DISSERTATION

Fault-Tolerant Distributed Algorithms for On-Chip Tick Generation: Concepts, Implementations and Evaluations

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften

unter der Leitung von

A.o.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger
Institut für Technische Informatik
Embedded Computing Systems Group
Technische Universität Wien

eingereicht an der

Technischen Universität Wien
Fakultät für Informatik

von

Dipl.-Ing. Mag.rer.soc.oec. Gottfried Fuchs

gottfried.fuchs@inode.at

Matrikelnummer: 9825939
Canavesegasse 14/4
A-1230 Wien, Österreich

Wien, August 2009

Kurzfassung

Im Zuge dieser Dissertation wird ein neuartiger Ansatz zur On-Chip Generierung eines fehlertoleranten Taktes entwickelt und im Detail vorgestellt. Die Relevanz der Forschungsarbeiten wird dabei mit den immer kleiner werdenden Strukturgrößen im Chip-Design und dem damit einhergehenden Anstieg der Fehlerraten motiviert. Um zukünftige Schaltungen ausreichend robust gestalten zu können, muss in Anbetracht der erhöhten Fehlerraten unter anderem auch das Taktsignal, welches einen kritischen "single point of failure" von synchronen Schaltungen darstellt, durch Fehlertoleranz-Mechanismen geschützt oder durch fehlertolerante Alternativen ersetzt werden. In der vorliegenden Arbeit wird daher eine solche Alternative zu herkömmlichen zentral getakteten Schaltungen erarbeitet.

Der in dieser Arbeit vorgestellte Taktgenerierungsansatz basiert auf der Hardware-Implementierung eines bekannten verteilten Algorithmus. Das Besondere an dieser Implementierung ist, dass in einem System von $n \geq 3f + 2$ Knoten f dieser Einheiten beliebig (byzantinisch) fehlerhaft sein dürfen. Einen weiteren wichtigen Vorteil stellt die Tatsache dar, dass im Gegensatz zu herkömmlichen Verfahren keine Taktquellen (Oszillatoren) benötigt werden. Die asynchrone Implementierung des präsentierten Ansatzes ermöglicht es, die Taktsignale synchron zu generieren statt auf die Synchronisation von exisitierenden Taktquellen zurückzugreifen. Desweiteren, und noch viel wichtiger, werden durch die vorgeschlagene Architektur Metastabilitätsprobleme an den Schnittstellen zwischen verschiedenen Taktdomänen gänzlich vermieden.

Die Transformation des im Bereich der Software angesiedelten Algorithmus in die Welt des asynchronen Schaltungsentwurfs stellte sich als nicht trivialer Prozess heraus und repräsentiert einen wichtigen Teil der durchgeführten Arbeiten. Um den aus den zuvor erwähnten Transformationen schlußendlich hervorgehenden VLSI (Very Large Scale Integration) Chip und die darauf aufbauende fehlertolerante Taktgenerierungsarchitektur charakterisieren zu können, wurden umfangreiche Messreihen durchgeführt und ausgewertet. Die Evaluierungen umfassen dabei sowohl die Validierung der aus formalen Modellen des Ansatzes hervorgehenden Verhaltensweisen unter möglichst ungünstigen Randbedingungen (worst-case Szenarien), als auch die detailierte Charakterisierung unter Normalbedingungen. Die durchgeführten Messreihen wurden zusätzlich durch Simulationen unterstützt, um einzelne Betriebsmodi genauer zu analysieren.

Im abschließenden Teil dieser Arbeit werden vorliegende Forschungsergebnisse kurz zusammengefasst. Desweiteren werden Limitierungen des Ansatzes aufgezeigt und mögliche Verbesserungen erwähnt.

Abstract

In the course of this thesis a novel approach for the on-chip generation of a fault-tolerant clock is developed. At first this is motivated by the fact that with shrinking feature sizes and the accompanying increase of transient failure rates it is more and more desirable to provide VLSI (Very Large Scale Integration) circuits that incorporate mechanisms for fault tolerance. In particular, the conducted research concentrates on the most prominent single point of failure of modern chip design, namely, the clock signal of synchronous circuits. After surveying alternative design approaches and existing schemes for achieving fault tolerance a novel fault-tolerant clocking scheme is introduced.

The proposed clock generation method is based on the hardware implementation of a well known distributed clock synchronization algorithm. Most notably, it provides scalable fault tolerance for up to f arbitrary (Byzantine) failures in a system of $n \geq 3f + 2$ tick generation nodes. Additionally, the clocking scheme's operation does not rely on the synchronization of clock sources, like quartz oscillators; in fact, the distributed clock signals are generated in a synchronized way. This unique property relieves the design from metastability issues at clock boundaries.

The transformation of the original software-based algorithm to the peculiarities of chip design proved to be an intricate task. Therefore, the major part of the work deals with the design and development process of the algorithm's hardware equivalent finally resulting in a fully operational VLSI chip design. To assess the properties of the novel fault-tolerant clocking approach and to show its feasibility exhaustive evaluations have been performed. The presented assessments aim at a thorough characterization of (i) the developed chip design and (ii) the distributed clock generation scheme on which these chips are based. Additionally, the conducted measurements allowed to validate worst-case measures which were derived in advance from the formal analysis of the clocking approach. In order to attain a more comprehensive characterization of the design, the presented worst-case evaluations have been supported by measurements and simulations for typical operating scenarios.

The presented work concludes with a short summary and a brief treatment of the most notable topics for ongoing and future research.

Acknowledgements

My personal thanks go to Andreas Steininger, for asking me to join his team and the excellent supervision of this thesis as well as the mentoring during the different phases of work. To Matthias Függer I am especially grateful for the perfect teamwork in the DARTS project[1], the numerous discussions leading to new insights and publications, and last but not least the detailed feedback on an earlier version of this thesis. In the context of the DARTS project I also want to thank Sigrid Heubeck, Gerald Kempf, Manfred Sust, Franz Zangerl and Roman Zangl from RUAG Aerospace Austria for the fruitful cooperation. For the initial project idea and the strength and endurance which finally led to the DARTS project I want to thank Ulrich Schmid. I also want to thank Thomas Handl for conceiving the standard node's test infrastructure. The discussions with my colleagues Josef Widder and Martin Biely — mostly starting in the evening and lasting half the night — have on the one hand been very exhausting, but on the other hand also helped to broaden my horizon. For her professional proofreading I am once again in deep debt to Angela Schörgendorfer.

Sincere thanks to my family, especially my parents and Sandra. Without your help, all of this would not have been possible.

Gottfried Fuchs
Vienna, Austria, August 25, 2009

[1]The work received funding from the FIT-IT program of the Austrian bm:vit (contract 809456-SCK/SAI).

For Anna Fennesz (1922-2006) and Susanna Fuchs (1915-2008)

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Preface

First things first, but not necessarily in that order.

Doctor Who

AS THE title of the thesis already suggests, the presented research follows a multi disciplinary approach. While on one side having VLSI (Very Large Scale Integration) design with electronic circuits and computer chips, the other aspect is given by the area of distributed algorithms and systems. At first sight both topics appear to be completely orthogonal to each other. A closer look at both worlds, however, reveals fundamental similarities, i.e., modern VLSI chips can be seen as microscopically small distributed systems. As a consequence it seems to be quite natural to combine both worlds and for instance use solutions from the distributed systems community to solve current and upcoming problems of VLSI design. High level approaches from distributed systems might be able to alleviate some of the challenging issues and burdens when dealing with modern VLSI circuits. As an example, coping with pronounced parameter variations due to difficulties related to the manufacturing process is a burning issue. Additionally, the increased susceptibility to radiation-induced soft-errors enabled by shrinking transistor feature sizes is also considered a major VLSI design problem [47]. These adverse effects of circuit miniaturization, together with concepts to overcome some of the involved challenges are used as foundation for the problem definition of the thesis. More specifically, the main effort of the conducted research is directed towards the important field of digital circuit clocking. In particular, reliability under variations in manufacturing and operation as well as faults is taken into account, thus leading to the core topic of the work, namely, fault-tolerant clocking (FT-clocking).

1.1 Motivation

Advancement in digital electronics is stated to be the most prominent driving force behind technological progress. Electronic devices are omnipresent and almost any innovation involves computer chips in one way or another. In everyday life we are surrounded by and accustomed to electronic devices like cell phones, personal computers, satellite technology, medical imaging, etc. The key enabler for the fact that computer chips are omnipresent is given by the ever increasing processing power and speed of modern digital circuits and the accompanied possibilities for new applications. Over the past 30+ years the increase of digital circuit complexity as well as their performance gain followed *Moore's Law* [65] quite well, which essentially predicts that the amount of transistors on a chip double about every two years. The miniaturization of transistors which allows Moore's law to persist not only enables the above mentioned performance increase, but also introduces adverse effects. Transistor feature sizes of recent and future computer chips are approaching the dimension of a few dozens of atom diameters. As shown in the recent version of International Roadmap for Semiconductors (ITRS07) [47] it is increasingly difficult to manufacture these extremely small structures in a reliable way. Additionally, process variations in the manufacturing process may lead to severe fluctuations of the processing speed of a chip's components [9]. Furthermore, the diminishingly small electrical charges present in modern transistors are susceptible to different sources of disturbance, e.g., electrical discharge induced by particle hits [5,75,94], crosstalk or electromagnetic interference [60] might yield multiple faults. To maintain reliable computing technological and/or architectural measures have to be taken.

An issue with current circuit design related to the problems described above arises from the fact that the greater part of modern digital circuits follow the synchronous design paradigm. Proper operation of a circuit following this design style utterly relies on a single source — the clock signal. The globally synchronous design with its one large isochronous region[1] is hard to maintain as clock periods are less than the time a signal needs to traverse the whole chip. The distribution of the clock signal from a single source (typically a quartz/crystal oscillator) all over a chip has become an art of its own given the die size of a modern VLSI chip with millions of transistors and its operating frequency in the GHz-range. Tremendous design effort has to be made to distribute the clock signal in such a way that it arrives with minimal skew at every component to guarantee the correct behavior of the synchronous circuit. Facilitated approaches use H-tree and X-tree clock distribution networks combined with mesh and grid interconnects. Additionally, large numbers of clock buffers and de-skewing units have to be placed at carefully chosen positions to guarantee minimal skew between any two clock sinks [31, 70, 74]. An issue arising with these large efforts necessary to achieve proper clock distribution is power consumption. Modern VLSI chips' clock distribution adds up to 25 to 50% of the overall power budget of a chip [21,68]. Referring to the design challenges presented in the previous paragraph the question arises whether or not the apparently strong, but critical clock signal suffers from any of the

[1] "The maximum distance that a switching signal can travel across a region, in which the time of flight does not limit the signal propagation, circumscribes a region known as the isochronous region." [61]

adverse effects of technology scaling mentioned above. Despite the fact that the clock tree is driven by strong buffers, in [76] radiation-induced errors have been reported for the clock distribution network of classic synchronous chips using recent semiconductor technology. Moreover, the mechanical properties of clock sources like quartz oscillators are severely limiting system reliability, since sensitivity to vibration, temperature and shock as well as problematic cold start-up behavior have to be noted. One could argue the globally synchronous design style with its single clock source should be replaced by a more promising alternative. In light of the current technology trends global clock distribution with reasonable synchrony is hard to attain, and comes at the price of great effort to properly design the sophisticated clock distribution with its power consuming buffers, grids, meshes and other more complex de-skewing circuits [31, 71].

To give a sound introduction to the research topic and the involved application fields of this thesis the reader is successively introduced to the underlying principles and technologies. In this context the concepts of logic design in general, fault tolerance, as well as distributed systems and algorithms are established to provide common terms for the rest of the thesis. Moreover, design alternatives as possible remedies for the hard-pushed synchronous approach will be surveyed.

1.2 Design principles

Electronic circuits in general comprise a multitude of (different) components to implement a desired functionality. Signal processing in such components follows a scheme similar to the one shown in Figure 1.1. A producer/source block generates a data item x that is further processed by a functional unit $f()$. The end of such a generic processing stage is given by a consumer/sink, operating on the results $f(x)$ of the previous blocks.

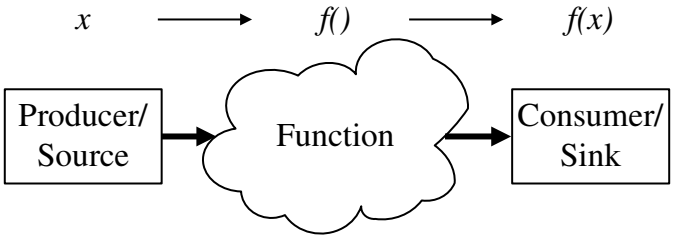


Figure 1.1: Generic design block

Typical design components for producer/source and consumer/sink blocks are given by storage elements (registers), whereas the function $f()$ may comprise simple logic gates like AND-, OR-gate, inverter, register, and the like. The processing of these design units has to be controlled in a way such that old and new data cannot interfere with each other in an unintended way. Hence, in general, data is first read from a producer/source-register, afterwards processed by some logic elements and in the end stored in a consumer/sink-registers as depicted in Figure 1.2. However, what is missing in Figure 1.2 as well as

Figure 1.1 is the mechanism which ensures the above demanded non-interference of different data waves. In other words, the issuing of data items and their consumption has to be coordinated somehow.

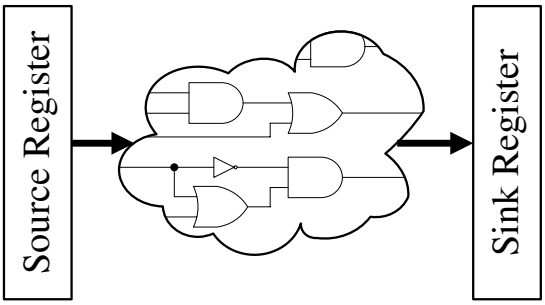


Figure 1.2: Logic design block

This fundamental kind of flow-control/data synchronization can be achieved by different means, the approaches typically being grouped into design styles following the synchronous or asynchronous design paradigm. It further has to be noted that the better part of currently available computer chips (processors, micro-controllers, FPGAs, etc.) are built to work according to the synchronous design paradigm. However, asynchronous design has its niche application fields, e.g., ultra low power circuits [56] and seems to provide superior properties when dealing with large parameter variations of current and future deep sub-micron and nanoscale devices [9, 10]. Furthermore, specialized designs, for instance high-speed random access memory (RAM) [17], field programmable gate arrays (FPGAs) operating at GHz speed [84] as well as terabit network routers [86], may also rely on asynchronous design styles to achieve elevated performance goals.

1.2.1 Synchronous design

A synchronous design is characterized by the centralized mechanism controlling the instants when data is stored into registers, i.e., the combinatorial logic has finished its computations

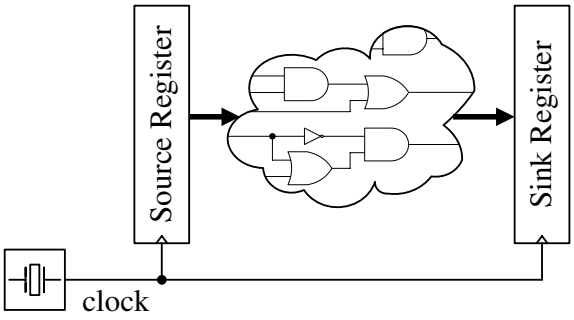


Figure 1.3: Synchronous logic design block

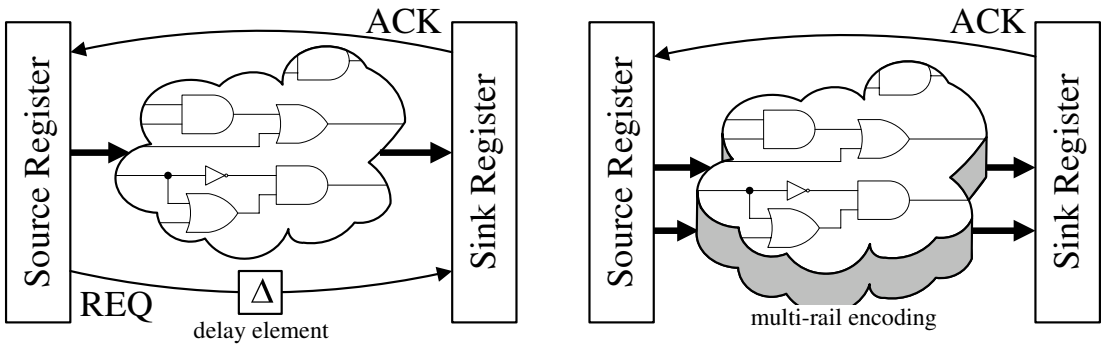


Figure 1.4: Bounded delay (BD) and delay insensitive (DI) asynchronous approach

and is ready for new data. As shown in Figure 1.3 each register is controlled by the same clock signal which originates from one single clock source (e.g., a quartz oscillator) and provides a global time base. All registers simultaneously store the currently available data as soon as a clock transition (for instance low-to-high) occurs at the registers' clock inputs. For proper operation of the synchronous approach it is of utmost importance that clock transitions are received and processed nearly at the same time at all registers (with only small timing margins). To ensure this global isochronous behavior clock distribution with minimal skew is the most critical and challenging part of synchronous circuit design [31].

1.2.2 Asynchronous design

In contrast to the global/centralized time base of the synchronous design approach, asynchronous methodologies rely on local handshakes between any source/sink pair. This handshake is partitioned into two parts, (i) the request (REQ) generated by the source indicating the sink that new data is available and (ii) the acknowledge (ACK) indicating that the sink has already stored the data and is ready to process new one. Unlike the synchronous approach, in asynchronous circuit design there is no single generally adopted methodology, but there are rather a few different design styles to solve the fundamental data synchronization problem.

By picking two popular asynchronous design methodologies the manifoldness of asynchronous data processing is illustrated. The bounded delay (BD) approach [80] depicted in Figure 1.4 (left panel) and the delay insensitive (DI) approach [26] shown in Figure 1.4 (right panel) are the chosen representatives. Both the bounded delay and the delay insensitive approach have in common that the generation of the ACK signal is typically triggered together with the storage of data into the sink register. The difference of the BD and DI methodology is in the way the REQ signal — indicating new valid data — is generated. In BD methodology the data source issues an explicit REQ signal together with the data. Correct circuit behavior is achieved by the insertion of individual matched delay elements into the signal-path of every REQ signal, cf. Figure 1.4. The delay elements have to be configured in a way that ensures that data — after its actual processing via some combina-

torial logic — is already valid when REQ arrives. In contrast to BD, the delay insensitive approach has no explicit request signal. The information that valid data is available is directly encoded into the data itself — there is no need to know timing parameters of the design or to insert matching delays[2]. To be able to convey this validity information to the sink some sort of extended signal encoding has to be used. For example in Four State Logic (FSL) [63] or Null Convention Logic (NCL) [26] two signal rails are used to be able to sufficiently encode every data bit. In essence there must be a coding for invalid data in addition to HI and LO. Independent of timings and delays a completion detection unit at the sink decodes the dual-rail data signal, extracts the validity information and decides accordingly if the data is valid and can be consumed.

A more thorough introduction to asynchronous design principles and methodologies can be found in Hauck [43] as well as the textbooks by Myers [67] and Sparsø/Furber [80].

1.3 Related design approaches

1.3.1 Globally asynchronous, locally synchronous

The Globally Asynchronous Locally Synchronous (GALS) approach provides an alternative alleviating the stringent clock distribution issues of the purely synchronous design style [12]. Systems following the GALS design style are based on the generic architecture depicted in Figure 1.5. Small (local) synchronous islands implement functions (sub-tasks) of the whole system. Each local island's function is executed using the traditional synchronous design style, whereas global interaction follows an asynchronous communication style. Each island is provided with its own oscillator as clock source for the locally synchronous computations. Compared to the high effort for global clocking of a purely synchronous system in local synchronous islands skew optimization of the clock signal is much easier to attain. Although GALS simplifies the clock distribution to some extent, some other issues are still left. The need for a particular oscillator for each synchronous island adds additional mechanical components (quartz oscillators) to a system which clearly decreases reliability of the whole design. As an alternative to the error-prone quartz oscillators (which are sensitive to, e.g., vibration, temperature, shock, etc.) on-chip RC-oscillators could be used. However, RC-oscillators are known for their strong dependence on operating conditions like temperature and supply voltage which leads to frequency changes in the range of 10 to 30%. There are several issues in the context of the local clocking of GALS systems. On the one hand the extremely unstable clock signal of RC-oscillators might be a problem for many applications. On the other hand, there are mechanical issues with quartz-oscillators which are known to be problematic in harsh environments like the aero-space domain. In general, if compared to a synchronous system the GALS concept has two major disadvantages. First of all, a

[2]To be more precise, strict DI circuits offer only very limited functionality. Therefore, implementations typically follow the quasi delay insensitive (QDI) approach which, however, introduces a constraint for forking paths [58].

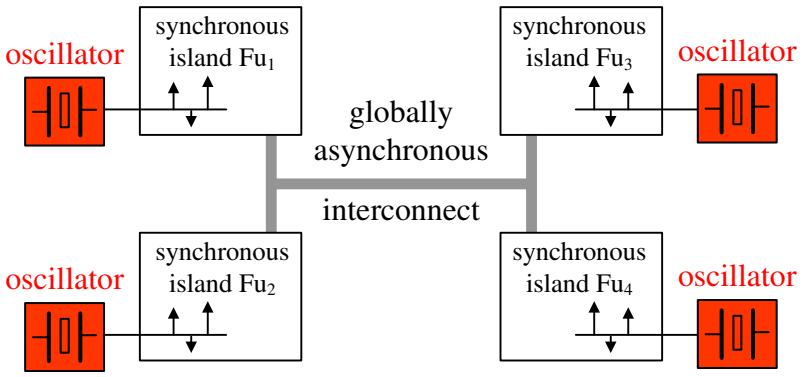


Figure 1.5: Globally asynchronous locally synchronous (GALS) architecture

GALS design does not implicitly provide the convenient common notion of time within the system which most hardware designers are used to and design tools are made for. All clock sources are free running — no synchronization precision among the clocks can be assumed as they may drift arbitrarily apart from each other. As long as communication only affects a local island there is no problem and everything stays the same way as in a synchronous design. However, with communication leaving the bounds of a local island's clock domain and asynchronously traversing the chip to another island introduces the need for synchronization. The fact that the interface between globally asynchronous communication and locally synchronous data processing has to incorporate some sort of synchronizer circuits poses the second, probably the most severe disadvantage of GALS. Unfortunately, synchronizing clock domains with arbitrary, possibly changing, relation to each other, cannot be solved in a safe way. Metastability issues might even upset the synchronizer circuits [41] and can only be made more unlikely by adding synchronizer stages. Taking parameter variations and clock jitter into account synchronizers have to be designed very conservatively, thus introducing significant performance penalties into the asynchronous/synchronous interfaces.

Recent synchronizer implementations incorporate sophisticated designs using stoppable (pausable) and/or stretchable clocks [22, 66] to reduce performance loss due to synchronization.

1.3.2 Interconnected rings and oscillators

This concept proposed by Maza and Aranda in [61, 62] also addresses the difficulties of synchronous clock distribution in GHz designs and presents an alternative approach for generating and distributing clocks. The design relies on the self-oscillation property when interconnecting an odd number of inverters in a ring topology (shown in Figure 1.6) and achieves high clock frequencies due to the simplicity of the design. Inverter and buffer placement of the proposed architecture determines wiring costs (in terms of wire length), speed and skew of the generated clocks. The design especially fits as on-chip clocking

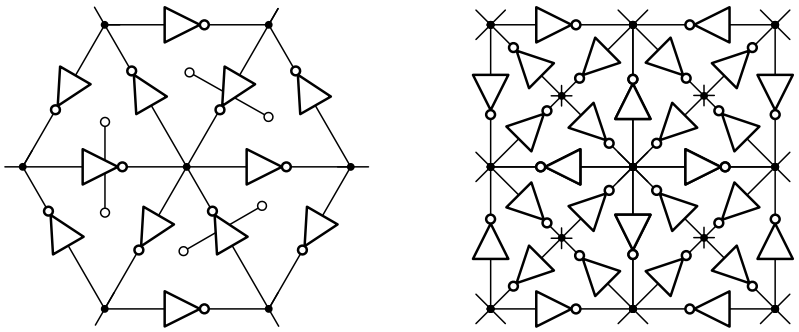


Figure 1.6: Interconnected ring oscillator architectures

scheme for the previously introduced GALS systems. It can be seen as a refinement of the GALS, RC-oscillator clocking. Due to the fact that all inverters of the clock generation scheme are interconnected directly (locally) or indirectly (globally, through some additional inverter stages) with each other, the local islands of a GALS system cannot arbitrarily desynchronize (at least in the fault-free case). This property severely eases synchronization within the GALS design since the synchronizers can take advantage of the fact that the local clocks are not entirely unrelated.

1.3.3 The distributed clock generator

The scheme of a distributed clock generator (DCG) introduced by Fairbanks and Moore [24, 25] represents a special form of asynchronous FIFO implementation for the purpose of on-chip generation and distribution of a synchronized clock. Similarly to the approach by Maza and Aranda, interconnected clock generation hardware is distributed in a grid all over the chip, but the locally generated clocks are generated at approximately the same instant having only small skew. Every DCG instance is interconnected with its four neighbors and half of the DCG units are initialized with a clock token. Due to the asynchronous FIFO implementation of each DCG the so-called Charlie effect [25] ensures that clock tokens are passed over to neighboring nodes in a synchronous way, generating a chip wide synchronized clock signal (the Charlie effect describes the force that slows down a subsequent token within a FIFO if it is closing in on a previous one).

