



FAKULTÄT FÜR **INFORMATIK**

Fault-Tolerant Hardware Implementation of a Consensus Algorithm

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Thomas Polzer, BSc.

Matrikelnummer 0325077

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:

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

Mitwirkung: Univ.Ass. Dipl.-Ing. Thomas Handl

Wien, 24.09.2009

(Unterschrift Verfasser)

(Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Thomas Polzer, Albrechtsgasse 88-94/10/6, 2500 Baden bei Wien

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

Baden, 24.09.2009

Acknowledgements

I would like to thank my advisors Prof. Andreas Steininger and Thomas Handl for their continuous help and advice. Additionally I want to thank Prof. Steininger for his excellent lectures in the field of Digital Design.

My thanks also go to Prof. Ulrich Schmid for his great lecture in Distributed Algorithms as well as for some very inspiring comments on my work.

Special thanks go to Mathias Függer for his guidance with the proofs.

I want to thank all my fellow students, especially Philipp Jahn, Robert Thullner and Alexandra Schuster, for the great time and companionship.

And of course none of it would have been possible if it was not for the continuous support from my parents and my two brothers.

This master's thesis was funded by the bm:vit FIT-IT project DARTS (proj. no. 809456-SCK/SAI).

Fault-Tolerant Hardware Implementation of a Consensus Algorithm

Abstract

This thesis develops a new communication model for digital electronic systems. The proposed scheme is comparable to a GALS (globally asynchronous locally synchronous) system with the difference that the clock sources have a bounded, a-priori known precision. This loose synchrony is exploited to establish a communication that (i) is free of metastability by design and (ii) has a fully predictable temporal behavior. As a consequence the communication scheme presents a synchronous behavior, thus allowing to employ techniques that are restricted to synchronous systems, while avoiding the central clock being a single point of failure. To compensate for the imperfect synchronization of the local clocks (within the defined precision), a FIFO buffer memory is used on each communication link.

Using the theory of distributed systems the correctness of the approach is formally proved. For this purpose the communication activity is modeled as a distributed algorithm. More specifically it is shown that metastability-free and correct communication is possible, given that the buffer is larger than a certain, formally proved minimum. Furthermore an efficient hardware implementation is given and used to experimentally show that the theoretical derived FIFO buffer size requirement represents a tight lower bound. A performance comparison with a traditional GALS system shows that the performance of our solution is superior.

Based on the new communication model, a fault tolerant electronic system, able to tolerate Byzantine faults even in case of non replica deterministic modules, is developed. First the usability of a TMR system in such a setting is analyzed and, as found inadequate, replaced by a hardware implementation of the commonly known Byzantine EIG consensus algorithm.

As the EIG algorithm is lockstep synchronous, the lockstep synchronous model is simulated on top of our communication model. The EIG algorithm is adapted such that it can be efficiently implemented in hardware based on the timings established by the lockstep rounds. The equivalence of the adapted algorithm and the original EIG algorithm is shown. Additionally the hardware implementation for a system tolerating a single Byzantine fault is sketched. Performance and complexity of the implementation are analyzed.

Fault-Tolerant Hardware Implementation of a Consensus Algorithm

Kurzfassung

Diese Master-Arbeit entwickelt ein neues Kommunikationsmodell für digitale elektronische Systeme. Das vorgeschlagene Schema ist vergleichbar mit einem GALS (globally asynchronous locally synchronous) System mit dem Unterschied, dass bei unserer Lösung die Taktquellen eine a-prior bekannte, beschränkte Präzision aufweisen. Diese schwache Synchronität wird ausgenutzt um ein Kommunikationssystem welches (i) konzeptuell frei von Metastabilität ist und (ii) ein vollkommen vorhersagbares zeitliches Verhalten hat zu entwickeln. Deshalb stellt das Kommunikationsschema ein synchrones Verhalten zur Verfügung, welches die Anwendung von Techniken gestattet, die auf synchrone Systeme beschränkt sind. Zusätzlich vermeidet dieses Schema den zentralen Clock als Single Point of Failure. Um die nicht perfekte Synchronisation zwischen den lokalen Clock Signalen (innerhalb der Präzision) zu kompensieren wird auf jeder Kommunikationsverbindung ein FIFO Buffer verwendet.

Mittels der Theorie der Verteilten Systeme wird die Korrektheit des Ansatzes formal bewiesen. Dazu werden die Kommunikationsvorgänge als verteilter Algorithmus modelliert. Genauer gesagt wird gezeigt, dass, unter der Voraussetzung dass die Buffergröße über einem gewissen, formal bewiesenen Mindestwert liegt, metastabilitäts- und fehlerfreie Kommunikation möglich ist. Weiters wird eine effiziente Hardware Implementierung vorgestellt und diese zur experimentellen Validierung der theoretischen FIFO Buffer Größe verwendet. Es zeigt sich, dass die bewiesene minimal benötigte Speichergröße eine größte untere Schranke darstellt. Ein Vergleich der Leistungsfähigkeit mit einem traditionellen GALS System zeigt, dass unsere Lösung einen höheren Datendurchsatz hat.

Basierend auf dem neuen Kommunikationsmodell wird ein fehlertolerantes elektronisches System entwickelt, welches auch dann in der Lage ist byzantinische Fehler zu tolerieren, wenn die Module nicht replikations-deterministisch sind. Dazu wird zuerst die Verwendbarkeit von TMR Systemen untersucht. Da diese jedoch als nicht einsetzbar eingestuft werden, muss stattdessen auf eine Hardware-Implementierung des bekannten byzantinischen EIG Consensus Algorithmus zurückgegriffen werden.

Da der EIG Algorithmus ein lockstep synchroner Algorithmus ist, wird basierend auf dem Kommunikationsmodell ein lockstep synchroneres Rundenmodell implementiert. Weiters wird der EIG Algorithmus so angepasst, dass er effizient in Hardware implementierbar ist. Die Äquivalenz des adaptierten Algorithmus mit dem

Original wird gezeigt. Weiters wird die Hardware-Implementierung eines Systems, welches einen byzantinischen Fehler tolerieren kann, beschrieben. Die Leistung und Komplexität der Implementierung werden ebenfalls analysiert.

Contents

1	Introduction	1
1.1	Problem Definition	1
1.2	Structure of the Thesis	2
I	Theoretical Background	3
2	Distributed Algorithm Basics	5
2.1	Message Passing System	5
2.1.1	Message Passing Network	6
2.1.2	Computational Node	6
2.1.3	Actions and Events	8
2.1.4	Messages	8
2.1.5	Visualization of Executions	9
2.2	Global Notion of Time	10
2.2.1	Properties of a Global Time Base	10
2.2.2	Clock Synchronization	12
2.3	Computational Models	13
2.3.1	Asynchronous Model	13
2.3.2	The Lockstep Round Model	13
2.4	Failure Handling in Distributed Systems	14
2.4.1	Failure Models	14
2.4.2	Agreement Problem	16

3	Digital Electronic Circuit Basics	19
3.1	Clocking Schemata	19
3.1.1	Synchronous Circuits	19
3.1.2	Asynchronous Circuits	20
3.1.3	Globally Asynchronous Locally Synchronous (GALS) Circuits	21
3.1.4	Multisynchronous Circuits	22
3.2	Intermodule Communication in Electronic Circuits	22
3.2.1	Synchronous Communication	23
3.2.2	Sourcesynchronous Communication	24
3.2.3	Asynchronous Communication	25
3.3	Metastability	25
3.3.1	Flip-Flop Timings	26
3.3.2	Avoiding Metastability	27
4	Communication Standards	29
4.1	Bit Representation	29
4.1.1	Single-Ended Signaling	29
4.1.2	Differential Signaling	30
4.2	Line Coding	31
4.2.1	Additional Clock Line	31
4.2.2	Data-Strobe Encoding	32
4.2.3	8B/10B Encoding	32
4.2.4	Asynchronous Parallel Communication	33
4.2.5	Asynchronous Serial Communication	33
4.3	Synchronous Communication Protocols	34
4.3.1	Basic Synchronous Transmission	34
4.3.2	SPI	34
4.3.3	I ² C	35
4.3.4	PCI	35
4.4	Sourcesynchronous Communication Protocols	35
4.4.1	Space-Wire	35

4.4.2	PCI-Express	36
4.4.3	Infiniband	36
4.5	Asynchronous Communication Protocols	36
4.5.1	Serial Port	36
4.5.2	Parallel Port	37
4.6	Transmission Protocol Comparison	37
5	Hardware Fault Models and Fault Tolerance	39
5.1	Hardware Fault Models	40
5.1.1	Stuck-At Faults	40
5.1.2	Stuck-Open Faults	40
5.1.3	Bridging Faults	41
5.2	Fault Tolerance	41
5.2.1	Triple Modular Redundancy	41
II	Framework Implementation	43
6	Problem Definition	45
7	Circuit Modeling	47
7.1	Creating the Model	47
7.2	Model Properties	49
7.3	Modeling Freeness of Metastability	49
8	Metastability-Free Intermodule Communication	51
8.1	Quasi-Synchronous Communication Scheme	51
8.2	Pipelined Communication Scheme	53
8.2.1	Algorithmic Model	54
8.2.2	Problem Definition	57
8.2.3	Relation Between Actions	57
8.2.4	Read–Write Order Proof	58
8.2.5	Bounded Buffer Size	60
8.2.6	Latency	62
8.2.7	Results	63

9	Pipelined Scheme Implementation	65
9.1	Circuit Design	65
9.1.1	Communication	65
9.1.2	Transmitter	66
9.1.3	Receiver	67
9.1.4	Communication Buffer	67
9.2	Implementation Mapping	68
9.2.1	Implementation without Input Register	68
9.2.2	Implementation with Input Register	69
9.3	Performance and Efficiency	70
9.3.1	Throughput	70
9.3.2	System Latency	71
9.3.3	Performance Comparison	71
9.4	Communication Example	72
9.5	Experiments	73
9.5.1	Test System	73
9.5.2	Clock Emulation	73
9.5.3	Test Conditions	75
9.5.4	Performed Tests	75
9.5.5	Results	76
10	Implementing a Lockstep Model	79
10.1	Algorithmic Model	79
10.2	Correctness Proof	81
10.3	Time Complexity	83
10.4	Mapping the Message Layer Implementation	83
10.5	Static Round Pattern Algorithm	84
10.6	Message Receive Event	85
10.7	Hardware Implementation	86

III	Consensus	89
11	Problem Definition	91
11.1	Problems of TMR Systems	91
11.2	May Hardware Act Non-Benign?	95
11.3	TMR Systems in Byzantine Environments	96
11.4	Alternatives to TMR Systems	97
12	Creating a Byzantine Fault Tolerant System	99
12.1	The EIG Algorithm	99
12.2	Algorithmic Model	101
12.3	Building the Resolve Tree	103
12.4	Resolve Function	106
12.4.1	Resolve Function for a Single Byzantine Fault	106
12.5	Complexity of the Adapted Algorithm	107
12.6	Circuit Design	109
12.7	Execution Example	111
IV	Conclusion and Future Work	115
13	Future Work	117
13.1	Flow Control and Error Detection	117
13.2	Resynchronization at Runtime	117
13.3	Ring Buffer Integration into the Clocking Chip	118
13.4	Resolve Functions for Stronger Fault Hypotheses	119
14	Conclusion	121

List of Figures

2.1	Example Message Passing System	6
2.2	Example Node with Three Neighbors	7
2.3	Message Delivery	8
2.4	Space-Time Diagram Example	10
2.5	Example of Drifting Clocks	11
2.6	Precision	12
2.7	Lockstep Synchronous System	14
2.8	Failure Model Example	15
2.9	EIG Tree for a $f = 1$ System	18
3.1	Synchronous Model	20
3.2	Asynchronous Model	21
3.3	GALS Model	21
3.4	Multisynchronous Model	22
3.5	Synchronous Communication	24
3.6	Source Synchronous Communication	24
3.7	Asynchronous Communication	25
3.8	D-Flip-Flop Timing Constraints	26
3.9	Metastable D-Flip-Flop	27
4.1	Unipolar Single-Ended Signaling (idealized)	30
4.2	Differential Signaling (idealized)	30
4.3	LVDS Transmitter and Receiver	31
4.4	Linecoding using an Additional Clock Line	31
4.5	Data-Strobe Encoding	32

4.6	Asynchronous Serial Communication	33
4.7	SPI Link	34
4.8	I ² C Bus	35
4.9	Full Duplex Space Wire Link	36
5.1	Stuck At Zero Fault	40
5.2	Stuck-Open Fault	41
5.3	OR Bridging Fault	41
5.4	Basic TMR System	42
5.5	TMR System using Replicated Voters	42
7.1	Modeling Example	48
8.1	Macrotick Generation	52
8.2	Macrotick Based Clock Generation	52
8.3	System Model used for the Proof	55
8.4	Execution of Tick k	57
9.1	Layout of a Node	66
9.2	Ring Buffer of Depth 4	68
9.3	Schematic Transmitter Circuit (No Input Register)	69
9.4	Schematic Transmitter Circuit (With Input Register)	70
9.5	Example Communication within the Test System	73
9.6	Layout of the Test System	74
9.7	Worst Case Precision Emulation	74
9.8	Results of the Experiments	77
10.1	Lockstep Round Generation Implementation Hierarchy	80
10.2	Visualization of the Assumptions	82
10.3	Simulation of the Lockstep Algorithm Implementation	87
11.1	Unsynchronized Voter Execution	92
11.2	Multisynchronous Voter Implementation	93
11.3	System with Replicated Multisynchronous Voters	94
11.4	Circuit used in the Proof	95

11.5	Distributed System Model of the Circuit	95
12.1	System Model Tolerating one Byzantine Fault	100
12.2	Fully Created Resolve Tree for $f = 1$	101
12.3	Structure of the Hardware Implementation	102
12.4	Building of a Resolve Tree Level	105
12.5	Resolve Function for $f = 1$	106
12.6	Runtime of the Consensus Algorithm [Microticks]	109
12.7	Node Implementation	110
12.8	Execution Example of a Consensus System	112
12.9	Execution Example of a Consensus System (Detail)	113
13.1	Clocking Chip – Ring-Buffer Integration	119

List of Tables

4.1	Transmission Protocol Comparison	38
9.1	Latency	71
9.2	Performance Comparison	72
9.3	Clock Primitives	75
9.4	Results of the Experiments	76
11.1	Voter Execution	92
12.1	Runtime of the Consensus Implementation	108

Chapter 1

Introduction

1.1 Problem Definition

Currently digital electronic systems are mainly implemented based on the synchronous paradigm. Due to ever increasing clock rates, smaller feature sizes and increasing gate counts the assumptions made by this model are harder and harder to meet. As only a single maximum clock rate is calculated for the whole system, long signal connections, even if only a few are present, decrease the system performance dramatically.

To circumvent these problems globally asynchronous locally synchronous (GALS) systems [Cha84] are used nowadays. Here the system is divided into several modules. Each module is driven by a single, independent clock source. The modules are developed independently and the intermodule signals are therefore not part of any timing analysis as they cross from one clock domain to another. Therefore the setup-hold window is not guaranteed to be maintained and the intermodule links may be subjected to metastability and therefore compromise the stability of the system.

The goal of this thesis is to develop a new system model which is (i) free of any potential for metastability and (ii) providing the possibility to implement independent modules. An efficient implementation for this problem is described, its correctness proved and a tight lower bound on the required buffer size is given.

This model is used as basis to implement a fault tolerant system with the ability to tolerate Byzantine faults, even if the implementation of the modules is not replica deterministic. The usability of a TMR system in such a setting is analyzed and, as found inadequate, replaced by a system using a hardware implementation of the widely known EIG consensus algorithm. The equivalence of the hardware implementation and the original EIG algorithm is shown.

1.2 Structure of the Thesis

The thesis is structured into four parts. The first part recapitulates the theory needed to understand the remainder of the thesis. It comprises Chapters 2-5. Chapter 2 gives an overview on the distributed systems theory including message passing systems, clock synchronization and system models. It is followed by a brief introduction into failure modeling and -handling in distributed systems. Chapter 3 introduces different clocking models as well as intermodule communication techniques and describes the problem of metastability. Chapter 4 outlines state of the art communication schemes based on the standards established in Chapter 3. Hardware fault models and fault tolerance mechanism (TMR systems) are discussed in Chapter 5.

Part II is devoted to the implementation of the basic framework including the metastability free communication layer and the simulation of the lockstep synchronous model based on the local microtick clock. Chapter 6 motivates the necessity of a new system model. Our approach for mapping circuits as distributed systems is shown in Chapter 7. Two different approaches for metastability free communication are presented in Chapter 8. Additionally the model with the higher performance is formally proved correct. Its hardware implementation and experimental results are presented in Chapter 9. Part 2 is concluded by defining a method to create lockstep synchronous rounds based on the local clock only (Chapter 10).

The implementation of a Byzantine fault tolerant system forms the focus of Part III. Chapter 11 motivates why such a system is important. It is shown that a naive TMR implementation would surely fail and that a more sophisticated implementation will also fail, if the system is not replica deterministic. The design of a hardware implementable adaptation of the exponential information gathering (EIG) algorithm [AW04] is described in Chapter 12. The equivalence of both algorithms is also shown and its hardware implementation is sketched.

Part IV concludes the thesis. Chapter 13 discusses open questions and future extensions, while Chapter 14 concludes the thesis by summarizing its most important findings.

Part I

Theoretical Background

Chapter 2

Distributed Algorithm Basics

To be able to prove the correctness of our solutions, we will model them using the theory of distributed systems. Therefore this chapter gives a brief overview on the distributed systems theory. In this chapter, except for Section 2.4, we assume that the system is reliable.

For more in depth information on the subject the very good book "Distributed Computing – Fundamentals, Simulations and Advanced Topics" [AW04] is recommended to the interested reader. Where not referenced otherwise, the information presented within this chapter is based on this book.

2.1 Message Passing System

As already mentioned before, the basis for the formal part of our work is the distributed systems theory, more specifically Message Passing Systems. Such systems consist of the following elements:

- Multiple computational nodes (shortly called nodes) ¹
- Communication links between the nodes (message passing network)

Directed graphs (see [Die05] for details on graph theory) are used to visualize message passing systems. Each computational node is represented by a node within the graph. The links within the message passing network are represented by directed edges. An example for a message passing system consisting of four nodes and using a fully connected message passing network is displayed in Figure 2.1.

¹In difference to [AW04] the computational elements are called nodes and not processors. This differentiation is made because their functionality is not necessarily implemented by a processor.

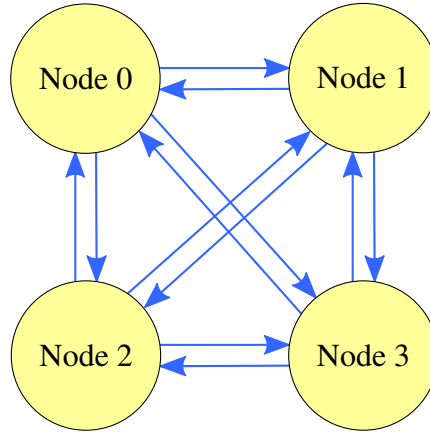


Figure 2.1: Example Message Passing System

2.1.1 Message Passing Network

The data transmission between any two nodes is modeled as the exchange of messages between these nodes. Each node can send a message to an adjacent node using the message passing network. Theoretically the topology of such a network can be arbitrary. Nevertheless for simplicity reasons it is assumed that the network is fully connected, which means there is a communication channel between any two nodes (see Figure 2.1 as an example).

2.1.2 Computational Node

A computational node is the basic element of a message passing system. A single node is referred to as n_i , where i is an artificial index identifying the nodes in the system. We introduce the set $P = \bigcup_i \{n_i\}$, called the set of nodes, containing all computational nodes of the system.

The computational nodes perform all calculations within the system. Figure 2.2 shows an example of a node with three neighbors. The node is modeled using the following three components:

- A state transition table modeling the internal logic of the considered node.
- An input buffer component (shortly called inbuf_j) for each adjacent node j .
- An output buffer component (shortly called outbuf_j) for each adjacent node j .

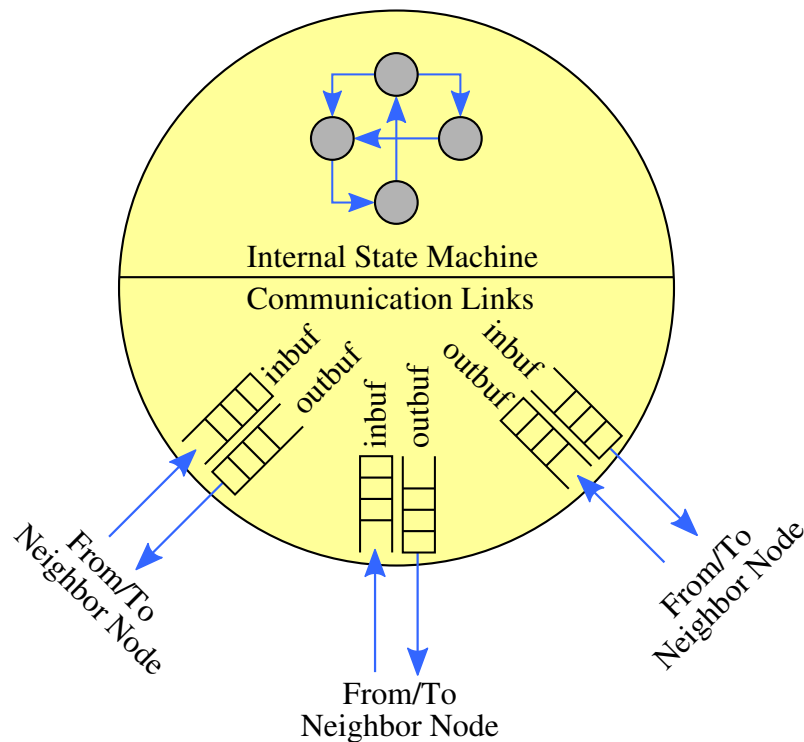


Figure 2.2: Example Node with Three Neighbors

The calculations performed by the nodes are modeled as state transitions. For every such state transition the node's state transition table contains an entry. The input to such a state transition is the current state of the node and all messages currently present in the input buffers. Based on this information the successor state is selected from the state transition table. Furthermore a subset (or all) messages can be removed from the input buffer, as well as new messages added to the output buffers of the node (sending messages to adjacent nodes).

The input buffers contain all delivered but not yet processed messages received from adjacent nodes. Messages are added to the input buffer when they are delivered to the node. A message is removed from the input buffer when having contributed to a state transition.

The output buffers contain all messages sent by the node to an adjacent node. Messages are added to the output buffer, when they are generated by a state transition. All messages present in the output buffer are not yet delivered. Nevertheless the node itself can not read from the output buffers. Therefore messages within the output buffers can never affect the selection of a state transition from the state transition table.

The messages are delivered using the message passing system. This is done by selecting a message from an output buffer of the sender node and placing it into the

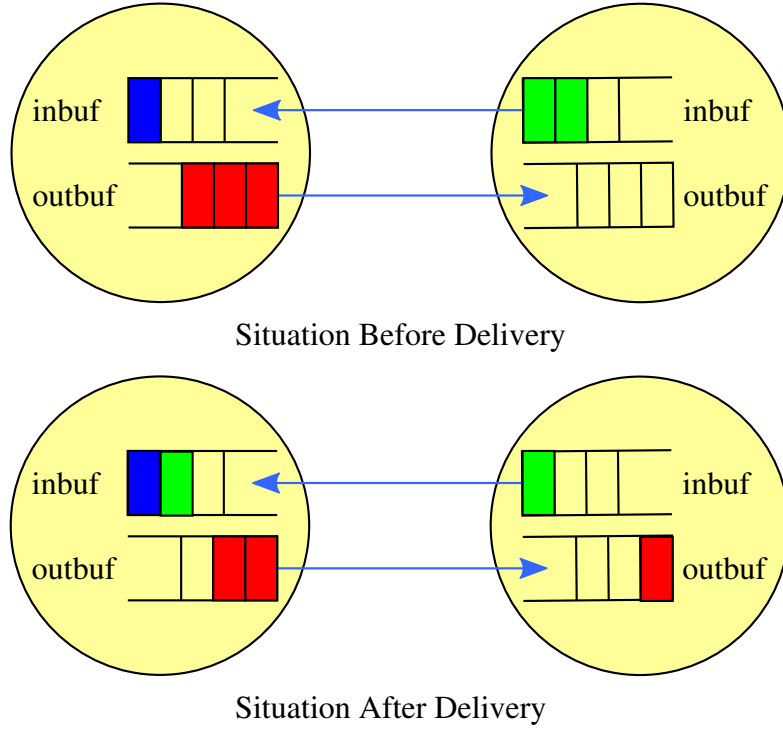


Figure 2.3: Message Delivery

input buffer of the receiver node. Figure 2.3 shows an example for the delivery of two messages (one in each direction) between two adjacent nodes.

2.1.3 Actions and Events

All activities occurring in the system (like state transitions, message delivery, \dots) are modeled as actions. A_i^k specifies the k^{th} action of processor i . The superscript and subscript are omitted, if a general action (A) not bound to any processor is described. Each action (A) has an associated start- and end time (shortly $t_s(A)$ and $t_e(A)$) and therefore a duration ($d(A) = t_e(A) - t_s(A) \geq 0$). If the duration of an action is zero ($d(A) = 0$), we call it an event (E). Since its duration is zero, the start- and end time are the same and we only speak of its time of occurrence ($t(A)$).

2.1.4 Messages

We shortly call a message with contents M $\langle M \rangle$. Each message has a specific length called $l(M)$ given in bits. Furthermore a message $\langle M \rangle$ has the following temporal properties.

End-to-End Delay

We have already discussed the concepts of sending, delivering and reading (consuming) messages. We will now formalize the message delivery process. Using the terminology of actions and events, we can model the sending, delivery and reading of messages as events. Therefore we can associate a send-, delivery- and read-time with each message $\langle M \rangle$ calling them shortly $t_{\text{send}}(M)$, $t_{\text{delivery}}(M)$ and $t_{\text{read}}(M)$, respectively.

The whole process of transmitting a message can be seen as an action (M_T) starting with the send event and ending with the read event. Therefore its duration is $d(M_T) = t_e(M_T) - t_s(M_T) = t_{\text{read}}(M) - t_{\text{send}}(M)$ and it is called the message End-to-End delay Δ . If available, the upper and lower bound of the end-to-end delay in the system are called Δ^+ and Δ^- , respectively.

Read-Write-Order Problem

It is important to note, that a message must be delivered before it can be read ($t_{\text{delivery}} < t_{\text{read}}$). Each implementation of a computational model must ensure that this order is met, otherwise the system behavior could become undefined.

FIFO Order

Without further assumptions messages sent on the same communication link may be delivered in arbitrary order. Nevertheless, it is often convenient to assume FIFO (first in first out) order on the communication links. FIFO order states that messages sent on the same communication link are delivered in the order they were sent. Formally the FIFO property can be written as: \forall two messages M_1, M_2 on the same communication link:

$$t_{\text{send}}(M_1) < t_{\text{send}}(M_2) \Leftrightarrow t_{\text{delivery}}(M_1) < t_{\text{delivery}}(M_2)$$

2.1.5 Visualization of Executions

Execution are visualized using the so called space-time diagram. Each node is represented by a horizontal line within the diagram and the time advances from left to right. For each node the events and actions are shown in the diagram using boxes or short vertical lines. Furthermore all messages are visualized by arrows between the action or events of the nodes. An (annotated) example execution can be found in Figure 2.4.

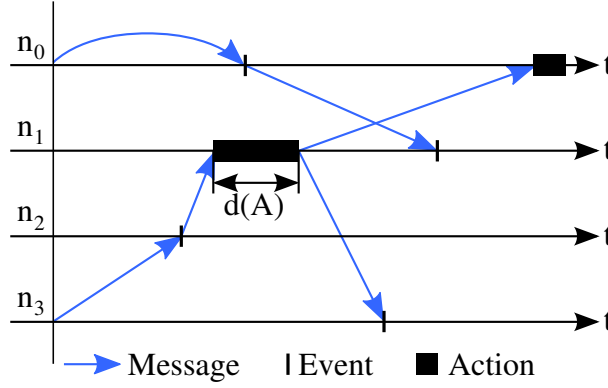


Figure 2.4: Space-Time Diagram Example

2.2 Global Notion of Time

So far we have discussed message passing systems with all their subcomponents. An important property of such a system is a global notion of time. A consistent view of time is a powerful property when modeling algorithms for distributed systems.

Time in distributed systems is measured in clock ticks. Each clock tick can be seen as an event C in the system having an assigned real time $t(C)$. The k^{th} clock tick on node i is referenced as C_i^k .

2.2.1 Properties of a Global Time Base

We will now discuss three major properties of a time base, namely the clock drift, the accuracy and the precision (π).

Clock Drift

As described in [Kop97], it is possible to deduce from the clock drift, whether a clock is running with its nominal frequency or it runs too fast or slow. As a reference, a fictional optimum clock with the nominal frequency is used. Figure 2.5 illustrates both cases.

The rate a clock runs slower or faster than this nominal clock is called the drift rate and defines how many real clock ticks occur within one clock tick of the fictional nominal clock. The example clocks have a drift rate of 1.14 and 0.89.

Each node in the system can have a clock with a different drift rate. Furthermore the drift rates of the clocks can vary over time.

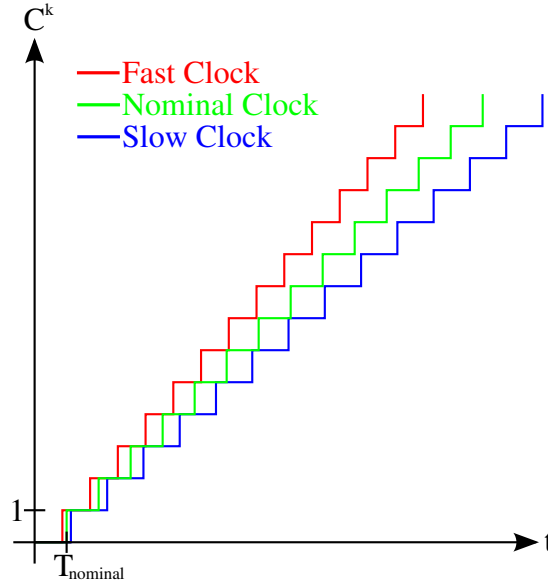


Figure 2.5: Example of Drifting Clocks

Accuracy

We use a more restrictive definition of accuracy than the one defined by [Sch87]. We specify the lower- and upper bound of the period length of all clocks in the system over time and therefore bound the rate the clocks may change with. Formally it can be written as:

$$\forall i \in P, k > 0 : \exists T^- = \min_{i,k} (C_i^{k+1} - C_i^k) > 0 \quad (2.1)$$

$$\forall i \in P, k > 0 : \exists T^+ = \max_{i,k} (C_i^{k+1} - C_i^k) \geq T^- \quad (2.2)$$

As apparent from Equations (2.1) and (2.2), unlike the drift rate, the accuracy is a property of the whole system.

