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

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

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Solving the Labeling Problem

A Byzantine Fault-Tolerant Self-Stabilizing FPGA Prototype based on the FATAL$^+$ Protocol

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Technische Informatik

eingereicht von

Markus Hofstätter
Matrikelnummer 0725034

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.techn. Ulrich Schmid
Mitwirkung: Univ.Ass. Dipl.-Ing. Dr.techn. Matthias Függer

Wien, 17. Juni 2013

(Unterschrift Verfasser/in) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Solving the Labeling Problem

A Byzantine Fault-Tolerant Self-Stabilizing FPGA Prototype based on the FATAL$^+$ Protocol

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

by

Markus Hofstätter
Registration Number 0725034

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Ulrich Schmid
Assistance: Univ.Ass. Dipl.-Ing. Dr.techn. Matthias Függer

Vienna, June 17, 2013

(Signature of Author)

(Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Markus Hofstätter
Dechant-Neidlgasse 10, 2223 Klein-Harras

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

(Ort, Datum) (Unterschrift Verfasser/in)

Acknowledgements

I want to thank my familiy and friends for their support over the years that made the thesis possible. Moreover I want to thank my advisor, Matthias Függer and Christoph Lenzen for their feedback, suggestions and effort put into my thesis.

Abstract

The topic of this thesis lies in the intersection of VLSI design and fault-tolerant distributed algorithms. It is devoted to the development of an FPGA implementation of the well-known synchronous Phase King consensus algorithm using single-bit serial communication and the design of a suitable testbench for verifying its operation. The implementation is finally integrated into an existing prototype and testbench of the self-stabilizing Byzantine fault-tolerant distributed clock generation scheme FATAL$^+$, where it is used to generate 17-bit wide synchronized clocks without increasing the stabilization time of the underlying scheme. The thesis also explores implementation alternatives and provides the correctness proofs of the employed algorithm.

Kurzfassung

Das Thema dieser Masterarbeit liegt im Querschnitt der Gebiete VLSI-Design und fehlertoleranter verteilter Algorithmen. Sie ist einer FPGA Implementierung des bekannten synchronen Phase King Konsensus Algorithmus mittels 1-bit serieller Kommunikation und der Entwicklung einer entsprechenden Testumgebung gewidmet. Die Implementierung wurde darüberhinaus mit einem bereits existierenden Prototyp eines selbst-stabilisierenden, fehlertoleranten verteilten Algorithmus zur Taktgenerierung integriert, um ohne Verlängerung der Stabilisierungszeit 17-bit breite synchronisierte Uhren zur Verfügung stellen zu können. Die Arbeit untersucht auch mögliche Implementierungsalternativen und enthält die Korrektheitsbeweise der verwendeten Algorithmen.

Contents

1 Introduction 1
1.1 Motivation . . . 1
1.2 The problem . . . 3
1.3 The solution . . . 3
1.4 Aim of the work . . . 4
1.5 Structure of the thesis . . . 4

2 State of the art 5
2.1 Hardware-level distributed systems . . . 5
2.2 Digital Logic . . . 16
2.3 Metastability . . . 22
2.4 Synchronous circuits and state machines . . . 25
2.5 Asynchronous circuits and state machines . . . 30
2.6 Globally Asynchronous and Locally Synchronous Systems . . . 39
2.7 Designflow . . . 44

3 FATAL$^+$ and the Labeling solution 51
3.1 Pulse synchronization with FATAL$^+$ . . . 51
3.2 (M, l)-labeling problem . . . 59
3.3 Related Work . . . 60
3.4 Solution algorithm . . . 62
3.5 Proofs . . . 64

4 Implementation 69
4.1 Components . . . 69
4.2 Threshold Modules . . . 70
4.3 Lock-step synchronous round communication . . . 79
4.4 Synchronous FSM . . . 83

5 Experiments 89
5.1 Hardware and Software . . . 89
5.2 Testbenches . . . 90
5.3 Experimental Validation . . . 96
5.4 Results . . . 99

6 Conclusion **103**
6.1 Summary of accomplishments 103
6.2 Critical reflection and future work 103

A Thesis directory contents **109**

Bibliography **113**

CHAPTER 1

Introduction

1.1 Motivation

Over the past years, continuously shrinking feature sizes facilitated *Very Large Scale Integration* (VLSI) circuits that contain more transistors, switch faster and are still energy efficient [33]. Most of these circuits are designed under the monolithic fault-free *synchronous* design paradigm. A single control signal, called the *clock*, is distributed via a *clock tree* [35] all across the circuit and a global *reset* is used to simultaneously initialize the entire circuit. The clock itself is generated by an oscillator circuit that produces a periodic signal of the desired frequency. However, due to varying delays (uncertainty) along the paths and logic in between, it is not possible in practice to distribute the clock and reset signals to the required elements such that signal transitions happen simultaneously. These transition variations are a result of environment effects (e.g. temperature and noise) and process variance in the production of the circuit. If faster circuits are desired, it is necessary to minimize delays, which in turn allows to decrease the clock period. Especially the delay uncertainty needs to be made as small as possible to maintain the assumption of simultaneity on an abstract level, i.e. to establish synchrony. Maintaining this requirement becomes increasingly complex due to the overwhelming amount of logic to which the clock must be distributed. As a result, the clock tree consumes a significant part of the overall chip area and energy, because the delay uncertainty is reduced by inserting delays either via longer wires or buffer-logic [52].

Another drawback of the synchronous design paradigm becomes apparent if parts of the chip shall be made fault-tolerant or at least more robust against faults. If clock or reset fail, the whole systems is likely to fail (single point of failure). Except for special safety critical and reliable applications such as in medical, aerospace and space products, not much effort has been put in fault-tolerance mechanisms in VLSI circuits [80]. However, due to increasing technology and process variations and shrinking signal voltages the assumptions of no or restricted faults do not hold anymore [7, 16, 22, 58, 61]. Thus, some effort has been put into improving architectural measures and process technology to ensure robust circuits.

Fault-tolerance or robustness measures involve redundancy to make a component less likely to fail. An often used architectural concept in fault-tolerant implementations is *Triple Modular Redundancy* (TMR). In its simplest form, a component that shall be made fault-tolerant is replicated three times (ensemble) [47]. The ensemble outputs are merged into a single output by a voting component (e.g. majority vote on each output bit) either in all receiver components or once in the TMR ensemble if the voter is sufficiently robust and not likely to fail. TMR is only able to cover the failure of a single component of the ensemble, which poses a problem in high failure rate scenarios. A more fine-grained approach is to increase the robustness of the modules itself used to build a component. For example, many components contain memory or storage cells to maintain the current state of algorithms, but any VLSI circuit can be affected by environmental effects like high-energy ionizing particles. These particles are able to induce charge into the circuits and may thus flip a stored bit in the memory cells, called *Single Event Upset* (SEU). One approach to mitigate induced charges in memory cells is to use *Dual Interlocked Storage Cells* (DICE) [12], which is a redundant storage cell. If one of the two cells flips its stored bit and the effect of the particle hit decays fast enough, the second cell will force the failed cell into the correct state again. Depending on the implementation, it is not guaranteed, however, that the readout is correct while one of the cells is still erroneous.

So, on one hand, maintaining a single global clock becomes more difficult and leads to a single point of failure. On the other hand, fault-tolerant distributed algorithms and fault-recovery algorithms benefit from the availability of global synchrony, because it allows to implement lock-step synchronous rounds and provides a global time base to the system. For example, such a time base may be used to construct global schedules necessary to coordinate message transmissions and actions performed by the system globally.

In many newer *Systems on Chip* (SoC) the single global clock is replaced by multiple, usually independent and unrelated, clocks. This *Globally Asynchronous Locally Synchronous* (GALS) [13] design paradigm is spreading rapidly [40, 77, 83]. Each subsystem is implemented synchronously, using its own clock domain, and is hence able to run at its own pace. The interfaces between the subsystems use asynchronous design styles. If the clocks are completely unrelated, the subsystems may violate each others' interface requirements sometimes, resulting in possible undesired behaviour like metastability (see Chapter 2.3). Fault-tolerant distributed algorithms, in which the nodes execute their algorithms independently of each other and just exchange information across their interfaces, share similar requirements as GALS systems.

The special multi-synchronous GALS systems [64] are of particular interest to designers. Herein the available clocks are not completely unrelated, but maintain the same nominal frequency with a possibly unknown but bounded phase relation. The combined standard local synchrony plus availability of a global time base allow to implement metastability-free high-speed communication across the clock domains [64].

However, it is apparent that fault-tolerant approaches, for generating multi-synchronous clocks for GALS systems, are needed. Due to the increasing vulnerability w.r.t. transient faults, it may even be necessary to implement self-stabilizing mechanisms in order to recover from fault bursts that may have left the whole system in an unspecified resp. undesired state.

1.2 The problem

Sound models, correctness proof and a prototype implementation of a Byzantine fault-tolerant and self-stabilizing pulse synchronization algorithm (*FATAL*$^+$) for multi-synchronous GALS systems have been developed in [26, 66]. The solution consists of a "fast" Byzantine fault-tolerant pulse generation algorithm *FATAL*$^+$, which generates tightly synchronized high-frequency clock pulses at all correct nodes. For self-stabilization, it relies on a lower-level slow pulse synchronization algorithm *FATAL* that is able to recover from an arbitrarily corrupted system state. Moreover *FATAL*$^+$ provides a *bounded synchronized clock* (BSCLK) driven by the fast pulses (which offers less than 8-bit in the current prototype).

Given the high clock frequency achieved by FATAL$^+$, fast wrap-arounds (overflows) of the BSCLK are inevitable. Since enlarging the width of the FATAL$^+$ BSCLK would increase the stabilization time of both FATAL and FATAL$^+$ significantly, the problem to be solved in this thesis was to extend the width of the synchronized clocks to $24, \ldots, 64$-bits atop of BSCLKs by solving the *labeling problem*: The challenge is to maintain labels at every node, which are the same at all correct nodes and are incremented whenever the BSCLK wraps around (i.e., overflows).

Since the BSCLK is self-stabilizing it must be ensured that once the underlying protocol has stabilized the extended clock stabilizes as well, i.e., fulfills its specification eventually at all correct nodes, even in the presence Byzantine faulty nodes. If all correct nodes have stabilized their labels must not be influenced by faulty nodes and a label that became corrupted by a transient fault at some node must be recovered to match the labels of the correct nodes. In a fault-free scenario, i.e., where all nodes are correct, metastable upsets must not occur after stabilization. During stabilization and in the presence of faults, the upset probability should be as low as possible. To guarantee scalability, the algorithm must be restricted to use serial (i.e., 1-bit) channels.

1.3 The solution

The solution algorithm [39] is based on the well-known synchronous *Phase King* [10] consensus algorithm, which outputs the same input value of a correct node at all correct nodes. The Phase King algorithm can be implemented efficiently if it is used to solve 1-bit (i.e., binary) consensus. Hence the labeling problem is first reduced to the 1-bit consensus problem, by agreeing on every label bit. If all correct nodes already maintain the same label, all correct nodes will keep it as their candidate otherwise they will reset the candidate label to 0. The required synchrony is provided by the lowest bit of the short clock of FATAL$^+$, and the label is set to the incremented candidate at the wrap-around of the BSCLK to ensure progress. Furthermore, the wrap-around of the BSCLK is used to reset every node into the initial state of the labeling algorithm, which implements the self-stabilization mechanism. As a result, the labeling algorithm guarantees stabilization within 2 short clock wrap-arounds after FATAL$^+$ has stabilized.

1.4 Aim of the work

The primary task of this thesis was to develop a *Field-programmable gate array* (FPGA) implementation of the solution described in [39] and a suitable testbench for verifying its proper operation. For integration testing the implementation was finally combined with an existing implementation and testbench of the FATAL$^+$ clock generation scheme [26]. The resulting solution algorithm provides wide synchronized clocks that stabilize eventually even in the presence of Byzantine faulty nodes.

The results are:

- An implementation of the above algorithm and all necessary components in digital logic using VHDL. This includes concepts for an efficient and glitch-free serial communication between partially asynchronous state machines as well as an efficient Phase King consensus implementation.
- Both a standalone testbench and an integration into the existing testbench of the FATAL$^+$ protocol. These testbenches shall provide insights into possible problems, limitations and performance as well as validate the correctness of theoretical results.
- Synthesis and experimental evaluation of a fully-fledged FPGA-prototype.

1.5 Structure of the thesis

As this thesis lies in the domain of VLSI design and distributed algorithms, an overview of the basics of both fields is presented in Chapter 2. Next, the structure and implementation of the underlying FATAL$^+$ protocol and its interfaces are explained, followed by a detailed description of the solution algorithm for the labeling problem along with its correctness proofs in Chapter 3. Chapter 4 evaluates some implementation options for the necessary modules and provides an overview of the final implementation. Finally the testbenches and the results of the experimental evaluation are provided in Chapter 5. The thesis is concluded in Chapter 6, which summarizes the accomplishments and reflects on the achieved goals.

CHAPTER 2

State of the art

The following chapter will provide a brief overview about digital hardware design and its challenges, the typical design flow, timing analysis, asynchronous and synchronous circuits and FPGA structures. Furthermore, it will provide important definitions and logic concepts of (fault-tolerant) distributed systems, which will be streamlined toward compatibility with digital hardware design. The description will integrate the general framework developed in [25, 40] with classic distributed computing models [55].

2.1 Hardware-level distributed systems

A distributed system consists of a finite set of (independent) computing *nodes* $\mathcal{V} = \{1, \ldots, n\}$ connected via communication *links*. The endpoints of a link are called input resp. output *ports*. A link may be a physical wire on the chip, on a *printed circuit board* (PCB) or even an off-board communication channel like a wireless link. Obviously, it takes some amount of time to communicate with another node over a link, which is known as the *link delay*. The delays are not stable and may vary within the interval $[d_{\text{min}}, d_{\text{max}}]$ as mentioned in the introduction. The lower and upper bound is the minimum delay d_{min} and the maximum delay d_{max} respectively, and their difference is referred to as *jitter* resp. *delay uncertainty*. If at least the bound d_{max} is known the link is called a *bounded delay link*.

Typical distributed systems employ one of the following two communication styles for communication among nodes:

Event-based: Event-based communication (classic message passing) transmits messages over the link, which are explicitly stored in a buffered output port by the sender and are delivered via a buffered input port after reception. The ports store the messages until the link is ready to transmit resp. the recipient is ready to process them. Starting the transmission resp. processing of a message will remove it from the corresponding buffer. Usually, messages are only generated upon the occurrence of some certain computing event on the sender.

State-based: State-based communication continuously transmits a certain value, a state, taken from some bounded value set, over a link. It assumes that the sender provides the state to be communicated on its output port. The receiver continuously provides the state seen at its input port to the recipient node. Note that these links are typically bounded delay links. Therefore there is delay in $[d_{\text{min}}, d_{\text{max}}]$ until a new state provided by the sender is visible at the recipient. Since there is no (or finite) buffering at the receiver, it may happen that the recipient overlooks a communicated state.

Note that state-based communication with bounded delay links is used as the primary communication style in VLSI circuits, which use m parallel wire for communicating m bits of state data.

A node is implemented using a certain set of arbitrary *modules*, such as *finite state machines*(FSM), *timers/timeouts*, *clocks* and *memory elements/flags*. These modules are connected via links and ports just like nodes, which are referred to as local ports and local links. In general a node implements an algorithm consists of multiple concurrently executing modules, each performing one or more instructions (also called operation resp. actions), typically realized by some sub-modules. Many algorithms are implemented resp. specified using a FSM, which communicates its state over the links to other nodes (see Figure 2.1).

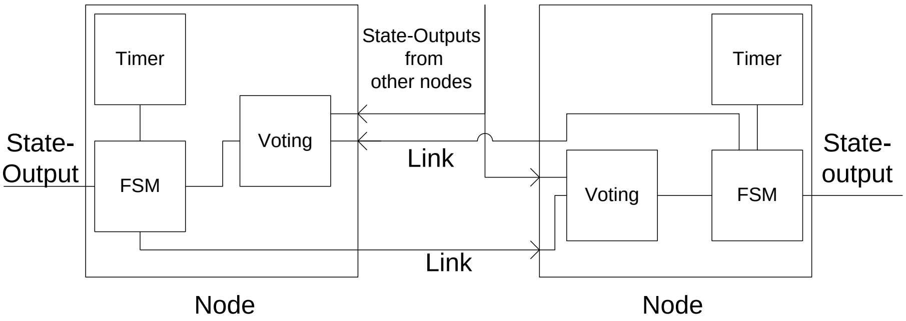


Figure 2.1: Depicting two nodes of an example distributed system containing sub-modules.

The basic instructions typically used in a (sub-)module range from variable assignments resp. memory element set and reset, sending messages resp. states to standard boolean operations.

2.1.1 Executions

In this chapter, I introduce the notation for specifying and reasoning about the executions of modules.

Computation Events: A computation event represents a state transition performed by some particular module at a given time instant. The granularity of events depends on the specification level. Usually, only (multiple) statements typical for programming languages and FSMs are considered (and are executed atomically in an event), but events of individual logic gates or low level modules (like timers) could be considered as well. However, computation events can only affect the local state and the output ports of a module. Hence, a (computation) *step* of a node will generate a new state at the output ports via the transitions of its module depending on the local state consisting of its (buffered) input ports and module-internal local ports of sub-modules. Note that in case of bounded delay computations, it is assumed that consecutive state transitions occur within $[\sigma_{\min}, \sigma_{\max}]$ of each other.

Send/Receive Events: A send event marks the time instance at which a message is put onto the link from an output port, whereas receive events indicate the change of an input port, i.e., the instance when a new message is buffered. As can be seen this is only necessary in message passing communication styles.

A *distributed execution model* defines how events at different modules are ordered and when events can occur. Due to uncertainties resulting e.g. from the link jitter $[d_{\min}, d_{\max}]$, computation time $[\sigma_{\min}, \sigma_{\max}]$, there is some freedom in the ordering of events and the occurrence time of an event. An *adversary* is responsible for choosing these (i.e., scheduling the events) in a way that does not violate the model. This freedom of choice results in different worst-, average and best-case scenarios depending on what is analyzed (e.g. time or message complexity).

An ordered infinite alternating sequence of module states and events beginning with an initial state, is called an *execution*. Every event in an execution must be applicable, i.e. allowed by the execution model and the module specification, i.e., the algorithm. For example, a computation event must not occur if it is not enabled by the current state (e.g., transition guard) of the algorithm, and a state can not be seen at an input port if it has not been received yet. Thus an event occurring later in the execution occurs later in time than a previous event (or at the same time).

2.1.2 Basic Modules

In this chapter, I provide an overview of the basic modules that are used in the algorithms considered in this thesis.

Clocks: Clocks represent functions that map real/reference time to some local (logical) time (see Figure 2.2). For example, synchronous systems contain a (single) 1-bit clock the transitions of which trigger computations (time-driven execution). A clock is referred to as a function $C : \mathbb{R}_0^+ \to \mathbb{R}_0^+$. For any time interval $[t^-, t^+] \in \mathbb{R}_0^+$ and any two time instants (t, t') and $t^- \leq t < t' \leq t^+$, a correct clock must fulfill $(t' - t) \leq C(t') - C(t) \leq \vartheta(t' - t)$.

As mentioned in the example of a 1-bit clock, digital clock implementations are finite and discrete, i.e. they can only approximate the real clock definition above, which is sufficient usually. Hence, the clocks maximum finite value must be chosen sufficiently large, s.t. the overflow is tolerable resp. desired by another module or does not occur during the operation of the system, and the rate, at which the digital clocks increment, must be sufficiently high to offer the required time resolution. Herein, ϑ is the clock ratio between the minimum and maximum clock frequency bounds ($[f_{\min}, f_{\max}]$) and $\rho = \vartheta - 1$

is the clock drift: Thus all clocks operate at some frequency of the range $[1, \vartheta] f_{\text{min}}$ in the context of digital clocks.

Note that I assume that the unavoidable clock jitter, which originates e.g. in the fact that digital clocks are discrete, is accommodated in ϑ. This is possible, since the intervals $t' - t$, considered in the analysis are large relative to the jitter.

Memory Elements/Flags: Memory elements are used for implementing buffered input and output ports. A typical implementation of a memory element provides a set and reset port. If the set resp. reset port is enabled the output port is set to some value resp. reset. At the lowest level, each memory element is able to store just one bit. Note that later in this thesis also different memory element interfaces will be used.

Timers/Timeouts: A timer comprises a reset port,a counter that is pre-set to a predefined or randomly chosen timeout value T upon reset, an associated clock, and an output port that fires within $[T/\vartheta, T]$ after the reset is disabled and stays enabled until the next reset.

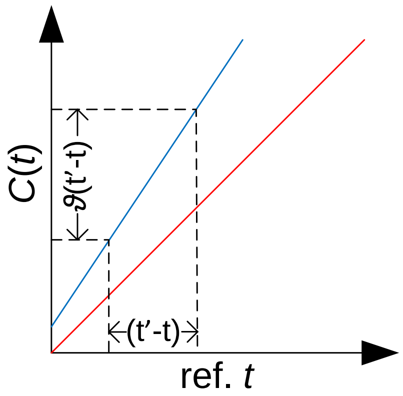


Figure 2.2: Example clock drift between a clock and the reference time.

Finite State Machines

A FSM module is specified using a set of *states* and *transitions* (edges) between those states. These transitions implement the instructions of the sequential algorithm implemented by the FSM and are usually guarded by a *transition guard* specified as an edge label: Before a transition can be executed, its guard must be enabled. In order to keep the FSM descriptions simple, all operations executed by a transition, in addition to the instructions necessary to modify the state, are also specified as (boxed) edge labels. The output of a FSM is some suitable encoding of the current state. Note that transition guards may involve the output ports of other modules (like the timers in Figure 2.1).

The example depicted in Figure 2.3 is the FSM specification of the Quick-Cycle algorithm used in the *FATAL*$^{+}$ protocol by each node(see Chapter 3.1.2). Additionally to the FSM, 3 timer modules (including clock modules) and at least one memory flag for every node is needed at each node, since they are used in the transition guards and implement buffered input ports. The FSM does not communicate all states to the other nodes, e.g., both the accept$^+$ and ready$^+$ state use the same encoded state none$^+$, whereas the state propose$^+$ uses its own unique encoded state. However, a node i will set its local memory flag module of a node j to memorize that it has seen node j in the state propose$^+$ at the input port of node i, in the current example.

The guards and actions will be explained briefly:

Timer guards: A timer guard like T_1^+ becomes true, when the output port of the timer T_1^+ fired (i.e., $T_1^+ = 1$). The timer T_1^+ is reset by the transition into the state Accept$^+$. Sometimes the timer reset state, which may be state of another FSM module too, is specified along with the timer output port (e.g., $(T_2^+, accept)$).

Voting/Threshold guards: A voting module guard like $\geq n - f \quad propose^+$ will be true at a node i if at least $n - f$ (i.e., the threshold) of node i's memory flags are set.

Actions: These are specified along with the transition guard in a box, which specifies that all memory flags of the contained states in the box are reset once the transition is performed (e.g., propose$^+$ in the example FSM).

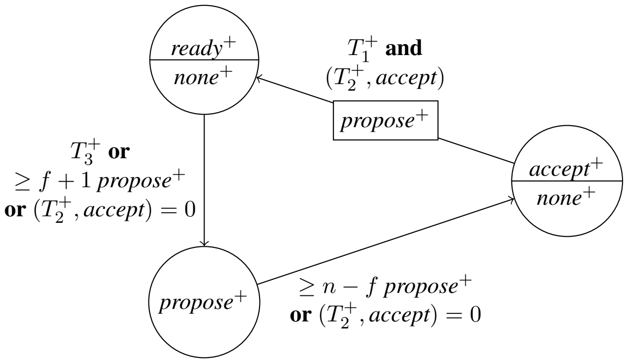


Figure 2.3: The quick cycle of the FATAL$^+$ protocol. [26]

How the FSM, i.e, its transitions are actually implemented in a VLSI circuit is of course not defined in the FSM specification.

Two basic execution styles exist for this purpose:

Event-driven (asynchronous): A state transition is performed once the transition guard leading from the current state to some successor state becomes true. Since there is no means to synchronize the evaluation of transition guards, neither w.r.t. consecutive enabling/disabling nor w.r.t. multiple transitions leading away from the current state, the potential of metastable upsets exist (see Chapter 2.3).

Timing-driven (synchronous): A state transition is performed using information of the time domain. For example, a clock module may provide the trigger for all state changes and continuously samples the transitions guards on this occasion. Although a transition guard may become true earlier, the transition is performed at the next predefined time instant.

Note that it is even possible to combine these execution styles, by using a timing-driven *Transition State Machine* (TSM) that is started asynchronously once the transition guard becomes true (see Chapter 2.5). The actions performed in the transition are triggered by the TSM clock, however. On the algorithmic/modeling level the transitions are often assumed to be executed in zero-time. State transitions executed by the real modules take some time to complete, however the minimum and maximum state transition delays must either be integrated into the link delays, leading to end-to-end delays, or must be considered in the analysis of the system separately.

2.1.3 Node execution models

In this chapter, I introduce the classic distributed computing models describing the behaviour of nodes, i.e., the described top-level modules. According to Figure 2.1 they consist of a simple FSM (with associated sub-modules) that implement some particular algorithm.

Asynchronous execution model

In the classical asynchronous execution model neither the actual link delays nor the bound of $d_{\max}$ is known. Moreover, only clocks with unknown or unbounded drift ρ may be available to the nodes, which also implies unbounded computation time bounds $\sigma_{\max}$. Usually event-based communication and event-driven executions are considered in this model. The adversary is just required to be fair, i.e., if there is an enabled computation a corresponding event occurs eventually. This implies that no node will starve.

Bounded delay asynchronous execution model

In this execution model, only bounded delay links resp. bounded delay computations exist and clocks with bounded drift ρ may be available to the nodes to implement timers and other time sensitive modules. The adversary must not violate the link delay bound $d_{\max}$, computation bounds $\sigma_{\max}$ and the clock drift ρ: All messages resp. states must be received within $d_{\max}$ and all computation events of a enabled computations must occur within $\sigma_{\max}$. Furthermore, most of these models require that no computation is infinitely fast, i.e., it always takes some time before another computation event at the same node is allowed to occur. As mentioned earlier the computation delays $\sigma_{\max}$ are integrated into the link delay bound $d_{\max}$ sometimes.

Lock-step synchronous execution model

All events occur in a well defined order in the lock-step synchronous execution model. The execution is organized in so called rounds. As the name indicates, all nodes perform their computations simultaneously in this high-level abstraction. A round is started by the adversary scheduling all pending send events. After that the adversary schedules the receive events for messages resp. provides the new states at the input ports. Finally, all pending computation events are scheduled before initiating the next round. Note that a computation event is generated even if no messages are present resp. new states are available. Hence, receiving no message in a round can also provide information. An example round and communication is depicted in Figure 2.5. Note that a lock-step synchronous execution model can be simulated using synchronized clocks in bounded delay execution models (see Chapter 2.1.5 for details).

2.1.4 Fault models

As mentioned in the introduction VLSI circuits become more and more vulnerable w.r.t. manufacturing defects, ionizing particle hits (e.g. radiation), electron migration and many more [7, 16, 22]. A system must reliable operate despite these circumstances. Thus, fault-tolerant architectures, which have been well-known in classic distributed computing for decades, are becoming an option for VLSI circuits too. In order to reason about the ability of some implementation to tolerate faults, one has to specify which faults are considered. This is the purpose of the failure model. The following concepts can be found more detailed in [6], but the most important definitions are explained shortly:

Fault: A fault is a source resp. the origin of undesired behaviour of a component or the environment that deviates from the optimal normal conditions. Examples of faults are ionizing particle hits, supply voltage drops due to power shortage or a faulty power supply and manufacturing faults (e.g., process variation).

Error: Once a fault becomes active, i.e. it occurs, it may affect the state of node resp. module such that it may lead to an inconsistent, unspecified or wrong state. In the case of an ionizing particle hit, a memory bit could flip, for example. If the affected state is a part of an arithmetic computation result, an error occurs. Depending on the algorithm, module and node implementation, such errors can be corrected after some time and vanish from the state automatically (e.g., a wrong sensor value).

Failure: Once a node resp. module violates its specification at its output ports due to an error, the node resp. module fails.

Moreover, there are to major kinds of faults distributed systems need to cope with:

Transient faults: Transient faults are short term undesired faults. For example, a power shortage or ionizing particle hits will not last forever and the operating conditions return to normal after some time. At this point, the affected part may resume correct operation as its circuits have not been damaged, albeit its state could still be corrupted.

Permanent faults: Once a permanent fault occurs, it remains persistent until it is repaired by manual maintenance usually. Such faults are a typical result of physical damage to the chip by aging (e.g., electron migration), destructive particle hits, overheating and large temperature gradients or short circuits. Sometimes parts of a chip are already malfunctioning right after manufacturing due to design errors or manufacturing process variations.

Two nodes connected via a link, where one experiences a fault, should not lead to a fault of the correct node. Hence, it is assumed that a node forms a *fault containment region*, by construction, e.g., by securing that faults can not cross node boundaries. Note that power supply, common clock and common reset among nodes are particularly problematic here. However, it can not be avoided that failures propagate across a node boundary to a successive node. In this case, a failure may lead to a fault of the successor node. Proper "shielding" of the successor node against such errors, e.g., by means of voting, can be used for implementing proper *error containment regions* (see Figure 2.1).

In this thesis only failures on the algorithm level are considered, i.e., a node fails if it does not execute its algorithm faithfully resp. at least one of its modules or ports is faulty within the execution model. Thus, a faulty node will be allowed to send messages/states from the specified value domain, but not excessive voltages or metastable signals (see Chapter 2.3). Since links may be faulty too, they are considered as a module of the sender usually.

The two most important failure behaviour classes a node may exhibit are:

Crash failure: A node fails and stops its operation from some time on (no more computation events, send and receive events). These failures are considered to be permanent usually and are the easiest class to tolerate.

Byzantine failure: A node fails and exhibits an arbitrary behaviour at its interfaces to the remaining system, i.e., the node will not execute its algorithm anymore. Many Byzantine faulty components may even collude to maximize their effect (clique) or distribute inconsistent information (i.e., "lie"). Such failures are difficult and costly to tolerate. For (lock-step) synchronous consensus [51], for example, the maximum number of Byzantine faulty nodes f a distributed system of n nodes is able to tolerate (i.e., the *resilience*) is bounded by $f \leq \lfloor (n-1)/3 \rfloor$, see Chapter 2.1.5.

Self-Stabilization

Although fault-tolerance mechanisms ensure that systems stay operational, even when a subset of nodes is allowed to fail, a high number (burst) of transient faults (e.g., a solar flare) can cause a system to become malfunctioning, because the maximum number of allowed faulty nodes may have been exceeded. In this case, there is a high risk that the system will fail completely and never recovers, especially if a system can not be maintained during its operation (e.g., satellites and air planes). Hence, recovery mechanisms are needed that allow a still functional system (that experienced a burst of transient faults only) to recover, i.e., become correct and operational again.

The simplest solution are *recovery* algorithms, which allow a failed node to rejoin the remaining correctly working system and become correct again. Since this requires enough nodes of the system to be still correct, further mechanisms are needed.

An advanced form of recovery algorithms are *self-stabilizing* algorithms that may start in any given system state, even after a burst of transient faults, and move the system into a specified and allowed state (i.e., context) using a *convergence* mechanism. In order for the system to continue the correct operation, a *closure* mechanism ensures that the algorithm remains within the specified context (in the presence of faulty nodes) and proceeds to allowed and correct successor states. Once the algorithm achieved convergence and closure the system is called *stabilized*.

Obviously, a subset of nodes may still experience permanent and transient faults, while others experience only corrupted states and could stabilize. Hence, stronger mechanisms to tolerate faults during stabilization of an algorithm are needed. For example, an algorithm is *Byzantine self-stabilizing* if it manages to stabilize the remaining corrupted but non-faulty nodes of the system, even in the presence of Byzantine faulty nodes during stabilization.

2.1.5 Distributed Problems

Clock and Pulse synchronization

A common notion of time among the nodes of a distributed system is advantageous in several aspects. Besides circumventing the impossibility of asynchronous solutions for distributed computing problems like consensus, the ability to trigger simultaneous actions and determine sequences of actions (i.e., schedules) in a distributed system is beneficial for control loops and data sampling [47].

Definition 2.1.1. *Pulse synchronization Problem. Each node provides an output pulse port* p_i *that may have either the value* 0 *resp.* 1 *(1-bit). A pulse synchronization algorithm produces synchronized pulses iff a correct node's pulse port* p_i *changes from* 0 *to* 1 *at the time instance* t *(positive transition) and any other port* p_j *of a correct node* j *either maintains already* 1 *or produces a positive transition at latest at* $t + \Sigma$*.* Σ *is referred to as the skew. Furthermore the period* P *between two successive positive transitions at a correct node is bounded by* $P \in \left[T^P_{min}, T^P_{max}\right]$*, if required.*

Since the above problem is very limited, to provide a large global time base with just 1-bit, it is necessary to solve the clock synchronization problem. Note that connecting a counter to the synchronized pulses, s.t. it increments whenever a pulse occurs, is not sufficient. Due to faults or initialization, offsets may still be present in the counter values, which may violate the clock synchronization condition. Nonetheless, pulse synchronization can be used to implement clock synchronization algorithms (see FATAL$^+$ [26]).

Definition 2.1.2. *Clock synchronization problem. Two correct clocks* C *and* C' *are called synchronized with precision* $0 \leq \pi < \infty$ *if they fulfill* $|C(t) - C'(t)| \leq \pi$ *at any instant* t*. Sometimes also the skew* Σ *is considered instead of the precision and gives a more accurate time definition:* $|C^{-1}(t) - C'^{-1}(t)| \leq \Sigma$*.*

A synchronization algorithm will try to minimize these two parameters. Obviously only small values of π resp. Σ are interesting as a trivial solution would be ∞ and for finite clocks these values should be small enough to be called synchronized, i.e., very small compared to the clock's maximum value. Although the clock definition provided earlier defines real-valued clocks, the clock synchronization definition applies to finite and discrete digital clocks too. A finite resp. real digital synchronized clock with $\pi = 1$ is called *bounded synchronized clock* (BSCLK).

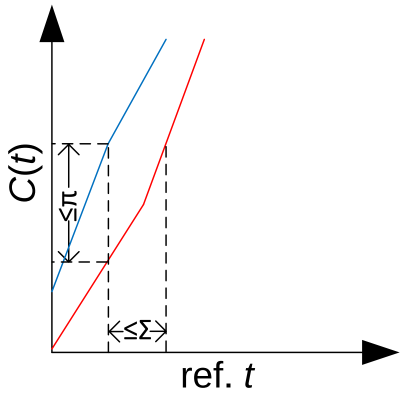


Figure 2.4: Clock synchrony bounds between to synchronized clocks.

A popular algorithm for synchronizing clocks is the fault-tolerant midpoint algorithm. It is used in the following frameworks and protocols [54, 63]. In short each node has a local free running clock. The nodes broadcast the clock values to each other and each node calculates the difference to their own value. This error values are sorted and the f smallest and largest values are omitted. The midpoint (i.e., median) of the remaining values is used to correct the local clock. Obviously, all previously mentioned uncertainties and clock drifts will affect the precision and skew.

It is known that clock and pulse synchronization needs a system with at least $n = 3f + 1$ nodes if f Byzantine faulty nodes must be tolerated [53]. At least a bounded delay execution model is needed: It has been proven that a fully asynchronous algorithm can not solve clock and pulse synchronization even if it has access to bounded drift clocks [27].

Using synchronized clocks it is possible to simulate lock-step synchronous rounds as follows: The round with the number k will be started at a node when its local clock reads kP, where P is the period or duration of a round. Due to the precision/skew of the synchronization, clock drift and the maximum delay on the links among the nodes P must fulfill the inequality $P \geq \vartheta(\Sigma + d_{\text{max}})$. The k^{th} computation event at a node occurs when it starts the $k + 1^{\text{th}}$ round, which is indicated by a node reading $(k + 1)P$ at its clock. This period ensures that no node starts a new round before all nodes finished the current round, i.e., all messages of the current round have been received. In chapter 4.3 I will provide some information about a problem that arises in real (lock-step) synchronous implementations at the input ports, which are not able to buffer messages resp. states instantly or buffering is actually included within some computation.

Consensus

Consensus [51, 55] is one of the most important distributed computing problems to be solved. Each node is provided with a local input of a given value domain. These local inputs may differ. The task of the consensus algorithm is to make a single and irrevocable output decision at every correct node, at the time the algorithm terminates at that node, which is equal to the output at all correct nodes and was the input of a correct node. A consensus algorithm then reaches agreement

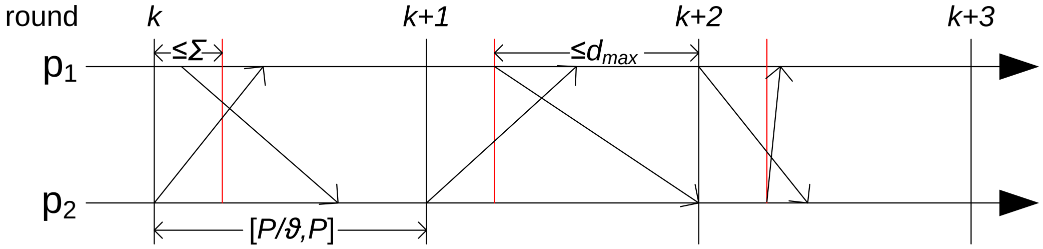


Figure 2.5: Example synchronous round communication between two nodes p_1 and p_2. The red line shows the skew bound and a new round initiated by the first node is indicated by a black line.

in the presence of input deviations among the nodes, and either probabilistic or deterministic algorithms can be used. For example, in computer systems distributed or replicated databases must be kept consistent. If there is a deviation between the databases a consensus algorithm must be used for agreeing on multiple input values among the replicated data.