1.3.4 Purely asynchronous design

Asynchronous design styles [43] are considered a viable alternative for synchronous design in the future. This is at least true for special application fields like low power circuits [56]. Moreover, some recent work [17, 86] shows the general applicability in high-speed design. With asynchronous design the burden of clock distribution can be entirely eliminated and the clock tree be substituted by far less timing critical local handshake signals. Parameter variations are much less problematic in the context of, e.g., quasi delay-insensitive

circuits [59] since only performance but not the correct function is influenced by variations. Furthermore, the inherent robustness of asynchronous design styles allows to address the issue of increased failure rates in future VLSI technology [32, 48] to some extent. On the downside of asynchronous design the manifoldness of design styles (bounded delay, quasi delay-insensitive, scalable delay-insensitive, etc.) prevent a generally accepted methodology. The huge diversity in asynchronous design styles additionally leads to lacking design and verification tools which further handicaps the wide-spread usage. Moreover, non-negligible area overhead, higher design complexity and the more intricate circuit testing also have to be mentioned. Despite the fact that a certain robustness is inherent to asynchronous designs, established techniques, like the majority voting of triple modular redundancy schemes (TMR), for implementing fault tolerance cannot be directly applied to asynchronous systems. This limitation is based on the fact that sometimes it might not be clear if data to be voted is faulty, e.g., missing due to a crashed sender, or if it simply is late [18].

1.3.5 Discussion

In summary, none of the above described alternatives to globally synchronous clocking sufficiently addresses the reliability issues of multiple transient and permanent faults that are upcoming with current and future technology scaling. In the presented methodologies the incorporation of mechanisms for fault tolerance and a special focus on robustness to cope with faults as well as parameter variations is mostly lacking. GALS in general has issues with interfacing multiple uncorrelated clock domains (synchronizer problem), furthermore, the lacking global time severely complicates the design process (which is also the case for purely asynchronous approaches). The interconnected rings and oscillators as well as the distributed clock generator approach are not able to cope with failures. To be able to tolerate arbitrary failures in a clock synchronization process theory shows that almost fully connected networks are needed [23] which is clearly not fulfilled by these two approaches. Therefore, a transient fault might lead to major clock deviation, over-clocking phenomena or could even stop the whole clock generation process. Table 1.1 summarizes the characteristics of the discussed design methodologies.

In the face of the challenges introduced at the beginning of this section and trends in semiconductor technology which lead to more powerful but also potentially less robust designs, the work described in this thesis focuses on the development of a robust clocking scheme for future dependable systems. Therefore an approach is employed which provides local synchrony similar to GALS. Compared to GALS the distinctive feature given by the enhancement that a globally distributed fault-tolerant time base (with slightly relaxed synchrony assumptions) is maintained also. The ongoing miniaturization and the speed-up of digital circuits in conjunction with accompanied adverse effects of technology scaling, suggest to incorporate mechanisms for fault tolerance to attain reliable systems. Especially in — but not limited to — safety- and mission-critical environments like in the automotive and aerospace domain robustness against faults is of utmost importance.

Table 1.1: Comparison of design methodologies

	variation robustness	fault tolerance	design/area overhead	established design method
purely synchronous	no	no	none	yes
GALS	partially	no	small (synchronizer)	somewhat
interconnected rings	partially	no	small (inverter chains)	no
distributed clock generator	partially	partially	small (DCG units)	no
purely asynchronous	yes	no	large (multi-rail coding and REQ/ACK scheme)	somewhat

1.4 Fault tolerance

Technological advancements, especially in the domain of computer design, introduce increasingly complex systems into everyday life. Cell phones, personal computers, cars, airplanes, (nuclear) power plants or even microwave ovens, for instance, are equipped with at least one, but more often a set of interacting electronic processing unit(s) executing comprehensive program code. As systems get more and more complex, the chance of failure of a single component — affecting the proper operation of the whole system — increases steadily. A failing cell phone would at least be annoying but usually not dramatic. In contrast, if the brakes of a car or the controls of an airplane fail due to a glitch in the control system lives are at stake. Therefore, fault tolerance is an important design aspect, at least in modern safety-critical systems. Further, it has to be noted that in the face of increasing failure rates of modern VLSI technology even non-critical, e.g., multimedia devices, will soon be augmented by fault tolerance mechanisms to allow for proper operation with reasonable failure rates [15, 77].

In the previous paragraph two important terms, *failure* and *fault* have been used without clear distinction. To treat these important items in more detail a short definition for fault, *error* and failure will be given below. The definitions (following [3]) will be used throughout the rest of this thesis.

Definitions of failures, errors, faults: If a system fails, its output behavior deviates from the specified characteristics — a *failure* can be observed at the interface. The cause of a failure can usually be traced back to an erroneous state within the system itself which led to the incorrect output. An *error* can therefore be defined as an unintended/corrupted internal state of the considered system. The cause for an error

is called *fault* which may indirectly lead to a failure if it is not masked in some way. The origin of faults can be manifold ranging from physical effects, e.g., broken wires, EMI, etc., over handling faults to design faults.

Many different definitions for fault tolerance and fault-tolerant computing can be found in literature, but the essential part is similar for most of them [3, 50, 78]. A system can be considered fault tolerant if it continues proper operation even after a fault has occurred — the fault does not lead to a failure. In other words, a fault that would affect the system's interface behavior is handled before it can propagate to the output.

Fault tolerance usually relies on error detection and some form of redundancy to cope with the erroneous data — via replicated components and/or repeated computation. An example for an approach using hardware replication is given by the widely used Triple Modular Redundancy (TMR) scheme. In TMR processing units are triplicated and the outputs are compared via a special voter circuit which is able to mask the failure of one component if both other replicas operate properly. Time diversity is also a method for achieving fault tolerance, e.g., repeated processing of data on the same hardware can mask short noise effects (at the price of decreased overall throughput) as long as no permanent faults are present. An important point regarding fault tolerance which has been neglected so far is concerned with the fault hypothesis. The fault hypothesis specifies assumptions about number and types of faults that a fault-tolerant system must be able to tolerate [50]. These assumptions have major impact on the fault tolerance concept of the system. Therefore, the fault hypothesis has to be thoroughly defined during the design phase, because faults not covered by the assumptions, but which may occur in real world scenarios, may bring the whole system fail.

Assumptions on the nature and type of failing components in the context of VLSI hardware design are usually classified using several different types of fault models.

1.4.1 Fault models in VLSI design

Fault models typically applied in VLSI consider faults on the abstraction level of single gates or transistors. Where a set of interconnected gates forms a circuit. If a gate or interconnect stops to work in its intended way a fault has occurred. It has to be noted that in the context of this fault classification only static fault scenarios are considered, i.e., once a component is affected by a fault, it will stay faulty. The most commonly treated faults are introduced below.

stuck-at fault There are two types of *stuck-at faults*, the stuck-at-0/LO and stuck-at-1/HI. A stuck-at-X fault occurring at time t_F at signal S manifests itself in a way that S takes on logic level X at time t_F and is no longer able to change its value after t_F. Note that, if for example signal S is currently LO and a stuck-at-HI occurs, the fault will generate one last erroneous transition to HI and remain there.

stuck-open fault A stuck-open fault appearing at t_F disconnects the affected signal S from its driving buffer which leads to an undefined voltage level, the signal is "floating". This floating state may lead to inconsistent perception of the logic level when read by multiple inputs.

delay fault A *delay fault* increases or decreases the time a signal change needs to propagate through the respective signal or gate. The altered timing behavior of the affected component may lead to the violation of timing constraints in subsequent circuits.

bit-flip fault A *bit-flip fault* changes the logic level of a component to the opposite value. This type of fault is usually treated with state-holding devices only, i.e., sequential logic. A bit-flip fault occurring at time t_F at a single bit storage component C manifests itself in a way that component C takes on the inverted state of C prior time t_F.

1.5 Distributed systems and algorithms

Distributed systems are typically defined as a set of autonomous computer/computing systems — often called nodes — that communicate and cooperate in some way with each other [2, 85]. This definition includes wide-area networks (e.g., the internet), local-area networks, multiprocessor computers as well as single VLSI chips. In general, this wide range of very different distributed systems can be subdivided and classified by the node's grade of coupling. For instance, a multiprocessor system is coupled much tighter than the internet. Another classification usually considers the purpose for building a system in a distributed way. Here the reasons are manifold, spanning from simple resource sharing over aiming at increased performance to the improvement of system reliability. As already sketched at the beginning of this chapter, as well as in Section 1.4 on fault tolerance, the *improvement of system reliability* is the main driving force for considerations within this thesis.

If a stand-alone component fails, the whole computing task is affected. By replication of computing nodes the overall reliability of a distributed system can be increased compared to a stand-alone component. This enhanced resilience against failures is usually provided by replicas performing the same tasks as the failed component augmented by some kind of voting mechanism(s) to mask faulty outputs, hence compensating for failed nodes. In contrast to the straight forward way of a single (stand-alone) component which performs a computation task, this distributed processing of and voting on data has to be enabled by incorporating elaborate algorithms coordinating all actions.

Algorithms operating in a distributed system have to cope with at least three main difficulties if compared to traditional centralized algorithms. In centralized systems algorithms usually have access to the *global state* of the system. This cannot be assumed for distributed algorithms, which only have direct access to their own local state. Even though state information can be exchanged between nodes, the state of other units may

have already changed and therefore be invalid when the information arrives. The second issue with distributed algorithms is related to *synchronization* or *global time*. Temporal ordering of computations is provided naturally in a centralized system by their sequential execution of tasks. In contrast, total ordering of events in distributed systems is sometimes not possible, e.g., it cannot be determined which of two events occurred first, given timing uncertainties incorporated with sending and receiving of data in a distributed system [51]. Another difficulty when dealing with distributed systems is given by *non-determinism* of executions (e.g., due to failures). A centralized processing node's operation sequence and global state can usually be determined exactly. In contrast, a node of a distributed system may process the same set of input data in a different way, for example by starting at a different time depending on the current state of all other nodes.

1.5.1 Modeling distributed systems

When taking a closer look at the modeling of distributed systems, different computation models can be identified. The main distinctive features are specified by the way nodes communicate with each other and whether or not and what kind of timing assumptions (e.g., synchronous, partially synchronous, asynchronous, etc.) are considered. Another dimension for distinction is given by the considered failures modes. However, the distributed system models presented in the following paragraphs are at first introduced for systems free of faults. Nevertheless, these models can be augmented for the faulty case by adapting them for certain failure models.

A widely-used system model operates on *message-passing* [2]. In message-passing systems, n nodes $p_0, \ldots, p_{n-1}$ communicate with each other by sending messages over communication channels. Typically a communication channel is modeled as bidirectional link between two nodes and the specific interconnections of nodes define the topology of the system. An algorithm in a message-passing system is specified by its program code executed on the system's set of n nodes, whereas each node can be modeled as state machine with a set of states Q_i. A node may perform actions like receiving from and sending messages to neighboring nodes as well as executing local computations. Each node contains as many input and output buffers as it has incoming connections from and outgoing links to other nodes, respectively. The outgoing buffers store messages already sent by the node that have not yet been delivered to the receiving node, similarly input buffers hold messages until they are processed. Transitioning from one state of a node to the next involves the processing of all messages stored in the input buffers, performing computation events, updating the local state and sending at most one message to neighboring nodes (by placing the message(s) into the respective output buffers). The behavior of a message-passing system over time is modeled as an execution, with an execution defined as an alternating sequence of events (computation and sending) and configurations, where configurations C are given by the states of the nodes $C = (q_0, q_1, \ldots, q_{n-1})$ with q_i being the state of p_i. What is usually demanded from a correct distributed algorithm is that some conditions are valid for its execution. The *safety condition* has to hold for any reachable sequence

of configurations and events, and states that "nothing bad ever happens". Additionally, a *liveness condition* has to hold at least several times and can be translated to, "eventually something good happens". To illustrate these two conditions consider a street crossing with two traffic lights as an example. A safety condition has to ensure that both traffic lights will **never** show green at the same time, while a possible liveness condition states that the a traffic light eventually changes from red to green. So far no assumptions on the timing of messages have been made. However, as initially stated, asynchronous, synchronous and some systems between can be distinguished. Asynchronous message-passing systems can be characterized by the fact that there is no fixed upper bound on message delivery and computation time. Therefore, algorithms designed for this kind of model have to operate without knowledge of such bounds. In contrast to the asynchronous model, a synchronous system works in a lock-step manner, that is, executions are partitioned into rounds. Relying on this round scheme, depicted in Figure 1.7, where every node can send one message to each neighbor and may perform a computing step on received messages, gives a convenient model with only little uncertainty. A problem that in the context of an asynchronous model is initially quite intricate may be substantially simplified by mapping it to synchronous lock-step rounds. However, from this the challenge of establishing this synchronous round scheme emerges. Unfortunately, for most distributed systems in practice a truly synchronous system cannot be achieved. On the other hand some distributed computing problems might not be solvable in a purely asynchronous framework, e.g., distributed consensus in the presence of faults [30].

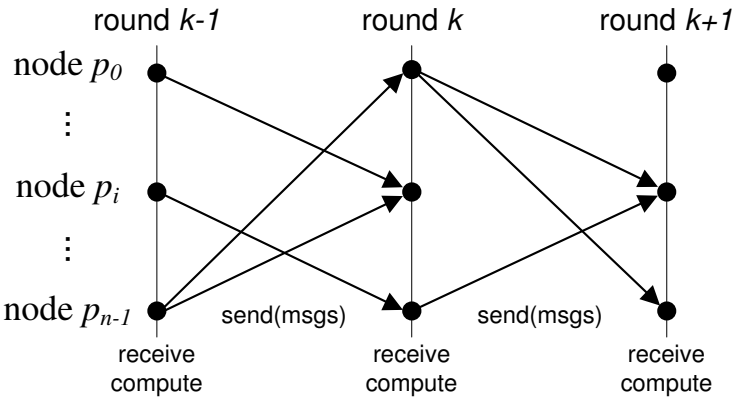


Figure 1.7: Execution of a synchronous message-passing algorithm

Shared memory [2] based distributed systems rely on communication via a set of shared variables. Various types of variables can usually be distinguished, where a characterization can be given by the values returned from and the operations allowed on these variables. Typically applied types of shared variables allow operations like read and write, however, several more complex functions can be employed. Similarly to the message-passing, systems of n nodes $p_0, \ldots, p_{n-1}$ are treated as state machines with executions represented by alternating sequence of configurations and events. Input and output buffers are not needed

for shared memory systems, however, m registers $R_0, \ldots, R_{m-1}$ representing shared variables are employed. Typical operations on registers are for instance reading a value v from register R, returning v and leaving R unchanged, or writing a new value v to R, whereas this operation does not return a value. In contrast to the message-passing approach, the configuration C in a shared memory system is defined by $C = (q_0, \ldots, q_{n-1}, r_0, \ldots, r_{m-1})$ with q_i again denoting node i's state and r_j being the value of register j. The events within an execution are computations which follow a specific scheme starting with node p_i issuing an operation on a variable determined by p_i's actual state q_i. Afterwards, p_i's state changes according to p_i's current state and the return value of the shared memory access. Analogous to the message-passing approach, executions of a correct algorithm have to fulfill safety and liveness conditions.

1.5.2 Distributed systems failure models

To augment the modeling techniques presented above by the notion and effects of faulty behavior various established component failure models may be applied. The most commonly used component failure models for distributed systems research are introduced below.

clean crash failures A node p is said to have cleanly crashed at time t if all messages sent by p before time t are correctly received by all other non-faulty nodes. Furthermore, all messages from node p sent at time t or later are not received by any node.

crash failures These type of failures are also called unclean crash. Crashes are characterized by scenarios where node p is said to have crashed at time t if all messages sent by p before time t are correctly received by all other non-faulty nodes. Furthermore, a message from node p sent exactly at time t is received only by a subset of the distributed system's nodes and messages sent by p later than t are not received by any node.

omission failures A node p is called omissive at time t, if messages sent by this node later than t are only received by a subset of the remaining non-faulty nodes of the distributed system. Hence, the omission failure model allows that messages are perceived asymmetrically for an unbounded number of times.

Byzantine failures represents an unrestricted failure type. Therefore, a node p affected by a Byzantine failure at time t may show arbitrary malicious behavior. This behavior mainly extends the crash and omission failure models by the threat of artificially generated additional and/or inconsistent faulty messages.

The failure models presented so-far are within the typically applied group of static failure models, that is, a component once faulty will stay faulty forever. However, sometimes a more detailed failure analysis of real-world scenarios is necessary, treating temporary fault effects as well as repair & recovery strategies. The formal analysis algorithms in conjunction with dynamic failure models, however, is much more complex than the treatment of the (classical) static case [1, 4, 7].

1.6 Fault-tolerant clocking

In the synchronous design paradigm introduced in Section 1.2.1 the global time is provided by the clock signal which originates from a single source. This clearly forms a single point of failure. To maintain the convenient because well established synchronous design paradigm, but also to get rid of its single point of failure, some kind of fault-tolerant clock has to be supplied — for example derived from multiple clock sources. Taking a look at processing nodes of distributed systems reveals that they usually have no access to a central clock. However, some common notion of time is often crucial for, or at least eases solving several types of distributed problems. Similar to the synchronous logic design, multiple clock sources, for instance one at each node, augmented by clock synchronization techniques can be used to obtain a globally synchronized clock (among all non-faulty nodes). Admittedly, synchronizing autonomous local clocks in the presence of faults and varying operation conditions can be quite problematic and hast at least some constraints and restrictions due to changing and/or unknown interconnection delays which have to be dealt with. Nevertheless, having the local clocks synchronized to each other improves the resilience of the whole distributed system against faulty processes (synchronous algorithms operate in rounds, hence a fault of, e.g., a crashed process can immediately be detected when an expected message is missing in the round schedule).

A closer look on the oscillators which actually form the local clock sources reveals two types of unintended effects:

clock shift denotes the effect of a certain offset between clocks despite the fact that the clocks might run at exactly the same frequency

clock drift refers to the effect that a clock does not operate at its designated speed (it might be slightly off the nominal frequency if compared to a reference clock)

For example, the adverse impact of parameter variations in crystal oscillators might lead to drifting clock frequencies. The typical drift rate — that is the deviation from a perfect reference clock — of e.g., a quartz oscillator, is in the range of 10^{-6} (one second offset in a period of about 277 hours). Whenever clock shift or drift is observed it leads to inhomogeneous advancement of the clock among a set of oscillators and therefore to different progress compared to real time, cf. Figure 1.8. The fact that physical clocks drift apart, and additionally the possibility for clock sources being faulty, make it evident that even an initially perfectly synchronized system will deteriorate after some time if no countermeasures are provided.

Clock synchronization schemes counteract the adverse effects of shift and drift in spite of faults by applying different strategies. The synchronization algorithms usually employed can coarsely be classified by the way they derive the local adjustments for the involved clocks. Within the class of deterministic approaches many of the convergence algorithms follow, for instance, a fault-tolerant averaging strategy over clock values, while consistency

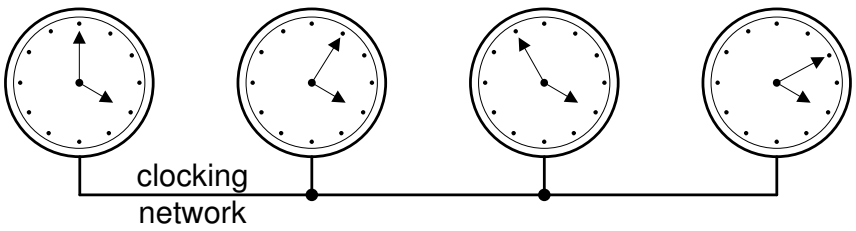


Figure 1.8: Example for de-synchronized clocks

schemes use a distributed agreement approach (more details on the clock adjustment strategies will be presented in Section 2.2). A common part for both approaches is the fact that clock adjustments are performed periodically following so-called resynchronization intervals. In contrast to the mentioned deterministic (convergence and consistency) schemes, there are also probabilistic algorithms for clock synchronization which rely on knowledge about delay distributions.

A more thorough examination of the vast variety of fault-tolerant clock synchronization approaches, ranging from pure software over hybrid to pure hardware schemes, analyzing their properties and special features will follow in Chapter 2.

1.7 Problem definition and contribution

Existing clocking approaches do not, or do not sufficiently address fault tolerance and robustness issues as they seem to be necessary for modern chip design. Hence, the mainline of the conducted research is to *design and implement a distributed fault-tolerant hardware clocking scheme without single point of failure*. Furthermore, evaluations and measurements to characterize the prototype ASIC and relating this data to properties derived in advance should be used to validate the proof-of-concept implementation of the fault-tolerant clocking scheme. In addition, a set of general conditions and prerequisites have to be taken into account for the development process following the research topic.

- The target technology is given by an 180nm ASIC fabrication process which is particularly suited for radiation hardened design.
- A single instance of a distributed clock generation unit is implemented per ASIC to keep the die size and thus manufacturing costs as low as possible[3].
- The ASICs have to be configurable for different numbers of active nodes in the clock generation system.

[3]The whole distributed system comprising several nodes could have been implemented on a single die, but the improved accessibility of a single node per chip better facilitated the prototype design.

- Improved access to internals of the design have to be provided to facilitate detailed evaluations and measurements.
- The design has to follow a fully asynchronous implementation since neither external clock sources (e.g., quartz) nor internal (on-chip RC-) oscillators should be used due to their already stated problems and limitations.

The main contribution of the research conducted in the course of this thesis is closely related to the research project *Distributed Algorithms for Robust Tick-Synchronization (DARTS)*[4] in whose context a novel fault-tolerant clocking scheme has been designed, formally proven, simulated, implemented and evaluated. The described approach, depicted in Figure 1.9, of a hardware implemented distributed algorithm as fault-tolerant alternative to common global clocking represents the foundation of the proposed approach. The focus of the work in this thesis is mainly concerned with hardware design issues and treats the development and the evaluation of the clocking approach. In particular, the mapping of software-based algorithms to an appropriate asynchronous hardware implementation presents a major part of the contribution. Furthermore, the assessment of the resulting tick generation scheme and the validation of formally derived performance and correctness measures represent other important parts of the conducted work.

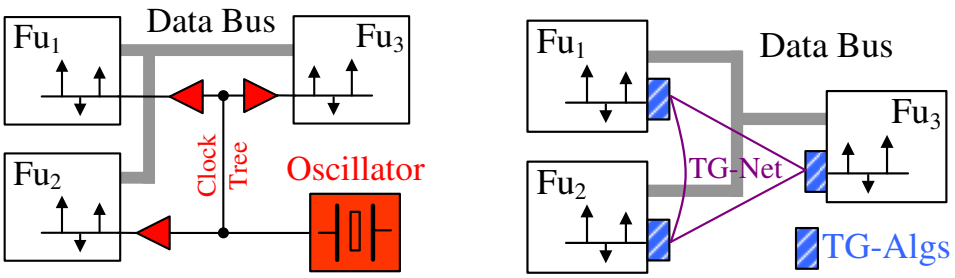


Figure 1.9: Replacing synchronous clocking by fault-tolerant distributed tick generation

As Figure 1.9 suggests, the elaborate global clock-tree of the purely synchronous approach with its single (external) oscillator as well as the strong and power consuming buffers is replaced by dedicated tick generation units. In more detail, a tick generation network (*TG-Net*) provides a less critical as well as less power-hungry replacement for the synchronous clock rails. Reduced power consumption is expected as no tight synchrony has to be achieved — there is no need for strong buffers, while enhanced robustness is provided due to the fact that single rails of the TG-Net might fail arbitrarily. Tick generation hardware modules (*TG-Alg*) attached to local functional units (Fu_i) and interconnected via the TG-Net implement a clock/tick generation algorithm. This distributed algorithm for generating a mutually synchronized clock that is tolerant to multiple Byzantine faulty

[4]The project DARTS received funding from the Austrian bm:vit (FIT-IT, contract no. 809456-SCK/SAI). *DARTS web-page*: `http://ti.tuwien.ac.at/ecs/research/projects/darts/`

nodes and links has been implemented in asynchronous hardware and fabricated using a recent $0.18\mu m$ ASIC manufacturing process which is radiation-hardened by design. Furthermore, the algorithm has been validated and characterized by extensive simulations and measurements of single nodes as well as the whole ensemble of eight nodes. Each node of the DARTS clocking scheme is implemented by a single chip holding an instance of the (same) distributed clock synchronization (in fact, clock generation) algorithm. The tick-generation ASICs have been designed to be operated in a configurable cluster of up to eleven nodes forming a synchronized clock resilient against up to three Byzantine-faulty components. Note that, as already mentioned above, in the scope of this thesis a cluster of eight nodes has been utilized for the experiments and measurements — a design of eight nodes is able to tolerate up to two Byzantine faults which is sufficient, less costly and more cost and hardware efficient to provide the targeted proof of concept of a multiple Byzantine fault-tolerant clocking scheme. Investigations as to which extent failure models other than the Byzantine model are appropriate when dealing with hardware faults are also covered within this work. Especially savings in terms of design costs — particularly the gain in chip area and lower system complexity — are addressed in this context. Another valuable insight gained while investigating robustness and stability characteristics of the DARTS clocks, namely conditions and properties for stable and oscillating behavior in asynchronous systems respectively, is addressed within this thesis.