Precision

Before discussing the last parameter of our global clock system, we need to define the precedence relation (\rightarrow). Introduced by Lamport in [Lam03], $A \rightarrow B$ informally means that action A must have been finished before action B starts.

We now can use the precedence relation to describe the last important parameter, namely the precision (π). It is defined as the maximum value any two clocks of the system can differ at any point in time [Kop97]. Formally the precision can be stated as:

$$\exists \pi : \forall i, j \in P, \forall k \geq 0 : C_i^k \rightarrow C_j^{k+\pi} \quad (2.3)$$

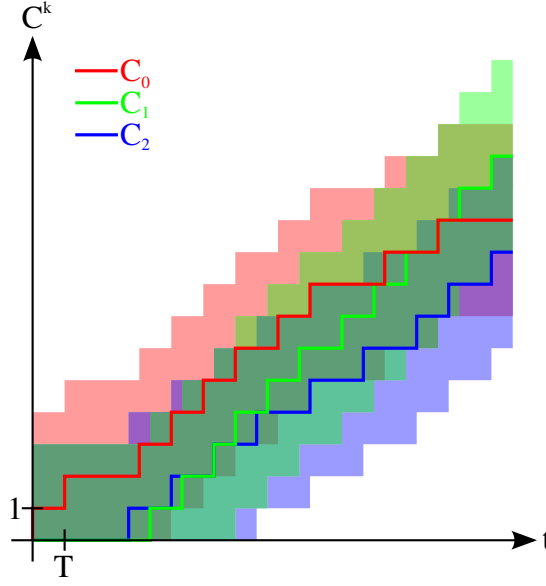


Figure 2.6: Example of a System with Three Clocks having a Precision of Three

To visualize this property Figure 2.6 shows an example clock system with three clocks. For each of the clocks the corresponding precision envelop is shown as a shaded area around the clock. For the system to hold its precision, all clocks must stay in the overlapping area of all three precision envelopes.

2.2.2 Clock Synchronization

Since all clocks of the system can have different, even non constant, drift rates, the precision π of any two clocks in the system changes over time and may, in particular, become arbitrary large. To keep the precision within predefined bounds, a clock synchronization algorithm must be applied, otherwise no global notion of time can be achieved.

There are many commonly known clock synchronization algorithms already available (see e.g. [AW04, ST03, LMS85] and specifically for VLSI implementations [WS05]).

We will assume the presence of a global clock system with a given precision and accuracy already available in our system. Therefore we will not discuss the concepts of clock synchronization algorithms in detail here.

2.3 Computational Models

There are two commonly known computational models that differ with respect to the assumption of a global notion of time.

- The asynchronous model
- The lockstep synchronous model

These models can be considered extreme variants, and there are plenty of other models in between. However, since most algorithms are targeted to one of these two models, we will concentrate on these two.

2.3.1 Asynchronous Model

As the name already suggests, no global notion of time is available in this model. Furthermore there are no local clocks available in the system. The algorithms describable by this model are completely time free and only driven by message delivery events (the message delivery event will trigger the message read event). Therefore there are no write-read order problems within this system.

Since no timing assumptions are made, no timing bounds can be violated. As convenient as this is, the absence of such bounds restricts the implementable algorithms dramatically (see e.g. [AW04] for some impossibility proofs).

2.3.2 The Lockstep Round Model

This model has a very accurate global notion of time ($\pi = 0$) and therefore is a very powerful tool to implement distributed algorithms. The algorithms themselves are easy to describe. The execution of the algorithm is split into successive rounds. Each round starts with a message send event. After the delivery of all messages, each node executes one computational action of specified length ($d(A)$) before the next round starts. The length of all rounds $0 < T \leq C < \infty$ is known in advance and is normally constant over time. To meet this round duration, all messages must be delivered timely, which means that the message end-to-end delay lies within:

$$\forall i \geq 0, \forall k \in P : 0 \leq \Delta \leq \Delta^+ < T - \max_{i,k} (d(A_i^k))$$

An example of an execution within this model can be found in Figure 2.7.

Within this model the write-read problem is solved by the a-priory knowledge of the message end-to-end delay bound and action durations. It will work as long as these

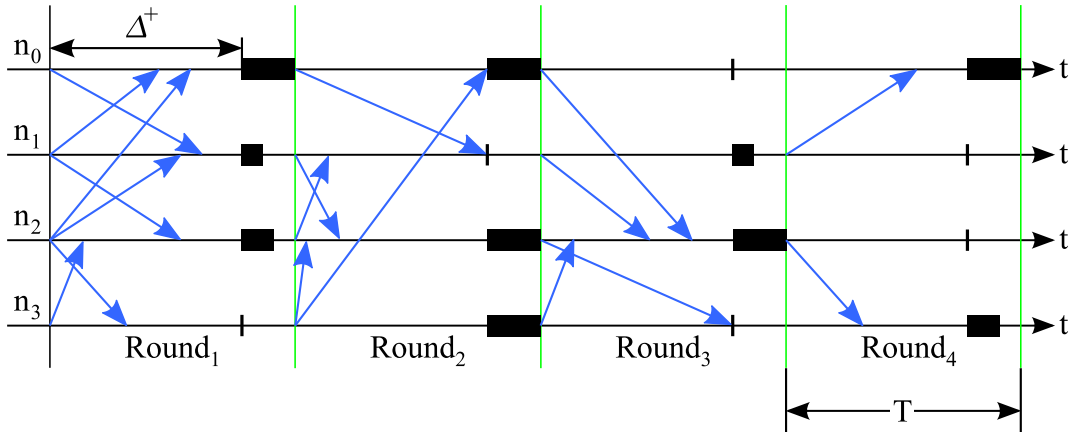


Figure 2.7: Example Execution within a Lockstep Synchronous System

bounds are met. If only once a bound is violated, the behavior of the system may become undefined. With respect to the implementation, the assumption of perfect precision is extremely strong, and therefore in practice additional considerations become necessary when implementing such an algorithm.

2.4 Failure Handling in Distributed Systems

So far we have only considered failure free systems. Unfortunately real system will not always work correctly. Parts of the system may fail over time. To describe such events the distributed system theory has established different failure models. These models can either be used to investigate the behavior of a system in case of a failure or to find mechanisms to make the system fault tolerant. For a fault tolerant system it is important to ensure that it stays operational even if some of the nodes get faulty.

An important function required in many distributed fault tolerant systems is to decide on a common value, even in case some nodes of the system are faulty. In such a setting, each node has its own, private input value. Based on all the input values in the system, each node calculates a result value. All results of non-faulty nodes must be the same. The problem is known as the agreement problem and algorithms solving it as consensus algorithms [AW04].

2.4.1 Failure Models

It is impossible to find and describe each single failure which can occur in a distributed system. Therefore the different failure types are grouped into different failure models. These models describe the manifestation of the failures on the algorithmic level.

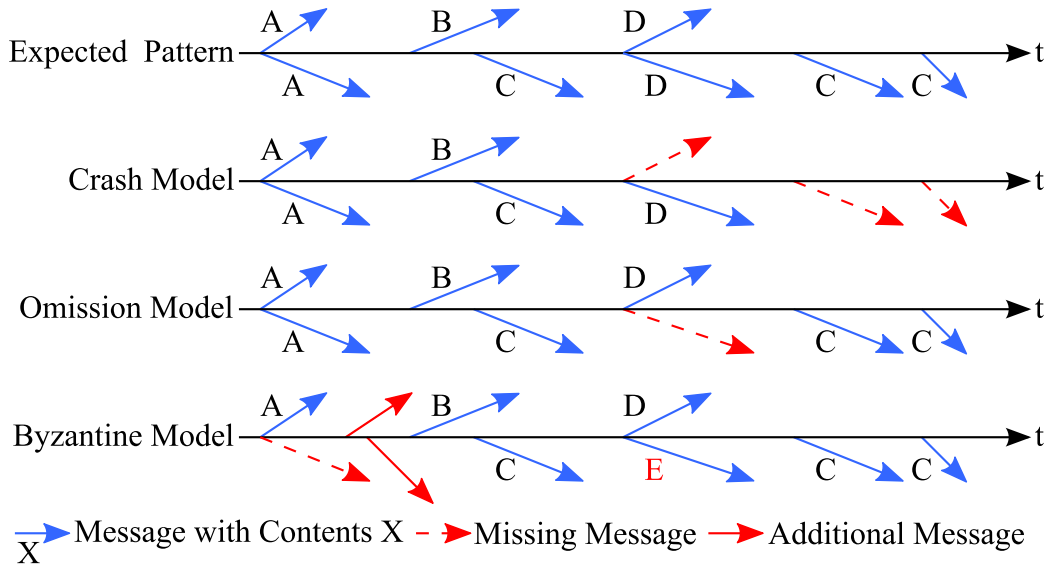


Figure 2.8: Failure Model Example

Depending on the underlying system model, the failures visible at the algorithmic level will be different. Therefore three basic failure models used in distributed systems have been established in literature, namely [DLS88]:

- Crash Failure Model
- Omission Failure Model
- Byzantine Failure Model

They differ in their complexity and the severity of the allowed failures. An important differentiation is, if a failure model allows benign failures [AW04]. In a benign failure model, no illegal operations (like sending additional or conflicting messages) may be executed. For non-benign failure models this constraint does not exist.

Figure 2.8 shows an example execution of a faulty node within the different failure models. It visualizes the differences in the allowed failure patterns. There are more models [AW04], but they are not needed here.

Crash Failure Model

When using the crash failure model, a faulty processor would behave non-faulty up to a certain point in time. At this point it will crash, which means that it will fail to send any further messages from this time onward. If at the time of crash multiple messages should be sent, only a subset of these messages may be sent. This model is a benign failure model.

Omission Failure Model

If a node fails to send or receive a message at a certain point in time, an omission failure has occurred. In contrast to the crash failure model, the node may send further messages afterwards. This model is a benign failure model.

Byzantine Failure Model

Byzantine or arbitrary failures do not limit the nature of the failures in any way. The nodes within the system may even behave malicious and therefore try to actively derail the execution of the algorithm. Additionally faulty nodes can coordinate with each other to maximize the effects of their malicious behavior. It is also possible that conflicting messages are sent to different nodes. Furthermore additional messages, which are not specified by the executed algorithm, may be generated. This model therefore is a non-benign failure model.

2.4.2 Agreement Problem

The calculation of a common (output) value on different nodes is a crucial functionality of a distributed system. No problem in a fault free environment, it becomes more challenging with the power of the used failure model.

An algorithm solving the agreement problem is called a consensus algorithm. Formally such an algorithm is defined as follows [AW04, DLS88].

The system consists of a set P of m nodes $(\{n_0, n_1, \dots, n_m\})$. Each node n_i has an input value v_i out of a value domain V . Goal of the algorithm is to compute a common output value v .

An f -resilient consensus algorithm is considered correct, iff the following properties hold, when at most f nodes are faulty:

- Consistency: All non faulty nodes decide to the same value.
- Termination: In every infinite execution each non faulty node decides eventually.
- Unanimity:
 - Strong Unanimity: If all initial values are v and if any non faulty node decides, it decides v .
 - Weak Unanimity: If all initial values are v , all nodes are non faulty and if any node decides, it decides v .

Depending on the chosen failure model of the underlying system, different implementations of a consensus algorithm are known [AW04]. It is important to note that there is no consensus algorithm available in the asynchronous model (see [FLP85]).

Byzantine Agreement

Most interesting is the agreement problem in the presence of Byzantine failures [LSP82]. At most f out of at least $3f + 1$ nodes (for a lower bound proof see [AW04]) may experience Byzantine failures. Independent of the behavior of the faulty nodes, the non faulty nodes must select a common output value in a bounded amount of time. An algorithm solving this problem is presented below.

Byzantine EIG Algorithm

The Byzantine EIG (exponential information gathering) algorithm [AW04] is a lock-step synchronous algorithm which solves the Byzantine agreement problem with strong unanimity, a minimum of required nodes ($3f + 1$), in the minimum number of rounds ($f + 1$), but with messages of exponential size. Its functionality is as described below.

The main component of the algorithm is a tree structure stored by each node. It contains all information gathered about the other nodes. An example of the tree can be found in Figure 2.9. The value V_{xy} means the value V received from node y , which had received it from node x . No node index will be present more than once in the subscript, therefore no information is processed multiple times by the same node. The tree is built as follows:

- Round 1: In the first round each node sends its input value to all other nodes. When receiving an input value it is stored in the current tree level, if no value is received, a default value is stored.
- Round k , $1 < k \leq f + 1$: In each successive round each node sends its current tree level to all other nodes. When receiving a tree level it is again stored in the tree.

After receiving tree level $f + 1$, the tree is locally used to calculate the output value using a resolve function (mainly a combination of multiple majority votes).

For a detailed description and a correctness proof of the algorithm, please see [AW04].

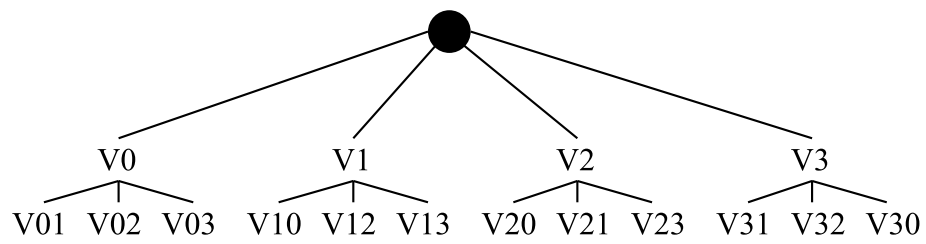


Figure 2.9: EIG Tree for a $f = 1$ System

Chapter 3

Digital Electronic Circuit Basics

Digital electronic circuits consist of combinational logic gates (like AND- and OR-gates) and sequential elements (like flip-flops and latches). The circuit function is defined by the interconnection of such gates. In this chapter we will summarize some of the theoretical concepts used to design high speed circuits.

3.1 Clocking Schemata

An important function within digital circuits is the coordination of the sequential elements. Several schemata for accomplishing this task are known.

3.1.1 Synchronous Circuits

The most popular clocking paradigm used today is the synchronous scheme [Wak01, FH90, Sei79]. Although in use for several decades, it is still state of the art. Virtually all commercially designed circuits are implemented using this paradigm.

It uses a centralized clock source (see Figure 3.1), mostly a quartz oscillator, and all operations on the sequential elements are aligned with respect to this global clock and no local synchronization information is needed. Conceptually all sequential elements receive a clock tick at the exactly same point in time.

As appealing as the presence of an ideal global clock is, as problematic the system analysis can get. Increasing clock- and signal frequencies [Con03], therefore decreasing timing safety margins, and increasing gate count tend to make the system analysis more and more challenging. Furthermore increasing error rates [Con03], due to smaller critical charges and lower voltage swings, must be taken into account nowadays when designing synchronous circuits. Thanks to the high degree

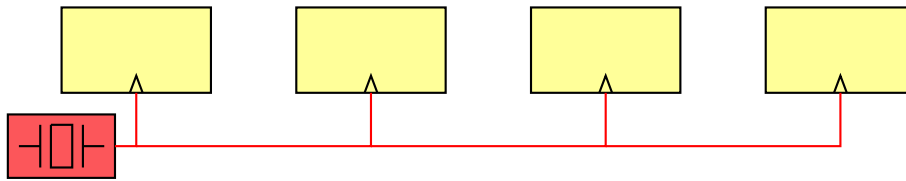


Figure 3.1: Synchronous Model

of automation in the design process and the very sophisticated tool support, these problems can still be handled.

Another disadvantage of this paradigm is its single point of failure introduced by the global clock network. A failure within this network can result in the malfunction of the whole system. Interestingly this problem is widely ignored [MFMR04].

Another highly problematic and challenging task, especially in high performance systems, is the design of the clock network. To keep the synchronous abstraction valid, the differences in the delays from the clock source to the different sequential elements (the skew within the clock network) must be rigorously controlled [Fri01]. This is especially problematic for large clock networks. The usage of special topologies, like e.g. forks and trees, nevertheless lead to acceptable results, but render clock routing an art of its own.

As a big advantage, the synchronous paradigm provides a very accurate time base with a precision of significantly less than one (nearly perfect synchronization, 0 in the idealized case).

3.1.2 Asynchronous Circuits

A completely different approach is the asynchronous paradigm [Hau95]. The most appealing form of asynchronous logic is the delay insensitive model. It uses local synchronization information, a so called handshake, to coordinate the operation of adjacent sequential elements. A sender uses a request (*req*) signal (either an explicit signal line or implicitly encoded into the data) to signalize the availability of new data. It does not change the data until the receiver has acknowledge their reception using an explicit acknowledge (*ack*) signal. Figure 3.2 shows an example of an asynchronous circuit using an explicit request and acknowledge signal.

Due to the absence of any timing bounds and any single point of failure conceptually very appealing, it suffers on the lack of tool support and is therefore commercially scarcely used. Furthermore delay insensitive solutions introduce a significant implementation overhead (like null convention logic (NCL) [FB96]) and/or a timing overhead. Nevertheless there are already working implementation examples available (e.g. [SFGP09]).

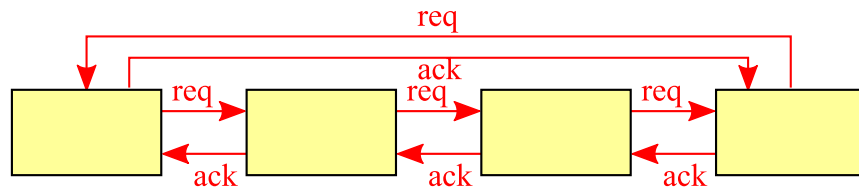


Figure 3.2: Asynchronous Model

Unfortunately no global timebase can be established when using the delay insensitive model due to the unknown message delays (infinite uncertainty)[AW04].

3.1.3 Globally Asynchronous Locally Synchronous (GALS) Circuits

An approach to circumvent the restrictions of the synchronous paradigm is the globally asynchronous locally synchronous (GALS) model [Cha84]. The system is split into multiple modules and these modules are internally clocked according to the synchronous model, each using a local clock source. They are unsynchronized. The inter-module communication is implemented asynchronously (see Figure 3.3).

An advantage of this approach is that each module can be implemented as synchronous circuit utilizing the existing powerful toolsets. Due to the small size of the modules, their design and analysis is much easier and faster than the analysis of a large fully synchronous system. A big drawback is the limited communication speed between the modules. Due to the lack of synchronization flow control is needed to enable a secure data transfer between the modules. This limits the communication throughput [TGL07] significantly. Furthermore no global timing information is available.

To circumvent these drawbacks, several approaches have been suggested to loosely synchronize the clocks of the different modules [TGL07]. Unfortunately the wide

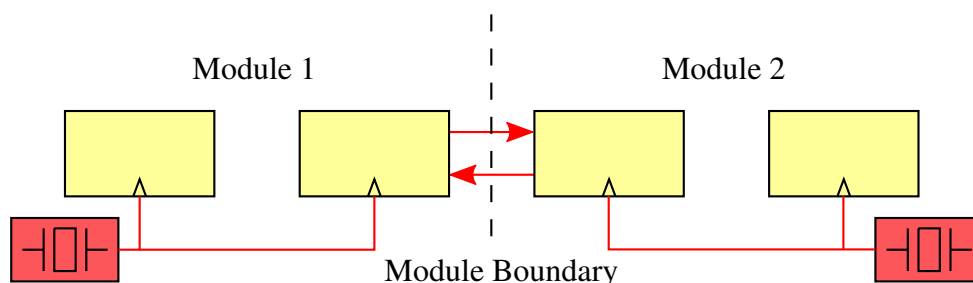


Figure 3.3: GALS Model

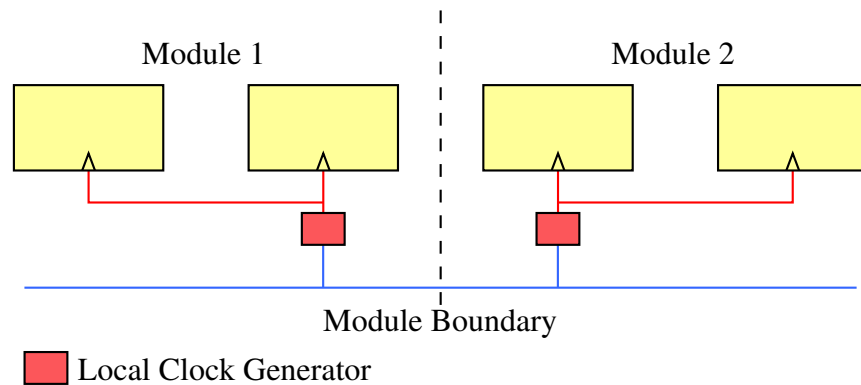


Figure 3.4: Multisynchronous Model

ranging assumptions made on the clock drift rate to securely implement these approaches may not hold in practice.

3.1.4 Multisynchronous Circuits

A relatively new clocking approach are multisynchronous circuits [SG03]. This model guarantees a bounded clock skew, but in contrast to the synchronous approach the skew can get larger than one clock cycle. To guarantee the skew bound, a certain amount of coordination is needed between the different clock sources.

Like in the GALS model, the circuit is divided into synchronous modules. Each of the modules is driven by a different clock of the multisynchronous ensemble (see Figure 3.4) and can be implemented using the standard synchronous model. Therefore the existing powerful toolsets can be used for the designs.

A big advantage of this paradigm is the presence of a global timebase. Its precision equals the skew bound of the clocking system.

As shown by the DARTS clocking scheme (algorithm introduced in [WS05]), it is possible to implement [FFSK06] such a clocking scheme in a fully distributed and fault tolerant fashion.

3.2 Intermodule Communication in Electronic Circuits

The last section has given an overview on how to synchronize different modules of an electronic circuit. Another important question is how to communicate between these modules. In the following we will discuss different approaches for communicating between such modules.

3.2.1 Synchronous Communication

The most commonly used approach for communicating within digital electronic circuits is the synchronous communication scheme [Wak01]. The data transmission is related to a global synchronous clock. This clock defines the validity of the data. Therefore it is only implementable within the synchronous paradigm. At each active clock edge (falling and/or rising edge) a new data item is written by the sender and the receiver reads the data at the next active edge.

Based on the physical implementation of the sequential circuit elements, the following properties arise:

- Output Delay (t_{out}): Specifies the time the data needs to appear at the output of a sequential element after an active clock edge.
- Transmission Delay (t_{data}): Specifies the time it takes a data bit to move from the sender's output to the receiver's input.

For a save operation the following requirements must be fulfilled:

- Setup Time (t_{su}): Specifies the time the data must be stable at the receiver input before an active clock edge to guarantee a safe operation.
- Hold Time (t_{h}): Specifies the time the data must be stable at the receiver input after an active clock edge to guarantee a safe operation.

Based on these properties and requirements, the following constraints for a safe operation are important (T is the period length of the clock signal):

$$T > t_{\text{out}} + t_{\text{data}} + t_{\text{su}} \quad (3.1)$$

$$t_{\text{h}} < t_{\text{out}} + t_{\text{data}} \quad (3.2)$$

Note that normally Equation 3.2 is easy to guarantee, while optimizing the period length of the clock signal according to Equation 3.1 is very challenging. To ensure a reliable operation, these constraints must be met at all operational conditions. To guarantee this, a timing analysis has to be done [Sei79, FH90, Fri01, HO71]. All paths in a system are analyzed under worst case conditions and based on the resulting information a maximum clock frequency is calculated. The use of worst case conditions leads to a clock frequency which is much lower than the frequency achievable in the average case.

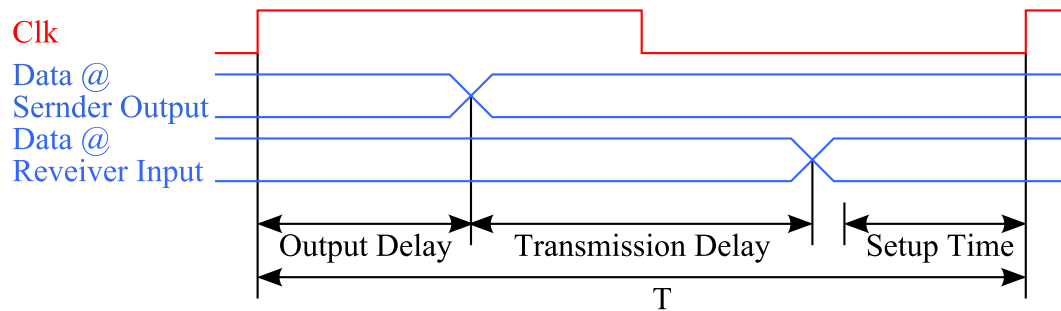


Figure 3.5: Synchronous Communication

3.2.2 Sourcesynchronous Communication

Due to the relatively inefficient nature of the synchronous communication scheme a new intermodule communication scheme was developed for high speed systems [AJTR98]. The advantage is that the transmission delay of intermodule traces, which is normally much greater than the transmission delays within a module, no longer limits the clock frequency of the system. Additionally this scheme can also be used in conjunction with GALS and multisynchronous clocking schemes.

To enable the bit regeneration at the receiver the sender clock signal is transmitted in addition to the data. This can be done either directly, by transmitting the clock signal itself on a separate line, or encoded within the data. In either case, a transition of the sender clock marks the validity of the data. An example can be found in Figure 3.6.

As in the synchronous case, no back pressure mechanism is implemented. Therefore the receiver clock must at least be as high as the sender clock, otherwise data will be lost. Therefore this scheme is usable only with restrictions in conjunction with the

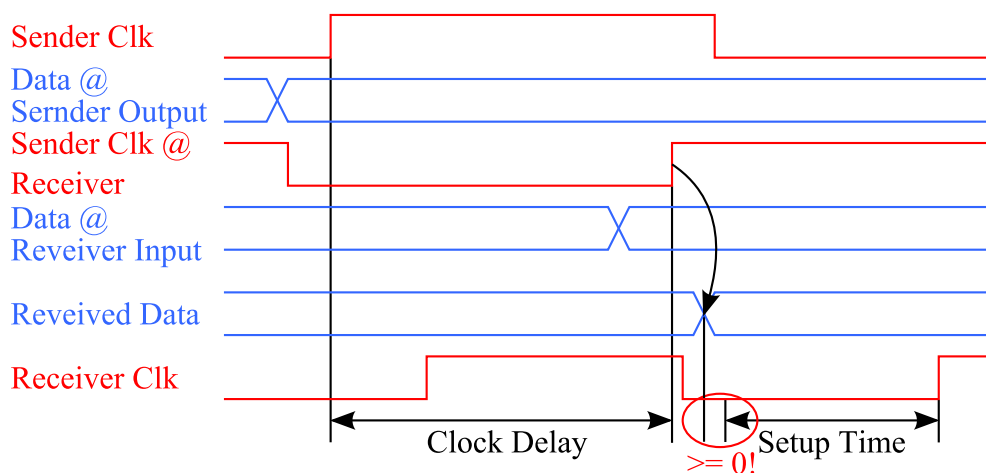


Figure 3.6: Source Synchronous Communication

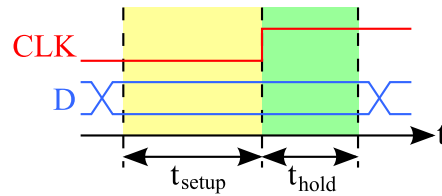


Figure 3.8: D-Flip-Flop Timing Constraints

changes at the exact moment the sequential element tries to capture it? The naive answer would be that the value is either the value before or after the transition. Unfortunately this is not true. In this section we will investigate this behavior in more detail.

3.3.1 Flip-Flop Timings

As already mentioned in Section 3.2.1, several timing constraints have to be maintained when working with synchronous sequential elements like flip-flops. As these elements are heavily used in synchronous systems, we will now examine these constraints in more detail. The analysis is based on the standard D-Flip-Flop described in [Wak01]. It can be easily adapted to all other kinds of synchronous sequential elements.

A D-Flip-Flop is controlled by its clock input. The rising or falling edge of the clock signal triggers the capture operation of the data (D) input. To ensure a safe operation, the data input must be stable a certain time before and after each active clock transition [Wak01] (see Figure 3.8). These times are called setup- and hold-time, respectively.

If one of these constraints is violated, the behavior of the flip-flop is undefined and therefore its output (Q) can get undefined, even oscillating, for an unbound range of time (see Figure 3.9) [KC87, CM73]. This behavior is called metastability. It can be transmitted throughout the system [KC87], which means that one flip-flop after another can get metastable. Another problematic possibility is, that two logic elements (combinational and/or sequential) can interpret a metastable input differently due to a slight mismatch of their internal logic thresholds [KC87], resulting in different output values even if they should be the same.

Therefore metastable states are very problematic and must be avoided at all cost. In global synchronous systems only the boundary flip-flops are affected. When using source synchronous or asynchronous communication within synchronous-, multisynchronous- or GALS-systems, metastability can arise at each clock boundary.

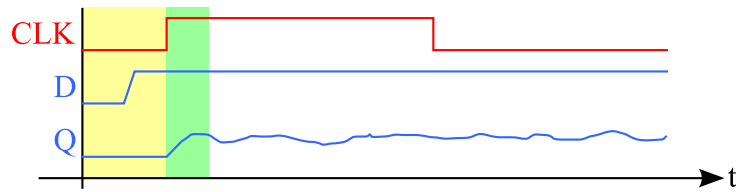


Figure 3.9: Metastable D-Flip-Flop

3.3.2 Avoiding Metastability

The simplest way to avoid metastability is to use a synchronous system with pre-calculated timings. As already discussed, in today's high speed systems this is not applicable any more. Furthermore the metastability occurring at the boundary flip-flops is not handled by this approach.