The most basic form is binary consensus. The inputs and outputs may obtain either of the two values 0 and 1. As someone can imagine the process involves some form of voting.

Definition 2.1.3. *Consensus Problem. Every node has an input variable x_i and an output variable o_i, which is unassigned initially. x_i and o_j are elements of a given value domain $\mathcal{I}$. A solution algorithm must fulfill the following assumptions:*

Exact Agreement: *The algorithm computes $o_i = o_j$ at any correct nodes i and j.*

Validity: *The computed output o_i at correct node i must be the input of some node j, i.e. $\exists j : o_i = x_j$.*

Termination: *The algorithm terminates at every correct node and the output is irrevocable.*

Other stronger or weaker definitions exist. It is known that consensus needs a system with at least $3f + 1$ nodes and $f + 1$ synchronous rounds (in a lock-step synchronous round model) if f Byzantine faulty nodes must be tolerated [32, 51, 55]. At least a bounded delay execution model is needed: It has been proven that a fully asynchronous algorithms aren't able to solve consensus with Byzantine faulty nodes [27]. Two well-known (lock-step) synchronous consensus algorithms are the Phase-King [10] and Phase-Queen consensus [11] algorithm, see Chapter 3.2. The first features optimal resilience $n > 3f$ using $4(f + 1)$ rounds and the latter needs at least $n > 4f$ nodes, but is able to solve it with a termination time of 2(f+1) rounds.

It is logical that the same node and model lower bound applies as in clock synchronization, since consensus is able to solve clock synchronization with $\pi = 0$, if the clocks are used as inputs and outputs. Note that due to the uncertainties (e.g., link delays) there will always be a skew and exact agreement can only be achieved in a high-level model like the lock-step synchronous round execution model, where the skew is neglectable.

2.2 Digital Logic

The following chapter will describe digital design and its problems, when implementing algorithms using VLSI circuits, and typical synchronous and asynchronous solution approaches.

2.2.1 Boolean Circuits

No matter what algorithm must be implemented one basic form of asynchronous circuits can be found on any chip: *boolean circuits*. A boolean circuit, also referred to as *combinatorial logic*, is a VLSI implementation of a boolean function with multiple input and output ports that may take one of the values $1/\text{TRUE}$ resp. $0/\text{FALSE}$. The basic forms of boolean circuits are not stateful, i.e., their output does not depend on the past input and just map the current input state to the specified output state. A boolean circuit is implemented by using standard boolean *gates* that represent the functions $\wedge, \vee$ and $\neg$ usually. Depending on the vendor other and high-level complex functions like *XOR,NOR, NAND*, arbitrary boolean function with 3 or 4 inputs, binary addition, multiplexers and multipliers might be already available to the designer without the need to implement them from scratch. Figure 2.6 depicts the boolean circuit for the function $(\neg S \wedge A) \vee (S \wedge B)$, which implements a multiplexer using the input port S to select between the input ports A and B.

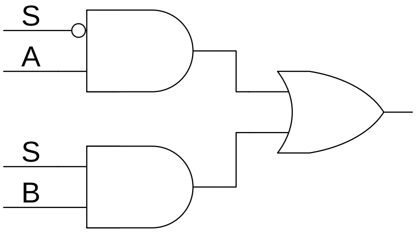


Figure 2.6: A 2-input multiplexer (MUX).

2.2.2 Time and Value domain issues

Although the boolean circuits and functions are simple to specify and analyze, this description contains no concept of time. As mentioned before any computation and communication causes delays. In this case the computations are performed by the boolean circuit and the communication is between the *source* that provides the input and the *sink* that uses the output of the circuit. Besides, someone might notice that the boolean circuit of Figure 2.6 is not elementary, i.e. the computation of the boolean function involves smaller sub-modules (e.g., standard boolean gates) that can be assumed to be elementary.

As mentioned before, such increased granularity gives the adversary more strength in asynchronous and bounded delay execution models as it may choose the execution order of computation events depending on the actual link and computation delays. These delays are often referred to as gate and propagation delays in the VLSI context. They are prone to jitter and the delay bounds depend on many factors like the temperature, production process, lifetime, transistor driver strengths and supply voltage noise. These problems arise from the fact that transistors are analog devices both in time and value (voltage) and the digital abstractions map their behaviour to discrete time and value domains. The physical world has its limits, however, in that no information may be propagated faster than the speed of light and that currents need time to build up the voltage at capacitances, i.e., charge them.

Signals

The voltage levels that represent a logical 1 resp. 0 are the supply resp. ground voltage, but the actual voltage output by any logic function will not exactly match these depending on production and environment effects. As a result, the discrete logic values are represented by a voltage range instead of a single discrete value. The region separating these ranges is a forbidden region, in the signal range, which may only be used during signal transitions, because it can not be guaranteed that every gate will map the voltage into the same specified region. Figure 2.7 depicts the problem with two inverters and the same characteristic curve, but slightly offset thresholds. The forbidden range is determined by selecting two input voltage ranges for logic level 0 resp. 1 represented by the thresholds V_{IL} resp. V_{IH}. The transfer function, $f : U_{\text{in}} \rightarrow U_{\text{out}}$ of an inverter must satisfy $f(V_{IL}) \geq V_{IH}$ and $f(V_{IH}) \leq V_{IH}$, for example. Note that even safe thresholds require the output to cross the forbidden region to reach the opposite logic level, however. In this case the discrete computation events occur when U_{out} crosses V_{IH} in a rising transition $\uparrow$ (resp. V_{IH} in a falling one $\downarrow$).

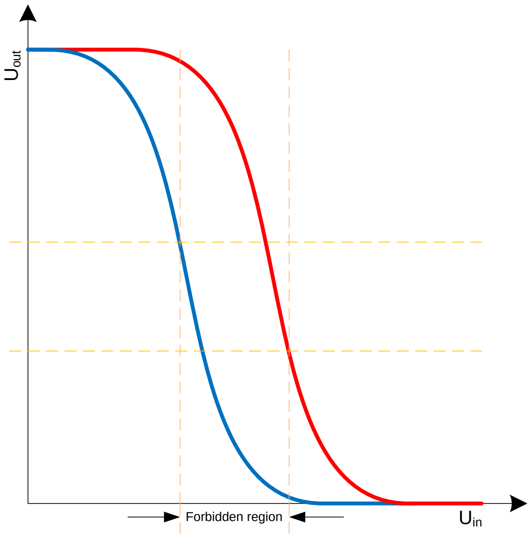


Figure 2.7: Example of two inverters with a slightly different threshold and the forbidden an safe regions.

Glitches

The possibility of reordering the gate computation events due to the non-zero propagation and gate delay jitter may cause undesired signal transitions at the output of a boolean circuit, called *glitches* [73], which would not be observed for an ideal (zero-delay) gate.

Every real boolean circuit might produce glitches, but the actual behaviour depends on the input pattern. For example a simple two-input **AND**-gate with inputs $a = 1$ and $b = 0$ is prone to produce glitches if a falling transition $a \downarrow$ occurs at a when a rising transition $b \uparrow$ occurs at b. If $b \uparrow$ is received slightly earlier by the gate it begins the transition to 1 instead of remaining at 0. The existence of such an input pattern is called a *hazard*. Just its existence does not imply that a glitch will ever occur, since this depends on the real delays and only when a critical pattern is actually applied by the source, but a priori glitch-free circuits are of course desirable.

Glitches may be categorized as follows (see Figure 2.8):

Static 0/1: A state change of the output occurs although the output should remain 0 resp. 1.

Dynamic ↑/↓: Although the output should perform just one rising (↑) or falling transition (↓) exactly, one or multiple intermediate pulses may occur.

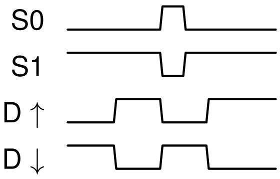


Figure 2.8: Possible behaviour of a single output circuit with static or dynamic hazards.

Standard binary truth tables and binary circuit analysis does not reveal these hazards and its enabling inputs. Hence, a 9-valued logic extended truth table is used instead with the symbols 0, 1, $\downarrow$, $\uparrow$, $S0$, $S1$, $D\uparrow$, $D\downarrow$ and $*$. Herein, $*$ represents any signal, S/D the above glitches and $0/1$ the standard stable logic states (see Table 2.1).

The attentive reader will notice that only glitches at the input resp. transitions of opposite *polarities* (i.e., ↑ and ↓) can cause glitches at the output of a standard gate. Unfortunately, even simple circuits with *multiple inputs and a single output* (MISO) can produce glitches if a *single input changes* (SIC) and the other inputs remain stable. The problem is that a signal may travel along multiple paths (i.e., forks) that are recombined by a single gate afterwards. Figure 2.9 shows an example of the function $a \vee \neg a$, which should be always 1. If the path through the inverter is assumed to be significant and the **OR**-gate is fast, a negative transition $\downarrow$ on a will cause a $S1$ glitch at the output, since the inverter output is still 0 and the $\uparrow$ arrives too late. An equivalent problem exists with the **AND**-gate.

Although nobody would implement such circuits directly, similar structures implicitly exist within many implementations (e.g., in the MUX in Figure 2.6). If $a = 1, b = 1$ and $s = 0$ the MUX output is 1. Obviously, the output of one **AND**-gate is $\neg s$ while the other is s, which is recombined by the **OR**-gate.

OR	**0**	**1**	$\downarrow$	$\uparrow$	**S1**	**S0**	$D\uparrow$	$D\downarrow$	*
0	0	1	$\downarrow$	$\uparrow$	S1	S0	$D\uparrow$	$D\downarrow$	*
1	1	1	1	1	1	1	1	1	1
$\downarrow$	$\downarrow$	1	$\downarrow$	S1	S1	$D\downarrow$	S1	$D\downarrow$	*
$\uparrow$	$\uparrow$	1	S1	$\uparrow$	S1	$D\uparrow$	$D\uparrow$	S1	*
S1	S1	1	S1	S1	S1	S1	S1	S1	S1
S0	S0	1	$D\downarrow$	$D\uparrow$	S1	S0	$D\uparrow$	$D\downarrow$	*
$D\uparrow$	$D\uparrow$	1	S1	$D\uparrow$	S1	$D\uparrow$	$D\uparrow$	S1	*
$D\downarrow$	$D\downarrow$	1	$D\downarrow$	S1	S1	$D\downarrow$	S1	$D\downarrow$	*
*	*	1	*	*	S1	*	*	*	*

AND	**0**	**1**	$\downarrow$	$\uparrow$	**S1**	**S0**	$D\uparrow$	$D\downarrow$	*
0	0	0	0	0	0	0	0	0	0
1	0	1	$\downarrow$	$\uparrow$	S1	S0	$D\uparrow$	$D\downarrow$	*
$\downarrow$	0	$\downarrow$	$\downarrow$	S0	$D\downarrow$	S0	S0	$D\downarrow$	*
$\uparrow$	0	$\uparrow$	S0	$\uparrow$	$D\uparrow$	S0	$D\uparrow$	S0	*
S1	0	S1	$D\downarrow$	$D\uparrow$	S1	S0	$D\uparrow$	$D\downarrow$	*
S0	0	S0	S0	S0	S0	S0	S0	S0	S0
$D\uparrow$	0	$D\uparrow$	S0	$D\uparrow$	$D\uparrow$	S0	$D\uparrow$	S0	*
$D\downarrow$	0	$D\downarrow$	$D\downarrow$	S0	$D\downarrow$	S0	S0	$D\downarrow$	*
*	0	*	*	*	*	S0	*	*	*

Table 2.1: 9-valued logic analysis results of the standard **OR** and **AND** logic gates [73].

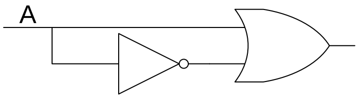


Figure 2.9: Example structure inside circuits that may cause SIC glitches for **OR**-gates.

If a circuit is free of any such structures, it is SIC glitch-free. The good news is that any boolean function can be made SIC glitch-free using Huffman circuits [73], see Figure 2.10 for the Huffman MUX implementation. It works by inserting the required redundant terms that mask the problematic signal transitions. In the MUX case, the term $(a \wedge b)$ must be added. The required terms can be found from the *Karnaugh-Veitch*-diagram (see [73] for more detailed information). The only problem is that it is very difficult to establish SIC patterns in every case.

Apparently, it is not possible to build glitch-free circuits in general *multiple input change* (MIC) resp. *multiple output* (MO) scenarios easily, since this would require to match all delays and forks such that any input change arrives almost simultaneously at the successive gates and no old inputs are present. In such cases, restricting the source input pattern, matching the delays or a different circuit implementation might be able to solve the problem.

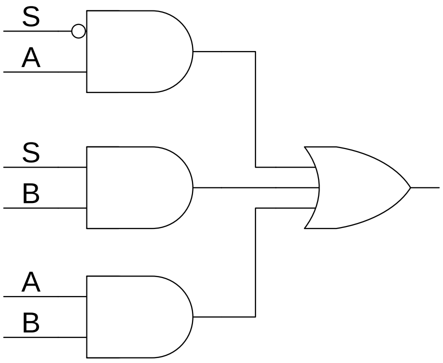


Figure 2.10: A 2-input Huffman multiplexer.

2.2.3 State

The standard boolean circuits compute the output for a given current input, no matter what its past inputs were (i.e., its history). In order to implement algorithms (i.e., FSM) and more complex functions, memory elements are needed to capture some sort of state. Their output will not only depend on its current inputs, but also on its state. Stateful circuits must contain feedback paths, i.e., parts of the output of a boolean function are used as inputs too.

One common memory element is the *SR-latch*. Matching the memory element definition in the previous chapter, it provides a set and reset input port (S and R) and at least one output port Q that is one if the memory element is set and zero otherwise. Figure 2.11 (left) shows an example implementation that also provides an enable port (E). Figure 2.12 provides an example timing diagram.

Another very popular memory element is event-triggered and called a *D-Flipflop* (DFF). It provides two input ports D and E and at least one output port Q. One way to build it is by using a master-slave SR-latch structure, see Figure 2.11 (right). If E is 1, the slave stage holds its output Q no matter what happens at the input, and the master stage is transparent and forwards the current value of D to the input of the slave stage. Once E changes to 0, the enable of the second stage is active and sets its output Q to the output of the master stage, while the master stage holds its output (i.e., the slave stage is transparent). Hence, in the current example, it sets Q to the value of D, once a falling transition on the input port E occurs. An example timing is depicted in Figure 2.13.

Note that the memory elements require some time to reset resp. set the output and, in particular, the feedback loop appropriately. Thus glitches, activating set and reset at the same time and too short set/reset pulses could violate the respective timing requirements and the memory element might not be able to store the input safely, which can lead to undesired behaviour and incorrect states (see Chapter 2.3). In case of the DFF, the resulting timing constraints are called the *Setup/Hold*-requirements. A new input D must be stable at least for the setup time before the enable's falling transition occurs, and must remain stable at least for hold-time after the enable performed the falling transition.

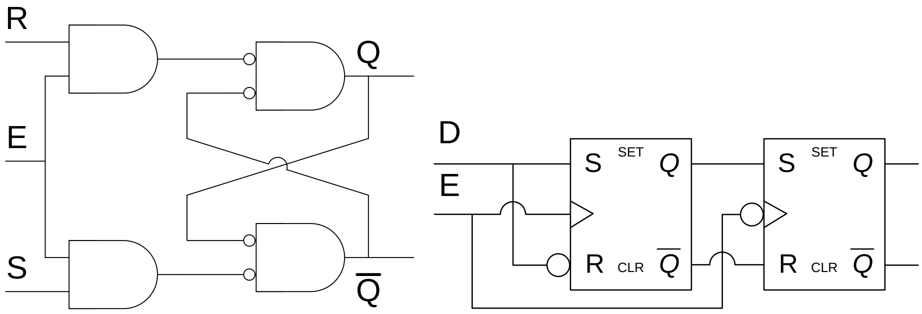


Figure 2.11: Implementation of an SR-Latch with enable (left), and a DFF with enable (right) using a master-slave SR-latch structure. The slave stage captures the output of the previous stage on the negative transition.

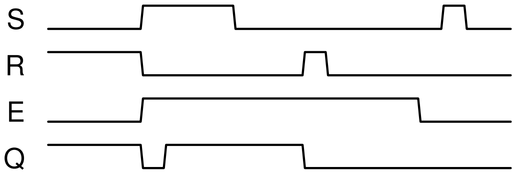


Figure 2.12: SR-latch example timing behaviour. Due to delays the outputs are not able to change instantly

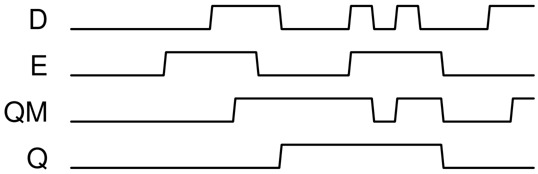


Figure 2.13: DFF example timing behaviour. Due to delays the outputs are not able to change instantly

2.2.3.1 Communication

Boolean circuits and memory elements provide state outputs only, i.e., they do not implicitly specify when an output is valid and consistent with the current input and may be safely captured by successive circuits. This creates a *fundamental design problem* (see Figure 2.14) for every circuit: How does the source know when it is safe to issue new data, and how does the sink know when it is safe to capture data? As a result of the non-zero gate and link delays, the source is not allowed to provide a new input until the sink has captured the old input, and the sink is not allowed to use the data until all bits are *valid* (i.e., stable 0 and 1) and *consistent* (i.e., all bits belong to the same data word). Chapters 2.4 and 2.5 of this thesis will be explaining some popular design principles for solving this problem. More detailed descriptions can be found in [19, 20].

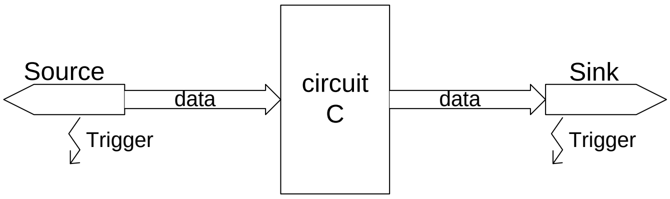


Figure 2.14: The fundamental design problem: How to generate safe issue and capture triggers? [20].

2.3 Metastability

As indicated above, input and timing specification violations (as well as forbidden input voltages) occurring during the operation of a memory element may result in a violation of its output specification either in the time or value domain. In the extreme case it might not reach one of the two stable states 0 or 1 within bounded time, i.e., it may become metastable [56]. Note that it is even uncertain which of the stable valid output states will be reached eventually. If a subsequent memory element captures the metastability, i.e., it has been propagated through the intermediate boolean logic, it becomes upset [46, 48].

Take the DFF as an example. A new value must have propagated to the slave stage already and lead to a stable output Q when the falling transition on the enable occurs. If the enable transition occurs before the feedback loop in the slave hes settled to the new value, Q may stay within the forbidden output region arbitrarily long.

In any asynchronous or synchronous closed loop system, almost all such timing violations can be avoided, but systems are never closed completely: Real systems always contain interfaces to the physical world, other components that are not able to comply with the setup/hold requirements, or possibly faulty components.

2.3.1 Metastability modeling

In order to understand the details of a metastable upset, consider the SR latch from Figure 2.11, which can be used for implementing the stages of a DFF.

Once the enable is deactivated, only the feedback loop, consisting of two inverters, remains. It represents a bi-stable memory element, which is frequently used in any digital system. Figure 2.15 shows this inverter loop and the input to output transfer function for both inverters. Besides the stable operating points $U_O = 0$ and $U_O = 1$, there exists also an inevitable metastable point $U_O = U_M$. Note that the gain A in the region around the metastable point can be reasonable approximated by a perfect linear amplification. In theory, the whole loop would come to rest at this point, without ever leaving it. Due to noise and other physical quantum effects, it is very unlikely to never resolve, but it may take long. Moreover, the intermediate output voltage U_M may cause successive stages to output intermediate voltages too.

Actually, one can distinguish between two forms of metastability [46, 69]:

Creeping: The output is settled within the forbidden region, close to U_M, from where it creeps,

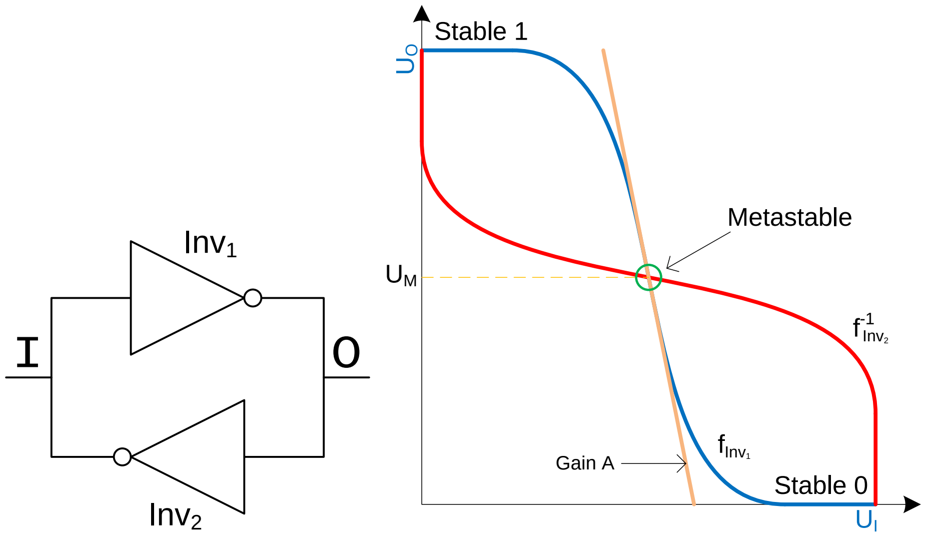


Figure 2.15: Inverter loop within a storage element and the input to output voltage relation.

i.e., moves slowly toward either 0 or 1, instead of performing a fast rising or falling transition. The closer the settling voltage is to U_M, the longer it will take for the storage loop to resolve.

Oscillation: The output oscillates significantly between the stable high and low state or within the forbidden region. This behaviour may be caused by input pulses with enough energy to put the output of the first inverter into a new state, but which is not long enough to set second the inverter before it vanishes. This produces a pulse that will circulate through the loop and eventually dies out, namely, when its duty cycle becomes sufficiently asymmetric. Note that oscillatory metastability may end up in creeping metastability as well.

For a quantitative analysis of the metastable behaviour of the inverter loop, the inverter model shown in Figure 2.16 is used. It consists of an ideal amplifier, a pure delay and a slope limiter modeled by an RC-circuit. The pure delay models the propagation delay, whereas the slope limiter determines the time needed to build up the output voltage, i.e., charge the capacitances.

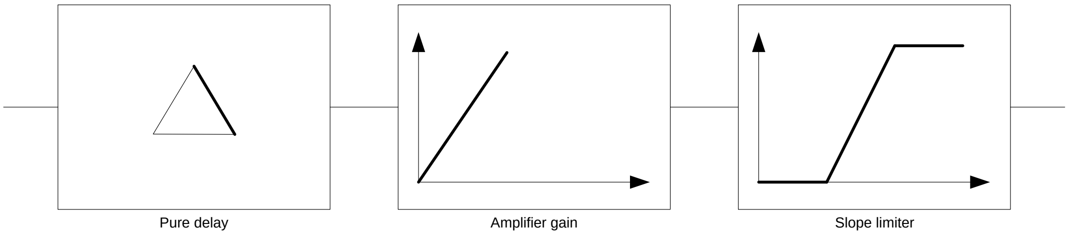


Figure 2.16: An analog model of a single inverter use for modeling metastable behaviour of a feedback loop.

As mentioned above, metastability can be triggered via time domain and via the value domain. If a pulse is introduced into the loop, which is able to charge the capacitance modeled by the first inverter, but does not charge the second inverter's capacitance sufficiently before the pulse has vanished, can create oscillatory metastability [69]. The condition for oscillatory metastability is $\delta > RC/A$, where δ is the pure delay and RC/A the inverse of the gain-bandwidth product. After the oscillations stops, the loop is within the stable state or still within the forbidden region and starts to creep.

Intermediate input voltages (close to U_M) that have fully propagated through the inverter loop always create creeping behaviour, i.e., stays within the margins of the forbidden region. Note that this holds true even when the pure delay δ is zero. Metastability modeling hence primarily addresses creeping metastability, and many experiments and simulations exist and the extracted memory element technology parameters (e.g., gain and bandwidth product) match the observed upset rate and resolution time very well. Note, however, that delays are increasing relative to the gain-bandwidth product, which makes oscillatory metastability more dominant in the future.

2.3.2 Metastability mitigation

If a circuit becomes metastable, it obviously behaves incorrect for some unpredictable time. Moreover, metastable behaviour may propagate to subsequent circuits. Oscillatory metastability might cause even more harm in (asynchronous) circuits, since the pulse transition could be taken for regular signal transitions.

Fortunately, deep metastable upsets are rather rare and metastability propagation can be made sufficiently improbable, by means of metastability filters.

The following standard equation is used to calculate the expected upset rate (mean time between upsets) of a single memory element [46]:

$$MTBU = \frac{1}{\lambda_{dat} f_{clk} T_0} e^{\frac{t_r}{\tau_c}} \tag{2.1}$$

Herein, $\tau_c = \frac{RC}{A-1}$, T_0 is time needed to cross the forbidden range (technology parameters), f_{clk} represents the enable frequency, t_r the granted resolution time and λ_{dat} the expected input data rate.

In order to compute the upset rate of a chain of memory elements, one has to plug in $t_r = \sum_{i=1}^{n} t_r^i$. where t_r^i is the available resolution time of stage i. In the case of synchronous state machines (see Chapter 2.4), the resolution time of a stage is computed by subtracting all (combinatorial) delays and setup times from the clock period (enable period). The results in [36] show similar results for other memory elements like the *Muller C*-gates (see Chapter 2.5).

The chance metastable upsets can be reduced by means of synchronizers, i.e., for synchronous designs and elastic pipelines [76] for asynchronous ones. The primary purpose of a synchronizer is to synchronize some arbitrary data input with a clock signal. It usually consists of a chain of latches as DFFs, all enabled by the clock signal. Efficient filters for creeping metastability can be built by means of *Schmitt-Triggers*, which are buffers with a hysteresis in their threshold voltages [65]. They effectively map U_M to a stable output of 0 or 1, hence prohibit metastability propagation. This mechanism trades performance for safety (i.e., delay), however.

Although, metastability can be avoided in (many) fault-free distributed system executions using (asynchronous) closed loop communication, module and node failures can lead to metastability. Obviously, any fault (e.g., ionizing particle hits) can be the origin of metastable upsets. Hence, a node may fail exerting a metastable behaviour instead of Byzantine behaviour at its interfaces, but Byzantine failures were restricted to states resp. messages of the specified algorithm output domain (e.g., stable 0 or 1). As mentioned above, metastability may output intermediate voltages of the forbidden region and produces oscillating pulses, for example. Thus, metastable failures are even more severe than Byzantine failures, since no (deterministic) fault containment is possible. Note that (creeping) metastability is one possible source of the typical inconsistent Byzantine behaviour, i.e., the invalid metastable output signal may be interpreted as a stable 0 by some subsequent node but as stable 1 by another node.

However, it is (often) assumed that nodes of a distributed system fail independently of each other. Therefore, it is necessary to reduce chance of metastability propagation and employ Schmitt-Triggers resp. synchronizers and elastic pipelines as probabilistic fault-containment mechanisms. Other mechanisms, like voting modules, may be even able to mask metastable upsets, and treat them like ordinary Byzantine faults. In this case, there must be sufficiently many inputs from other correct nodes that arrive in time, and are not metastable.

Metastability propagation is also a problem in asynchronous fault-tolerant clock generation schemes like [36], due to (Byzantine) faulty nodes, which can put the system into an invalid state. If no fault-tolerance, error containment, masking and recovery mechanisms are applied a system might fail due to possible continued inconsistent states. Metastability is another reason, why self-stabilizing algorithms are so important.

2.4 Synchronous circuits and state machines

The dominant approach to solve the issue of validity and consistency, introduced at the end of Chapter 2.2.1, is to design the circuit according to the synchronous design principle. Within a synchronous design there exists one control signal, the clock signal, indicating the launch of new input data and output data (time-triggered execution). It is a periodic 1-bit signal, the transition of which indicates when an output is valid and consistent in the time domain. A full period is called a clock *cycle*. Any algorithm implemented as a synchronous design, is being executed in the lock-step synchronous round model.

Figure 2.17 shows a typical multi-stage synchronous circuit. In every state, a new input must be processed by the boolean function before the next rising edge on the clock occurs. The input is issued at (some) previous clock transition, usually. Due to the delays on the paths in the combinatorial circuit, glitches may occur and the output is either not valid or inconsistent until all intermediate results have propagated through the circuit and stabilized. The period of the clock ensures that the triggering edge only occurs when the output has stabilized such that a DFF may be used to safely capture the data at the next clock transition. Additionally, the clock must ensure that no new data arrives before the current one has been stored by a DFF. In the example shown in Figure 2.18, the grey data parts indicate when the data is an invalid or

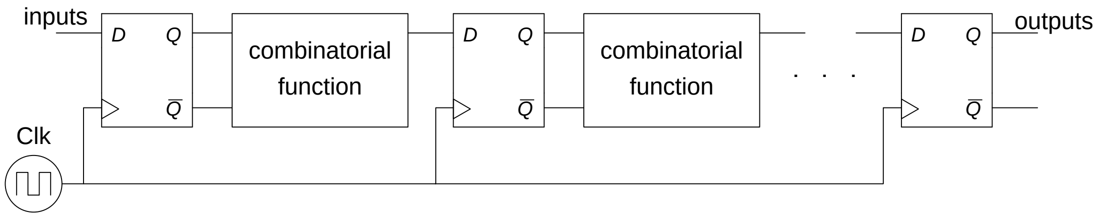


Figure 2.17: A typical synchronous pipeline using DFFs and combinatorial boolean functions to process the data. A DFF is called a stage in this context.

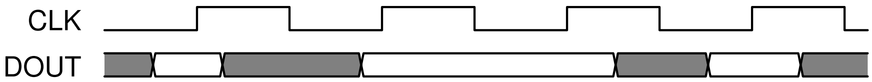


Figure 2.18: The data output of a stage changes only after every successor component has captured the current valid and consistent data indicated by the clock signal. In this case the rising edge is used as trigger.

inconsistent output of the combinatorial circuit, while the white part is valid and consistent. As also shown in this example, there is no need for the data output to change every cycle, e.g., if the input stays the same or the output does not change, and no glitch occurs despite a changing input. Note that it even takes some time before a new input causes the combinatorial function's output to become invalid or inconsistent , i.e., it still reflects the old output, as can be seen in the example.

2.4.1 Pipelining

A complex combinatorial function, like multiplication or division, can be split into smaller combinatorial functions, e.g., by inserting DFFs on intermediate links like in Figure 2.17, to reduce the combinatorial and link delay between two DFFs. Herein, a pair of DFF and the combinatorial function processing its output is called a *stage*. As a result, the number of cycles it takes to complete the computation increases to the depth of the pipeline (number of stages). In the best case a pipeline stage influences a successor stage only. Hence, pipelining is similar to conveyor belts inside factories. Each stage performs a small part of the overall specification and determines the input state of the next stage.

Sometimes, components or pipeline stages need to share some resources like communication channels and so on. Then an arbiter module is needed to control the access to these resources and only grants one access request at a time per resource.

Pipelines are very common to process instructions in a CPU to increase the operating frequency and average throughput, for example: A simple CPU pipeline contains stages for fetching, decoding and processing an instruction, as well as a stage for storing the results.

Assume that someone was able to reduce the delay of 10 seconds of a combinatorial circuit to 1 second, using a 10-stage deep pipeline, which allows to operate it using a clock with a period of 1 second instead. An input at the pipeline still needs the same time to be processed completely

(10 cycles), but the remaining systems runs at the higher clock frequency too, which can be a significant improvement. Moreover, if the pipeline stages are independent, i.e., they do not need to wait for the completion of a successive stage, a new input may be provided every cycle. This implies that after the first element is available at the output of the pipeline the next output will be available one cycle later. In the current example, the second input will reach the output after 11 seconds using the pipeline instead of 20 seconds. Hence, the overall throughput has been increased by a factor of 10 too, but only if a new input can be provided every cycle on average. Therefore, if a new input can only be provided to the pipeline once the last input is available at the output of the pipeline the throughput stays the same. Another example, which illustrates the latency very well, is that of a whiskey distillery. When it first starts distilling whiskey a batch is stored for some years before it is sold, e.g., 10 years. If the distillery continues to distill whiskey every year, it will produce its first batch after 10 years and from now on a batch is produced every year instead of every 10 years. The pipeline has a latency of 10 years and even parallel storing stages are needed.

2.4.2 Clocking

A crucial part of any synchronous design is clock distribution [35]: The clock must be distributed to all DFFs such that clock transitions occur at the same time, at all DFFs. Note that excessive skew could result in setup and hold violations of the DFFs, which may require to decrease the clock frequency or to change the implementation. The typical *skew*, i.e., the maximum difference of clock tree delays between any two DFFs, within the clock trees of modern chips is just a few (hundred) ps at most to achieve high clock frequencies GHz-region without violating timing constraints. Typically, a carefully engineered clock tree is used for this purpose (see Figure 2.19) [67, 68]. Each leaf in the tree or sometimes a node of the tree is an exit point that is connected to a DFF. If the amount of DFFs and logic becomes larger the routing of the clock tree becomes more complex. After all, the problem of validity and consistency has been moved into the delays of the clock tree, as it must ensure that the race condition between the clock and the data never becomes a problem. Furthermore, larger clock trees require the insertion of buffers/inverters for signal conditioning and skew compensation, which consumes much energy and create considerable power dissipation. Moreover, the clock continues to toggle even if there is no new data to be processed. Besides, the clock tree may act as an large antenna receiving noise from the environment [57].

Consequently, the clock tree is one of the most energy consuming parts of a modern high-performance VLSI circuit and is a single point of failure [40, 52],. Improvements of VLSI technology further exacerbate these problems: Timing margins (period) shrink even further due to increasing frequencies, chips contain even more logic and long clock trees also suffer from many parasitic effects (i.e., resistance, inductance and capacitance) and hence act as low pass frequency filters, etc. Power dissipation also becomes worse as the dissipated heat must be moved to the heat spreader through multiple layers, including the clock tree layer.

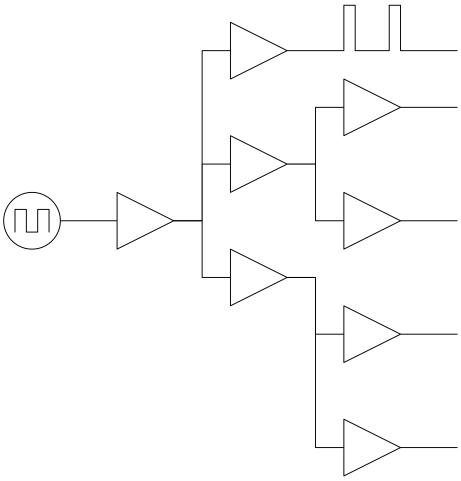


Figure 2.19: An abstract example of a clock tree. Buffers are inserted to keep the fan-out (capacitance) low or to add delay. Another possibility to adjust delays is the wire length. [35]

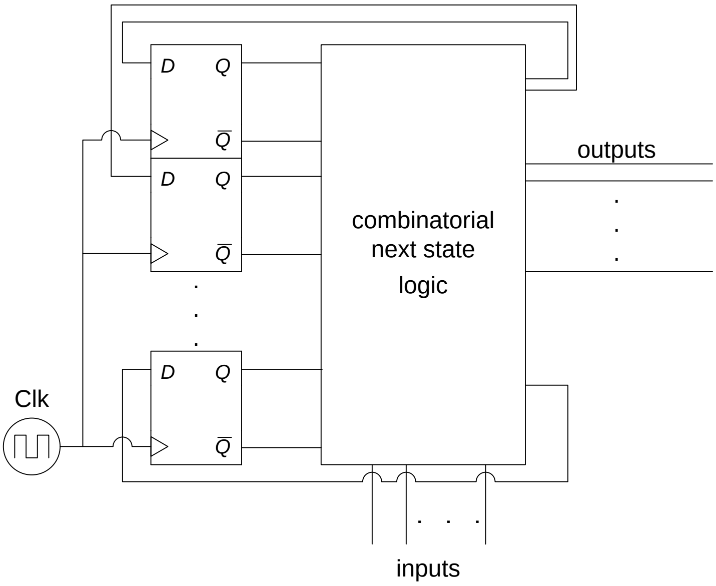


Figure 2.20: Abstract representations of synchronous circuits for general FSMs.

2.4.3 FSM

Figure 2.20 shows how a FSM is usually implemented in a synchronous design. The DFFs, which are used to hold the current state of the FSM, are referred to as *registers*. For each bit of the state a separate register is used. The registers feed the combinatorial function, which generates the next state and the outputs of the FSM module. Apart form the FSM state, the combinatorial functions receive module external inputs via the FSM input ports. As mentioned in Chapter 2.3, these inputs must be either synchronized to clock domain of the FSM, if that is not already case, or the inputs are generated by other modules of the same clock domain. Hence, a combinatorial function implements the next state logic, i.e., from the current state and its inputs it computes a new value for each of the registers. The transition to a new state, i.e., execution of an algorithm, is controlled by the clock at either every rising resp. falling edge.