1.8 Structure of the thesis

After an introductory motivation for this work a brief treatment of the basics of synchronous and asynchronous logic design has been given. The survey on related design approaches pointing out their deficiencies is followed by an explicit presentation of the research problem, as well as an overview of the contribution of this thesis concludes the preface. The remainder of this thesis is organized as follows:

Chapter 2 gives a general overview on distributed clock synchronization starting with the introduction of the most important terms. Furthermore, established clock synchronization approaches are presented, classified and related to the research topic. The special class of tick generation schemes and algorithms are surveyed more thoroughly in the context of different failure models. Moreover, the challenges arising with the targeted fully asynchronous hardware tick generation scheme are discussed. In Chapter 3 the architecture of the chosen tick generation algorithm is developed. In particular, the design considerations yielding the building blocks of the asynchronous hardware implementation are treated in detail. The chapter concludes with the presentation of the formal modeling framework and its application to deriving the refined algorithm's correctness and performance measures. After the identification and partitioning of the algorithm's building blocks shown before, Chapter 4 presents the implementation of the HITS ASIC design. Additionally, comparisons to alternative tick generation implementations are provided. In order to enhance testability and to provide improved evaluation capabilities, the hardware components pre-

sented in Chapter 5 have been added to the tick generation design. Exhaustive assessments of the tick generation prototype's properties has been a crucial task, therefore Chapter 6 provides the theoretical and operational background for the measurements and validations conducted in Chapter 7. In the final Chapter the presented work is concluded by giving an outlook on open issues and related research topics.

Chapter 2

Distributed Fault-Tolerant Clocking

The only reason for time is so that everything doesn't happen at once.

Albert Einstein

THE ABILITY to reliably order events within a distributed system, even in the presence of faults and unreliable communication and components, is a prerequisite for the correct operation of many critical applications, e.g., real-time factory control, drive/fly-by-wire, accounting systems, etc. To provide each distributed component with the required time base that allows for a consistent view of events and states of the distributed system, some underlying coordinating technique has to be created.

The synchronization of distributed clock sources with the objective to achieve a common notion of time has been an active research topic in computer science from the late 1970s up to now. A wealth of existing research deals with theoretical results for clock synchronization in general and also transforms these results with respect to faults and distinct features and peculiarities of the underlying system models, as well as requirements given by the implementation technology. The features of existing schemes for fault-tolerant clocking are multifaceted and can be classified as ranging from deterministic to probabilistic approaches, dealing with different kinds of fault models (e.g., crash, omission, and Byzantine), aiming at different implementation platforms (hardware, software, hybrid). The variety in clock synchronization schemes leads to a broad spectrum of synchronization and fault-tolerance properties achieved by the individual approaches. For instance, the number of faults that can be tolerated largely depends on the fault type as well as the overall number of components involved. Similarly, the quality of synchronization depends on measures like system topology, implementation technique (e.g., hardware, software) and several other properties.

Overall this variety may easily lead to a difference in synchronization quality of distinct approaches by a factor of 100 and even more.

The following subsections are devoted to the introduction of a set of clock synchronization algorithms to develop the most important terms, features and properties of fault-tolerant clock synchronization. Furthermore, the distinct class of tick-generation algorithms will be stressed in more detail, since they form the algorithmic foundation of the novel clocking scheme presented in this thesis. Moreover, the challenges that arise when implementing an algorithm entirely in (asynchronous) hardware, while it has primarily been designed as software approach, are discussed.

2.1 Definitions and common terms

A distributed system as it is considered further on, is defined by a set of n nodes (processing units/processors) that are interconnected by communications links. If not stated otherwise, each node has access to a dedicated local clock source (e.g., a count register that is advanced by the oscillations of a quartz). Without going into details of the physical implementation of the local clocks, it is assumed that no global time is available to the processing nodes. The high-speed hardware clock (operating at micro-tick level) artificially relates real-time t to a discrete logical clock time $C_p(t) = T$ with a certain time granularity (makro-tick level) and bounded drift ρ_p (bounding the deviation from its nominal frequency). As already roughly sketched in Section 1.6, crystal-based oscillators which generate the local clocks for each node are not flawless in the way that their speed may drift. In detail, bounded drift means that within an interval Δt clock time deviates from real-time by a certain amount specified by the drift rate ρ. For crystal oscillators ρ typically ranges from 10^{-5} to 10^{-6}.

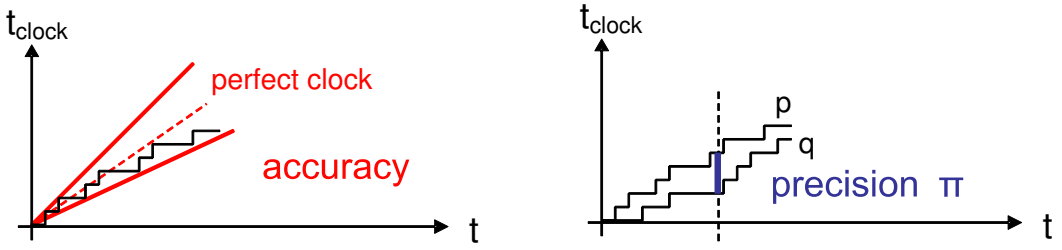


Figure 2.1: Accuracy and precision

Figure 2.1(a) depicts the deviation of a single clock from real-time. Moreover, a bound for this divergence, further called *accuracy*, is shown. Considering that a clock's speed typically deviates from the nominal value illustrates that even the synchronization of initially

perfect synchronized clocks might deteriorate over time. The reasons for the desynchronization are given by variations in the clocks' physical structure as well as effects of power supply and temperature fluctuations. To maintain bounded precision π (cf. Figure 2.1(b)) for all non-faulty clocks of nodes p, q and all real-times t, explicit synchronization measures (e.g., following a periodic resynchronization strategy) have to be taken. The precision can be defined as the smallest bound such that

$$| C_p(t) - C_q(t) | \leq \pi \tag{2.1}$$

holds, with C_p and C_q being the logical clock values of node p and q, respectively, i.e., the clock values of any two clocks is never off by more than π ticks. This synchronization problem can generally be defined as a set of constraints on the sending order of clock events/messages. Following Lamport's definition [52, 53], an algorithm solves a synchronization problem, if and only if constraints like precision π and accuracy are fulfilled. A further typically applied confinement, known as causality, is that a producer node has to send data before a consumer node may process the same portion of data.

A restriction to the general framework to better represent typical implementation topologies is that the only way for p to obtain q's clock value is by (direct or indirect) communication with q, e.g., by virtue of exchanging clock ticks or timing messages. For this kind of synchronization data, end-to-end send/receive delays within an interval $[d, d+\varepsilon]$ with a delay uncertainty ε — or alternatively, probability distributions of these delays — have to be assumed to be able to reflect meaningful real-world conditions. Furthermore, the end-to-end delay of message m is denoted by $\delta(m)$. The uncertainty in the message delivery time yields that only approximate time values can be obtained with respect to other nodes in a system. Theoretical results on clock synchronization reveal that the worst-case precision largely depends on the delay uncertainty ε, the drift rate ρ, the severity and number of failures f, as well as the resynchronization period and granularity of the underlying hardware clock [23, 55]. Furthermore, it is shown by impossibility results that no (deterministic) clock synchronization algorithm can achieve a precision

$$\pi \leq \left(1 - \frac{1}{n}\right) \varepsilon \tag{2.2}$$

Moreover, it has been proven in [29] that building a system tolerant to f Byzantine failures requires at least $n \geq 3f + 1$ nodes and a $2f + 1$-connected network.

2.2 Clock synchronization

Several approaches for clock synchronization algorithms have been proposed in scientific literature, a general overview can be found in [73] and [79]. In general, two approaches for synchronizing clocks can be employed:

- free running high-speed clock (micro-tick) with adjustable clock divider generating synchronized makro-ticks

- direct modification of the (primary) physical clock source (voltage controlled oscillator)

The synchronization schemes are usually classified according to the way the clock correction values are derived. Consistency-based and convergence averaging plus non-averaging as well as probabilistic and phased-locked loop (PLL) approaches are distinguished.

Consistency-based algorithms like the COM algorithm presented in [54] or the clock synchronization algorithm of the SIFT (Software Implemented Fault Tolerance) computer [90] are targeted at achieving agreement on clock values among non-faulty nodes by the application of an interactive consistency algorithm. The concept behind this approach is that every node dispatches its private clock value in a round-based manner throughout the system. This is achieved by direct communication as well as forwarding of clock values by other nodes, taking several rounds of message exchange. For example, in a setup of four nodes every node would get three messages with node p's clock value (one direct message and two forwarded by the remaining nodes). The agreement process has to ensure that all correct nodes decide on the same private clock values for every node — in the case that a correct node sent its clock value, it clearly has to agree on the initially dispatched value. At the end of each resynchronization interval, after the application of the message exchange and the agreement function, each node estimates its median skew with respect to the other nodes and makes appropriate adjustments to the local clock before initiating the next resynchronization process. Note that the number of message exchange rounds to achieve consistency in a system that allows for f Byzantine faults is given with $f + 1$ which considerably limits synchronization performance.

Convergence averaging algorithms are similar to the consistency-based scheme in the way that every node broadcasts resynchronization information at a predefined algorithm-specific time. After collecting and time-stamping the incoming resynchronization messages over some given period of time, each node computes a correction value for the local clock according to its own clock and the time values obtained from the other nodes. The actual correction is usually performed at the end of the resynchronization period by setting the local clock to the previously computed value. The fault-tolerant average function that is applied to derive the clock correction value essentially constitutes the difference among the convergence averaging approaches. In [54] the arithmetic mean — augmented by a threshold function to limit the impact of excessive faulty clocks — is used to compute the fault-tolerant average. In contrast to that, in [55] the f largest and f smallest values (where f is the maximum number of faulty clocks to be tolerated) are discarded before deriving the actual average for the clock correction. This fault-tolerant mid-point algorithm has also been used in the MAFT (Multi-Computer Architecture for Fault-Tolerance) design [49]. In general, the most severe limitation of convergence averaging algorithms is given by the need for a known upper bound on the clock read error (message uncertainty ε) since

this directly influences the attainable precision, however, this value is (at least in large distributed systems) hard to obtain.

Convergence non-averaging clock synchronization algorithms also follow a round-based resynchronization strategy. However, instead of computing the averages of clock values and applying corrective measures at fixed intervals, each node updates its clock either if the underlying hardware clock has reached a predefined resynchronization time, or if a valid resynchronization message from another node has arrived. In the case of the algorithm by Srikanth and Toueg [81] each node considers all messages suspicious and only re-synchronizes after receiving messages from at least $f+1$ other nodes indicating that they have already re-synchronized. This ensures that the clock of at least one non-faulty node has reached the time for resynchronization (f again represents the maximum number of Byzantine faulty nodes in the system). The worst case precision π of convergence non-averaging algorithms is a strong function of the maximum message delivery time between any pair of non-faulty nodes. Therefore, algorithms of this class are not particularly suitable for large networks with multiple message hops if tight synchronization has to be achieved.

Probabilistic clock synchronization algorithms are targeted to provide a remedy for the most restricting limitation of the deterministic approaches presented so far, namely the clock read error (which is directly related to the message uncertainty ε and therefore limits precision π). The algorithm introduced by Cristian [16] is based on the idea that the message delay's probability distribution is known and that each node can take numerous readings of the other nodes' clocks to estimate the clock read error. This scheme allows to achieve arbitrarily small worst-case skew, however, it comes at the price of high computational effort and increased message count. Furthermore, the algorithm cannot guarantee synchronization, in fact there is a non-zero probability that synchronization is lost. Moreover, since the approach follows a master/slave architecture it is not fault-tolerant unless additional measures for fault-detection and a process for electing a new master clock are provided. A strategy for this purpose proposed by Cristian is the introduction of master groups instead of a single master clock.

Almost all of the algorithms presented above are usually implemented in software with no, or rather little hardware support. Nevertheless, some hardware implementations as well as pure hybrid approaches are noteworthy:

The four-clock design in [64] essentially presents a hardware implementation of the convergence averaging algorithm of [55] using the fault-tolerant midpoint as averaging function. Like other algorithms of this class initially synchronized clocks are required to circumvent the formation of cliques at startup. In contrast to the underlying provable correct clock synchronization algorithm which is able to tolerate Byzantine faults, some restrictions apply for the hardware implementation. The skew between different clocks is limited to a small number of ticks, furthermore, excessive clock jitter of the local clock in

the range of 10% of the period can upset the whole synchronization approach. Another critical issue is given by the fact that interconnection delays among the four nodes adversely affect the fault-detection capabilities.

A hybrid design incorporating a mostly software-based approach with some hardware support for the time-stamping of incoming synchronization messages was presented in [45]. The distinctive features of this clock synchronization scheme are given by the hardware supported time-stamping and a high resolution adder-based clock. The hardware assisted time-stamping feature ensures that the clock read error is kept to a minimum. Additionally, the adder-based clock, instead of just increasing the clock-register with every oscillation, adds some adjustable value. Following this scheme allows the implementation of state as well as rate correction abilities in an elegant way.

Phase-Locked loop clock synchronization aims at the direct adjustment of the oscillation frequency of the involved hardware clocks. For this purpose a voltage-controlled oscillator, representing the clock source at its respective node, is controlled in a way that it always stays tightly synchronized with all other nodes' clock sources. In contrast to the clock synchronization schemes introduced above no clock-register and counting mechanism is needed. All operations and adjustments directly influence the *up* and *down* transitions of the oscillator. The common setup for the PLL approach is depicted in Figure 2.2. All clock signals originating in different nodes are processed via an input logic block which essentially selects one of its inputs to be the active reference clock. The subsequent phase detector uses the reference signal in conjunction with its local clock to generate the voltage controlling the local oscillator's frequency. As long as every node selects the same clock and adjusts its local oscillator to this reference signal, the whole set of (non-faulty) clocks will stay synchronized to each other. The condition that the same clock has to be chosen as reference at all nodes denotes the most critical point, its violation can yield globally unsynchronized cliques of clocks. However, even slightly skewed clock arrivals, e.g., due to crosstalk [27] might lead to inconsistencies among the nodes. Unfortunately, when implementing a PLL-based clocking scheme in large network topologies or when using modern clocking technology, delay jitter in relation to clock period might not be negligible. The second limitation of PLL-based fault-tolerant clocking is given by the fact that most approaches need fully connected networks, which i) can be very costly in large topologies and ii) aggravates the issue of erroneously selecting different reference clocks. Last but not least, the limited pulling range of every PLL, when switching from one reference clock to another, has to be considered, too.

Several implementations presenting a wide difference in clock speed and synchronization precision can be found in literature. The Fault-Tolerant Micro-Processor (FTMP) [46] uses a four clock approach capable of tolerating one Byzantine failure. However, due to the selection of the reference clock on the basis of the median this scheme is not easily extendable to cope with multiple Byzantine faults. Another implementation using four clocks and a digitally controlled oscillator was proposed

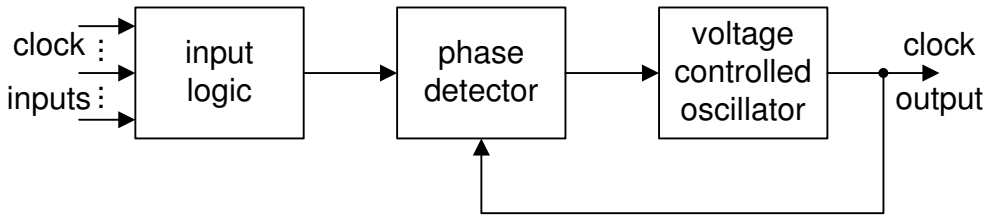


Figure 2.2: Phase-locked loop hardware clock synchronization

in [88]. Again the selection of the reference clock (implemented by a voter circuit) forms the most critical design component. The case of input signals experiencing various skew at different nodes may lead to discrepancies in the voter inputs and outputs with the possibility of de-synchronization of the system. Clocking schemes able to tolerate multiple faults are presented in [14, 89] with both designs relying on enhanced methods for selecting a reference. In [89] the whole clock network, the input voting circuit as well as the phase detector are duplicated. This additional redundancy in conjunction with an extra unit averaging the correction values provided by both phase detectors aims at increased robustness against delay variations. The implementation proposed in [14] uses elaborate reference clock voting employing a local high-speed clock for sequencing and preselecting incoming remote clock signals. Naturally, this capturing of signals across clock domains introduces the risk of generating metastability issues in the voter/phase detector circuits.

Given all the presented algorithms and schemes for the generation of a fault-tolerant time-base within a distributed system — with the dimension of this system ranging from a single chip to the size of a worldwide computer network — it is obvious that no single approach will be a feasible solution for all imaginable application fields. For instance, when summarizing the presented schemes it has to be noted that, despite some mentioned issues both, the hardware implementations of convergence as well as PLL-based clock synchronization yield substantially better synchronization precision than software schemes. In fact, the presented hardware approaches' precisions are in the range of some ten nanoseconds in contrast to several hundreds of microseconds of software-based designs.

In the light of the targeted research of this thesis, namely, the provision of a fault-tolerant alternative to global clocking for VLSI computer chips, some of the clocking approaches presented so far can generally be ruled out due to various shortcomings. Recalling the design requirements described in Section 1.7, following a software-based approach is no viable option since the targeted solution is to directly provide a reliable high speed clock to synchronous circuits. Consistency and convergence-based schemes and even more demanding probabilistic algorithms require rather complicated computations and therefore are far too complex and costly to be directly built in hardware. Taking a closer look at existing hardware implementations like the convergence-averaging approach in [64], or PLL-based designs as [14, 89] reveals two major problems. At first, all of the presented

schemes require multiple clock sources, e.g., crystal oscillators as foundation of their operation. Again the requirements from Section 1.7 do not allow external clock sources due to their limited reliability and question the stability of commonly used on-chip RC-oscillators. Moreover, synchronizing multiple clock domains automatically introduces metastability issues. Another limitation is given by the excessive skew expected for future VLSI chips clock network, since it has the potential to create havoc within the above presented synchronization schemes.

Unlike the schemes presented above a new strategy incorporating algorithms which are computationally affordable and especially suitable for hardware implementation has to be found. Moreover, the particular class of investigated algorithms has to ensure that the troublesome local clock sources (oscillators) are no longer needed, while still being able to provide a globally available synchronized clock. These requirements directly lead to the class of tick generation algorithms which are presented in the subsequent section.

2.3 Tick generation

A distributed system implementing message passing can be classified as a tick generation scheme if it satisfies a particular set of properties. In a tick generation system the characteristics of each node's message passing algorithm (with all nodes executing identical copies) are the following. The algorithm consists of a set of rules which are triggered whenever a message arrives at a node. These rules may possibly update the respective node's local memory as well as send messages to other nodes. Additionally, to represent a tick generation approach the class of distributed systems is restricted to those which send only messages of ascending natural numbers at each node, i.e., it is demanded that every node p sends messages $\langle 0\rangle, \langle 1\rangle, \langle 2\rangle, \ldots$ in the given order during its executions. However, the properties presented so far are not necessarily sufficient to achieve synchronization among all non-faulty nodes of a distributed system. A tick generation algorithm solves the initially sketched synchronization problem if precision π and accuracy hold. Furthermore, it is called a fault-tolerant algorithm if it maintains the conditions described above even in the presence of faults. To give a distinct relation of tick generation to the hardware clock synchronization methods presented in detail, the natural numbers of tick generation messages $\text{TICK}(k)$ mod 2 can be seen as discrete *up* and *down* transitions of a PLL-based clock synchronization approach.

An algorithm by Srikanth and Toueg [81] that actually fulfils all criteria for tick generation has already been mentioned in the context of non-averaging clock synchronization. A detailed introduction of this approach is presented in Algorithm 1 and Algorithm 2. Moreover, an in-depth description of the algorithms, which in conjunction with each other implement a fault-tolerant tick generation approach, follows next.

Algorithm 1, the so-called non-authenticated clock synchronization part works in a way that node p broadcasts message $\text{TICK}(k)$ as soon as its clock counter $C^{k-1}(t)$ reaches the value indicating that the next tick has to be issued. The internal clock counter is directly

Algorithm 1 Non-authenticated algorithm for clock synchronization at node p [81]

```
1: variables
2:    k : integer := 0
3: end variables
4: if C^{k-1}(t) = kP then //ready to start C^k
5:    → broadcast(TICK(k))
6: end if
7: if accepted the message(TICK(k)) then //according to a selection/voting function
8:    → C^k(t) := kP + α
9: end if
```

driven by a local oscillator. The round-based threshold value is given by kP, where P denotes the predefined resynchronization interval. Due to the fact that a faulty local clock source could, for example, erroneously speed up node p, the clock counter $C^k(t)$ is set to TICK(k) (re-synchronized), only if an accepted TICK(k) message has already been received from another node (actually, $C^k(t)$ is adjusted to $kP + \alpha$ where α denotes a constant ensuring that the clock always steps forward in time). The execution described above is also depicted in Figure 2.3.

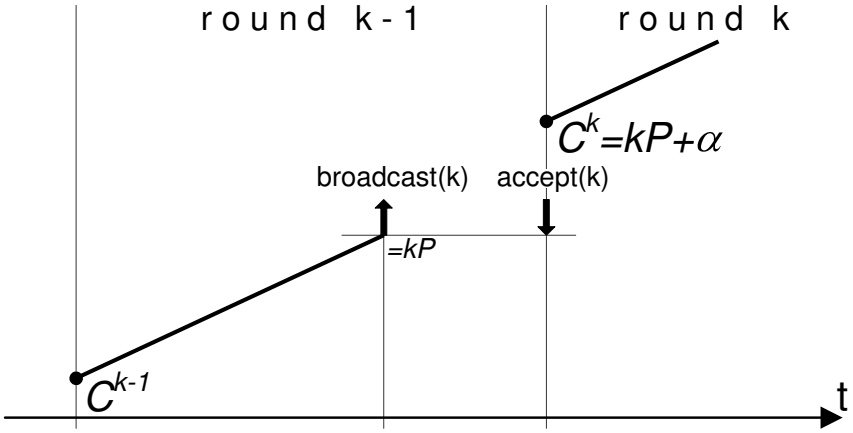


Figure 2.3: Non-authenticated broadcast execution at node p

The function responsible for "accepting" TICK(k) messages, whose activation in return leads to resynchronization and progress of the clock, is shown in Algorithm 2 and comprises three parallel rules for message processing. In response to the reception of particular TICK(k) messages from at least $f + 1$ distinct nodes, either via *init* or *echo* messages, each node relays a *echo* TICK(k) message to all other nodes (Relay rules). The actual generation of an "acceptance event" for advancing the clock, however, requires the reception of at least $2f + 1$ distinct *echo* TICK(k) messages (Accept rules). It has been shown by Srikanth and Toueg that in a system of $n \geq 3f + 1$ nodes Algorithm 1 in cooperation with the consistent broadcast primitive of Algorithm 2 solves the clock synchronization problem, even in the

presence of up to f Byzantine faulty nodes if the conditions hold that:

- the local clocks' maximum drift rate is known and bounded by ρ.
- message end-to-end delays are within a certain known bound of $[d, d+\varepsilon]$.
- two specific timing assumptions are ensured by properly chosen values for P and α.

Algorithm 2 Acceptance function selecting valid clock ticks [81]

```
1: variables
2:    k : integer := 0
3: end variables
4: for each correct process do
5:    if received(init,TICK(k)) from at least f + 1 distinct nodes then //Init Relay Rule
6:       → send(echo,TICK(k)) to all
7:    end if
8:    if received(echo,TICK(k)) from at least f + 1 distinct nodes then //Echo Relay
9:       → send(echo,TICK(k)) to all
10:   end if
11:   if received(echo,TICK(k)) from at least 2f + 1 distinct nodes then //Accept Rule
12:      → accept(TICK(k))
13:   end if
14: end for
```

Let us reconsider the initial motivation for taking a closer look at tick generation algorithms, namely, getting synchronized clocks without the need for local clock sources. As described above, the non-authenticated algorithm for clock synchronization, presented in Algorithm 1, still requires a local clock source at each node to be able to assess the instant of time ($= kP$) when, from a local point of view, the next TICK(k) message has to be broadcast — the next resynchronization event has to be triggered. Fortunately, a minor modification of Algorithm 1 yields a solution which does no longer rely on local clocks. In Algorithm 3 node p starts issuing TICK(k+*1*) immediately after TICK(k) has been broadcast. Similarly to the acceptance function of the original consistent broadcast scheme of Algorithm 2, Algorithm 3 responds to *init* TICK(t) with *echo* TICK(k) messages and relays *echo* TICK(k) messages if sufficiently many ($= f + 1$) distinct nodes issued those TICK(k) messages. In contrast to the clocked consistent broadcast primitive, Algorithm 3 broadcasts a new TICK(k+*1*) message to all other nodes as soon as it has received *echo* TICK(k) from at least $2f+1$ distinct nodes, instead of waiting for a timed resynchronization interval P to elapse.