When using asynchronous communication the handshake signals may be subjected to metastability, while in source synchronous communication the data signal itself may be.

Today metastability is avoided on a statistical basis and not by design, which means that the mean time between failures (MTBF) is increased to a very high value such that metastability becomes very unlikely [DP98]. This is achieved by means of synchronizers. Synchronizers are circuits with the purpose to resolve metastable states. The most commonly used and simplest version is the Two-Flop Synchronizer, a circuit consisting of two serially connected flip-flops [KC87]. The MTBF achievable is good enough for most non-safety critical systems. For safety critical systems more sophisticated circuits have been designed [DP98, KC87].

However, synchronizers alone are not enough. The circuit design must be adapted as well [Gin03, Kin08]. For example in the asynchronous transmission case, only the request and acknowledge signals are allowed to be synchronized, otherwise, due to different transmission and input delays, the data could be interpreted erroneously.

Chapter 4

Communication Standards

When designing a dependable inter-module communication scheme, it is important to analyze and understand existing communication standards. Therefore this chapter introduces different bit representation techniques followed by commonly used line coding algorithms. Afterwards widely used communication standards are described in the light of these concepts.

4.1 Bit Representation

Depending on the method how bits are represented on the transmission line, two fundamental methods for data transmission on electrical lines, namely single-ended- and differential signaling, are known.

4.1.1 Single-Ended Signaling

Single-ended Signaling uses a single rail (transmission line) to transmit the data. Two voltage levels are defined representing the high and low state, respectively, while the ground level is used as reference. When using unipolar signaling [GG04], one of the voltage levels is represented by the ground level itself, while the other is modeled using a different voltage (e.g. 5V for TTL, 3.3V for LVCMOS33). When using bipolar signaling [GG04] two voltage levels symmetric to the ground level ($\pm V$) are used. Figure 4.1 shows an example of unipolar single-ended signaling.

While unipolar signaling is widely used within digital electronic circuits, bipolar signaling, due to the possibility of DC free transmission, is primarily used in middle and wide range signal transmission. Several other mechanisms of single-ended signaling are known [GG04], but since they are primarily used in telecommunication links, they are not described here.

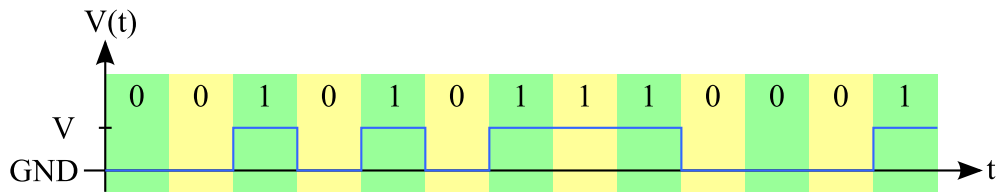


Figure 4.1: Unipolar Single-Ended Signaling (idealized)

4.1.2 Differential Signaling

While single-ended signaling uses only the absolute voltage level on a single rail to transmit a signal, differential signaling [Nat08] utilizes the voltage difference between two rails to encode the data. These rails are called positive (P) and negative (N). Figure 4.2 shows an example signal transmission using the differential transmission scheme. As apparent from the example, one rail transmits the data directly, while the other rail transmits the inverted data. Therefore both rails must be routed length matched, in the best case parallel, to minimize the skew between both rails.

Normally the voltage difference between the rails is kept low, so the rails can change from one state to the other very fast resulting in higher transmission rates as in the single-ended case.

A problem with high speed signaling, single-ended or differential likewise, is that the impedance of the transmission lines must be matched to the impedance of the transmitter as well as the impedance of the line termination [Nat08], otherwise reflections will occur and in the worst case destroy the signal completely. Therefore extra care must be taken when designing systems with high speed signal lines.

A widely used differential signaling standard is LVDS [Nat08]. The direction of a constant current is used to encode the bits. The line is terminated using a resistor of $100\ \Omega$ between the two signal rails. The receiver detects the voltage drop (= the differential voltage) at the terminating resistor to decode the bits. This voltage drop is only about $\pm 350\text{ mV}$, enabling high speed signaling up to 3.125 Gbps [Nat08]. A typical LVDS sender/receiver combination can be found in Figure 4.3.

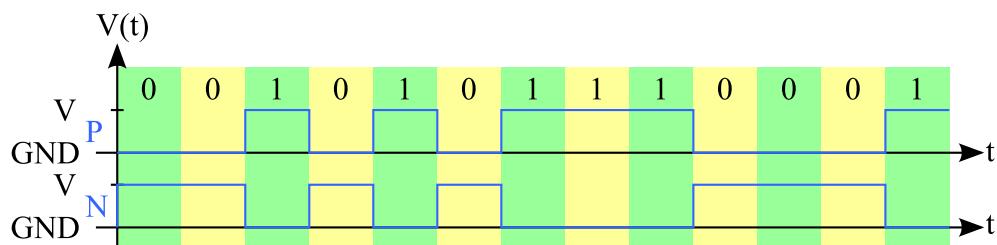


Figure 4.2: Differential Signaling (idealized)

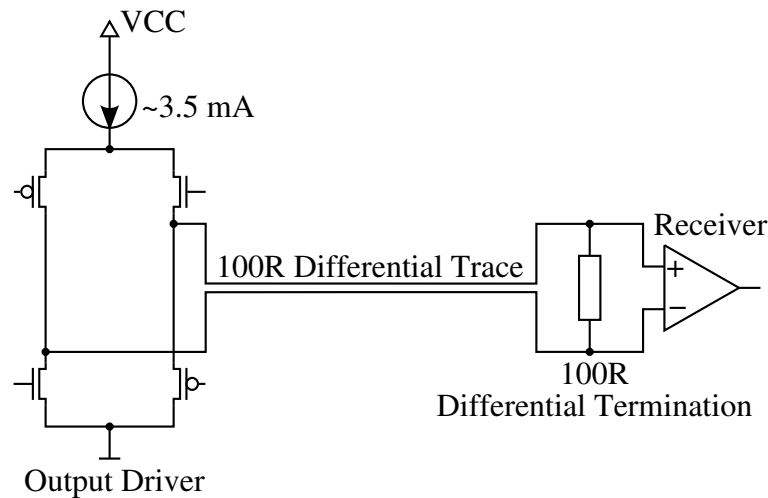


Figure 4.3: LVDS Transmitter and Receiver

4.2 Line Coding

Line coding is needed to enable the receiver to distinguish between the individual bits. This is achieved by adding additional redundancy to the transmitted data stream. Some of the most important techniques will be discussed in this section. They differ in the overhead introduced to the data stream as well as in their implementation costs.

4.2.1 Additional Clock Line

The data itself is transmitted unchanged on the first rail, while the clock is transmitted on a second rail. Therefore the receiver has an exact knowledge of the beginning of each new bit on the data rail. Figure 4.4 contains an example using single-ended signaling. The sampling times are marked by red arrows.

When using differential signaling two rails are needed for each of the transmitted

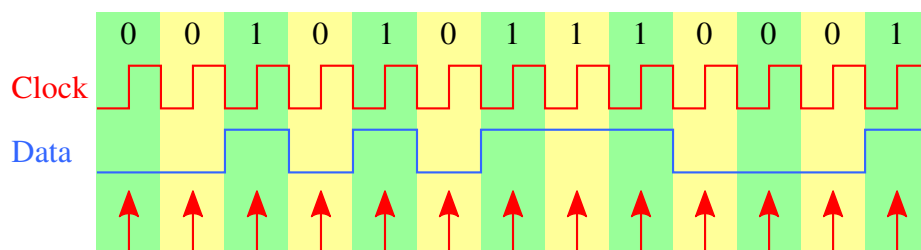


Figure 4.4: Linecoding using an Additional Clock Line

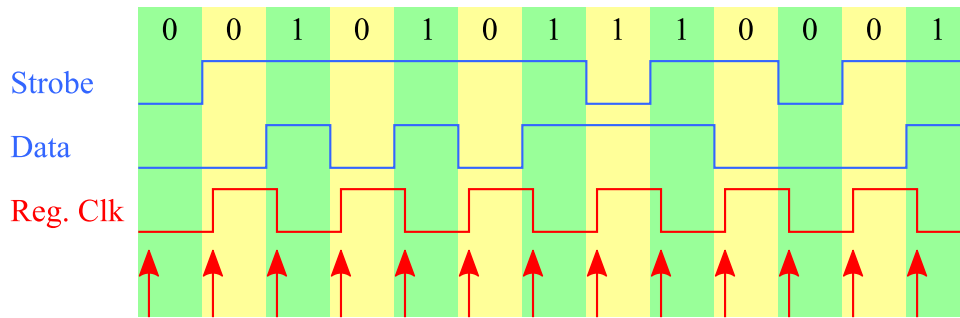


Figure 4.5: Data-Strobe Encoding

signals, namely *DATA_P* and *DATA_N* for the data signal and *CLK_P* and *CLK_N* for the clock signal.

The receiver implementation is straightforward since the clock is known by the receiver. This method introduces a significant bandwidth overhead (the clock frequency is twice the maximum data frequency), as well as doubling the number of needed data rails.

4.2.2 Data-Strobe Encoding

The data strobe encoding technique [IEE95] uses two rails to transmit a bit. They are called Data (D) and Strobe (S). On the Data rail the data stream is transmitted unchanged while on the Strobe rail a transition is made, if two successive bits on the data rail have the same value. An example utilizing single-ended signaling can be found in Figure 4.5.

When using differential signaling, two rails are needed for each of the transmitted signals, namely *DP* and *DN* for the data signal and *SP* and *SN* for the strobe signal.

The receiver can regenerate the clock by simply xor-ing the Data and the Strobe rails. It is important to note that both edges of the regenerated clock signal are active and therefore the receiver must react on both. In Figure 4.5 the sampling times are again marked by red arrows.

The overhead this technique introduces is the second rail. The maximum frequency, and therefore the needed bandwidth on both rails equals the bandwidth of the unencoded data signal.

4.2.3 8B/10B Encoding

The line coding algorithms already discussed transmit the sender clock to the receiver. This is achieved by adding a second data rail. Both methods are very sensitive

to skew between the data rails and therefore the usage of length matched routing is essential.

This restriction can be circumvented, if the timing information is directly encoded into the data stream. 8B/10B encoding [Nat08], which is a special case of the general aB/bB encoding scheme, uses this paradigm. It encodes an eight bit data word into 10 bits. The encoding and decoding is table based. The advantage is that the resulting code can be made DC offset free [Nat08] and the clock can be regenerated out of the stream using a PLL circuit [Nat08] at the receiver. There is no additional bandwidth needed, but the resulting transfer rate is slightly decreased due to the additionally added data-bits. Since only a single rail is used, no skew effects can occur. Unfortunately the implementation with a PLL requires clocks with a low drift rate, otherwise the PLL may lose its lock and the data regeneration will fail.

4.2.4 Asynchronous Parallel Communication

When using asynchronous parallel communication handshake signals define the validity of the data bus. As already described in Section 3.2.3, a signal *req* is used to mark the validity of the data and a signal *ack* to acknowledge the data reception. A timing example can be found in Section 3.2.3.

4.2.5 Asynchronous Serial Communication

When using asynchronous serial communication with a predefined baud-rate, the data transmission is marked with a start- and one or multiple stop-bits [GG04]. Each of these bits has a predefined value (e.g. start-bit is low and stop-bit is high). The start-bit is used to signalize the start of a transmission while the stop-bit marks its end. The receiver can (re-)synchronize to this bit combination and its internal timing tolerance must be only as good as to receive one block, often only a single byte, of data. To enable a appropriate alignment of the sampling point the internal clock rate of the sender and receiver are normally considerably higher than the maximum baud rate. Figure 4.6 shows a transmission example using eight data-, one start- and one stop-bit.

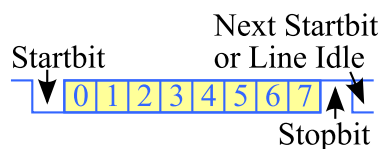


Figure 4.6: Asynchronous Serial Communication

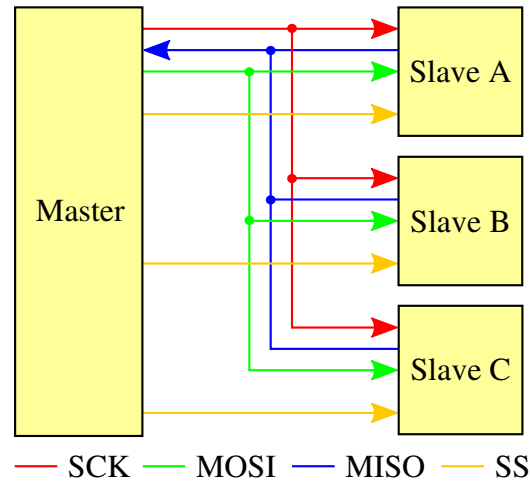


Figure 4.7: SPI Link

4.3 Synchronous Communication Protocols

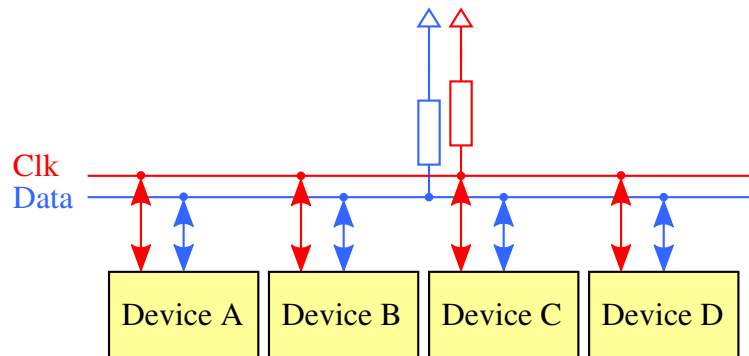
4.3.1 Basic Synchronous Transmission

Used in serial and parallel fashion, the synchronous data transmission is the easiest way to transmit data streams between modules. As already described in Section 3.2.1, all modules have the same clock and therefore only the plain data stream has to be transmitted between the modules. This scheme typically uses single-ended signaling. The capture times of the data bits are directly derived from the global clock signal.

4.3.2 SPI

SPI (Serial Peripheral Interface) [Mil04] is a widely used serial communication method (e.g. SD memory cards, microcontrollers). Unfortunately there is no official standard, nevertheless the devices of the different manufacturers are compatible with each other.

SPI is a single master system which uses a separate clock line (*SCK*) and single ended signaling. The data is transmitted on a single line from master to slave (*MOSI*) and on a second one from slave to master (*MISO*). A specific slave is selected through a dedicated slave select line (*SS*). Therefore $(3 + \#slaves)$ rails are necessary (see Figure 4.7). No maximum data rate is specified.

Figure 4.8: I²C Bus

4.3.3 I²C

I²C [Nxp07] uses one clock and one data rail for the communication. The output drivers of the devices are open-drain drivers and the bus is pulled up by resistors (see Figure 4.8). Therefore the high level is resistive while the low level is dominant. Like SPI it uses single ended signaling on both rails.

I²C is a multi-master system. Bus arbitration is done by monitoring both lines. If a line unexpectedly goes low, the device assumes that another transmission is in progress and aborts its transmission attempt. The maximum specified data transfer rate is 1000 kbps.

4.3.4 PCI

The PCI bus [Pci98] is a parallel high speed bus used in personal computers to communicate with add-in cards. It uses a single clock line and a 32 or 64 bit combined data- and address-bus for data transmission. Furthermore a set of control and status signals are present. On all rails single-ended signaling is used. The access to the bus is controlled by an arbiter which selects a single master for each bus transaction.

The clock rate for the PCI bus is specified as either 33 MHz or 66 MHz. Therefore the peak transmission rate is defined as 4224 Mbps.

4.4 Sourcesynchronous Communication Protocols

4.4.1 Space-Wire

The Space-Wire standard [Esa03] was developed with a focus on communication buses for space devices. It is a full duplex point to point connection and uses Data-

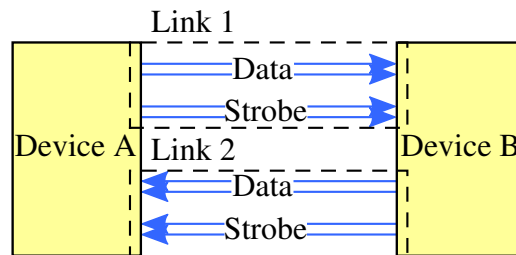


Figure 4.9: Full Duplex Space Wire Link

Strobe encoding with LVDS signaling. Therefore four rails are needed for each direction. Figure 4.9 shows a full duplex Space Wire link.

The maximum transmission rate is not defined explicitly in the standard. Nevertheless, the Space-Wire homepage¹ states 200 Mbps as maximum rate.

4.4.2 PCI-Express

PCI-Express [BAS03] replaces the old parallel buses (PCI, AGP) within personal computer systems. Several speed grades are available (from x1 to x32). The difference between the grades is the number of parallel lanes, starting with a single receive and a single transmit lane at x1 and going up to 32 receive and 32 transmit lanes for x32.

PCI-E uses 8B/10B line coding and a differential signaling technique which is not compatible to LVDS. The maximum data transfer rate on a x1 link is 2.5 Gbps.

4.4.3 Infiniband

Infiniband [Sha02] is widely used for CPU interconnect networks in clusters. The basic (serial) x1 link supports a data transfer rate of 2.5 Gbps. It supports LVDS signaling (besides optical fiber) and uses 8B/10B line coding.

4.5 Asynchronous Communication Protocols

4.5.1 Serial Port

The standard PC serial port, aka. RS-232 or V.24, is a direct implementation of the asynchronous serial data transmission (see Section 4.2.5). Additionally to the data bits, a parity bit could be transmitted giving a limited degree of data protection.

¹<http://spacewire.esa.int/content/Home/HomeIntro.php>

The serial port is full duplex, which means that there is a separate rail from the PC to the device and one in the other direction. Therefore the basic implementation uses only two data rails. Additional signals for flow control and other control functions could be added to the basic implementation [GG04].

4.5.2 Parallel Port

The standard PC parallel port based on the IEEE standard IEEE1284-2000 [IEE00] is a half-duplex parallel bus consisting of an eight bit wide data bus, the handshake signals and additional status and control signals. It is a direct implementation of the asynchronous parallel transmission scheme as described in Section 4.3. All rails use single-ended signaling.

4.6 Transmission Protocol Comparison

As conclusion to this chapter, Table 4.1 gives a short summary of all presented communication protocols and their main properties.

Table 4.1: Transmission Protocol Comparison

Transmission Protocol	Number of Rails	Signaling	Maximum Transmission Speed	Synchronization
Synchronous Transmission	≥ 1	Single-Ended	Limited only by System Delays	Synchronous
SPI	3 + # of Slaves	Single-Ended	Not Defined	Synchronous
I ² C	2	Single-Ended	1000 kbps	Synchronous
PCI (32 Bit) @33 MHz	33 + control/status	Single-Ended	1056 Mbps	Synchronous
PCI (32 Bit) @66 MHz	33 + control/status	Single-Ended	2112 Mbps	Synchronous
PCI (64 Bit) @33 MHz	65 + control/status	Single-Ended	2112 Mbps	Synchronous
PCI (64 Bit) @66 MHz	65 + control/status	Single-Ended	4224 Mbps	Synchronous
Space-Wire	4x2	Differential	200 Mbps	Source Synchronous
PCI-Express x1	2x2	Differential	2.5 Gbps	Source Synchronous
PCI-Express x32	32x2x2	Differential	80 Gbps	Source Synchronous
Infiniband x1	2x2	Differential	2.5 Gbps	Source Synchronous
Serial Port	2	Single-Ended	Not Defined, Commonly 115 kbps	Asynchronous
Parallel Port	10 + control/status	Single-Ended	Not Defined	Asynchronous

Chapter 5

Hardware Fault Models and Fault Tolerance

The system- and communication schemes presented in Chapter 3 have assumed that the underlying circuitry is fault free. Unfortunately, due to manufacturing imprecisions and material variations, not all circuits are manufactured fault free. Even if a high percentage of these faults are found and most of the defective circuits are rejected in the factory test, faults may still be present in some of the manufactured circuits. Additionally adverse operation conditions, like radiation, electromagnetic fields or extreme temperature, can cause runtime errors, even in fault-free circuits. The resulting faults can either be permanent or transient.

A permanent fault, like a break in a signal line (originating in electro migration [Bla69] or deformation due to extreme temperature differences, e.g.), will not disappear on itself. The fault is present until the system is manually repaired.

A transient fault, like a wrong signal state (caused by a single event upset (SEU) [KH04], e.g.), on the other hand is only temporarily present in the system and will disappear on its own. Nevertheless its consequences, like a compromised system state, may be visible much longer.

To be able to model such events consistently, several fault models have been described on the functional level. Due to the importance of reliable circuitry, a sound theory on fault modeling was established, as summarized in [EA97].

Today's, high requirements on the safety and reliability of electronic circuits have led to mechanisms for introducing a certain level of fault tolerance into the circuits. One of the most important concepts used today are TMR (triple modular redundancy) circuits [LV62]. They achieve fault tolerance by replicating the application logic and use a majority vote on their results. Therefore one out of three replicas can be faulty without affecting the result value of the system.

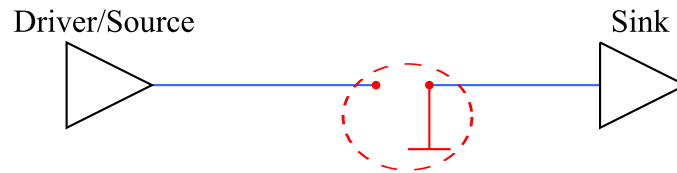


Figure 5.1: Stuck At Zero Fault

This chapter will give an overview on hardware fault models and afterwards introduce the TMR model.

5.1 Hardware Fault Models

As already mentioned above, several functional fault models were established ([EA97]). We will continue with summarizing the properties of some important fault models.

5.1.1 Stuck-At Faults

The stuck-at fault model describes faults manifesting themselves as constant signal values independent of the value driven by their corresponding source. A stuck-at-0 fault therefore describes a signal constantly tied to low, while the stuck-at-1 fault describes a signal constantly tied to high. In CMOS circuits this faults are equivalent with a short circuit of the signal with GND or VCC, respectively. Figure 5.1 shows an example of a stuck at zero fault.

As up to 95% of all circuit faults are detectable using test vectors generated for the stuck-at model, in practice mostly this model is applied.

5.1.2 Stuck-Open Faults

Stuck-open faults have a similar behavior as stuck-at faults. The difference is that, in case of stuck-open faults, the signal line is broken instead of tied to a constant value. Depending to the position of the break and the functionality of the circuit, this can lead to floating signals or even turn, by disconnecting parts of the circuit, a combinational logic element into a sequential one. An example of a stuck-open fault, creating a floating signal line, can be found in Figure 5.2.

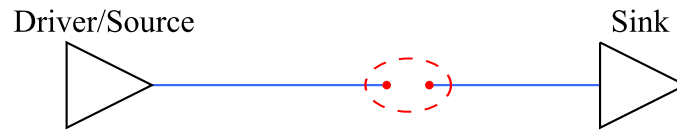


Figure 5.2: Stuck-Open Fault

5.1.3 Bridging Faults

The stuck-at and stuck-open fault models can handle faults of a single line only. As an example for a model describing faults between multiple sources, the bridging fault model is introduced. It models unwanted connections between multiple signal lines. Depending on the underlying circuit, the bridging of the lines may cause different behavior (wired-and, wired-or).

An example for a simple or-bridging of two signal lines can be found in Figure 5.3.

5.2 Fault Tolerance

After discussing the modeling of faults, we will now show how to make a circuit fault tolerant. For safety-critical circuits or systems without a repair possibility, like unmanned space crafts, it is important to tolerate a predefined number of faults. The fault hypothesis, specifying the maximum number of tolerable faults, must be defined before a fault tolerant system can be designed.

A very common assumption is a system which can tolerate a maximum of one fault. For this assumption, triple modular redundancy (shortly TMR) systems are widely used.

5.2.1 Triple Modular Redundancy

As mentioned before, TMR systems [LV62] can handle at most one fault. The basic layout can be found in Figure 5.4. The system consists of three replicas of the

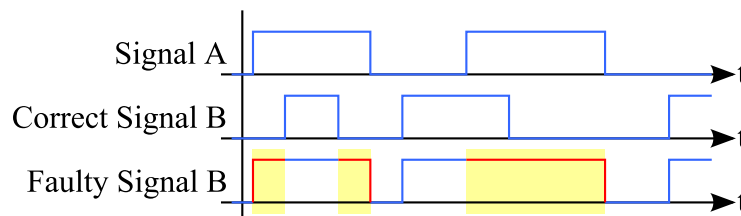


Figure 5.3: OR Bridging Fault

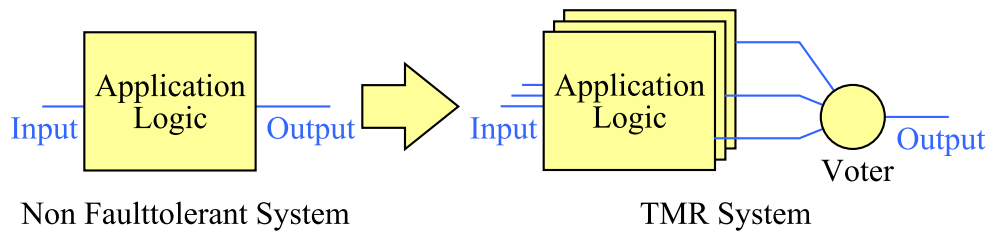


Figure 5.4: Basic TMR System

application logic and a single voter. The task of the voter is to perform a majority vote on its inputs. If the system complies with the fault hypothesis, at most one input will be faulty and therefore the majority of the input values is correct and selected as result. As the voter is a single point of failure, it is normally implemented as simple as possible and is therefore a combinational circuit only [Sho02].

As the voter is a single point of failure, it is apparent that such a combination is not favorable. Therefore the concept has been extended by duplicating the voter [LV62] leading to a slightly different system as shown in Figure 5.5. In this combination the system will still be operational, even if a voter fails. The drawback here is that there is no single, voted result available, but three voter outputs instead.

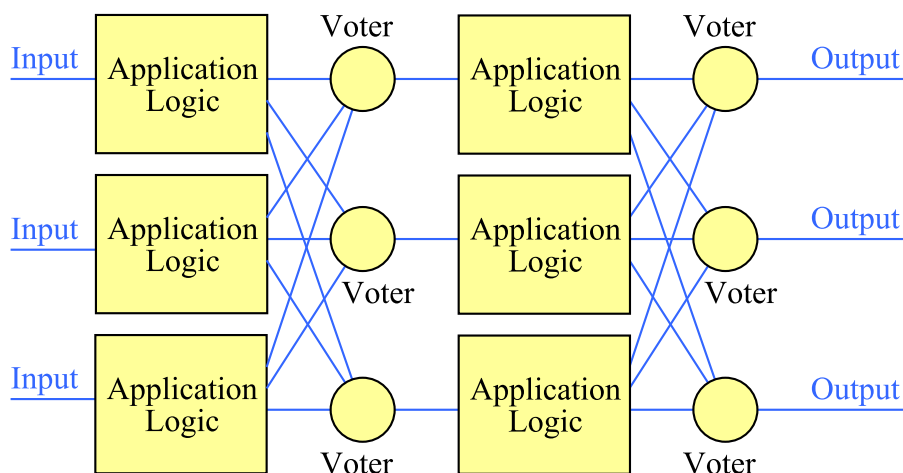


Figure 5.5: TMR System using Replicated Voters

Part II

Framework Implementation

Chapter 6

Problem Definition

To be able to design a dependable hardware implementation of a consensus algorithm, the underlying framework implementation must be reliable. The framework must support the implementation of independent nodes and a reliable way to communicate between these nodes. Even if we could tolerate faults introduced by the framework, as metastability, e.g., we do not want to generate additional faults in an otherwise fault free execution. If such faults would occur, the reliability of the whole system may be compromised. After analyzing existing solutions, we have found that none of them fulfills all of our requirements.

At a first glance, the synchronous model is very appealing. It supports metastability free communication links, is supported by a powerful toolset and is widely used. On the other hand the whole system relies on a single clock source. If it fails, the whole system will crash. Therefore it is impossible to implement independent nodes using this model.

In the asynchronous model the absence of a clock signal in conjunction with the adaptive transmission delays prevents the establishment of a global time base. Therefore only asynchronous algorithms can be executed. Since no asynchronous consensus algorithm is possible [FLP85], this system model is not suited for our needs.

The module structure of a GALS system with its independent clock sources is perfectly suited to implement independent nodes of a distributed system. Unfortunately the absence of a global time base makes the implementation of a consensus algorithm impossible. Assuming bounded delays on the communication links would enable the generation of a global time base. Nevertheless its implementation would be costly. Additionally the tendency for metastability on the intermodule communication links, even in the fault free case, prevents a reliable communication.

Fortunately, the DARTS clocking scheme developed at our institute suits all our needs. It is a Byzantine fault tolerant implementation of a multisynchronous clocking scheme. As in the case of a GALS system, the different nodes can be implemented using independent clock sources. In contrast to the GALS clocking scheme

the precision of the clock sources is known and therefore a bounded global time base is available in the multisynchronous case. In the following sections we will show how to *implement a metastability free communication layer* based on this infrastructure. To be able to prove its correctness, we will model the multisynchronous circuit using mechanisms of the distributed system theory.

We will start with the introduction of our modeling technique. Afterwards, in Chapter 8, the used communication layer model is built and proved correct. This is followed by an outline of the hardware implementation of the communication layer and its performance. Experimental results collected using a test system will also be presented. These results will be used to show that the proved lower bound is tight.

Finally Chapter 10 uses the communication layer and its properties to develop an algorithm which establishes a lock step synchronous round model.

Chapter 7

Circuit Modeling

To be able to prove the correctness of a multisynchronous circuit, we first must map it to a distributed system model. The resulting algorithmic description together with a set of properties will later be used as basis for our proofs.

7.1 Creating the Model

The system is separated into several modules guided by the used clock sources. All sequential elements driven by the same clock source are grouped into the same module. Therefore only a single module will fail, if a clock source is faulty. Each module is represented by a node of the distributed system model.

