^1\). Therefore they need to be initialized prior to operation using an external interface. The input port sequencer memory, on the other hand, acts as a write-only memory and stores the values received by the input port. The content of the input port sequencer memory can not be read directly by the port processor\(^2\) but only by an external interface.

B.2.3 Instruction & Port Memory Initialisation

To assure easy programmability and testability all memories can be accessed from Automatic Test Equipment (ATE) over separate input and output buses. In order to simplify operation, all memories are accessed serially. To improve access times, 10-bit buses are used. This simplifies vector generation as an entire 10-bit instruction can be shifted into the instruction memory at once.

Figure B.2 provides a detailed view of the memory access configuration. Note that two ports are coupled together for read and write accesses. As an example the output sequencer memory for PortA and PortB are written and read simultaneously using the 10-bit buses (two signals are left unconnected). The figure does not detail the block selection process. Each coupled block receives a separate serial clock (shown as $SCLKxCI$) and a separate write enable signal (not shown). These signals are provided by a configuration circuit located in every GALS module.

\(^1\)The output port memory location addressed by the counter is an exception. This location can be written by STA or STAA commands. Although it is not very practical to initialize the whole memory using these commands.

\(^2\)However, the LDA command can be used to read the value of the memory location pointed by the counter.
B.2.4 Execution Unit

Also the execution unit is optimised for the intended use as a test bed. A simple accumulator based ALU without overflow control is used. The output of the accumulator can be transferred to any combination of output registers simultaneously. Normally all GALS ports return a $T_a$ signal. However for normal instructions the state of the $T_a$ is not checked. There is a specialized circuitry that generates a signal if all (simultaneously) issued $P_e$ signals have received a $T_a$ signal.

All instructions are decoded and executed in a single cycle. Each clock cycle samples the current instruction to be executed. The port enable signals and the next address are computed from this sampled instruction. Sufficient time needs to be given to enable the program memory to decode and select the next instruction within a clock cycle.

B.3 Instructions

Specialized instructions allow efficient communication and optimal usage of the limited instruction memory. Table B.1 lists all the instruc-
The main instruction concerning port access is AP (Activate Port). This instruction uses an 8-bit mask and so activates any combination of input and output ports simultaneously.

The TP (Transfer Port) instruction is used to transfer the input data from a port to the output of the same port. Both read and write channels of the ports specified with the mask are activated, and the output port sends the value previously received.

The STA (STore Accumulator) command stores the value of the accumulator into the output ports identified by the port mask. The STAA (STore Accumulator and Activate) command performs the same operation and activates the ports that have new values assigned.

There are three flow control instructions. JMP (JuMP) makes an unconditional jump to the specified address. The BNZ (Branch if Not Zero) instruction is a conditional branch instruction based on the value of the accumulator. The RCMPJ (ReadCoMPareJump) instruction selects one input port, and the contents of this input port are compared to the accumulator. All the bits in the accumulator being at logic1 form a mask. If all bits of the input port that matches the mask, program execution will continue at the specified address.

There are two separate address counters within the port processor, one for the input pattern memories and one for the output pattern memories. The counters can be set to a specific value using the LDIC and LDOC instructions. Both counters are enabled using the SETC instruction. This instruction sets the Counter Control Bit to '1'. All AP instructions that contain an output port, increase the output counter and all AP instructions that contain an input port, increase the input port counter. The counter control bit can be reset using the RESETC instruction.