A further simplification of Algorithm 3 has been presented by Widder and Schmid [93] and yields Algorithm 4. This algorithm, besides its operation without a local clock source, no longer requires the distinction of *init* and *echo* events, which severely eases message

Algorithm 3 Consistent broadcast primitive without local clock source

```
1: variables
2:    k : integer := 0
3: end variables
4: for each correct process do
5:    if received(init,TICK(k)) from at least f + 1 distinct nodes then //Init Relay Rule
6:       → send(echo,TICK(k)) to all
7:    end if
8:    if received(echo,TICK(k)) from at least f + 1 distinct nodes then //Echo Relay
9:       → send(echo,TICK(k)) to all
10:   end if
11:   if received(echo,TICK(k)) from at least 2f + 1 distinct nodes then //Accept Rule
12:      → broadcast(TICK(k+1))
13:   end if
14: end for
```

handling. Furthermore, the previously used assumption on message delays $[d, d + \varepsilon]$ can be weakened to the one that for any two messages in transit m_1, m_2 it has to hold that,

$$\frac{\delta(m_1)}{\delta(m_2)} \leq \Theta \tag{2.3}$$

with $\delta(m_1)$ and $\delta(m_2)$ being the respective message delays of m_1 and m_2, and Θ being constant. The analyses in [93] show that despite the presented substantial simplifications, Algorithm 4 still solves the clock synchronization problem — in this particular case, this is maintaining precision π as well as accuracy even in the presence of Byzantine faults. The "Relay Rule" of a correct node fires as soon as TICK(ℓ) messages from at least $f + 1$ distinct nodes have been received — this way it is ensured that at least one of these TICK(ℓ) messages has been issued by a correct node. A property making this algorithm extremely suitable for hardware implementation is the fact that it does not immediately set its local clock k to ℓ in the case of triggering the "Relay Rule" since this could possibly lead to skipping some values of k if the respective node is lagging more than one tick behind. The strategy followed in Algorithm 4 explicitly ensures that all messages TICK(k), ..., TICK(ℓ) are issued when catching up with faster nodes, resulting in a continuous progression of the clock without potentially troublesome leaping effects. Especially when recalling the targeted application of clocking synchronous circuits, skipped clock ticks might result in varying progression of distributed circuit blocks leading to inconsistent state information. As an example for the catch up scenario, consider a slow node p with its local clock value $k = 3$. If this node, for instance, receives at least $f + 1$ TICK(ℓ) messages with $\ell = 6$, node p will consecutively broadcast TICK(*3*), TICK(*4*), TICK(*5*) and TICK(*6*) and set its local counter to $k = \ell = 6$.

Algorithm 4 Modified version of Srikanth & Toueg's Byzantine-tolerant tick generation [93]

```
1: variables
2:     k : integer := 0
3: end variables
4: initially send TICK(0) to all
5: if received TICK(ℓ) from at least f + 1 rem. processes with ℓ ≥ k then //Relay Rule
6:     send TICK(k), ..., TICK(ℓ) to all
       k := ℓ
7: end if
8: if received TICK(k) from at least 2f + 1 remote processes then //Increment Rule
9:     send TICK(k+1) to all
       k := k + 1
10: end if
```

2.3.1 Algorithms for weaker failure models

The tick generation algorithms presented above are designed for worst case failure assumptions, i.e., the Byzantine faulty case. However, the resilience to arbitrary failure scenarios comes at a price. The theoretical results by Dolev, Halpern and Strong [23], giving lower bounds on the connectivity as well as the number of nodes in a distributed system, also apply to tick generation. Therefore, being able to deal with up to f Byzantine faulty components requires $n \geq 3f + 1$ nodes and at least $2f + 1$ connectivity. It has to be noted that the connectivity constraint can be reduced to $f + 1$. This reduction applies if, for example, authentication schemes ensure that messages cannot be altered or completely faked in an undetected way. However, implementing authentication is far too costly in terms of hardware resources and hence does not provide a viable solution in the targeted application domain. The resulting (almost) fully connected point-to-point network of a Byzantine-tolerant scheme hence implies a quadratic growth of the number of links with the number of nodes n of a distributed system and thus with the number of tolerable failures f, i.e., is in $\mathcal{O}(f^2)$.

Another possibly troublesome part in tick generation is given by the implementation of the rules responsible for the fault-tolerant distribution of TICK() messages. In the case of Algorithm 4 these are the "Relay Rule" (triggering as soon as $f+1$ TICK(k) messages have arrived) and the "Increment Rule" (broadcasting TICK($k+1$) after the $2f + 1^{st}$ TICK(k) message has been received). These m-out-of-n threshold functions ($f+1$-out-of-$3f+1$ and $2f + 1$-out-of-$3f + 1$ in the case of Algorithm 4) can result in rather complex and costly implementations. A detailed analysis of the implementation complexity is presented in Chapter 4.

In contrast to Byzantine faults, benign failure modes are more restricted. When dealing with benign failures, node p receiving TICK(k) from node q can be sure that q sent this message and that it also arrived timely (within an interval bounded by Θ). These

restrictions to the allowed failure modes lead to weaker failure semantics but also yield simpler algorithms.

Clean crash Recalling the definitions in Section 1.5.1, a node p is said to have cleanly crashed at time t if all messages sent by p before time t are correctly received by all other non-faulty nodes. Furthermore, all messages from node p sent at time t or later are not received by any node. Algorithm 5, presented together with a proof in [92], solves the clock (tick) synchronization problem in the presence of clean crashes. To be able to tolerate up to f clean crashes, $n \geq f+1$ nodes are required. The presented algorithm only comprises a single rule, substantially simplifying an implementation. Furthermore, this rule is rather elementary since it has to trigger as soon as a single TICK(k) message has arrived.

Algorithm 5 Crash-tolerant tick generation

```
1: variables
2:     k : integer := 0
3: end variables
4: initially send TICK(0) to all
5: if received TICK(ℓ) from at least 1 remote processes with ℓ ≥ k then
6:     send TICK(k), ..., TICK(ℓ + 1) to all
       k := ℓ + 1
7: end if
```

Crash failures, also called potentially *unclean crashes*, are characterized by scenarios where node p is said to have crashed at time t if all messages sent by p before time t are correctly received by all other non-faulty nodes. Furthermore, a TICK() message from node p sent exactly at time t is received only by a subset of the distributed system's nodes. Messages sent by p later than t are not received by any node. In other words, when node p crashes the last TICK() message may be inconsistently received by some of the nodes. Interestingly, the approach presented in Algorithm 5 not only tolerates clean crashes, but also unclean ones. However, the algorithm's precision π becomes dependent on f when used in the presence of unclean crashes [92].

Omission failure semantics are much less restrictive than crash scenarios since asymmetric behavior is permitted an unbounded number of times. In more detail, a node p is called omissive if a message sent by this node is only received by a subset of the remaining non-faulty nodes of the distributed system. This condition renders the previously presented crash-resilient Algorithm 5 useless. Algorithm 6 has been presented in [92] to tolerate unclean crashes. This algorithm, however, is also able to handle up to f omissive nodes in a system of $n \geq 2f+1$ nodes

In contrast to the (unclean) crash-tolerant algorithm presented above, Algorithm 6 provides a precision independent of f in the presence of up to f omissive nodes. Given

Algorithm 6 Omission-tolerant tick generation

```
1: variables
2:     k : integer := 0
3: end variables
4: initially send TICK(0) to all
5: if received TICK(ℓ) from at least 1 remote processes with ℓ ≥ k then //Relay Rule
6:     send TICK(k), ..., TICK(ℓ) to all
       k := ℓ
7: end if
8: if received TICK(k) from at least f + 1 remote processes then //Increment Rule
9:     send TICK(k+1) to all
       k := k + 1
10: end if
```

that crashes can always be modeled as omissions, Algorithm 6 can be used to achieve a precision π independent of f in the case of unclean crashes. However, the price for being able to tolerate an unbounded number of asymmetric message receptions yields a significant increase in the algorithm's complexity if compared to the crash-tolerant version presented in Algorithm 5. Similarly to the Byzantine-tolerant Algorithm 4, the omission-tolerant version has to rely on two rules to be able cope with asymmetric behavior. However, it has to be noted that the "Relay Rule" of the omission-tolerant algorithm is as simple as the crash tolerant algorithm's rule, since it triggers as soon as a single TICK(k) message has arrived. On the downside, the "Increment Rule" responsible for advancing the clock, incorporates a rather complex n-out-of-m rule, actually $f + 1$-out-of-$2f + 1$, like the ones presented with the Byzantine-tolerant schemes.

In addition to the static failure models presented above and suitable algorithms for achieving clock synchronization, more elaborate dynamic failure models, as already mentioned in Section 1.5.1, can be utilized. For instance, a crash-tolerant system applying a dynamic failure model could be characterized by the following conditions:

- Any node may crash and later recover in time as long as at least one process stays up during a sliding time window.
- The rate of (unclean) node crashes has to be bounded.

It has been shown in [7] that Algorithm 5, which was initially introduced as crash resilient for the case of static crashes, also solves clock synchronization in a dynamic failure model. However, its precision π is dependent on the rate of unclean crashes.

2.4 Hardware implementation challenges

All presented tick generation algorithms have initially been designed for software implementation purposes, therefore they are innately not particularly suitable to implement a hardware clock signal. A tick generation algorithm, as it has been defined in Section 2.3, operates on unbounded natural numbers, whereas a hardware clock signal simply transits between the two logic values *high* and *low*. An appropriate mapping between these two representations of clock ticks, shown in Figure 2.4, has to be implemented to build the demanded fault-tolerant hardware clock.

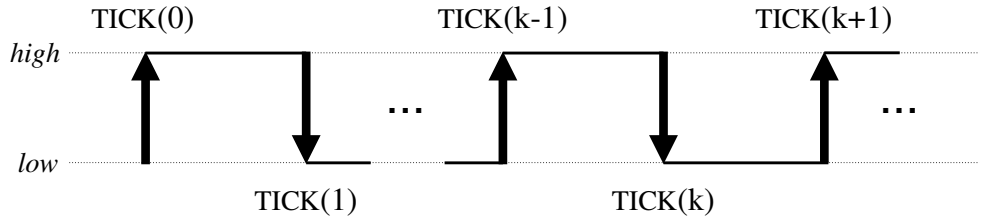


Figure 2.4: Hardware clock signal vs. tick numbers

Clock frequencies in state of the art VLSI design typically range into hundreds of MHz or even some GHz. This incredible clock speed directly translates to the fact that hundreds of millions TICK() messages have to be transferred every second to implement tick generation. This huge amount of data, of additionally ever increasing values of k, transmitted via TICK(k) messages poses a severe implementation challenge. To enable a reasonable hardware implementation of the Byzantine-tolerant tick generation scheme of Algorithm 4, further adaptations will be necessary. The transmission of unbounded size TICK(k) messages clearly cannot be implemented, thus the tick numbers k have to be bounded. The restriction of bounding k has to be accounted for in the algorithmic design since it leads to wrap-around effects in the numerical representation of the TICK(k) numbers, i.e., after sending the largest value of k in the chosen integer representation, follows the smallest one, e.g., TICK(0). Hence, it is important that:

- it is ensured that no TICK(k) messages of different wrap-around phases interfere with each other, and a bound on the maximum offset of any two clocks holds.
- both parallel rules (Increment- and Relay-Rule) of a node p never generate and sequentially transmit the same TICK(k) message, since this might erroneously trigger an extra tick generation at remote nodes due to above described ambiguity of TICK(k) messages.

To account for these effects, Algorithm 7 comprises a minor augmentation to better accommodate for bounded values of k — it requires that every TICK(k) message is only sent **[once]** regardless of the fact that multiple rules might be eligible to generate this particular TICK(k) message. Guaranteeing this mutual exclusiveness of actions in the case of

parallel computations, however, poses another implementation challenge, especially when considering the required asynchronous implementation approach.

Algorithm 7 Byzantine-tolerant tick generation [93] suitable for bounded tick numbers

```
1: variables
2:     k : integer := 0
3: end variables
4: initially send TICK(0) to all [once]
5: if received TICK(ℓ) from at least f + 1 rem. processes with ℓ ≥ k then //Relay Rule
6:     send TICK(k), ..., TICK(ℓ) to all [once]
       k := ℓ
7: end if
8: if received TICK(k) from at least 2f + 1 remote processes then //Increment Rule
9:     send TICK(k+1) to all [once]
       k := k + 1
10: end if
```

The shown algorithmic feasibility of a fully asynchronous implementation for tick generation, following the refinements presented in this chapter, clears the path for the desired design which does no longer need additional clock sources. Thus the next chapter is mainly devoted to finding a suitable mapping of algorithmic statements to hardware building blocks

Chapter notes

This chapter presented extensive introduction to the concepts, algorithms and implementation of distributed clock synchronization. Software, hardware, as well as hybrid solutions have been surveyed and compared to the requirements developed in Section 1. Tick generation algorithms, especially the one based on Srikanth and Toueg's consistent broadcast primitive have been treated in more detail, since an adaptation by Widder and Schmid proved to be attractive for the targeted design of a distributed fault-tolerant hardware clock. To get a rough sense for the complexity coming along with Byzantine-tolerant tick generation, algorithms for weaker failure models (crash- and omission-tolerant, respectively) have been compared to the Byzantine-tolerant scheme. In the context of the examinations on weaker failure models two own publications can be noted: in [33] the differences of hardware fault models to distributed failure models are discussed. More precisely, the complexity of algorithms following different failure models is analyzed and resulting hardware implementations are compared to each other. As a result, the appropriateness of usually used distributed systems models are questioned for application in hardware design. The second publication in this context [34] is also concerned with hardware implementations considering different failure models. However, failure transformation methods to

emulate less restrictive failure models are additionally presented. This failure transformation relies on the usage of multiple nodes following weaker failure models. For instance, an omission-tolerant algorithm could be replicated and augmented by a transformation algorithm to implement the less restrictive Byzantine failure model. By following this strategy, a set of rather simple omission-tolerant nodes together with its transformation algorithm form a so-called "super-node" which enables Byzantine fault-tolerant operation among the group of "super-nodes" (failure transformation will be treated in more detail in Chapter 4). Additionally, in the technical report [7] Algorithm 5 has been analyzed in the context of the dynamic crash/recovery failure model. Yielding a synchronization precision π which depends on the rate of unclean crashes.

Chapter 3

Hardware Implemented Fault-Tolerant Tick Generation

Even a stopped clock is right twice a day.

Marie Von Ebner-Eschenbach

FAULT-TOLERANT TICK generation schemes in general, despite their origin in the software-based realm, seem to be implementable in hardware with reasonable effort. The main part of this chapter is therefore devoted to going into more detail regarding the algorithmic assumptions as well as hardware related peculiarities. The in-depth analysis is aimed at revealing requirements and constraints involved with the asynchronous nature of the targeted tick generation implementation. Furthermore, it will be shown that the high-level description of the algorithms presented so far has to be substituted by one that more precisely models asynchronous hardware design blocks. This specific mapping from algorithmic high-level description to hardware design elements, which finally can be implemented by following an asynchronous design style, shows the huge gap between the two design spaces. In particular, abstractions for simple operations are assumed to be available to and employed by distributed algorithms. However, at gate-level hardware design these abstractions have to be explicitly created by employing some more or less complicated implementations, e.g., the atomic receive-compute steps usually assumed in distributed systems pose an intricate problem for asynchronous hardware design. In this context, the main difficulty originates from the fact that data items are concurrently processed by a large number of independent hardware units.

3.1 Constraints, requirements and characteristics

High-level requirements for the hardware implementation of the fault-tolerant tick generation approach have already been introduced in Section 1.1. Targeting a CMOS ASIC fabrication process and allowing that the number of nodes in a system is configurable, as well as including augmented measurement support, has no direct impact on the architectural design of the tick generation algorithm. Considering that wiring in distributed systems is among the most costly and error-prone components, multiple clock rails per communication link (introducing additional skew issues) are not considered a feasible option. Even for the targeted rather localized implementations of systems-on-chip or board-level designs the above mentioned issues cannot be neglected. Moreover, it has to be considered that the chosen tick generation algorithm assumes a fully connected tick generation network (TG-Net) leading to the generally high amount of n^2 links. Given the restrictions and requirements stated above, multiple rails per clock line have to be avoided whenever possible.

To meet this requirement, serialization of TICK(k) messages has to be implemented, or alternatively, tick numbers need to be maintained at the receiving side of the clock network. Unfortunately, serialization and de-serialization of tick numbers stands in serious contrast to the main design requirement in clock design, which is to achieve high speed. On the other hand, maintaining tick values at the receiver side and using simple *up/down* transitions to convey the clock information will require additional hardware resources at every node. Additionally, this strategy is challenging from an algorithmic point of view too, since no state information can be conveyed over the clock network. The only information available at the receiving node implementing such a scheme would be the respective arrival times of *up*/↑ or *down*/↓ transitions. The first approach, in which entire TICK(k) messages are conveyed, is schematically depicted in Figure 3.1(a), whereas Figure 3.1(b) shows the alternative scheme that relies on receiver-sided tick counters and the transmission of *up/down* transitions only.

The most demanding design requirement is certainly given by the need for an entirely asynchronous hardware implementation, since this restriction completely rules out the usage of the well established synchronous design methods. The tick generation design has to follow a handshake-based flow control of asynchronous design styles (like the one presented in Section 1.2.2) which ensures that no old data interferes with new one — it provides *interlocking* between subsequent data waves. However, in the context of fault-tolerant design a fundamental problem arises for the request (REQ) and acknowledge (ACK) schemes of asynchronous design methodologies. If a design is waiting until *all* request signals have arrived before broadcasting the next acknowledge, this would allow a single faulty unit to inhibit any further processing. Hence, a strategy has to be followed where processing is halted until an algorithm-dependent threshold of request signals has been reached. While such an approach now enables the incorporation of fault tolerance, it necessarily breaks the implementation's REQ/ACK feedback loops for the slowest paths. Without additional measures or constraints such open loops tend to run out of sync, endangering the correct

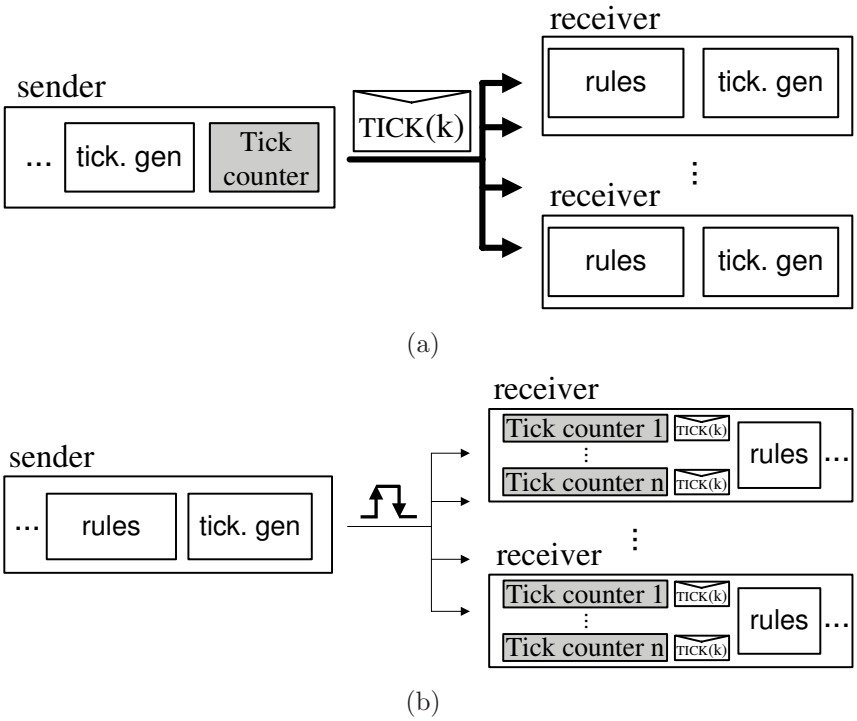


Figure 3.1: Schemes for conveying tick number k

operation of the whole system. In order to obtain a fault-tolerant asynchronous design the traditional closed REQ/ACK control loops have to be augmented by explicit timing constraints, this way supporting an interlocking scheme for consecutive data waves. In addition to the fault tolerance issue, another argument against acknowledging all requests is closely related to the above discussion on the method for conveying TICK(k) messages: acknowledging every single clock signal would yield additional globally distributed signal rails for every link of the clock distribution network which clearly is uneconomic in the particular case of clocking.

In summary, the most important design requirements and challenges for the tick generation hardware can be itemized as the following:

tick generation network: Integer TICK(k) messages have to be conveyed in a way to keep the clock network as simple as possible without jeopardizing clock speed. Therefore, strategies having more than a single wire per clock signal are assumed too costly.

tick messages: Simple TICK(k) messages have to be used to enable highest possible speed while still operating on a single rail per clock signal — clock transitions (*up/down*) on a single signal rail, as depicted in Figure 2.4, appear to be the only viable implementation option.

asynchronous design: In the context of this work no clock sources are allowed in the implementation of the tick generation scheme — a fully asynchronous approach has to be followed. However, rigorous closed-loop operation of typical asynchronous design styles is not applicable due to the demanded fault tolerance properties.

atomicity of actions: For any distributed computing model that the author is aware of, atomic computing steps at the level of a single node are assumed. However, this abstraction cannot be adopted when an algorithm is implemented directly in hardware, since all computations are performed by numerous concurrently operating digital logic gates. The challenging part in this case is given by the parallel processing of multiple algorithmic rules (e.g., "Relay Rule" and "Increment Rule" of Algorithm 7) in conjunction with concurrently arriving TICK() messages. To handle this issue explicitly, synchronization of local computations is needed. In a fully asynchronous design atomicity of actions — non-interference of subsequent data waves — in the absence of handshaking can only be guaranteed by the introduction of timing constraints[1].

3.2 Identifying building blocks for an asynchronous hardware implementation

Refining the software-based tick generation approach by Srikanth and Toueg, presented in Algorithm 3, to make it conform with the needs of an asynchronous hardware implementation, led to Algorithm 7 by Widder and Schmid. However, the conducted adaptations only denoted a few first steps in the process of transforming the fault-tolerant tick generation scheme to a hardware implementation. The mapping of algorithmic statements to design units and the identification of appropriate implementation techniques to achieve provably correct hardware with reasonable speed, proved to be an intricate task. As elaborated later on in this section, several adaptations to Algorithm 7 have been necessary to make it conform to the constraints and requirements presented in Section 2.4 and Section 3.1. Figure 3.2 shows the basic architecture of a single tick generation node's hardware design resulting from the above described specifications.

The most notable peculiarity of the design depicted in Figure 3.2 is given by the dissemination strategy for TICK(k) messages which follows the approach depicted in Figure 3.1(b). That is, no explicit tick numbers are transmitted over the tick generation network (TG-Net). Anonymous *up* and *down* signal transitions (zero-bit messages) are used instead of conveying integer values. As indicated in the previous section and shown in Figure 3.1(b), this scheme requires that the actual tick number k is maintained at every receiver. For this purpose the major hardware building blocks of a single tick generation algorithm (TG-Alg), following the basic design of Algorithm 7, are given by the remote counters in

[1]The implementation of fault tolerance in general as well as constraints on the clock network inhibit that all signals may be handshaked.

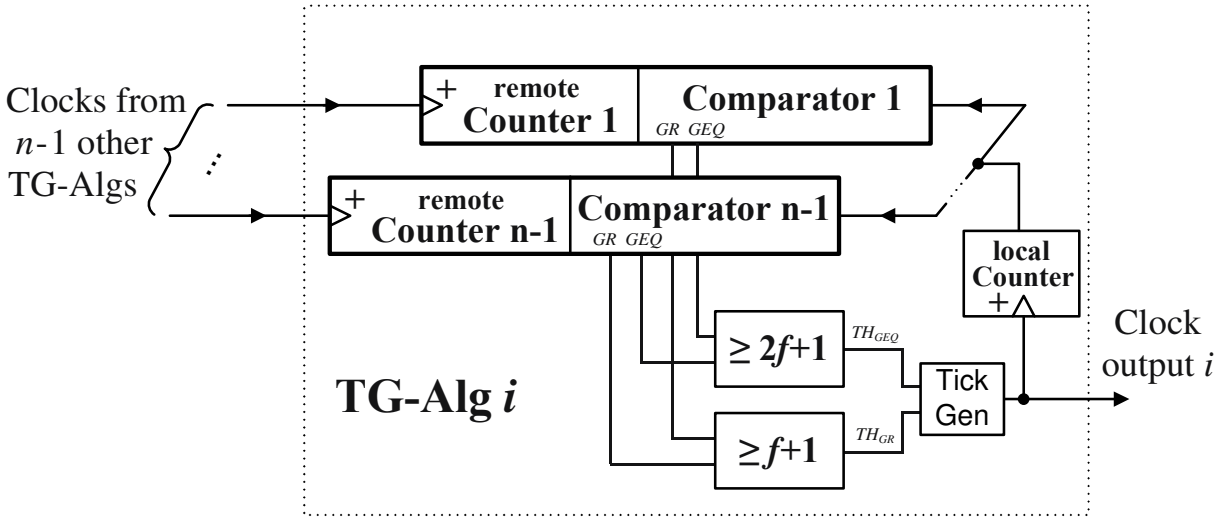


Figure 3.2: Basic hardware architecture of Algorithm 7

conjunction with the local counter unit. Each TG-Alg instance has to incorporate $n-1$ of these *remote counters*, one for every of the $n-1$ other tick generation nodes in the system. To indicate whether or not a remote clock has generated more or at least as many ticks as the local node, every remote counter i has to be augmented with a comparator which provides two binary status signals, *GR* and *GEQ*. With *GR* being set to *true* when the remote node's tick number is greater than that of the local node, i.e., the difference of the actual counter values is greater zero. *GEQ* becomes active with the remote counter's value greater or equal to the local counter, thus representing the condition that at least as many ticks have been issued locally as the number of ticks already received from the respective remote node. The "Relay Rule" (`line 5`) and "Increment Rule" (`line 8`) in Algorithm 7, evaluate these "greater" and "greater or equal" conditions. The respective design units, representing the evaluation of the *GR* and *GEQ* signals are given by the $\geq f+1$ and $\geq 2f+1$ circuit blocks. These *threshold circuits* prepare the generation of the next ticks in a fault-tolerant manner by masking faulty inputs. The actual and ultimate generation of clock ticks is controlled by the device named "Tick Gen" in the architectural TG-Alg schematic. This *tick generation unit* has to ensure that the concurrently evaluated rules ("Relay Rule" and "Increment Rule") do not interfere with each other in an unintended way during tick generation. In fact, correct execution has to be ensured even though "old" clock ticks are still arriving asynchronously and possibly influence the remote counters' values.