Since the whole module uses the same clock source, the idealized precision within the module is zero ($\pi = 0$). Additionally the clock frequency of the modules' clock signals is calculated such that the maximum signal delay on the intramodule communication links is less than one clock cycle ($\Delta^+ < 1$). Therefore the modules are purely synchronous circuits and can be designed using the existing powerful toolsets.

Within the distributed system model, the intramodule logic (sequential and combinational) elements are represented by the state transition specification of the corresponding node.

For each identified intermodule communication link the boundary memory elements are identified. All combinational logic elements on the links are merged into this link.

The resulting model consists of a set $P = \{n_0, \dots, n_m\}$ containing the identified nodes. The nodes are interconnected by simple communication links only. An example of modeling a single module can be found in Figure 7.1.

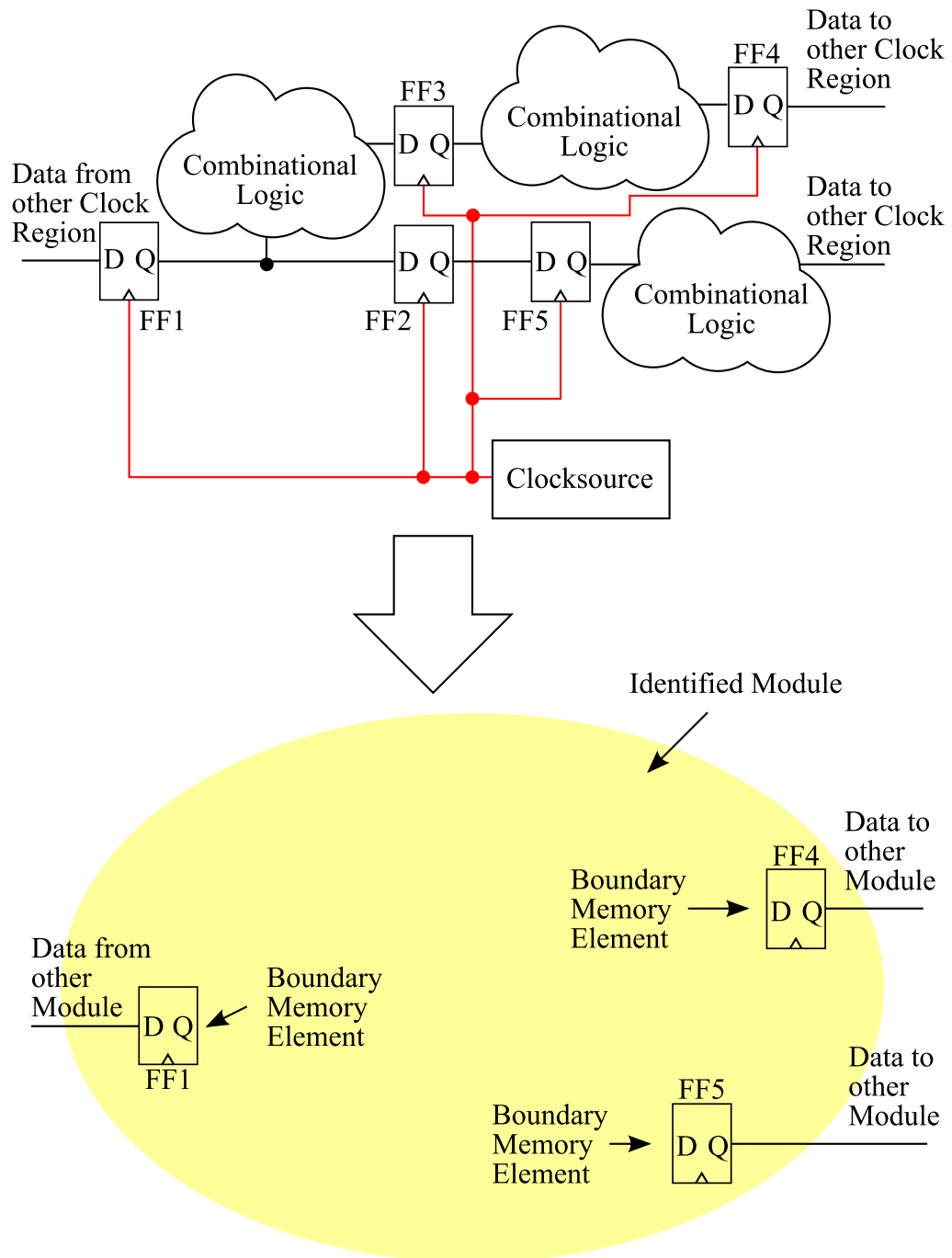


Figure 7.1: Modeling Example

7.2 Model Properties

Due to the usage of a multisynchronous clocking scheme, each node uses a different local clock source. These clocks are synchronized with an a-priory known precision π . The precision π equals the precision of the multisynchronous clocking system. Formalizing the precision π we get:

$$\exists \pi : \forall i, j \in P, \forall k \geq 0 : C_i^k \rightarrow C_j^{k+\pi}.$$

It is important to note that for GALS system no such precision exists and therefore no global time base can be derived from the local clocks.

Additionally, by the accuracy of the clocking system, we know that a minimum clock period T^- exists. Formally this minimum clock period can be specified as:

$$\forall i \in P, k > 0 : \exists T^- = \min_{i,k} (C_i^{k+1} - C_i^k) > 0.$$

For the intermodule communication links, the message delay Δ is bounded using a minimum (Δ^-) and a maximum (Δ^+) delay, respectively. The calculation of the message delay Δ and its bounds will be fixed later (see Section 9.2). The identified boundary elements are necessary to accomplish this task.

7.3 Modeling Freeness of Metastability

Revisiting the definition of metastability (see Section 3.3) it is apparent that a signal, captured by a sequential element, must not be changed within the setup/hold window of the reading element.

As already mentioned we have a system of n nodes exchanging information. We model the access to a signal as unidirectional link connecting an arbitrary, fault-free sender- and receiver pair. The access to the signal is modeled using actions executed by the nodes. We define a write action W , changing the signal, and a read action R , reading it. If we define the end of a write action W as the earliest possible time the read action R can safely start (such that the setup time is not violated), it is sufficient for a metastability free operation that $W \rightarrow R$. Additionally we define the end of the read operation such that the hold time is not violated and the next write action W may safely start.

Additionally to metastability-freeness, we require that each value written is read exactly once. This can be achieved by requesting that $R \rightarrow W$ holds. Therefore a correct execution is defined by the sequence W, R, W, R, \dots .

Using these definitions, a sufficient condition for metastability free communication can be described as:

$$\forall k > 0 : W^k \rightarrow R^k \rightarrow W^{k+1} \tag{7.1}$$

where the action W^k is the k -th write- and the action R^k the k -th read operation. Note that the condition must only hold for links between non-faulty nodes.

As the intramodule logic is purely synchronous and Equation (7.1) is inherent for the synchronous model, the problem is solved for intramodule links by the toolset. Therefore only the intermodule links have to be considered. The next sections will use the previously formulated condition to develop a provable multisynchronous metastable-free intermodule communication infrastructure.

Chapter 8

Metastability-Free Intermodule Communication

In Section 7.3 we have already formulated a sufficient condition for a metastability-free communication. Within this chapter we will use this condition together with the properties of the circuit model (see Section 7.2) to develop a provable metastability free communication scheme. We will proceed by comparing two alternative ideas for implementing metastability-free communication between different modules in a multi-synchronous system, namely:

- “Quasi-synchronous” communication based on a divided clock (“macroticks”) and
- “Pipelined” communication using the native clock (“microticks”).

8.1 Quasi-Synchronous Communication Scheme

Let us call the native clock tick available in the multi-synchronous environment a “microtick”. Dividing this microtick by some fixed d creates a “macrotick” with d -fold period, with the same (absolute) synchronization precision. Still, the precision expressed in the unit macroticks improves by a factor of d : $\pi' = \pi/d$. With M_i^k denoting the k^{th} macrotick of node i , a fixed divisor d and assuming a synchronous start of all nodes, we can formally express this, using the definition of the precision, as

$$\forall i, j \in P, \forall k > 0 : M_i^k = C_i^{dk} \rightarrow C_j^{dk+\pi} \begin{cases} = C_j^{(k+\frac{\pi}{d})d} = M_j^{k+\pi'}, d|\pi \\ \rightarrow C_j^{(k+\lceil \frac{\pi}{d} \rceil)d} = M_j^{k+\pi'}, d \nmid \pi \end{cases}$$

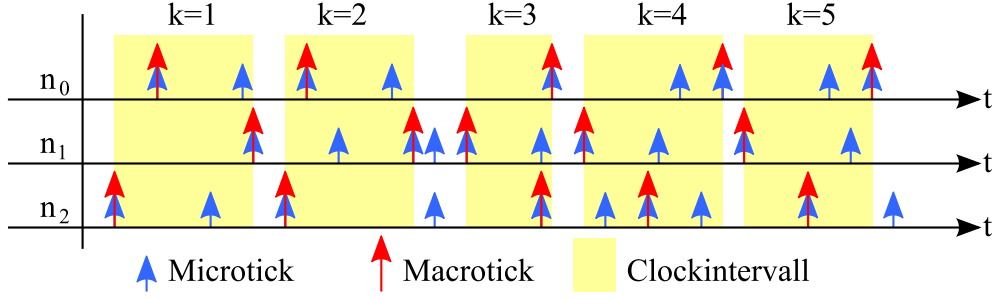


Figure 8.1: Macrotick Generation

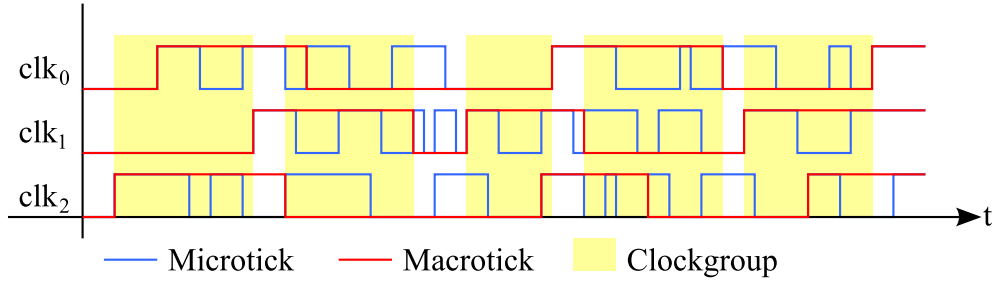


Figure 8.2: Macrotick Based Clock Generation

By choosing $d \geq \pi$, we can reduce the relative precision π' to a value of one, which means that all macroticks with the same index can be grouped into non overlapping time intervals (see Figure 8.1).

Unfortunately we cannot infer anything about the precedence of macroticks with the same index on different nodes. Additionally the delay between macroticks with the same index can be larger than the transmission delay, and the new data could be received too early. Therefore we will distinguish between odd and even macrotick indices and perform a write upon an odd macrotick, and a read upon an even macrotick only. This yields two disjoint subsets R_i and W_i for each node $i \in P$ with $R_i \cap W_i = \emptyset$:

$$\forall i \in P : W_i = \{\forall k > 0 : W_i^k = M_i^{2k-1}\}$$

$$\forall i \in P : R_i = \{\forall k > 0 : R_i^k = M_i^{2k}\}$$

With $\pi' = 1$, the definition of the precision can be used to ensure

$$\forall i, j \in P, \forall k > 0 : W_i^k = M_i^{2k-1} \rightarrow M_j^{2k} = R_j^k \text{ and}$$

$$\forall i, j \in P, \forall k > 0 : R_i^k = M_i^{2k} \rightarrow M_j^{2k+1} = W_j^{k+1}.$$

Such a system is easily implemented using the rising edge of the macrotick-clock as write event and the falling edge as read event. Note that $d = 2\pi$ for this implementation, since both clock edges are used (see Figure 8.2 as example). Basically, this implementation simulates a globally synchronous system, inheriting all its advantages

and disadvantages. On the positive side, it allows to substitute a (non-fault tolerant) central clock by a multi-synchronous clocking scheme without further changes. The major deficiency, however, is the bad communication performance, which only is $\frac{1}{d}$ times the throughput achievable with the native clock.

As in the synchronous model, it is necessary that the bit data on the communication line is delivered timely, since no handshake protocol is implemented. Therefore each bit must be delivered within the time between the latest possible occurrence of a write with index k and the first possible occurrence of a read with the same index. If the time is not sufficient, the divisor d must be increased to a value such that the time difference is large enough for delivering the bits timely.

8.2 Pipelined Communication Scheme

Let us recall the requirement for metastability-free communication: For the transfer of any given data item, we need to pair write and read transitions such that the write action has finished safely before the read action starts. In the above quasi-synchronous approach, clock transitions of the same direction (rising or falling) are considered indistinguishable. Hence, this pairing is applied strictly via subsequent *alternating* edges. Consequently the phase relation between any two nodes is of central importance and must be maintained within tight bounds.

However, if we could distinguish edges on an *individual* basis (e.g. by their index), then we could establish relations between arbitrary clock transitions, such as $C_i^{13} \rightarrow C_j^{22}$. Clearly this requires a globally consistent numbering of clock ticks, which is, however, nothing else than the global time base established by our multi-synchronous clock, provided that a consistent edge numbering is ensured by the synchronous start of all node's local clocks at start-up.

Based on this idea, we can pipeline transmission activities at the microtick level, thereby avoiding the throughput penalty of macrotick-based communication. We simply exploit the precedence given in the definition of the precision:

$$\forall i, j \in P, \forall k > 0 : W_i^k = C_i^k \rightarrow C_j^{k+\alpha} = R_j^k$$

with α being a sufficiently large time margin that separates writes and reads.

Note that writes and reads can be performed at every microtick here, which maximizes the throughput. The pipelined approach hence considerably surpasses the quasi-synchronous scheme, and is therefore our preferred solution. Due to the bad synchronization precision, however, which can be in the order of several microticks, one needs a FIFO buffer in between communicating nodes to avoid data loss. Clearly, minimizing the required buffer size is important, both with respect to costs and communication latency. In the following section, we will provide our solution and its proof of correctness.

8.2.1 Algorithmic Model

To be able to prove the correctness of our approach, we first create an algorithmic model of our system. The links of the basic model from Chapter 7 are replaced by a single-writer single-reader buffer memory of unbounded size (see Figure 8.3). Our proof will reveal that finite buffer size is sufficient.

From Section 7.2 we already know the two basic properties of the model. As basis for the proof we rewrite these properties as assumptions:

Assumption 8.2.1 (Precision). $\exists \pi : \forall i, j \in P, \forall k \geq 0 : C_i^k \rightarrow C_j^{k+\pi}$

Assumption 8.2.2 (Accuracy). $\forall i \in P, k > 0 : \exists T^- = \min_{i,k} (C_i^{k+1} - C_i^k) > 0$

Note that in Assumption 8.2.2 only the lower bound of the accuracy, as it was defined in Section 2.2.1, is used and sufficient to guarantee the correct behavior of our solution. Furthermore we require the clocking system to comply to the following assumptions:

Assumption 8.2.3 (Startup). *Before the first clock tick (initial state, $k = 0$), all buffer memories are prefilled with α elements (all zero) and the precision π is zero ($\pi_0 = 0$).*

Note that Assumption 8.2.3 is usually easy to guarantee in systems with a common reset, since all nodes start with zero clock ticks received.

Assumption 8.2.4 (Message Order). *All message channels (clock as well as data channels) provide FIFO ordering. Furthermore, the actual delays must be such that every read and write action is finished before the next one starts.*

The behavior of the system is modeled by a sender algorithm (Algorithm 1) and a receiver algorithm (Algorithm 2).

Informal description of the algorithms The following messages, actions and events can be handled and/or produced by the sender node i :

- C_i^k - This is the k -th clock tick of the sender node i . It is a zero-length action, i.e., an event.
- $\langle \text{tick}, k \rangle$ - At every event C_i^k the clock generator of node i sends a message to its message generator to initiate the delivery of the data message. Its message delay $\Delta_{\text{send}}(i, k)$ is in the interval $0 < \Delta_{\text{send}}^- \leq \Delta_{\text{send}}(i, k) \leq \Delta_{\text{send}}^+$.
- D_i^l - This is the receive action for the l -th $\langle \text{tick}, k \rangle$ message at its message generator. It is a zero-length action, i.e., an event.

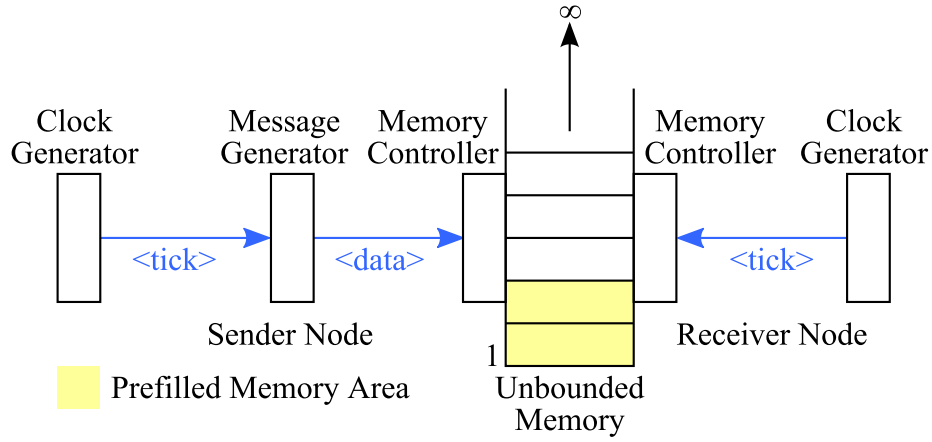


Figure 8.3: System Model used for the Proof

Algorithm 1 Sender Algorithm for Node i

```

1: // Clk Delay
2: on  $C_i^k$  do
3:   Send  $\langle \text{tick}, k \rangle$  to generate  $D_i$ 
4: end on
5: // Message Delay
6: on  $D_i^l$ :  $l$ -th receive of any  $\langle \text{tick}, k \rangle$  from node  $i$  do
7:   Send  $\langle \text{data}, l \rangle$  to generate  $W_i$ 
8: end on
9: // Write Event
10: on  $W_i^m$ :  $m$ -th receive of any  $\langle \text{data}, l \rangle$  from node  $i$  do
11:    $\text{mem}(m + \alpha) := \text{data}$ 
12: end on

```

Algorithm 2 Receiver Algorithm for Node j

```

1: // Clk Delay
2: on  $C_j^k$  do
3:   Send  $\langle \text{tick}, k \rangle$  to generate  $R_j$ 
4: end on
5: // Read Event
6: on  $R_j^l$ :  $l$ -th receive of any  $\langle \text{tick}, k \rangle$  from node  $j$  do
7:    $\text{data} := \text{mem}(k)$ 
8: end on

```

- $\langle \text{data}, l \rangle$ - At every event D_i^l node i 's message generator sends a message to its memory controller to initiate the memory write action. Its message delay $\Delta_{\text{msg}}(i, l)$ is in the interval $0 < \Delta_{\text{msg}}^- \leq \Delta_{\text{msg}}(i, l) \leq \Delta_{\text{msg}}^+$.
- W_i^m - This is the actual write action of the buffer memory. It is triggered by the reception of the m -th $\langle \text{data}, l \rangle$ message and has a non zero duration $\Delta_{\text{mem}}(i, m)$ within the interval $0 < \Delta_{\text{mem}}^- \leq \Delta_{\text{mem}}(i, m) \leq \Delta_{\text{mem}}^+$.

The receiver node j can produce and/or handle the following actions, events and messages:

- C_j^k - This is the k -th clock tick of the receiver node. It is a zero-length action, i.e. an event.
- $\langle \text{tick}, k \rangle$ - At every event C_j^k the clock generator of node j sends its memory controller a message to initiate the memory read. Its message delay $\Delta_{\text{recv}}(j, k)$ is in the interval $0 < \Delta_{\text{recv}}^- \leq \Delta_{\text{recv}}(j, k) \leq \Delta_{\text{recv}}^+$.
- R_j^l - This is the actual read action of the buffer memory. It is triggered by the reception of the l -th $\langle \text{tick}, k \rangle$ message and has a specified length of $\Delta_{\text{rd}}(j, l)$ within the interval $0 < \Delta_{\text{rd}}^- \leq \Delta_{\text{rd}}(j, l) \leq \Delta_{\text{rd}}^+$.

It is important to note that R_j^k reads memory location k , while W_i^k writes memory location $k + \alpha$.

As a consequence of the shifted write index, the memory must be prefilled with α elements (all zero), simulating that the writes $W_i^{-\alpha+1}, \dots, W_i^0$ to the memory locations $1, \dots, \alpha$ have already been finished before the first clock tick ($k = 0$, initial state).

To fulfill Assumption 8.2.4, it is sufficient that the system delays comply with the following equations:

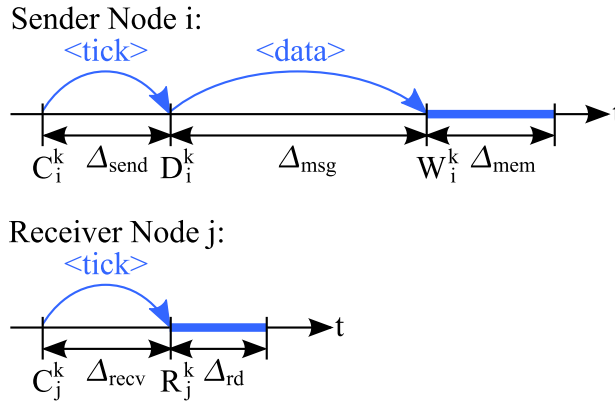
$$T^- + \Delta_{\text{send}}(i, k+1) + \Delta_{\text{msg}}(i, k+1) - \Delta_{\text{send}}(i, k) - \Delta_{\text{msg}}(i, k) > \Delta_{\text{mem}}(i, k) \quad (8.1)$$

and

$$T^- + \Delta_{\text{recv}}(i, k+1) - \Delta_{\text{recv}}(i, k) > \Delta_{\text{rd}}(i, k)$$

A more in-depth discussion of the delays in real systems can be found in Section 9.2.

An example execution of tick k for both algorithms can be found in Figure 8.4.

Figure 8.4: Execution of Tick k

8.2.2 Problem Definition

The correct operation of our communication scheme is formalized by a slightly changed version of the condition introduced in Section 7.3. Therefore the following properties must hold:

- (WR) The write of memory location k must be finished before the read of this location starts ($\forall k > 0 : W_i^{k-\alpha} \rightarrow R_j^k$).
- (OV) In case of a bounded-size buffer, the read of an element must be finished before it is overwritten ($\forall k > 0 : R_j^k \rightarrow W_i^{k+\pi+\beta}$, the size of β will be fixed later).

8.2.3 Relation Between Actions

We will now prove essential relations between the actions in our system model.

Lemma 8.2.1. *Algorithm 1, line 6: $\forall k \geq 1$ it holds that $k = l$ and $D_i^k \rightarrow D_i^{k+1}$.*

Proof. We prove this Lemma by induction.

- Induction start ($k = 1$): C_i^1 triggers the first send of a message $\langle \text{tick}, 1 \rangle$. By the FIFO property of the links it is also the first message to be delivered and therefore triggering event D_i^1 . Since it is the first event the precedence relation is obviously true.
- Induction hypothesis: Assume the lemma holds for k .
- Induction step ($k \rightarrow k + 1$): We know that the first k $\langle \text{tick}, l \rangle$ messages trigger the events $D_i^l, l \leq k$. By FIFO order, message $\langle \text{tick}, k + 1 \rangle$ (generated by event C_i^{k+1}) will be the next one delivered, thereby triggering the event D_i^{k+1} . Since D_i^l is an event (a zero-length action), this implies the precedence relation. \square

Lemma 8.2.2. *Algorithm 1, line 10: $\forall l \geq 1$ it holds that $l = m$ and $W_i^l \rightarrow W_i^{l+1}$.*

Proof. The proof is similar to above.

- Induction start ($l = 1$): D_i^1 triggers the first send of a message $\langle \text{data}, 1 \rangle$. By the FIFO property of the links it is also the first message to be delivered and therefore triggering action W_i^1 . Since it is the first action the precedence relation is valid.
- Induction hypothesis: Assume the lemma holds for l .
- Induction step ($l \rightarrow l + 1$): We know that the first l $\langle \text{data}, m \rangle$ messages trigger the actions $W_i^m, m \leq l$. By FIFO order message $\langle \text{data}, l + 1 \rangle$ (generated by event D_i^{l+1}) will be the next one delivered, thereby triggering the action W_i^{l+1} . From Equation (8.1) we know that $t_s(W_i^{l+1}) > t_s(W_i^l) + \Delta_{\text{mem}}(i, l)$ and therefore W_i^l is finished before W_i^{l+1} is started. Therefore $W_i^l \rightarrow W_i^{l+1}$ holds.

□

We now define a new relation \rightsquigarrow . It is used to model the triggering of actions. $A \rightsquigarrow B$ means that action B was triggered by action A . Note that $A \rightsquigarrow B$ implies the precedence relation ($A \rightarrow B$). Using this notation, the trigger dependencies implied by Lemma 8.2.1 and 8.2.2 read:

$$C_i^k \rightsquigarrow D_i^k \rightsquigarrow W_i^k.$$

Lemma 8.2.3. *Algorithm 2, line 6: $\forall k \geq 1$ it holds that $k = l$ and $R_j^k \rightarrow R_j^{k+1}$.*

The proof is equivalent to the one of Lemma 8.2.2. Lemma 8.2.3 in conjunction with the definition of the trigger relation implies:

$$C_j^k \rightsquigarrow R_j^k.$$

8.2.4 Read–Write Order Proof

For the proof of (WR) we fix an arbitrary sender-receiver pair. The sender node has the index i , the receiver node the index j . We will now derive the latest possible end of a write action to a certain data item. We start with the first α items.

Lemma 8.2.4. $\forall -\alpha + 1 \leq k \leq 0 : t_e(W_i^k) = 0$

Proof. Follows directly from Assumption 8.2.3. □

The following Lemma 8.2.5 gives the latest possible end time for all other write actions.

Lemma 8.2.5. $\forall k > 0 : t_e(W_i^k) \leq t(C_i^k) + \Delta_{\text{send}}^+ + \Delta_{\text{msg}}^+ + \Delta_{\text{mem}}^+$

Proof. We already know that

$$C_i^k \rightsquigarrow D_i^k \rightsquigarrow W_i^k.$$

Since D_i^k is triggered by the k -th $\langle \text{tick}, k \rangle$ message, we get:

$$t(D_i^k) = t(C_i^k) + \Delta_{\text{send}}(i, k).$$

Since W_i^k is triggered by the k -th $\langle \text{data}, l \rangle$ message, we get:

$$\begin{aligned} t_s(W_i^k) &= t(D_i^k) + \Delta_{\text{msg}}(i, k) \\ &= t(C_i^k) + \Delta_{\text{send}}(i, k) + \Delta_{\text{msg}}(i, k) \\ &\leq t(C_i^k) + \Delta_{\text{send}}^+ + \Delta_{\text{msg}}^+. \end{aligned}$$

Furthermore, we know that the action takes $\Delta_{\text{mem}}(i, k)$ time to finish, so its end time is:

$$\begin{aligned} t_e(W_i^k) &= t_s(W_i^k) + \Delta_{\text{mem}}(i, k) \\ &\leq t(C_i^k) + \Delta_{\text{send}}^+ + \Delta_{\text{msg}}^+ + \Delta_{\text{mem}}^+. \end{aligned}$$

□

In the next step, we determine the earliest possible time a read action can start.

Lemma 8.2.6. $\forall k > 0 : t_s(R_j^k) \geq t(C_j^k) + \Delta_{\text{recv}}^-$

Proof. We already know that

$$C_j^k \rightsquigarrow R_j^k.$$

Since R_j^k is triggered by $\langle \text{tick}, k \rangle$, we link:

$$t_s(R_j^k) = t(C_j^k) + \Delta_{\text{recv}}(j, k) \geq t(C_j^k) + \Delta_{\text{recv}}^-.$$

□

For proving (WR), we need to relate the latest possible end of a write action with the earliest possible start of a read action of the same item, namely, $t_s(R_j^k) - t_e(W_i^{k-\alpha}) \geq 0$. In particular, we will show that this condition is true if:

$$\alpha \geq \pi + \left\lceil \frac{\Delta_{\text{send}}^+ + \Delta_{\text{msg}}^+ + \Delta_{\text{mem}}^+ - \Delta_{\text{recv}}^-}{T^-} \right\rceil$$

Lemma 8.2.7. $\forall k > 0 : t_s(R_j^k) - t_e(W_i^{k-\alpha}) \geq 0$

Proof. We use a case differentiation to prove this Lemma:

- $1 \leq k \leq \alpha$:

$$t_s(R_j^k) - \underbrace{t_e(W_i^{k-\alpha})}_{=0 \text{ by Lemma 8.2.4}} = t_s(R_j^k) \geq 0.$$

- $k > \alpha$:

$$\begin{aligned} t_s(R_j^k) - t_e(W_i^{k-\alpha}) &\geq t(C_j^k) + \Delta_{\text{recv}}^- - t(C_i^{k-\alpha}) - \Delta_{\text{send}}^+ - \Delta_{\text{msg}}^+ - \Delta_{\text{mem}}^+ \\ &= \underbrace{t(C_j^k) - t(C_i^{k-\pi})}_{\geq 0 \text{ by Assumption 8.2.1}} + \underbrace{t(C_i^{k-\pi}) - t(C_i^{k-\alpha})}_{\geq (\alpha-\pi)T^- \text{ by Assumption 8.2.2}} \\ &\quad + \Delta_{\text{recv}}^- - \Delta_{\text{send}}^+ - \Delta_{\text{msg}}^+ - \Delta_{\text{mem}}^+ \\ &\geq \left(\underbrace{\pi + \left\lceil \frac{\Delta_{\text{send}}^+ + \Delta_{\text{msg}}^+ + \Delta_{\text{mem}}^+ - \Delta_{\text{recv}}^-}{T^-} \right\rceil}_{\leq \alpha} - \pi \right) T^- \\ &\quad + \Delta_{\text{recv}}^- - \Delta_{\text{send}}^+ - \Delta_{\text{msg}}^+ - \Delta_{\text{mem}}^+ \\ &\geq T^- \frac{\Delta_{\text{send}}^+ + \Delta_{\text{msg}}^+ + \Delta_{\text{mem}}^+ - \Delta_{\text{recv}}^-}{T^-} + \\ &\quad + \Delta_{\text{recv}}^- - \Delta_{\text{send}}^+ - \Delta_{\text{msg}}^+ - \Delta_{\text{mem}}^+ = 0. \end{aligned}$$

□

This proof shows that if the buffer is prefilled with at least

$$\alpha = \pi + \left\lceil \frac{\Delta_{\text{send}}^+ + \Delta_{\text{msg}}^+ + \Delta_{\text{mem}}^+ - \Delta_{\text{recv}}^-}{T^-} \right\rceil$$

elements, no element is read before it is written.