Depending on whether all FSM module outputs are connected to further combinatorial functions resp. to register outputs directly, the FSM is called Mealy-FSM (as in Figure 2.20) resp. Moore-FSM. Obviously, all Moore-FSM outputs are glitch-free and due to the missing combinatorial function, no additional combinatorial delay is introduced. One drawback of the Moore-FSM remains, i.e., it is not able to generate an output for the current input within the same clock cycle. The number of memory elements to implement the registers and the next state logic can become rather large. Unfortunately, the logic depth of the latter determines the delay and, hence, the achievable clock frequency.

In order to speed-up a synchronous FSM, the clock frequency must be increased, which requires decreasing the maximum delays between the memory elements, i.e., the delays of links and the combinatorial next state logic. Within the general FSM implementation (shown in Figure 2.20), a single DFF output may influence many other DFF inputs via the next state logic. Such increased influence can result in much larger delays than an implementation in which a DFF only influences a few other DFFs. An approach to solve this problem is to use pipelining, i.e., breaking up some deep next state logic into multiple stages with interspersed DFFs. Herein, each stage represents a reduced FSM. The only problem is that the hardware effort may increase compared to the general FSM implementation, due to additional combinatorial logic and DFFs. The attentive reader may have noticed that only the clock skew between DFFs influencing each other via a combinatorial circuit is of interest. A pipeline is able to relax the skew constraints. In the ideal pipeline, only the skew between two adjacent stages, as shown in Figure 2.17, must be small, the overall skew could be larger.

As a result, such a pipeline is still a FSM, since all stages represent FSMs as well, but the combinatorial logic depth (delay) between two DFFs has been decreased. Any modern CPU is based on synchronous circuits and contains pipelines of lengths of up to 14 or more. In order to meet the performance requirements and increase the clock frequencies the production process (e.g., feature size) and the architecture are enhanced every year [33].

To sum up, the most important advantage of synchronous designs it is relatively easy to describe and develop algorithms. It is almost possible to immediately transform sequential problems into synchronous finite state machines using the clock to perform the steps and still many different design possibilities exist (e.g., pipelining and combinatorial circuit implementation). The designer does not need to care about any glitches, due to the chosen clock period, and may always assume to have consistent and valid inputs or outputs. The clock ensures that the receive events and computation events occur in a well defined order (triggered by the rising resp. falling clock transition). Even simulations are much easier, since the clock tree's delays can be assumed to be 0 (lock-step round simulation), and thus the circuit's functional behaviour can be simulated at a much higher level without caring about any delays or glitches.

2.5 Asynchronous circuits and state machines

Until now, only a single global control signal, the clock signal, distributed to all stateful components has been used, for controlling data exchanges. This chapter will provide an overview of design alternatives including multiple clock signals, handshaking, self-timed asynchronous circuits, hybrid forms of synchronous and asynchronous state machines and asynchronous communication. The overall goal is to remove the need for global clock signal, i.e., to implement circuits at least partially asynchronous.

2.5.1 Circuit classification

Data validity and consistency of a circuit's outputs is determined by the delays of links and logic, which can be classified as follows (based on the example in Figure 2.21):

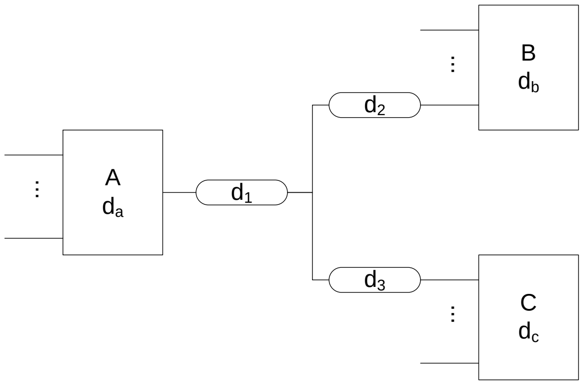


Figure 2.21: A typical fork found in any combinatorial function or feedback paths [73].

Speed Independent: If the wire delays d_1, d_2 and d_3 can be assumed to be 0, but the module resp. gate delays can be arbitrary, the circuit is called speed independent (SI).

Delay Insensitive: If all delays can be arbitrary, the circuit is called delay insensitive (DI). Delay insensitive circuits are the most robust circuits possible, but are not always implementable.

Quasi Delay Insensitive: If all fork delays d_2 and d_3 must be equal (called an isochronic fork), the circuit is quasi delay insensitive (QDI). In the clock tree of a synchronous system, this would require the clock skew to be 0 at all nodes of the same level. It is of course impossible to make delays perfectly equal in practice, but the error can be sufficiently minimized.

Self Timed: Any asynchronous circuit that is not within one of the above classes is self timed (mixed circuits).

Note that the above delay assumptions are only required in basic modules, to ensure no glitches or metastability can occur due to the link, memory and gate delays. A circuit that is built from these modules can always be made DI, by using proper asynchronous communication protocol in-between. See [43, 73] for further details.

2.5.2 Signal Transition Graphs and Muller C-gates

Many asynchronous circuits can be described by means of *Signal Transitions Graph*(STG). Like a FSM, a STG specifies the sequences of allowed transitions at the input ports and the output ports transition sequences. I will explain the STG using the example of a *Muller C*-gate [79].

The C-gate is a memory element often found in asynchronous circuits. Unlike the DFF it stores the input state in an event-triggered fashion. It features two input ports and a single output port. If both input ports are 1 resp. 0, the output port is 1 resp. 0. Otherwise the C-gate holds its output. Figure 2.22 shows a C-gate's timing behaviour, interface and its *Signal Transition Graph* (STG) extracted from the timing specification. The C-gate is embedded in a simple environment, which just feeds back its output to its inputs via two inverters.

The STG is a simple representation of a *Petri net*. It is built of:

Edges: Edges represent states of input and output ports of the circuit. The current/initial state is indicated by a single token on the according edges.

Nodes: Each node specifies either a rising or falling signal transition on the input or output ports.

Once tokens are present on all input edges of a node, the corresponding transition is allowed to occur. As a result, the tokens are removed from the input edges as soon as the transition occurs and one token is forwarded to each output edge. Note the apparent compatibility with the asynchronous execution model, which only considers that computations get eventually enabled and the adversary chooses the time instants of the computation events.

The simple test environment in Figure 2.22 shows that the C-element is operated metastability-free if new input transitions are guaranteed to occur after the C-element generated the required output transition, i.e., the input ports may only change once the C-element changed its output port. Note that the C-element must contain a storage loop, because it needs to hold the last output as long as the inputs are not equal. Hence, the timing requirements of the employed memory element resp. circuit, which is used to implement the C-element, must be met, i.e., the inputs must stay stable long enough such that the specified transition can be performed. In the case of the C-gate, the feedback path via the output port c is typically longer than the storage loop. Besides, a source waits for the required transition of the output, before providing the next input transitions. Thus, the storage loop will have settled before new inputs arrive usually.

A C-gate can be implemented using either an SR-latch. The set resp. reset input port is connected to the combinatorial output of the boolean function $a \wedge b$ resp. $\neg a \wedge \neg b$. In this

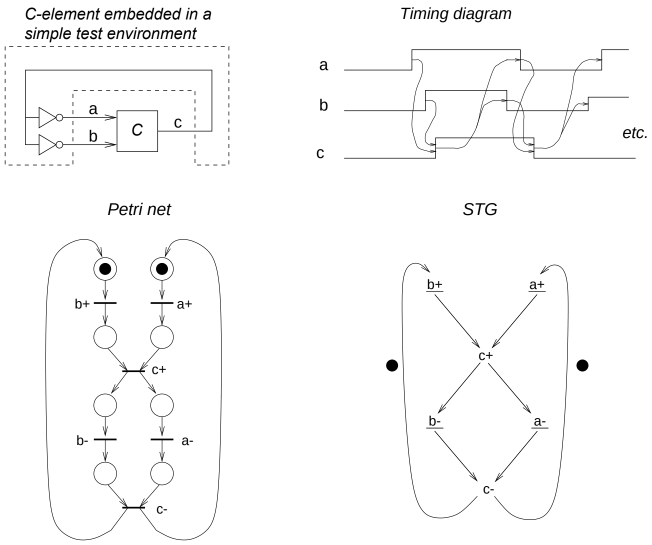


Figure 2.22: A Muller C-gate specification and its corresponding STG [73].

case no enable is needed (see Figure 2.23). Since this implementation is not optimal, other implementations are used in VLSI circuits typically [72].

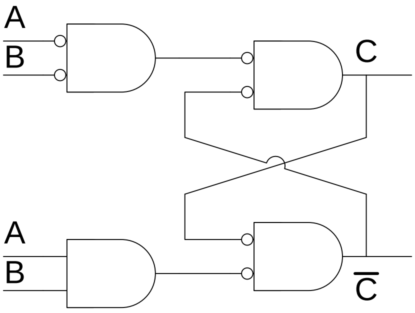


Figure 2.23: Schematic of a Muller C-element using an SR-latch (without an enable port).

A program like *Petrify* [73] is able to generate speed independent circuits from the STG specifications of a circuit. Typically, C-gates are used for state-holding purposes in the generated circuit. Obviously, there are some requirements on the STG, which must be fulfilled in order to implement a STG as a speed independent, glitch-free and metastability-free circuit (see [73]). Basically, everything that is similar to the *Producer/Consumer*-problem can be solved by such circuits (see [50]). The (generated) circuit implementations typically contain closed loop communication protocols as specified in Chapter 2.5.3.

2.5.3 Asynchronous communication

Any interface of a circuit to other synchronous or asynchronous circuits, either needs synchronization or metastability filters, if setup/hold violations can not be avoided by design. Thus, robust and fast means of metastability-free asynchronous communication are needed.

Existing approaches can be classified according to the behaviour of sink and source in Figure 2.14:

Fundamental mode: If the inputs of the sink are valid and consistent, the source may change exactly one output port and must wait until all inputs are valid and consistent again, before the next input (port) may change. As mentioned before, Huffman circuits can be operated glitch-free in this mode (single input change).

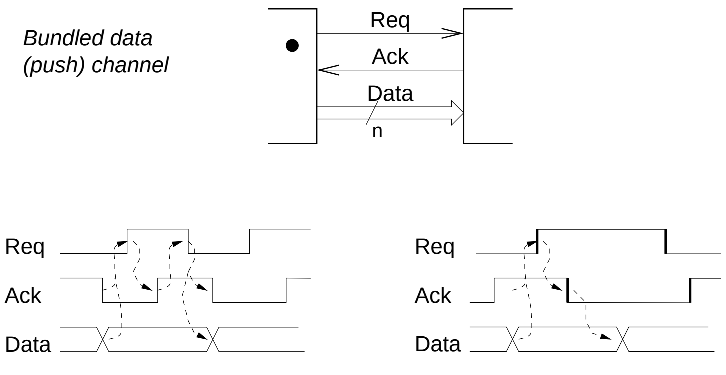
Input/Output mode: If the inputs of the sink are valid and consistent, it will provide the source with an output signal which allows the source to provide new data. Therefore, this communication mode is a closed loop protocol, as source and sink need to wait for each other.

2.5.3.1 Handshaking

A common communication style, used to establish input/output mode communication and indicate new data transmission, is called a *handshake* protocol. A handshake works as follows: Whenever a source wants to initiate a transmission it sends the data along with a request signal. Once the request is received by the sink and it is ready to process resp. store the data it will acknowledge the transmission. At the time the source receives the acknowledgment, it may issue a new request. A handshaking protocol forms a closed loop, i.e., sender and receiver need to wait for the corresponding handshake signal, before taking further steps. Obviously, the originator of a new transmission is the source in this example, in which case the link is referred to as push channel (see Figure 2.24). If the request resp. acknowledgment signal needs to return to zero, before a new request can be issued, the handshake is called 4-phase. A 2-phase handshake protocol uses both rising and falling transitions, which doubles the transmission rate approximately.

Although there is no need for both nodes to share a clock in order to use handshaking, the issue to recognize the validity and consistency of the data remains: A race between the request and the bundled data exists, but this problem is of a much smaller scale than distributing a global clock. Usually, the request and data race is solved in the time domain by compensating the skew between the arrival of the request and data signals, at the sink via suitable delays.

In fact the request resp. acknowledge signal is able to violate timing margins if the sender and receiver circuit implementations are not able to avoid it by design, which will be addressed in the following Chapters. Moreover, the overall communication speed is limited by the delays of the request and data signals, because the sender needs to wait for the acknowledge of the receiver. Another problem arises in fault-tolerant systems: Apparently, the sender and receiver handshake signals and controller states must be consistent during operation, a deadlock could occur otherwise.



4-phase protocol *2-phase protocol*

Figure 2.24: An example of a push channel handshake communication either 4-phase (RZ) or 2-phase NRZ) [73].

2.5.3.2 Micropipelines

Micropipelines [76] are a frequently used approach for implementing a pipeline in asynchronous designs.

Similar to the synchronous pipeline, it consists of stages (i.e., registers and combinatorial functions) connected in series (see Figure 2.25). Instead of using a common clock, each stage contains an additional Muller C-element to provide the trigger for the so-called *capture and pass registers*. Like in the standard handshake case, the (bundled) data must be available before the trigger at the corresponding stage's register, which captures the input. Again, delays need to be inserted to eliminate the possible race condition, thereby ensuring validity and consistency.

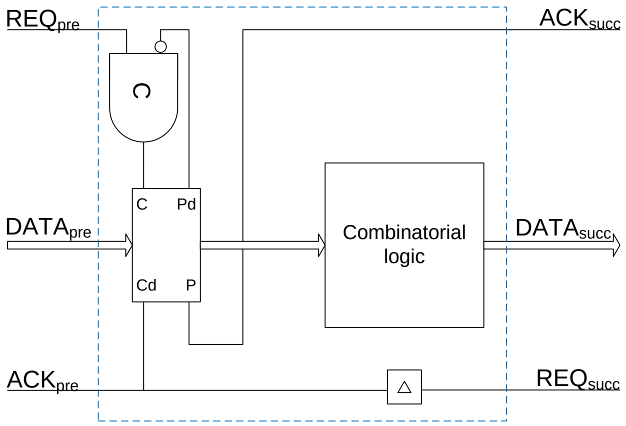


Figure 2.25: A example of a single micropipeline stage, including a combinatorial function, for processing the data. [73]

The capture and pass register is transparent and connects the input to the output (i.e., passes) if both the capture and pass signal are equal, and otherwise holds its output. The capture done signal is used as both a request for the next stage and a acknowledgment for the previous one. The pass signal is the acknowledgment received from the next stage. The capture resp. pass signal are delayed by the capture and pass register to the output ports capture done and pass done. These delays ensure the setup and hold requirements of the employed memory element(s). For possible memory element implementation options see [76].

Obviously the capture and pass registers can either implement a 2-phase or 4-phase handshaking protocol. The advantage of handshaking compared over communication between two synchronous designs without a handshake is obvious: The request, acknowledge and data signals can not change at inappropriate time instants due to the handshaking protocol and inserted delays. Hence, the pipeline is metastability-free. Since there are no discrete time instants triggering a state transitions, the execution is entirely event-triggered, i.e., asynchronous. This may even result in a performance improvement compared to a synchronous pipeline. On the downside, the additional hardware and delay elements required, make micropipelines unsuitable for complex systems and nodes. Consequently, they are typically employed for implementing fast asynchronous buffers or small modules, rather than complex processing algorithms.

2.5.3.3 Transition signaling

The previously introduced circuits all share the same problem: Properly chosen delays need to be introduced to compensate the skew between data and trigger signals, since the validity and consistency of data signals could not be recognized from the data alone. The goal of transition signaling is to encode data signals such that it is immediately apparent from the data itself whether it is valid and consistent or not. Therefore, a new request must encoded in the data itself, for example. In general data communication scenarios, i.e., where arbitrary data bits are allowed to change in each new data word, an extra validity signal is required for each bit due to the fact that a single bit is able to represent either 0 or 1 only. This is called a *two-rail* communication.

The following encodings are commonly used (see [19–21]):

Null convention logic (NCL): A NCL-gate accepts two rail inputs with the following values: 10, 01, 00 and 11. The latter is illegal and must be ignored. The code words 10 and 01 represent logic high(h) and logic low(l) and the codeword 00 represents the *NULL* value (n). Once all inputs of an NCL-gate are h or l the gate will change its outputs h or l as specified by the desired boolean function. If all inputs are n, the outputs will switch to n. A NCL-gate holds its outputs if any other input combinations are applied. Figure 2.26 shows an example NCL data stream: Between two successive data waves a null wave is inserted.

Four state logic(FSL): A FSL-gate accepts two rail inputs with any of the four encoding from 00 to 11. The encodings are grouped in two phases: 10 resp. 01 (h/l) represent one phase and 11 resp. 00 (H/L) the other phase. Once all inputs of an FSL-gate are from the same phase, the gate will output the corresponding value specified by the desired boolean function using the encoding of the input phase.

Both NCL and FSL gate implementations often offer an additional acknowledgment output port, using single-rail encoding, which acts as acknowledgment for the source to reduce the communication effort back to the source. The acknowledgment signal reflects the current output phase (FSL) or the current output wave (NCL). For example, a null wave acknowledgment is encoded using 0 and a data wave is acknowledge with 1.

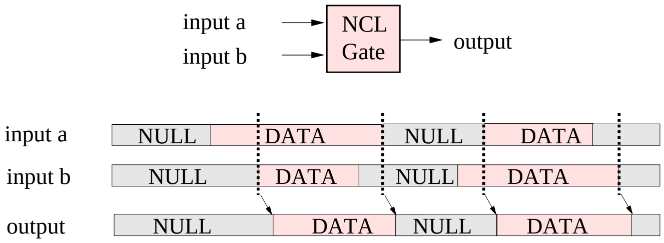


Figure 2.26: Data and null wave communication using NCL gates [20].

AND	**h**	**l**	**n**	**i**
h	h	l	*	*
l	l	l	*	*
n	*	*	n	*
i	*	*	*	*

AND	**H**	**L**	**h**	**l**
H	H	L	*	*
L	L	L	*	*
h	*	*	h	l
l	*	*	l	l

Table 2.2: Truth table of a two input NCL(left) resp. FSL(right) **AND**-gate. The symbols represent the codes in the description and * means hold last output.

Obviously, the outputs of such a gate will not glitch and the delays on the links do not need to be considered. Table 2.2 shows the truth tables of an **AND**-gate for both encodings. Apparently, the NCL-gate is similar to a 4-phase handshake communication, whereas the FSL implements a 2-phase handshake, since any of the two phase transmits data. Using the acknowledgment and one of the above data encodings, it is possible to safely transmit data as the source is able to establish an input/output mode communication.

Nonetheless, NCL and FSL circuits come with the problem of an increased hardware effort due to one additional rail per bit and at least two memory elements (e.g., SR-latch) for each rail per gate. Furthermore, it must be ensured that the combinatorial function, used to set/reset the memory element output port of a rail, does not glitch. For example, the NCL **AND**-gate's set function for the SR-latch of rail 1 could be the equation: $(a.r_1 \land \neg a.r_0) \land (b.r_1 \land \neg b.r_0)$. The NCL/FSL encoding enforces that only one rail per (two-rail) input port performs a transition and all transitions are of the same polarity, i.e., all rails may perform either a rising transition or a falling transition only. Hence, at least the combinatorial set/reset functions of basic gates, like the **AND**-gate of Table 2.2, can be implemented glitch-free.

Note that (possibly) different delays of the set and reset functions, it may occur that the set and reset port of a memory element are active at the same time. It must be ensured that this does not lead to an early or invalid deactivation of the set and reset function due to new arriving encoded data words, i.e., before the memory element has settled. Hence, the QDI isochronic fork assumptions must hold to establish a correct input/output mode, which is satisfied by the longer feedback path via the source, usually.

To sum up, NCL/FSL circuits are QDI asynchronous circuits.

2.5.4 Pipelined asynchronous communication

The communication protocols introduced so far, use closed loop asynchronous channels, which poses a problem if the communication speed shall be increased without increasing the data width by means of pipelined communication. The handshake signals must always propagate back and forth between the sender and receiver, i.e., the transmission rate is limited by the overall delay d on the channel (link and combinatorial functions). In the case of pipelined communication, the sender could send a new data bit every $\tau < d$ seconds after initiating the transmission, without the need to wait for the receiver's acknowledgment for every single bit. Therefore the serial transmission of B bits in this example would be completed within $d + B \cdot \tau$ seconds. Inevitably, pipelined communication involves some timer driven execution, like sending new data (bits) every τ seconds.

Obviously, metastability robustness will depend on the actual protocol and the sender resp. receiver implementation. In general, metastability will not be avoidable by design like before, however, because implementations must resort to drifting resp. unrelated clocks.

2.5.4.1 UART

Serial Universal Asynchronous Receiver and Transmitter protocols are the most used and widespread communication protocols in embedded systems, for example it is used by the [78] standard. Typical UART connections convey serial terminal interface data, configuration data, sensor data or logging data. It requires only a single wire and is easy to implement.

A data frame is encoded using a typical data format of one start bit, 4-9 data bits, 0-1 parity bit and 1-2 stop bits (see Figure 2.27). In idle mode, i.e., when there is no data to transmit, the output signal at the transmitter outputs logical high (1). A new frame is indicated by a transition to the logic level 0 (start bit). After that, the data bits are transmitted beginning with the LSB. At last, a parity bit (**XOR** or **XNOR** sum of the data bits) may be transmitted. The frame is completed by a transition to the idle state for 1 or 2 bits (stop bits).

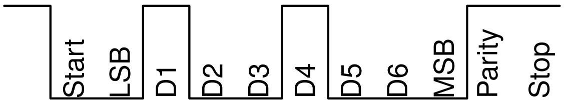


Figure 2.27: Example UART frame with eight data bits, one odd parity bit and one stop bit.

If f_{baud} gives the frequency (i.e., baudrate) of the transmit clock, each bit takes $1/f_{\text{baud}}$ seconds for transmission.

The achievable baudrate is limited due to noise, filtering(jitter) and DC offsets, which increase with the wire length. Although, these problems can be neglected for (short) on-chip communication, clock drifts are a limiting factor:

Assume the initial start bit offset introduced by the receiver implementation is at most d_{off}. A simple receiver will now sample each bit, including the start bit, periodically using its clock, which frequency drifts within the interval $[1/(\vartheta f_{\text{baud}}), 1/f_{\text{baud}}]$. The worst case accumulated offset is then $d_{\text{off}} + [(W-1)/f_{\text{baud}}](1-1/\vartheta)$, where W is the number of symbols in the frame. Since the offset must be less than $1/(2\vartheta f_{\text{baud}})$ to guarantee proper decoding at the receiver, the following baudrate inequality must hold:

$$f_{\text{baud}} < (1/2 - (W-1)(\vartheta - 1))/(\vartheta d_{\text{off}}) \tag{2.2}$$

Assuming the simple receiver circuit oversamples the UART data signal by a factor of 16 and it takes at most 2 oversampling clock cycles to detect the start of a transmission, which starts the periodic sampling too, the worst case initial offset is $d_{\text{off}} = 1/(2f_{\text{baud}}) - 1/(2\vartheta f_{\text{baud}}) + 2/(16 f_{\text{baud}})$. Note that this is equal to $(5\vartheta - 4)/(8\vartheta f_{\text{baud}})$. The analysis is based on the results in [29].

2.5.4.2 SPI

Alternatively, the clock may be passed along with the data by the sender via a separate wire. Such an interface is often used between peripheral components in embedded systems like sensors, flash, LCD and others. It hence suffers from similar problems as the UART over longer distances. Typical interfaces are implemented bi-directional on the data link to allow slave devices to answer over the same channel. The receivers and transmitters are very simple to implement, but more careful routing needs to be performed to ensure a tolerable skew between clock and data at the receiver.

A typical SPI interface comprises 3 wires: Data, Clock and Slave-Select (request). An example communication is depicted in Figure 2.28. Note the potential race between the request, clock and data. The data and clock are (often) connected to a shift register, which is an "empty" pipeline (without combinatorial logic), see Figure 2.17. The first input is connected to the data channel. Some receivers and transmitters allow to define the clock phase, clock period and clock polarity (sampling edge) for the transmission. In the standard setting, no full handshake is performed due to the missing acknowledgment signal. A new request is indicated by a low-active Slave-Select, before the first bit is transmitted. It rises again after the transmission is complete. Once a clock phase, clock polarity and clock period is chosen, the maximum skew between data, request and clock must be at most half a clock period minus setup and hold times.

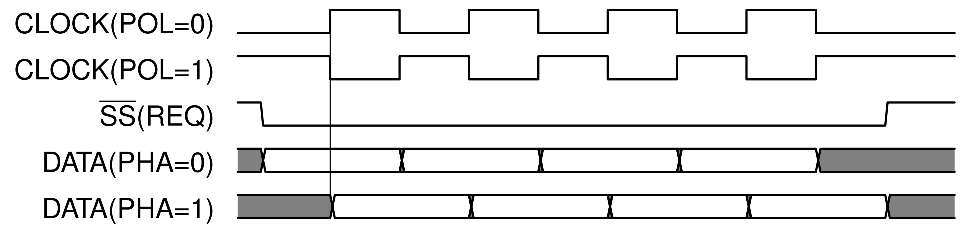


Figure 2.28: Example SPI communication of 4 bits.

2.5.4.3 Self-clocked codes

The two previous protocols suffer from the drawback that either the clock drift (UART) or the data-clock skew (SPI) limits the frame length and data rate. Self-clocked codes ensure periodic resynchronization points, s.t. the error introduced by the clock drift can be reduced at those instants and remains at an acceptable level. Some receivers even extract the sender clock from the data stream [84]. Examples of modern high-speed protocol standards that use these techniques are PCIe, Gigabit Ethernet [34], USB (3.0), SATA and others.

Most of the protocols mentioned are used for off-chip communication between peripherals or to other systems over a network. For DC and noise compensation, due to the increased interference and longer distances, it is often required to use (low voltage) differential signals (LVDS), which comprises two wires. Nonetheless, some of these principles may be used for SoC communication as single bit channels.

A well known self-clocked code is the *Manchester*-code [15]. It uses 2 symbols to encode a single bit (10 or 01). Obviously the Manchester-code needs a baudrate twice as high as in UART to achieve the same bitrate. Apart from that problem the receiver may resynchronize on every bit instead of every frame keeping the error minimal and allows larger clock drifts and still receive large frames. This code has been used in older Ethernet LANs with differential channels. Since there is no idle symbol in a single channel implementation a start-bit can be used just like in the UART setting to indicate a new transmission.

One of the most popular codes is the $8B/10B$ encoding [82], where a byte is encoded using a symbol with 10 bits length, but the actual meaning may vary. The encoding allows to specify special code words for idle states, as frame delimiters, for resynchronization or other commands that can not be found in the data code words. The encoding ensures that the running disparity, i.e, the difference in the amount of transmitted ones and zeros, within 20 bits of the transmitted data stream is less than 2. Besides, the amount of successive ones and zeros is at most 5. The above mentioned modern high-speed protocol standards, make use of such a code.

2.6 Globally Asynchronous and Locally Synchronous Systems

Globally Asynchronous and Locally Synchronous Systems (GALS) [13] are creating more asynchronous systems([40, 77, 83]). It is a very common technique in SoCs due to the variety of processors and interfaces. It comprises locally synchronous subsystems, each of which runs at its own pace. The GALS principle tries to avoid the problem of distributing a single clock to all components by distributing multiple clocks to these locally synchronous subsystems instead (see Figure 2.29). These subsystems are processors, signal processors, system buses and other components. Hence, the subsystems employ asynchronous communication protocols for means of interfacing. Therefore, the GALS principle is used to implement distributed systems on a chip in general.

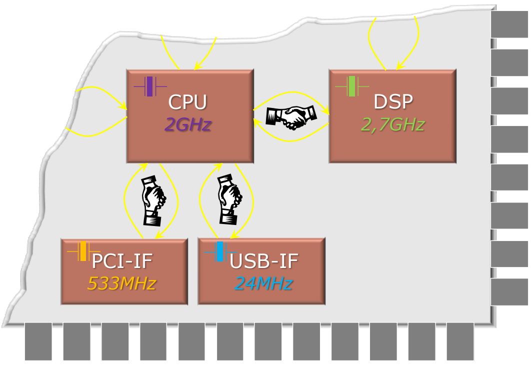


Figure 2.29: An example architecture of a SoC using the GALS principle [75].

The advantages a GALS system are obvious:

- Use of standard tools for locally synchronous subsystems.
- Local clock distribution simpler.
- Clock frequency not limited to the slowest subsystem clock.
- Reuse of existing synchronous circuits.
- Simple and fast system design, i.e., less complex than pure synchronous or asynchronous design principles.
- Many efficient and robust asynchronous communication protocols exist.

But the global notion of time is lost. This implies that in general timing violations will be inevitable due to the asynchronous communications protocols, which may result in metastable upsets. Furthermore, synchronized actions and global schedules are not guaranteed.

Hence, GALS systems can be further classified depending on which kinds of clocks are available to the nodes ([75]):

Synchronous: Clocks with identical frequency and constant phase relation are available. Classical synchronous design with one clock source usually.

Multi-synchronous: The available clocks maintain a global synchrony without accumulating drift, but unknown bounded and varying phase relation (skew). Examples are employed clock synchronization algorithms or a *PLL* driven by one clock source.

Plesiochronous: All clocks offer the same nominal frequency, but there is a (bounded) accumulating drift. Examples are independent oscillators with the same nominal frequency.

Heterochronous: All clocks are unrelated and have different nominal frequencies.

The attentive reader will notice that the following two major issues must be solved in (fault-tolerant) GALS systems:

- Clock generation.
- Communication.

Providing these services in a fault-tolerant and self-stabilizing fashion is preferred by the (multi-synchronous) GALS designer.

2.6.1 Clock generation

Most systems use asynchronous circuitry as clock distribution techniques. A single clock source feeds *Phase-locked Loops* (PLLs) [5, 42, 70] to compensate the clock tree skew. A PLL may even generate higher or slower clocks from this base input clock (i.e., perform a clock frequency division or multiplication). Besides, more complex clock nodes, than simple buffers, can be employed to possible reduce the skew even further by delaying the clock edge propagation, until a certain amount (e.g., threshold) of its inputs has observed this clock edge [30]. Although, a PLL can generate multiple sufficiently synchronized clock sources for multi-synchronous GALS, which do not suffer from an accumulated clock drift since it is based on a single clock source, the PLL is in general still not fault-tolerant and a single point of failure.

Hence, fault-tolerant clock (tick/pulse) generation algorithms must be employed to build reliable multi-synchronous GALS. A popular fault-tolerant algorithm to provide local synchronized clock ticks is [74]. It assumes that sufficiently many correct nodes remain synchronized within the specified precision, to be able to reintegrate failed nodes after the startup procedure. This algorithm has been extended in [81] to hybrid failure models (e.g., link and node failures) of different severity (e.g., crash and Byzantine failures). A reduced approach of this algorithm has been implemented in the DARTS [38] scheme, using asynchronous design styles. It comes without the need to distribute linear sized messages (i.e., synchronous round number) and fits the needs of VLSI circuits by just exchanging binary clock signals. On the downside, it looses the ability to reintegrate failed nodes. Each DARTS module provides tightly synchronized clock pulses for a single GALS clock domain.

The missing availability of Byzantine self-stabilizing pulse synchronization algorithms, that suit VLSI requirements (e.g., constant sized messages and metastability), has lead to the development of the *FATAL*$^+$ algorithm [26]. It offers probabilistic self-stabilization, even in the presence of Byzantine faulty nodes, within linear time. Each *FATAL*$^+$ node provides fast and slow synchronized pulses and a (short) bounded synchronized clock that increments in lock-step manner at every fast pulse and resets to zero at every slow pulse (see Chapter 3). The only assumptions are those of the bounded delay model (e.g., bounded drift, delay and computation time).

2.6.2 Communication

Any of the previously described protocol can be used for this purpose (e.g., pipelined communication and handshaking). Existing interface standards include *PCIe*, *GbE*, *SATA*, *DDR3* including the very popular embedded system bus protocols *AMBA* and the *Wishbone*.

The most common asynchronous communication techniques, to interface with the locally synchronous subsystems, are handshaking controllers [41], shared memory or synchronizing FIFOs [23, 24, 64, 77]. Sometimes asynchronous communication methods or on-chip networks are employed like *GbE* [34] between the modules. Obviously, metastability can not be avoided, in general, if no global synchrony exists, but the probability is usually minimized by metastability filters liker synchronizers.

The Wishbone and AMBA protocols make use of handshaking techniques, by organizing the modules into masters and slaves. The bus communication is typically driven by its own synchronous clock to overcome the problems of metastability. In (fault-tolerant) multi-synchronous GALS systems, each subsystem could use its own clock for this purpose. In order to grant the bus to a single master only, arbitration modules, which grant only one access per time, are needed. A slave is selected by the master by applying the slave address(es) to the bus, which waits for the acknowledgment.

As mentioned above FIFOs like the Micropipeline can be used as fast means of communication, but an input element needs to pass the whole pipeline (even if the elements are not processed). Instead, some FIFO implementations use on-chip memories with read and write ports. The sender is allowed write as long as the FIFO is not full and the receiver as longs the FIFO is not empty. This requires to remember the current read and write address locations of the memory, which must be managed either by the sender resp. receiver or an asynchronous or synchronous arbiter module. Most FIFOs are based on handshaking mechanisms using either dual-clock, i.e reader/writer clock, shared memories as buffers to improve the burst data rate if the sink is slower in accepting new transmissions than the source. The FIFO decouples the source and sink s.t. they do not need to wait for each other. The handshaking controller between the FIFO and the sender resp. receiver act as metastability filters. Hence, this communication is usually reliable.

Nonetheless, if modules resp. nodes of a GALS system may fail the handshaking mechanisms, FIFO read and write addresses, bus protocols and transition signaling may become inconsistent due to faults and introduced errors. Hence, self-stabilizing mechanisms or recovery circuits like watchdogs that try to detect wrong states and deadlocks, must reset the communication controllers accordingly or other recovery mechanisms are needed (see for NCL/FSL self-healing circuits [62]).

Apart from global schedules and simultaneous actions, a global notion of time in multi-synchronous GALS facilitates metastability-free communication either via simple DFFs, fast FIFOs [64] or handshake-free synchronizers like the phase predicting *Even-Odd*-synchronizer [18].

2.6.3 Hybrid state machines

In GALS systems, a synchronous FSM needs to be interfaced with another synchronous FSM without introducing unnecessary delays and, if possible, without metastable upsets. *Hybrid State Machines* (HSMs) have been used successfully for this purpose [26, 66]. A HSM implements a transaction-based scheme: A state transition is performed by a pausable synchronous state machine, in multiple steps, but triggered asynchronously. For this purpose the synchronous part of a HSM, called the *transition state machine* (TSM), which stops executing once a state transition is complete, by pausing its local clock. Figure 2.30 shows an example transition including the transition state machine (TSM) of FATAL$^+$.

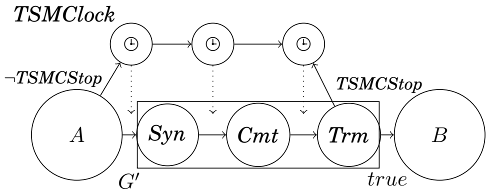


Figure 2.30: Example transition including the TSM [26].

All state transitions guards are evaluated asynchronously by (standard) combinatorial circuits from the node's external (data) input ports, the HSM local state (i.e., FSM state) as well as those of any other local module (e.g., timers, memory flags and voting modules). Once a transition guard is true, the TSM clock is resumed. Hence, the TSM module executes the required state transition. Since multiple transition guards may become true close to each other, the HSM resolves the issue using a tie-breaking mechanism s.t only one transition guard remains enabled (e.g., by assigning static priorities).

The first state of the TSM (Syn) sets a lock signal to prohibit that other transitions guards become true, since the HSM has chosen its next state candidate already. This also enforces that this transition guard remains true. The FSM state transition computations are implemented in the state (Cmt) of the TSM, i.e., it resets associated memory elements and timeouts and stores the new FSM state. At last the TSM terminates (Trm) and completes the state transition by clearing all resets and the lock signal. Usually, at this point no transition guard is true, which pauses the clock.

Apparently, each HSM needs its own pausable clock, but the available amount of external precise quartz oscillators is rather low (cost and area). Hence, internal imprecise (i.e., large ϑ) ring oscillators are used for this purpose. Basic ring oscillators are just a loop containing an odd number of inverters (see [66]). Note that the combinatorial circuits must be glitch-free, i.e., the designer needs to choose suitable state encodings for state communication among HSM nodes (see [66]). Unfortunately, the HSM can suffer from metastable upsets:

- Multiple transitions may become true at inadequate instants s.t. either tie-breaking resp. the lock signal has not settled.
- Faulty nodes may introduce glitches and runt pulses, which can upset the memory elements of transition guards.

In many (self-stabilizing) algorithms such upsets can not occur after stabilization in fault-free executions, since one guard will be true at a time only. Besides, (deep) metastable upsets are rare and may be mitigated using metastability filters and voting modules (see Chapter 2.3). Consequently, metastable upsets can only delay stabilization.