The architecture described above and represented in Figure 3.2 handles incoming ticks by increasing a counter value with every clock transition. The strategy of counting absolute tick numbers for every tick received from a remote node would thus require maintaining infinitely large numbers. To get rid of this demanding requirement, Figure 3.3 presents an enhancement of the counter design. The characteristic of the new counting approach

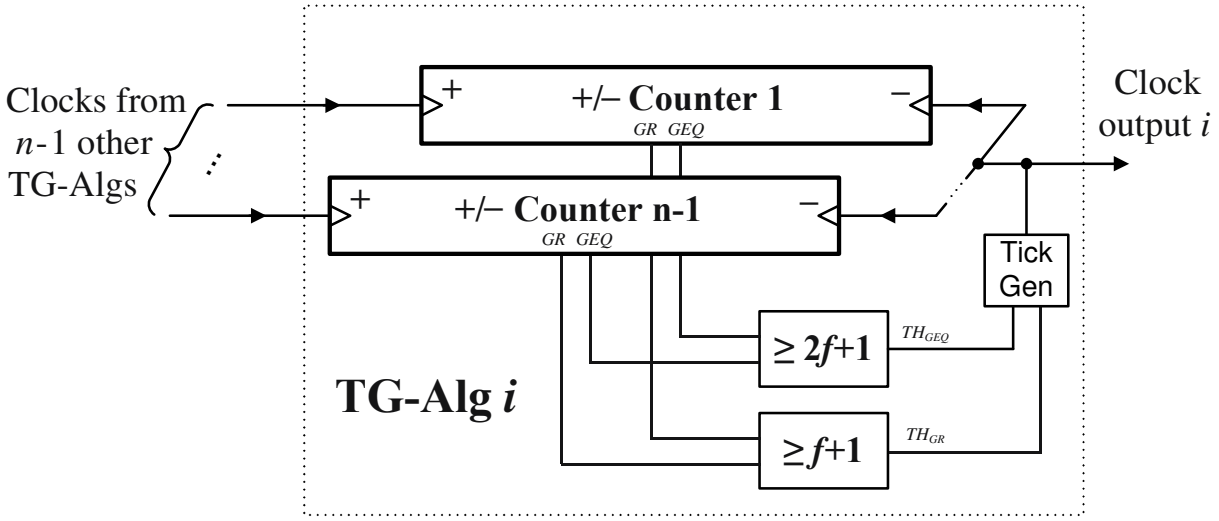


Figure 3.3: Tick generation architecture handling relative tick numbers

is that only the difference of locally and remotely issued TICK(↑)/TICK(↓) transitions is handled for every remote node. Hence, rather than working on absolute numbers only the difference of received and generated ticks has to be stored. This functionality is achieved by employing up/down counters, further referred to as +/− Counters, instead of the previously used incrementer[2]. Fortunately, taking a closer look at Algorithm 7 reveals that proper operation of the algorithm only requires the difference for the deviation of received and generated tick messages. Thus, each +/− Counter simply derives the difference of,

- the number of clock ticks seen from the respective remote node, and
- the number of clock ticks generated locally so far,

with the difference being bounded by precision π.

3.3 Hardware design considerations

The architecture described above and shown in Figure 3.3 appears to be quite simple and easy to map to a suitable hardware implementation. However, major challenges arise again when trying to find an appropriate asynchronous design for all presented modules. This is mainly due to the lack of a common control mechanism that could be used for the separation of actions not allowed to interfere with each other. In contrast to the synchronous approach with its single control signal — namely, the global clock — control mechanisms in asynchronous designs are known to be much more complicated. The asynchronous design

[2]For simplicity of the illustration the comparator blocks are not explicitly shown, but are considered to be part of the +/− Counters.

style of (quasi) delay insensitive circuits (recall Section 1.2.2) was preferred because of its inherent robustness against varying parameters and its conceptual elegance. However, a thorough examination of the algorithmic statements in conjunction with the hardware building blocks has to be performed to be able to come up with a suitable implementation for the architecture of Algorithm 7 depicted in Figure 3.3. Considering the requirement that the tick generation design should only convey simple, zero-bit TICK(↑)/TICK(↓) transitions, the natural and most appealing asynchronous design approach is given by transition signaling [43].

In *transition signaling* information is solely conveyed via signal transitions, rather than by issuing discrete state updates like in conventional logic. Hence, the expressiveness of transition signaling is limited to the causal ordering of events in a basically time-free system. However, to retain its delay insensitiveness the class of allowed circuit elements is fairly restricted. Permitted elementary units are for instance Muller C-Elements, inverters, XOR gates and a few rather complicated and quite exotic building blocks like the toggle unit [8, 57, 83]. Even simple logic operations have to be treated in a different way in the scope of transition signaling. As an example, an equivalent to the state logic's elementary two-input AND function in the context of transition signaling is represented by the Muller C-Element (in Chapter 4 a detailed description of the Muller C-Element is presented). In contrast to the logic AND, no direct equivalent for the logic OR function can be identified. This fact is of particular interest because it highlights the concept of only handling events instead of states in an asynchronous closed loop system. When considering the case of a two-input gate with either one or the other input being able to issue a transition, an XOR (exclusive OR) gate can be used to safely process this transition — as depicted in Figure 3.4(a), every input transition on the XOR gate leads to a transition on output z, thereby maintaining causality. On the contrary, it cannot be handled in a meaningful way in transition signaling if both inputs might be able — but do not need — to produce a related transition. The behavior of a logic OR, that is generating an output as soon as the first input event has occurred would thus destroy the causality relation of the late input with the issued output transition of the OR gate (cf. Figure 3.4(b)).

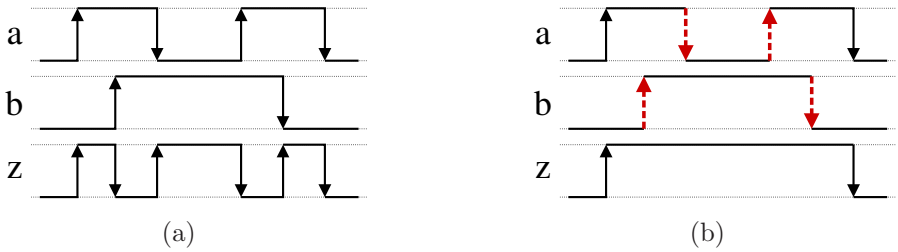


Figure 3.4: Transition processing of (a) XOR-gate and (b) OR-gate

3.3.1 Combining fault tolerance and transition signaling

When recalling the demand for fault tolerance it becomes evident that an OR-like behavior as it has been described above is required. The strategy to achieve fault tolerance incorporated by the previously introduced threshold circuits ($f+1$ and $2f+1$ block in Figure 3.3) is to ignore late inputs and issue new signals as soon as a sufficient number of signals have arrived. By applying this n-out-of-m voting it is ensured that erroneously late nodes cannot block the entire tick generation process. However, this concept clearly breaks the important causality relation for non-faulty, but late, nodes with the early generated output transitions. To allow for fault tolerance without ruining the causal relations of the asynchronous circuits, a switch back and forth from transition signaling and state logic has to be performed. The counting of TICK(k) (TICK($\uparrow$)/TICK($\downarrow$)) messages and the generation and transmission of exactly one TICK(k) event for any k can be handled in transition logic. On the other hand the comparators inside the $+/-$ Counters and threshold functions, which evaluate the current *state* of the counters, have to be implemented in state logic. An intricate design problem of the different building blocks is given by the clean switch between transition signaling and state logic. Asynchronous state logic tends to produce glitches on the outputs while processing its inputs (at least as long as no strict restriction on the input sequence is ensured [69]). Unfortunately, in transition signaling every signal change is treated as meaningful data. Therefore, hazardous input sequences of the *GR* and *GEQ* signals at the threshold circuits ($\geq f+1$ and $\geq 2f+1$ units) have to be circumvented by all means.

3.3.2 Glitch avoidance

The required glitch-free circuit behavior can be achieved by exploiting the strictly alternating sequence of the binary-valued TICK($\uparrow$) and TICK($\downarrow$) messages. In more detail, the distinction of *GR* and *GEQ* signals that already contributed to the generation of TICK(k), from *GR* and *GEQ* signals being conducive to create the consecutive TICK($k+1$) message is necessary. The separate treatment of low-to-high TICK($\uparrow$) and high-to-low TICK($\downarrow$) transitions is enabled by the introduction of two independent sets of *GR* and *GEQ* signals. One set is responsible for TICK($\uparrow$) transitions — further called *odd* clock ticks — while the other set of *GR* and *GEQ* signals treats TICK($\downarrow$) messages which are referred to as *even* clock ticks. Consequently, the independent signals represent the fill levels of the $+/-$ Counters for odd ($k \in \mathbb{N}_{odd} := 2\mathbb{N}+1$) and even ($k \in \mathbb{N}_{even} := 2\mathbb{N}$) clock ticks and are evaluated separately by distinct threshold circuits. Figure 3.5 depicts the resulting architecture with the individual signals GR^e, GEQ^e and GR^o, GEQ^o for handling even and odd ticks, respectively.

The presented separation of even and odd ticks enables that output glitches due to asynchronously arriving input signals of the asynchronous threshold circuits can be masked. As stated before, this glitch-free behavior is mandatory to protect the subsequent transition signaling building blocks from spurious transitions. The actual masking operation is per-

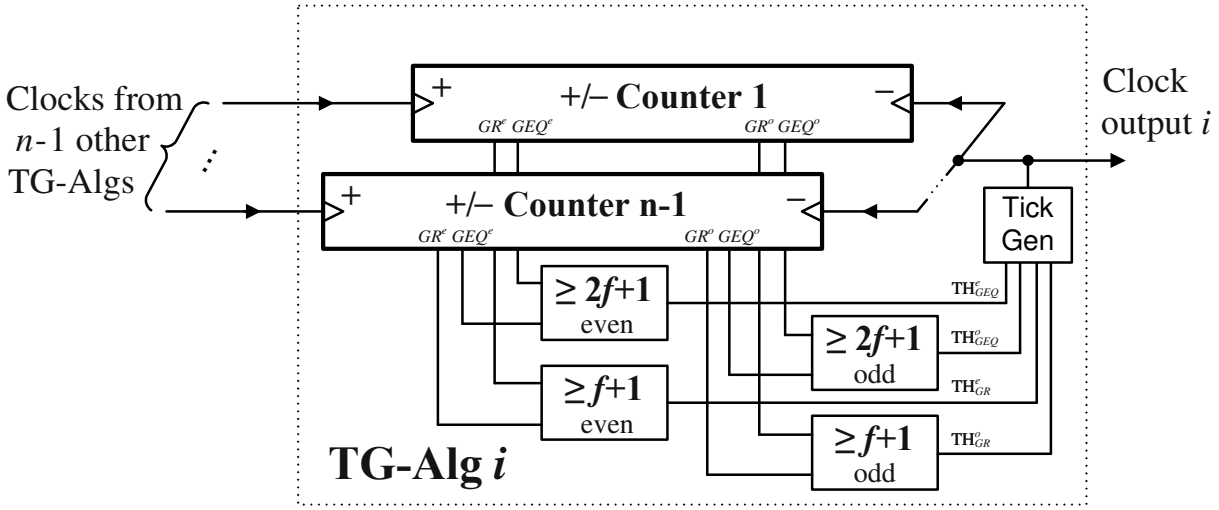


Figure 3.5: Tick generation design with separate treatment of even and odd ticks

formed by simply ignoring the outputs of the threshold circuits which have generated the last clock tick. For example the signals TH^o_{GR} and TH^o_{GEQ} which initiated the generation of the even TICK(k) can be safely ignored when the next tick to be generated, TICK($k+1$), is odd. However, as soon as this odd TICK($k+1$) has been issued, glitches in the signals TH^o_{GR} and TH^o_{GEQ} are prohibited again. This "gap" in the significance of the TH^o_{GR} and TH^o_{GEQ} signals allows all underlying GR^o and GEQ^o signals to become inactive again. Note that the GR^o and GEQ^o signals which activated TH^o_{GR} and/or TH^o_{GEQ} to generate TICK(k) have to get inactive during TICK($k+1$) and will afterwards become active again to trigger the generation of TICK($k+2$). This masking functionality is implemented by the tick generation (tick-gen) block in Figure 3.5. Moreover, the tick generation unit performs the switch from asynchronous state logic (the threshold circuits) back to transition signaling. Of course, due to symmetry of even and odd ticks the presented strategy for treating TH^o_{GR} and TH^o_{GEQ} which generated the even TICK(k), can be applied also for threshold circuit outputs TH^e_{GR} and TH^e_{GEQ}.

3.3.3 Ensuring non-interference of subsequent ticks

The architecture for avoiding glitches also enables to cope with the ambiguity of the *GR* and *GEQ* signals for generating successive clock ticks and hence ensures that no spurious TICK(k) messages are generated. The modifications to the basic architecture introduced above and depicted in Figure 3.3 enhanced the initially (in Section 3.2) presented operation of the tick generation process to the one described below:

odd ticks: The generation of an odd tick $k+1 \in \mathbb{N}_{odd} := 2\mathbb{N}+1$ is triggered only if the last tick generated was even ($k \in \mathbb{N}_{even} := 2\mathbb{N}$). Furthermore, it has to hold that the respective node has either received (i) the same or a greater number of ticks from at

least $2f+1$ TG-Algs (GEQ^e true via "Relay Rule"), or (ii) a greater number of ticks have been received from at least $f+1$ TG-Algs (GR^e true via "Increment Rule"). Note that (i) ensures that the even TICK(k) has in any case been seen from at least $2f+1$ TG-Algs, whereas (ii) guarantees that the odd TICK(k+1) has already been seen from at least $f+1$ nodes.

The treatment of even/TICK($\downarrow$) and odd/TICK($\uparrow$) is symmetric. Hence, analogous rules involving GEQ^o and GR^o instead of GEQ^e and GR^e can be applied for generating the even tick $k+1 \in \mathbb{N}_{even}$.

even ticks: The generation of an even tick $k+1 \in \mathbb{N}_{even}$ is triggered only if the last tick generated was odd ($k \in \mathbb{N}_{odd}$). Furthermore, it has to hold that the respective node has either received (i) the same or a greater number of ticks from at least $2f+1$ TG-Algs (GEQ^o true via "Relay Rule"), or (ii) a greater number of ticks have been seen from at least $f+1$ TG-Algs (GR^o true via "Increment Rule"). Again, condition (i) ensures that the odd TICK(k) has been seen from at least $2f+1$ TG-Algs, whereas (ii) guarantees that the even TICK(k+1) has already been received by at least $f+1$ nodes.

A thorough analysis of the described tick generation process, conducted in the context of the DARTS project and published in [39], shows that the presented approach is sufficient to avoid that old and new instances of GR^o and GEQ^o get mixed up. Parts of this formal analysis — deriving timing constraints for the hardware implementations — will be introduced later on in this chapter in Section 3.4.3 and 3.4.4.

3.3.4 Count and compare

The considerations above treated the glitch-free tick generation via state logic threshold circuits and involved the switch from state logic back to transition signaling. What is still missing in the detailed analysis of the tick generation algorithm's refined architecture is a more thorough analysis of the $+/-$ Counters. The fact that all issued clock TICK($\uparrow$) and TICK($\downarrow$) transitions are processed by $+/-$ Counters emphasizes the importance of the counters as vital parts of the architecture. The purpose of the $+/-$ Counters as depicted in Figure 3.5 is twofold. At first, each up/down counter stores incoming ticks and derives the difference of the number of clock ticks received from its respective remote node, and the number of clock ticks generated locally so far. The second part of the $+/-$ Counter design is devoted to the generation of individual signals for even and odd clock ticks indicating the counter fill levels. Therefore, this part of the design performs the translation from transition signaling to state logic. More precisely, the $+/-$ Counters have to create the threshold circuit's state input signals GR^o, GEQ^o and GR^e, GEQ^e. The challenging part of designing an asynchronous up/down counter is given by the fact that the TICK($\uparrow$) and the TICK($\downarrow$) transitions can occur arbitrarily close to each other. Fortunately, an implementation based on the well-known transition signaling elastic pipeline approach by

Sutherland [83] can be employed here. The resulting +/− Counter design uses two elastic pipelines per remote node to buffer clock transitions. Additionally, the pipelines have to be augmented by interconnecting logic to provide the needed up/down counting functionality. In fact it has to enable the removal of already processed clock ticks. Following this strategy provides a suitable approach for the counting part of the +/− Counter. Most notably, metastability problems inside the counters where local and remote clock ticks encounter each other are circumvented by this design. The avoidance of metastability issues relies on the employed tick removal process which follows a strict execution sequence. At first a tick is removed from the remote pipeline and only after its removal has been acknowledged enables tick processing at the local side. The second part of every +/− Counter is responsible for creating the counter's status signals GR^o, GEQ^o and GR^e, GEQ^e. Hence, it has to provide correct counter fill level, at least during times when they are used by the respective threshold circuits. For instance, GR^o and GEQ^o have to be valid if the last tick generated has been odd, since the next even tick to be issued will be triggered by TH^o_{GR} or TH^o_{GEQ}. Of course the same applies for GR^e, GEQ^e in conjunction with TH^e_{GR} or TH^e_{GEQ} for generating an odd tick. The generation of the counter fill level signals benefits from the above introduced remote-before-local tick removal because the immediate decrementing of the remote counter avoids wrongly activated GR^o, GEQ^o and GR^e, GEQ^e signals due to local clock ticks arriving late.

3.4 Refined tick generation algorithm

The quite extensive adaptations to the underlying tick generation approach of Algorithm 7 with the reduction of integer tick numbers to zero-bit TICK(↑)/TICK(↓) messages led to the architecture depicted in Figure 3.3. Further refinements to this architecture have been conducted by separating the status signals provided by the +/− Counters for even and odd (TICK(↓)/TICK(↑)) clock ticks. As a consequence, the threshold circuits ("Relay Rule" and "Increment Rule") had to be duplicated to be able to separately evaluate the GR^o, GEQ^o and GR^e, GEQ^e signals, respectively. The adaptations of the hardware architecture yielded the design depicted in Figure 3.5. To reassess whether or not the derived architecture still implements the demanded fault-tolerant tick generation, an algorithm more detailed than Algorithm 7 has to be derived for the analysis. However, first of all, a system model suitable of reflecting the asynchronous transition signaling and state logic design units has to be defined. For this purpose the respective system model has to account for the fact that even the simplest sequential control flow comes with some delay, since it actually involves sending a signal over a wire. Thus, a wire can for instance be treated as a zero-bit first-in-first-out (FIFO) message channel. Relying on this more elaborate system model allows to derive and analyze an algorithm that more accurately reflects the TG-Alg's asynchronous hardware design units. Moreover, this formal treatment allows to derive conditions and properties under which the correctness of the implementation can be guaranteed.

3.4.1 Signals and zero-bit message channels

To reflect the design decisions made so far, all components of the tick generation algorithm considered from here on are solely implemented in asynchronous digital logic and deal with binary signals only. Such a signal S may represent the possible values $\bot$ and $\top$, denoting logical *low* (=*false, inactive*) and logical *high* (=*true, active*), respectively. An event on signal S, for instance, a state transition $S{-}\uparrow\ (t^*)$, occurs when S changes its state from $\bot$ to $\top$ at time t^*. Similarly, an $S{-}\downarrow\ (t^*)$ state transition happens when S changes its state from $\top$ to $\bot$ at time t^*. The status $S(t)$ of a signal S at time $t \geq t^*$ is $S(t) = \bot$ if, and only if, the last event at or before t was $S{-}\downarrow\ (t^*)$. Again, by analogous means the status $S(t)$ of a signal S at time $t \geq t^*$ is $S(t) = \top$ if, and only if, the last event at or before t was $S{-}\uparrow\ (t^*)$.

In the treatment of the refined tick generation approach events/transitions on, and the status/state of, binary signals will be used. To keep the notation as simple and clear as possible the convention is employed that depending on the respective context $S(t)$ will denote either:

- the status of signal S with $S(t) \in \{\bot, \top\}$, where t denotes the observation time, or
- the event on signal S with either $S{-}\uparrow\ (t)$ or $S{-}\downarrow\ (t)$, where t denotes the time of the last transition to the active state $\top$ or $\bot$, respectively.

As indicated before, all components of the tick generation system are interconnected by simple signal wires. The wires are modeled as reliable FIFO channels with finite delay that carry zero-bit messages. The semantics of a zero-bit message channel X is as follows: Let X^s be the channel's input signal, which is controlled by a single sender component. The sender generates the events/transitions $X^s{-}\uparrow\ (t)$ and $X^s{-}\downarrow\ (t)$, where t denotes the sending time. The content of these messages is defined by the symbols $\uparrow$ and $\downarrow$. Additionally, the associated input state $X^s(t)$ can be viewed as the information content of the last message sent into X. Furthermore, the output of channel X is fed into a receiver, which perceives the respective messages for every sent transition. In more detail, every message $X^s{-}\uparrow\ (t)$ and $X^s{-}\downarrow\ (t)$ transmitted by a sender via channel X is received within finite time $t' \geq t$ as $X^r{-}\uparrow\ (t')$ and $X^r{-}\downarrow\ (t')$, respectively. Analogous to the input state $X^s(t)$, the receiving state $X^r(t)$ at time t can be viewed as the message content of the last received transition. To provide the basis for a clean startup (time t_0) the channel state $X^r(t_0)$ is initialized to $\bot$.

Obviously, the zero-bit message channels employed here can only convey messages with strictly alternating content. Nevertheless, this type of communication is sufficient for analyzing the developed tick generation approach. It can be further noted that zero-bit channels are compatible with Lamport's happened-before $\rightarrow$ relation [51]. That is, for matching send and receive events, it holds that $X^s{-}\uparrow\ (t) \rightarrow X^r{-}\uparrow\ (t')$ and $X^s{-}\downarrow\ (t) \rightarrow X^r{-}\downarrow\ (t')$. However, to simplify the notation when using a channel X, the employed notation is adapted in the way that $X{-}\uparrow\ (t)$ and $X{-}\downarrow\ (t)$ abbreviate the send events

$X^s{-}\uparrow\ (t)$ and $X^s{-}\downarrow\ (t)$, respectively, whereas $X(t)$ abbreviates the state $X^r(t)$ at the receiving side of the channel.

3.4.2 Component and architecture specification

A last refinement step applying the system model presented above to the tick generation architecture derived in Section 3.3 and depicted in Figure 3.5 will be conducted in this section. This further detailing of the design components has the purpose to finally yield:

- a fault-tolerant tick generation architecture being fully implementable in asynchronous hardware, and
- an algorithmic representation of this architecture that models with sufficient accuracy the hardware's peculiarities to be able to derive correctness and performance measures for the design.

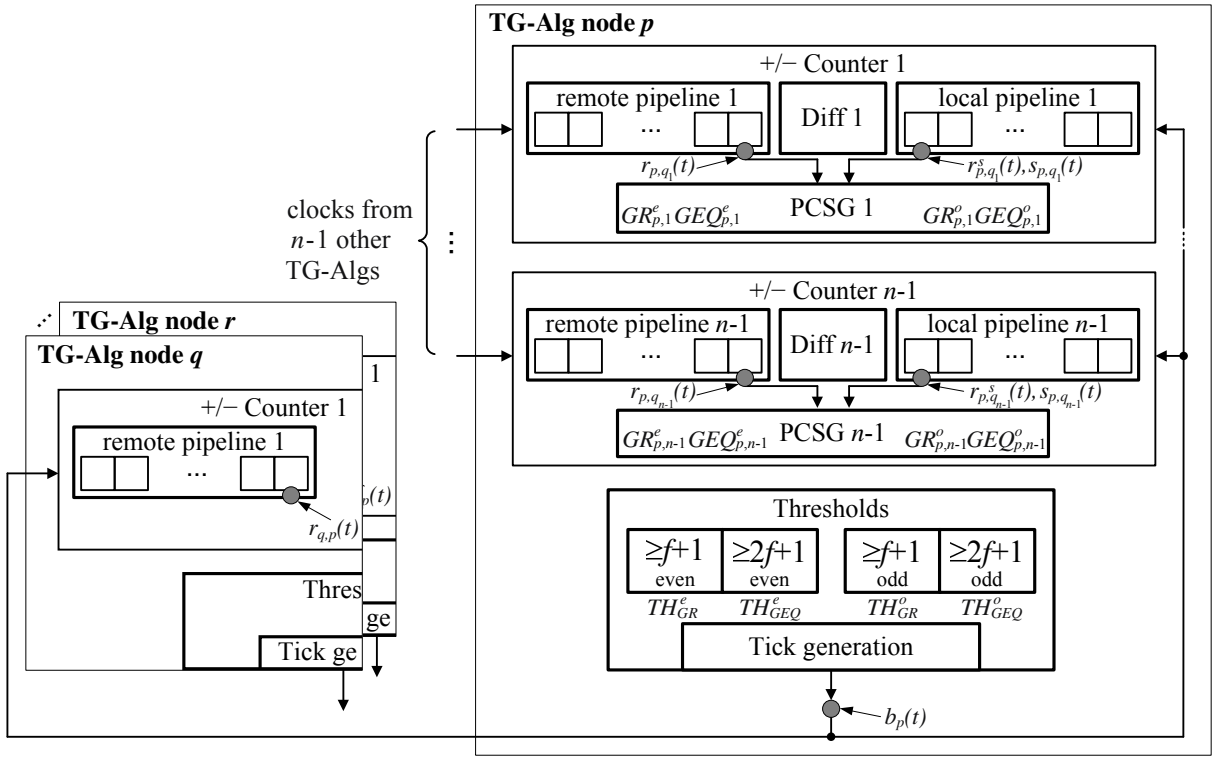


Figure 3.6: TG-Alg architecture including observation points

The schematics in Figure 3.6 present the components of a refined TG-Alg instance. Moreover, important observation points for the formal analysis and the computation of performance measures are also shown in these schematics. In this more detailed treatment

the +/− Counters of Figure 3.5 are partitioned into several submodules. The respective building blocks of the +/− Counter are the *remote pipeline, local pipeline, Diff* and *PCSG* design units. These modules together with the *Thresholds* block are accurately characterized below using the previously introduced modeling approach of Section 3.4.1.