8.2.5 Bounded Buffer Size

Now we replace the unbounded memory with a FIFO buffer of bounded size. We will continue with determining a lower bound for a sufficient buffer size such that (OV) holds.

As in the previous section, we will show the start and end time, respectively, of the actions. To determine the required buffer size, we need to correlate the earliest

possible start time of the write action with the latest possible end time of the read action (excluding the first α writes, since they are prefilled and therefore can not overwrite any item in the buffer).

Lemma 8.2.8. $\forall k > 0 : t_s(W_i^k) \geq t(C_i^k) + \Delta_{\text{send}}^- + \Delta_{\text{msg}}^-$

Proof. We already know that

$$C_i^k \rightsquigarrow D_i^k \rightsquigarrow W_i^k.$$

Since D_i^k is triggered by the k -th $\langle \text{tick}, k \rangle$ message, we get:

$$t(D_i^k) = t(C_i^k) + \Delta_{\text{send}}(i, k).$$

Since W_i^k is triggered by the k -th $\langle \text{data}, l \rangle$ message, we get:

$$\begin{aligned} t_s(W_i^k) &= t(D_i^k) + \Delta_{\text{msg}}(i, k) \\ &= t(C_i^k) + \Delta_{\text{send}}(i, k) + \Delta_{\text{msg}}(i, k) \\ &\geq t(C_i^k) + \Delta_{\text{send}}^- + \Delta_{\text{msg}}^-. \end{aligned}$$

□

Lemma 8.2.9. $\forall k > 0 : t_e(R_j^k) \leq t(C_j^k) + \Delta_{\text{recv}}^+ + \Delta_{\text{rd}}^+$

Proof. We already know that

$$C_j^k \rightsquigarrow R_j^k.$$

Since R_j^k is triggered by the k -th $\langle \text{tick}, k \rangle$ message, we get:

$$t_s(R_j^k) = t(C_j^k) + \Delta_{\text{recv}}(j, k) \leq t(C_j^k) + \Delta_{\text{recv}}^+.$$

Knowing that a read action finishes within $\Delta_{\text{rd}}(j, k)$, we get:

$$\begin{aligned} t_e(R_j^k) &= t_s(R_j^k) + \Delta_{\text{rd}}(j, k) \leq t_s(R_j^k) + \Delta_{\text{rd}}^+ \\ &\leq t(C_j^k) + \Delta_{\text{recv}}^+ + \Delta_{\text{rd}}^+. \end{aligned}$$

□

After calculating the start and end time of the actions, we will now show the maximum possible number of unread messages in the buffer.

Lemma 8.2.10. *There are always equal or less than $\pi + \alpha + \beta$ unread elements in the buffer (i.e., $\forall k \geq 0 : t_s(W_i^{k+\pi+\beta}) - t_e(R_j^k) \geq 0$) with $\beta = \left\lceil \frac{\Delta_{\text{recv}}^+ + \Delta_{\text{rd}}^+ - \Delta_{\text{send}}^- - \Delta_{\text{msg}}^-}{T^-} \right\rceil$.*

Proof. We have to distinguish two cases:

- $k = 0$ (Initial State): At the beginning there are the α prefilled elements in the buffer. Therefore the buffer size is surely sufficient.
- $k > 0$:

$$\begin{aligned}
t_s(W_i^{k+\pi+\beta}) - t_e(R_j^k) &\geq t(C_i^{k+\pi+\beta}) + \Delta_{\text{send}}^- + \Delta_{\text{msg}}^- \\
&\quad - t(C_j^k) - \Delta_{\text{recv}}^+ - \Delta_{\text{rd}}^+ \\
&= \underbrace{t(C_i^{k+\pi+\beta}) - t(C_j^{k+\beta})}_{\geq 0 \text{ by Assumption 8.2.1}} + \underbrace{t(C_j^{k+\beta}) - t(C_j^k)}_{\geq \beta T^- \text{ by Assumption 8.2.2}} \\
&\quad + \Delta_{\text{send}}^- + \Delta_{\text{msg}}^- - \Delta_{\text{recv}}^+ - \Delta_{\text{rd}}^+ \\
&\geq \left\lceil \frac{\Delta_{\text{recv}}^+ + \Delta_{\text{rd}}^+ - \Delta_{\text{send}}^- - \Delta_{\text{msg}}^-}{T^-} \right\rceil T^- \\
&\quad + \Delta_{\text{send}}^- + \Delta_{\text{msg}}^- - \Delta_{\text{recv}}^+ - \Delta_{\text{rd}}^+ \\
&\geq \Delta_{\text{recv}}^+ + \Delta_{\text{rd}}^+ - \Delta_{\text{send}}^- - \Delta_{\text{msg}}^- \\
&\quad + \Delta_{\text{send}}^- + \Delta_{\text{msg}}^- - \Delta_{\text{recv}}^+ - \Delta_{\text{rd}}^+ = 0.
\end{aligned}$$

□

This means that if the buffer has a size of at least $\pi + \alpha + \beta$ elements, no item can be overwritten, before it was read.

8.2.6 Latency

The message latency is defined as the number of clock cycles between the write- ($W_i^{k-\alpha}$) and the read-action (R_j^k) of all non prefilled messages ($\langle \text{data}, k \rangle, k > \alpha$). The calculation is based on the local clocks of the sender and receiver nodes.

Lemma 8.2.11. *The message latency of all messages $\langle \text{data}, k \rangle, k > \alpha$ is α .*

Proof. Each messages $\langle \text{data}, k \rangle$ is written at the corresponding action $W_i^{k-\alpha}$ to the buffer memory (follows directly from line 11 of Algorithm 1). It is read out at action R_j^k (following directly from line 7 of Algorithm 2). Therefore the message latency L is:

$$L = k - (k - \alpha) = \alpha$$

□

8.2.7 Results

Theorem 8.2.1 (Buffer Size). *For $\alpha = \pi + \left\lceil \frac{\Delta_{\text{send}}^+ + \Delta_{\text{msg}}^+ + \Delta_{\text{mem}}^+ - \Delta_{\text{recv}}^-}{T^-} \right\rceil$ a sufficient FIFO buffer size is given by*

$$2\pi + \left\lceil \frac{\Delta_{\text{send}}^+ + \Delta_{\text{msg}}^+ + \Delta_{\text{mem}}^+ - \Delta_{\text{recv}}^-}{T^-} \right\rceil + \left\lceil \frac{\Delta_{\text{recv}}^+ + \Delta_{\text{rd}}^+ - \Delta_{\text{send}}^- - \Delta_{\text{msg}}^-}{T^-} \right\rceil.$$

Theorem 8.2.2 (Message Latency). *For $\alpha = \pi + \left\lceil \frac{\Delta_{\text{send}}^+ + \Delta_{\text{msg}}^+ + \Delta_{\text{mem}}^+ - \Delta_{\text{recv}}^-}{T^-} \right\rceil$, the message latency ($R_j^k - W_i^{k-\alpha}$) is defined as:*

$$\pi + \left\lceil \frac{\Delta_{\text{send}}^+ + \Delta_{\text{msg}}^+ + \Delta_{\text{mem}}^+ - \Delta_{\text{recv}}^-}{T^-} \right\rceil.$$

Theorem 8.2.1 gives a lower bound on the sufficient buffer size such that the read-write order is always maintained. Therefore it is guaranteed that no memory location of the bounded FIFO buffer is changed while it is read, ruling out *all* possibilities for metastability in the fault free case.

Additionally our solution guarantees that each bit is seen exactly once by the receiver so no bit is duplicated or swallowed as it could happen when using a communication scheme without handshake in conventional GALS systems.

Chapter 9

Pipelined Scheme Implementation

Chapter 8 has illustrated the principle of the proposed communication scheme and theoretically proven its correctness. We will now show how it can be efficiently implemented in practice.

9.1 Circuit Design

The layout of a node is shown in Figure 9.1. Internally the nodes are implemented using the synchronous paradigm, enabling the usage of standard development tools and testing facilities.

The needed buffering of the transmitted data is performed at the receiver input.

9.1.1 Communication

Note that the clock domain boundary is in the buffer that in turn is located at the receiver. The transmitter clock is used for clocking the data into the buffer memory. Therefore a source synchronous communication protocol is needed.

After evaluating several different communication schemes (see Chapter 4 for details) we decided to use an unidirectional SPI connection without slave select signals as communication infrastructure. Therefore each link exists of two rails, namely a clock- and a data-rail. The transmission of the clock signal is necessary because it is impossible to regenerate the clock signal from the data-rail using a PLL since it can not be locked to the multisynchronous clock signal (unpredictable clock phase differences between succeeding clock cycles). It is also superior to the usage of other clock transmission schemes (like Data-Strobe encoding) in case the transmission rate is increased using multiple data-rails (only $1 + n$ rails needed for a transmission using n data-rails). For a safe transmission, it is necessary to keep the skew

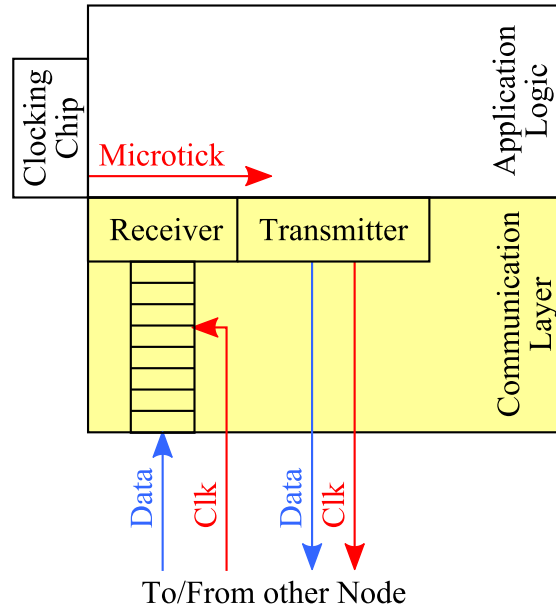


Figure 9.1: Layout of a Node

between these rails within the margins known from synchronous designs. To relax these margins as far as possible we introduce a 180° phase shift between data and clock.

Furthermore we decided to use point-to-point links in favor of buses. When using a bus topology, faulty nodes can, by sending messages outside the assigned communication slots, interfere with the communication between non faulty ones which would contradict a reliable communication between non faulty nodes (babbling idiot [Kop97]).

9.1.2 Transmitter

The transmitter is intended to operate as a peripheral slave device of a controller implemented in the nodes application logic. It is in the same clock domain as this controller. Data to be transmitted is passed by the controller via an 8 bit register interface.

In case no data is available from the controller, the transmitter is responsible for inserting idle patterns to keep the buffer memory filled. In any case line coding is applied according to the standard 8B/10B coding algorithm and the data is serialized. The encoding is needed to distinguish between data and idle patterns. For future extensions a functionality to group multiple data bytes into packets is implemented by means of a start- and an end-packet-symbol.

9.1.3 Receiver

The receiver operates as a synchronous peripheral slave device for a controller unit implemented in the receiving node. It simply takes the data out of the buffer using its own (i.e. its controller's) local clock. After deserializing and decoding the data it provides them to its local controller unit via a memory mapped parallel 8-bit interface. Additionally the reception of the three defined control symbols (idle, start- and end-packet) is signaled at this interface

9.1.4 Communication Buffer

As explained above, both, the transmitter as well as the receiver independently write data to and read data from, the communication buffer, respectively, each using its own local clock, while these clocks may have an arbitrary relative but bounded difference. It is the duty of the buffer to accommodate this difference and allow for a metastability-free data transfer. As shown in Chapter 8 this can be accomplished with a sufficiently large buffer. Still the hardware implementation must be suitable for the purpose. A naive approach like a shift register, e.g., would not work, as such a structure updates *all* registers with every write access, due to the need for shifting. Even with infinite buffer size this solution would not allow a metastability-free communication.

A correct approach is to use a ring buffer with individual address pointers for input and output (Figure 9.2). Here only one memory element is updated per write access, and our only concern is to prevent an overlap between write and read of the same entry. This, however, is ensured by the proof from Chapter 8 for a sufficient buffer size (Theorem 8.2.1).

As a result we have a memory structure that can be written to and read from independently, since the only potential conflict, namely an overlapping access to the same address from both sides is ruled out by design. This at the same time also rules out metastability – in sharp contrast to other communication schemes that make metastability highly improbable but cannot completely eliminate it. Note that this is only possible due to the known, bounded precision that the multisynchronous clocking scheme provides.

A similar buffer implementation was previously used by [PG07]. Since their approach does not utilize a clocking scheme with bounded precision, special handshake signals (*full* and *empty*) are needed to prevent buffer over- or underflows. As the *full* signal crosses the clock boundary between sender and receiver, it is subjected to the possibility of metastability. Therefore a synchronizer is necessary, contradicting the advantages of the buffer. Their second approach, for mesosynchronous systems, only works for constant or slowly drifting clock differences and would fail in our setting.

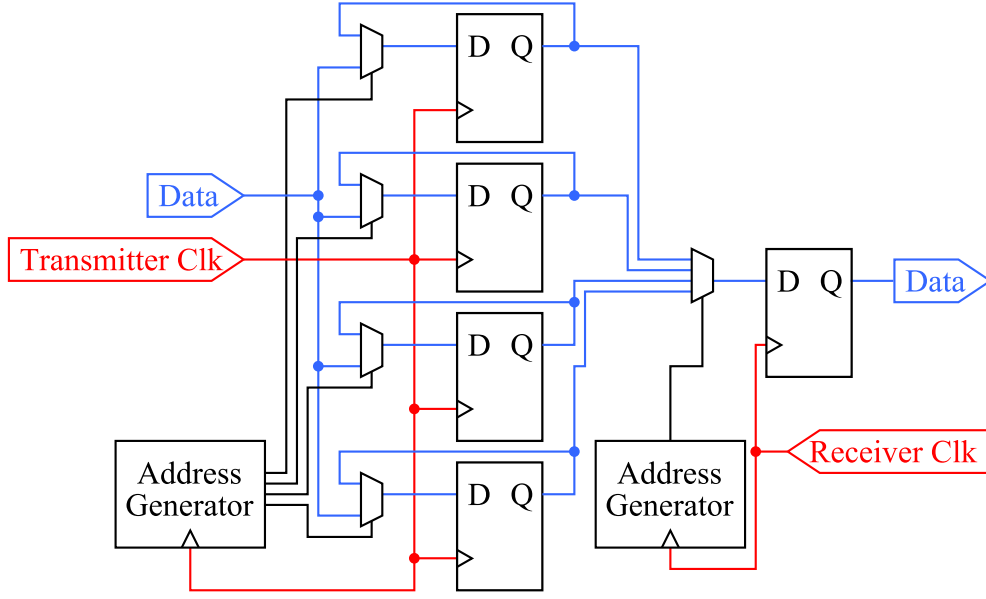


Figure 9.2: Ring Buffer of Depth 4

9.2 Implementation Mapping

It is important to analyze, whether the implementation respects all the assumptions made by the theoretical model.

The Assumptions 8.2.1 and 8.2.2 are guaranteed by the clocking system (see Section 7.2).

A global asynchronous reset signal, initializing all communication layers and clocking chips, ensures the validity of Assumption 8.2.3. Since the clocking chips are also reset and have a certain startup time, it is ensured that no clock edge will be present in the vicinity of the reset pulse. Therefore the metastability problems [Kin08], normally associated with asynchronous reset signals, do not apply here.

9.2.1 Implementation without Input Register

We will now show how the delay assumptions are mapped to the implementation. Figure 9.3 shows a schematic transmitter circuit including the corresponding delays.

Assumption 8.2.4 is naturally inherent in the synchronous paradigm. In detail it specifies that the delays only change by a certain amount between two ticks. Since differences in delays only stem from part variations (constant over time) and (slowly) changing operation conditions this assumption holds.

The messages $\langle \text{tick}, k \rangle$ are in fact the DARTS clock ticks. The delays $(\Delta_{\text{send}}(i, k))$

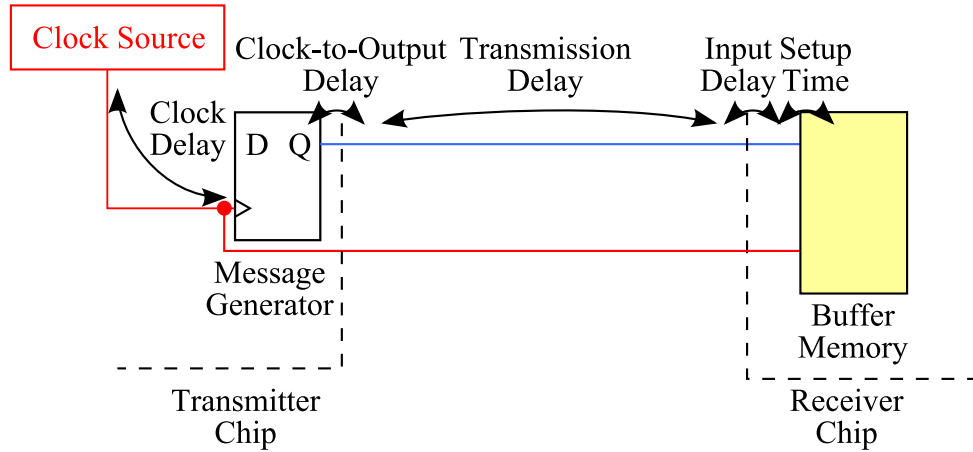


Figure 9.3: Schematic Transmitter Circuit (No Input Register)

and $\Delta_{\text{recv}}(j, k)$) are the times needed for the ticks to reach the sender output element and the read port of the memory element, respectively.

The message $\langle \text{data}, l \rangle$ is implemented as the data and clock edges running on the line between the sender and receiver. The corresponding delay ($\Delta_{\text{msg}}(i, k)$) is the sum of the clock-to-output-delay, line-delay, input-delay and the setup-time (t_{su}) of the buffer memory. Obviously the lines that convey all these messages respect FIFO order.

The execution length of the read and write operations can also be directly mapped to the hardware. The length of the write operation ($\Delta_{\text{mem}}(i, k)$) is the sum of the time needed for the new data to reach the memory output element (including all combinational logic on its path) and the setup time (t_{su}) needed for this output element. The length of the read operation ($\Delta_{\text{rd}}(j, k)$) is mapped to the hold time (t_{h}) of the memory output element.

9.2.2 Implementation with Input Register

As the implementation of a source synchronous interface on an FPGA is very challenging, especially if no dedicated I/O register is used, a specialized implementation scheme for FPGAs was created. To relax the routing constraints required at the receiver node, and therefore simplifying the task of implementing the interface drastically, the transmitted data can be buffered using a single register before it is written to the buffer memory. It can be implemented as I/O register on FPGAs. This moves the message generator register from the physical transmitter- to the receiver-chip. Nevertheless it is under control of the transmitter. Figure 9.4 shows the new situation.

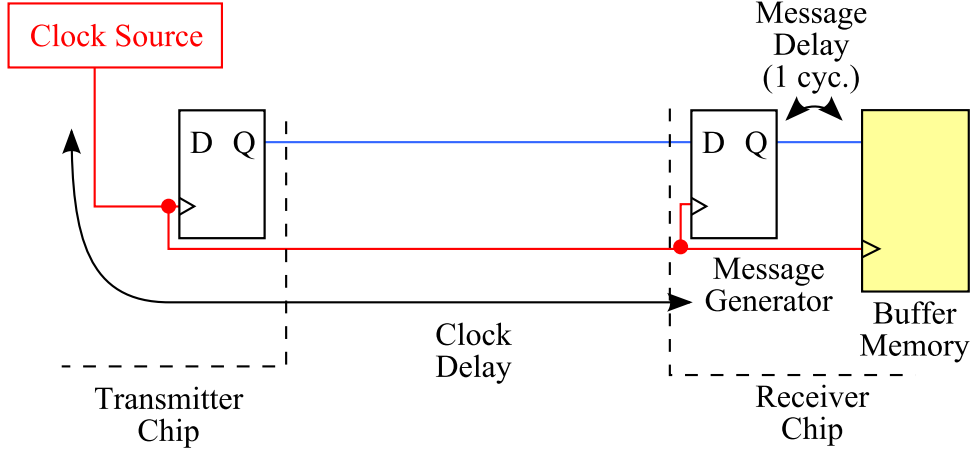


Figure 9.4: Schematic Transmitter Circuit (With Input Register)

The delay $\Delta_{\text{send}}(i, k)$ add to the time it takes the clock ticks to travel from the clock source, through the transmitter chip, over the transmission line, into the receiver chip and to the message generator register. The delays of the data- and clock-rail must still be matched.

The message delay ($\Delta_{\text{msg}}(i, k)$) is decreased to exactly one clock cycle, since the communication between the input register and the buffer memory is fully synchronous and guaranteed to hold by the design tools.

All other delay mappings are unchanged in comparison to the implementation without an input register.

9.3 Performance and Efficiency

The implementation costs for our solution are currently very low (see Figure 9.2). Two important performance parameters of a communication subsystem are its throughput and its latency.

9.3.1 Throughput

Our implementation achieves a (gross) data rate of 1 Mbps/MHz since a new data bit is transferred with every active clock edge. Multiplying this throughput by using parallel data-rails is straightforward. Therefore we reach a data rate of up to 24 Mbps within our test system. A system using an 100 MHz clock and 10 parallel data-rails would achieve a data rate as high as 1 Gbps.

Table 9.1: Latency

Input Register Precision [cyc.]	No			Yes		
	3	4	5	3	4	5
Sender Latency [cyc.]	2	2	2	2	2	2
Single Bit Latency [cyc.]	4	5	6	5	6	7
Receiver Latency [cyc.]	3	3	3	3	3	3
Message Length [bit]	10	10	10	10	10	10
Overall Latency [cyc.]	19	20	21	20	21	22

9.3.2 System Latency

The latency of a single byte message ($C_j^{\text{recv}} - C_i^{\text{send}}$) is calculated as follows:

- The message must be processed by the transmitter implementation. In the current version this takes 2 clock cycles.
- The message travels on the communication line between the nodes bit by bit. The latency of each bit is defined by Theorem 8.2.2 as

$$\pi + \left\lceil \frac{\Delta_{\text{send}}^+ + \Delta_{\text{msg}}^+ + \Delta_{\text{mem}}^+ - \Delta_{\text{recv}}^-}{T^-} \right\rceil.$$

Since each byte is sent encoded as 10 bits, an additional latency of 10 cycles must be added (time needed to transmit the whole message).

- The current version of the receiver needs additional 3 clock cycles to process the data.

The latency of an eight bit message is the sum of all these terms. Table 9.1 contains the message latency for different precision values. The latency values were calculated for a system that has a single bit latency of $\pi + 1$ cycles, if no additional input register is used, and a single bit latency of $\pi + 2$ cycles, if one is used.

All latency values are calculated for the case that the controller and the transmitter are synchronized, which means that the message is handed to the transmitter exactly at the start of a new transmission slot. If the message is sent unsynchronized, an additional latency up to 9 clock cycles may occur (synchronization latency).

9.3.3 Performance Comparison

To get a better understanding of the performance of our implementation, we will compare the single bit latency and throughput of our system with the values of synchronous and GALS systems (see Table 9.2).

Table 9.2: Performance Comparison

	Single Bit Latency	Throughput
Synchronous System	1 Cyc.	1 Mbps/MHz
GALS Feedforward System	$2 + \lceil \frac{\Delta}{T} \rceil$ Cyc.	1 Mbps/MHz
GALS Feedback System	$2 + \lceil \frac{\Delta}{T} \rceil$ Cyc.	$\frac{1}{1+2 \text{ Latency}}$ Mbps/MHz
Multisynchronous Communication	$\pi + \lceil \frac{\Delta}{T} \rceil$ Cyc.	1 Mbps/MHz

As in a synchronous system the data bit is written at an active clock edge and read at the next one, the single bit latency of such a system is exactly one cycle. Due to the fact that on each clock edge one bit is transmitted, the throughput is 1 Mbps/MHz.

The corresponding values for GALS systems are a little bit harder to calculate. In a simple feedforward system with a two-flop synchronizer [KC87], the message latency is given by the two cycles the synchronizer needs for processing the data and the communication latency on the transmission line (Δ) measured in clock cycles of the receiver clock (with period T). As such a setting does not guarantee, due to the different clock drift of the sender and receiver clock, the reception of all data bits, its applicability is very limited.

If the reception of each data bit is necessary, a feedback GALS system [Gin03] must be used. Therefore a signal defining the validity of the data is added. Additionally the reception of the data is acknowledged using a signal generated by the receiver and read by the sender. The latency of the data transmission is the same as in the previous case. As the sender must wait with sending the next data bit until the receiver has acknowledged the previous one, the throughput is reduced by the forward latency, the time to generate the acknowledge signal and the latency of the acknowledge signal. If we assume a symmetric system, the two latency values are the same. The minimum time needed to generate the acknowledge signal is one cycle.

9.4 Communication Example

To illustrate the proposed hardware implementation let us take a look at an example communication. A logic analyzer trace of a successful node to node communication can be found in Figure 9.5. It was generated by the test system (see Section 9.5) using the random clock emulation with a precision of 4. The figure displays two channels (one from node 0 to node 1 and the other one back). The clock traces illustrate under which unfavorable clocking conditions our approach is still working. The largely varying phase relation between the clocks would definitely upset any traditional (phase-)synchronous system. At the same time many assumptions made for synchronizers in GALS systems would be invalidated as well.

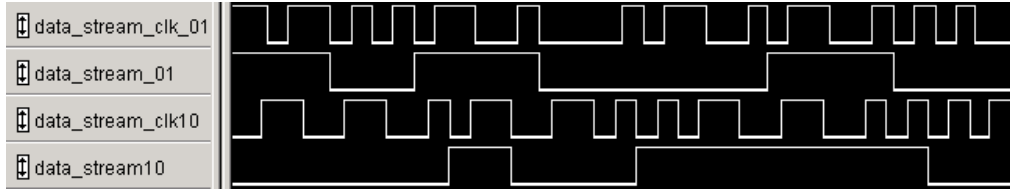


Figure 9.5: Example Communication within the Test System

9.5 Experiments

To verify our communication scheme in practice we have executed several experiments. In particular we want to use this system to support our claim that the buffer size derived in Chapter 8 (i) is sufficient for fault-free and metastability-free operation with clocks showing a precision of several clock cycles and (ii) represents a tight lower bound. Figure 9.6 illustrates the general layout.

9.5.1 Test System

The test system consists of 3 Xilinx Virtex-4 FPGAs and a host PC. One of the FPGAs acts as a global test controller. It coordinates the two local controllers and generates the required clock signals by virtue of a clock emulation (see Section 9.5.2). The other two FPGAs represent target nodes that exchange messages. These messages are randomly generated at the host PC and stored in both targets such that the receiver of a message can check its correctness. If an error is detected, communication stops until the test controller has re-initialized the test system.

9.5.2 Clock Emulation

To systematically investigate worst case scenarios and reproduce interesting effects, we need full control over the speed and the relative position of the targets' clocks. To this end we decided to use a clock emulation instead of the actual DARTS clock that would be much harder to control. Furthermore the communication scheme should be independent of DARTS and work with every other multisynchronous clocking system. This emulation is performed by the controller FPGA. The respective clock patterns are downloaded from the host PC where they have been a-priori calculated. In essence they are a sequence of integer multiples of a base clock period that determines the resolution of the clocking system.

In our experiments we have used the following two types of clock emulation:

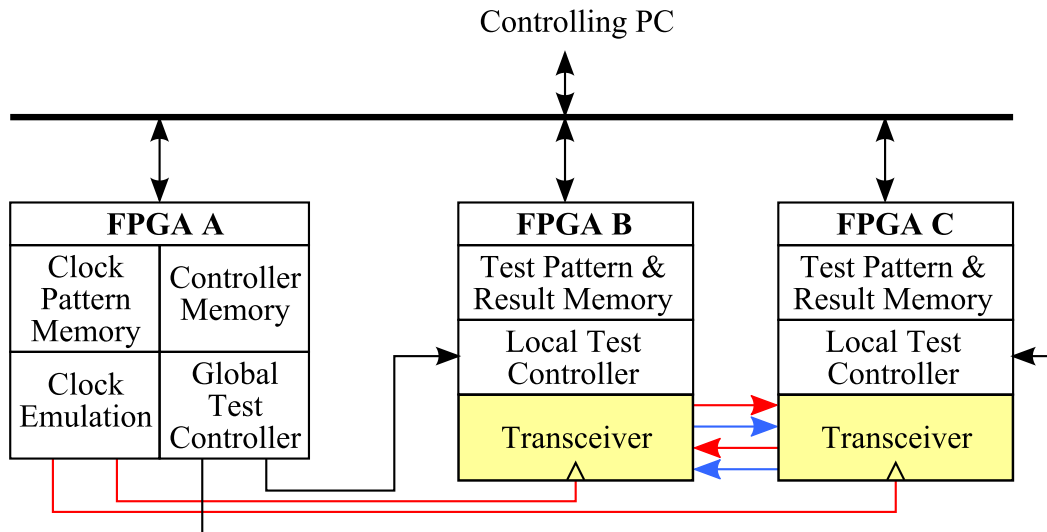


Figure 9.6: Layout of the Test System

Worst Case Precision Clock Emulation

Here the two clock signals are deliberately kept as far apart as the precision allows. This way we can check whether the system can indeed operate under such unfavorable conditions. At the same time chances are that the communication will fail if the buffer is too small, which gives an indication of whether the size calculated in Chapter 8 is a tight lower bound.

To achieve the worst case scenario we artificially stop one clock while the other one runs at its full frequency. As soon as the precision limit is reached, the stopped clock is speeded up to its full frequency again. An example is shown in Figure 9.7.