The WAIT instruction halts execution until all transfer acknowledge signals Ta signals related to the last AP, STSRC, STAA or TP command have arrived. The remaining instructions are common logic operations, with different input sources.
<table>
<thead>
<tr>
<th>Instr.</th>
<th>Code</th>
<th>Args.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AP</td>
<td>00 rrrr vvvv</td>
<td>port mask</td>
<td>Activate port</td>
</tr>
<tr>
<td>RCMPJ</td>
<td>01 ssaa aaaa</td>
<td>src1 addr</td>
<td>Read port, compare with accu, jump if bit match</td>
</tr>
<tr>
<td>JMP</td>
<td>10 00aa aaaa</td>
<td>address</td>
<td>Jump to address</td>
</tr>
<tr>
<td>BNZ</td>
<td>10 01aa aaaa</td>
<td>address</td>
<td>Branch if accu Not Zero</td>
</tr>
<tr>
<td>?</td>
<td>10 10-- ----</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>LDIC</td>
<td>10 1100 vvvv</td>
<td>value</td>
<td>Load Input Counter</td>
</tr>
<tr>
<td>LDOC</td>
<td>10 1101 vvvv</td>
<td>value</td>
<td>Load Output Counter</td>
</tr>
<tr>
<td>?</td>
<td>10 1111----</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>ADDi</td>
<td>11 0000 vvvv</td>
<td>value</td>
<td>Add immediate value to accu</td>
</tr>
<tr>
<td>XORi</td>
<td>11 0001 vvvv</td>
<td>value</td>
<td>XOR immediate value with accu</td>
</tr>
<tr>
<td>ANDi</td>
<td>11 0010 vvvv</td>
<td>value</td>
<td>AND immediate value with accu</td>
</tr>
<tr>
<td>ORi</td>
<td>11 0011 vvvv</td>
<td>value</td>
<td>OR immediate value with accu</td>
</tr>
<tr>
<td>ADD</td>
<td>11 0100 sstt</td>
<td>src1 src2</td>
<td>Add src1 &amp; src2, result to accu</td>
</tr>
<tr>
<td>XOR</td>
<td>11 0101 sstt</td>
<td>src1 src2</td>
<td>XOR src1 &amp; src2, result to accu</td>
</tr>
<tr>
<td>AND</td>
<td>11 0110 sstt</td>
<td>src1 src2</td>
<td>AND src1 &amp; src2, result to accu</td>
</tr>
<tr>
<td>OR</td>
<td>11 0111 sstt</td>
<td>src1 src2</td>
<td>OR src1 &amp; src2, result to accu</td>
</tr>
<tr>
<td>LDi</td>
<td>11 1000 vvvv</td>
<td>value</td>
<td>Load immediate value to accu</td>
</tr>
<tr>
<td>STA</td>
<td>11 1001 mmmm</td>
<td>port mask</td>
<td>Store accu to all ports in port mask</td>
</tr>
<tr>
<td>STAA</td>
<td>11 1010 mmmm</td>
<td>port mask</td>
<td>Store accu to ports and activate</td>
</tr>
<tr>
<td>TP</td>
<td>11 1011 mmmm</td>
<td>port mask</td>
<td>Transfer data from all input to all output ports in port mask</td>
</tr>
<tr>
<td>ADDa</td>
<td>11 1100 00ss</td>
<td>src1</td>
<td>Add src1 to accu</td>
</tr>
<tr>
<td>XORa</td>
<td>11 1100 01ss</td>
<td>src1</td>
<td>XOR src1 with accu</td>
</tr>
<tr>
<td>ANDa</td>
<td>11 1100 10ss</td>
<td>src1</td>
<td>AND src1 with accu</td>
</tr>
<tr>
<td>ORa</td>
<td>11 1100 11ss</td>
<td>src1</td>
<td>OR src1 with accu</td>
</tr>
<tr>
<td>NOTa</td>
<td>11 1101 00ss</td>
<td>src1</td>
<td>Invert src1 and store to accu</td>
</tr>
<tr>
<td>LDa</td>
<td>11 1101 01ss</td>
<td>src1</td>
<td>Load src1 to accu</td>
</tr>
<tr>
<td>NOT</td>
<td>11 1101 10--</td>
<td>none</td>
<td>Invert accu</td>
</tr>
<tr>
<td>?</td>
<td>11 1101 11--</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>CLEARC</td>
<td>11 1111--00</td>
<td>none</td>
<td>Clear counter control bit</td>
</tr>
<tr>
<td>SETC</td>
<td>11 1111--01</td>
<td>none</td>
<td>Set counter control bit</td>
</tr>
<tr>
<td>WAIT</td>
<td>11 1111--10</td>
<td>none</td>
<td>Wait until all TA have arrived</td>
</tr>
<tr>
<td>NOP</td>
<td>11 1111--11</td>
<td>none</td>
<td>No operation</td>
</tr>
</tbody>
</table>

Table B.1: Port Processor Instructions
Bibliography


167


Frank K. Gurkaynak, Stephan Oetiker, Thomas Villiger, Norbert Felber, Hubert Kaeslin, and Wolfgang


[VSI] Processor Local Bus, Architecture Specifications, IBM Corp.


Curriculum Vitae

Thomas Villiger was born in Zug, Switzerland, on July 31, 1971. After finishing the college at the Kantonschule Zug (Matura Type C with distinction), he enrolled in Electrical Engineering at the Swiss Federal Institute of Technology (ETH) with emphasis on integrated circuits and high speed computing. He received his Diploma in Electrical Engineering (Dipl. El.-Ing. ETH corresponding to a M.Sc.) in spring 1998. The same year, he joined the Integrated Systems Laboratory (IIS) of ETH as a scientific associate in the VLSI design and test group and later became a research and teaching assistant to work towards his PhD. His research interests embrace all aspects of multi-synchronous systems-on-chip and system level data exchange. In June 2004, he joined Philips Semiconductors as a senior development engineer. His responsibilities are advanced multi-chip packages and Systems in Package (SiP) for mobile communication systems.