Pairs of remote and local pipelines: In a set P of n distributed nodes, every TG-Alg node p incorporates $n-1$ pairs of elastic pipelines where each of these pipes can be seen as a FIFO shift register for signal transitions [83]. Moreover, every pair of remote and local pipelines corresponds to a dedicated remote TG-Alg node $q \in P \setminus \{p\}$. The remote pipeline can store up to S_{rem} messages TICK(↑)/TICK(↓) sent by node q, and similarly the local pipeline can hold up to S_{loc} messages TICK(↑)/TICK(↓) generated and sent by node p locally. The numbers S_{rem} and S_{loc} represent implementation-dependent parameters that have to be chosen in accordance with some bounds derived from the formal analysis of the algorithm (presented in Section 3.4.4).

To enable an accurate description of the TG-Alg's architecture and to be able to build and analyze a detailed algorithm, some additional terms have to be defined: $r_{p,q}(t)$ and $r^s_{p,q}(t)$ denote the number of messages that arrived at the end of the remote and local pipe by time t, respectively. Moreover, $s_{p,q}(t)$ represents the number of tick messages stored in the local pipeline at time t. However, from the algorithm's point of view those quantities are not directly available. Therefore, the algorithm uses the respective binary status signals which are generated following the rules given below:

- $r_{p,q}(t) \geq r^s_{p,q}(t)$: more than or at least as many ticks as locally generated have been received from the remote side (this enables the activation of *GEQ* signals), and
- $r_{p,q}(t) > r^s_{p,q}(t)$: more ticks than locally generated have been received from the corresponding remote node (corresponding to the *GR* signals), and these conditions are treated in conjunction with the clause that,
- $s_{p,q}(t) = 1$: all stages of the local pipeline hold the same value.

Additionally, all pipelines are initialized in a way that each contains exactly one single even TICK(↓) message upon startup/reset. It is further assumed that the tick counting variables $r_{p,q}(t_{0,p})$ and $r^s_{p,q}(t_{0,p})$ are initialized to 0 and the local pipeline's fill level indicator $s_{p,q}(t_{0,p}) = 1$ at reset time $t_{0,p}$

Difference Module: A Diff block of Figure 3.6 represents the design unit that rests in-between every pair of remote and local pipelines and interconnects both with each other. The purpose of this interconnecting block is to detect whether or not matching TICK(↑)/TICK(↓) messages are in both pipelines and to remove such ticks from both. This is necessary to avoid the need for infinite storage for local and remote pipelines. The exact behavior of a Difference Module is as follows:

If $r_{p,q}(t) \geq r^s_{p,q}(t) \wedge s_{p,q}(t) > 1$, there is some $t' \in t + [\tau^-_{Diff}, \tau^+_{Diff}]$ such that $s_{p,q}(t') = s_{p,q}(t' - dt) - 1$, for some infinitesimally small $dt > 0$. In other words, the removal

of a matching tick from the local as well as the remote pipeline is issued after the condition holds that more than ("Relay Rule") or at least as many ("Increment Rule") TICK($\uparrow$)/TICK($\downarrow$) messages have been remotely received as locally generated. It has to hold further that at least one TICK($\uparrow$)/TICK($\downarrow$) message is stored inside the local pipeline. Note that due to this removal of "common" tick messages no information is lost, since the underlying tick generation algorithm is only interested in the difference of the number of messages received so far. It is important to notice that ticks can only be removed successively, i.e., one after another and following the before introduced remote-before-local strategy. This subsequent tick removal might, however, lead to malicious queueing effects in a node's pipelines if ticks arrive at a higher rate than the Difference Module's removal speed.

Pipeline Compare Signal Generation Modules (PCSGs): The remaining part of the $+/-$ Counters is given by the PCSG units which are directly connected to each pair of remote and local pipelines.

The Pipeline Compare Signal Generation Module's function can be partitioned in the processing of even and odd ticks. The PCSG part treating incoming even ticks ultimately triggers the generation of odd ticks by issuing GEQ^e, GR^e signals. Similarly, the circuit concerned with odd ticks and controlling GEQ^o and GR^o is responsible for generating even clock ticks. In more detail, the PCSG's detection circuit generates the status signals $GEQ^o_{p,q}(t)$, $GR^o_{p,q}(t)$ and $GEQ^e_{p,q}(t)$, $GR^e_{p,q}(t)$ for odd and even clock ticks, respectively.

In particular, $GEQ^o_{p,q}(t')$ becomes active, that is, $GEQ^o_{p,q}(t') = \top$, thereby generating the event $GEQ^o_{p,q}{-}\uparrow$ at t' at some time $t' \in t + [\tau^-_{GEQ}; \tau^+_{GEQ}]$ (if the previous state was $\bot$) when

- $r^s_{p,q}(t) \in \mathbb{N}_{odd}$: the number of locally received tick messages is odd, and
- $[r_{p,q}(t) \geq r^s_{p,q}(t)] \wedge [s_{p,q}(t) = 1]$: at least as many TICK($\uparrow$)/TICK($\downarrow$) messages have been remotely received as locally generated and additionally, exactly one tick is left inside the local pipeline.

Similarly, $GR^o_{p,q}(t')$ becomes active at time $t' \in t + [\tau^-_{GR}; \tau^+_{GR}]$ when

- $r^s_{p,q}(t) \in \mathbb{N}_{odd}$: the number of locally received tick messages is odd, and
- $[r_{p,q}(t) > r^s_{p,q}(t)] \wedge [s_{p,q}(t) = 1]$: more TICK($\uparrow$)/TICK($\downarrow$) messages have been remotely received than locally generated and again, only a single tick message is left inside the local pipeline.

The signals $GEQ^e_{p,q}(t)$ and $GR^e_{p,q}(t)$ for handling even TICK() messages have the same definition, except that the first condition is given by $r^s_{p,q}(t) \in \mathbb{N}_{even}$ denoting the case of even ticks.

Threshold Module: If the number of active $GEQ^o_{p,q}(t)$ or $GR^o_{p,q}(t)$ signals exceeds the respective $2f + 1$ or $f + 1$ threshold, the corresponding threshold output signal

$TH^o_{GEQ}(t)$, $TH^o_{GR}(t)$ will become activated within $[\tau^-_{TH};\tau^+_{TH}]$. By analogous means the activation of a sufficient number of $GEQ^e_{p,q}(t)$ or $GR^e_{p,q}(t)$ signals will trigger $TH^e_{GEQ}(t)$, respectively $TH^e_{GR}(t)$, to become active within $[\tau^-_{TH};\tau^+_{TH}]$. Due to the fact that $TH^e_{GEQ}(t)$, $TH^e_{GR}(t)$ and $TH^o_{GEQ}(t)$, $TH^o_{GR}(t)$ will be deactivated as soon as the number of active inputs is below the respective threshold value the threshold functions can rely on a purely combinatorial design, i.e., no hysteresis is required.

Tick Generation Module: As part of the *Thresholds* block in Figure 3.6, the *Tick Generation Module* is responsible for ultimately generating and broadcasting TICK($\uparrow$)/ TICK($\downarrow$) messages if indicated by the $TH^o_{GR}(t)/TH^o_{GEQ}(t)$ and $TH^e_{GR}(t)/TH^e_{GEQ}(t)$ signals with $b_p(t)$ denoting the number of ticks generated by node p by time t. In the context of the tick generation it has to be noted again that potential glitch phenomena of asynchronous state logic as it is used for the threshold circuits cannot be entirely avoided in an asynchronous implementation. Hence, a strict activation pattern for generating and broadcasting ticks has to be followed. The generation of an odd TICK($\uparrow$) is triggered only if,

- both threshold signals for the previously generated tick TICK($\downarrow$), TH^o_{GEQ} and TH^o_{GR} are inactive again, and
- at least one threshold signal TH^e_{GEQ} or TH^e_{GR} for the current tick becomes active.

The generation of the even TICK($\downarrow$) follows by analogous means and rests upon the deactivation of TH^e_{GEQ} and TH^e_{GR}. It further requires that at least one signal TH^o_{GEQ} or TH^o_{GR} gets activated.

The refinement process applied to the initial tick generation approach of Algorithm 7 led to the architecture depicted in Figure 3.5 and Figure 3.6. Furthermore, the detailed description of the algorithm's building blocks given above and the application of the system model from Section 3.4.1 allows to introduce Algorithm 8. This algorithm further narrows down the gap between the abstraction of hardware modeling and algorithmic design. Hence, this algorithm is used as a basis for both the asynchronous hardware design as well as the formal analysis of the tick generation approach.

Algorithm 8 Refined TG-Alg reflecting the asynchronous VLSI building blocks

1: **variables**
2: $\forall q : r_{p,q}(t_{0,p}) = r^s_{p,q}(t_{0,p}) = 0$; $s_{p,q}(t_{0,p}) = 1$; $\forall channels\, X : X^r(t_{0,p}) = \perp$
3: **end variables**
//Generation of comparison signals (PCSG) for remote process q
4: **if** $[r_{p,q}(t) \geq r^s_{p,q}(t)] \wedge [r^s_{p,q}(t) \in \mathbb{N}_{odd}] \wedge [s_{p,q}(t) = 1]$ **then** //treatment of odd ticks
5: $\rightarrow$ send $GEQ^o_{p,q_i}(t)-\uparrow$
6: **else**
7: $\rightarrow$ send $GEQ^o_{p,q_i}(t)-\downarrow$
8: **end if**
9: **if** $[r_{p,q}(t) > r^s_{p,q}(t)] \wedge [r^s_{p,q}(t) \in \mathbb{N}_{odd}] \wedge [s_{p,q}(t) = 1]$ **then** //treatment of odd ticks
10: $\rightarrow$ send $GR^o_{p,q_i}(t)-\uparrow$
11: **else**
12: $\rightarrow$ send $GR^o_{p,q_i}(t)-\downarrow$
13: **end if**
14: **if** $[r_{p,q}(t) \geq r^s_{p,q}(t)] \wedge [r^s_{p,q}(t) \in \mathbb{N}_{even}] \wedge [s_{p,q}(t) = 1]$ **then** //treatment of even ticks
15: $\rightarrow$ send $GEQ^e_{p,q_i}(t)-\uparrow$
16: **else**
17: $\rightarrow$ send $GEQ^e_{p,q_i}(t)-\downarrow$
18: **end if**
19: **if** $[r_{p,q}(t) > r^s_{p,q}(t)] \wedge [r^s_{p,q}(t) \in \mathbb{N}_{even}] \wedge [s_{p,q}(t) = 1]$ **then** //treatment of even ticks
20: $\rightarrow$ send $GR^e_{p,q_i}(t)-\uparrow$
21: **else**
22: $\rightarrow$ send $GR^e_{p,q_i}(t)-\downarrow$
23: **end if**
24: **if** $GEQ^o_{p,q_i}(t)$ for at least $2f+1$ remote processes q_i **then** //odd Increment Rule
25: $\rightarrow$ send $TH^o_{GEQ}-\uparrow$
26: **else**
27: $\rightarrow$ send $TH^o_{GEQ}-\downarrow$
28: **end if**
29: **if** $GR^o_{p,q_i}(t)$ for at least $f+1$ remote processes q_i **then** //odd Relay Rule
30: $\rightarrow$ send $TH^o_{GR}-\uparrow$
31: **else**
32: $\rightarrow$ send $TH^o_{GR}-\downarrow$
33: **end if**
34: **if** $GEQ^e_{p,q_i}(t)$ for at least $2f+1$ remote processes q_i **then** //even Increment Rule
35: $\rightarrow$ send $TH^e_{GEQ}-\uparrow$
36: **else**
37: $\rightarrow$ send $TH^e_{GEQ}-\downarrow$
38: **end if**
39: **if** $GR^e_{p,q_i}(t)$ for at least $f+1$ remote processes q_i **then** //even Relay Rule
40: $\rightarrow$ send $TH^e_{GR}-\uparrow$
41: **else**
42: $\rightarrow$ send $TH^e_{GR}-\downarrow$
43: **end if**
44: **if** $[TH^o_{GR}(t) \vee TH^o_{GEQ}(t)] \wedge \neg[TH^e_{GR}(t) \vee TH^e_{GEQ}(t)]$ **then** //generate tick$-\downarrow$ messages
45: $\rightarrow$ send tick$-\downarrow$
46: **end if**
47: **if** $[TH^e_{GR}(t) \vee TH^e_{GEQ}(t)] \wedge \neg[TH^o_{GR}(t) \vee TH^o_{GEQ}(t)]$ **then** //generate tick$-\uparrow$ messages
48: $\rightarrow$ send tick$-\uparrow$
49: **end if**

3.4.3 Timing constraints

As already noted above, the correct behavior of the tick generation approach of Algorithm 8 relies on some particular timing constraints. In fact, implementation-specific constraints on path delays have to hold. The most important one is given by the *Interlocking Constraint* that ensures that TICK(k) and TICK(k+2) messages do not interfere with each other.

Constraint 3.4.1 (Interlocking) $T_{max,dis} \leq T_{min} + T_{min,dis}$ *must hold.*

With the delay paths:

$$\begin{aligned} T_{max,dis} &:= \tau^{+}_{TH} + \max(\tau^{+}_{GR}, \tau^{+}_{GEQ}) + \tau^{+}_{loc} \\ T_{min} &:= \tau^{-}_{TH} + \min(\tau^{-}_{GR}, \tau^{-}_{GEQ}) + \tau^{-}_{loc} + \tau^{-}_{Diff} \\ T_{min,dis} &:= \tau^{-}_{TH} + \min(\tau^{-}_{GR}, \tau^{-}_{GEQ}) + \tau^{-}_{loc} \end{aligned} \tag{3.1}$$

$T_{max,dis}$ represents the slowest disabling path starting and ending at the tick generation output of the respective node. The second part of Constraint 3.4.1 is defined by T_{min} and $T_{min,dis}$. T_{min} corresponds to the fastest path for generating a clock tick, whereas $T_{min,dis}$, analogously to $T_{max,dis}$, accounts for the minimum deactivation time of the previous clock tick which in turn enables the generation of the next clock tick. Figure 3.7 graphically presents a TG-Alg node's opposing interlocking delay paths at the architectural abstraction level of Figure 3.6. Note that the involved paths only include design units at a local node. This locality of Constraint 3.4.1 considerably facilites to design the path delays accordingly.

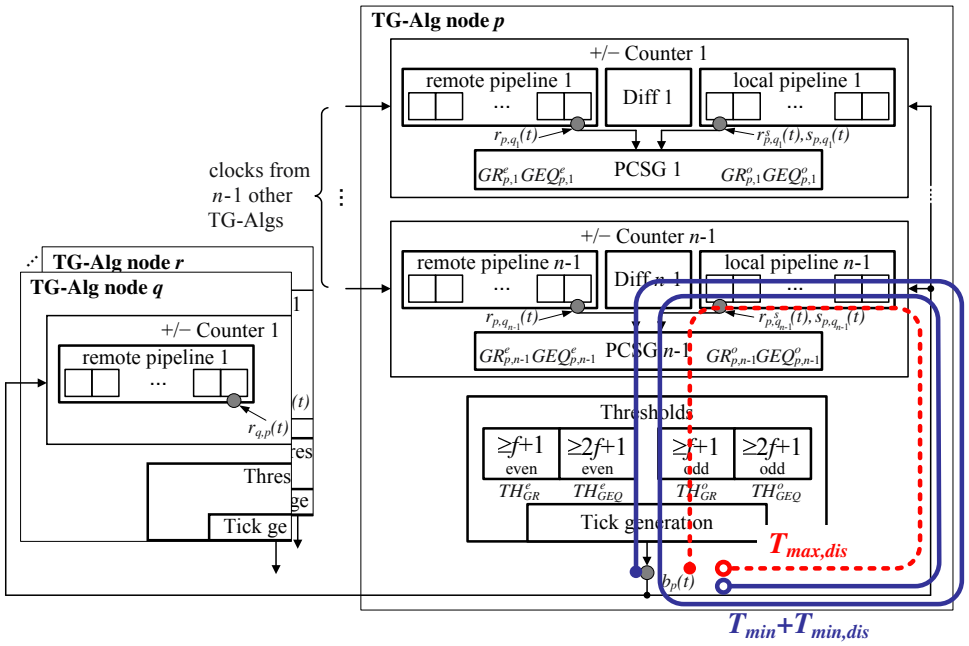


Figure 3.7: Timing paths of the interlocking constraint

Constraint 3.4.2 (Tick removal) $\tau^{+}_{Diff} \leq T_{min}$

Recall the functionality of the Difference Module which is to remove matching ticks from both related pipelines, i.e., the remote as well as local one. To ensure that the speed of this tick removal does not influence the rate of tick generation, Constraint 3.4.2 has to hold. Informally speaking, the slowest Difference Module's processing of TICK(k) (determined by the delay τ^{+}_{Diff}) has to be at least as fast as the fastest generation of the subsequent TICK($k+1$) and its propagation to the respective Difference Module (given by T_{min}). In other words, the Difference Module is not allowed to cause malicious queueing of ticks.

Constraint 3.4.3 (Fastest progress) $T^{-}_{first} \geq T^{+}_{loc}$

The involved paths are depicted in Figure 3.8 and defined as follows:

$$T^{+}_{loc} := \tau^{+}_{loc} + \max\{\tau^{+}_{Diff} + \tau^{+}_{GR}, \tau^{+}_{GEQ}\} + \tau^{+}_{TH}$$
$$T^{-}_{first} := \tau^{-}_{rem} + \tau^{-}_{Diff} + \tau^{-}_{GEQ} + \tau^{-}_{TH}$$

The condition of Constraint 3.4.3 expresses the requirement that the fastest correct node of the tick generation ensemble is not allowed to issue ticks arbitrarily fast. To be able to achieve and retain synchrony, the above constraint on the fastest remote tick generation and the slowest local processing of ticks has to hold. The constraint on the fastest progress denotes, besides the restrictions for booting, the only non-local constraint of the tick generation approach.

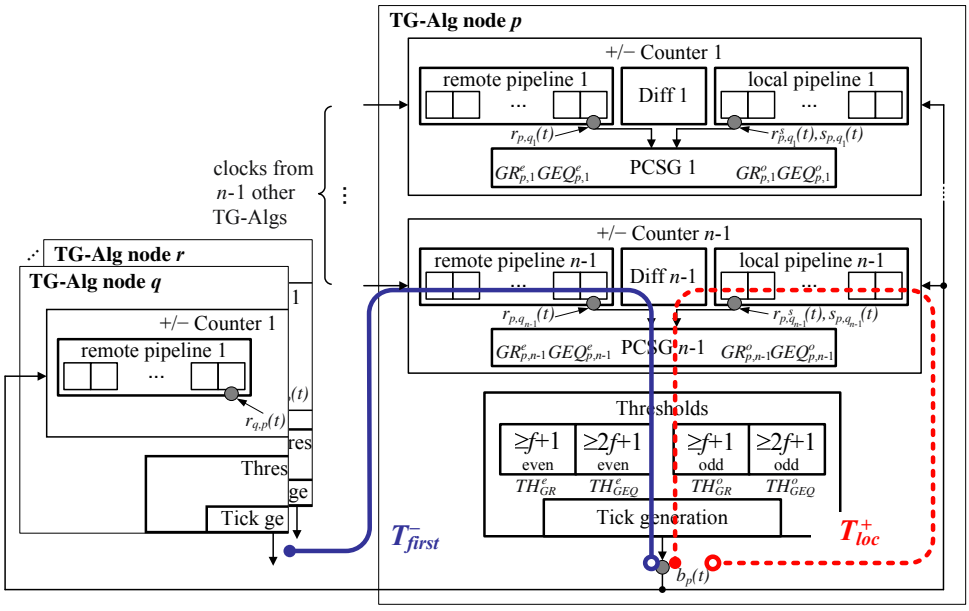


Figure 3.8: Timing paths of the synchronization constraint

Constraint 3.4.4 (Booting) $B \leq \tau^-_{rem}$

All correct nodes have to complete booting until time $t_{p,b}$ where $t_{p,b}$ is in the interval $[0, B]$ with $B \leq \tau^-_{rem}$. This constraint ensures that no clock tick is lost at start-up since no correct tick may arrive earlier than the fastest remote interconnection delay τ^-_{rem}.

3.4.4 Correctness and performance measures

As indicated above, the correct behavior of the tick generation approach relies on several path delay conditions. These constraints have been retrieved by formal proofs[3]. To be able to conveniently compute general synchronization characteristics like precision π and accuracy, the underlying synchronization properties *Progress*, *Unforgeability*, *Quasi-Simultaneity* and *Booting-Simultaneity* have been derived.

Progress (P): If all correct processes send TICK(k) by time t, then every correct process sends at least TICK(k+1) by time $t + T_P$, with

$$T_P := \max\{\tau^+_{loc} + \tau^+_{Diff} + \tau^+_{GEQ}, \tau^+_{loc} + \tau^+_{GR}, \tau^+_{rem} + \tau^+_{Diff} + \tau^+_{GEQ}\} + \tau^+_{TH}.$$

Unforgeability (U): If no correct process sends TICK(k) by time t, then no correct process sends TICK(k+1) by time $t + T^-_{first}$ or earlier.

Quasi-Simultaneity (QS): If some correct process p sends TICK(k+1) by time t, then every correct process (including p) sends at least TICK(k) by time $t + T_{QS}$, with

$$T_{QS} := \max\left\{ \begin{array}{l} (\tau^+_{rem} - \tau^-_{rem}) + (\tau^+_{GR} - \tau^-_{GEQ} - \tau^-_{Diff}), \\ B + (\max\{\tau^+_{GEQ}, \tau^+_{GR}\} - \min\{\tau^-_{GEQ}, \tau^-_{GR}\}) - T^-_{first} \end{array} \right\} + (\tau^+_{TH} - \tau^-_{TH}).$$

Booting-Simultaneity (BS): If some correct node sends TICK(k) by time t with $k \geq 1$, then every correct node sends at least TICK(k) by time $t + T_{BS}(k)$, with

$$\begin{aligned} T_{BS}(k) := B + \max\{\tau^+_{GEQ}, \tau^+_{GR}\} - \min\{\tau^-_{GEQ}, \tau^-_{GR}\} + \\ + (\tau^+_{TH} - \tau^-_{TH}) + (T_P - T^-_{first})(k-1). \end{aligned} \tag{3.2}$$

Based on these four properties, bounds for the synchronization characteristics precision π and accuracy can be given. Additionally, the queue sizes S_{loc} and S_{rem} of the local and remote pipelines can be computed.

Precision π among all correct nodes may not exceed

$$\pi := \left\lceil \frac{T_{QS}}{T^-_{first}} \right\rceil + 1 \tag{3.3}$$

[3]The detailed proofs have been published in [39].

Accuracy For a time interval given by t_1 and t_2 with $t_2 > t_1$, the accuracy (number of ticks generated in the given interval) $b_p(t_2) - b_p(t_1)$, of any correct node p is bounded by

$$\begin{aligned}&\max\left\{0, \left\lfloor \frac{t_2 - t_1 - \max\left\{T_{BS}(1), \min\left\{T_{BS}(k), T_{QS} + T_P\right\} \mid k \geq 2\right\}}{T_P} \right\rfloor\right\} \\ &\quad \leq b_p(t_2) - b_p(t_1) \leq \\ &\min\left\{\left\lceil \frac{t_2 - t_1}{T^-_{first}} \right\rceil + \pi, \left\lceil \frac{t_2 - t_1}{T_{min}} \right\rceil\right\}. \end{aligned} \tag{3.4}$$

Queue size The bound for the local and remote queue size of correct nodes can be derived by

$$S_{loc} := \max\left\{\left\lceil \frac{T_{del} - \tau^-_{loc}}{T^-_{first}} \right\rceil + 2, \left\lfloor \frac{T_{del} - \tau^-_{loc}}{T^-_{first}} \right\rfloor + 3\right\}. \tag{3.5}$$

$$S_{rem} := \max\left\{\left\lceil \frac{T^{loc}_{del} - \tau^-_{rem}}{T^-_{first}} \right\rceil + 2, \left\lfloor \frac{T^{loc}_{del} - \tau^-_{rem}}{T^-_{first}} \right\rfloor + 3, \left\lfloor \frac{\tau^+_{rem} + \tau^+_{Diff} - T^{loc}_{del} - \tau^-_{rem}}{T_{min}} \right\rfloor + 1\right\}. \tag{3.6}$$

with

$$\begin{aligned} T_{del} &:= T_{QS} + \max\{\tau^+_{loc}, \tau^+_{rem}\} + \tau^+_{Diff} \\ T^{loc}_{del} &:= T_{QS} + \tau^+_{loc} + \tau^+_{Diff}. \end{aligned}$$

The presented treatment of the correctness and performance measures concludes the formal analysis of the tick generation algorithms within this thesis. The following chapters are concerned with the detailed presentation of the hardware design and the experimental evaluations. However, it is self-evident that the insights gained from the formal analysis will contribute to the design decisions of the hardware design and will also form the basis for the evaluations.