Random Clock Emulation

This emulation type is used to assess the performance of the system under continuously changing relative clock speed. Therefore clocks are varied over time. This is achieved by defining a set of “clock primitives” the host PC can use when planning the emulation. Table 9.3 lists the clock primitives we derived from the base clock.

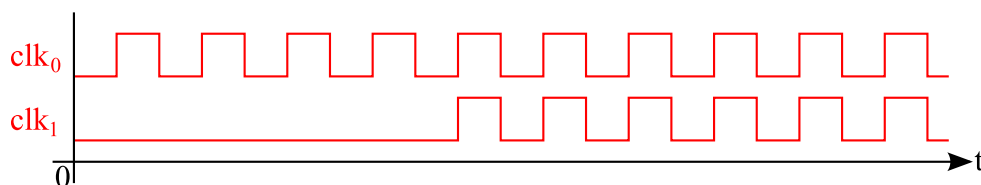

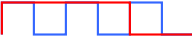




Figure 9.7: Worst Case Precision Emulation

Table 9.3: Clock Primitives

Clock Primitive	Base Clocks	Graphic
Fast Tick	1x High, 1x Low	
Normal Tick	2x High, 1x Low	
Slow Tick	2x High, 2x Low	
No Tick	3x Low	

These are then randomly assembled to two different sequences representing the full clock traces. During this construction process care is taken to keep the emulated clocks within the precision limits. An example for such a clock was already shown in Figure 9.5.

9.5.3 Test Conditions

Considering our measurements on the DARTS VLSI chip we decided to use a clocking system with a precision of 4. The emulation base clock (emulation resolution) was set to 48 MHz. This leads to a maximum clock frequency of 24 MHz for the “Random Clock Emulation” and a constant clock frequency of 24 MHz for the “Maximum Precision Clock Emulation”, as well.

The test system was implemented in a way that the equation for the buffer size (see Theorem 8.2.1) leads to a minimum. This is achieved by designing the system in a way such that the last term is becoming 0, while the middle term evaluates to 1. Therefore it is supposed that a buffer size of $2\pi + 1 = 9$ should be sufficient.

9.5.4 Performed Tests

The purpose of the following tests is to give practical evidence that the calculated buffer size is (i) sufficient and (ii) represents the minimum requirement. Since it is not possible to exhaustively emulate all possible relations between the clocks, we can not prove the absence of failures for buffer size 9. We can, however, check the failure-free operation under adverse conditions during some period. Thus we can substantiate our formal proof by this practical application. Furthermore, we will reduce the buffer size below the calculated limit. If the limit of 9 is tight, we can expect to observe failures for buffer sizes of 8 and less.

The test runs are executed 5000 times for each buffer size and each clock emulation type. For each run we calculate new clock traces in the emulation. If no failure is encountered after 2 seconds of observation time, the run is considered fault-free.

Table 9.4: Results of the Experiments

Buffer Size	Errors [%]		Min. Clock Cycles to First Error		⊗ Clock Cycles to First Error	
	Worst C.	Random	Worst C.	Random	Worst C.	Random
3	100	99	45	26	45	68
4	100	99	44	26	48	412
5	100	92	44	27	48	1600
6	100	35	26	29	46	4727
7	100	12	28	1	46	7718
8	19	0	27	-	61	-
9	0	0	-	-	-	-

After the experiments are finished, the minimum and mean times to the first error are extracted and an error percentage is calculated.

9.5.5 Results

Table 9.4 presents the collected test results. It is interesting to note, that the “Random Clock Emulation” did not produce any error in case of a buffer size of eight, the worst case emulation, however, did. This indicates that the failure probability is very low in this case, and becomes visible within limited observation time only if worst case conditions are artificially established. For smaller buffer sizes the expected failures could be observed without problems. The trend towards higher failure rate for smaller buffer size is visualized in Figure 9.8. In summary the results give a good confirmation of our theoretical findings.

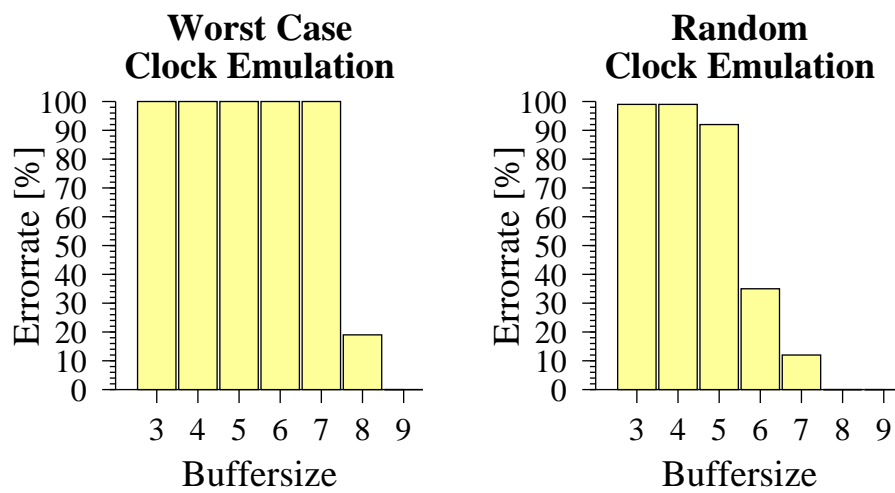


Figure 9.8: Results of the Experiments

Chapter 10

Implementing a Lockstep Model

When designing a hardware implementation of a lockstep synchronous algorithm, referenced as application algorithm in the text, one does not want to have to establish the round structure within the algorithm. Therefore a service indicating the round starts would be advantageous.

Our solution is to create a small circuit which signalizes the start of a round by a roundtick signal available to the application algorithm. This circuit is implemented using the synchronous paradigm based on the local microtick clock. It utilizes its knowledge on the timing of the communication layer to create rounds of sufficient length to guarantee a safe operation. It is ensured that each message sent by any node at the round start is safely delivered within the current round. All timeout values are solely based on the local microtick clock.

Figure 10.1 shows how this circuit is placed within the implementation hierarchy. The new roundtick signal as well as the microtick clock are available to the application algorithm.

In this chapter we will describe an algorithm creating a lockstep synchronous round structure and its mapping to the communication layer implementation, presented in Chapter 8. Additionally an example execution will be sketched.

10.1 Algorithmic Model

To be able to use the algorithm with different communication layer implementations, the required properties are abstracted into Assumption 10.1.1 - 10.1.4. We will later show that the communication layer presented in Chapter 8 respects all these assumptions.

The communication layer supports the sending of independent, fixed length messages. A message can only be sent at the beginning of a message slot. The first slot

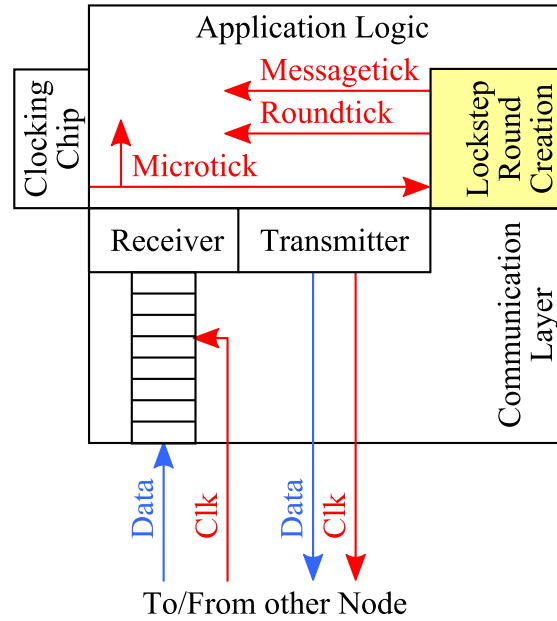


Figure 10.1: Lockstep Round Generation Implementation Hierarchy

starts directly after the startup of the communication layer is finished. Each message slot has an a-priori known length of t_{slot} microticks. The slots are aligned back to back.

Assumption 10.1.1 (Messages Slot Time). *The send operation of a message takes a constant, a-priori known time of t_{slot} microticks (exactly the length of a message slot), assuming that the send operation was aligned to the start of a message slot.*

Assumption 10.1.2 (Transmission Latency). *The transmission latency t_1 (measured in microticks) of a message is constant and known in advance.*

Assumption 10.1.3 (Message Count). *In each round a maximum number m of messages may be sent. The maximum number m is known in advance.*

Assumption 10.1.4 (Startup Time). *The message layer has a specific startup time t_{start} (measured in microticks). It is constant on all nodes and known a-priori.*

The implemented functionality is described using Algorithm 3.

Informal Description of the Algorithm

At startup, the node waits until the message layer is ready (waiting for t_{start} microticks). This guarantees the alignment of the first round start to the start of a message slot and therefore minimizes the length of the message send operation (as assumed in Assumption 10.1.1), because no additional wait cycles are needed before the message transmission starts.

Algorithm 3 Constant Message Count Algorithm

```

1: // Global Information
2: const  $t_{\text{start}}$  – Communication Layer Startup Time [microticks]
3: const  $m$  – Maximum Number of Messages per Round
4: const  $t_{\text{slot}}$  – Message Slot Time [microticks]
5: const  $t_l$  – Transmission Latency of a Message [microticks]
6: const  $t_{\text{calc}}$  – Maximum Calculation Time
7: const  $t_{\text{sync}}$  – Synchronization Time ( $\left\lceil \frac{t_{\text{calc}} + t_l - t_{\text{slot}} + 1}{t_{\text{slot}}} \right\rceil$ ) [microticks]
8: // Startup
9: wait  $t_{\text{slot}}$  microticks
10: // Create the Successive Rounds
11: for ever do
12:   Signal Round Start
13:   // Wait for Message Sending
14:   wait  $m \cdot t_{\text{slot}}$  microticks
15:   // Wait for the Message Arrival and the Calculation to Finish
16:   wait  $t_{\text{sync}}$  microticks
17: end for

```

After the startup phase is completed, the algorithm creates successive rounds. At the beginning of each round its start is signalized. For each round the algorithm waits until all messages are sent ($m \cdot t_{\text{slot}}$ microticks) and afterwards compensates for the message latency ($t_l - t_{\text{slot}} + 1$ microticks), waits for the algorithm to finish its execution (t_{calc}) and resynchronizes to the start of the next message slot. The resynchronization is achieved by rounding up the synchronization time to a multiple k of the message slot time (t_{slot}). The resulting synchronization time t_{sync} is therefore calculated as:

$$t_{\text{sync}} = t_{\text{slot}} \left\lceil \frac{t_l - t_{\text{slot}} + t_{\text{calc}} + 1}{t_{\text{slot}}} \right\rceil.$$

At this point, all messages are surely delivered and the calculation is finished. Therefore the round execution is completed and the next round is started. The round generation is repeated indefinitely.

The execution of the algorithm for a message slot time t_{slot} of 4 microticks and a synchronization time t_{sync} of the same value is shown in Figure 10.2.

10.2 Correctness Proof

For Algorithm 3 to be correct, it must ensure that all m messages are delivered within the same round, compensate for the message latency and the calculation time of the application algorithm and align the round starts to the beginning of a message

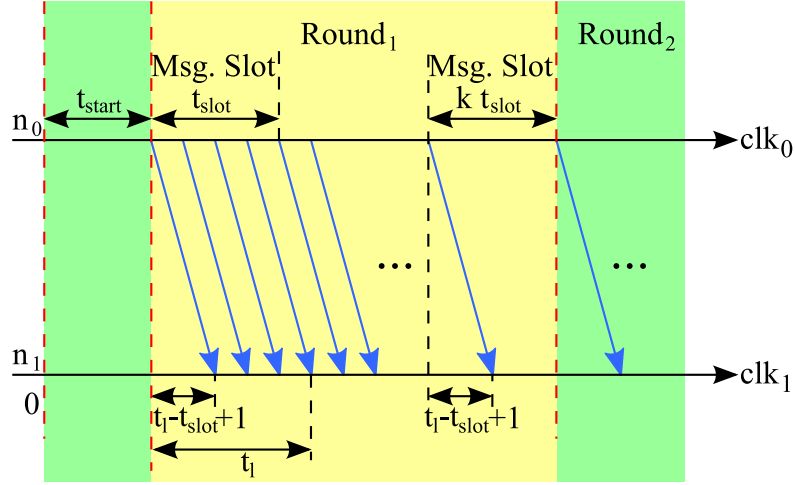


Figure 10.2: Visualization of the Assumptions

slot. It is obvious that it takes $m \cdot t_{\text{slot}}$ microticks to send m messages, if the send operation of the first message was aligned to a message slot. Therefore the message send operation of the rounds's last message starts $(m - 1) \cdot t_{\text{slot}}$ microticks after the round start. By Assumption 10.1.2 it takes exactly t_1 microticks to deliver it. Therefore it is delivered no later than $m \cdot t_{\text{slot}} + t_1 - t_{\text{slot}} + 1$ microticks after the round start. The term $m \cdot t_{\text{slot}}$ is equivalent to the first delay loop of Algorithm 3. The other terms are part of the synchronization time. By the definition of t_{sync} , the message bit latency ($t_1 - t_{\text{slot}} + 1$) and the execution time (t_{calc}) of the application algorithm are safely compensated.

We have shown that the round length is sufficient to deliver all messages and for the execution of the application algorithm. The last thing to prove is that the algorithm aligns the round start to the beginning of a message slot.

Lemma 10.2.1. *All rounds are aligned to the beginning of a message slot.*

Proof. This Lemma is proved by induction.

- Induction start (Round 1): By Assumption 10.1.4, the first round is aligned to the beginning of a message slot.
- Induction hypothesis: Assume the Lemma to hold for round n .
- Induction step ($n \rightarrow n + 1$): The start of round n was aligned to the start of a message slot. By sending all m messages of round n , the alignment is not changed (because each message send operation has the length of a message slot). The length of the synchronization loop (t_{sync}) is also a multiple of the message slot time. Since the next round starts directly after this time

has elapsed, round $n + 1$ must obviously start at the beginning of a message slot. \square

10.3 Time Complexity

The execution time of a lockstep synchronous algorithm is always given in rounds (t_{round}). In our model, however, the basic unit of time is a microtick and not a round. Therefore it would be advantageous to know the execution time of an implemented algorithm measured in microticks (t_{micro}).

Knowing the round execution time t_{round} of an algorithm, t_{micro} can easily be calculated as:

$$t_{\text{micro}} = t_{\text{round}} (m + t'_{\text{sync}}) t_{\text{slot}},$$

where $t'_{\text{sync}} = \left\lceil \frac{t_l - t_{\text{slot}} + t_{\text{calc}} + 1}{t_{\text{slot}}} \right\rceil$ is the synchronization time measured in message slots. Because the message latency t_l , the message slot time t_{slot} and the calculation time t_{calc} , and therefore the synchronization time, are normally constant ($O(1)$), the time complexity (in O -notation) in such a case can be calculated as:

$$t_{\text{micro}} = t_{\text{round}} m t_{\text{slot}}.$$

For example, having an algorithm with a time complexity t_{round} of $O(n)$ and a round message complexity m of $O(n)$, the microtick based time complexity t_{micro} will evaluate to $O(n^2)$.

A more problematic example is an algorithm with a round time complexity t_{round} of $O(n)$ and an exponential message complexity. Its microtick based time complexity t_{micro} would be exponential! This fact is often neglected in theoretical distributed algorithm literature. This is due to the fact that the round length is proportional to the number of messages sent per round.

10.4 Mapping the Message Layer Implementation

After describing the algorithmic model in detail, it must be mapped to the previously implemented communication layer (see Chapter 8). The layer supports eight bit messages with a slot time of 10 microticks. As already discussed in Section 9.3.2, it has a constant message latency. The startup time is 2 microticks. Therefore all assumptions made by the algorithmic model hold for this implementation.

10.5 Static Round Pattern Algorithm

Some important lock step synchronous algorithms have a static round pattern, which means that their execution consists of a predefined sequence R of rounds which is repeated indefinitely. Each round in R has a defined, a-priory known maximum message count m_i . Using R , Assumption 10.1.3 can be redefined as:

Assumption 10.5.1 (Message Count). *The execution is structured into sequences of n rounds $R_i, i = 1 \dots n$. The maximum number of messages sent in each round R_i is defined as m_i and known in advance. The values m_i are stored using an n -tuple $R = [m_1, m_2, \dots, m_n]$.*

Using Assumption 10.5.1, Algorithm 3 can be redefined to Algorithm 4.

Algorithm 4 Static Round Pattern Algorithm

```

1: const  $t_{\text{start}}$  – Communication Layer Startup Time [microticks]
2: const  $R$  –  $n$ -tuple Containing the Message Pattern
3: const  $t_{\text{slot}}$  – Message Slot Time [microticks]
4: const  $t_l$  – Transmission Latency of a Message [microticks]
5: const  $t_{\text{calc}}$  – Maximum Calculation Time
6: const  $t_{\text{sync}}$  – Synchronization Time ( $\lceil \frac{t_{\text{calc}} + t_l - t_{\text{slot}} + 1}{t_{\text{slot}}} \rceil t_{\text{slot}}$ ) [microticks]
7: // Startup
8: wait  $t_{\text{start}}$  microticks
9: // Repeat the Algorithm Round Pattern for Ever
10: for ever do
11:   // Execute All Rounds of the Round Pattern
12:   for all  $m_i \in R$  do
13:     Signal Start of Round  $R_i$ 
14:     // Wait for Message Sending
15:     wait  $m_i t_{\text{slot}}$  microticks
16:     // Wait for the Message Arrival and the Calculation to Finish
17:     wait  $t_{\text{sync}}$  microticks
18:   end for
19: end for

```

The execution of Algorithm 4 is similar to the execution of Algorithm 3 with the difference that the number of messages sent in each round is not constant. For algorithms with a static round pattern the number of idle message slots can therefore be decreased, optimizing the microtick complexity of the algorithm.

Algorithm 5 Round Pattern Creation Algorithm including the Signalization of the Message Arrival Times

```

1: const  $t_{\text{start}}$  – Communication Layer Startup Time [microticks]
2: const  $R$  –  $n$ -tuple Containing the Message Pattern
3: const  $t_{\text{slot}}$  – Message Slot Time [microticks]
4: const  $t_l$  – Transmission Latency of a Message [microticks]
5: const  $t_{\text{calc}}$  – Maximum Calculation Time
6: const  $t_{\text{sync}}$  – Synchronization Time ( $\left\lceil \frac{t_{\text{calc}} + t_l - t_{\text{slot}} + 1}{t_{\text{slot}}} \right\rceil t_{\text{slot}}$ ) [microticks]
7: // Startup
8: wait  $t_{\text{start}}$  microticks
9: // Repeat the Algorithm Round Pattern for Ever
10: for ever do
11:   // Execute All Rounds of the Round Pattern
12:   for all  $m_i \in R$  do
13:     Signal Start of Round  $R_i$ 
14:     // Wait for the First Message to Arrive (Compensate for the Message Latency)
15:     wait  $t_l$  microticks
16:     Signalize Message Reception
17:     // Wait for all other Messages to Arrive ( $m_i - 1$  Messages)
18:     for  $i \in [0, \dots, (m_i - 2)]$  do
19:       wait  $t_{\text{slot}}$  microticks
20:       Signalize Message Reception
21:     end for
22:     // Realign the Execution to the next Message Slot
23:     wait  $t_{\text{sync}} - (t_l - t_{\text{slot}} + 1)$  microticks
24:   end for
25: end for

```

10.6 Message Receive Event

Additionally to establishing the round pattern, the exact times for reading the messages from other nodes must be signalized. In non faulty systems this can be achieved by using the signalization mechanism of the receiver. If faults may occur, it could happen that some messages get lost and the timing information of non faulty nodes may be compromised.

Therefore the message reception pattern must be established relying on local information only. Algorithm 5 establishes such a pattern based on the deterministic round definition already shown in Figure 10.2.

Informal Description of the Algorithm: The algorithm is a slightly changed version of the round creation algorithm presented as Algorithm 4. The compensation for the message latency is divided into a portion at the start of the algorithm (aligning the execution to the message reception times) and another at its end (realigning

to the next message slot), otherwise it is equivalent.

10.7 Hardware Implementation

As Algorithm 3 can be seen as a special case of Algorithm 4 ($R = [m]$), only Algorithm 4 has to be implemented in hardware. Additionally Algorithm 5 is an improved version, therefore only this algorithm was implemented.

It is implemented using a state machine for creating the round pattern. The wait instructions are realized by counters. The round start- and the message receive events are signaled by the state machine using the *roundtick* and the *messagetick* signals. Additionally two signals specifying the number of cycles left before the next round switch and message reception, respectively, are available.

An example simulation of the VHDL implementation can be found in Figure 10.3. It is based on a round definition $R = [2, 4, 1]$ and a startup time of 2 microticks. The startup of the system (*roundtick* = 0) as well as the signaling of the message receive-events (transition on signal *messagetick*) and the signaling of round starts (transition on signal *roundtick*) are shown.

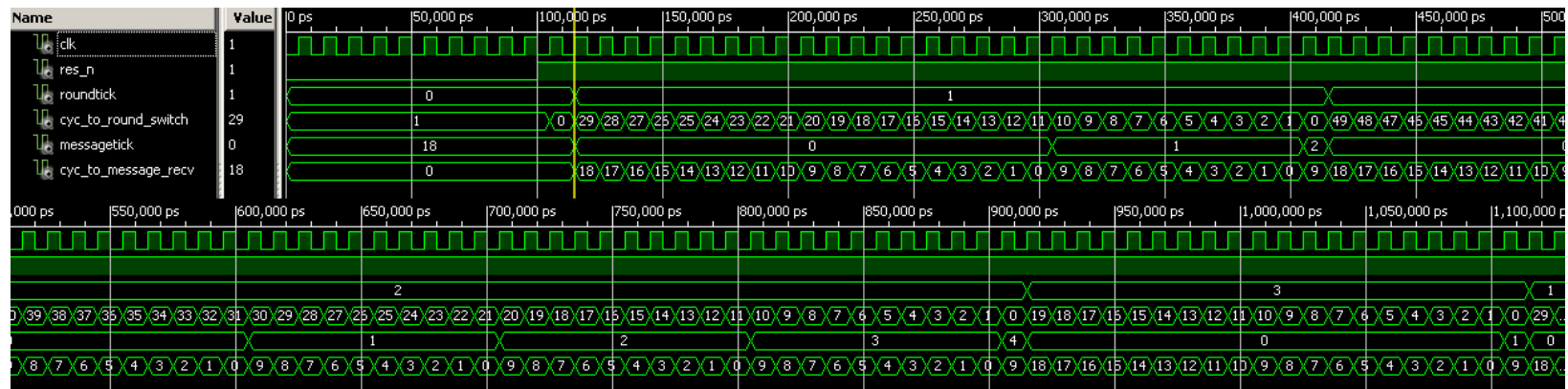


Figure 10.3: Simulation of the Lockstep Algorithm Implementation

Part III

Consensus

Chapter 11

Problem Definition

After establishing a reliable, but not fault tolerant framework for implementing lock-step synchronous algorithms, we will proceed by developing a fault tolerance layer for our system.

As we have already discussed, TMR systems (Section 5.2.1) are widely used and state of the art. Therefore it would be tempting to adapt such a system to our multisynchronous model. Unfortunately TMR systems have several drawbacks, which we will discuss in this chapter. Additionally we will see that digital electronic circuits can easily behave non-benign, a situation a TMR system may be incapable to handle.

11.1 Problems of TMR Systems

The simplest way to implement a TMR system is to use the globally synchronous paradigm. All application logic components use the same clock, while the toolset compensates for the delays introduced by the asynchronous voter(s). This also guarantees the correct alignment of the voter's input values (correct temporal relation).

Unfortunately the global clock signal is a single point of failure, contradicting the fault tolerance of the system. To circumvent this problem one may use a GALS system with its independent, unsynchronized clock sources. As the voter is asynchronous, obviously no metastability problems will occur at its inputs. Metastability problems may nevertheless arise, if the voter output is fed into another sequential circuit. Since these two circuits have different clock sources with an arbitrary phase shift, the input stage of the following synchronous logic may behave metastable.

Another problem, when using multiple unsynchronized clock sources or a multisynchronous clocking scheme is that the temporal relation between the input values may get lost.

Table 11.1: Voter Execution

Event	Output Value	Global Time	Local Time
Initially all Values are 0	0	@0	
Value of Node 2 = 1	0	@1	@2.1
Value of Node 0 = 1	1	@2	@0.1
Value of Node 2 = 0	0	@2	@2.2
Value of Node 0 = 0	0	@3	@0.2
Value of Node 1 = 1	0	@3	@1.1
Value of Node 1 = 0	0	@4	@1.2

To show this, we will create a fault-free multisynchronous execution within a basic TMR system leading to an illegal output value, even if voting is executed on a single bit signal only and the modules are replica deterministic.

Let us call the single instances of the replicated application logic node 0 - node 2. The clock of node 0 is one clock cycle behind realtime, the one of node 1 two cycles behind realtime, while the clock of node 2 runs synchronously with realtime. Therefore this execution is valid for any multisynchronous system with a precision π of at least 2.

It is also valid for GALS systems. Based on the inevitable drift of the independent clock sources, after a sufficient execution time such a constellation may arise.

Since the execution is fault free, all nodes will output the correct value at their corresponding local clock edge. Due to the purely asynchronous implementation (see [Sho02] as example), the voter is not aware of any shifts in the data signals. The detailed execution can be found in Table 11.1 and is visualized in Figure 11.1.

As apparent from the execution, the time shift of the nodes is enough to compromise the output. Since the majority value is always 0, the output is never set to one or worse only a spike is created. The length of the spike depends on the skew between

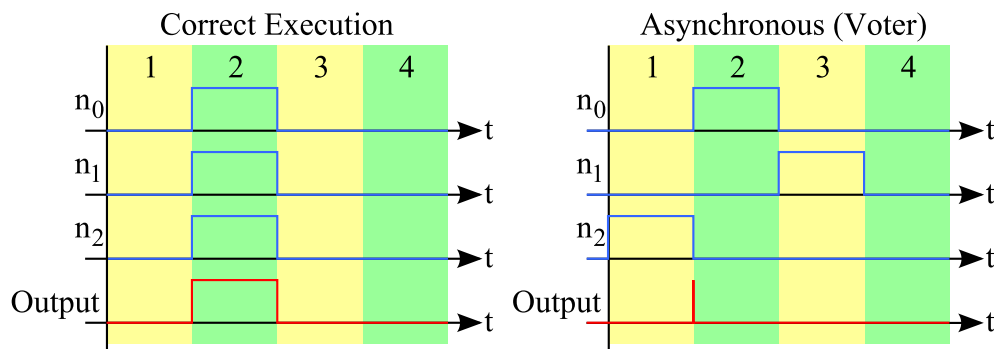


Figure 11.1: Unsynchronized Voter Execution

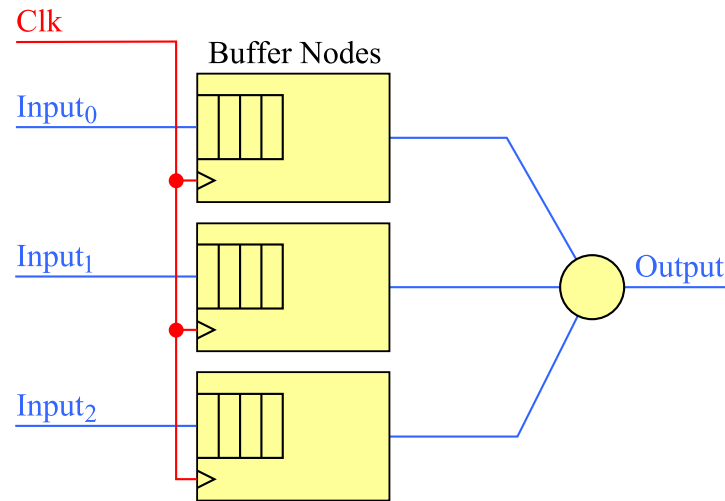


Figure 11.2: Multisynchronous Voter Implementation

the data signals. Depending on which data link is faster, spikes may be created.

Comparing the result to the correct one, shown in Figure 11.1, it is obvious that the output signal of the voter is wrong.

This problem even may occur, if the voter is the last element of a fully synchronous circuit and the skew between the data signals is too large. Due to the asynchronous nature of the voter circuit and the fact that no sequential element follows, the skew of the voter input signals is not automatically controlled by the synchronous toolset.

Additionally, for a voter deciding on multi-bit values, a fault free execution, having no majority values at all, can be sketched. This is achieved by using the local clock tick as output value of the application logic while using the same clock shift as in the previous example. Due to the shifted clock index, at any point in time each value will be present at most on a single voter input.

Therefore extreme care must be taken when designing a system with an asynchronous voter (e.g. control skew, usage of a clock synchronization algorithm).

The only way to implement a sufficiently dependable voter circuit in a multisynchronous environment would be to artificially synchronize the input values. This can be done by adding a buffer node for each input value. All of these nodes must have the same clock signal (out of the multisynchronous ensemble) and use the previously presented communication layer (see Chapter 8) to receive the data and synchronously output them (see Figure 11.2). As all these nodes use the same clock, the clock signal would be a single point of failure in the system. Nevertheless this drawback can be circumvented, if the voter is also replicated (as described in [LV62], see Figure 11.3).