To sum up, combining event and time-triggered execution styles, allows to use the simplicity of synchronous state transitions, and thereby ensuring all timing requirements of local modules are met, and still start a state transition as soon as possible. Furthermore, it does not need extra stabilization mechanisms if the algorithm FSM actually executes some stabilization protocol. Should the HSM stop in an unspecified FSM state, this can be covered by a default transition guard to some specified state. Moreover, pausing the clock implicitly saves power.

2.7 Designflow

Today's vendor tools allow designers to describe circuits at the *register-transfer level* (RTL) and even at lower levels. Hereby, circuits may be directly described as FSMs or boolean circuits (functions) or by assembling already available components. The remaining transformation to the completed circuit is done by automatic synthesis tools (using technology libraries), which map the desired functions to the vendor-specific low-level gates and components. If the generated circuit is not able to fulfill the requirements, the description may be changed or manually transformed to a lower level. A synthesis tool performs the following operations automatically:

1. Analyze and compile design entry description (VHDL/Verilog RTL). Allows behavioural simulations (fast).
2. Synthesis (technology mapping to gates and components of the vendor). Allows pre-Layout simulations and estimated (gate) delays (slow).
3. Place and Route (place the components on the chip and connect them). Allows post-Layout simulations using actual delays (very slow).
4. Timing analysis

Modern synthesis tools are able to perform various optimizations and metastability or power estimation analysis.

However, typical cost factors in VLSI design are:

- Power
- Area
- Performance (Frequency, Jitter)
- Robustness (glitch-free, metastability-free, noise, etc.)

Each of these are influenced by:

- Link (length, amount)

- Fan-out (amount of links to a single output port)

- Logic (gates,memory)

- Depth/Delay (propagation delay of cascaded logic)

All these cost factors are also influenced by the choice of technology (e.g., feature size, production process, transistors and internal structure). Synthesis tools provide coding guidelines to automatically infer specific lower-level blocks like multiplexers, multipliers, adders, SRAM-memory, FIFOs, registers, latches, FSMs and many more, which can be tailored to optimize certain cost factors.

2.7.1 Field Programmable Gate Arrays

A *Field Programmable Gate Array* (FPGA) contains many predefined vendor configurable hardware modules, which may be connected via a programmable *interconnect*. FPGAs are frequently used as rapid hardware prototyping platform or if creating an *Application Specific Integrated Circuit* (ASIC) is to expensive due to a low manufacturing volume or when specifications change frequently. Most FPGAs and vendor tools are streamlined to be used in synchronous designs using on-board oscillators or external clock sources. Hence, most FPGAs contain well-balanced clock trees. In the following, I introduce a typical structure and basic elements using the *Cyclone IV*-FPGA [3] as an example.

Logic Array Blocks

The FPGA layout is organized in a grid structure (i.e., columns and rows) of so called *logic array blocks* (LAB), each consisting of 16 *logic elements* (LE). The inputs and outputs of a LAB may be tied to the row resp. column interconnects connecting it to adjacent LABs of the same row resp. column. These interconnects actually drive the local interconnect of a LAB, which physically interfaces to the LEs: Each LE output can be routed either to the local, column or row interconnect. Furthermore, each LE may use a direct link to a LE in an adjacent LAB on the left or right to save row and column interconnects.

Figure 2.31 depicts the grids structure using row, column and local interconnects of a LAB containing 16 LE. Further inputs and outputs of a local interconnect can be embedded RAM blocks, multipliers, global clocks or dedicated input and output pins.

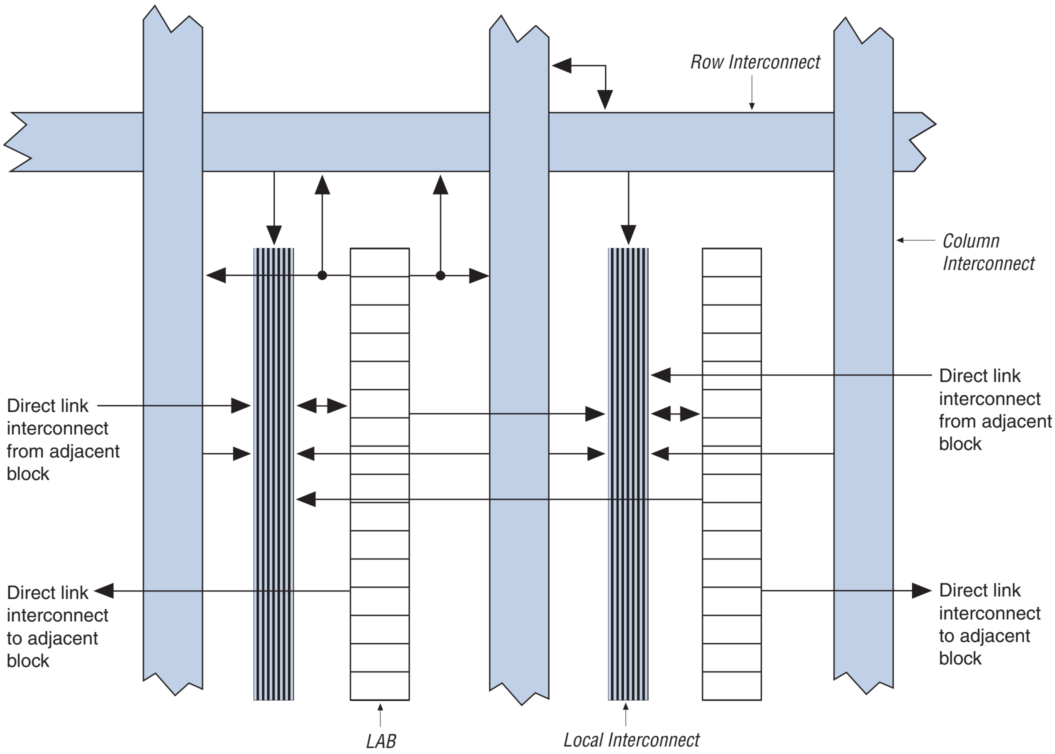

Row Interconnect
Column Interconnect
Direct link interconnect from adjacent block
Direct link interconnect from adjacent block
Direct link interconnect to adjacent block
Direct link interconnect to adjacent block
LAB
Local Interconnect

Figure 2.31: Structure of a *Cyclone IV*-FPGA LAB and interconnect [3].

Logic Elements

A *logic element* (LE) contains several components and is used to implement combinatorial circuits and memory elements (see Figure 2.32). The most important part is the *Look-up Table* (LUT). It implements an arbitrary 4-input and single output boolean truth table. Its output can either be routed directly to the interconnects or via a memory element, represented by a standard register (i.e., a DFF). The LUT may also pass its output fast to the next LE in the same LAB to implement arithmetic functions like addition with carry logic. Note that, the DFF can select between one of two LAB-wide clocks, and that unused memory elements may even be used by another LE (register packing). Obviously, an LE can implement a single stage in a synchronous pipeline, like the one of Figure 2.17.

Clocking

The *Cyclone IV*-FPGA provides up to 30 low-skew clock networks. Moreover, the FPGA offers 4 PLLs, i.e., input clock frequency division and multiplication circuits with adjustable phase relation, which can offer up to 5 output clocks sources, each capable of driving a clock network. Each LAB can select two clocks out of these or can use a source from its local interconnect Sometimes, a clock is generated internally by the user circuit (e.g., by ring oscillators or clock tick generation algorithms). Note that due to the limited number of available global

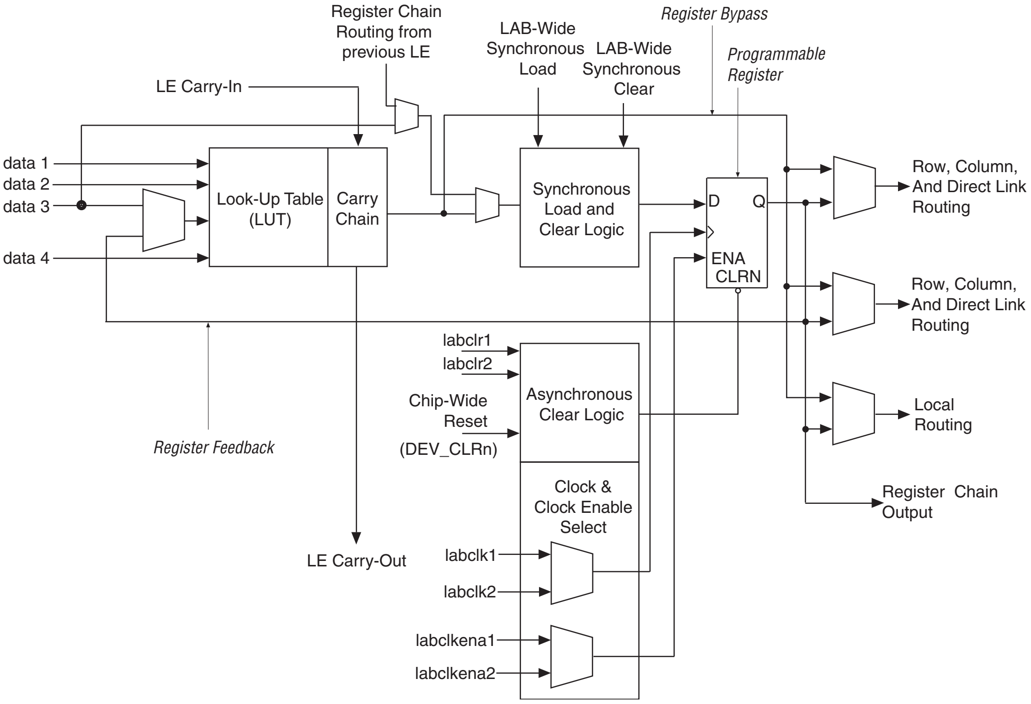


Figure 2.32: Structure of a *Cyclone IV*-FPGA logic element [3].

clock networks, it is sometimes necessary to route internally generated clocks via the standard interconnects to the required LABs, which increases the clock skew and limits the achievable clock frequency typically.

Timing Analysis

A synthesis tool is used for automatically mapping a net-list design onto the suitable LABs in an FPGA. Obviously, that does not imply that all timing constraints are met. Therefore, the TimeQuest timing analyzer [1] verifies every path between two DFFs (*register-to-register path*) using the minimum and maximum delays introduced by the clock network, local, row and column routing, LUTs and memory elements on the path. Note that it is only capable of analyzing synchronous or multi-synchronous designs for timing violations. The most important timings constraints to be verified are the setup and hold constraints of a DFF. For this purpose, the timing analysis uses different models for environmental effects like temperature and supply voltage to cover different (worst case) scenarios.

The following explanation will serve as an example how synchronous or multi-synchronous designs can be verified under the assumptions of a bounded delay links and logic.

The general setup slack equation is given by [2]:

$$\text{Clock Setup Slack} = \text{Data Required Time} - \text{Data Arrival Time} \tag{2.3}$$

$$\begin{aligned}\text{Data Arrival Time} = {} & \text{Current Launch Edge} + \text{Clock Network delay to Source} \\ & + \mu t_{\text{CO}} + \text{Register-to-Register Delay}\end{aligned} \tag{2.4}$$

$$\begin{aligned}\text{Data Required Time} = {} & \text{Next Latch Edge} + \text{Clock Network Delay to Sink} \\ & - \mu t_{\text{S}} - \text{Setup Uncertainty}\end{aligned} \tag{2.5}$$

The computed clock setup slack is a margin by which a timing constraint is either met or not. If the Clock Setup Slack > 0, a new issued (launched) data by the source DFF will always arrive before the sink DFF next capture clock edge occurs (where it latches the data). See the example in Figure 2.33. In order for the timing analyzer to analyze the communication between two DFFs, the clock of the source and sink DFF must be specified, i.e., the clock transition time instants. Note that the source and destination clock specifications are the same in a synchronous design.

Either the rising or falling clock transitions are plugged in into the above equation as *Current Launch Edge* resp. *Next Latch Edge*, depending on whether the rising resp. falling clock transition is used to control the source resp. sink DFF. Since the worst case setup relation must be analyzed, the closest two triggering source and destination clock transitions have to be chosen, which are exactly one clock cycle apart from each other in a synchronous design. The remaining parameters are extracted from the clock distribution network (maximum *Clock Network delay to Source* and minimum *Clock Network delay to Sink*), combinatorial path delays (maximum *Register-to-Register Delay*) and DFF parameters (clock to output μt_{CO} and setup time μt_{S}).

This results in the following reduced equation for synchronous designs, where the same clock edge is used for both launching and latching the data:

$$\text{Clock Setup Slack} = \text{Minimum Clock Period} - \text{Skew} - d_{\text{max}} \tag{2.6}$$

$$\begin{aligned}d_{\text{max}} = {} & \mu t_{\text{S}} + \text{Setup Uncertainty} \\ & + \mu t_{\text{CO}} + \text{Register-to-Register Delay}\end{aligned} \tag{2.7}$$

As already mentioned, a sink can overlook a received state in bounded delay link models using state-based communication. Since a clock transition is used for both receiving and sending new data, it can happen that new data (*Next Launch Edge*) is sent by the source before the current one has been stored by the sink (*Current Latch Edge*), i.e., it overlooked the old state data. Hence, it must be ensured that the data holds until the sink successfully latched the data.

The general hold slack equation is given by [2]:

$$\text{Clock Hold Slack} = \text{Data Arrival Time} - \text{Data Required Time} \tag{2.8}$$

$$\begin{aligned}\text{Data Arrival Time} = {} & \text{Next Launch Edge} + \text{Clock Network delay to Source} \\ & + \mu t_{\text{CO}} + \text{Register-to-Register Delay}\end{aligned} \tag{2.9}$$

$$\begin{aligned}\text{Data Required Time} = {} & \text{Current Latch Edge} + \text{Clock Network Delay to Sink} \\ & + \mu t_{\text{H}} + \text{Hold Uncertainty}\end{aligned} \tag{2.10}$$

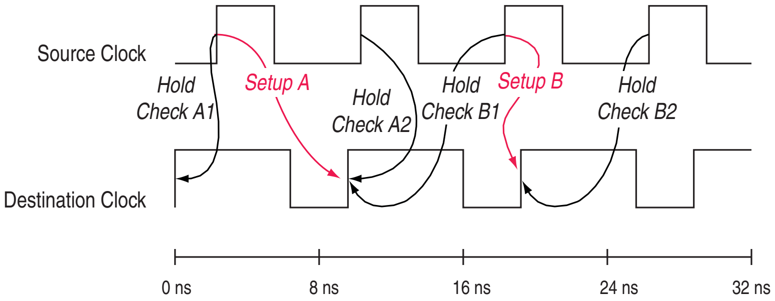


Figure 2.33: In this example hold relation *A2* and setup relation *B* are the most restrictive ones [2].

If the Clock Hold Slack > 0 no hold violations are going to occur. Again, the closest two clock transitions must be chosen and plugged into the above equation. In a synchronous design these transitions are the same, i.e., there is no time in between. The remaining parameters are extracted from the clock distribution network (minimum *Clock Network delay to Source* and maximum *Clock Network delay to Sink*), combinatorial path delays (minimum *Register-to-Register Delay*) and DFF parameters (clock to output μt_{CO} and hold time μt_{H}).

If a synchronous design is analyzed (using the same edge to launch and latch data), the hold slack can be reduced to:

$$
\begin{aligned}
& && (2.11)\\
\text{Clock Hold Slack} &= d_{\text{min}} - \text{Skew} && \\
&\quad - \mu t_{\text{H}} - \text{Hold Uncertainty} && (2.12)\\
d_{\text{min}} &= \mu t_{\text{CO}} + \text{Register-to-Register Delay} && (2.13)
\end{aligned}
$$

Apparently, it is possible to solve setup violations by reducing the clock period in these designs, but not hold violations. This poses a problem in designs with large clock skews. A solution is presented in Chapter 4.

The timing analysis tool automatically extracts the above delays and the worst case clock transitions from the clock specification, but this specification is static. Hence, all clock transitions occur at the periodic specified time instants (no drift), which implies also predefined phase relations for clocks with the same nominal frequency. Since clock synchronization algorithms introduce a significant amount of skew, that is not a result of the clock network delay, this part of the overall skew must be specified using the uncertainty parameters in the equation. Besides, the timing analysis tool is not able to analyze the properties of a clock synchronization algorithm. Therefore, either the designer needs to provide own tools to extract the required information and verify these against the constrains or must make assumptions about it. Providing automatic tools, is not part of this thesis. Hence, the assumed delays and theoretical results of a clock synchronization algorithm can be plugged into the above equations manually (see Chapter 5.2).

CHAPTER 3

FATAL$^+$ and the Labeling solution

In this chapter, an overview about FATAL$^+$ (see [26]), the labeling problem to be solved, and the solution algorithm and its proofs will be given. It will summarize the important features and properties of FATAL$^+$ that are needed to provide and implement a solution for the self-stabilizing and fault-tolerant labeling problem.

3.1 Pulse synchronization with FATAL$^+$

As noted in Chapter 2, many distributed problems need algorithms to be run within the synchronous lock-step round or bounded delay model in order to solve them in the presence of Byzantine failures. Among these are the pulse and clock synchronization problems. Fault-tolerant Algorithms for Tick-Generation in Asynchronous Logic (FATAL$^+$ [26]) provides a self-stabilizing solution to the clock and pulse synchronization problem in the presence of at most $f \leq \lfloor (n-1)/3 \rfloor$ Byzantine faults, which stabilize with high probability.

The FATAL$^+$ pulse synchronization algorithm is designed to cope with the typical problems of VLSI designs, like metastability, drifting clocks and delay at once and still provides linear probabilistic stabilization/convergence. The prototype has been implemented in [66]. The synchronized slow and fast pulses are provided by the main algorithm in Figure 3.3 and Quick-cycle algorithm in Figure 3.5 and the associated memory flags for the states *Accept* resp. *Accept*$^+$, which are referred as *FATAL*- and *FATAL*$^+$-pulses. This protocol stack, including the important interface to the application layer as well as the interface between those two algorithms, is depicted in Figure 3.1. M fast *FATAL*$^+$-pulses are generated by the Quick-cycle algorithm, which rely on the basic FATAL algorithm for self-stabilization, such that the provided BSCLK counts these fast pulses and overflows to 0 with the next FATAL-pulse. These are the most important algorithms concerning this thesis as these provide the interface to the labeling algorithm. A more detailed explanation is provided later in this chapter.

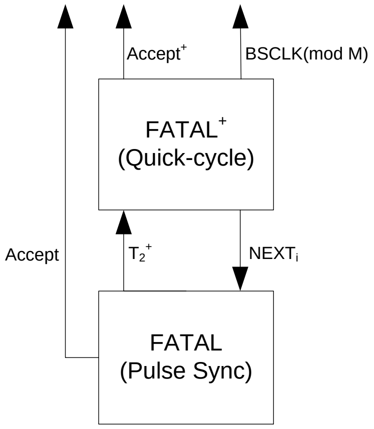


Figure 3.1: Generated synchronized slow and fast pulses including a bounded synchronized clock.

3.1.1 Modules

A node comprises the following elements: State machines, memory flags, links, (random) timers and clocks. [1]

Clocks and timers

The FATAL$^+$ prototype implementation uses pausable 1-bit clocks for the timeouts and state machines. A timeout is referred to by the model (and the state machine descriptions) as a tuple (T, s, C) or short (T, s), where C is a clock. The timeout T is reset (i.e., started) upon entering some specific state s in one of the state machines of FATAL$^+$. Sometimes, no state s is specified, then the timer is reset, for example, when entering a state that uses the timeout output port T in one of its outgoing transition guards.

Ports, memory flags and links

Each FSM provides a local output port S_i and input ports $S_{i,j}$ for each node i and j. $S_{i,j}$ is connected via a link to S_j, which conveys node j's state to node i. In the current FATAL$^+$ prototype, a code word with a Hamming distance of 2 with a delay insensitive decoding function is used on the links. At node i, port $S_{i,j}$ is fed into memory flags $Mem_{i,j,s}$, which are set when node i has seen node j in state s since the last reset by node i's state machines. Figure 3.2 depicts the prototype implementation of the an input and output port between two nodes and the associated memory flags of the main algorithm. It is apparent that the memory flags implement buffers for the individual states conveyed via the link. Note that some state (e.g., Actual ACCEPT) is also available without buffering.

---

[1] The general definitions of Chapter 2.1.2 apply, so only the most important differences in notation and some more detailed informal explanations FATAL$^+$ are provided.

A statement in a transition guard like *in* s refers to some state machine being in state s. Similarly, if a state is used without the keyword *in*, it refers to the output port of the corresponding memory element. Hence, a guard like $(\geq n - f s$ **or** $)$*in* s' will become true at a node i if at least $n - f$ of node i's memory elements of state s are set, i.e., $\left(\sum_{j \in V} Mem_{i,j,s}\right) \geq n - f$, or some local state machine has the state s'. Besides, a transition of the FSM will perform some additional action, i.e., it resets all memory elements specified as a edge label in a box along with the transition guard (see Figures 3.3 and 3.5).

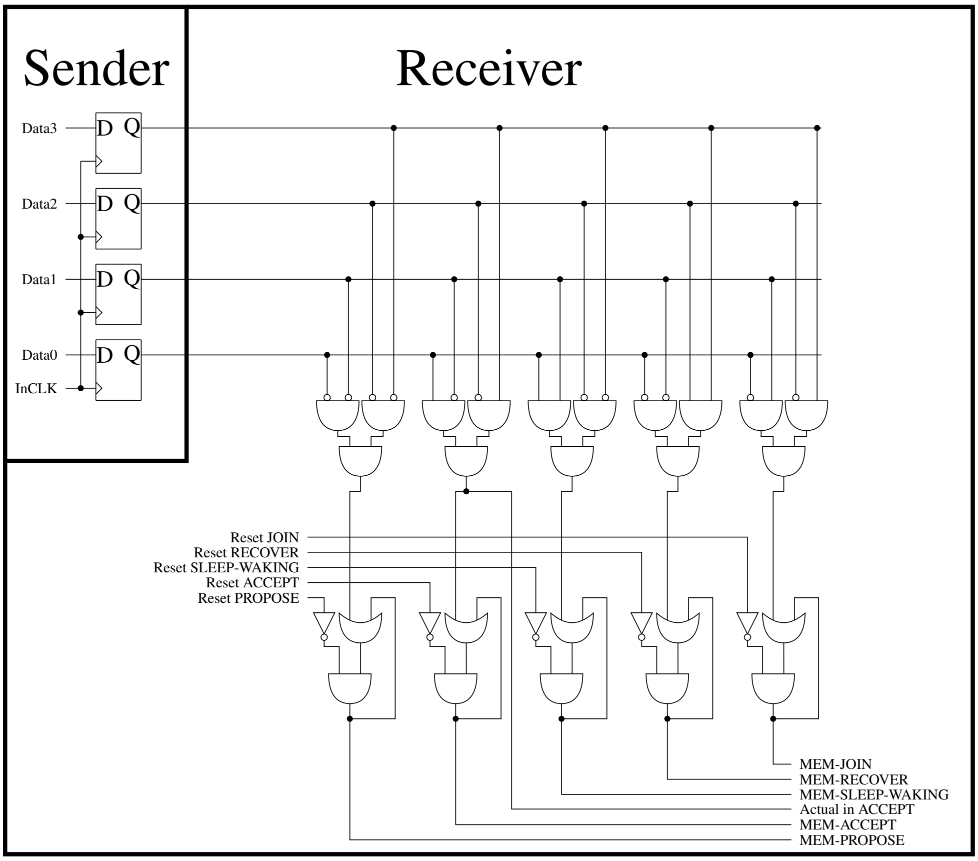


Figure 3.2: Memory flags sender and receiver port implementations of the main algorithm. [66]

Threshold modules

Since some transition guards require that at least a certain number (i.e., the threshold) of memory flags of some specific states to be set, threshold modules set their output once this threshold is reached (e.g., at least $n - f$ memory flags are set). Due to the asynchronous evaluation of the transition guards using combinatorial logic, the threshold modules' output ports must be glitch-free. In the current FATAL^{+} prototype, these modules are implement using the *Sum of Products* approach (see Chapter 4.2).

State machines

The FATAL$^+$ state machines are implemented by means of HSMs introduced in Chapter 2.6.3. The transition combinatorial functions, which determine which transition of the FSM shall be performed and enable the TSM clock, have been generated by Petrify using STG specifications of the FATAL$^+$ protocol (see Chapter 3.1.2).

Metastability

The HSM and the FATAL$^+$ algorithm itself have been designed to make the window of vulnerability for metastable upsets small. Consequently, on average, such upsets are very rare and can only delay the stabilization. An analysis of possible upsets in the current FATAL$^+$ implementation can be found in [26]. However, analytical results on the upset probability are still lacking.

Once the system has stabilized, it is guaranteed to be metastability-free due to the used timeouts and the delays and clock ratios covered by them: The HSM is started safely as all signals are supplied long enough to be stored correctly by the memory flags and no glitches will occur. Furthermore, in case of multiple outgoing transitions, only one transition will be enabled at a time and thus no tie-breaking is needed.

The only way metastability may be introduced after stabilization is by faulty nodes, which may send runt pulses and upset the memory flags. This in term, may produce undefined behaviour of the HSM processing the corresponding transition guards. Nonetheless, the probability of metastability propagation is small (due to the threshold modules and the TSM), and additional synchronizers or elastic pipelines in the links can be used to decrease the probability of metastable upsets due to faulty nodes.

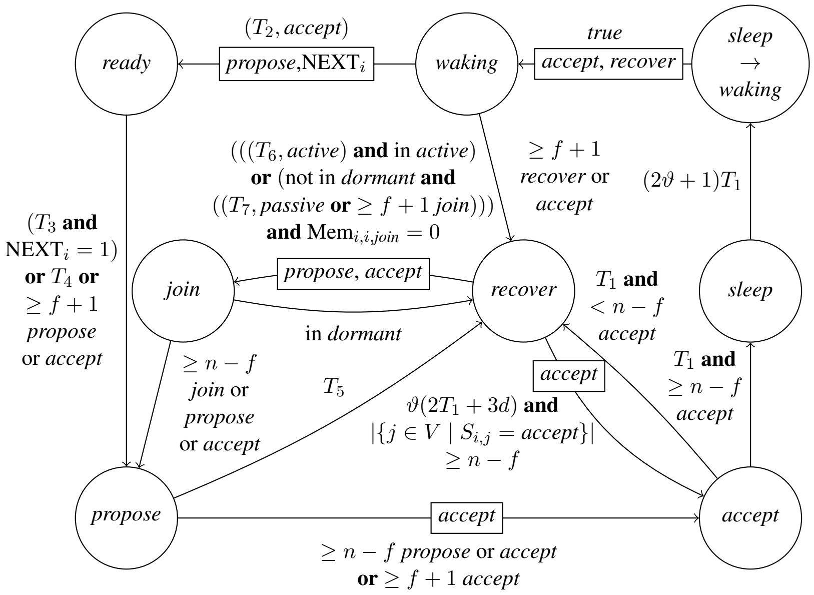
3.1.2 Protocol

FATAL$^+$ assumes a fully connected network of n nodes in the bounded delay asynchronous model. All link delays, including the computation delays, and the clock ratio ϑ are known and bounded. The adversary may choose any initial state of the distributed system and up to f Byzantine faulty nodes: link failures are modeled as node failures. If the number of node failures exceeds f, the whole system may need to restabilize.

The main actual parameters influencing the timeouts and execution of the protocol algorithms are:

Delays: The maximum delay d between the launch of a new state on an output port sent over a link by another node or an internal component of a node (e.g., timeout) until it is safely stored (link delay, decoding/combinatorial delay and memory flag set/reset delay), including the necessary time to perform a transition if the a guard became enabled by that signal (transition guards combinatorial functions, clock resume delay and TSM delay). Since the Quick-cycle algorithm is likely to have smaller delays it provides a further distinction into d^+_{max} and d^+_{min}, which represents the maximum resp. minimum delay within the Quick-cycle algorithm only.

Clock ratio: Is defined by ϑ like before.



$(T_2, accept)$
ready
propose,NEXT$_i$
waking
true
accept, recover
sleep → waking
$(((T_6, active)$ **and** in active) **or** (not in dormant **and** $((T_7, passive$ **or** $\geq f+1$ join))) **and** Mem$_{i,i,join} = 0$
$\geq f+1$ recover or accept
$(2\vartheta+1)T_1$
(T_3 **and** NEXT$_i = 1$) **or** T_4 **or** $\geq f+1$ propose or accept
join
propose, accept
recover
T_1 **and** $< n-f$ accept
sleep
in dormant
$\geq n-f$ join or propose or accept
T_5
accept
$\vartheta(2T_1+3d)$ **and** $|\{j \in V \mid S_{i,j} = accept\}| \geq n-f$
T_1 **and** $\geq n-f$ accept
propose
accept
accept
$\geq n-f$ propose or accept **or** $\geq f+1$ accept

Figure 3.3: Overview of the core routine of node i's self-stabilizing pulse algorithm. Once stabilized only the outer cycle is performed. [26]

FATAL pulse synchronization

The main algorithm, which is responsible for generating the slower synchronized FATAL-pulses (see Figure 3.4), progresses through the states depicted in Figure 3.3. The inner core restabilizes the system or (late) joining nodes, should there ever be a loss of synchronization. There are further algorithms not shown here, which the main algorithm relies on to implement the actual resynchronization mechanisms, using randomly chosen timeouts to cope with Byzantine faults. Note that once the FATAL algorithm has stabilized, all correct nodes execute the outer cycle of the main algorithm, which is the main source of the accuracy and precision/skew bounds.

Put precisely, the timeouts of the main algorithm ensure, that all correct nodes enter the state *ready* resp. *accept* (i.e., last pulse) before the first node leaves it. Once the a node reaches ready, there are two timeouts T_3 and T_4 that may lead the first correct node to switch to the state *propose*. T_3 can be much smaller than T_4 and allows an application layer to provide the $NEXT_i$ trigger to force the algorithm to increase the pulse frequency, i.e., the application layer is ready for the next pulse at node i. Due to the clock ratio ϑ and the skew, some nodes will leave the state earlier than others, i.e., they are fast. After a node is in propose, a node may generate a new pulse, i.e., switch to accept, either once it has seen at least $f+1$ correct nodes in propose or accept or if it has seen at least one correct node in accept. So, when the first node enters accept, all other nodes have seen at least $f+1$ correct nodes in propose and switched to propose within d. Therefore all nodes have seen $n-f$ correct nodes in accept and switch to accept within d.

Hence, any node will switch to accept within $2d$ after the first correct node switched to accept. At this point, a new iteration of this cycle begins.

To sum up, the main algorithm solves the pulse synchronization problem (see Chapter 2.1.5) and Figure 3.4) with up to f Byzantine failures within stabilization time $T(k) \in \mathcal{O}(kn)$ with a probability of at least $1 - 2^{-k(n-f)}$, for some $k \in \mathbb{N}$. If at least $n - f$ correct nodes are synchronized other correct nodes synchronize within $\mathcal{O}(R_1)$ deterministically. After stabilization it guarantees the accuracy $[T_{\text{min}}, T_{\text{max}}] = [(T_2 + T_3)/\vartheta - 2d, T_2 + T_4 + 7d]$ and the precision (skew) $\Sigma = 2d$. Obviously, the minimum FATAL-pulse width is T_1/ϑ.

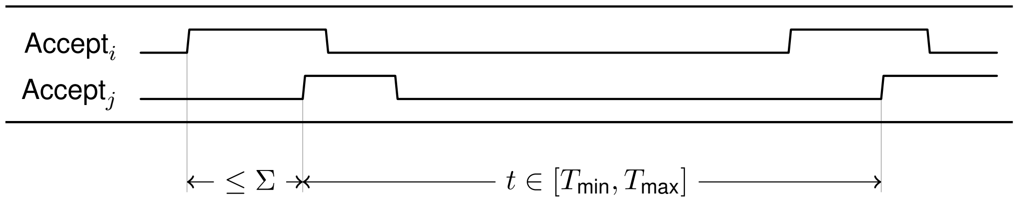


Figure 3.4: Interface timing to provided by *FATAL* to higher level algorithms.

The FATAL timeouts must meet the following constraints to ensure stabilization and correct operation:

$$\lambda = \sqrt{(25\vartheta - 9)/(25\vartheta)} \in (4/5, 1) \tag{3.1}$$

$$\Delta_g = (2\vartheta + 3)T_1 + 2d \tag{3.2}$$

$$T_1 \geq 4\vartheta d \tag{3.3}$$

$$T_2 \geq 3\vartheta\Delta_g + 7\vartheta d \tag{3.4}$$

$$T_3 \geq (2\vartheta^2 + 4\vartheta)T_1 - T_2 + \vartheta T_6 + 7\vartheta d > (\vartheta - 1)T_2 + 6\vartheta d \tag{3.5}$$

$$T_4 \geq T_3 \tag{3.6}$$

$$T_5 \geq \max\{(\vartheta - 1)T_2 - T_3 + \vartheta T_4 + 7\vartheta d, (\vartheta - 1)T_1 + \vartheta(T_2 + T_4) - T_6\} \tag{3.7}$$

$$T_6 \geq \vartheta T_2 - 2\vartheta T_1 + 2\vartheta d > (2\vartheta^2 + 3\vartheta)T_1 + 6\vartheta d \tag{3.8}$$

$$\begin{aligned} T_7 &\geq (4\vartheta - 2)T_1 + \vartheta(T_2 + T_4 + T_5) + T_6 + 2\vartheta d \\ &> (2\vartheta^2 + 7\vartheta - 2)T_1 + \vartheta(T_2 + T_4 + T_5 + 8d) \end{aligned} \tag{3.9}$$

$$R_1 \geq \max\left\{\vartheta T_7 + (4\vartheta^2 + 8\vartheta)d, \vartheta(2T_1 + 2T_2 + 2T_4 + T_5 + 12d) - 2T_1\right\} \tag{3.10}$$

$$R_2 \geq \frac{2\vartheta(R_1 + 6\Delta_g + T_1 + (8\vartheta + 11)d)(n - f)}{1 - \lambda} \tag{3.11}$$

$$\begin{aligned} R_3 = {} & \text{uniformly distributed random variable on} \\ & [\vartheta(R_2 + 3d), \vartheta(R_2 + 3d) + 8(1 - \lambda)R_2] \end{aligned} \tag{3.12}$$

$$\lambda \leq \frac{T_2 - 2\vartheta\Delta_g - (\vartheta - 1)T_1 - 2\vartheta d}{T_2 - (\vartheta - 1)T_1 - \vartheta d}. \tag{3.13}$$

FATAL$^+$ pulse synchronization

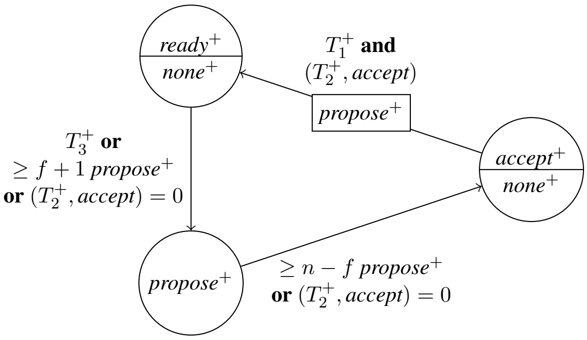


Figure 3.5: The quick cycle of the FATAL$^+$ protocol. [26]

The attentive reader should have noticed that the main-cycle is rather large and depends on a lot of timeouts. Therefore the Quick-cycle algorithm in figure 3.5 is used to perform a shortened Main cycle to generate faster pulses. Such a pulse is generated at a node by entering the state *accept*$^+$. Moreover, it provides a special counter, which increments at every Quick-cycle pulse modulo M (see Figure 3.6). Since the counter shall implement a BSCLK, it is desired that the Quick-cycle algorithm resp. the counter are held at the state accept$^+$ resp. at 0 until the next FATAL-pulse occurs.

For this purpose, the special timeout T_2^+ is used to reset the counter and move the Quick-cycle algorithm into the state accept$^+$ if this has not been the case already. Hence, the Quick-cycle algorithm and the counter rely on the FATAL-pulse protocol for self-stabilization. Once the counter reaches 0, i.e., a overflow modulo M occurs at some node, it must not leave the state accept$^+$. This requires the FATAL main algorithm to switch to accept and reset T_2^+ before T_1^+ fires. Thus, M must be large enough s.t. M Quick-cycle pulses fit between two consecutive FATAL-pulses, the main algorithm must be in the state ready and the timeout T_3 must have fired. Once the counter overflows, the Quick-Cycle algorithm can issue a $NEXT_i$-pulse to start the transition of the main algorithm towards the accept state. Basically, T_1^+ must be just large enough to cover this transition of all correct nodes to accept as well as accept$^+$. On the downside, if M is increased it may be necessary to slow down the generation of the FATAL-pulses by increasing the timeouts. Hence, the stabilization time of the FATAL-algorithm depends linearly on M.

In this shortened main-cycle all correct nodes enter the state accept$^+$-state within $\Sigma^+ = 2d^+_{\text{max}} - d^+_{\text{min}}$. The current FATAL$^+$-prototype utilizes a synchronous state machine for the BSCLK, which is driven by the rising transition of the FATAL$^+$-pulse as clock. Since a FATAL$^+$-pulse needs at most d time to reach the counter's synchronous state machine and perform the synchronous transition, all correct nodes increment the counter within $\Sigma^b = \Sigma^+ + d$. Therefore, the counter implements a BSCLK with skew Σ^b. Note that this skew may be covered by Σ^+, if desired, by setting d^+_{max} appropriately to include the additional delay. However, this skew is lower than the period between to consecutive FATAL$^+$-pulses. Hence, the BSCLK may be used to directly implement lock-step synchronous rounds. Besides, the i^{th} counter bit implements a clock frequency divider, since it increments every 2^i FATAL$^+$-pulses.