Chapter notes

Within this chapter the mapping from algorithmic statements to design units implementable in asynchronous hardware has been presented. In the course of the translation process several challenging problems regarding the asynchronous handling of ticks had to be resolved which finally yielded a provably correct hardware architecture. For achieving and maintaining correct operation of the tick generation system several implementation

constraints have been identified. In this context the formally interested reader is referred to [39] where a proof outline of the presented correctness and performance measures can be found together with some results of the hardware implementation. A more detailed formal treatment of the tick generation approach can be found in [40]. This work includes the introduction of a novel, hardware-related system model and its application for proving correctness and performance characteristics. In [36] alternative approaches for conveying TICK(k) messages have been discussed quite extensively. In addition, limitations and constraints for the different schemes are pointed out in the course of this work. A detailed presentation of the hardware implementation resulting from the considerations of this chapter will be given subsequently.

Chapter 4

The DARTS ASIC Implementation

All truths are easy to understand once they are discovered;
the point is to discover them.

Galileo Galilei

The algorithmic design considerations regarding hardware outlined in Chapter 2 yielded a tick generation algorithm which is further detailed in Chapter 3 and finally presented as Algorithm 8. The development process which led to the architecture depicted in Figure 3.6 appears to be a straightforward approach taking into account predefined algorithmic and hardware design requirements and constraints. Unfortunately, the situation is much more complicated. The presumed sequential design refinements which finally resulted in the aforementioned algorithm and architecture had to follow a recurrent design approach with several iterations. More precisely the design on the formal/algorithmic level and on the layer of asynchronous hardware had to be iterated several times on both levels of abstraction. On the one hand, design requirements on the algorithmic level did not always allow for a suitable hardware implementation. On the other hand, asynchronous circuits being implementable with reasonable effort and performance required adaptations on the algorithmic level and recalculations of the formal analysis. Due to these tight mutual dependencies the mentioned feedback and refinement strategy between hardware and algorithm design had to be employed. The main focus in this chapter is on presenting the development process of the asynchronous hardware blocks of the TG-Alg ASIC implementation. Therefore, basic building blocks for composition of more complex parts of the circuit are introduced and characterized. Moreover, design alternatives for the TG-Alg's components are analyzed yielding the hardware blocks for the final ASIC implementation. Additionally, as already sketched in Chapter 2, failure models following weaker assumptions than the Byzantine case may provide substantial potential for simplifications and thus savings in terms of hardware resources. Implementations of weaker failure models

are studied to provide a quantitative assessment of the different approaches, but the main direction of the conducted research still focuses on Byzantine fault-tolerant design. In particular, a strategy relying on the duplication of nodes implementing weaker failure models together with some additional transformation schemes is presented in more detail, since this approach might be able to provide a viable implementation alternative to the purely Byzantine-tolerant approach.

4.1 The big picture

As the title of this section suggests the presentation of the TG-Alg's hardware design starts on a big scale before going into details of the respective sub-designs. The development process treated in Chapter 3 resulted in the architecture shown in Figure 3.6. This schematic representation was found to be sufficiently accurate to allow for a thorough formal analysis of the TG-Alg design, yielding several implementation constraints. Nevertheless, from a hardware designer's point of view the abstraction of the TG-Alg design has to undergo further steps of detailing to enable a successful mapping to an ASIC manufacturing process. Therefore, Figure 4.1 presents a more accurate architecture of a single TG-Alg node. The given schematic further details the architecture's representations of Figure 3.5 and Figure 3.6 by showing substantial implementation details. In particular, the transition-logic based remote and local elastic pipelines as well as the Difference Module are presented in detail. Additionally, the Pipe Compare Signal Generation (PCSG) block which translates the transition signaling events into state based counter fill-levels and the Tick Generation block which conversely performs the back transition to transition signaling are also shown on a more fine-grained level than before.

An important characteristic of the given TG-Alg design that has not been addressed so far is given by the fact that local clock ticks are compared only to remote ticks, i.e., no self-reception is included. This design decision is based on the fact that remote links are expected to incorporate much higher propagation delays than the local clock feedback. However, when recalling Section 2.3 it becomes evident that the synchronization precision largely depends on a constant Θ which is derived from the ratio of the fastest and slowest feedback delays within the tick generation scheme. Hence, an approach incorporating self-reception would lead to much more unbalanced delays and thus to precision degradation. As a consequence of omitting the self-reception path, the presented tick generation system has to comprise at least $3f + 2$ TG-Algs instead of the usually applied (lower bound of) at least $3f + 1$ nodes to attain the targeted degree of fault tolerance. In the schematic of Figure 4.1 the above argumentation is accounted for by providing input ports for $3f + 1$ remote TG-Algs. To give some real numbers for the ASIC design, a resilience for up to 3 Byzantine faults is demanded which results in a tick generation system of at least 11 TG-Alg nodes. The above presented architecture of the TG-Alg ASIC — in the context of the entire distributed tick generation system — is subsequently followed by a detailed description of the underlying submodules.

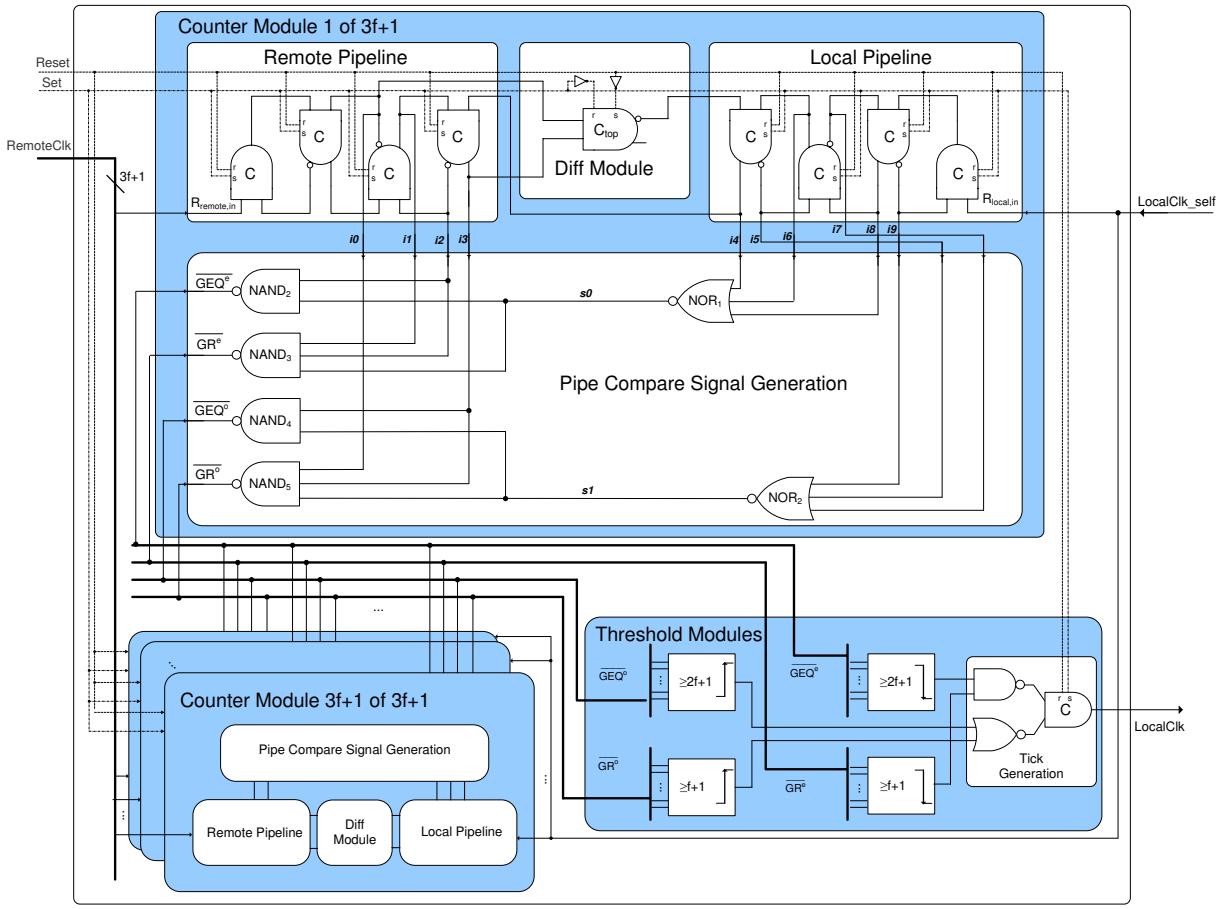


Figure 4.1: TG-Alg ASIC design architecture

4.2 Queueing ticks

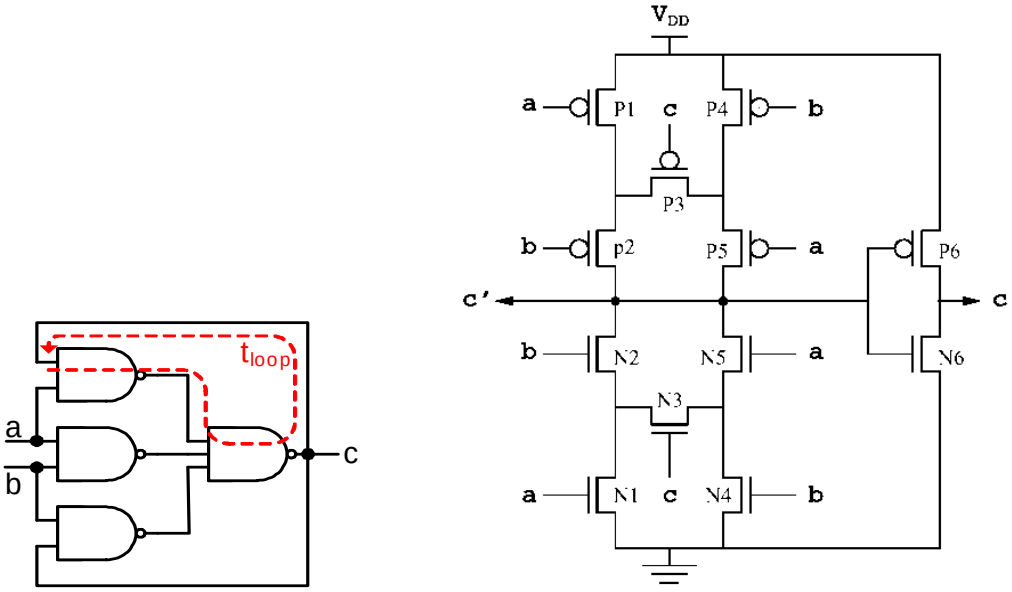
Let the detailed analysis start with the design block responsible for processing locally and remotely incoming clock signals, namely the elastic pipeline. As already mentioned in Section 3.3, elastic pipelines can be viewed as FIFO buffers for transitions. This type of pipeline was first introduced by Ivan Sutherland in [83] and in that context implemented the control path of his asynchronous micropipelines communication approach. As shown in Figure 4.1, the better part of an elastic pipeline consists of Muller C-Elements which are characterized in more detail in the following. This in-depth treatment and analysis of the Muller C-Element and other basic building blocks is crucial since the whole, generally delay insensitive, implementation of the TG-Alg rests on the properties of its subcomponents.

4.2.1 Muller C-Element

The Muller C-Element has already been mentioned as fundamental building block during the introduction of transition signalling in Section 3.3 where it was presented as a logic AND equivalent for signal transitions. The exact functionality of a two-input Muller C-Element can informally be described as the following: the output `c` is assigned with the same logic value as the inputs `a` and `b` whenever both inputs are equal (`c=a=b=0` or `c=a=b=1`). On the contrary, if the inputs have different logic values the output `c` retains its previous value `c=c_old`. As a Boolean function this can be expressed as:

$$c = c_{old} \cdot (a + b) + a \cdot b = c_{old} \cdot a + c_{old} \cdot b + a \cdot b$$

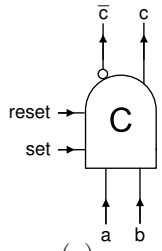
The Muller C-Element's behavior of retaining the old value of output c even if one of the two inputs changes its value to a contrary logic level clearly demands some sort of storage loop. However, the imperfectness of any real world design implementing a storage loop implies that some constraints on the input sequence have to hold to guarantee that the value of the storage loop can settle before a further input change occurs. Figure 4.2(a) illustrates one storage loop of a Muller C-Element in the example of an implementation which is based on NAND gates. From the schematic it becomes evident that the inputs a



(a) (b)

Figure 4.2: Muller C-Element implementation on (a) gate level (b) transistor level

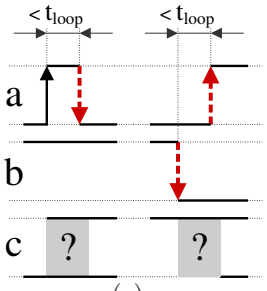
and b have to stay stable for at least t_{loop}, which is defined by the propagation delay through two NAND gates plus some wiring delays. More precisely defined a Muller C-Element's correct behavior rests upon the assumptions that:



(a)

a	b	*reset*	*set*	c	$\overline{c}$
0	0	1	0	0	1
0	1	1	0	c_{old}	$\overline{c_{old}}$
1	0	1	0	c_{old}	$\overline{c_{old}}$
1	1	1	0	1	0
X	X	0	X	0	1
X	X	1	1	1	0

(b)



(c)

Figure 4.3: Customized ASIC Muller C-Element

- a single input is not allowed to toggle faster than t_{loop} if the initial transition would cause output c to change its value. For example, if input $a = 0$, $b = 1$ and output $c = 1$, input b is not allowed to toggle faster than t_{loop} since the output preserving feedback loop needs time to settle the new value of $c = 0$ (see left part of Figure 4.3(c)).

- input a and b never change their logic level to the opposite value too close to each other. For instance, again starting with $a = 0$ and $b = 1$ both inputs must not change to the opposite polarity $a = 1$ and $b = 0$ within an interval smaller than t_{loop} (right part of Figure 4.3(c)).

The presented NAND implementation of the Muller C-Element only represents one of several implementation variants. Nevertheless, a storage loop with respective timing restrictions is common to all designs. However, the extent of the timing loop's delay t_{loop} as well as the overall speed, chip area and power consumption among different implementations are subject to large design-dependent variations. Due to the fact that the TG-Alg implementation aims at high speed combined with high reliability, it was decided to employ a transistor implementation following the design of Van Berkel [87]. Figure 4.2(b) presents the schematic of a Van Berkel-type Muller C-Element which takes the above considerations into account. Note that the storage loop in this design comprises of three transistors only, which, compared to the NAND-based Muller C-Element, considerably shrinks the critical time window for input changes. The actual Muller C-Element as it is employed in the TG-Alg ASIC design additionally incorporates some extensions to the Van Berkel scheme. First of all, a `set` and a `reset` input allow to bring the C-Element into a predefined initial state. Furthermore, for improved performance of the Muller C-Element (when it is used for implementing the elastic pipelines described below) two output signals, c and its inverted equivalent $\overline{c}$ are provided. The specialized Muller C-Element's circuit symbol is shown in Figure 4.3(a). Note that this Muller C-Element has already been used in the schematic of Figure 4.1. The exact functional description of the customized Muller C-Element is given by the truth table of Figure 4.3(b).

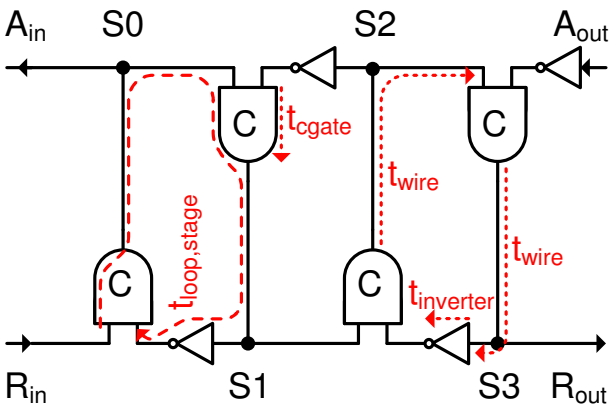


Figure 4.4: Elastic pipeline design

4.2.2 Elastic pipeline

Figure 4.4 shows a four-stage implementation of an elastic pipeline that mainly consists of Muller C-Elements. The pipeline's regular structure consisting of a Muller C-Element and one inverter per stage allows for effortless configuration of the FIFO's buffer depth. Ivan Sutherland described the functionality of a single stage of the micropipeline approach as the following: *"if the predecessor and the successor differ in state then copy predecessor's state else hold present state"* [83]. To illustrate the exact behavior of the elastic pipeline assume that the Muller C-Elements of all stages $S0$ to $S3$ hold the same value, e.g., all stages are at logic 0, i.e., the pipeline is empty. As an example, assume an incoming rising transition -↑ at R_{in} leading to $R_{in} = 1$. This logic value is propagated by the Muller C-Element of stage $S0$ to the subsequent pipeline stage only if stage $S1$ has different polarity than $S0$ which is the case because the Muller C-Element at stage $S1$ was initialized to a value of logic 0. This logic 0 which is inverted and fed back to the Muller C-Element of stage $S0$ enables $R_{in} = 1$ to pass through stage $S0$ and establishes a stable 1 at this Muller C-Element. Likewise, the propagation of the initial $R_{in} = 1$ transition continues until stage $S3$ is reached. As long as the acknowledge signal A_{out} is inactive, stages $S0$ to $S2$ are at the same logic value of 1. Further input transitions may propagate in a similar manner through the elastic pipeline until the pipeline is entirely filled (a full pipeline is characterized by alternating logic values among all pipeline stages). It has to be further noted that the input transition that led to $R_{in} = 1$ also changes the feedback path of every stage. This feedback mechanism forms the elastic pipeline's implicit flow control scheme. Hence, a pipeline stage is ready for processing the next input transition if the subsequent stage has also acknowledged the previous input. Applied to the above example, this translates to the fact that a new $R_{in} = 0$ transition can be handled by stage $S0$ as soon as the previous R_{in} input transition propagated through $S1$ (and the respective feedback inverter).

Similar to storing transitions into the pipeline, emptying the elastic pipeline starts as soon as the acknowledge signal A_{out} is issued. A_{out} triggers the removal of the elastic pipeline's rightmost transition. In the above mentioned example, $A_{out} = 1$ will remove

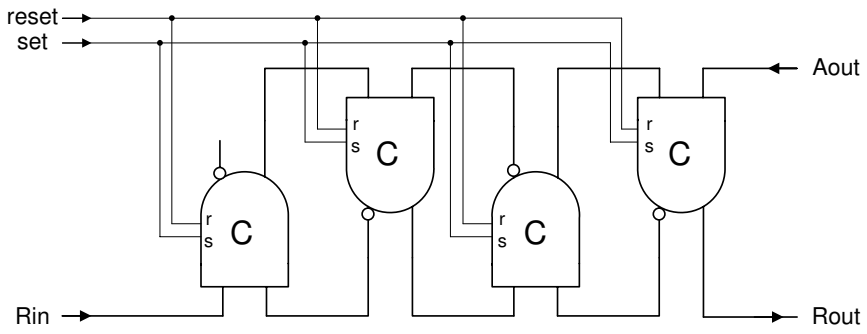


Figure 4.5: TG-Alg ASIC elastic pipeline design

the logic 0 from $S3$, resulting in the fact that the value logic 1 of $S2$ propagates to $S3$, the content of $S1$ moves on to $S2$ and so on. In other words, the "empty slot" generated by issuing A_{out} moves upstream stage by stage until it reaches stage $S0$. Sutherland used the metaphor of *"air bubbles that rise through water"* to illustrate this effect of data value progression.

In general, the elastic pipeline's way of transition processing provides a very elegant flow control and buffering mechanism as long as some basic timing constraints are maintained. The involved timing paths are depicted in Figure 4.4. Similar to the Muller C-Element itself, the feedback loops of the elastic pipeline introduce additional timing condition restricting the input sequence. The path delay $t_{loop,stage}$ limits the distance between two subsequent input transitions on R_{in}. Compared to the Muller C-Element's input constraints which are characterized by t_{loop} and have been presented in Section 4.2.1, $t_{loop,stage} \approx 2t_{cgate} + 2t_{wire} + t_{inverter}$ is obviously the more restricting factor since $t_{loop,stage} \gg t_{cgate} \approx t_{loop}$, with t_{cgate} being the determining variable of $t_{loop,stage}$'s value[1].

As already indicated in the elastic pipeline schematic of Figure 4.1, some characteristic features of the design used in the TG-Alg design have to be mentioned. Figure 4.5 depicts the implemented four-stage elastic pipeline in detail. The first distinction to the previously presented typical elastic pipeline architecture is given by the fact that the customized Muller C-Element of Figure 4.3(a) is employed. This version of the Muller C-Element provides its output signal via an inverted and a non-inverted port and hence allows to remove the additional inverter of the elastic pipeline's feedback path. Furthermore, the *reset* and *set* inputs allow to bring the elastic pipeline in a predefined state. In the case of a TG-Alg an empty pipeline is usually used as starting point, i.e., all stages hold a value of logic 0 forced by the *reset* signal. Another peculiarity of the TG-Alg-specific elastic pipeline is given by the fact that only the input signal R_{in} is used and output A_{in} is not connected. Again a closer look at Figure 4.1 reveals that R_{in} corresponds to the

[1]Note that, assuming the feedback loop's $t_{loop,stage}$ delay components t_{cgate} and t_{wire} being identical for all stages clearly poses a simplification. However, this assumption seems to be reasonable for the comparison to the propagation delay of a single Muller C-Element.

clock input signal (remote or local). In turn the feedback output A_{in} would correspond to an acknowledge signal for the incoming clock signal transitions. However, a strategy including A_{in} cannot be reasonably employed in the TG-Alg design for the reason (already discussed in Section 3.1) that the clock network has to be as small as possible, i.e., only one rail is allowed per node-to-node synchronization communication. Moreover, the explicit generation of an acknowledge signal would require some coordinating measures in the receiving pipelines to jointly acknowledge a sender's clock transitions. In contrast to the clock input side of the elastic pipeline, the far end interconnection to the Difference Module includes the entire pipeline interface R_{out} and A_{out}. As long $R_{out} = A_{out}$, the pipeline is empty and waiting for input transitions and no tick can be removed by the Difference Module. However, as soon as $R_{out} \neq A_{out}$ the pipeline holds at least one clock tick which can be consumed by altering A_{out} to the value of R_{out}. A more detailed description of the Difference Module's tick removal operation in conjunction with the local and remote elastic pipelines will be given in the following paragraphs.

4.3 Counting ticks

Each of the elastic pipelines presented above manages to buffer incoming clock transitions — four clock transitions in the particular case of the proposed TG-Alg ASIC standard node design[2]. This buffering scheme is essential because compared to the local tick generation, remote clock signals may arrive at (slightly) different instants. Therefore, to prevent these FIFO pipelines from overflowing, a tick removal strategy incorporating both remote and local pipelines has to be employed. Based on the clock ticks propagating through the pipelines, conditions have to be derived indicating for each pipeline pair whether or not the remote or the local pipeline has seen more clock ticks ($remote \geq local$ and $remote > local$). This essentially describes computations performed by the block introduced as $+/-$ Counter. Additionally, it can be seen as the necessary translation from the scope of transition signaling (clock ticks and elastic pipelines) to common asynchronous state logic (Pipe Compare Signal Generation and Threshold Modules).

4.3.1 Difference Module

As already mentioned before, the used tick generation algorithm is only interested in the difference of locally generated and remotely received ticks. The required difference computation is enabled by removing matching clock ticks from both the remote and the corresponding local pipelines. Note that the employed removal strategy is necessary due to the fact that a reasonable implementation of elastic pipelines can only store a very limited number of ticks. This constraint on the buffer capacity however is sufficient for the underlying tick generation algorithm since its proper operation solely relies on the relative offset

[2]The number can be easily adapted by adding more buffer stages.