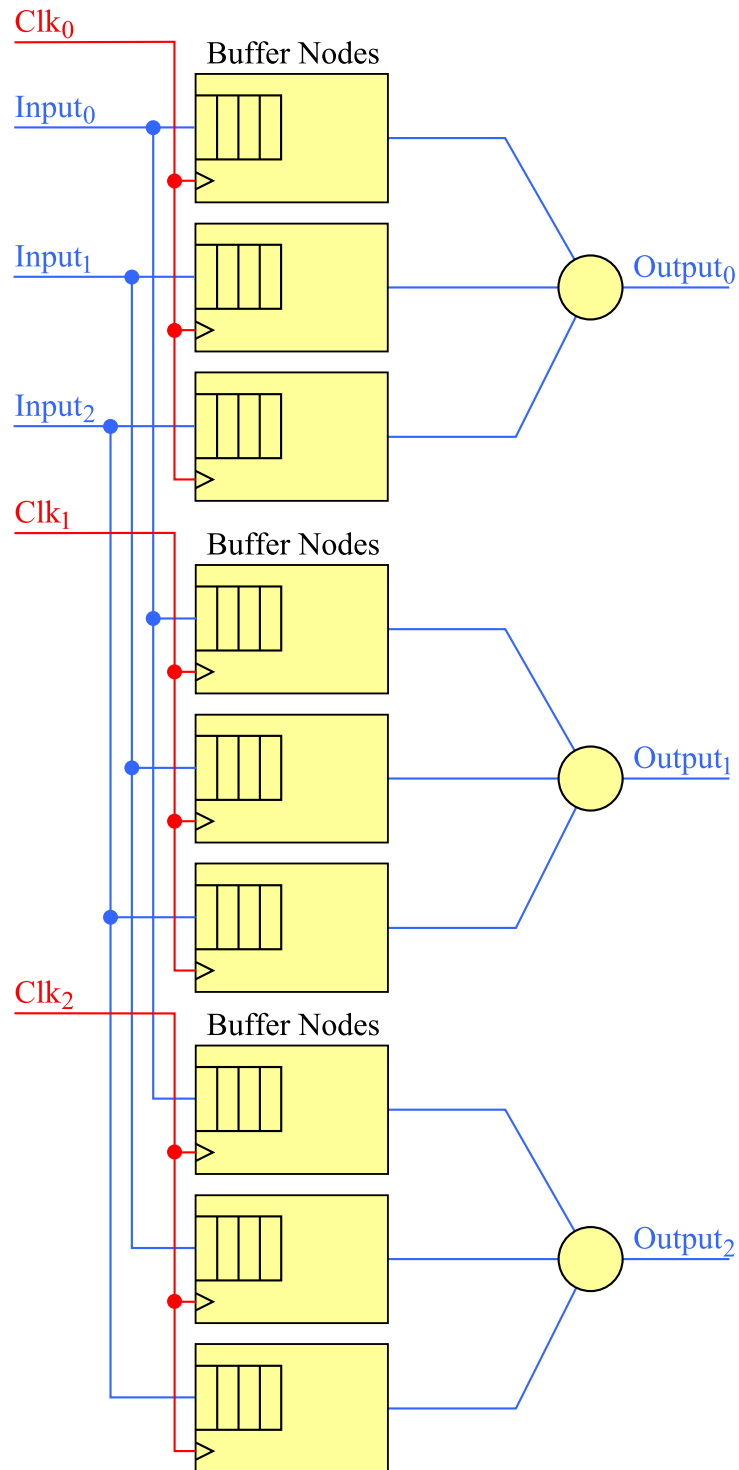


Figure 11.3: System with Replicated Multisynchronous Voters

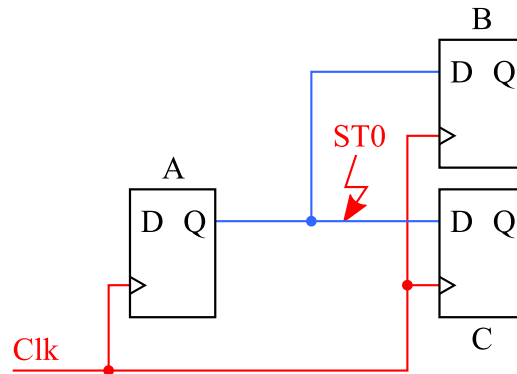


Figure 11.4: Circuit used in the Proof

11.2 May Hardware Act Non-Benign?

We will now show, why we have to assume non-benign behavior of electronic circuits. This is done by analyzing a simple circuit experiencing a single fault described by one of the presented fault models (see Section 5.1). Since all faults defined by any of the fault models may occur in practice, it is sufficient to show that the circuit behaves non-benign in at least one of the fault models.

Assumption 11.2.1 (Non-Trivial Electronic Circuit). *As non-trivial electronic circuit we define a circuit with at least one fork.*

Assumption 11.2.2 (Link Failures). *In our model, faults occurring on the communication links are handled as if they are happening at the sending node.*

Based on Assumption 11.2.2, a communication error is modeled by changing the executed algorithm of the sending node.

Lemma 11.2.1. *Every non-trivial electronic circuit can produce non-benign faults.*

Proof. We prove the Lemma for synchronous circuits. The proof for asynchronous circuits is equivalent.

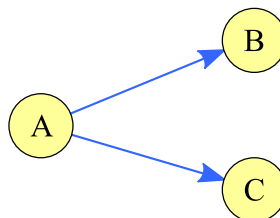


Figure 11.5: Distributed System Model of the Circuit

Algorithm 6 Algorithm of Node A in the Fault-Free Case

```

1: var  $val$  - The node's local value (either 0 or 1)
2: on active edge of  $clk$  do
3:   Send  $\langle val \rangle$  to B
4:   Send  $\langle val \rangle$  to C
5: end on

```

Algorithm 7 Algorithm of Node A in the Faulty Case

```

1: var  $val$  - The node's local value (either 0 or 1)
2: on active edge of  $clk$  do
3:   Send  $\langle val \rangle$  to B
4:   Send  $\langle 0 \rangle$  to C
5: end on

```

The lemma is proved by contradiction. Assume only benign faults can occur in a synchronous circuit.

We use the circuit shown in Figure 11.4. Based on the distributed systems theory, it can be modeled as fully synchronous algorithm. The resulting system can be found in Figure 11.5, while its functionality is described by Algorithm 6.

Assume that the input of Flip-Flop C is subjected to a stuck-at-zero fault or a transient fault setting the input of Flip-Flop C long enough to 0 to get captured. Therefore the executed algorithm is changed to Algorithm 7.

As we can easily see, if node A's value is 1, node B would receive a message containing the value 1, while node C receives a message containing the value 0, which is obviously a non-benign behavior, since the messages do not match. Therefore our assumption that the circuit can only produce benign faults is contradicted and the Lemma is correct. \square

The proof has revealed that every non-trivial electronic circuit may behave non-benign. We will formulate this finding fact as a theorem.

Theorem 11.2.1 (Non-Benign Hardware). *When modeling a non fault-free non-trivial electronic circuit using the distributed systems theory the hardware may act non-benign and therefore the Byzantine failure model is assumed.*

As Theorem 11.2.1 describes, even the most simplistic circuits, as long as they have at least one fork, can experience non-benign faults.

11.3 TMR Systems in Byzantine Environments

As stated in [Kop97], one needs at least $3f + 1$ replications of the application logic to tolerate Byzantine faults in electronic circuits (which is equivalent to the lower

bound of nodes needed for a consensus algorithm in a distributed system, as stated in [AW04]). Since a TMR system consists of three replicas, it is definitely incapable of handling Byzantine faults.

To clarify this fact, consider the following scenario: Depending on its input value, the system decides whether to set a railway signal to halt or free. Node 0 decides that the signal must show halt, while node 1, due to slightly different sampling of the input value, decides to free. Node 2 on the other hand is faulty and transmits the value stop to node 0's voter and the value free to node 1's voter. On both non faulty systems, a majority value is found. Node 0 will decide to stop, while node 1 decides to free resulting in an inconsistent decision. Therefore the system is in an undefined state.

As the example reveals, the decision circuit must either be able to handle Byzantine faults correctly or the nodes must be replica deterministic. Replica deterministic means that non faulty nodes always decide to the same value as they receive the same inputs. In this case the output values of the correct nodes surely form a majority and a faulty node can no longer effect the output value.

Especially when using processors with embedded operating systems, replica determinism is very hard to achieve. Therefore circuits tolerating Byzantine faults would be advantageous.

11.4 Alternatives to TMR Systems

As Theorem 11.2.1 states, we assume a byzantine fault model for electronic circuits. Since TMR systems are incapable of handling such faults in the general case, their fault tolerance is only limited.

Nevertheless a lot of modern solutions are based on the TMR paradigm (like [OKS08], e.g.).

Other approaches, as consensus algorithms, are mainly implemented at software level (like [HLD95], e.g.). Circumventing most problems of TMR systems, these implementations suffer from a low performance, due to the lack of direct hardware interaction and therefore increased communication latency.

To overcome these problems, a completely different strategy must be pursued. Since we have already implemented a lockstep synchronous round model in hardware, the adaptation of a lockstep synchronous consensus algorithm for implementation in hardware is obvious. As for the direct hardware implementation, the communication latency will be decreased and the algorithm will be implemented with a much higher performance. On the other hand we will be able to handle, in contrast to TMR systems, Byzantine faults in the general case, creating a much higher degree of fault

tolerance. The system even decides consistently, if the input values of the non-faulty nodes are different, as may happen in non replica deterministic systems.

In the next chapter, we will design a hardware implementation of the EIG consensus algorithm and analyze its time complexity. It is followed by the description of the hardware implementation.

Chapter 12

Creating a Byzantine Fault Tolerant System

Inspired by the TMR system with replicated voters (see Section 5.2.1), we designed a fault tolerant system based on a consensus circuit. The voters of the TMR system are replaced by the consensus implementation. Since for the consensus algorithm at least $3f + 1$ nodes are necessary [AW04], a fourth application logic replica was added. Such a system can tolerate a single Byzantine fault. A conceptual drawing can be found in Figure 12.1.

As apparent from the figure, the system uses multiple clock sources out of a multisynchronous ensemble. A suitable implementation would be our DARTS clocking scheme [FFSK06]. Due to the multisynchronous nature of the clocking system, the communication between the different clock domains can be implemented metastability-free by means of the communication layer developed in Chapter 8.

As a suitable consensus algorithm for hardware implementation, using a minimum number of nodes, we identified the EIG algorithm.

Our basic solution tolerates one Byzantine fault. If a higher degree of fault tolerance is needed, the consensus implementation must be created using a stronger fault hypothesis (e.g. $f = 2$ for tolerating two Byzantine faults). According to the lower bound proof (see [AW04]) $3f + 1$ replicas of the application logic would be needed. For the example of two Byzantine faults ($f = 2$) 7 replicas are necessary and sufficient.

12.1 The EIG Algorithm

Before designing the hardware implementation of the consensus algorithm, let us recapitulate the definition of the Byzantine EIG algorithm (see Section 2.4.2).

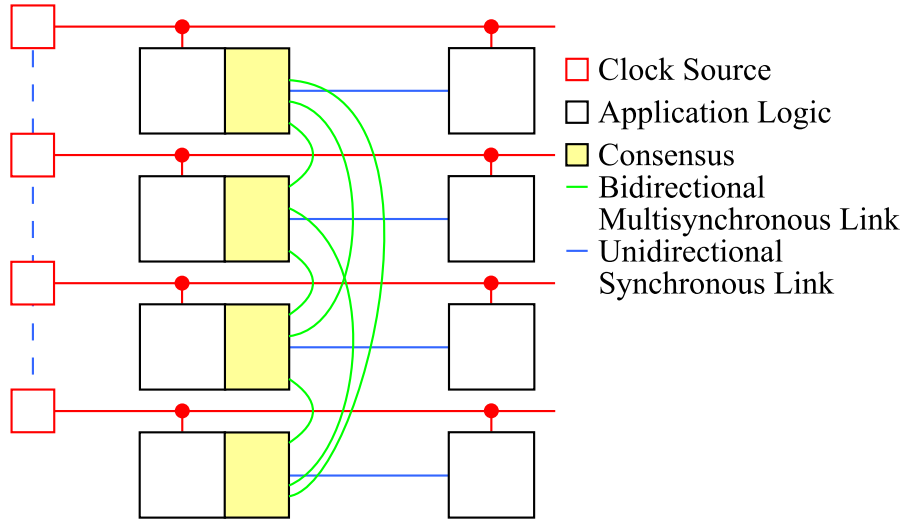


Figure 12.1: System Model Tolerating one Byzantine Fault

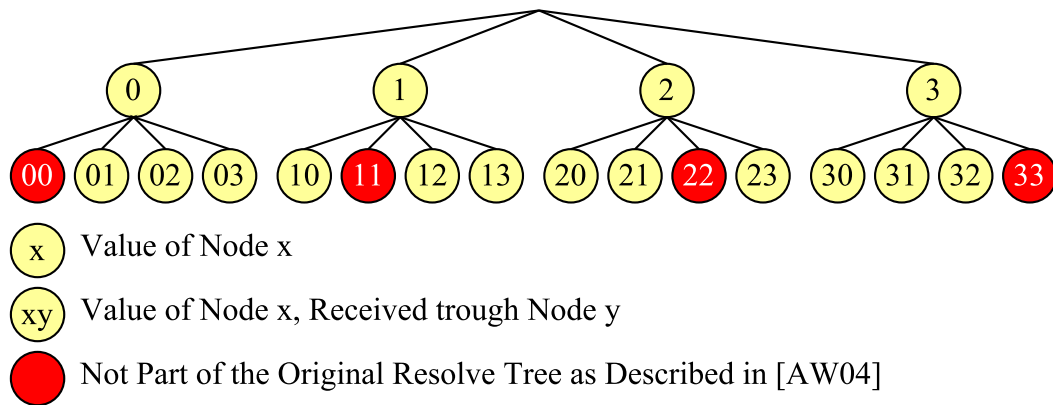
The basic element of the algorithm is the resolve tree. It is used to store all data collected on the nodes of the system. To identify the stored values and their history, each tree-node is labeled using the indices of all nodes which have already processed the value.

The algorithm consists of two phases, namely an information gathering phase and an output value resolve phase. The first phase consists of $f + 1$ rounds. In the first round each node j broadcasts its input value v_j . These values are stored by each node in its resolve tree on the first level. If a node does not send a value, the receiver node uses a default value instead. The label assigned to a stored item v_j is $\langle j \rangle$, the index of its originating node.

In the remaining rounds $(2, \dots, f + 1)$, the tree level built in the previous round is broadcasted. These levels are again stored in the resolve tree of the nodes. If a tree level is not received, a default value is used instead. The label assigned to a stored item is calculated as follows: If the received item had a label of $\langle n_0, \dots, n_d \rangle$ at the sending node, the new label is set to $\langle n_0, \dots, n_d, j \rangle$, where j is the index of the node, the item was sent by. Each label is checked whether it already contains the index j . If so, the item is discarded. Therefore only items processed at most once by each node are kept.

Figure 12.2 contains an example of a resolve tree for a system tolerating one Byzantine fault ($f = 1$). Note that the red tree-nodes are filtered out of the resolve tree.

After finishing the creation of the resolve tree, the second phase starts. Here a resolve function is executed on the resolve tree. Its leaves are grouped such that all items with the same index prefix (the label of the leaves without the last index, e.g. $\langle n_0, \dots, n_{d-1} \rangle$, if the label was $\langle n_0, \dots, n_{d-1}, n_d \rangle$) are in the same group. On each

Figure 12.2: Fully Created Resolve Tree for $f = 1$

group a majority vote is executed. If no majority exists, a default value is used. The results are labeled with the prefix used to create the group. They are again grouped, according to the same rule as above, and subjected to a majority vote. This operation is repeated until only a single value, the output value, remains.

The correctness, time- and message-complexity of the algorithm was already proved in [AW04]. The proof is not repeated here.

12.2 Algorithmic Model

We will now modify the Byzantine EIG algorithm to be suitable for a hardware implementation. The algorithm definition is based on the lockstep generation scheme presented in Chapter 10 and is timed using the *roundtick* and *messagetick* signals. Based on the hardware's property to execute multiple tasks in parallel, we can implement several parts of the algorithm independently. Therefore the algorithm is split into four parts, namely:

- A storage definition, containing the interface data between the different parts of the algorithm
- A component broadcasting the current tree level
- A component building the next tree level
- A component calculating the resolve value

Figure 12.3 contains a schematic visualization of the different parts of the algorithm. For a safe operation, a stable interface must be defined. It is important that the input values of the algorithm parts only change at appropriate points in time.

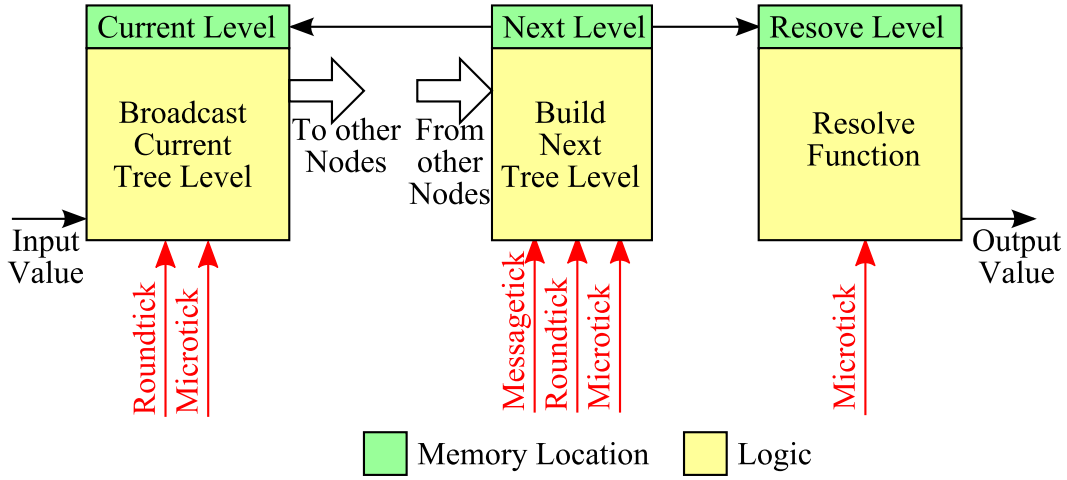


Figure 12.3: Structure of the Hardware Implementation

As will be discussed in Section 12.3, only the current level of the resolve tree is stored by the implementation (*current_level*). Since it is the input for the sender algorithm, this information must stay unchanged for the duration of the broadcast operation.

Parallel to the broadcast operation the next tree level is built. Writing the new level directly to the tuple, would compromise it. Therefore a separate instance (*receive_level*) is used to store these values. Based on the deterministic definition of the round structure and its static message delivery pattern (see Chapter 10), a safe point for overwriting the current level can be stated. It is ensured that after the last value of a tree level was received, the broadcast operation of the current level has safely finished and the tuple containing it can be overwritten (Line 18 of Algorithm 10).

By implementing the resolve function as independent circuit, running parallel to the next instance of the resolve tree build operation, it is required that its input value (*resolve_level*) stays unchanged for the duration of this operation. Therefore it may only be overwritten in an atomic action at the end of the tree build operation (Line 21 of Algorithm 10).

Algorithm 8 contains the definition of the globally accessible storage values, modeling the interface between the different algorithm parts. The size requirements of the tuples will be proved in the Section 12.3. Algorithm 9 describes the broadcast operation of the current tree level, while Algorithm 10 describes how the next level is created. The resolve function will be described later.

12.3 Building the Resolve Tree

In contrast to the definition of the EIG algorithm in Section 2.4.2, the hardware implementation builds the full resolve tree and therefore supports the filtering of the labels containing duplicate indices without storing them explicitly (see Lemma 12.3.3). The drawback of this design is that the runtime of the algorithm is slightly increased. Nevertheless the time complexity stays unchanged. An example for a resolve tree with a fault hypothesis of one Byzantine fault ($f = 1$) was already shown in Figure 12.2. The full tree consists of all tree-nodes, including the red ones.

As already mentioned, the tree level is stored using a tuple. The size of this tuple will be calculated in the following. We start with determining the size of the tree levels.

Lemma 12.3.1. *The tree level of depth d is calculated in round d of the algorithm and has a size of $(3f + 1)^d$.*

Proof. The Lemma is proved by induction.

- Induction Start ($d = 1$): The tree level of depth 1 is created in the first round. It is composed of exactly one value for each node in the system (the broadcasted input values). Since there are $3f + 1$ nodes in the system, the size of the first tree level is $3f + 1$.
- Induction Hypothesis: Assume the correctness of the Lemma for d .
- Induction Step ($d \rightarrow d + 1$): In round $d + 1$, the previous level is broadcasted (Line 11 of Algorithm 9) by each node $j, j = 0, \dots, 3f$. Therefore the node receives one copy of the last tree level from each node (including itself, Line 14 of Algorithm 10). Since there are $3f + 1$ nodes in the system, $3f + 1$ copies are received. The size of the tree levels of depth d is, by the induction hypothesis, $(3f + 1)^d$. This leads to a size of $(3f + 1)(3f + 1)^d = (3f + 1)^{d+1}$ for the tree level of depth $d + 1$.

□

Lemma 12.3.2. *The maximum size of a tree level is $(3f + 1)^{f+1}$.*

Algorithm 8 Consensus – Storage Definition

```

1: // Current Level of the Resolve Tree
2: var current_level as  $(3f + 1)^{f+1}$ -Tuple
3: // Buffer for Storing the Received Values while Building a new Tree level
4: var receive_level as  $(3f + 1)^{f+1}$ -Tuple
5: // Input Value of the Resolve Function
6: var resolve_level as  $(3f + 1)^{f+1}$ -Tuple

```

Algorithm 9 Message Sender Implementation

```

1: for ever do
2:   // Wait for Round Start
3:   wait for roundtick = 1
4:   // Broadcast the Input Value
5:   Broadcast  $\langle input \rangle$ 
6:   for all  $round \in [2, \dots, f + 1]$  do
7:     // Wait for Round Start
8:     wait for roundtick = round
9:     // Broadcast the Current Tree Level
10:    for all  $k \in [0, \dots, (3f + 1)^{round-1} - 1]$  do
11:      Broadcast  $\langle current\_level[k] \rangle$ 
12:    end for
13:  end for
14: end for

```

Algorithm 10 Message Receiver Implementation

```

1: for ever do
2:   // Execute  $f + 1$  Rounds
3:   for all  $round \in [1, \dots, f + 1]$  do
4:     // Wait for Round Start
5:     wait for roundtick = round
6:     // Receive all Messages of the Current Level
7:     for all  $k \in [0, \dots, (3f + 1)^{round-1} - 1]$  do
8:       // Wait for Message Reception
9:       wait for messagetick
10:      // Retrieve the Current Messages of all Nodes
11:      for all  $j \in 0, \dots, f$  do
12:        Read  $\langle val \rangle$  from  $n_j$ 
13:        // Store the Value in the Current Level
14:         $receive\_level[k(3f + 1) + j] := val$ 
15:      end for
16:    end for
17:    // Update the Current Tree level
18:     $current\_level := receive\_level$ 
19:  end for
20:  // Update the Input Value of the Resolve Function
21:   $resolve\_level := receive\_level$ 
22: end for

```

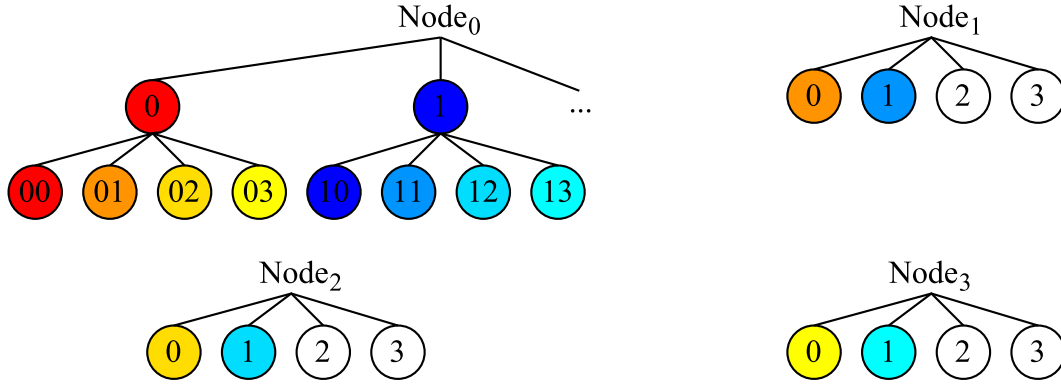


Figure 12.4: Building of a Resolve Tree Level

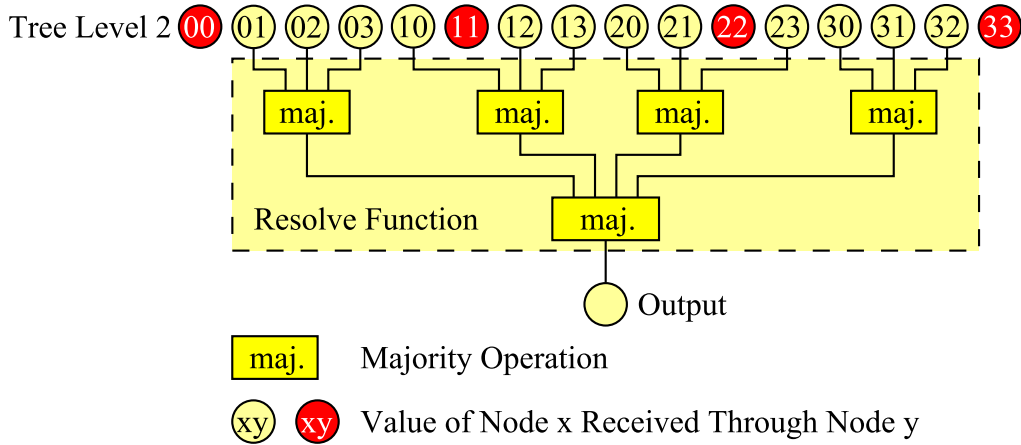
Proof. As the execution needs $f + 1$ rounds to finish [AW04], by Lemma 12.3 a tree of depth $f + 1$ is built and has a size of $(3f + 1)^{f+1}$. Since the size of the tree level is monotonically increasing with the depth of the tree, this is the largest possible tree level built in any execution. \square

After showing that the size of the tuples defined in Algorithm 8 is sufficient, we will prove that the resolve tree labeling of the original algorithm is consistent with the index calculation, the implicit labeling, of our hardware implementation (except for the non filtered tree-nodes). An example of the tree build operation and the labeling of the tree-nodes can be found in Figure 12.4.

Lemma 12.3.3. *A tree-node of depth d labeled as $\langle n_0, \dots, n_{d-1}, n_d \rangle$, where $\langle n_0, \dots, n_{d-1} \rangle$ is the label of its parent tree-node, is stored by the hardware implementation at the tuple position $n_0(3f + 1)^d + n_1(3f + 1)^{d-1} + \dots + n_d$.*

Proof. The Lemma is proved by induction.

- **Induction Start ($d = 1$):** In this round a single value is received from each node $j, j = 0, \dots, 3f$ (including itself) and is labeled as $\langle j \rangle$. It is stored to the position j .
- **Induction Hypothesis:** Assume the correctness of the Lemma for d .
- **Induction Step ($d \rightarrow d + 1$):** From each node $j, j = 0, \dots, 3f$, all values of the tree level of depth d are received, due to the FIFO order of the links, in the order they are stored on node j (Line 12 of Algorithm 10). The algorithm selects the k -th element, $k = 0, \dots, (3f + 1)^d - 1$, of the tuple received from each node j and stores it to position $k(3f + 1) + j$ (Line 14 of Algorithm 10). Since the k -th element of the tuple has, by the induction hypothesis, the label $\langle n_0, \dots, n_d \rangle$, the label of the newly stored item is set to $\langle n_0, \dots, n_d, j \rangle$. As

Figure 12.5: Resolve Function for $f = 1$

the position k was calculated as $n_0(3f + 1)^d + n_1(3f + 1)^{d-1} + \dots + n_d$ at the sending node, the storage position of the new element is:

$$\begin{aligned} k(3f + 1) + j &= (n_0(3f + 1)^d + n_1(3f + 1)^{d-1} + \dots + n_d)(3f + 1) + j \\ &= n_0(3f + 1)^{d+1} + n_1(3f + 1)^d + \dots + n_d(3f + 1) + j \end{aligned}$$

as required by the Lemma. □

Since the resolve function is defined only on the labels of the tree-nodes, the resolve tree is equivalent to the one of the original algorithm. The filtering done by the original algorithm is implemented as part of the resolve function in our version.

12.4 Resolve Function

The resolve function was developed for specific fault hypotheses only. Currently an implementation for a systems tolerating one Byzantine fault is available.

12.4.1 Resolve Function for a Single Byzantine Fault

Figure 12.5 shows an example for a resolve operation of a system tolerating one Byzantine fault ($f = 1$). The resolve function is formalized in Algorithm 11. The Function *majority* takes a tuple and returns the value most often present in the tuple. If multiple values have the maximum frequency a default value (0) is returned.

Algorithm 11 Resolve Function ($f = 1$)

```

1: var temp as  $3f$  Tuple
2: var maj as  $3f + 1$  Tuple
3: var k as Integer
4: for ever do
5:   // Create a Majority Value for Each Label i
6:   for  $i \in [0, \dots, 3f]$  do
7:      $k := 0$ 
8:     // Copy the  $3f$  Values to temp
9:     for  $j \in [0, \dots, 3f]$  do
10:      if  $i \neq j$  then
11:         $temp[k] := resolve\_level[i(3f + 1) + j]$ 
12:         $k := k + 1$ 
13:      end if
14:    end for
15:    // Calculate the Majority for Label i
16:     $maj[i] := majority(temp)$ 
17:  end for
18:  // Calculate the Output Value
19:   $output := majority(maj)$ 
20: end for

```

Informal Description: The algorithm creates a majority value for each label $\langle i \rangle$, $i = 0, \dots, 4$. Therefore all items with a label of $\langle i, j \rangle$, $j = 0, \dots, 4 \wedge i \neq j$ are copied from the *resolve_level* to a temporary tuple (*temp*). The index calculation is based on the definition in Lemma 12.3.3. The values processed multiple times by the same node ($\langle 0, 0 \rangle$, $\langle 1, 1 \rangle$, $\langle 2, 2 \rangle$, $\langle 3, 3 \rangle$) are skipped and the tuple *temp* therefore contains 3 elements. The resulting majority value $maj[i]$ is determined by calling the function *majority*.

The output value is calculated by again calling *majority* but this time on the 4 previously calculated majority values stored in the tuple *maj*.

12.5 Complexity of the Adapted Algorithm

As already discussed, each tree level has $(3f+1)^r$ entries. As in each round $r \geq 2$ the tree level of depth $r - 1$ is broadcasted, $(3f + 1)^{r-1}$ broadcast messages are sent by each node. In round 1 the input value is broadcasted, therefore a single ($= (3f + 1)^0$) broadcast message is sent by each node.