For example, the application layer may use the LSB of this counter ($i = 1$) as clock signal to implement further synchronous algorithms. This synchronized FATAL$^+$ clock signal divides the FATAL$^+$-pulse frequency by 2 and toggles exactly M times, i.e., the upper counter bits overflow after $M/2$ clock cycles, each of which may be a full lock-step synchronous round if the resulting clock period is sufficient (see Chapter 4.3).

To sum up, the Quick-Cycle algorithm solves the pulse synchronization problem with up to f Byzantine failures within stabilization time $T(k) + T_1^+ + T_3^+ + \Sigma^+ + 3d + 5d^+_{\text{max}}$ with a probability of at least $1 - 2^{-k(n-f)}$. After stabilization it guarantees the accuracy $[T^+_{\text{min}}, T^+_{\text{max}}] = [(T_1^+ + T_3^+)/\vartheta - \Sigma^+, T_1^+ + T_3^+ + 2\Sigma^+ + 3d^+_{\text{max}}]$ and the precision (skew) $\Sigma^+ = 2d^+_{\text{max}} - d^+_{\text{min}}$. Obviously the minimum FATAL$^+$-pulse width is T_1^+/ϑ.

More over it provides a BSCLK, which increase exactly by 1 mod M at each FATAL$^+$-pulse and is $\lceil ld(M) \rceil$ bits wide. Since the BSCLK is implemented by a synchronous state machine driven by the *accept*$^+$-pulse the BSCLK's skew is at most $\Sigma^b = \Sigma^+ + d$. Hence, the accuracy (i.e., the clock period) of the i^{th} BSCLK bit is $\left[2^i T^+_{\text{min}}, 2^i T^+_{\text{max}}\right]$, where $i = 1$ represents the LSB.

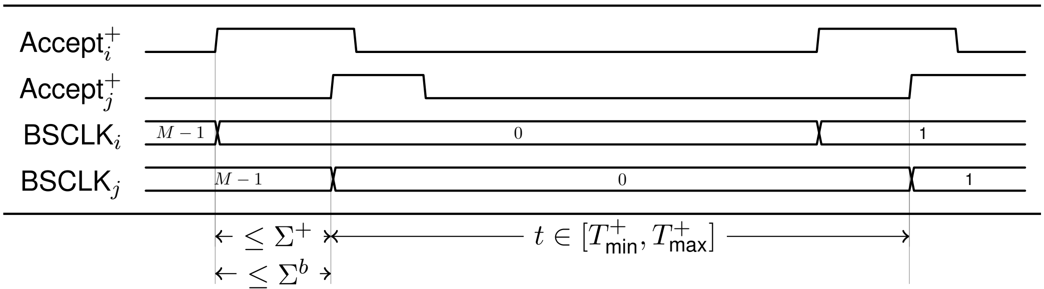


Figure 3.6: Interface timing to provided by *FATAL*$^+$ to higher level algorithms.

Note that the BSCLK is zero and the Quick-cycle algorithm is in the state accept$^+$ once the FATAL-pulse is generated. The Quick-cycle timeouts must meet the following constraints:

$$T_1^+ \geq \vartheta(T_2^+ + \Sigma^+ + 3d + d^+_{\text{max}}) \tag{3.14}$$

$$T_2^+ \geq \vartheta(3d + 3d^+_{\text{max}}) \tag{3.15}$$

$$T_3^+ \geq \vartheta(T_1^+ + d^+_{\text{max}}) \tag{3.16}$$

$$M \in \left[\frac{\vartheta(T_2+T_3+3d)+T_1^+-T_2^+}{T_1^++T_3^+}, \frac{T_2+T_4-3\vartheta d}{\vartheta(T_1^++T_3^++\Sigma^++4d^+_{\text{max}})}\right] \tag{3.17}$$

3.2 (M, l)-labeling problem

The following chapter provides an overview of the labeling problem and existing algorithms. Afterwards the high-level code of the solution algorithm, its proofs and explanations of its behaviour are provided.

The previous introduced FATAL$^+$ BSCLKs are synchronized to each other with a skew Σ^b and increment with a period from the interval $\left[T^+_{\text{min}}, T^+_{\text{max}}\right]$, i.e., at every FATAL$^+$-pulse. At the M^{th} FATAL$^+$-pulse the BSCLK overflows and obtains the value 0 again. This overflow occurs right before the next FATAL-pulse and the BSCLK remains at 0 until then. Unfortunately, these M FATAL$^+$-pulses need to fit between two consecutive slow FATAL-pulses, i.e., the stabilization time of both FATAL and FATAL$^+$ increases linearly with M. Therefore, the Quick-cycle algorithm is only able to provide small size BSCLKs (of width 8 to 9 typically), which are not sufficiently wide to be used as global time base due to fast overflows.

In order to generate a wide global time base, some algorithm is required that is able to extend the short BSCLKs by l upper bits, called labels L_i. These labels must increment at each BSCLK overflow at a correct node and must be equal at all correct nodes once all BSCLKs overflowed, i.e., they are synchronized with a skew Σ^l and increment in lock-step manner to implement a suitable extension.

The labeling algorithms are run on n independent nodes, where each node is provided with one BSCLK modulo M. In this case the BSCLKs are generated by the n nodes of the FATAL$^+$ algorithm. The n nodes are deployed in a fully connected network, i.e., each node has a separate input port for each node's output port, which are connected via bounded delay 1-bit links only for scalability.

The algorithm solving this (M, l)-labeling problem (or just called labeling problem) must be self-stabilizing once the underlying BSCLK stabilized, and tolerates up to $f \leq \lfloor (n-1)/3 \rfloor$ Byzantine faulty nodes during both stabilization and operation. Formally, the primary convergence and closure property hold eventually:

Definition 3.2.1. *(1) At each overflow of correct node i's BSCLK and correct node j's BSCLK $L'_i = L'_j \bmod 2^l$. As long as the previous property is satisfied by at least at $n - f$ correct nodes these nodes are called stabilized and must satisfy the following secondary closure property:*
(2) At each following overflow of the short BSCLK the new label is $L'_i = (L_i + 1) \bmod 2^l$ at a correct node i.

These properties ensure that a node recovering from a failure (late-joining) adapts the label of a already correct stabilized node instead of allowing a recovering or faulty node to reset all labels.

The attentive reader should have noticed that these properties define a Byzantine fault-tolerant self-stabilizing clock synchronization algorithm using exact agreement on each BSCLK overflow resp. FATAL-pulse. Due to inevitable non-zero skew, the combined clock of the label value and BSCLK value can provide approximate agreement with precision $\pi \geq 0$ only. If Σ^l can be kept smaller than the minimum FATAL$^+$-pulse period T^+_{min}, the combined precision is bounded by $\pi \leq 1$.

3.3 Related Work

In this chapter, a brief overview on existing Byzantine self-stabilizing clock synchronization algorithms, for solving the labeling problem, based on lower level pulse synchronization protocols, is provided. The ideal solution would match the requirements of VLSI circuits, i.e., constant sized messages, only basic building blocks and fast stabilization, i.e., the algorithm completes at the every BSCLK overflow (resp. slow FATAL-pulse).

3.3.1 Non-Byzantine fault-tolerant algorithms

Classic clock synchronization algorithms [55] exchange the local (real-time) clock values and compute a simple arithmetic average function from the difference of the received values to the own clock. This result can be refined for real-time clocks by adding the expected communication and adding the computation delay to this candidate (e.g., $(d_{\text{max}} - d_{\text{min}})/2$). In order to improve the synchronization precision the algorithms may also use a ping-pong mechanisms to estimate and measure the link delays between two nodes, by capturing the ping send and the pong receive time instants at the requesting node.

Alternatively, all nodes may perform a single majority vote on (each bit of) the clock values to select a candidate or select a single clock value [49] (e.g., the maximum) among the received values.

These approaches work well as long as some initial synchronization exists. However, as soon as some nodes are Byzantine faulty, by sending inconsistent resp. incorrect messages, these naive approaches will not stabilize from all initial states, since the correct nodes may not share the same set of input values. Especially, the approaches using simple averaging or which select the maximum from all received values can not tolerate Byzantine faulty nodes, even if $n - f$ nodes are correct and should share the same label at a BSCLK overflow.

3.3.2 Byzantine fault-tolerant algorithms

Early fault-tolerant algorithms were designed to be used by processors with free running real-time clocks, i.e., a local counter, which increments regularly at a chosen local clock transition. A synchronization algorithm reads the local real-time clock and distributes it via the communication network to other nodes. Usually, all nodes receive these local clock values within bounded time, upon which the algorithm computes a convergence function. The basic convergence functions are purely arithmetic, i.e., the convergence function computes either a correction term for the clock rate or the real-time clock value. A popular convergence function, is the fault-tolerant midpoint algorithm (FTM) [71]. Basically, algorithms using this approach first compute the differences of the local clock to the received clock values and sort these differences afterwards. The f largest and smallest values are discarded and the FTM selects the median of the remaining set of differences as correction candidate. Some protocols take even clock drift estimations into account to increase the precision further [63]. Unfortunately, these convergence functions can only provide approximate agreement (precision $\pi \geq 0$) and not exact agreement at each BSCLK overflow, as required by the labeling problem. Furthermore, some protocol implementations rely on initially synchronized real-time clocks or rather strong communication network primitives, which ensure that Byzantine nodes are not able to forge messages of correct nodes resp. distribute inconsistent information, but only incorrect clock values [54, 63]. Hence, they are not implicitly Byzantine self-stabilizing in the weak bounded delay model. Besides, all nodes need to exchange linear sized messages and keep the whole received clock values in their memory.

Another approach [4], is to allow a node access to the labels of other/adjacent nodes directly and proceed through these in a round robin manner at every pulse. The the execution of this algorithm needs not to be finished at the next pulse, i.e., it can take multiple pulses before it stabilizes. On the downside, it requires linear sized messages again and more complex arithmetic resp. sorting functions and can not tolerate Byzantine faults during stabilization.

3.3.3 Byzantine fault-tolerant and self-stabilizing algorithms

In order to establish exact agreement some form of a consensus algorithm is needed. There has already been put some effort into Byzantine self-stabilizing agreement algorithms based on pulse synchronization protocols for clock synchronization. Note that pulses correspond to the overflows of BSCLKs.

The algorithm presented in [44], for example, uses k Byzantine (non self-stabilizing) consensus instances in parallel at each node. Thereby, it takes into account that a consensus protocol may need multiple pulses (overflows) to complete (e.g., k). The oldest instance is restarted at each pulse (overflow). Besides, the output of the oldest instance is used to determine if the the current label value shall be incremented or not. After k pulses, it is guaranteed that all correct processors execute the consensus algorithms consistently, i.e., the algorithm execution stabilizes. Although the algorithm does not need to be finished at the next pulse, k depends on the number of nodes n due to the chosen (deterministic) consensus algorithm, and makes use of linear sized messages as well as a separate communication channel for each consensus instance.

The algorithms proposed in [9, 28] require that one iteration of the algorithm finishes at the next pulse (overflow). Thus, a node may use the pulse to consistently reset the algorithm instance into the initial state for self-stabilization. The Byzantine consensus algorithms [9, 28] provide probabilistic consensus solutions based on a short bounded synchronized clock. Still these algorithms exchange the full labels and perform a (majority) vote among the received labels. In the last phase, a process that is not convinced by the voting outcome will toss a coin to either reset resp. keep the outcome of the voting process. Despite linear sized messages, some random number generator is required to provide a (perfect) coin tossing process. Nonetheless, [9] offers a expected constant convergence time, whereas [28] can only provide exponential convergence time (in the number of label bits l).

Currently, the only suitable algorithm for $n > 3f$, is the solution presented in [39]. It is based on [17], hence suffers from linear sized message like the already mentioned algorithms. [39] bypasses this problem, by first reducing the labeling problem to binary consensus. The reduction and binary (Phase King) consensus (see Chapter 2.1.5) require messages of size 1 only, i.e., a serial (1-bit) channel. Moreover, it only consist of relative small logic blocks compared to the other solutions. Hence, the proposed algorithm has been implemented and evaluated in this thesis and is explained in the next chapter.

3.4 Solution algorithm

The labeling algorithm (Algorithm 1) uses 1-bit channels to broadcast partial state information between the nodes. As noted in the previous chapter, other clock synchronization algorithms send their whole label (clock) in one message and calculate their correction terms from the received labels. In this case, only one bit of the label is considered at a time, which can result in large message and hardware complexity reduction.

At first the problem instance is reduced to binary consensus. The reduction outputs a label candidate c_i and a boolean trust value b_i, which is set if the node could identify a candidate. The reduction result b_i is then fed into a binary consensus algorithm. If the consensus outputs ***true*** the node will keep and increment its label candidate c_i otherwise if the output is ***false*** it will reset the candidate to a predefined value.

A new message is always sent (i.e., broadcast) to every node, including itself, if the node and its links are correct. In the provided algorithms a new round is initiated after a broadcast statement. If a broadcast sends multiple bits at once each bit is being sent separately and a new round is initiated by each bit. Note that the algorithm is executed using the (simulated) lock-step synchronous round execution model on top of BSCLKs. Hence, the skew and delays can be assumed to be 0 at the algorithm level. For now, assume that M lock-step synchronous rounds are available to the algorithm using the BSCLKs.

Algorithm 1: (M, l)-round labeling algorithm at node $i \in V$. $\mathcal{P}$ is a consensus algorithm. [39]

```
while true do
    $(c_i, b_i) :=$ red$(L_i)$ //reduction
    $o_i := \mathcal{P}(b_i)$ //run consensus
    if $o_i =$ false then $c_i := 1$ //agree on default
    else $c_i := c_i + 1 \mod 2^l$ //agree on $c_i$
    wait until round number modulo $M$ is 0
    $L_i := c_i$ //update label
```

Algorithm 2 is used as the reduction algorithm. It is split in two phases, consisting of a for-loop of l rounds each. In the first loop it will try to determine if enough (at least $n - 2f$) other correct nodes share the same label. If the label is not equal to its own input or not enough correct nodes share the same label it will exit the first loop with $b_i =$ ***false*** at a correct node i. A correct node will tell the others if it could identify a candidate c_i. If a correct node could not identify a candidate its messages will be ignored by other correct nodes in the second for-loop (i.e., it is disabled). The set of nodes, whose messages shall be considered, is represented by S_i. Hence, all (enabled) correct nodes broadcast the same candidate label bits in the second for-loop. The second loop guarantees that if one correct node i exits with $b_i =$ ***true*** that all correct nodes exit with the same label candidate.

Since a failed node might be correct in the next iteration of the labeling algorithm, e.g., due to transient fault, and needs to resynchronize to other nodes without affecting their labels, I need to strengthen the validity condition of the (binary) consensus definition given in Chapter 2.1.5. For

Algorithm 2: Algorithm red(L_i): reduces (M, l)-round labeling to binary consensus. [39]

```
input : label $L_i$ ($l$ bits)
output: candidate clock value and boolean trust value $(c_i, b_i)$
$c_i := L_i$ //holds candidate label
$b_i :=$ true
for $j \in \{1, \ldots, l\}$ do
    broadcast $c_i(j)$ //$j^{th}$ bit of candidate value
    if received $\geq n - f$ times value $c = c_i(j)$ then
        $c_i(j) := c$
    else
        $b_i :=$ false
broadcast($b_i$)
store set $S_i$ of nodes that sent true
$b_i :=$ true
for $j \in \{1, \ldots, l\}$ do
    broadcast $c_i(j)$
    if received $\geq n - f$ times c from $S_i$ then
        $c_i(j) := c$ //all others see $\geq f + 1$ times $c$
    else if received $\geq f + 1$ times c from $S_i$ then
        $b_i :=$ false // input values differ
        $c_i(j) := c$ //candidate bit still known
    else $b_i :=$ false //inputs differ, known to all
return $(c_i, b_i)$
```

example, if this failed node recovers right at the time the binary consensus begins, a consensus algorithm that only satisfies this weaker validity property is allowed to choose some input of a correct node, i.e., also the input of the recovering node, because it executes the binary consensus algorithm faithfully. Thus, it may reset the label of the already $n - f$ correct synchronized nodes if its input is ***false*** and the consensus algorithm determines this input as output for all correct nodes. Therefore, any employed binary consensus algorithm must satisfy the following strengthened validity property:

Validity: The result must be the input of a correct node and if at least $n - f$ correct nodes share the same input the result must be the input of these nodes.

Algorithm 3 implements the Phase-King consensus algorithm. In each iteration of this algorithm, consisting of 3 resp. 4 rounds each (i.e., 3 2-bit resp. 4 1-bit broadcasts), a correct node verifies if enough correct nodes (at least $n - 2f$) propose the same value. If so, each of these correct nodes will transmit the same identified value and the remaining correct nodes indicate that no candidate has been identified. If a node is able to verify that enough correct nodes (at least $n - 2f$) transmitted an identified value they will update their candidate to the identified value. Due to Byzantine faulty nodes, there might still be an inconsistency among the correct nodes

Algorithm 3: Algorithm $\mathcal{P}(b_i)$: Phase-King consensus algorithm. [10]

```
input  : $b_i$ //binary value to be decided on
output: $o_i$ //binary value representing the consensus decision.
for $k \in \{1, \dots, f+1\}$ do
    broadcast $b_i$ //identify candidate value
    if received $\geq n - f$ times value $b$ then
        broadcast $1b$ //bc identified candidate
    else
        broadcast $00$ //bc no candidate identified
    if received $\geq n - f$ times value $1b$ then
        $b_i := b$ //identified value
        broadcast $b$ //Phase King's bc
    else
        if received $\geq f + 1$ times value $11$ then
            broadcast $1$ //Phase King's bc, convince nodes with $0$
        else
            broadcast $0$ //Phase King's bc, convince nodes with $1$
        $b_i :=$ received $b$ from $k$ //assign value from Phase King
return $b_i$
```

and thus each correct node, which couldn't update its candidate, will listen to the broadcast of the current Phase-King (i^{th} node, for the i^{th} iteration).

Since the prototype system consists of $n = 8$ nodes the Phase-King algorithm was the natural choice as a low effort (time complexity and message complexity) and optimal resilience algorithm. Note that from the algorithmic point of view it is not always necessary to perform a broadcast (e.g. Phase-King), but it makes it easier to understand where a new round is initiated without inserting commands to indicate the end of a round.

3.5 Proofs

In order for this thesis to be self-contained I will provide the correctness proofs of the above algorithms and their informal description taken from [39].

At first the properties that are satisfied after the first for-loop in the reduction algorithm are proofed by Lemma 3.5.1. *(1)*: If at least $n - f$ correct nodes execute the reduction with the same label L, these nodes identify a candidate by setting b_i. *(2)*: If a correct node identifies a candidate c_i is equal to its own input. *(3)*: All correct nodes identify the same candidate label.

Lemma 3.5.1. *Algorithm 2 satisfies the following post-conditions after the first for-loop:*

1. $(\exists L : |S = \{i | L_i = L \wedge i \textit{ is correct}\}| \geq n - f) \Rightarrow \forall i \in S : b_i$
2. $\forall \textit{ correct } i : (b_i \Rightarrow c_i = L_i)$

3. $\forall$ *correct* $i, j : (b_i \wedge b_j \Rightarrow c_i = c_j)$

Proof. If at least $n - f$ correct nodes enter the first loop with their labels being equal to L, including a correct node i, node i will receive at least $n - f$ times c in the k-th iteration with $c = c_i(k) = L(k)$ and keep $b_i =$ ***true***. Note that a correct node j with $c_j(k) = L_j(k) \neq L(k)$ will execute the **else**-statement and set $b_j =$ ***false*** (proves *(1)*).

Assume some correct node i exits the first loop with $b_i =$ **true**. Then, i executed the **if**-statement in all iterations and thus received at least $n - f$ times c in each iteration with every bit being equal to its own input $c_i = L_i$ (proves *(2)*).

Assume in contradiction to *(3)* that $c_i \neq c_j$. By *(2)* the candidates after the loop are $c_i = L_i$ and $c_j = L_j$. The values broadcast by i and j in the k^{th} iteration of the first for-loop equal $c_i(k)$ resp. $c_j(k)$, it must hold that $\exists k : L_i(k) \neq L_j(k)$, since a node accepts only its own input. W.l.o.g. node i executed $c_i(k) := c$ and node j $c_j(k) := \neg c$. As a result, all correct nodes have received at least $n - 2f > f$ times c from correct nodes, hence at most $n - f - 1$ times $\neg c$. Contradiction that j accepted its own input and $L_i(k) \neq L_j(k)$ (proves *(3)*). ☐

The following Theorem 3.5.2 shows that if at least $n - f$ nodes enter the reduction with the same label L all correct nodes exit with the label L and the trust value $b_i =$ ***true***. Besides, if at least one correct node exits with $b_i =$ ***true*** all correct nodes hold the same candidate label.

Theorem 3.5.2. *Algorithm 2 terminates in* $2l + 1$ *rounds and satisfies the following post-conditions:*

1. $(\exists L : |S = \{i | L_i = L \wedge i \text{ is correct}\}| \geq n - f) \Rightarrow \forall$ *correct* $i : c_i = L \wedge b_i$

2. $\forall$ *correct* $i, j : (b_i \Rightarrow c_i = c_j)$

Proof. By Lemma 3.5.1, if at least $n - f$ correct nodes enter the first loop with the same label L those nodes exit the first loop with the label L and $b_i =$ ***true*** and thus any correct node i will receive at least $n - f$ times c from correct nodes in S_i in the k-th iteration of the second for-loop with $c = c_i(k) = L(k)$, which proves *(1)*.

For proving *(2)* assume some correct node i exits the second for-loop with $b_i =$ **true**. Thus node i has executed the **if**-statements and received at least $n - f$ times c from nodes in S_i in each iteration of the second loop. Assume in contradiction that some correct node j exited with $c_i \neq c_j$ and thus $\exists k : c_i(k) \neq c_j(k)$. Since there are at least $n - 2f \geq f + 1$ correct nodes in S_i hence node j shares at least $f + 1$ of these correct nodes in S_j as well. By Lemma 3.5.1 *(3)* all correct nodes in S_i broadcast the same value c in the second loop's k-th iteration, since a node r can only be in S_i if $b_r =$ ***true*** after the first loop. Hence, node i executed Line 15 and node j executed Line 15 or 18 and set $c_i(k) = c_j(k) = c$. Contradiction that $c_i(k) \neq c_j(k)$.

The termination time is calculated by summing the iterations of all loops and the intermediate broadcast in between. ☐

Obviously, it is not necessary to enforce that $c = c_i(j)$ in Line 5 to be able to reduce correctly, but it is a logical choice to either synchronize all correct nodes to some input of a correct node or reset them all. Furthermore, it will simplify the implementation of a Byzantine-faulty node. Note that the reduction algorithm would solve the labeling problem even without consensus if less than $n - 2f$ nodes enter with the same labels, since all correct nodes will exit with $(c_i, \textbf{false})$.

At this point, I switch to Algorithm 3 to prove the remaining properties. Lemma 3.5.3 shows that if at least $n - f$ correct nodes start a phase (i.e., an iteration of the for-loop) with the same candidate value b (i.e., they prefer b) all correct nodes assign $b_i = b$ at the end.

Lemma 3.5.3. *If at least $n - f$ correct nodes prefer b at the beginning of an iteration in Algorithm 3, all correct nodes prefer b at the end of the iteration.*

Proof. Since there are at least $n - f$ correct nodes preferring b at the beginning of the iteration those nodes broadcast b in line 2 and all correct nodes receive at least $n - f \geq f + 1$ times b and thus at most $f < n - f$ times $\neg b$. Therefore all correct nodes broadcast $1b$ in line 4 and thus every correct node i receives at least $n - f$ times $1b$ and sets $b_i = b$. Hence, all correct nodes return b. ☐

Lemma 3.5.4. *If two correct nodes i and j execute line 4 in the k^{th} iteration of Algorithm 3, they broadcast $1b$ in the same iteration.*

Proof. Assume in contradiction that node i broadcasts $1b$ and node j $1\neg b$. Then, node i receives at least $f + 1$ times b from broadcasts of correct nodes in line 2. Thus, node j can receive at most $n - f - 1 < n - f$ times $\neg b$. A contradiction that node j broadcasts $1\neg b$. ☐

Lemma 3.5.5. *If the Phase-King node $k \leq f + 1$ is correct, then at latest at the end of iteration k in Algorithm 3 all correct nodes prefer the same value b.*

Proof. Case (1): Assume some correct node i receives at least $n - f$ times $1b$ from line 4 and sets $b_i = b$. Then, k (like all other correct nodes) receives at least $n - 2f \geq f + 1$ times $1b$ and at most f times $1\neg b$ (by Lemma 3.5.4). Hence, node k will broadcast b in line 9 resp. 12 or 14 and thus every correct node j will set $b_j = b$.

Case (2): Assume no correct node receives $n - f$ times $1b$. Then node k broadcasts $b = 1$ in line 12 resp. $b = 0$ in line 14 and every correct node j executes the **else**-statements and set $b_j = 1$ resp. $b_j = 0$. ☐

I will now prove that the Phase-King algorithm satisfies exact agreement, the extended validity property and termination at each correct node.

Theorem 3.5.6. *Algorithm 3 offers termination in $4(f + 1)$ rounds and satisfies the following post-conditions:*

1. $(\exists b : |S = \{i | b_i = b \wedge \textit{ is correct}\}| \geq n - f) \Rightarrow \forall i \in S : o_i = b$
2. $\forall$ *correct* $i, j : o_i = o_j$

Proof. Lemma 3.5.3 proves property *(1)*, since b is preferred initially by at least $n - f$ correct nodes and thus will be preferred over all iterations by any correct node, which implies property *(2)* in that case.

Property *(2)* is ensured by Lemma 3.5.5, because there is at least one iteration with a correct node as the Phase-King and thus all correct nodes prefer the same value b at the end of that iteration. Lemma 3.5.3 insures that this value will be preferred in all further iterations by any correct node.

Obviously, there are 4 broadcasts per iteration ($1b$ or 00 need 2 broadcasts on a single-bit channel), thus the algorithm terminates in $4(f+1)$ rounds. ☐

I will now prove the overall stabilization and closure properties of Algorithm 1:

Theorem 3.5.7. *Algorithm 1 needs* $M \geq 4(f+1)+2l+1$ *rounds per iteration, where* L_i *is the input label at the start of the iteration (at the current BSCLK overflow) and* L_i' *the output after one iteration (at the next BSCLK overflow) of a correct node* i*:*

1. $(\exists L : |S = \{i | L_i = L \wedge \text{ is correct}\}| \geq n - f) \Rightarrow \forall \text{ correct } i : L_i' = (L+1) \mod 2^l$
2. $\forall \text{ correct } i, j : L_i' = L_j' \mod 2^l$

Proof. Assume that at least $n-f$ correct nodes execute the iteration with the input label L, then any correct node i exits Algorithm 2 with ($c_i = L, b_i =$ **true**) by Theorem 3.5.2. Therefore, any correct node i exits Algorithm 3 with $o_i =$ **true** by Theorem 3.5.6. Thus, all correct nodes wait until the next overflow and set $L_i' = (L+1) \mod 2^l$, which shows the first and second property.

Assume less than $n-f$ correct nodes execute the iteration with the input label L, then any correct node i resp. j exits Algorithm 2 either with $c_i = c_j$ or $b_i = b_j =$ **false** by Theorem 3.5.2. In the first case, node i resp. j exits Algorithm 3 with $o_i = o_j$ by Theorem 3.5.6 and thus wait until the overflow and either set $L_i' = L_j' = (c_j + 1) \mod 2^l$ or $L_i' = L_j' = 1$. In the second case, any correct node i exits Algorithm 3 with $o_i =$ **false** by Theorem 3.5.6 and thus waits until the next overflow and sets $L_i' = 1$.

The runtime can be computed by summing the runtime of the above algorithms. ☐

It is obvious that property *(2)* of Theorem 3.5.7 implies the primary convergence and closure property and *(1)* ensures the secondary closure property of Definition 3.2.1, which completes the proof that the solution algorithm solves the labeling problem.

CHAPTER 4

Implementation

This chapter describes the implementation of Algorithms 1-3 presented in Chapter 3.4. Starting out from a low-level view on the necessary building blocks, I will give a state machine description of the entire solution. Furthermore, its connections to the prototype implementation of the FATAL$^+$ protocol are described.

4.1 Components

Since Algorithms 1-3 are synchronous distributed algorithms, the implementation will be designed using the synchronous design principle. Herein, the BSCLK of the FATAL$^+$ Quick Cycle algorithm provides the necessary lock-step synchronous round abstraction. Note that it would be also possible to implement parts asynchronously or by means of HSMs to improve performance, but due to the increased complexity of those approaches, the synchronous design principle is preferred. A first design analysis reveals that the following building blocks are required:

- Synchronous FSM
    - Synchronous state elements (flip-flops)
    - Next-State logic
- Threshold modules
- Synchronous transmission buffer (1-bit)
- Synchronous reception buffers (1-bit)

Figure 4.1 depicts the needed modules and required abstract interfaces of the labeling algorithm, including the lower-level FATAL$^+$ Quick Cycle module. The threshold module provides the necessary input for the next state logic of the FSM; it determines whether a certain set of received broadcasts occurred a certain number of times (i.e., the threshold). The FSM implements the necessary states (held in registers) and next state logic to progress through the Algorithms 1-3, i.e., processes the received broadcasts of the reduction and consensus algorithm to compute the new label. The FSM also provides the next broadcast value to the transmission buffer and controls the threshold module, i.e., it may disable/mask some of the inputs (like in Algorithm 2 in the second loop, which uses the set S_i to consider a subset of the received broadcasts only). Activities of the FSM to higher level algorithms are triggered by the BSCLK overflow (FATAL-pulse).

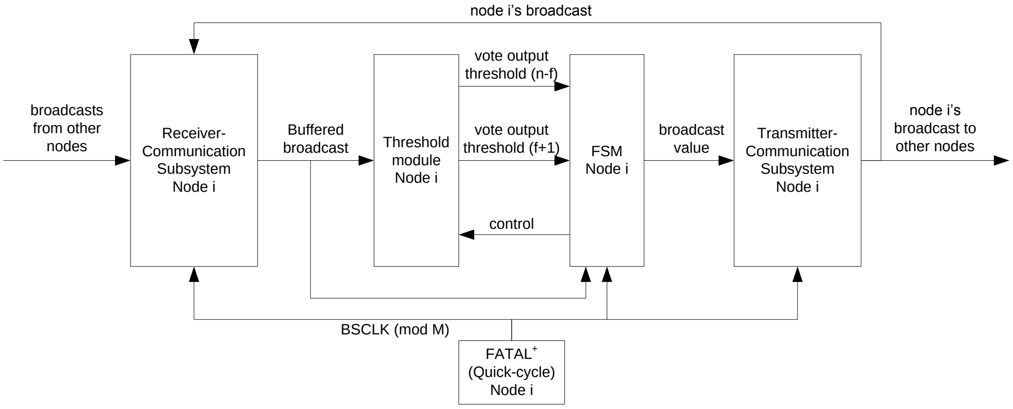

node i's broadcast
broadcasts from other nodes
Receiver-Communication Subsystem Node i
Buffered broadcast
Threshold module Node i
vote output threshold (n-f)
vote output threshold (f+1)
control
FSM Node i
broadcast value
Transmitter-Communication Subsystem Node i
node i's broadcast to other nodes
BSCLK (mod M)
FATAL+ (Quick-cycle) Node i

Figure 4.1: The modules needed to implement the labeling algorithm.

In the following sections, I will present some possible implementations for the major building blocks in Figure 4.1, as well as the variant eventually chosen.

4.2 Threshold Modules

A threshold module has n inputs taken from some finite input domain and a single binary output. It determines whether a certain value of the input domain appears a certain number of times (called the threshold) among these inputs. For example, if the input domain is binary $\{0,1\}$, a standard $k-$threshold module will output 1 if the input set contains at least k-times 1 and 0 otherwise, usually. It may be defined formally with the given input set $\mathcal{I} = \{(i,v)|i \in P \wedge v \in \mathcal{V}\}$, $|\mathcal{I}| = |\mathcal{P}| = n$ and input domain $\mathcal{V}$, where $i \in \mathcal{P}$ represents the unique input port and v the actual value of the input (i,v). Furthermore, let $\mathcal{I}_v = \{(i,v)|(i,v) \in \mathcal{I}\}$ and

$$thr(\mathcal{I},v,k) = \begin{cases} 1 & \text{if } |\mathcal{I}_v| \geq k \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

be the general threshold module. Since the output is invariant under permutation of the inputs, it is called a symmetric function [8].

The labeling algorithm needs a threshold module with a 2-bit input domain (see the thresholds in Algorithm 2 resp. 3) and two thresholds $n-f$ and $f+1$ for each of the input values $\{10,11\}$, where the input set size $n \geq 3f+1$. The output of $thr4(\mathcal{I})$ defined in (4.2) is a 3-tuple, where the first two elements are 1 if the threshold $n-f$ resp. the threshold $f+1$ is satisfied as required by the algorithms. The last element is 1 if the $f+1$-threshold for the input value 11 is reached and 0 otherwise; note that 11 has a higher priority than 10.

$$thr4(\mathcal{I}) = \begin{cases} (1,1,1) & \text{if } |\mathcal{I}_{11}| \geq n-f \\ (1,1,0) & \text{if } |\mathcal{I}_{10}| \geq n-f \\ (0,1,1) & \text{if } |\mathcal{I}_{11}| \geq f+1 \\ (0,1,0) & \text{if } |\mathcal{I}_{11}| < f+1 \wedge |\mathcal{I}_{10}| \geq f+1 \\ (0,0,0) & \text{otherwise} \end{cases} \tag{4.2}$$

This specification can be built using a general threshold module for each of the two interesting values and thresholds:

$$thr4(\mathcal{I}) = \begin{pmatrix} thr(\mathcal{I},11,n-f) \vee thr(\mathcal{I},10,n-f) \\ thr(\mathcal{I},11,f+1) \vee thr(\mathcal{I},10,f+1) \\ thr(\mathcal{I},11,f+1) \end{pmatrix} \tag{4.3}$$

Since only the two values 10 and 11 are of interest, the most significant bit can be seen as an input enable bit (i.e., its mask). It is thus possible to reduce the threshold modules to the standard threshold modules for binary inputs, as shown in Figure 4.2: Each input is compared (using a comparator function) against the two input values. The logic functions used as comparators are $ic0 \wedge ic1$ resp. $ic0 \wedge \neg ic1$ for the input value 11 resp. 10, where $ic1$ is the MSB and $ic0$ the LSB of the input. The outputs represent the 3-tuple (**nf,f1,c**) according to Equation (4.2). Note that MSB will be provided by the FSM state, i.e., the FSM selects if the upper bits represent the set S_i in the reduction algorithm resp. the dual bit broadcasts of the Phase King algorithm, but in the case of the remaining broadcasts the upper bits are set to 1 by default.

Since threshold modules are primarily used for fault-masking, the issue of glitches is important. From (4.1), it is apparent that almost simultaneous deletions and insertions of elements in the input set in case of $|\mathcal{I}_v| = k$ may generate glitches at the output - even if the threshold module implementation is glitch-free. For inputs supplied by correct components, this can sometimes be avoided by construction (suitable timing constraints). For faulty components, this can not usually be avoided entirely; however, as long as the threshold is not crossed, a glitch-free implementation will even mask faulty inputs. In synchronous design styles, glitches are masked by the clock as usual, and are hence less critical. Nevertheless, a glitch-free implementation could help masking against metastable upsets of synchronizers and is hence preferable also here.

A brief overview and a comparison of the implementation choices will follow. Since the size, i.e., amount of logic, and the easy adaptability are more important in the current prototype implementation, only the necessary parameters are considered here. More detailed descriptions and further analysis can be found in [8,14,37]. Especially, [14,37] have performed some analysis on the performance (delays) on a typical FPGAs. An analysis and formal model for glitches in threshold modules can be found in [45].

The effort to implement such a threshold module depends on the number of inputs, as well as the size of the input domain and the value of k. The typical complexities considered are gate, adder and/or comparator complexity, which give a hint on the required amount of logic (i.e., area). Furthermore, the depth, i.e., the maximum amount of such logic functions (stages) that

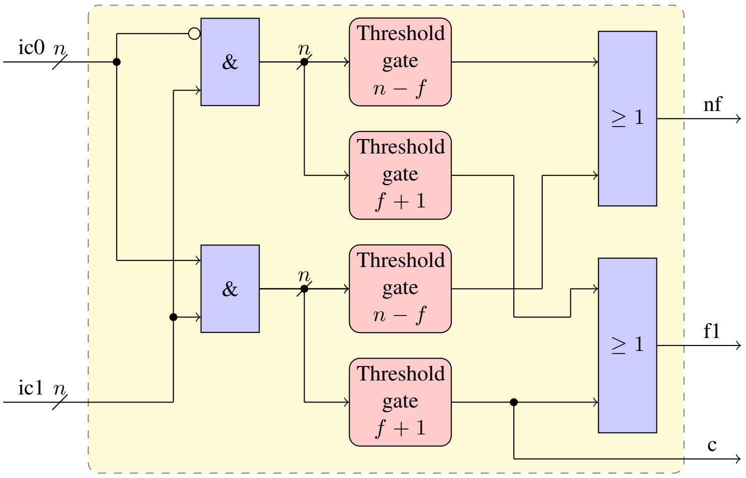


Figure 4.2: Labeling threshold module structure(yellow)

need to be passed before the input results in the correct output (i.e., the maximum path length), is evaluated as indication for the input-to-output delay. Additionally, the maximum number of interconnect intersections, at a certain logic level/depth, is a valuable information for estimating the routing complexity (i.e., it affects the *interconnect distance*), and hence the effect of the interconnect on the input-to-output delay. See Figure 4.3 for an example with an interconnect distance of 4.