Based on the definition of the lockstep round creation, described in Chapter 10, t'_{sync} additional synchronization messages are generated to compensate for the precision of the underlying multisynchronous clocking scheme as well as the message delays,

leading to a synchronization time of t'_{sync} message slots. With a time complexity of $f + 1$ rounds [AW04], the number of broadcasted messages per node are:

$$\sum_{r=1}^{f+1} ((3f + 1)^{r-1} + t'_{\text{sync}}) = t'_{\text{sync}} (f + 1) + \frac{(3f + 1)^{f+1} - 1}{3f}$$

For the current implementation of the message layer and a precision of 4 a synchronization time t'_{sync} of one message slot is sufficient (see Chapter 10 for the calculation description), leading to

$$\underbrace{f + 1}_{\text{Synchronization Messages}} + \underbrace{\frac{(3f + 1)^{f+1} - 1}{3f}}_{\text{Application Messages}}$$

broadcast messages per node.

The fact that these messages are tightly packed (see Chapter 10) leads to an execution time of:

$$t_{\text{micro}} = t_{\text{slot}} \left(f + 1 + \frac{(3f + 1)^{f+1} - 1}{3f} \right)$$

microticks. The message slot time t_{slot} of the current communication layer implementation is 10 microticks. Therefore the time complexity is:

$$t_{\text{micro}} = 10 \left(f + 1 + \frac{(3f + 1)^{f+1} - 1}{3f} \right)$$

leading to an exponential time complexity. Additional time is needed for calculating the output value. Since this calculation is done in parallel to the next information gathering phase it is not included into the runtime analysis.

Table 12.1 gives an overview on the runtime of the algorithm for different fault hypotheses and different clock rates. The values are based on the current implementation of the communication layer. The growth rate of the runtime is visualized in Figure 12.6.

Table 12.1: Runtime of the Consensus Implementation

	Microticks	50 MHz	100 MHz	200MHz
$f = 1$	70	1.4 μs	700 ns	350 ns
$f = 2$	600	12 μs	6 μs	3 μs
$f = 3$	11150	223 μs	111.5 μs	55.75 μs
$f = 4$	309460	6.18 ms	3.09 ms	1.55 ms
$f = 5$	11184870	223.7 ms	111.8 5ms	55.93 ms

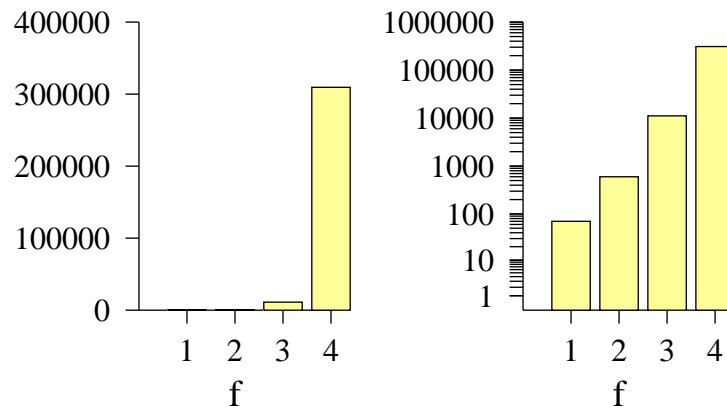


Figure 12.6: Runtime of the Consensus Algorithm [Microticks]

As apparent from the algorithms, $3(3f+1)^{f+1}$ memory locations are needed to store the tuples (*current_level*, *receive_level*, *resolve_level*). The temporarily defined values (as the counters and the tuples *temp* and *maj*) are implemented as aliases, selecting but not storing the corresponding values.

12.6 Circuit Design

The consensus algorithm implementation is based on the basic node model introduced in Chapter 10. Figure 12.7 shows how the consensus algorithm is integrated into the node.

As already mentioned, the circuit is timed by the *roundtick* signal of the lockstep synchronous round creation circuit. Due to the precalculated round timing it is assured that the messages of all non faulty nodes are delivered timely.

An instance of the communication layer presented in Chapter 8 is used to communicate with the other nodes. As a faulty node may not send all messages, the signalization mechanism for message receptions of the communication layer is not used. Instead the message reception is locally timed by the *messagetick* signal of the round counter module. As the creation of this signal is solely based on the local microtick clock, the timing of the message reception can not be affected by a faulty node.

The reception of the next tree level is implemented using a single state machine. The currently built tree level (*receive_level*) is stored within a register bank directly located in the receiver state machine component. After having received the complete

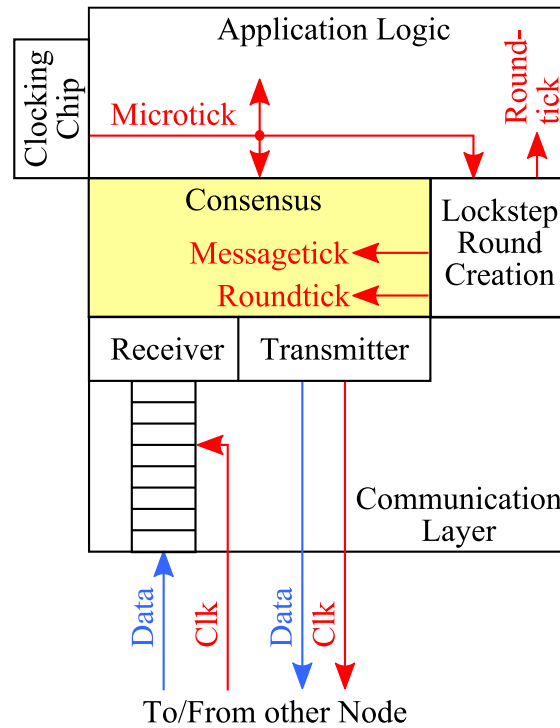


Figure 12.7: Node Implementation

tree level, its contents is copied to a second register bank (*current_level*), so it can be used as input for the broadcast operation of the next round.

The message broadcast operation is implemented using two tightly coupled state machines. The sender state machine is responsible for establishing the round pattern (based on the *roundtick* signal) and the transmission of the input value at the appropriate time (the beginning of the first round). The tree level serializer, also a state machine, coordinates the serialization of the current tree level (*current_level*) based on the round pattern established by the sender state machine.

The resolve function is implemented using purely combinational logic. As this implementation has a high logic depth, its runtime may take multiple microticks. To ensure a clean output signal (eliminating all spikes), the result of the resolve function is buffered using an output register. This register is updated solely when a new information gathering phase has finished. The input value of the resolve function is stored within a register bank (*resolve_level*). The timing of the copy operation is determined by the receiver state machine. Therefore the execution time of the resolve function takes as long as a full information gathering phase. As the implementation of the system is fully synchronous, the toolset ensures that the execution time of the resolve functions meets its bound.

This form of implementing the resolve function leads to a pipelined architecture,

which means that while the output value of the current iteration is calculated, the information gathering phase of the next iteration is already executed.

12.7 Execution Example

Figure 12.8 contains a VHDL timing simulation for a system tolerating one Byzantine fault. The shown execution is fault free. As you can see, the output values are always the majority of the input values, except if no majority exists. If no majority exists, a default value (0) is used as output value. As the *data_stream* signals are no longer readable in the figure, Figure 12.9 additionally shows a detail of the execution such that the communication between the nodes becomes visible.

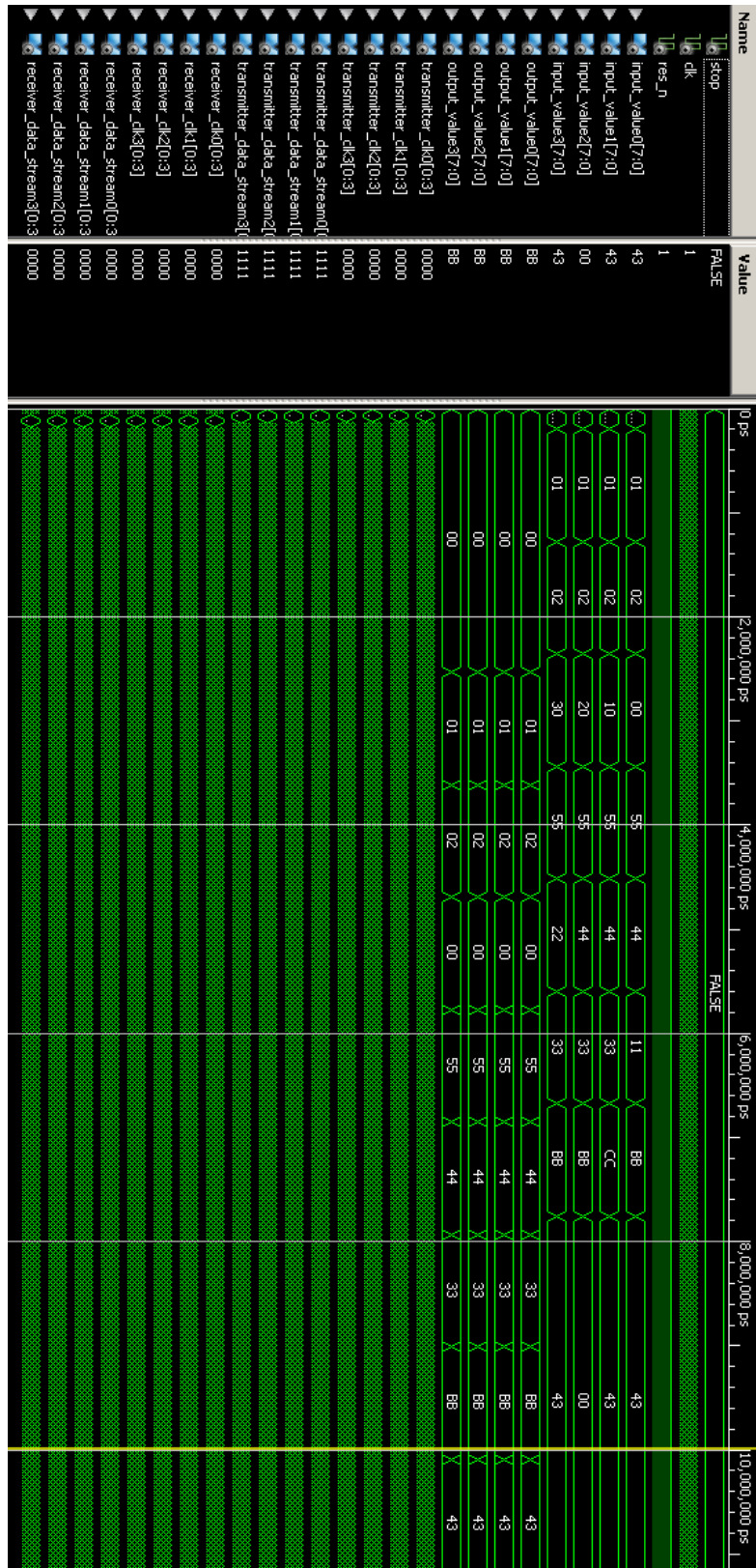


Figure 12.8: Execution Example of a Consensus System

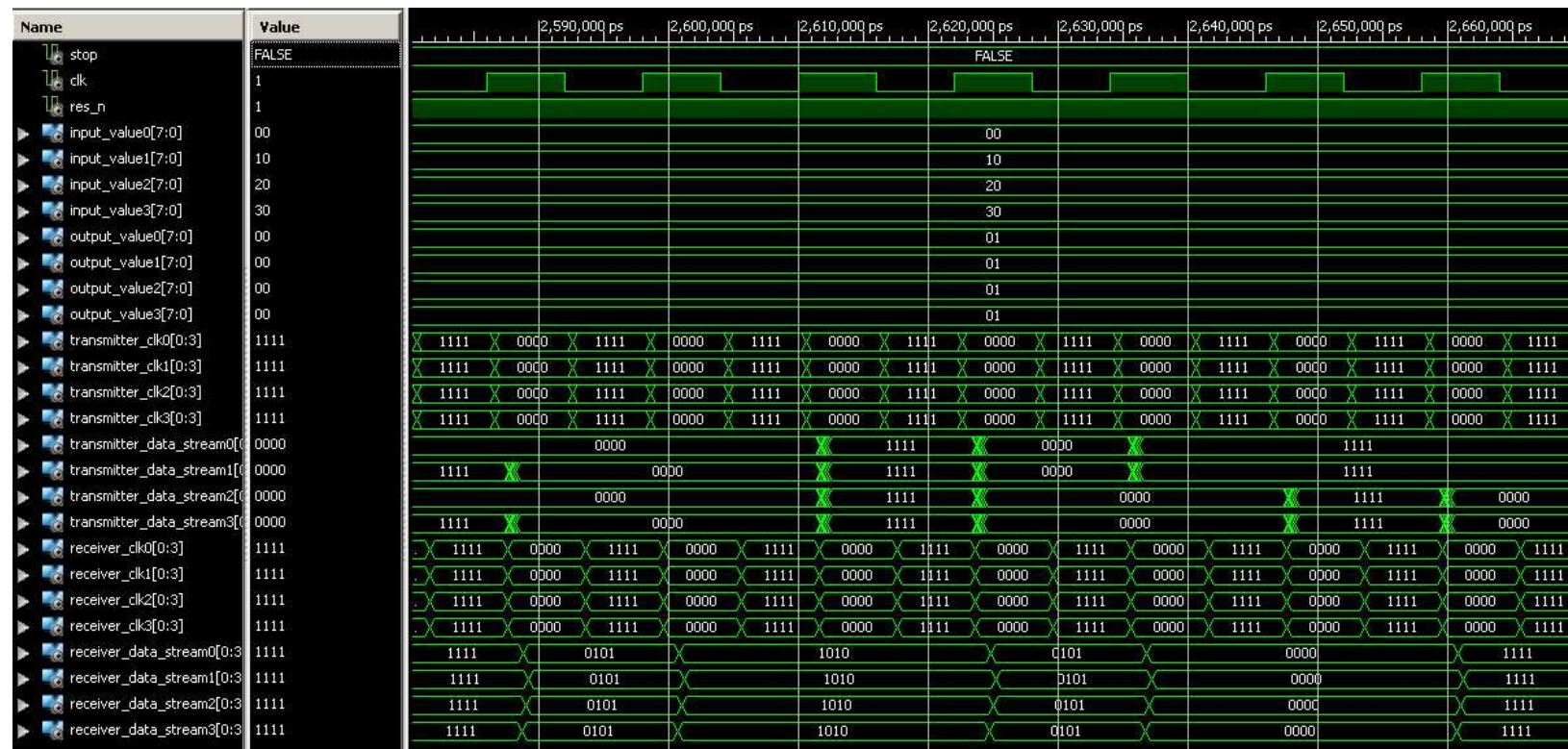


Figure 12.9: Execution Example of a Consensus System (Detail)

Part IV

Conclusion and Future Work

Chapter 13

Future Work

As in every project, the available time is limited. Therefore not all questions could be solved and some potential for future enhancements has been identified.

13.1 Flow Control and Error Detection

Currently it is assumed that the application logic can handle a data transmission with the full communication speed or otherwise will implement flow control on the architectural level. In future extensions we want to support flow control as service of the communication layer.

Additionally we will investigate different options for data protection. A way to achieve this is by grouping multiple data bytes into packets, and each packet is protected by a CRC byte.

Another important point is the detection of timing violations to identify nodes not synchronized correctly to the multisynchronous ensemble. The current solution can handle this, but if identified, such a node could be reinitialized and reintegrated into the system. This would increase the reliability of the system.

As for the flow control, we currently assume that all error detection and correction mechanisms are implemented within the application logic.

13.2 Resynchronization at Runtime

An important question when handling data transmissions is how to resynchronize a receiver to a data stream, once it has lost its lock. Using a dedicated start-packet symbol, a resynchronization at the message level is simple by waiting for this symbol

to occur. Unfortunately the correct initialization of the ring buffer's read- and write-pointer is not solvable that easily. As our proof heavily depends on the fact that the precision at startup is zero, a resynchronization will fail with a high probability, because of the unknown precision at such an event.

Two possibilities to circumvent the problem are:

- Usage of a larger buffer
- Support from the clocking system

The approach of using a larger buffer is simple and will work without changing the clocking system. Instead of assuming a perfectly synchronous startup, the precision maximum is assumed. Therefore the buffer will be sufficiently large to compensate for any arbitrary precision, as long as it is within the given bounds, at the resynchronization point. Unfortunately a larger buffer size also increases the message latency.

Support from the clocking system would enable us to correctly set the write and read pointer in case of a resynchronization and the buffer will defiantly not over- or under run.

13.3 Ring Buffer Integration into the Clocking Chip

A problem of the current implementation is that the FPGA needs to maintain multiple clock sources (one per data line) to control the respective buffer write accesses. Each of these clocks needs to be routed to the FPGA along with the associated data.

For the next release we plan to integrate buffers and transmitter synchronization, for a system tolerating one Byzantine fault ($f = 1$), directly into the clocking chip. Therefore three ring-buffer instances are needed. The fourth communication link, transmitting data to the node itself, is routed internally in the FPGA and is fully synchronous. Therefore the ring buffer can easily be implemented fully synchronous.

A scheme of such a chip is shown in Figure 13.1. In this solution the buffer handling is naturally performed inside the clocking chip, where all required clocks are available anyway. Hence there is no need for extra clock routing, the data lines can simply be aligned to the associated clock traces in the (existing) clock-net. At the same time the FPGA can be implemented fully synchronous, using one single clock source only.

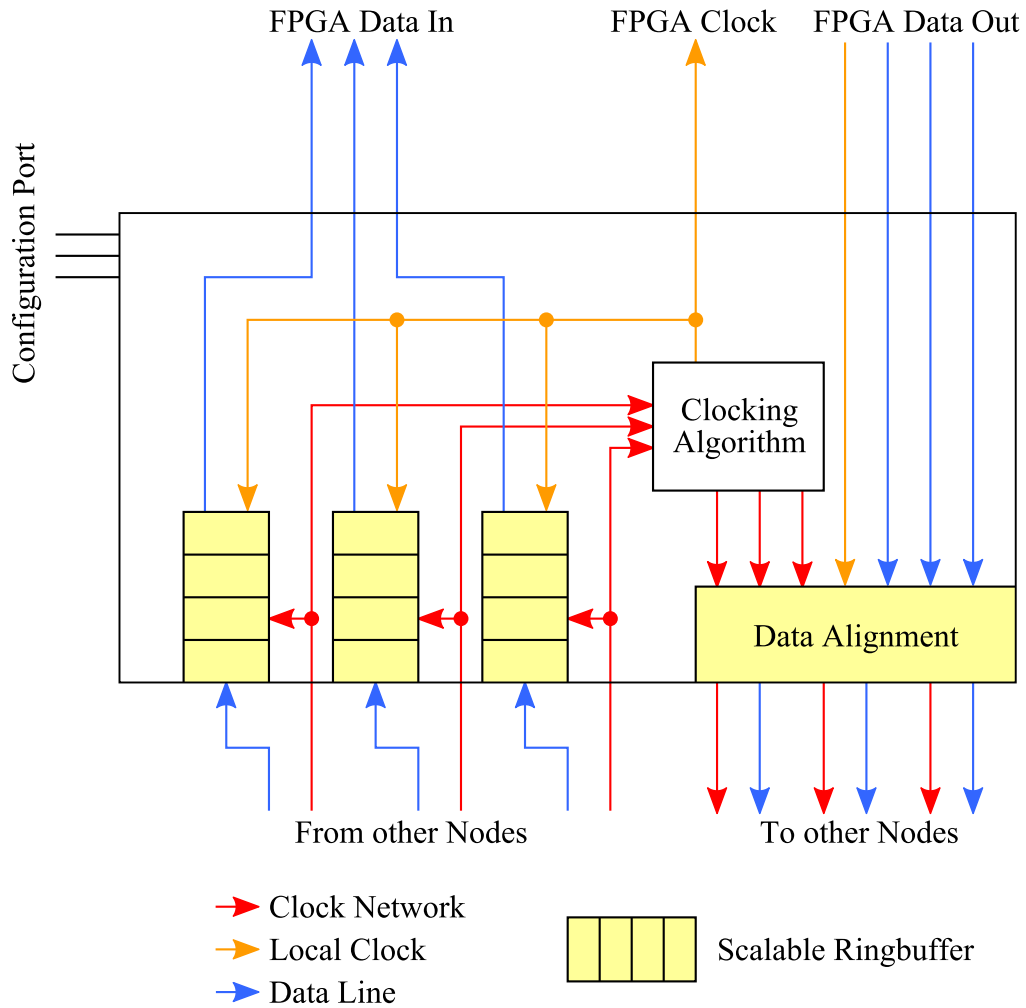


Figure 13.1: Clocking Chip – Ring-Buffer Integration

13.4 Resolve Functions for Stronger Fault Hypotheses

Currently only a resolve function for a fault hypothesis of one Byzantine fault ($f = 1$) has been implemented. In future at least an implementation for two faults will be implemented. This is not a fundamental problem but need efficient implementation.

Chapter 14

Conclusion

Multisynchronous clocking is an attractive alternative to globally synchronous clocking in modern high-speed VLSI circuits, such as complex SoCs, which also allows to avoid the single point of failure usually represented by a central clock source.

We have shown how to employ the loose synchrony provided by multisynchronous systems for implementing a high-speed pipelined communication scheme that is metastability-free by construction. It employs a bounded-size FIFO buffer for compensating the skew between the sender and receiver clock. We derived a reasonably tight lower bound for the required buffer size, and provided a formal proof of correctness and freedom of metastability. Furthermore, we have described an efficient implementation of our communication scheme, and experimentally demonstrated its feasibility using a custom test system.

Based on the communication layer we have shown how to implement a lockstep synchronous round model relying only on the local microtick clock. Therefore faulty nodes can never affect the timing of non faulty ones. The round structure is defined for a maximum number of messages transmitted in each round. It is guaranteed that all of these messages, sent by non faulty nodes, are delivered timely within the current round. Additionally the receive event for those messages is also derived from the local clock and is based solely on the a-priory known constant message latency.

We have shown that a naive TMR implementation in a multisynchronous environment will definitely fail and have given an implementation example for a multisynchronous voter. Nevertheless such a system will never be able to tolerate Byzantine faults in the general case.

Our goal was to increase the level of fault tolerance implementable in hardware. Therefore we have taken a well known consensus algorithm tolerating Byzantine failures, namely the EIG algorithm, and have modified it such that it is implementable in hardware. We have shown that the two algorithms are equivalent and sketched a hardware implementation for our solution.

Bibliography

- [AJTR98] T. Arabim, J. Jones, G. Taylor, and D. Riendeau. Modeling, Simulation, and Design Methodology of the Interconnect and Packaging of an Ultra-High Speed Source Synchronous Bus. In *IEEE 7th Topical Meeting on Electrical Performance of Electronic Packaging*, 1998, pages 8–11, 1998.
- [AW04] Hagit Attiya and Jennifer Welch. *Distributed Computing – Fundamentals, Simulations, and Advanced Topics*. John Wiley and Sons, Inc., second edition, 2004.
- [BAS03] Ravi Budruk, Don Anderson, and Tom Shanley. *PCI Express System Architecture*. Addison-Wesley Professional, 2003.
- [Bla69] J.R. Black. Electromigration - A Brief Survey and some Recent Results. *IEEE Transactions on Electron Devices*, 16(4):338–347, Apr 1969.
- [Cha84] D. M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Stanford Univ., 1984.
- [CM73] T. J. Chaney and C. E. Molnar. Anomalous Behavior of Synchronizer and Arbiter Circuits. *IEEE Transactions on Computers*, 22(4):421–422, 1973.
- [Con03] Cristian Constantinescu. Trends and Challenges in VLSI Circuit Reliability. In *IEEE Micro*, volume 23, 2003.
- [Die05] Reinhard Diestel. *Graph Theory*. Springer-Verlag, Heidelberg, 2005.
- [DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the Presence of Partial Synchrony. *J. ACM*, 35(2):288–323, 1988.
- [DP98] William J. Dally and John W. Poulton. *Digital Systems Engineering*. Cambridge University Press, 1998.

- [EA97] F. Najm E. Amerasekera. *Failure Mechanisms in Semiconductor Devices*. Wiley, second edition, 1997.
- [Esa03] SpaceWire – Links, Nodes, Routers and Networks, 2003.
- [FB96] K.M. Fant and S.A. Brandt. NULL Convention Logic: A Complete and Consistent Logic for Asynchronous Digital Circuit Synthesis. In *ASAP 96. Proceedings of International Conference on Application Specific Systems, Architectures and Processors, 1996.*, pages 261–273, Aug 1996.
- [FFSK06] Markus Ferring, Gottfried Fuchs, Andreas Steininger, and Gerald Kempf. VLSI Implementation of a Fault-Tolerant Distributed Clock Generation. *IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT2006)*, pages 563–571, 2006.
- [FH90] Paul Forshaw and Reinhard Hahn. Synchronous Design: The Right Technique for Digital ASIC's. In *ASIC Seminar and Exhibit*, pages P6/1.1 – P6/1.5, 1990.
- [FLP85] Michael J. Fischer, Nancy A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with one Faulty Process. *Journal of the ACM*, 32(2):374–382, April 1985.
- [Fri01] Eby G. Friedman. Clock Distribution Networks in Synchronous Digital Integrated Circuits. In *Proceedings of the IEEE*, volume 89, pages 665–692, 2001.
- [GG04] Ian A. Glover and Perer M. Grant. *Digital Communications*. Pearson Prentice Hall, second edition, 2004.
- [Gin03] Ran Ginosar. Fourteen Ways to Fool Your Synchronizer. In *ASYNC '03: Proceedings of the 9th International Symposium on Asynchronous Circuits and Systems*, pages 89–96. IEEE Computer Society, 2003.
- [Hau95] S. Hauck. Asynchronous Design Methodologies: An Overview. *Proceedings of the IEEE*, 83(1):69–93, Jan 1995.
- [HLD95] R.E. Harper, J.H. Lala, and J.J. Deyst. Fault Tolerant Parallel Processor Architecture Overview. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'*, pages 62–67, Jun 1995.
- [HO71] Richard A. Harrison and Daniel J. Olson. Race Analysis of Digital Systems without Logic Simulation. In *DAC '71: Proceedings of the 8th Design Automation Workshop*, pages 82–94. ACM, 1971.

- [IEE95] IEEE Standard for Heterogeneous InterConnect (HIC), 1995.
- [IEE00] IEEE Standard Signaling Method for a Bidirectional Parallel Peripheral Interface for Personal Computers, 2000.
- [KC87] Lindsay Kleeman and Antonio Cantoni. Metastable Behavior in Digital Systems. *IEEE Design and Test of Computers*, 4(6):4–19, 1987.
- [KH04] T. Karnik and P. Hazucha. Characterization of Soft Errors Caused by Single Event Upsets in CMOS Processes. *IEEE Transactions on Dependable and Secure Computing*, 1(2):128–143, April-June 2004.
- [Kin08] David J. Kinniment. *Synchronization and Arbitration in Digital Systems*. Wiley, 2008.
- [Kop97] Hermann Kopetz. *Real-Time Systems – Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.
- [Lam03] Leslie Lamport. Arbitration-Free Synchronization. *Distributed Computing*, 16(2-3):219–237, 2003.
- [LMS85] Leslie Lamport and P. M. Melliar-Smith. Synchronizing Clocks in the Presence of Faults. *J. ACM*, 32(1):52–78, 1985.
- [LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.
- [LV62] R.E. Lyons and W. Vanderkulk. The Use of Triple-Modular Redundancy to Improve Computer Reliability. *IBM Journal of Research and Development*, 6(2):200–209, April 1962.
- [MFMR04] C. Metra, S. Di Francescantonio, T.M. Mak, and B. Ricco. Implications of Clock Distribution Faults and Issues with Screening them During Manufacturing Testing. *IEEE Transactions on Computers*, 53(5):531–546, 2004.
- [Mil04] Gene H. Miller. *Microcomputer Engineering*. Prentice Hall, third edition, 2004.
- [Nat08] *LVDS Owner’s Manual*. National Semiconductors, fourth edition, 2008.
- [Nxp07] I²C-Bus Specification and User Manual, 2007.

- [OKS08] R. Obermaisser, H. Kraut, and C. Salloum. A Transient-Resilient System-on-a-Chip Architecture with Support for On-Chip and Off-Chip TMR. In *EDCC 2008. Seventh European Dependable Computing Conference, 2008.*, pages 123–134, May 2008.
- [Pci98] PCI Local Bus Specification, 1998.
- [PG07] Ivan Miro Panades and Alain Greiner. Bi-Synchronous FIFO for Synchronous Circuit Communication Well Suited for Network-on-Chip in GALS Architectures. In *First International Symposium on Networks-on-Chip, 2007. NOCS 2007.*, pages 83–94, May 2007.
- [Sch87] Fred B. Schneider. Understanding Protocols for Byzantine Clock Synchronization. Technical Report, Cornell University, Department of Computer Science, aug 1987.
- [Sei79] Charles L. Seitz. System Timing. In Carver Mead and Lynn Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1979.
- [SFGP09] Yebin Shi, S. B. Furber, J. Garside, and L. A. Plana. Fault Tolerant Delay Insensitive Inter-chip Communication. In *15th IEEE Symposium on Asynchronous Circuits and Systems*, pages 77–84, May 2009.
- [SG03] Yaron Semiat and Ran Ginosar. Timing Measurements of Synchronization Circuits. In *ASYNC '03: Proceedings of the 9th International Symposium on Asynchronous Circuits and Systems*, page 68. IEEE Computer Society, 2003.
- [Sha02] Tom Shanley. *InfiniBand Network Architecture*. Addison-Wesley Professional, 2002.
- [Sho02] Martin L. Shooman. *Reliability of Computer Systems and Networks: Fault Tolerance, Analysis, and Design*. Wiley, 2002.
- [ST03] T. K. Srikanth and Sam Toueg. Optimal Clock Synchronization. *Journal of the ACM*, 34:626–645, 2003.
- [TGL07] P. Teehan, M. Greenstreet, and G. Lemieux. A Survey and Taxonomy of GALS Design Styles. *Design and Test of Computers, IEEE*, 2007.
- [Wak01] John F. Wakerly. *Digital Design Principles and Practices*. Prentice-Hall, third updated edition, 2001.
- [WS05] Josef Widder and Ulrich Schmid. Achieving Synchrony without Clocks. Research Report 49/2005, Technische Universität Wien, Institut für Technische Informatik, 2005. (submitted).