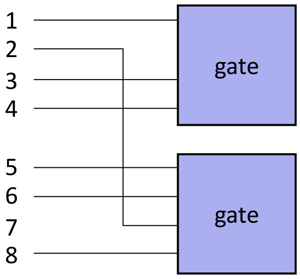


Figure 4.3: An abstract combinatorial logic function with 8 inputs and an interconnect distance of 4.

4.2.1 Sum of Products

The most intuitive way to implement a threshold module, like any boolean function, is to implement it as the sum of all products. Logic designers know such implementations under the term *Disjunctive Normal Form* (DNF). A product refers to the **AND** combination of a single input pattern (vector) that returns 1. The sum is the **OR** combination of all such products.

A threshold module may be reformulated as selecting all k subsets of the n inputs and checking whether one of them has all inputs set to 1. The resulting boolean function contains exactly $\binom{n}{k}$ terms of k not negated inputs. An example for $n = 3$ and $k = 2$ is given in Equation (4.4).

$$thr(\mathbf{x}_4, 1, 3) = (x_0 \wedge x_1 \wedge x_2) \vee (x_0 \wedge x_1 \wedge x_3) \vee (x_0 \wedge x_2 \wedge x_3) \vee (x_1 \wedge x_2 \wedge x_3) \tag{4.4}$$

Since the products share at most $k - 1$ non-negated inputs between each other, the DNF representation can't be minimized anymore. The only way of reducing the amount of gates is by reusing some of the already computed outputs: Depending on the fan-in of the available gates or logic functions, the number of gates (and possibly the overall interconnect complexity) of the implementation of Equation (4.4) could be reduced, if already computed subsets (e.g., such as $x_2 \wedge x_3$) are reused, although this will increase the fan-out and the interconnect/routing complexity at the gate level. For example, Equation (4.5) shows the tree for $n = 4$, $k = 3$ and $m = 2$.

$$thr(\mathbf{x}_4, 1, 3) = \{[(x_0 \wedge x_1) \wedge x_2] \vee [(x_0 \wedge x_1) \wedge x_3]\} \vee \{[(x_0 \wedge x_2) \wedge x_3] \vee [(x_1 \wedge x_2) \wedge x_3]\} \tag{4.5}$$

The advantage of the sum of products implementations is that it can easily be implemented in logic trees. Furthermore, it is glitch-free due to the missing negation paths: Transitions of the same polarity (rising resp. falling) can only cause transitions of the same polarity at the output. Note however, that synthesis tools perform optimizations of their own that may result in non-glitch-free implementations.

Table 4.1 shows the complexity of a direct sum of products implementation without gate output reuse. m represents the AND/OR input width and $AO(n)$ the number of AND/OR gates. As can be seen, the overall complexity increases very drastically ($\mathcal{O}(n^k)$) and is only suitable for very small n and k. Furthermore, the actual complexity depends on the threshold value. Besides, a very drastic limitation is the fan-out since an input is connected to $\binom{n-1}{k-1}$ AND gates to compute the products, which contain this input in the corresponding subset. The interconnect distance is computed by assuming that a single input connection (e.g., of the first input) intersects with all k inputs of all $\binom{n}{k}$ product terms. Note however, that there is probably a tighter and lower worst case bound than this one, but I conjecture that it is probably of the same order of magnitude.

Complexity $AO(n)$	$\left\lceil \frac{\binom{n}{k}-1}{m-1} \right\rceil + \binom{n}{k} \left\lceil \frac{k-1}{m-1} \right\rceil$
Depth $AOS(n)$	$\left\lceil log_m \left(\binom{n}{k}\right) \right\rceil + \lceil log_m (k) \rceil$
Fan-out	$\binom{n-1}{k-1}$
Interconnect distance	$k\binom{n}{k}$

Table 4.1: Sum of Products complexity, $n = 2^p$

4.2.2 Counter and Comparator

This implementation of a threshold module produces an output which is the dense binary representation of the number of 1s in the input vector, by performing binary addition. Such a counter is called $(n, \lfloor ld(n) \rfloor + 1)$-counter [85], since it outputs the count of 1s of width $\lfloor ld(n) \rfloor + 1$ in the n-bit wide input. Obviously, the result can be compared to another binary number, i.e., a threshold X, by a comparator. A counter and comparator implementation is very easy to describe and still relative hardware-efficient. The variant shown in Figure 4.4 uses a linear implementation of an adder structure. Other implementations arrange these counters in trees, which are able to reduce the maximum depth [85].

The major disadvantage is that these circuits are not glitch-free, which makes them rather impractical for asynchronous fault tolerant circuits. Multi-threshold modules may be described easily by just adding comparators for each threshold. Furthermore, there exist less hardware-efficient counter/comparator implementations, which reduce the maximum delay and power requirement of the circuit [31, 60, 85]. A typical improvement are carry-lookahead implementations. Note that a lot of FPGAs provide fast and efficient built-in adder modules, which add m-bit binary numbers. However, they should be used only in multi-cycle/pipelined implementations as they are rare and get easily used up by a larger counter implementations in combinatorial logic, especially if only small binary numbers are added.

Table 4.2 shows the counter complexity of a linear $(2^p, p + 1)$-counter. The complexity represents the amount of full adders (single bit adder with carry-in and carry-out) needed to implement this counter. The interconnect distance and fan-out to successive adders and the fan-out of the input connected to the adders is constant.

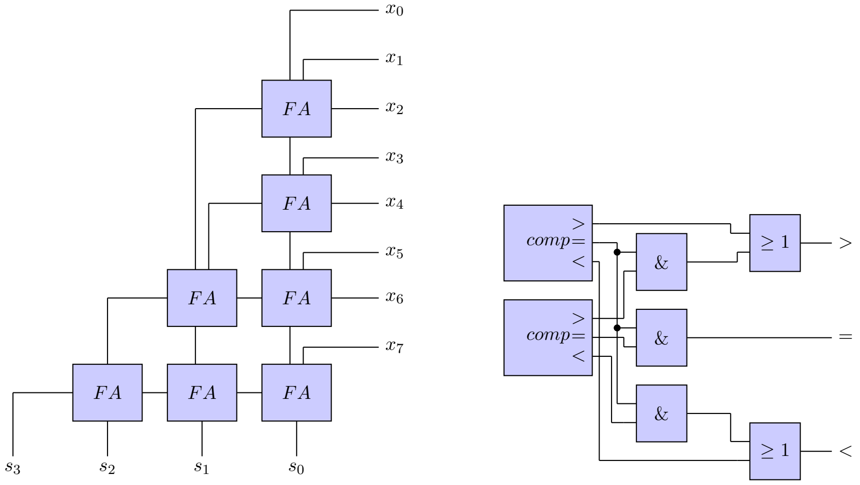
Complexity $FA(2^p)$	$2^p - 1$
Depth $FS(2^p)$	$2^{p-1} + p - 1$
Fan-out	1
Interconnect distance	1

Table 4.2: Counter complexity, $n = 2^p$.

Obviously, the number of inputs into the comparator for a $(2^p, p + 1)$-counter is $2(p + 1)$, when the output of the above counter is compared to some binary number of equal width (i.e., $p + 1$). In order to implement comparators for large p efficiently, comparator trees as shown in Figure 4.5 are usually employed. At the base of the comparator tree single bits (e.g., the MSB of the counter output and the MSB of the threshold) are compared against each other with the comparator defined by Equation (4.6), called the base comparator. At higher levels, the computed outputs are merged using the circuit shown in Figure 4.4b, called a merging comparator.

$$comp(a_i, b_i) = (a_i \wedge \overline{b_i}, (a_i \wedge b_i) \vee (\overline{a_i} \wedge \overline{b_i}), \overline{a_i} \wedge b_i) \tag{4.6}$$

Table 4.3 shows the complexity for a comparator tree. $C(n)$ represents the number of comparators and comparator merging stages needed; The depth $S(n)$ is logarithmic and the interconnect distance resp. the fan-out inside the comparator is constant. Since the input bits, which are compared by the base comparator, are not at the appropriate input position initially, they must be routed to these position resulting in a interconnect complexity of $p + 1$. For example, the initial input order is $a_3, a_2, a_1, a_0, b_3, b_2, b_1, b_0$ typically, which is not identical to the order shown in Figure 4.5. Typical LUTs have 3 or 4 inputs and thus a full adder can be implemented



(a) Serial $(8, 4)$-counter.

(b) Merging two comparators (MSBs, LSBs)

Figure 4.4: A typical implementation of a (8,4)-counter and a comparator tree. [59, 60]

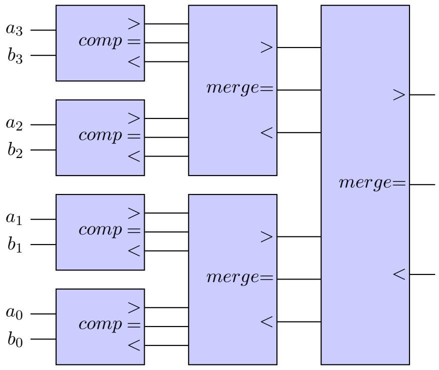


Figure 4.5: Example of a comparator tree implementation using the base comparator and the merging comparator tree to compare two numbers of width 2.

with 2 LUTs and a (merging) comparator with 3 LUTs. If LUTs or gates with larger fan-in are available, the tree/counter depth and LUT complexity may be further reduced. Obviously, synthesis tools may perform optimizations, especially if one of the values (e.g., the threshold) is a constant.

Complexity $C(n)$	$2p + 1$
Depth $S(n)$	$\lceil ld(p + 1)\rceil + 1$
Fan-out	1
Interconnect distance	$p + 1$

Table 4.3: Comparator complexity, $n = 2^p$

4.2.3 Sorting Networks

A sorting network sorts an input sequence based on a defined relation $\leq$ on the input value domain. In my case, this relation represents the standard relation $\leq$ for binary numbers, represented as vectors: The sorting function may be formulated by means of a permutation of a sequence of inputs:

$$sort(\mathbf{x}) = \pi(\mathbf{x}) = \mathbf{y} | y_{i-1} \leq y_i, \quad \text{for } 2 \leq i \leq n \tag{4.7}$$

Herein $y_i, 1 \leq i \leq n$ represents the i^{th} component in the output vector $\mathbf{y}$ and the permutation must be chosen s.t. that the output represents a sorted sequence.

Thus the threshold module for binary inputs is:

$$thr(\mathbf{x}, 1, k) = sort(\mathbf{x})_{n-k+1} \tag{4.8}$$

Therefore, the $n - k + 1^{\text{th}}$ output represents the output of a threshold module with threshold k in the binary domain [14]. The sorting network produces all possible thresholds at once. The inverse threshold, i.e., of the amount of zeros in the input, is just the logical complement of the k^{th} output.

A sorting network is built out of compare-and-swap elements for two values, which are connected according to some chosen sorting algorithm. The actual implementation of the basic comparator element (see Figure 4.6c) for a two 1-bit binary numbers with the included swap is:

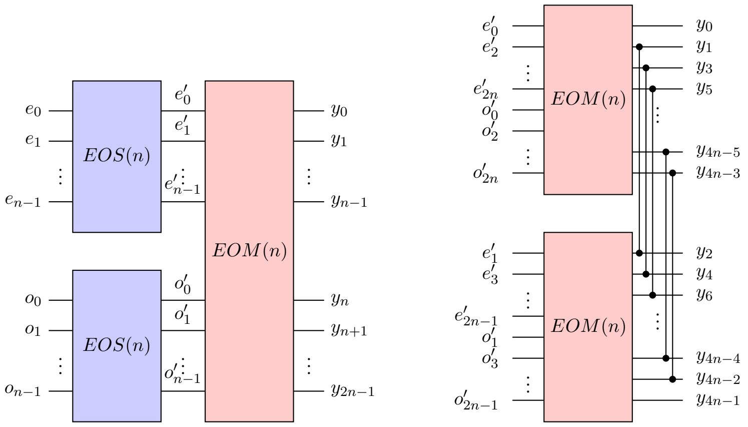
$$comp(e, o) = (e \wedge o, e \vee o) \tag{4.9}$$

In sorting networks designed for CPU implementations, different compare-and-swap operations may be executed for different inputs, i.e., in some cases more or less operations may be performed. These algorithms are more complicated and less efficient to implement in the VLSI context, where all compare-and-swap elements need to be present in hardware anyway and only the worst case performance is of interest. Therefore, the compare-and-swap elements used in a VLSI sorting network or steps performed by the algorithm should be input-independent, which also facilitates glitch-free implementations.

Typical algorithms in the VLSI context are the even-odd merge and bitonic sort algorithms with efficient implementations for input sizes of $n = 2^p$ [59].

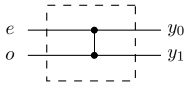
Figure 4.6 depicts the recursive definition of an even-odd merge sort algorithm. An even-odd merge sort takes two sorted sequences as inputs and merges them using the even-odd merger network definition to produce a sorted output sequence. As the figure shows, its recursive definition is rather complex as lot of index (rewiring of comparator) outputs occur in in order to merge those.

From the comparator implementation it can be seen that each output of the sorting network is glitch-free due to missing negation paths and rising resp. falling transition at a comparator input may cause only rising resp. falling transitions at its output, except if opposite transitions occur such that the threshold is crossed. On the one hand, each input and intermediate signal in the network has a fan-out of 1 on the other hand the interconnect complexity increases, i.e., the distance between compared elements grows exponentially with the depth, for larger problem



(a) Recursive definition of the $EOS(2n)$ sorter.

(b) Recursive definition of the $EOM(2n)$ merger.



(c) Basic $EOS(2) = EOM(1)$ compare-and-swap element.

Figure 4.6: Even-odd merge sort. [59]

sizes (see Figure 4.7). Table 4.4 contains the complexity results of the even-odd merge sorting network.

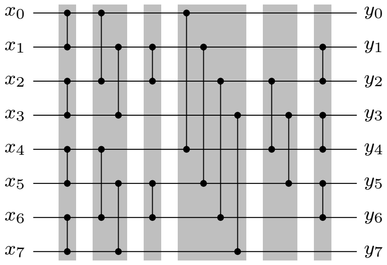


Figure 4.7: A 8 input even-odd merging sorting network. [59]

A popular alternative is a bitonic sorting network, which merges a ascending sorted sequence and a descending sorted sequence. It consists of bitonic sorters (see [59]), the recursive definition of which is simpler compared to the even-odd merge sorters. Figure 4.8 shows an example. The bitonic sorting network has an interconnect distance that grows even more with each stage (at its maximum 2^p), because the order of the sorted lower half of the input at every recursion level of needs to be reversed to make it bitonic (i.e, a combined ascending and descending sorted sequence), which results in a compare-and-swap of the first and last element at the last recursion level (see Figure 4.8). If the lower half of the input is already a bitonic sequence, the interconnect distance is the same as for the even-odd merge sort. A descending sorted sequence may be achieved by swapping the comparator outputs ($\leq \rightarrow \geq$). The major advantage of the bitonic sorter is that its depth (i.e., the number of compare-and-swap elements) is equal on all paths, which results in reduced jitter and glitches at the price of a (small) increase of the number of compare-and-swap elements.

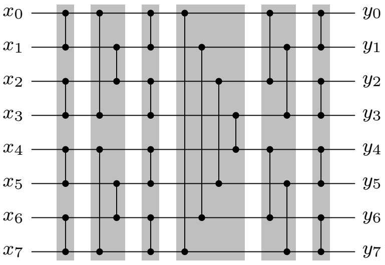


Figure 4.8: A 8 input bitonic sorting network. [59]

Complexity $C(2^p)$	$(p^2 - p + 4)2^{p-2} - 1$
Depth $S(2^p)$	$\frac{p(p+1)}{2}$
Fan-out	1
Interconnect distance (max)	2^{p-1}

Table 4.4: Sorting Network complexity (EOM) for $n = 2^p$. [59]

Table 4.4 shows the complexity of the even-odd sorting network. $C(n)$ represents the compare-and-swap element count of the full sorting network. The actual number of LUTs depends on the placement and routing, it can be estimated by $2C(n)$. Note that all sorting network outputs, which are not used, can and will be removed by the synthesis tool.

4.2.4 Further options

Efficient implementations of small threshold modules may be used to construct larger threshold modules by combining their outputs in a tree. For example, a threshold module with $k = 3$ and 4 inputs, the inputs may be divided into groups of 2 and threshold modules for $k = 1, 2$ may be applied to these groups and recombined. Such an approach has been considered in [8] for constructing threshold modules for neural networks.

Obviously, all the above threshold module implementations could be pipelined to shorten the critical paths between the stages by just inserting flip-flops where needed. However, the labeling solution would not benefit from pipelined threshold modules, since Algorithm 1 can not be pipelined, because of the incorporated consensus algorithm.

In order to reduce the hardware effort, the sorting network and counter implementation could be implemented in a multi-cycle scheme, i.e., with a synchronous FSM. Using multiple cycles per operation would slow down the execution speed multiplicatively, however.

Other options are analog threshold modules, which work by summing currents or voltages. A transistor with a specified threshold or analog comparator will output logic high if the voltage/current sum reaches the required threshold (see [14]).

4.2.5 Chosen implementation

Since the prototype implements a system of only $n = 8$ nodes, I chose the counter and comparator implementation as an appropriate solution, due to its genericity and flexibility. Glitches can be neglected due to the synchronous environment. Listing 4.1 shows an excerpt of the threshold module VHDL description at the RTL level, which is automatically transformed by the synthesis tool to LUTs, using an adder and comparator style threshold module. At the end, the outputs are recombined for the inputs 10 and 11 to form the threshold module output $(\mathbf{nf}, \mathbf{f1}, \mathbf{c})$ according to Equation (4.2.

```
...
inc_0  := unsigned(not input_chan0 and input_chan1);
cnt_0  := 0;

for i in inc_0'range loop
  cnt_0  := cnt_0+to_integer(inc_0(i downto i));
end loop;

nf0  <= cnt_0 >= (N-F);
f10  <= cnt_0 >= (F+1);
...

nf   <= nf0 or nf1;
f1   <= f10 or f11;

c    <= '1' when f11 else '0';
```

Listing 4.1: Excerpt of the VHDL threshold module description at the RTL level.

4.3 Lock-step synchronous round communication

The actual communication in a lock-step round system proceeds like in any synchronous system: In some round/clock cycle r, the FSM at a node selects a new state and a broadcast value, using its current local state and the received values; The broadcast value is present at the transmission buffer output at the beginning of the next round. In this round $r + 1$, the value is received by all recipients and processed in the very same way.

The remaining problem is that only imprecise (bounded-precision) BSCLKs and bounded delay links are available at the nodes. In Chapter 2.1.5 and 2.7, I described how lock-step synchronous rounds can be simulated in such a system. I will now streamline this description to the available bounded-precision imprecise BSCLKs and state-based bounded-delay links.

A lock-step synchronous execution performs a computation every round at a node, which is triggered by the BSCLK, using either the falling or rising transition of the nodes clock signal. Assume that the clock frequency does not violate any local timing constraint (i.e., those of local computations using synchronous (state) elements). Furthermore, assume that buffering an input port state (i.e., sampling) is performed by a node when it starts a new round. Essentially, if some node broadcasts a new state in round r at time t, it will be received at latest after d^l seconds on a bounded delay link with the upper bound d^l, i.e., at latest at $t + d^l$. Due to both the BSCLK imprecision and the node's clock distribution network, which is represented by the skew Σ^l, nodes do not start a local round at the same time. Made specifically, the latest state put on the link by some node in round r is seen at the input port of all nodes at latest $\Sigma^l + d^l$ after the first node started round r. Hence, the minimum lock-step synchronous round period P_{min} must satisfy $P_{\text{min}} \geq \Sigma^b + d^l$, i.e., the interval between sending new states must be at least that long.

Assume that minimum period between local computations/state buffering is T^l_{min}, based on the BSCLK's LSB as clock signal. Then, setting $P_{\text{min}} = kT^l_{\text{min}}$, where k is the minimum number of synchronous rounds between two successive broadcasts, results in the following inequality for k:

$$k \geq \left\lceil (\Sigma^l + d^l)/T^l_{\text{min}} \right\rceil \tag{4.10}$$

Note that all nodes execute the same algorithm and broadcast a new state simultaneously in the same round (r), i.e., all BSCLKs have the same value in that particular round. Therefore, every node knows it may buffer/use the new received state safely at the time it starts round $r + k$. Hence, inequality (4.10) ensures the setup requirement of a labeling node, but not the hold requirement. Any hold requirement can only be satisfied if an input port state remains stable (i.e., holds) when a node buffers this state at the time it starts round $r+k$. Thus, any node, which sent some new state in round r, must guarantee that any state sent in some later round $r + k + l, \quad l \geq 0$ is not received at the input port of some node, before this node started round $r + k$. This results in the following inequality for the parameter $l \geq 0$:

$$l \geq \left\lceil (\Sigma^l - d^l_{\text{min}})/T^l_{\text{min}} \right\rceil \tag{4.11}$$

If the minimum delay d^l_{min} of the bounded delay links among the nodes is unknown, it can be assumed to be 0. On the downside, these requirements decrease the transmission rate and the available number of (lock-step) synchronous rounds for communication, since a node may broadcast a new state every $k + l$ rounds at most.

Figure 4.9 shows the Trellis timing diagram of the described problem. The solid vertical line represents the first start of a new round and the first dotted vertical line (with short strokes) the last possible start (skew). The last dotted line (with long strokes) marks the maximum delay of a link and thus the point when the last state may be received.

The first Trellis diagram shows the standard lock-step round model for two synchronized nodes with some skew. In the second round, node p_1 has received a new state from node p_2 before it sent its own state, i.e., started the next round, which results in a hold violation. Besides, the setup time is violated in round r_3 too. Note that this is not a problem in the abstract model,

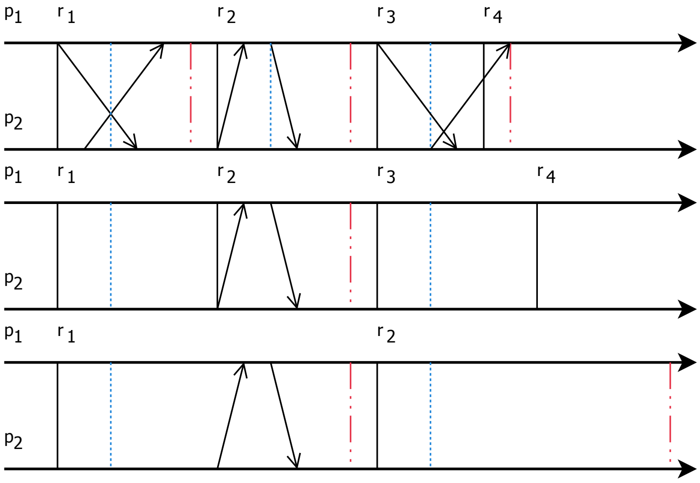


Figure 4.9: Synchronous computation model timing violations and new abstraction.

i.e., zero skew, or when using event-based communication, since all states/messages are buffered upon reception or sent at all nodes concurrently.

This problem can be solved by using “macro”-rounds that consist of two rounds each: In even rounds, processing/buffering takes place, in odd rounds, values are sent and received. The second Trellis diagram in Figure 4.9 shows the resulting execution; all setup/hold violations have been resolved. The last one displays the corresponding “abstract” lock-step synchronous round execution.

4.3.1 Chosen implementation

In Chapter 2, I provided an overview of typical synchronous and asynchronous communication schemes. Since the algorithm is executed within a synchronous environment and low hardware effort is desired, the actual choice to use flip-flops as the communication buffers is inherent: DFFs need no extra handshake, control modules or further signals, except the clock. Thus, both the transmitter and receiver communication subsystem contain DFFs only. The transmitter register output is connected to the output port of a node, which is connected to all input ports and hence to the receiver register of the receiver communication subsystem of all nodes, including itself. In this prototype, a node needs $n = 8$ such receiver registers. Their outputs are connected to the threshold module directly.

Since the BSCLK’s LSB clock period of the current prototype is large (see Chapter 5) and the (assumed) delays and skew are small enough, I can choose $k = 1$ and $l = 1$. This holds even if both rising and falling transitions of the BSCLK are used to drive local rounds (i.e., computations and buffering); Using both transitions increases the available number of synchronous rounds before the BSCLK overflows by a factor of 2. Note that this also increases the clock frequency at which the labeling nodes operate, i.e., T^l_{min} is now half the BSCLK’s LSB clock period, because a synchronous round is triggered on every transition. The FSM and transmitter subsystem has been chosen to operate at each falling transition and the receiver subsystem samples the input ports at each rising transition to maintain the constraints of $k = 1$ and $l = 1$. Consequently, one synchronous round is used to perform the computations and send a state, while the other round is used to buffer the input states. This implies that the actual number of (lock-step) synchronous rounds, between overflows of the BSCLK, is $M/2$ at the algorithm level. Furthermore, this solution does not require any additional hardware or further states in the FSM to count the number of passed rounds to ensure these constraints, since the DFFs need to trigger at the resp. other clock signal transition only.

Although this approach features fast and metastability-free serial synchronous communication, faulty nodes may still violate the setup and hold requirements. Therefore, instead of just one DFF acting as receiver, a 2 stage synchronizer is used to mitigate the effects of metastable upsets. Figure 4.10 shows the entire communication subsystem between two nodes, using the two stage synchronizer as receiver register and a single DFF as transmitter register. The clock signals are connected to the BSCLK's LSB of the corresponding node. Unfortunately, the synchronizer affects the runtime of Algorithm 3 multiplicatively, because the synchronizer pipeline delays the reception of a broadcast by one additional round and the Phase King algorithm is only able to broadcast a new state once all previous broadcasts have been received, which leaves the synchronizer pipeline empty.

This results in the following round complexity of the labeling algorithm in my setting:

$$r(l, f, sync, i) = 2^i(2l + 2sync + (3sync + 2)(f + 1) + 4) \tag{4.12}$$

In the above equation, l is the label width, f the number of Byzantine faulty nodes that must be tolerated, $sync$ is the length of the synchronizer pipeline, and i is the clock division factor; $i = 1$ represents the BSCLK's LSB. Thus, the size M of the BSCLK must satisfy $M \geq r(l, f, sync, i)$ and the last transition before the BSCLK overflow must be a falling transition of the chosen clock bit.

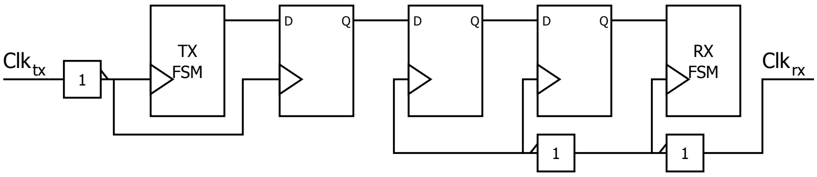


Figure 4.10: Transmit and receive buffer implementation.

Figure 4.11 provides a detailed timing diagram of this communication subsystem, c.p. also Figure 4.9. As can be seen in this example, the overall frequency has increased by a factor of 2, since both clock transitions are used by the nodes to trigger synchronous rounds for computation and buffering. Therefore, T^l_{min} must be set to half the clock period.

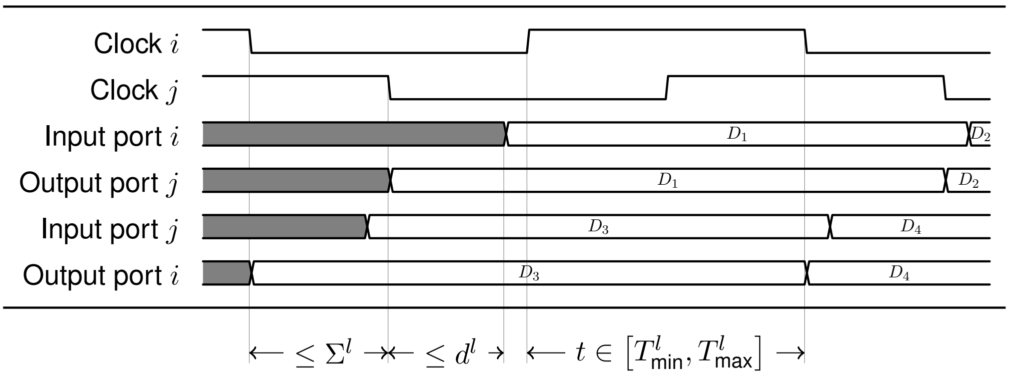

Clock i
Clock j
Input port i
Output port j
Input port j
Output port i
D_1
D_2
D_1
D_2
D_3
D_4
D_3
D_4
$\leftarrow \leq \Sigma^l \rightarrow$
$\leftarrow \leq d^l \rightarrow$
$\leftarrow t \in [T^l_{\text{min}}, T^l_{\text{max}}] \rightarrow$

Figure 4.11: This figure depicts an example of two correct nodes' clock signals and the communication solution timing. Both nodes sample the input using the rising clock transitions and broadcast on falling clock transitions.

4.4 Synchronous FSM

The FSM implements Algorithms 1-3 and is implemented according to the standard current state and next state logic synchronous FSM design (see Figure 2.20), without pipelining except for Algorithm 2.

The choice of state encoding is left to synthesis tool and may be forced if needed. I implemented the state-machine using macro states and synchronous state elements (flip-flops) to encode variables of the above algorithms. The synthesis tool chooses a one-hot encoding (i.e., one bit for each macro state) in the default setting if the number of states is rather small (< 50). As mentioned, the FSM execution, receiving and broadcasting implemented by the communication subsystem utilizes the BSCLK's LSB as clock signal. For execution control, it uses the falling transition of its clock signal.

States

First, it is important to identify the implicit state of Algorithms 1, 2 and 3. Apparently, these algorithms must maintain the following implicit state variables:

- Label L_i.
- Label candidate c_i.
- Threshold module mask bits i_{c_1}. Depending on the particular threshold module, they which are either all set, set accordingly to S_i in Algorithm 2, or represent the high-order bit of the only dual-bit broadcast in Algorithm 3.
- Trust value b_i, represented by the state variable $lock$ in the FSM.
- Broadcast register tx_i.

Besides these state variables, the FSM needs to know which broadcast it must perform resp. which of the above state variables must be modified by the next state logic, i.e., it needs some states to implement the control flow. Obviously, these explicit states are implementation-dependent. In order to maintain scalability and to automatically adapt the implementation to the (main) generic parameters n, f and l, the FSM is implemented using the macro states described in Chapter 4.4.1 and the following auxiliary micro states:

- Bit and synchronizer counter cnt_{bit}.
- Phase King phase/king counter cnt_f.
- Phase King $listen$ bit, tells a node whether it must listen to the Phase King broadcast or not.
- Overflow bit ovf to indicate a BSCLK overflow.

4.4.1 Detailed description

Figure 4.12 and 4.13a depicts the current FSM implementation of the labeling algorithm, which is very similar to actual the `VHDL` description. Each macro state implements a specific state of the labeling algorithm. As in the already used FSM specifications in Chapter 2, the transition edges of the graph specify the performed actions (shaded boxes) as well as the transition guards (labels). The current macro-state is represented by labeled circles as usual. The transitions and associated actions are performed when a new synchronous rounds starts, i.e., upon a falling clock transition.

The actions specify the modifications to the above state variables, i.e., if no modification is specified for some state variable, it keeps the current value.

The task of each macro state from Algorithm 1 can be specified informally as follows:

RED_INIT: Initializes the label candidate to the current value of the label.

RED_1_0: Broadcasts the candidate bits 1 to $sync$ sequentially. Waits until the first bit is present at the synchronizer output. First for-loop in Algorithm 2.

RED_1_1: Broadcasts the candidate bits $sync + 1$ to l sequentially and computes the threshold of the received broadcasts. First for-loop in Algorithm 2.

RED_1_FIN: Intermediate broadcast between the first and second for-loop in Algorithm 2.

RED_2_0: Broadcasts the candidate bits 1 to $sync$ sequentially. Waits until the first bit is present at the synchronizer output. Second for-loop in Algorithm 2. Also stores the set S_i, i.e., sets i_{c_1}.

RED_2_1: Broadcasts the candidate bits $sync + 1$ to l sequentially and computes the threshold of the received broadcasts. Second for-loop in Algorithm 2.

PHK_1_0: First broadcast in each phase of Algorithm 3. Waits until the broadcast is received. If there is no further phase it updates the label candidate c_i accordingly.

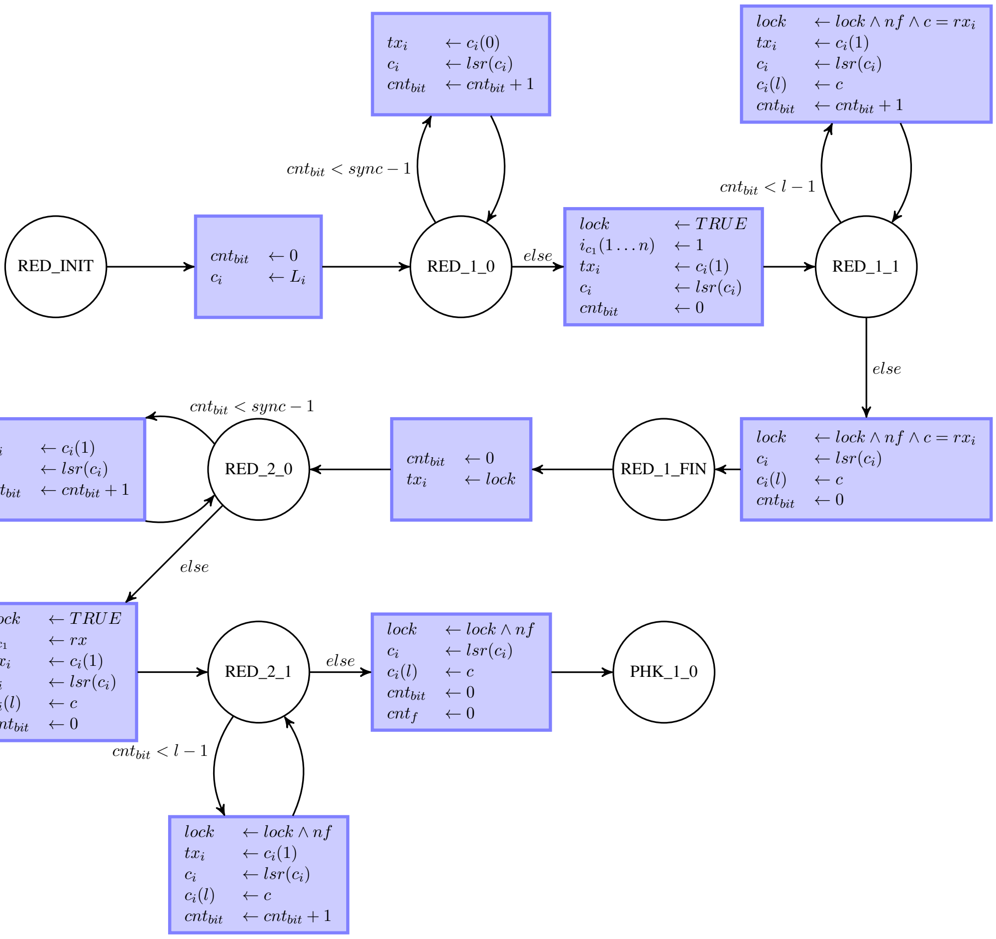


Figure 4.12: RED part of the transformed algorithm into synchronous FSM

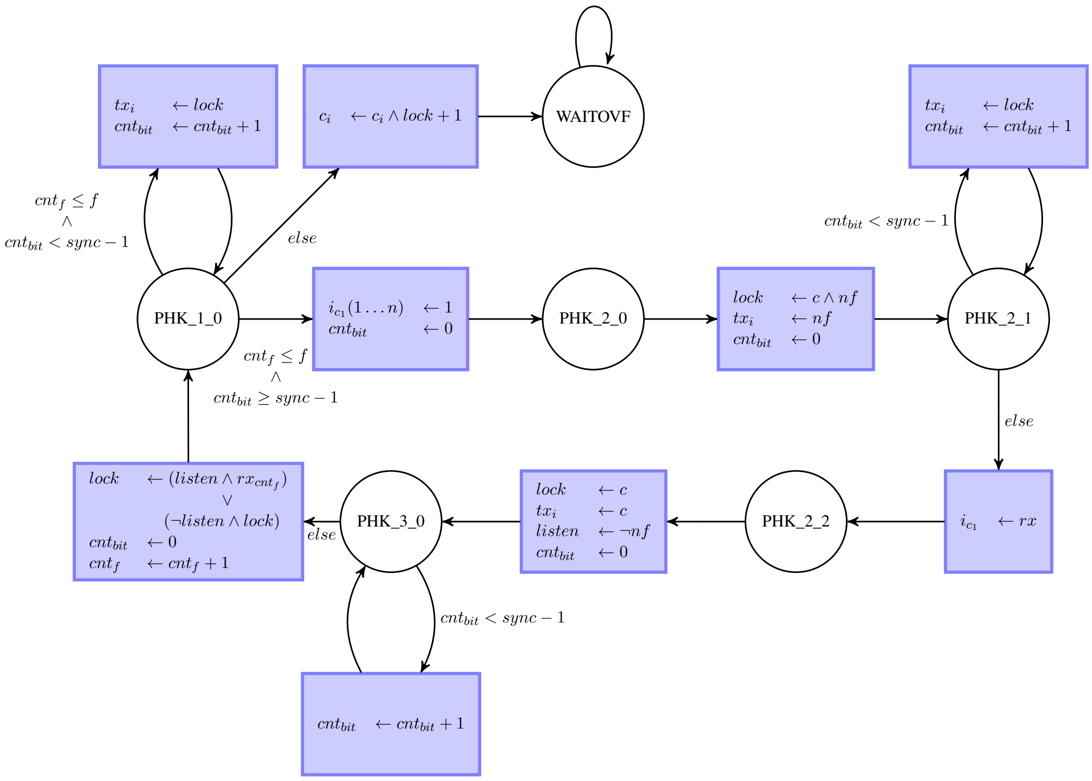
PHK_2_0: Broadcasts the MSB of the second (dual bit) broadcast in each phase of Algorithm 3.

PHK_2_1: Broadcasts the LSB of the second (dual bit) broadcast in each phase of Algorithm 3. Waits until the LSB is received.

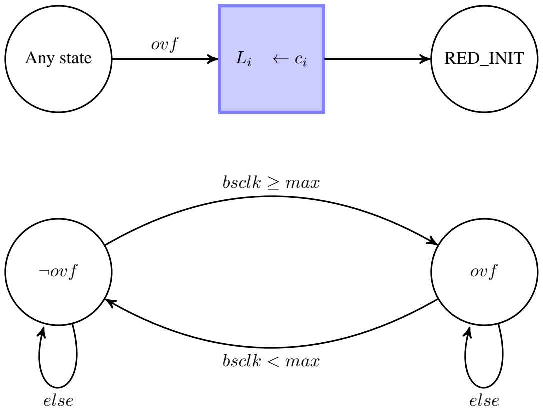
PHK_2_2: Computes the second threshold to determine whether there is a candidate or the node listens to the Phase King broadcast instead. Broadcasts the current local candidate (Phase King).

PHK_3_0: Waits until the Phase King broadcast arrives and listens to the broadcast if $listen$ is set.

In order to reduce the impact of the synchronizers on the runtime of Algorithm 2, processing of the label bits is pipelined, i.e., in each cycle, a new bit of the candidate label is broadcast


$tx_i \leftarrow lock$
$cnt_{bit} \leftarrow cnt_{bit} + 1$
$c_i \leftarrow c_i \wedge lock + 1$
WAITOVF
$tx_i \leftarrow lock$
$cnt_{bit} \leftarrow cnt_{bit} + 1$
$cnt_f \leq f \wedge cnt_{bit} < sync - 1$
$else$
$cnt_{bit} < sync - 1$
PHK_1_0
$i_{c_1}(1 \ldots n) \leftarrow 1$
$cnt_{bit} \leftarrow 0$
PHK_2_0
$lock \leftarrow c \wedge nf$
$tx_i \leftarrow nf$
$cnt_{bit} \leftarrow 0$
PHK_2_1
$cnt_f \leq f \wedge cnt_{bit} \geq sync - 1$
$else$
$lock \leftarrow (listen \wedge rx_{cnt_f}) \vee (\neg listen \wedge lock)$
$cnt_{bit} \leftarrow 0$
$cnt_f \leftarrow cnt_f + 1$
PHK_3_0
$else$
$lock \leftarrow c$
$tx_i \leftarrow c$
$listen \leftarrow \neg nf$
$cnt_{bit} \leftarrow 0$
PHK_2_2
$i_{c_1} \leftarrow rx$
$cnt_{bit} < sync - 1$
$cnt_{bit} \leftarrow cnt_{bit} + 1$

(a) Phase king part of the transformed algorithm into synchronous FSM.



(b) Stabilization mechanism of the macro state machine and the label update.

Figure 4.13: Phase King consensus FSM and the BSCLK overflow mechanism for self-stabilization.

until there is none left. Hence, after waiting for the initial bit to be present at the synchronizer output port, the next bit is already available in the next round. The need to wait for the first bit at the output can been identified by the transition guards $cnt_{bit} \leq sync - 1$ in the FSM figures. Unfortunately, this does not work for Algorithm 3, because the nodes must vote on the current received broadcasts before starting the next broadcast. This explains the multiplicative effect of the synchronizer pipeline on its runtime.

Note that the input channel 0 (i_{c_0}) of the threshold module is connected to the synchronizer pipeline output directly. Furthermore, I saved the LUTs of the candidate bit multiplexer ($c_i(j)$) in Algorithm 2 by shifting the candidate label register ($lsr(c_i)$) every time a new broadcast is performed instead. This allows to insert resp. retrieve the next broadcast value from static locations at the top $c_i(l)$ resp. at the bottom $c_i(1)$. Hence, each bit is at its specified location after all bits are received.

4.4.2 Self-stabilization

During stabilization or after recovering from transient faults, it may occur that a node is not in the macro-state corresponding to the BSCLK value (e.g., in the initial state if the clock is 0). Since the algorithm needs to be in the initial state after each overflow of the BSCLK, it is a logical choice to use its wrap-around to reset the FSM into the initial state, no matter what the current state is. Besides, the generated label must be updated at the overflow of the BSCLK with the current candidate label too.

For this purpose, the overflow flag ovf is used. If it is set, the node will update its label and set its state to the initial state `RED_INIT`. Note that this is an additional transition from any of the FSM states to the initial state (see Figure 4.13b), which occurs at the falling transition of the FSM clock. Obviously, this flag must not violate the FSM timing constraints, which makes a solution based on an asynchronous reset rather impractical. Therefore, a comparator compares the higher order bits of the BSCLK against $M/2 - 1$, which is sampled by the ovf DFF at each rising clock transition instead.

The attentive reader should notice that this is the opposite clock transition of the one used by the FSM. This approach is similar to the one used by the receiver subsystem (i.e., synchronizer pipeline). Since the higher order bits of the BSCLK can only change upon some Accept^{+}-pulse, which creates a falling clock transition on the BSCLK's LSB (i.e., the clock signal), the overflow flag samples the comparator output only when it is stable and safe, i.e., cannot change. Figure 4.14 shows the synchronous abstraction of the overflow flag and the different clocks. The comparator compares the BSCLK high order bits against `7F` in that example, which corresponds to $M = 256$.

To sum up, the progression of the state machine depends on its local state only, which implies that only an incorrect behaviour of the BSCLK, during stabilization, can disturb the control flow of the FSM. Besides, the chance of metastable upsets due to faulty inputs is sufficiently reduced by the employed synchronizer. The self-stabilization mechanism ensures that any corrupted state of the FSM will be removed at the next correct BSCLK overflow to ensure a consistent and correct algorithm execution.

Finally, the overall implementation of a node is shown in Figure 4.15.

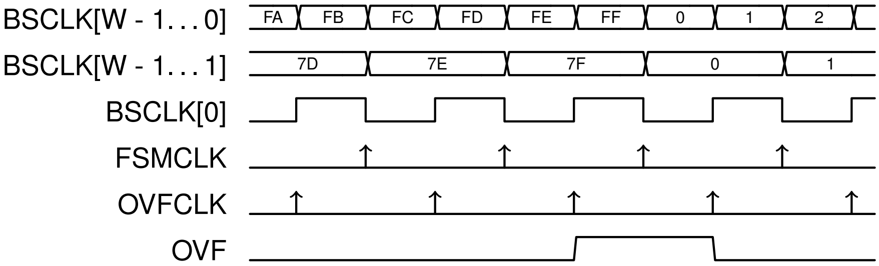


Figure 4.14: Overflow flag and FSM abstract timing. W represents the BSCLK width.

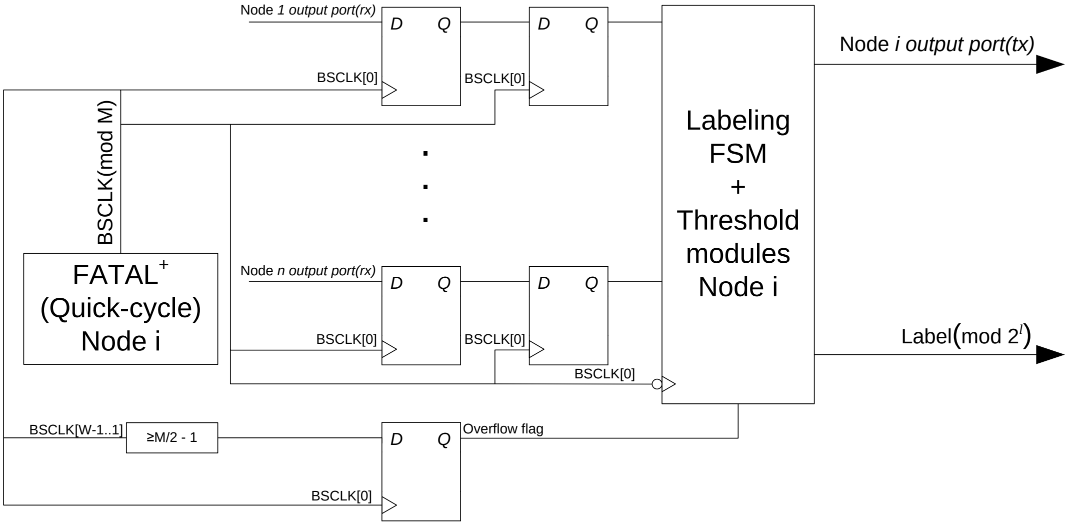


Figure 4.15: System overview of the labeling implementation and its interfaces at a node i.

CHAPTER 5

Experiments

This chapter will cover the experiments performed on the implementation and its results. It will also provide an overview of the standalone and integrated testbench structure and the Byzantine node implementation. The purpose of the experiments is to verify the circuits and to show that it is indeed possible to implement a synchronous algorithm, like the discussed labeling algorithm, based on the FATAL$^+$ prototype.

5.1 Hardware and Software

Hardware

The well known *DE2-115 Development and Education Board* from Altera, provided by Terasic, has been chosen as the target platform. It inhabits a Cyclone IV FPGA, which may be programmed using the built-in USB-Blaster. Furthermore, it provides 18 switches and a 40-pin expansion header. Since the labeling algorithms prototype needs to be integrated into the existing FALTAL$^+$ prototype [66] I used the same mechanisms and interfaces to build my own testbenches. It turned out that they were sufficient for all envisioned experiment, which relieved me from developing a new test interface for the entire system.

The USB-blaster was used for downloading the FPGA configuration file and to gather information from the in-circuit logic analyzer *Signal Tap II*, while developing and testing the algorithms and experiment setup manually. The expansion header was used for the main testing interface between the *bigAVR6* μC -board from Mikroelectronica, containing an *ATmega 1280* microcontroller, the FPGA-board, and the Agilent logic analyzer *16800*. As shown in Figure 5.1, the μC-based was connected via a serial interface to a PC acting as the experiment controller, and to the FPGA-board using an SPI interface (see Chapter 2.5.4.2). The μC-board and the FPGA-board operate at different voltages (5 V and 3.3 V), which required a logic level shifter based on the 8-bit bi-directional *TXB0108PWR* shifter from Texas Instruments. The PC was operated under Linux.

Software

The FPGA configuration file was synthesized by the design and synthesis tool Altera *Quartus II*: Additionally, it has been used to perform a simple timing analysis of the labeling algorithm, i.e., that the internal state machine implementation of the labeling algorithm works at the given frequencies. The host program (PC) for experimental control was implemented in *Python*, which uses the RS-232 connection to communicate with μC-board. The software for the μC-implementation has been provided as C-source code, compiled using the *avr-gcc* and deployed with *avr-prog2*. Note that it also incorporates hardware drivers for UART, SPI and timers.

5.2 Testbenches

The testbenches consist of the following compounds:

Experiment controller (PC): The PC application is implemented in Python and uses the serial interface to communicate with the μC. The application is used to generate new random system reset states, which starts a new experiment run, and to receive and log the experiment results (i.e., the required stabilization time). Once the result has been logged, the PC chooses a new random reset state.

Microcontroller (μC): The μC receives the experiment system reset states and a desired reset procedure to start the experiment. Prior to applying the reset procedure the received reset state will be shifted into the FPGA. The reset is used to load the new system state. After that the μC waits an adjustable amount of time for the *correct*-signal, provided by the FPGA side of the experiment, to become true, which indicates that the labeling algorithm has stabilized, and observes it for another adjustable amount of time to ensure that it remains true. The recorded stabilization time is sent to the PC and the μC waits for a reset state sent by the PC.

FPGA: The FPGA incorporates the prototype implementation of the labeling algorithm and its testbench environment. In order to verify its correct operation, a validation node is integrated into the testbench, which samples the labels of the correct nodes at each overflow of the BSCLK using the FATAL-pulses and compares them against the labeling definition. Once the labeling specification holds, it indicates stabilization via the correct-signal. Should the label specification be violated, some time after the correct-signal is set to true, the correct-signal will be set to false again.

This experiment flow is depicted in Figure 5.2 used in the following two testbench settings:

Standalone testbench

Its main purpose is to validate the functional behaviour of the labeling algorithm using simulated synchronized BSCLKs: Instead of the underlying FATAL$^+$ algorithm it provides four counters driven by a single PLL utilizing four of its clock outputs to simulate stabilized BSCLKs. Since there are eight nodes altogether, a pair of nodes will share such a counter. Once these clocks overflow, a pulse is generated to simulate the slow FATAL-pulses (see Chapter 3.1.2).

It provides the following features:

- Simulation of Byzantine faulty labeling nodes to show the Byzantine self-stabilizing feature of the labeling algorithm.
- Start the experiments with downloadable random reset states.
- Support for late joining experiments.
- Detect the correct operation via the validation node.

Integrated testbench

The integrated testbench is essentially the same, but the FATAL$^+$ prototype is used instead of the BSCLK simulation in the standalone testbench. Only the Byzantine test case is performed to measure the end-to-end stabilization time, since the correct behaviour follows from the previous experiments of FATAL$^+$ [66], the standalone testbench and the combined Byzantine testbench.

It provides the following features:

- Integrated Byzantine testbench with the existing FATAL$^+$ prototype.
- Start the experiments with downloadable random reset states for the labeling and FATAL$^+$ prototype.
- Detect the correct operation via the validation node.

Byzantine nodes

First of all it is important to define which nodes should exert Byzantine behaviour. In the case of the labeling algorithm, which is synchronous, only the number of messages containing the same value is important, but not which node sent the information. The only exception is Algorithm 3, where the nodes acting as Phase Kings are statically defined to be in $\{1, \ldots, f+1\}$. statically defined. Therefore the first f nodes will be assigned statically to be Byzantine faulty in those experiments, hence gain the maximum power to delay label synchronization (up to the last phase).

The Byzantine faulty nodes use the same state machine as a correct node, but choose a message that will most likely introduce a considerable inconsistency between correct nodes. More specifically, they try to separate the (correct) nodes into two groups with different decision outcomes, one of which consisting of $n-2f$ (correct) members if possible and needed. In the case of $n = 8$ and $f = 2$ this will split the nodes into one group of 4 correct nodes and another group consisting 2 correct nodes. The decision outcome of Byzantine faulty nodes is not considered.

The Byzantine nodes thus exert very strong adversarial behaviour except, in the case that the Byzantine nodes and the correct nodes start in an inconsistent state of the FSM (destabilized system): Since the Byzantine nodes uses its local state and received messages to choose the worst-case faulty messages, their adversarial power is reduced. I do not consider implementation of Byzantine behaviour that is even strong in inconsistent states.

In the first loop of Algorithm 2, the Byzantine nodes broadcast the opposite value of their threshold modules' output if all correct nodes sent the same value or send every node its own message back to force them to keep their label if at least $n - 2f$ correct nodes send the same label bit. In the last case at least $n - 2f$, but less than $n - f$, correct nodes will tell the others that they want to propose the candidate label ($b_i =$ ***true***) in the second loop, while the remaining correct nodes inform all nodes that they couldn't identify the candidate label.

In the second loop the Byzantine nodes propose the same label as the correct nodes to at most the first f correct nodes proposing the candidate label too. All other correct nodes will receive the opposite label bit. Therefore if less than $n - f$, but at least $n - 2f$ correct nodes proposed a candidate label, f correct nodes have identified a candidate label and exit with $b_i =$ ***true***, while the other correct nodes exit with $b_i =$ ***false***. Hence, if no consensus would be run after the reduction, the Byzantine nodes would be able to delay the stabilization of the labeling algorithm infinitely long, in those cases.

In the first broadcast of algorithm 3 the Byzantine nodes broadcast the value that occurs least often to all correct nodes. Now, all correct nodes broadcast 00. In the second broadcast, the Byzantine nodes broadcast $1b$, where b is the least frequent value again. In the Phase King's broadcast the Byzantine nodes send their inconsistent messages containing ***true*** to the first f correct nodes and ***false*** to the others. Hence, the correct nodes are not able to stabilize before reaching the phase with the correct Phase King if there are not $n-f$ correct nodes with the same initial consensus candidate value already.

Two approaches are available to actually implement this Byzantine behaviour: Either individually within each correct node or as one separate "multi-node" connected to the broadcast channel synchronizers, which emulates the behaviour of all faulty nodes. The first approach has the advantage that it may have more adverse impact when FATAL$^+$ hasn't stabilized or before the first correct BSCLK overflow occurred, but it needs more hardware. The second approach needs more combinatorial logic in the link (communication) path to the correct nodes, which increases the delay. Nonetheless the provided (frequency) accuracy bounds are large enough in the current setting to cover the additional delay. Therefore the Byzantine node has full control about the message sent to each correct node and is able to provide inconsistent information.

I chose to implement the second approach, in which all Byzantine faulty nodes are implemented in a single node. Note that if the correct nodes had accepted any label in first loop of the reduction, instead of accepting only their own, it would have been more complicated to implement the worst case behaviour in all initial settings. For example, assume that all nodes are correct ($n = 4$) and their labels are initialized to $(1110, 1101, 1011, 0111)$. Then all nodes will identify the label 1111 as candidate instead of no correct node identifying a label candidate ($b_i =$ ***false***). In such an implementation the current Byzantine nodes implementation will result in $b_i =$ ***false*** at all correct nodes, although f correct nodes could exit with ***true*** using another behaviour.

Reset procedure

The whole state of the labeling algorithm of one node consists of the following components for $(n = 8, l = 17, f = 2, sync = 2)$:

state: The macro state of the FSM (enumeration with one bit for each state). Size 12-bit.

$\mathbf{cnt_{bit}}$: Size 4-bit.

$\mathbf{cnt_f}$: Size 2-bit.

lock: Size 1-bit.

listen: Size 1-bit.

$\mathbf{c_i}$: Size 17-bit.

$\mathbf{L_i}$: Size 17-bit.

rxchan: The buffer connected to the most MSB (i_{c_1}) of the threshold module. Size 8-bit.

tx: Size 1-bit.

ovf: The overflow bit indicating a BSCLK overflow. Size 1-bit.

rxSync: The synchronizer pipeline used to buffer input ports of all nodes. Its output is connected to LSB (i_{c_0}) of the threshold module too. Size 16-bit.

Almost all state components above are represented by registers clocked by the falling edge of the bounded synchronized clock, except for the last two, which use the rising edge.

A new reset state, randomly chosen by the PC, is transferred to the FPGA by using a *SPI*-shift register, which doesn't need much hardware beside the actual reset-registers for each state-bit. The μC outputs the reset state using its *SPI* hardware (see Chapter 2.5.4.2), which comprises out of one 1-bit data channel and a clock. Each bit of the serial shift register corresponds to a bit of a state within one nodes complete state. Since this is the same state transmission procedure as in [66] I appended the state of the labeling algorithm at the end of the serial shift register in FATAL$^+$.

Additionally, each labeling node has a reset enable bit within the shift register for choosing which labeling node will be affected by a reset. This is necessary, because I don't have enough pins available at the expansion header.

The testbenches provide 3 different types of resets:

SwRstClk: Asynchronous reset of the underlying FATAL$^+$ algorithm states resp. its simulation.

SwRstNode: Asynchronous reset of the labeling algorithm states (for enabled nodes).

SwRstTest: Reset of the validation node (see Chapter 5.3), which is synchronized to the validation node's clock.

Note that as long as the *SwRstClk* is activated, all bounded synchronized clocks will stop as the FATAL$^+$ algorithm resp. its simulation is held at the reset state. In the future, it may be reasonable to use a signal which just freezes the algorithm instead of resetting it. Such a signal must be accurately timed to ensure a correct resume of the operation of all algorithm, however.

The actual two reset procedures performed are:

Experiment 1: $SwRstClk \to SwRstNode, SwRstTest$. In this case, the labeling algorithm, of all nodes is guaranteed to start properly.

Experiment 2: $SwRstNode$. In this case a node may suffer from a metastable upset.

All resets are deactivated in the reverse order. The time between the activation/deactivation of the resets is $5\ ms$, which ensures that all asynchronous and synchronized resets are long enough and deactivated completely before the next reset (e.g., $SwRstClk$) is released.

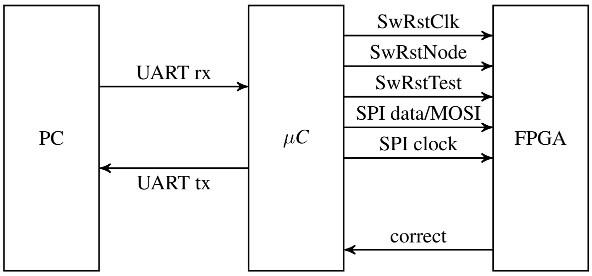


Figure 5.1: Hardware testbench setup.

Timing analysis

The correctness proofs rely on the assumption that certain timing constraints are met that ensure the correct distributed synchronous execution of the algorithm and a meta-stability free operation of the FSM. These constraint of my implementation have been validated by means of standard analysis of synchronous circuits (see Chapter 2.7), as follows:

The timing bounds provided by the existing FATAL$^+$ implementation are:

Accuracy bounds Quick-Cycle: $[T^+_{\text{min}}, T^+_{\text{max}}] = [57.67\,\mu\text{s}, 76.16\,\mu\text{s}]$

Accuracy bounds Main: $[T_{\text{min}}, T_{\text{max}}] = [2.57\,\text{ms}, 13.18\,\text{ms}]$.

Skew/Precision bounds: $\Sigma^+ = 1.6\,\mu\text{s}$ and $\Sigma = 5.12\,\mu\text{s}$.

Delays: $d^+_{\text{max}} = 800\,\text{ns}$, $d^+_{\text{min}} = 0\,\text{ns}$ and $d = 2.56\,\mu\text{s}$.

Clock-Ratio: $\vartheta = 1.3$ and thus clock drift is $\rho = 0.3$.

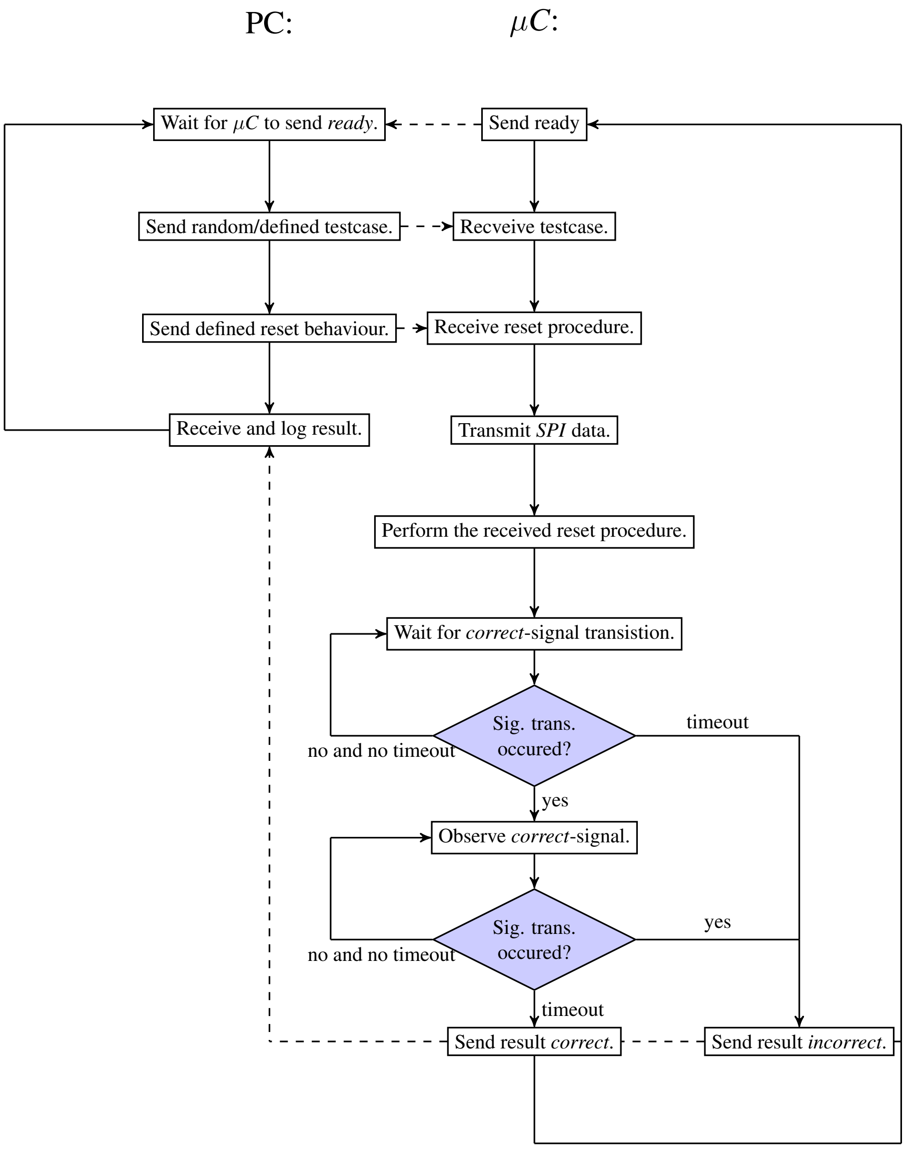


Figure 5.2: *PC* (left) and μC (right) data exchange and control flow, including the FPGA generated *correct* signal (see Figure 5.3).

The BSCLK is incremented with a precision/skew $\Sigma^b = 4.16\,\mu$s and its LSB accuracy bound, i.e. the time interval between transitions of the same polarity, is just twice the accuracy provided by the Quick-Cycle state machine $[T^b_{\text{min}}, T^b_{\text{max}}] = [115.34\,\mu\text{s}, 152.32\,\mu\text{s}]$. Assuming that the bounded delay of the labeling algorithm $d^l = d$, then the resulting labeling skew is $\Sigma^l = \Sigma^b + d^l = 5.76\,\mu$s. The delay d^l must include all communication delays of the labeling algorithm including the propagation delays of asynchronous logic functions between the registers. The skews or delays should also include all clock uncertainties.

Then the following timing constraints between all interfaces and nodes of the labeling algorithm must be met (see Chapter 2.7):

Setup FSM↔FSM: $T^+_{\text{min}} - \Sigma^l - d^l - t_{\text{S}} > 0.$

Hold FSM↔FSM: $T^+_{\text{min}} - \Sigma^l - t_{\text{H}} > 0.$

Setup BSCLK→FSM: $T^+_{\text{min}} - d^l - t_{\text{S}} > 0.$

Hold BSCLK→FSM: $T^+_{\text{min}} - d^l - t_{\text{H}} > 0.$

The FSM constraints copes with all local paths and registers of a labeling node as well as the paths between the nodes' input and output port (registers).

The BSCLK constraints needs to be verified to ensure the correct operation of the BSCLK overflow flag used by the FSM, since the upper bits of the BSCLK are part of the Quick-Cycle clock domain (*FATAL*$^+$-pulse).

Inserting the above *FATAL*$^+$ bounds into the constraints yields no violations, because the setup and hold parameters, t_{S} resp. t_{H}, of latches and registers can be assumed to be just several nano seconds, and there are still a reserve of some μs. In my case, the *Quartus II* uses latches to emulate the flip-flops to be able to asynchronously load those registers, but as these requirements can be fulfilled in the standalone testbench using 12.5 MHz as the BSCLK LSB clock frequency, it is reasonable to assume there are no problems after integration. Moreover, the timing analysis revealed that the standalone testbench would be able to run at 38 MHz.

5.3 Experimental Validation

There a lot of possibilities to validate a given design. Apart from simulations, it is necessary to experimentally validate a design in a given physical setting to observe its real behaviour and validate it against its specified behaviour in the case that some assumptions don't hold or a mistake has been made either in the implementation specification or by the synthesis tools. Note that no specific timing analysis has been performed in the integrated testbench setting. This should be no problem here, because the synchronized round period is large enough, but this need not always be the case.

Possible validation procedures range from golden nodes, capturing all outputs and important states fast enough and transmitting them at the end for analysis, to using an in-circuit validation node, which monitors the important part of the system state and validates it against certain assumptions. Obviously, I don't have a reference node available, the expansion header is too limited to capture all bits externally with a logic analyzer and the internal memory capabilities are limited. Thus, state transfers to an external memory, storage device or network device would be necessary for implementing this option. Fortunately, for my purpose, it is sufficient to validate the execution at each correct node against the proofs assumptions, which is that the labels at correct nodes must be equal after a full iteration of Algorithm 1 and always incremented in the following iterations. Formally:

- After $L_i = L_j$ at any correct node i and j, the next label is $L_i + 1$.

This assumption must hold at each BSCLK overflow. Since the label is updated at the overflow (last falling transition) and the BSCLK must be 0 when the FATAL-pulse (i.e., the node is in the *Accept*-state) occurs, the use of these pulses is sufficient for the validation node to recognize the overflow.

All assumptions are validated by a synchronous validation algorithm driven by a separate PLL-clock at $70\ MHz$ described at the RT-level in VHDL, which provides the *correct* signal to indicate the stabilization of the labeling algorithm. For this purpose, the validation node continuously samples the labels of the correct nodes and a *indication* signal, which is active if all correct nodes are in the FATAL-pulse state, using a two stage synchronizer chain.

Therefore, the validation node waits for this indication signal to become active to detect the BSCLK overflow at all correct nodes. Once the indication signal is inactive again, the validation node increments the stabilized label candidate, if one is present, and compares the sampled label values of the correct nodes against each other for equality. If this first part of the assumption holds this label will be remembered as stabilization candidate by the validation node for the next BSCLK overflow. If there is a stabilized label candidate already and the correct node labels match this candidate, the correct signal is set to true, otherwise the correct signal is set to false. Hence, the stabilization candidate label is used to verify that the correct nodes increment their labels appropriately. After that the validation node waits for the next BSCLK overflow.

Note that the correct signal becomes true one BSCLK overflow after the labeling algorithm stabilized, since validating that the correct node labels increment needs an additional iteration of the labeling algorithm (e.g., BSCLK overflow).

Due to the sampling of the validation node and the labeling algorithm skew Σ^l, timeouts have been employed to cover these, i.e., if the assumption does not hold at least $\Sigma^l + d^l$ after the indication signal became false, the validation node may wait for the next overflow. Moreover, a further timeout, which requires the indication signal to become active again within $T_{\text{max}} + d$, ensures that the labeling state or FATAL state machine can't freeze. If any such a timeout occurs, before the required assumptions become true, the correct signal will be reset to false. All these timeouts are configurable and provided as VHDL generics.

This validation flow is depicted in Figure 5.3.

Standalone testbench

The PLL clock output frequency used by the BSCLK simulation counters is set to $12.5\ MHz$, but the clocks 2,3 and 4 are shifted by $45°$, $90°$ and $135°$ to simulate non-zero precision/skew.

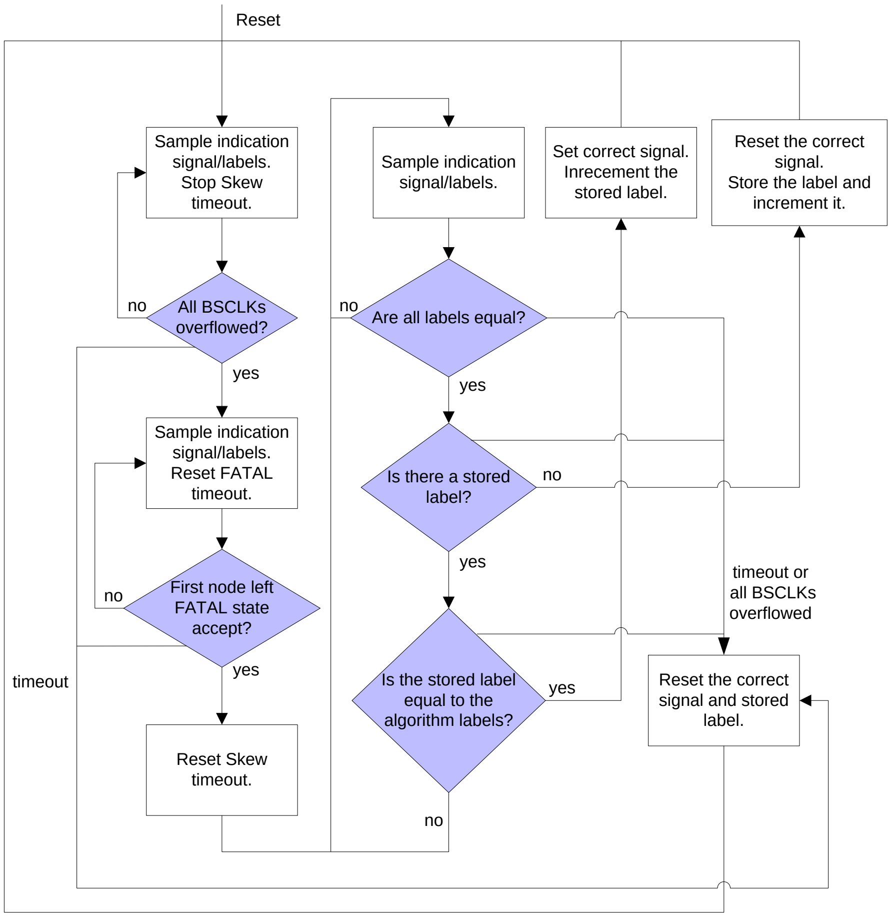

Reset
Sample indication signal/labels. Stop Skew timeout.
no
All BSCLKs overflowed?
yes
Sample indication signal/labels. Reset FATAL timeout.
no
First node left FATAL state accept?
yes
Reset Skew timeout.
timeout
Sample indication signal/labels.
no
Are all labels equal?
yes
Is there a stored label?
no
yes
Is the stored label equal to the algorithm labels?
yes
no
Set correct signal. Inrecement the stored label.
Reset the correct signal. Store the label and increment it.
timeout or all BSCLKs overflowed
Reset the correct signal and stored label.

Figure 5.3: Shows the labeling monitoring control flow and correct signal generation by the validation node.

Additionally the BSCLK counter value is affected by the chosen reset state too, i.e., it starts with an randomly chosen value.

The BSCLK reset ($SwRstClk$) is synchronized to the first PLL output clock to ensure that all counters are reset synchronously and no upsets occur.

As mentioned before, the first 2 nodes are used to represent the Byzantine faulty nodes and the validation node is connected to the label outputs of the labeling nodes and the simulated slow FATAL-pulses. The *Experiment 1*-reset procedure is used to start the Byzantine node experiments. Note that the correct operation of the validation node has been verified with some manual test cases.

Since the μC is used to measure the required stabilization time, the validation node provides an additional signal that toggles each time the BSCLK overflows. The μC counts the amount of toggles and transmit this required number of overflows of the BSCLK as stabilization time to the PC.

Besides, I use the second reset procedure to start the late join experiments, i.e., all nodes are correct, but at most $f = 2$ nodes may be reset arbitrarily and the rest of the system remains untouched. In this case, the validation node additionally ensures that the remaining nodes keep operating correctly, i.e., without resetting their labels, and the failed must adapt the labels of the already correct nodes.

As mentioned above, the validation node will output the correct signal one BSCLK overflow later than the labeling algorithm stabilized. Furthermore, the first toggle can never be a full overflow of the BSCLK, i.e., a full iteration of the labeling algorithm, resulting from chosen reset value of the BSCLK in the Byzantine experiment or its current value at a reset of a node in the late join experiment. For example, if the BSCLK modulo $M = 132$ and the BSCLKs are reset to 0 by the chosen reset state, a FATAL-pulse is generated that produces a toggle at the signal provided by the validation node and is counted by the μC.

Hence, subtracting 2 from the recorded stabilization time gives the amount of full overflows, e.g., full iterations of the labeling algorithm.

The standalone testbench hardware effort with about 3407 logic cells is pretty low (about 375 LC per node). The late join experiment uses 4193 logic cells (about 375 LC per node). Using the current FATAL$^+$ prototype with $f = 2$ and $M = 132$ a 17-bit label can be achieved. Besides, the timing analysis reveals that the current standalone testbench would allow an BSCLK LSB clock frequency of up to 38 MHz.

Integrated testbench

In this case, the PLL outputs are set to 25 and 30 MHz, which is further divided by 4 of the clock-switch logic that selects either of these two outputs to drive the timers and the HSMs of FATAL$^+$ to simulate a larger clock drift ρ. Only the Byzantine test case is performed, since the correct behaviour follows from the previous experiments of FATAL$^+$ [66], the standalone testbench and the combined Byzantine testbench.

Furthermore, the stabilization time is measured by a μC software timer with a resolution of about 1 ms instead of counting the toggles provided by the validation node, because in general FATAL$^+$ has not stabilized yet. Hence, the required end-to-end stabilization can be measured. The μC waits at most 60 seconds for a single experiment to complete. Once the correct signal occurs it observes the correct signal for further 10 seconds before it accepts the correct event time instance as the stabilization time as result of the performed experiment.

Overall, 62543 logic cells have been used by the testbench, but only 2832 logic cells correspond to the Byzantine and correct nodes of the labeling algorithm resp. the validation node.

5.4 Results

Standalone testbench

As expected the stabilization time is at most two full BSCLK overflows proofing the fast stabilization of the developed algorithm. Table 5.1 depicts the relative frequency of stabilization times given in complete overflows in 50000 runs (late join, Byzantine experiment with and without consensus). As mentioned in the description of the reduction Algorithm 2 and the Byzantine node implementation, the reduction is sufficient to solve the labeling problem, without the need of a consensus to be run afterward, if either less than $n - 2f$ correct nodes enter the algorithm with the same label or the Byzantine faulty nodes are reduced in adverse power.

Any requirement and assumption could be fulfilled, i.e. neither the late joining nodes were able to affect the already stabilized nodes, but rejoined the correct nodes, nor could the Byzantine nodes stop the correct nodes from stabilizing. Moreover, the results show an significant amount of runs can never stabilize if no consensus algorithm is used. This implies that the consensus algorithm is indeed necessary to guarantee stabilization in any case, if no other means are available or different failure assumptions can be made.

Consensus	faults f	**1**	**2**	∞
Phase King	late join	1.0	0	0
Phase King	2	0.737	0.263	0
none	2	0.586	0.073	0.341

Table 5.1: Relative frequency of stabilization times of 1 (50000 randomly initialized runs).

Integrated testbench

The integrated testbench revealed a design error in the current FATAL$^+$ prototype, which becomes active as soon as an output of some link delay chains are used instead of the direct output port of a node. These delay chains have been deployed to simulate longer delays than the actual available link delay provided by the interconnect in the FPGA and its logic. As a result, the BSCLK is interrupted at a non-zero value by the slow FATAL-pulse, which is a violation of the BSCLK specification since it must be 0 once the FATAL pulse occurs.

Therefore the current testbench has been reduced to delay 0 experiments, i.e., without resetting the FATAL$^+$-prototype ($SwRstClk$), and using any output of the delay chain. Hence, only correctness and no stabilization time analysis have been performed using 50000 random initialized runs of the labeling algorithm in the context of FATAL$^+$-prototype, which showed that the labeling prototype would work in a stabilized environment. Using the current FATAL$^+$ prototype with $f = 2$ and $M = 132$ a 17-bit label can be achieved.

These problems couldn't be removed via minor adjustments of the FATAL$^+$-timeouts. I conjecture that there is either a design flaw in the current FATAL$^+$ prototype or still an error in the formulas used to set the timeout values given in Chapter 3.1.2. Before handing in my thesis, the timeout equations of FATAL$^+$ have been corrected. After updating the timeouts accordingly, the issue reduced to just one case in 3000 runs. The preliminary result of the end-to-end stabilization times in seconds, using the full power of the integrated testbench (and not just zero delay experiments), is shown in Figure 5.4. As can be seen, the stabilization times have not increased significantly, except by at most 2 overflows for the stabilization of the labeling algorithm. Moreover, the experiment shows fast stabilization within 10 seconds, where about 40% of the runs stabilized within 26 to 50 ms. Considering these facts, a solution must be found in the future for this (rare) stabilization issue.

Figure 5.5 shows the start of a new integrated testbench experiment. The change of the labels can be observed once the node reset occurs.

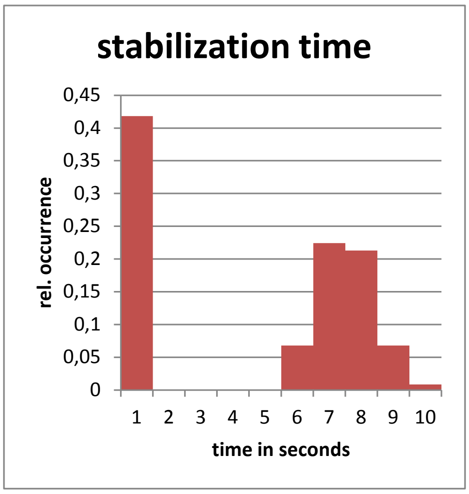


Figure 5.4: Preliminary results on the FATAL$^+$ and labeling algorithm end-to-end stabilization times (3000 randomly initialized runs).

Figure 5.6 resp. Figure 5.7 show the logic analyzer screen shots of a detailed BSCLK overflow at the lock-step synchronous round with the number 65 and the correct incrementing of the labels resp. the continued correct operation of the labeling algorithm over multiple BSCLK overflows. Once the labels increment as required the correct signal, produced by the validation node, indicates the stabilization by switching to logical 1.

Note that only the lowest 5-bits of a label at two nodes are depicted due to the limitation on the available output pins. The *AllFatal*-pulse is the **AND** combination of all slow FATAL-pulse signals (i.e., all nodes are in the state *Accept*).

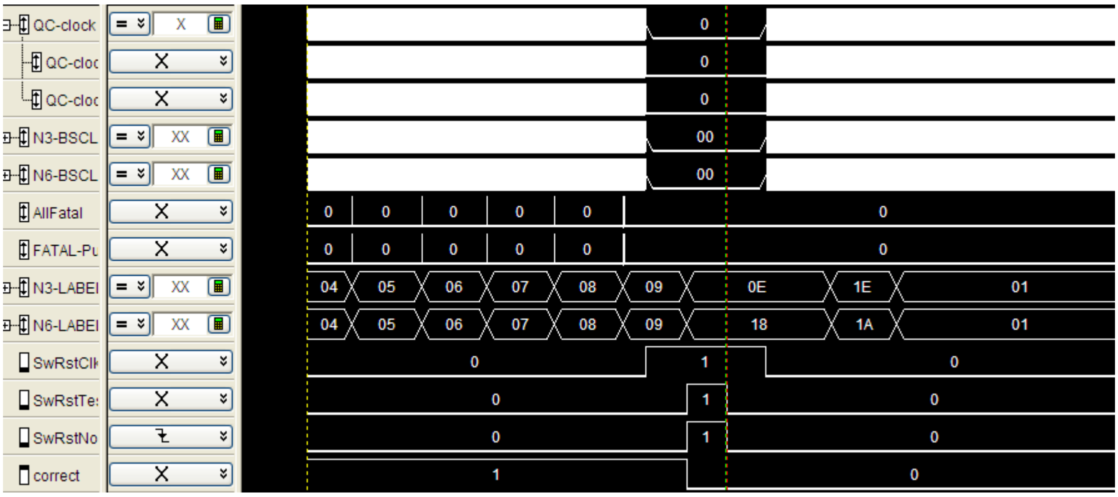


Figure 5.5: Reset behaviour in the integrated testbench.

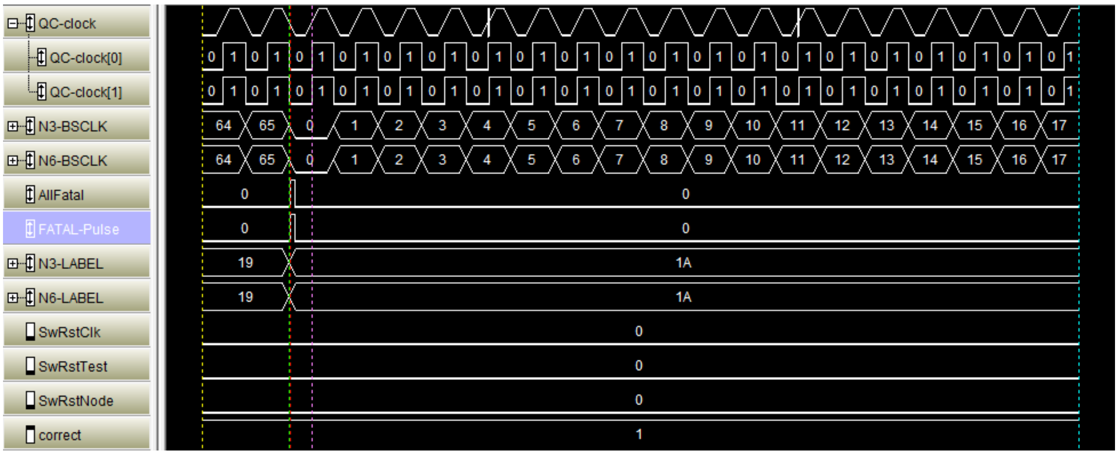


Figure 5.6: Detailed label increment at a BSCLK overflow.

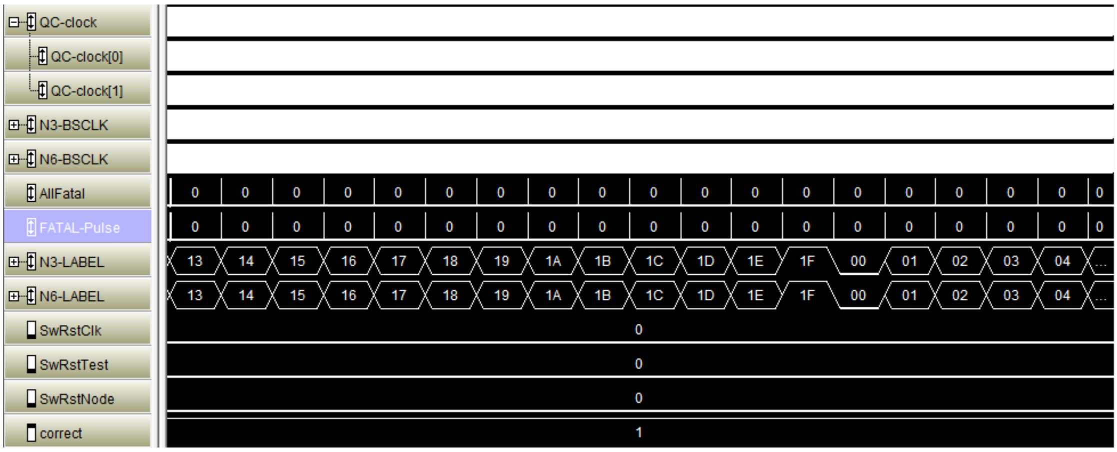


Figure 5.7: Correct label increments at multiple BSCLK overflows.

CHAPTER 6

Conclusion

6.1 Summary of accomplishments

In this thesis it has been shown that is possible to extend a short clock with reasonable small additional hardware effort in a self-stabilizing and Byzantine fault-tolerant manner. The presented implementation increases the label width by a factor of 3-4 in a 8 node system, if a full 8-bit clock is available, using the standard synchronous design principle. Since the BSCLK is used to simulate lock-step synchronous rounds, the implementation is even metastability-free once the BSCLK has stabilized.

Due to the generic design, it is easy to adjust the prototype and let the synthesis tools automatically synthesize the labeling algorithm for arbitrary system sizes.

Moreover, a configurable standalone testbench, that validated the design using 50000 randomly initialized runs on an FPGA successfully, is provided. The validation technique proofed to be reliable, since I was able to discover a violation in the BSCLK generation of the FATAL$^+$ prototype after integration. Updating the timeouts according to the correct timeout equations of FATAL$^+$ reduced this issue. The preliminary results of 3000 runs show that the overall stabilization time FATAL$^+$ has not increased after integration. Therefore, the FATAL$^+$ prototype and the labeling algorithm provide a sound basis for SoC designs to create more reliable systems by providing a large global time base to the independent subsystems.

6.2 Critical reflection and future work

6.2.1 Achievable clock frequency

Quartus II reports a maximum BSCLK LSB frequency of 38 MHz for the standalone testbench. By changing the clock trigger to the falling transition at the second synchronizer stage of the receiver buffers at a node and incrementing the label candidate in the next state logic of the label value when the overflow flag is set, increased the frequency up to 90 MHz, but due to the lack of time no further validation has been performed.

If the single synchronous FSM, which implements both the reduction and consensus algorithm, limit the frequency, separating the reduction and consensus algorithm into two FSMs may decrease the undesired delays. Note that it may be even possible to use (synchronous) pipelining to increase the achievable clock frequency. The most likely candidate for a pipelined implementation is the threshold module, because its complexity depends on the system size n. The attentive reader should not forget that this approach has a multiplicative effect on the consensus algorithm run time.

Since Quartus II reports a maximum BSCLK simulation frequency of 150 MHz in the standalone testbench, which implies a BSCLK LSB frequency of at most 75 MHz, an achievable maximum FSM clock frequency of 90 MHz is more than sufficient.

6.2.2 Label size l

Plugging in $n = 8$, $f = 2$ and the BSCLK size $M = 132$ into Equation 4.12 it is apparent that a label width larger than 17-bit couldn't be achieved. If $M = 256$, i.e., a full 8-bit clock was available, a label width of 32-bit could be reached. There are two basic possibilities that can increase the number of label bits resp. faults that can be tolerated: Either the number of available rounds is increased or the required run time reduced. A straight forward way to double the number of available synchronous rounds is to directly use the Accept$^+$-pulses, rather than the BSCLK's LSB, for triggering round switches. Fortunately, Algorithm 2 can be sped up by means of parallel processing and/or communication of the label bits.

A straight forward approach would be to use $B > 1$ instances of the labeling solution, applied to different parts of the label, concurrently. This obviously multiplies the gate complexity of a node by B and also requires B-bit channels between nodes. Alternatively, the label can be extended recursively: A shorter logical clock can be further extended in the same way as the BSCLK has been extended, using the reduction and the Phase King consensus algorithm, but this approach obviously increases the overall stabilization time.

Another possibility is to only "parallelize" communication. In the current implementation only 1-bit per round is transmitted using a single physical channel. Using parallel communication or a UART-like transmitter and receiver would enable to transmit $B > 1$ bits per round. In chapter 2, examples of fast (and almost serial) communication protocols suitable for reducing the effect of a large synchronous round period on the transmission rate by continuously transmitting and possibly acknowledging only words or frames consisting of multiple state bits have been provided. The FSM of Algorithm 2 can be adapted easily such that only $\lceil l/B \rceil$ iterations of the two for-loops are needed.

Unfortunately, parallel processing (or communication) can only be used for decreasing the number of rounds required for processing the label bits, but not the number of rounds required for the consensus algorithm: Deterministic consensus has a lower bound of $f+1$ rounds [32,51,55]. Fortunately, randomized algorithms do not suffer from this overhead and can be used in this context. A suitable algorithm shown in Algorithm 4 has been introduced in [39].

6.2.3 Alternative consensus

The first part of the loop in Algorithm 4 is equivalent to the Phase King consensus. In the second part a correct node i tells the others if it would like to proposes a certain output ***true*** or ***false***, but only when its persistent variable $\Delta_{i,i} = 0$ and the outcome of the random process indicates a success. Herein, a persistent variable keeps its value at the end of the consensus algorithm until it is executed again. Node i only accepts a wish to propose a value from node j if its persistent variable $\Delta_{i,j} = 0$ and thus sets its $\Delta_{i,j} := n$ to start over and reduce the effect of possible Byzantine nodes, i.e., it limits how often a (Byzantine) node may influence a correct node. At last step, a node tells the others that it has accepted to propose some output value. All correct nodes that receive at least $n - 2f$ such accepts from correct nodes update their output candidate b_i and lock their decision. Since a correct node i needs the random process to succeed before it performs the broadcasts, indicating it would like to propose a value, there exists a significant chance that $\Delta_{j,i} = 0$ at all other correct nodes j at some time node i performs the broadcast. If not all correct nodes have locked their output to another value all correct nodes will lock their decision to the proposed value now.

Algorithm 4: Randomized consensus algorithm including a uniform independent distributed random source and persistent variables at node a node i. The persistent variables $\Delta_{i,j}$ can take values from the range $0, \ldots, n$. [39]

```
input : $b_i$ //binary value to be decided on
output: $o_i$ //binary value representing the consensus decision.
for $l$ times do
    broadcast $b_i$
    if received $\geq n - f$ times $b$ then
        broadcast $1b$ // $b$ is unique
    else broadcast $00$ // 2 rounds for 2-bit bcast
    locked := received $\geq n - f$ times $1b$
    if received $\geq f + 1$ times $1b$ then
        $b_i := b$ // $b$ remains unique
    for $j \in V$ do $\Delta_{i,j} := \max\{0, \Delta_{i,j} - 1\}$
    for $b \in \{0, 1\}$ do
        $p_b :=$ false // indicates if $b$ is proposed
        if $\Delta_{i,i} = 0$ and $b_i = b$ then
            broadcast $1$ with probability $1/n$
        else broadcast $0$
        for each node $j$ that sent $1$ do
            if $\Delta_{i,j} = 0$ then $p_b :=$ true
            $\Delta_{i,j} := n$
        broadcast $p_b$
        if locked = false and received $\geq n - f$ times true then
            $b_i := b$
            locked := true
return $b_i$
```

As source of randomness the same linear feedback shift register as in [66] has been chosen and its output is compared against $2^{16}/8$ to get a uniform distribution. Furthermore, each node needs additional memory elements for each other node, including itself, to implement the persistent variables. Although it needs more hardware, its run time does not depend on f anymore. If the number of nodes resp. the tolerated number of faults f is not much larger than the label width l there won't be any advantage using this algorithm instead of the Phase King algorithm since the label width will have the most impact on the round complexity and will be lower in the Phase King case.

In the current prototype only a label width of 4-bits could be achieved using the same BSCLK and the randomized algorithm due to the synchronizer chain. The attentive reader should notice that the number of iterations in this probabilistic consensus can also be independent of the label width l and just be some arbitrary constant, but the smaller this constant is the more BSCLK overflows it will take for the algorithm to stabilize. The histogram in Figure 6.1 shows the observed number of overflows in the standalone testbench. Currently, the effect of the Byzantine node implementation on the stabilization time is rather low, because it is almost equivalent to the Phase King implementation, and it mostly depends on the counters to reach 0 before the labels stabilize. However, the stabilization time increased to 8 overflows at most in the performed experiments, but large stabilization times are rather improbable (less than 0.01).

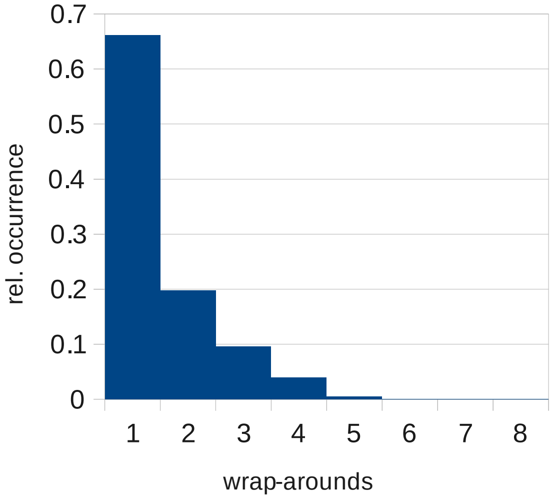


Figure 6.1: Rel. frequency of stabilization times of 4(50000 randomly initialized runs in the standalone testbench).

6.2.4 Miscellaneous

Note that further validation of FATAL$^+$ and the labeling algorithm using the integrated testbench is needed, since the new timeouts only reduced the stabilization problem. In the 3000 runs of the experiment, only a single case occurred in which the integrated testbench could not stabilize (within 50 seconds). Since, such a case was not reproducible, I conjecture that there still is an BSCLK overflow issue at some node. For this purpose, automatic timeout verification tools are needed to verify all required constraints, since some vendor tools are currently unable to process real custom asynchronous timing requirements easily. Currently, all these delays must be verified manually as the tools are optimized for synchronous circuits usually.

APPENDIX A

Thesis directory contents

The thesis directory is split into the following important directories:

Directory	**Description**
`prj`	Contains all necessary `.tcl` files and further source files to create the integration and standalone testbench *Quartus* projects.
`python`	Contains all necessary `Python` sources to provide the standalone resp. integration testbenches with new (random) states and receive, log and parse the results.
`sim`	Contains some standalone testbench files for behavioural as well as post-layout simulation in *Modelsim*.
`thesis`	All files needed to build this document (including `Make`-files).
`uC`	The μC standalone and integration testbench μC implementations including `Make`-files.
`vhdl-src`	All `VHDL`-files introduced by this thesis.

Before a testbench can be run it must be verified by the user that all parameters match (e.g. the label width, connected inputs and output pins, number of nodes, etc.), which can be found in the following files:

File	**Description**
`prj/standalone/` `fatal_labeling_test.tcl`	Standalone testbench project setups including input and output pin connection settings.
`prj/integration/` `FATALsim.tcl`	Integration testbench project setups including input and output pin connection settings.
`python/standalone/main.py`	Standalone testbench host-PC implementation. Widths and reset vector positions must match the `VHDL` package files. UART communication speed must match the μC setting.
`python/integration/main.py`	Integration testbench host-PC implementation. Widths and reset vector positions must match the `VHDL` package files. UART communication speed must match the μC setting.
`uC/standalone/main.c`	Contains the standalone testbench μC-implementation. Its input and output pins must match the definitions in the `.tcl` and `VHDL` package files.
`uC/integration/main.c`	Contains the integration testbench μC-implementation. Its input and output pins must match the definitions in the `.tcl` and `VHDL` package files.
`vhdl-src/` `pb_fatal_labeling.vhd`	A package that contains all necessary testbench parameters.

The remaining important labeling `VHDL` sources are:

File	**Description**
`ea_fatal_labeling.vhd`	Connects all configured correct nodes, faulty nodes and the validation node.
`ea_fatal_labeling_byzantine.vhd`	Byzantine node algorithm FSM, buffers and overfow flag implementation.
`ea_fatal_labeling_fsm.vhd`	A single correct node algorithm FSM, buffers and overfow flag implementation.
`ea_fatal_labeling_sim.vhd`	Standalone testbench implementation. Connects the labeling entity with the simulated BSCLK.
`ea_fatal_labeling_threshold.vhd`	Threshold module implementation.
`ea_fatal_test_assump.vhd`	Validation node FSM implementation.
`ea_fatalp_bsclk_sim.vhd`	BSCLK standalone simulation implementation

After adjusting the parameters and making sure that the testbench setup and physical connections match, the `.tcl` files can be used to create the project, compile and download the design to the FPGA using *Quartus II*. Next, the required *Python* host implementation must be started by running `main.py` and the μC-implementation compiled and downloaded using the `Makefile` to start the testbench. If no parameters need to be changed, i.e., the settings of this thesis' experiments are used, the project can be build directly using *Quartus II* and the provided `.tcl` files.

Bibliography

[1] Altera. *The Quartus II TimeQuest Timing Analyzer*, 2012.

[2] Altera. *Timing Analysis Overview*, 2012.

[3] Altera. *Cyclone IV Device Handbook*, 2013.

[4] Anish Arora, Shlomi Dolev, and Mohamed Gouda. Maintaining digital clocks in step. In Sam Toueg, PaulG. Spirakis, and Lefteris Kirousis, editors, *Distributed Algorithms*, volume 579 of *Lecture Notes in Computer Science*, pages 71–79. Springer Berlin Heidelberg, 1992.

[5] L. Ashby. Asic clock distribution using a phase locked loop (pll). In *ASIC Conference and Exhibit, 1991. Proceedings., Fourth Annual IEEE International*, pages P1–6/1–3, 1991.

[6] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions*, 1(1):11–33, 2004.

[7] R.C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *Device and Materials Reliability, IEEE Transactions*, 5(3):305 – 316, sept. 2005.

[8] Valeriu Beiu, Jan A. Peperstraete, Joos Vandewalle, and Rudy Lauwereins. Digital implementation of neural networks using threshold gates, 1994.

[9] Michael Ben-Or, Danny Dolev, and Ezra N. Hoch. Fast self-stabilizing Byzantine tolerant digital clock synchronization. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, PODC ’08, pages 385–394, New York, NY, USA, 2008. ACM.

[10] Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Computer science. chapter Bit optimal distributed consensus, pages 313–321. Plenum Press, New York, NY, USA, 1992.

[11] Piotr Berman and JuanA. Garay. Asymptotically optimal distributed consensus. In Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and SimonettaRonchi Rocca, editors, *Automata, Languages and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 80–94. Springer Berlin Heidelberg, 1989.

[12] Daniel R. Blum, Mitchell J. Myjak, and José G. Delgado-frias. Enhanced Fault-Tolerant Data Latches for Deep Submicron CMOS. In *International Conference on Computer Design*, pages 28–34, 2005.

[13] D. M. Chapiro. *Globally-asynchronous locally-synchronous systems*. PhD thesis, Stanford Univ., CA., 1984.

[14] Markus Chmelar. Building a threshold-gate with a sorting network with application to the darts project. Research Report 50/2012, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2012.

[15] Dhiman Deb Chowdhury. *High Speed LAN Technology Handbook*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 2000.

[16] C. Constantinescu. Trends and challenges in vlsi circuit reliability. *Micro, IEEE*, 23(4):14 – 19, july-aug. 2003.

[17] Ariel Daliot, Danny Dolev, and Hanna Parnas. Linear time byzantine self-stabilizing clock synchronization. In Marina Papatriantafilou and Philippe Hunel, editors, *Principles of Distributed Systems*, volume 3144 of *Lecture Notes in Computer Science*, pages 7–19. Springer Berlin Heidelberg, 2004.

[18] W.J. Dally and S.G. Tell. The even/odd synchronizer: A fast, all-digital, periodic synchronizer. In *Asynchronous Circuits and Systems (ASYNC), 2010 IEEE Symposium*, pages 75 –84, may 2010.

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

[20] Martin Delvai. *Design of an Asynchronous Processor Based on Code Alternation Logic - Treatment of Non-Linear Data Paths*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2005.

[21] Martin Delvai and Andreas Steininger. Asynchronous logic design - from concepts to implementation. *The 3rd International Conference on Cybernetics and Information Technologies, Systems and Applications - Volume 1*, Jan. 2006.

[22] A. Dixit and A. Wood. The impact of new technology on soft error rates. In *Reliability Physics Symposium (IRPS), 2011 IEEE International*, pages 5B.4.1 –5B.4.7, april 2011.

[23] R. Dobkin, R. Ginosar, and A. Kolodny. Fast asynchronous shift register for bit-serial communication. In *Asynchronous Circuits and Systems, 2006. 12th IEEE International Symposium*, pages 10 pp. –127, march 2006.

[24] R. Dobkin, R. Ginosar, and C.P. Sotiriou. High rate data synchronization in GALS SoCs. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions*, 14(10):1063 –1074, oct. 2006.

[25] Danny Dolev, Matthias Függer, Christoph Lenzen, Markus Posch, Ulrich Sch mid, and Andreas Steininger. FATAL$^+$: A Self-Stabilizing Byzantine Fault-tolerant Clocking Scheme for SoCs. *Computing Research Repository*, abs/1202.1925, 2012.

[26] Danny Dolev, Matthias Függer, Christoph Lenzen, and Ulrich Schmid. Fault-tolerant algorithms for tick-generation in asynchronous logic: Robust pulse generation. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 6976 of *Lecture Notes in Computer Science*, pages 163–177. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-24550-3_14.

[27] Danny Dolev, Joseph Y. Halpern, and H. Raymond Strong. On the possibility and impossibility of achieving clock synchronization. *Journal of Computer and System Sciences*, 32:230–250, 1986.

[28] Shlomi Dolev and Jennifer L. Welch. Self-stabilizing clock synchronization in the presence of byzantine faults. *J. ACM*, 51(5):780–799, September 2004.

[29] W. Elmenreich and M. Delvai. Time-triggered communication with UARTs. In *Factory Communication Systems, 2002. 4th IEEE International Workshop*, pages 97 – 104, 2002.

[30] S. Fairbanks and S. Moore. Self-timed circuitry for global clocking. In *Asynchronous Circuits and Systems, 2005. ASYNC 2005. Proceedings. 11th IEEE International Symposium*, pages 86 – 96, march 2005.

[31] K.H. Fatemeh and A.K. Horestani. New structure for adder with improved speed, area and power. In *Networked Embedded Systems for Enterprise Applications (NESEA), 2011 IEEE 2nd International Conference*, pages 1 –6, dec. 2011.

[32] MichaelJ. Fischer, NancyA. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, 1986.

[33] International Technology Roadmap for Semiconductors. International technology roadmap for semiconductors, 2011 edition, executive summary, 2011.

[34] H. Frazier and H. Johnson. Gigabit ethernet: from 100 to 1,000 mbps. *Internet Computing, IEEE*, 3(1):24 –31, jan/feb 1999.

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

[36] G. Fuchs, M. Fugger, and A. Steininger. On the threat of metastability in an asynchronous fault-tolerant clock generation scheme. In *Asynchronous Circuits and Systems, 2009. ASYNC '09. 15th IEEE Symposium*, pages 127 –136, may 2009.

[37] Gottfried Fuchs, Julian Grahsl, Ulrich Schmid, Andreas Steininger, and Gerald Kempf. Threshold Modules – Die Schlüsselelemente zur verteilten Generierung eines fehlertoleranten Taktes. *The Austrian National Conference on the Design of Integrated Circuits and Systems (Austrochip 2006)*, Oct. 2006.

[38] Gottfried Fuchs and Andreas Steininger. VLSI implementation of a distributed algorithm for fault-tolerant clock generation. *JECE*, 2011:4:4–4:4, January 2011.

[39] Matthias Függer, Christoph Lenzen, Ulrich Schmid, and Markus Hofstätter. Efficient Construction of Global Time in SoCs despite Arbitrary Faults. In *16th Euromicro Conference on Digital System Design (to appear)*, 2013.

[40] Matthias Függer and Ulrich Schmid. Reconciling fault-tolerant distributed computing and systems-on-chip. *Distributed Computing*, 24:323–355, 2012.

[41] R. Ginosar. Fourteen ways to fool your synchronizer. In *Asynchronous Circuits and Systems, 2003. Proceedings. Ninth International Symposium*, pages 89 – 96, may 2003.

[42] V. Gutnik and A.P. Chandrakasan. Active GHz clock network using distributed PLLs. *Solid-State Circuits, IEEE Journal of*, 35(11):1553 –1560, nov. 2000.

[43] S. Hauck. Asynchronous design methodologies: an overview. *Proceedings of the IEEE*, 83(1):69 –93, jan 1995.

[44] Ezra N. Hoch, Danny Dolev, and Ariel Daliot. Self-stabilizing Byzantine Digital Clock Synchronization. In AjoyK. Datta and Maria Gradinariu, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 4280 of *Lecture Notes in Computer Science*, pages 350–362. Springer Berlin Heidelberg, 2006.

[45] A.B. Howe and C.L. Coates. Logic hazards in threshold networks. *Computers, IEEE Transactions*, C-17(3):238 – 251, march 1968.

[46] L. Kleeman and A. Cantoni. Metastable behavior in digital systems. *Design Test of Computers, IEEE*, 4(6):4 –19, dec. 1987.

[47] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, 1997.

[48] Stanford University. Computer Systems Laboratory and C.L. Portmann. *Characterization and reduction of metastability errors in cmos interface circuits*. 1995.

[49] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[50] Leslie Lamport. Arbitration-free synchronization. *Distrib. Comput.*, 16(2-3):219–237, September 2003.

[51] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[52] Dong-Jin Lee, Myung-Chul Kim, and I.L. Markov. Low-power clock trees for CPUs. In *Computer-Aided Design (ICCAD), 2010 IEEE/ACM International Conference*, pages 444 –451, nov. 2010.

[53] Jennifer Lundelius and Nancy Lynch. An Upper and Lower Bound for Clock Synchronization. *Information and Control*, 62(2-3):190–204, 1984.

[54] Jennifer Lundelius and Nancy Lynch. A new fault-tolerant algorithm for clock synchronization, 1984.

[55] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[56] Leonard Marino. General theory of metastable operation. *IEEE Transactions on Computers*, C-30(2):107–115, February 1981.

[57] Y. Massoud, S. Majors, J. Kawa, T. Bustami, D. MacMillen, and J. White. Managing on-chip inductive effects. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions*, 10(6):789–798, 2002.

[58] M.S. Maza and M.L. Aranda. Analysis of clock distribution networks in the presence of crosstalk and groundbounce. In *Electronics, Circuits and Systems, 2001. ICECS 2001. The 8th IEEE International Conference*, volume 2, pages 773 –776 vol.2, 2001.

[59] Rene Mueller, Jens Teubner, and Gustavo Alonso. Sorting networks on FPGAs. *The VLDB Journal*, 21(1):1–23, February 2012.

[60] V.G. Oklobdzija, B.R. Zeydel, H.Q. Dao, S. Mathew, and R. Krishnamurthy. Comparison of high-performance VLSI adders in the energy-delay space. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions*, 13(6):754 –758, june 2005.

[61] A.K. Palit, V. Meyer, W. Anheier, and J. Schloeffel. Modeling and analysis of crosstalk coupling effect on the victim interconnect using the ABCD network model. In *Defect and Fault Tolerance in VLSI Systems, 2004. DFT 2004. Proceedings. 19th IEEE International Symposium*, pages 174 – 182, oct. 2004.

[62] T. Panhofer, W. Friesenbichler, and M. Delvai. Fault tolerant four-state logic by using self-healing cells. In *Computer Design, 2008. ICCD 2008. IEEE International Conference*, pages 1 –6, oct. 2008.

[63] M. Paulitsch and W. Steiner. Fault-tolerant clock synchronization for embedded distributed multi-cluster systems. In *Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference*, pages 249–256, July.

[64] Thomas Polzer, Thomas Handl, and Andreas Steininger. A metastability-free multi-synchronous communication scheme for SoCs. In *Stabilization, Safety, and Security of Distributed Systems, 11th International Symposium, SSS 2009, Lyon, France, November 3-6, 2009. Proceedings*, pages 578–592, 2009.

[65] Thomas Polzer and Andreas Steininger. C-element metastability mitigation using Schmitt-Triggers. Research Report 1/2012, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2012.

[66] Markus Posch. Selbststabilisierende Byzantinisch fehlertolerante Takterzeugung in FPGAs. Master's thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2012.

[67] P.J. Restle and A. Deutsch. Designing the best clock distribution network. In *VLSI Circuits, 1998. Digest of Technical Papers. 1998 Symposium*, pages 2–5, 1998.

[68] P.J. Restle, T.G. McNamara, D.A. Webber, P.J. Camporese, K.F. Eng, K.A. Jenkins, D.H. Allen, M.J. Rohn, M.P. Quaranta, D.W. Boerstler, C.J. Alpert, C.A. Carter, R.N. Bailey, J.G. Petrovick, B.L. Krauter, and B.D. McCredie. A clock distribution network for microprocessors. *Solid-State Circuits, IEEE Journal of*, 36(5):792–799, 2001.

[69] L.M. Reyneri, D. Del Corso, and B. Sacco. Oscillatory metastability in homogeneous and inhomogeneous flip-flops. *Solid-State Circuits, IEEE Journal of*, 25(1):254 –264, feb 1990.

[70] M. Saint-Laurent and M. Swaminathan. A multi-PLL clock distribution architecture for gigascale integration. In *VLSI, 2001. Proceedings. IEEE Computer Society Workshop*, pages 30 –35, may 2001.

[71] Fred B. Schneider. A paradigm for reliable clock synchronization. In *Proceedings Advanced Seminar of Local Area Networks*, pages 85–104, Bandol, France, April 1986.

[72] Maitham Shams, Jo C. Ebergen, and Mohamed I. Elmasry. Modeling and Comparing CMOS Implementations of the C-Element. *IEEE Transactions on VLSI Systems*, 6(4), 1998.

[73] Jens Sparso. Asynchronous circuit design – a tutorial, 2006.

[74] T. K. Srikanth and Sam Toueg. Optimal clock synchronization. *J. ACM*, 34(3):626–645, July 1987.

[75] Andreas Steininger. Advanced Digital Design Lecture Slides - GALS, 2012.

[76] I. E. Sutherland. Micropipelines. *Commun. ACM*, 32(6):720–738, June 1989.

[77] P. Teehan, M. Greenstreet, and G. Lemieux. A Survey and Taxonomy of GALS Design Styles. *Design Test of Computers, IEEE*, 24(5):418 –428, sept.-oct. 2007.

[78] RS232 interface standard, 1997.

[79] C.H. van Berkel, M.B. Josephs, and S.M. Nowick. Applications of asynchronous circuits. *Proceedings of the IEEE*, 87(2):223 –233, feb 1999.

[80] J.H. Wensley, L. Lamport, J. Goldberg, M.W. Green, K.N. Levitt, P.M. Melliar-Smith, R.E. Shostak, and C.B. Weinstock. Sift: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240 – 1255, oct. 1978.

[81] Josef Widder and Ulrich Schmid. Booting clock synchronization in partially synchronous systems with hybrid process and link failures. *Distributed Computing*, 20(2):115–140, 2007.

[82] A. X. Widmer and P. A. Franaszek. A DC-Balanced, Partitioned-Block, 8B/10B Transmission Code. *IBM Journal of Research and Development*, 27(5):440 –451, sept. 1983.

[83] M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts. A fully integrated multi-CPU, GPU and memory controller 32nm processor. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, pages 264 –266, feb. 2011.

[84] A. Zargaran-Yazd, K. Keikhosravy, H. Rashtian, and S. Mirabbasi. Hardware-efficient phase-detection technique for digital clock and data recovery. *Electronics Letters*, 49(1):20 –22, 3 2013.

[85] R. Zimmer. *Binary Adder Architectures for Cell-Based VLSI and their Synthesis*. PhD thesis, Swiss Federal Institute of Techonology Zurich, 1997.