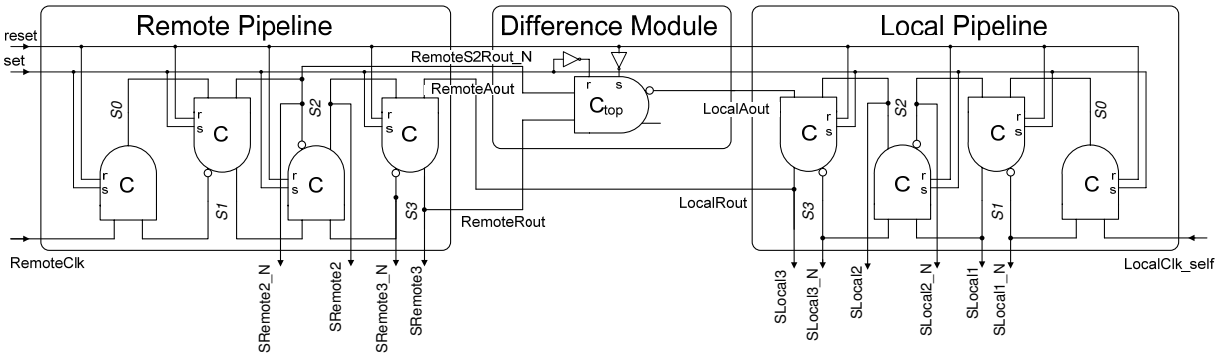


Figure 4.6: TG-Alg ASIC Difference Module and elastic pipelines

of remotely and locally generated clock ticks (recall Section 3.4.2). The so-called Difference Module which forms the core part of the previously introduced $+/-$ Counter implements the removal strategy by interconnecting remote and local pipeline in a special way. A schematic showing the Difference Module (which is essentially a Muller C-Element) in conjunction with the corresponding remote and local elastic pipeline is given in Figure 4.6. It should be noted that on reset the Muller C-Element of the Difference Module C_{top} is initialized to logic 1 while all other C-Elements of the pipelines are set to 0. This initialization is crucial for the tick removal strategy since it has to follow a strict sequence. The restrictions of the tick processing have to guarantee that the fill-level signals which are derived by the subsequent Pipe Compare Signal Generation Modules always show valid information. Due to the facts that the fill-level computation is performed continuously and that in practice removing remote and local ticks simultaneously cannot be implemented, another method has to be used to ensure correct fill-level values. In detail, a strategy has to be followed securing that if two matching ticks are present in corresponding pipelines the tick is first removed from the local pipeline and then from the remote side. This ensures that the conditions $remote \geq local$ and $remote > local$, which directly translate to the fill-level signals GEQ^e, GR^e and GEQ^o, GR^o are never falsely activated. To illustrate the restricted tick removal process, assume an example starting directly after reset, that is, remote and local Muller C-Elements are initialized to 0 while the Difference Module's C_{top} is set to logic 1, i.e., its inverted output also displays 0. This initialization allows a clock tick-$\uparrow$ message from the local side to propagate through $S0, S1$ and $S2$. The Muller C-Element at stage local $S3$, however is not ready to process the tick-$\uparrow$ until a corresponding tick-$\uparrow$ at the remote side has propagated down to $S2$ (where it is also halted). As soon as remote $S2$ has processed the tick-$\uparrow$, C_{top} enables $S3$ at the local side to propagate the pending tick-$\uparrow$. This in turn allows remote $S3$ to process the waiting tick-$\uparrow$. Tick processing follows this "local before remote" strategy continuously for every arriving clock transition.

4.3.2 Pipeline compare signal generation

The circuit blocks presented so far, i.e., elastic pipelines and Difference Module, which mainly consist of Muller C-Elements, are operating in the scope of transition signaling. In contrast to that, the Pipeline Compare Signal Generation (PCSG) Modules are implemented in asynchronous state logic since fill-levels, i.e., states, have to be assessed. Unfortunately, state logic circuits tend to produce glitches if input changes occur. However, such unintended glitches are catastrophic when later reverting back to transition signaling. Hence, as already mentioned in the previous subsection, it is crucial for the tick generation process that the respective fill-level signals *never* switch to the active state if $remote \geq local$ and $remote > local$ conditions, respectively, do not hold[3]. A hardware design optimized for the targeted ASIC manufacturing and implementing the whole $+/-$ Counter Module is presented in Figure 4.7. The PCSG part of this circuit block comprises fundamental logic elements like NAND and NOR gates and generates the fill-level indicator signals GEQ^e, GR^e and GEQ^o, GR^o. As introduced in Section 3.3, *even* clock ticks are tick-$\downarrow$ transitions while tick-$\uparrow$ messages denote *odd* clocks. The description of the Pipeline Compare Signal Generation Module's function can be partitioned in the processing of even and odd ticks. Hence, the PCSG part treating incoming even ticks ultimately triggers the generation of odd ticks by issuing GEQ^e, GR^e signals. Similarly, the circuit concerned with odd ticks and controlling GEQ^o and GR^o is responsible for generating even clock ticks.

The exact behavior of the PCSG unit in conjunction with the elastic pipelines and the Difference Module will be described in the following. In general it should be noted that all output signals of the Pipeline Compare Signal Generation Module (GEQ^e, GR^e, GEQ^o, GR^o) as well as all internal logic operations are active *low*, i.e., a value of logic 0 indicates that the respective signal is activated. This peculiarity of active low outputs and the fact that exclusively inverting basic gates (NAND instead of AND, NOR instead of OR) are used within the PCSG design, originates from optimizing the circuits for a fast ASIC implementation. Retrieving state information from the transition-logic-based elastic pipelines has to be designed carefully since it obviously does not follow the elastic pipeline's original design goal. Three taps of the local pipeline are combined to ensure that no dynamic effects during tick arrival or removal can compromise the fill level signal. In detail, the signals $SLocal1$, $SLocal2$ and $SLocal3$ in conjunction with the NOR_1 gate are used to indicate whether or not the pipeline holds a single even tick. Likewise, the inverted local pipeline signals $SLocal1\_N$, $SLocal2\_N$, $SLocal3\_N$ together with the NOR_2 are used to determine if an individual odd tick is stored inside the pipeline. The fill-level indicators on the remote side ($SRemote2$, $SRemote3$ and $SRemote2\_N$, $SRemote3\_N$) are responsible for checking if one or more clock ticks are currently stored in the pipeline. An appropriate combination of local and remote side fill-level signals allows to generate the output signals GEQ^e, GEQ^o and GR^e, GR^o which represent the conditions $remote \geq local$ and $remote > local$, respectively (see Table 4.1(a) and 4.1(b)). To attain an active fill-level

[3]However, signals may stay active even if the above conditions are no longer fulfilled.

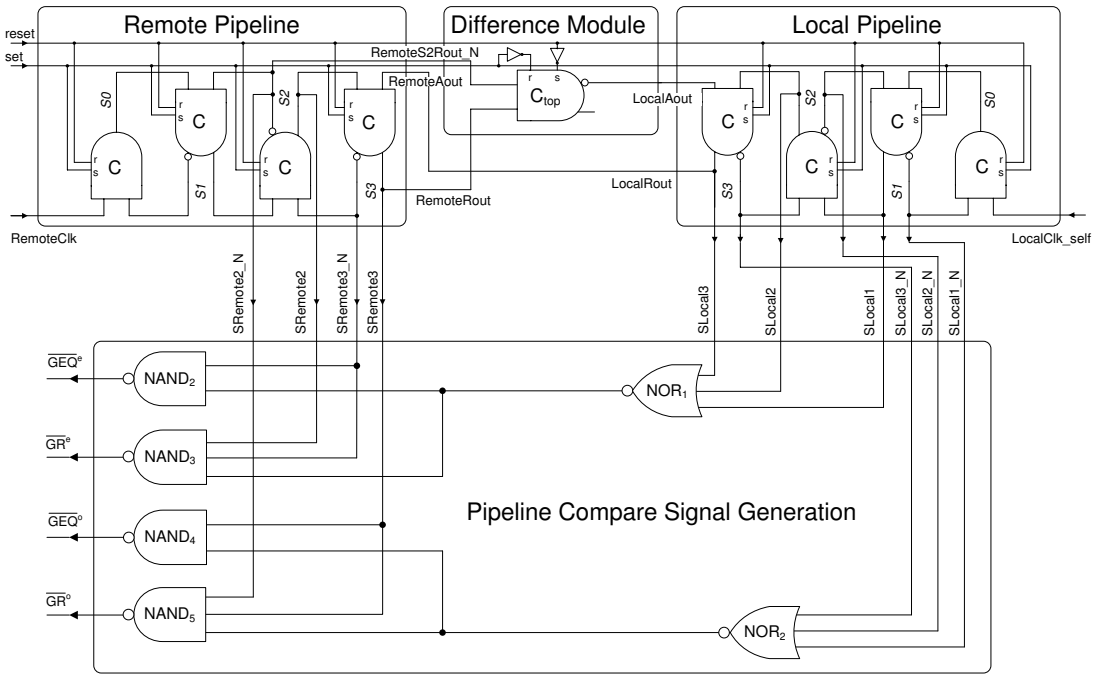


Figure 4.7: TG-Alg ASIC +/− Counter Module

signal GEQ^e, corresponding to $remote \geq local$ it has to hold that:

- only one even (tick-↓) clock transition is stored inside the local pipeline, which is indicated by NOR_1
- at least one even clock tick is present in the remote pipeline, indicated by signal $SRemote3\_N$

These two conditions are combined in a final step via $NAND_2$, generating the output GEQ^e. Similarly, for the activation of the low active signal GR^e implementing the condition $remote > local$ the following constraints have to be fulfilled:

- again only one even (tick-↓) clock transition is allowed inside the local pipeline, which is assessed by the NOR_1 gate
- more than one clock tick has to be present in the remote pipeline, an even clock tick in pipeline stage $S3$ and additionally an odd tick in stage $S2$.

These conditions are evaluated by the gate $NAND_3$ via signals $SRemote2\_N$ and $SRemote3\_N$ in conjunction with the output of NOR_1. The activation of the signals GEQ^o and GR^o follows by analogous means, simply treating odd instead of even input signals. Table 4.1(a) gives the detailed signal conditions for activating GEQ^e and GR^e, whereas Table 4.1(b) presents the respective signal conditions for treating odd ticks and generating GEQ^o as well as GR^o.

Table 4.1: Activation patterns for fill-level signals

(a)

	remote		local		
	$S2$	$S3$	$S3$	$S2$	$S1$
GEQ^e	0 1	0 0	0	0	0
GR^e	1	0	0	0	0

(b)

	remote		local		
	$S2$	$S3$	$S3$	$S2$	$S1$
GEQ^o	1 0	1 1	1	1	1
GR^o	0	1	1	1	1

4.4 Generating ticks

The final processing step of every TG-Alg node is concerned with the evaluation of the counter fill-levels and has to generate new clock ticks according to the tick generation algorithm's rules. The application of threshold functions to the GEQ^e, GEQ^o and GR^e, GR^o signals corresponds to evaluating the conditions of the "Relay Rule" and "Increment Rule" from Algorithm 8 for even and odd ticks, respectively. Similarly to the Pipe Compare Signal Generation, the threshold functions are implemented in asynchronous state logic, while the actual tick generation process performs a back-transformation to transition signaling.

4.4.1 Threshold Modules

Four distinct threshold circuits allow to separately evaluate all output signals of a node's $(3f + 1)$ $+/-$ Counter Modules. As depicted in Figure 4.8, two threshold circuits are responsible for processing the fill-level signals GEQ^e and GR^e for even ticks. This way they implement the tick generation algorithm's "Relay Rule" and "Increment Rule" by virtue of $f + 1$ and $2f + 1$ threshold circuits, respectively. Similarly to the even PCSG output signals, the odd counterparts GEQ^o and GR^o are treated by distinct threshold circuits. This separation of even and odd ticks has already been motivated in Section 3.3 and originates in the problems that threshold circuits (m-out-of-k, e.g., $2f+1$-out-of-$3f+1$) cannot be implemented glitch-free in asynchronous logic and the fact that non-interference of subsequent data waves (ticks) has to be ensured. The interleaved operation of threshold circuits responsible for even and odd ticks allows to mask interfering glitch phenomena. More precisely, the special properties of the TG-Alg implementation as a whole allow that every threshold circuit might produce glitches within a certain portion of its "inactive" phase. This inactive phase denotes the time interval in which this threshold circuit does not contribute to the generation of the next clock tick. Thus, for example the input sequence and therefore the output of the $2f + 1$ threshold circuit might change its value several times after a new odd clock tick has been triggered according to the status of the GEQ^e signals. This toggling does not pose a threat as long the GEQ^e signals and the respective threshold circuit output finally stabilize to enable the generation of the subsequent tick.

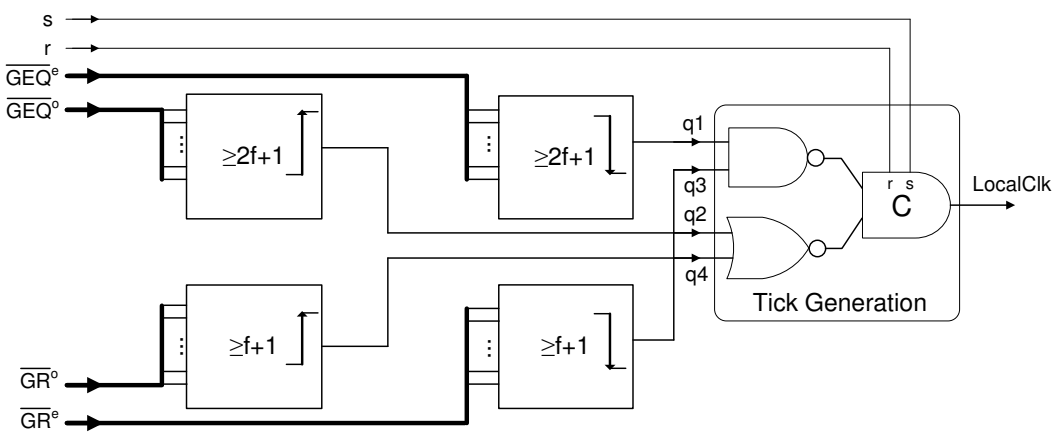


Figure 4.8: TG-Alg ASIC Threshold Modules and Tick Generation

The implementation constraint which ensures that all threshold circuits stabilize in a timely manner has already been introduced by Constraint 3.4.1 in Section 3.4.3.

The implementation of a single threshold circuit relies on a conventional sum of products scheme consisting of simple standard cell gates only. Figure 4.9 illustrates the design and basic structure of a 3-out-of-4 sum of products threshold circuit. The four product terms marked in Figure 4.9(a) represent all valid three-input combinations in the 3-out-of-4 example. When considering a more general m-out-of-k design it can be derived that $\binom{k}{m}$ product terms have to be computed and then summed up. The threshold circuits of the ASIC TG-Alg implementation have been designed for 11 input signals. The resulting 4-out-of-11 implementation of the $f+1$ threshold circuit yields 330 product terms. These product terms clearly have to be summed up in a tree-like cascaded architecture since no elementary gates with a fan-in of 330 are available in the ASIC target technology. A notable peculiarity of the sum of products implementation is the fact that the $2f+1$ threshold function requires exactly the same amount of product terms as the $f+1$ design. A closer look at both designs reveals that for the given configuration of $n-1$ inputs with $n-1 = 3f+1 = 11$, the required threshold functions $f+1 = 4$ and $2f+1 = 7$ lead to the same complexity of the threshold circuit $\binom{11}{4} = \binom{11}{7} = 330$. In general, this fact allows that only one threshold function, either the $f+1$ or the $2f+1$ circuit, has to be designed since both threshold circuits can be converted into each other by simply inverting all input and output signals.

Even though the sum of products scheme has exponential scaling with the number of inputs $n-1$, in [37] the approach has been identified to be the best match with respect to the given implementation constraints and requirements. Beside the aim for high-speed the most stringent requirement that led to the selection of the sum of products scheme was the design for low propagation delay variations. This demand for low delay jitter reflects that the proposed tick generation algorithm's correctness and performance ultimately rely on the ratio Θ of different timing paths within the TG-Alg design (cf. Section 2.3). Detailed constraints on path delay relations which have to hold to guarantee correct algorithm ex-

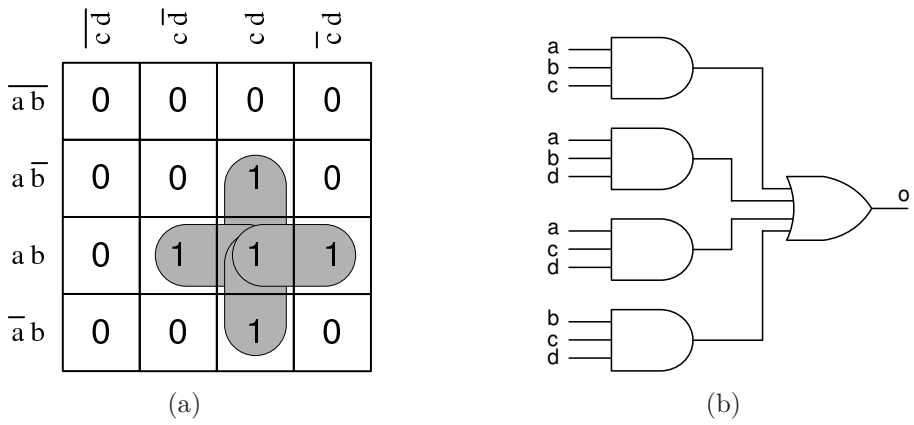


Figure 4.9: 3-out-of-4 threshold circuit (a) Karnaugh-Veitsch diagram and (b) sum of products implementation based on standard gates

ecution have already been presented as Constraints 3.4.1, 3.4.2 and 3.4.3. In addition, Equation 3.3 (Precision π) revealed that matching path delays to attain low jitter directly translates to performance increase in terms of enhanced precision. Among the system-level requirements presented in Section 1.7 it has been defined that the number of nodes n and thus the thresholds $f+1$ and $2f+1$ have to be configurable for different numbers of f. This demand is accounted for by enhancing each threshold circuit input with additional two-stage masking logic. Even though the extra logic adds delay to the threshold circuits, it increases the delay for all paths, which does not worsen the critical delay jitter. The requirement to provide a configurable threshold m, i.e., 7, 5, 3 for the $2f+1$ and 4, 3, 2 for the $f+1$ circuit, however, led to distinct $2f+1$ and $f+1$ designs. The reason for this optimization is based on the observation that $\max\{\binom{11}{7}, \binom{11}{5}, \binom{11}{3}\} = \max\{330, 462, 165\} = 462$ while $\max\{\binom{11}{4}, \binom{11}{3}, \binom{11}{2}\} = \max\{330, 165, 55\} = 330$ which leads to substantial savings if a separate $f+1$ threshold circuit design is used.

4.4.2 Tick generation

The threshold circuits described above individually process even and odd ticks and provide output signals to indicate when the next tick message has to be generated according to the algorithm's rules. The actual generation and broadcasting of a tick, however, is handled by the design unit labeled Tick Generation in Figure 4.8. In the Tick Generation Module the four threshold circuit outputs q_1, q_2, q_3 and q_4 are combined by simple logic gates in a way such that only valid clock ticks are generated, in essence it handles the concurrency of the two rules of the algorithm. Furthermore, the Tick Generation Module has to ensure that after generating a clock tick the TG-Alg's clock output remains stable despite the fact that the outputs of the threshold circuits might toggle due to glitches. This retention of the clock output is enabled mainly by the final Muller C-Element which only issues a

new tick if both inputs indicate to do so. However, since the storage loop of the Muller C-Element needs stable inputs during its settling time (cf. Section 4.2.1) the outputs of the threshold circuits have to be stable for a small time interval before and after a new tick is generated. This safety window is ensured by the already presented so-called Interlocking constraint (Constraint 3.4.1). Assuming that all implementation constraints are fulfilled and taking the above mentioned considerations into account, a new tick is generated only if:

- the threshold circuits responsible for the generation of the previous tick (by providing enabled input signals *GEQ*, *GR*) have become inactive again
- at least one of both threshold circuits concerned with evaluation of the last tick, and hence responsible for the generation of the subsequent tick, gets activated

To illustrate the operation of the tick generation circuit shown in Figure 4.8, an example trace of the activation and deactivation patterns of q_1, q_2, q_3 and q_4 indicating the tick generation instants is depicted in Figure 4.10. Note that signals q_1 and q_4 are active low, while q_2 and q_3 are high when activated. In the example trace it can be observed that a new odd tick is triggered as soon as both q_2 and q_3 are inactive again and one of the signals q_1 or q_4 gets activated (in the example trace q_4 triggers the tick generation). In detail, the trace shows that ❶ the active signals q_2 and q_3, one of which previously triggered the generation of the currently active *even* tick, will start to get deactivated ❷ as soon as the generated even tick propagates to sufficiently many of the GR^o and GEQ^o signals to get below the respective threshold. Due to activations arriving late near the threshold while other paths have already started the deactivation process, q_2 and q_3 may toggle several times before finally being stable deactivated ❸. The activation of q_1 or q_4, indicating that the threshold of GEQ^e or GR^e has been reached ❹, will in turn immediately trigger the generation of the subsequent *odd* tick.

4.5 TG-Alg implementation characteristics

After the detailed component descriptions of a TG-Alg this section proceeds with the presentation of implementation properties and deficiencies. The analysis of the whole TG-Alg design is conducted by putting together the characteristics of all sub-units. For this purpose exact numbers for the hardware effort in terms of gate equivalents and die size will also be assessed. Taking a closer look at the basic architecture of a cluster of TG-Alg nodes reveals that an almost fully connected point-to-point network is assumed (it is not fully connected because no self-reception loops are present, cf. the discussion on the number of nodes in Section 4.1). This network topology clearly implies a quadratic growth of the TG-Net's number of links with node count n and thus also with f, i.e., $\mathcal{O}(f^2)$ and has direct impact on the complexity of a TG-Alg's implementation.

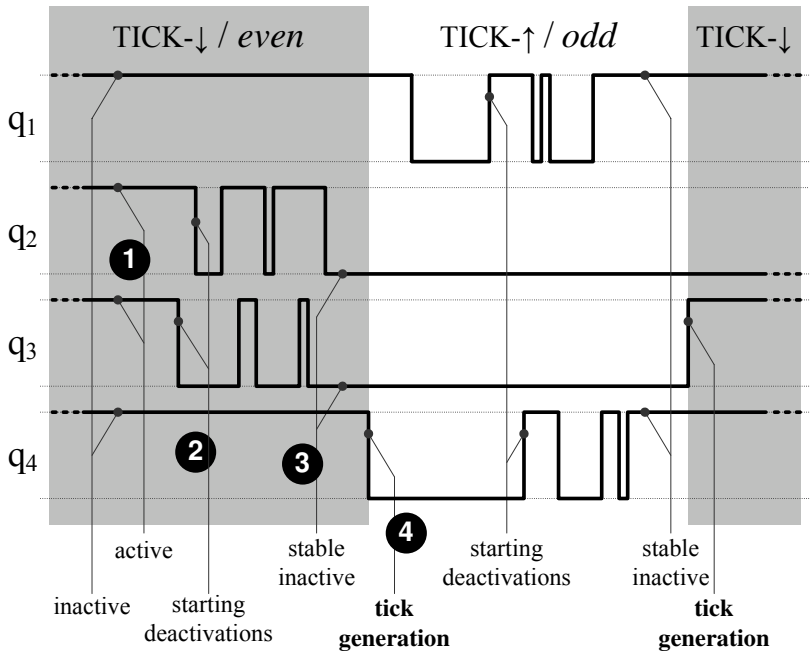


Figure 4.10: Example trace of the tick generation signals q_1, q_2, q_3 and q_4

Staying with the flow of the previously presented sub-blocks the tick queueing and tick counting mechanisms are treated first. The hardware effort for building a TG-Alg's queueing and counting blocks is for a considerable part determined by the amount of incorporated Muller C-Elements. Considering the remote and local elastic pipelines as well as the Difference Module, the Muller C-Element presents the only relevant building block, whereas the Pipeline Compare Signal Generation module is assembled using a few basic gates with two and three inputs, respectively. To assess a TG-Alg's hardware effort for tick queueing and counting it has to be taken into account that the implementation of a single node consists of $n-1=11$ individual $+/-$ Counter Modules — one for each remote TG-Alg. Table 4.2 presents numbers for gate count and silicon area (in the $0.18\mu m$ ASIC target technology) treating sub-modules as well as the whole design of a $+/-$ Counter. Furthermore, the hardware effort is added up to account for the 11 $+/-$ Counters of the actual TG-Alg implementation. It can be observed that the elastic pipelines are the main contributors to the chip area of each $+/-$ Counter. Investigating the reason for this circumstance reveals the Muller C-Element's responsibility due to its rather complex structure and the resulting high area effort when compared to other basic gates of the $+/-$ Counter (NAND and NOR gates). It has to be noted that the used custom cell transistor-level Muller C-Element already substantially reduced the Muller C-Elements impact on hardware effort. In case of a gate-level implementation (e.g., the NAND gate design of Figure 4.2(a)) the even more pronounced complexity and hardware effort of the Muller C-Element would have additionally resulted in notable loss of performance.

In contrast to the queueing and counting blocks ($+/-$ Counter) every TG-Alg holds

Table 4.2: Hardware effort for queueing and counting ticks

	# of basic gates	# of C-Elements	area in $[\mu m^2]$
Remote Pipeline	-	4	944
Local Pipeline	-	4	944
Difference Module	2	1	270
PCSG	6	-	395
$+/-$ Counter	8	9	2,553
11 $+/-$ Counters (for an entire TG-Alg)	88	99	28,083

Table 4.3: Hardware effort for Threshold Modules

	# of basic gates	# of C-Elements	area in $[\mu m^2]$
$f+1$ circuit	550	-	51,641
$2f+1$ circuit	1,013	-	176,083
Tick Generation	2	1	303
Threshold Modules: $2 \times f+1$, $2 \times 2f+1$ and Tick Generation	3,128	1	455,751

only one Threshold Module — incorporating four threshold circuit units and the Tick Generation Module. As thoroughly described in Section 4.4.1, threshold circuits are purely combinatorial blocks following a sum of products implementation. Given an input width of $n-1=11$, the presented complexity growth with the number of inputs yields 330 and 462 product terms for each of the $f+1$ and $2f+1$ threshold circuits, respectively. Therefore the exponential increase with approximately $\binom{3f+1}{f+1}$ is one of the prominent cost driving factors when scaling the tick generation system's resilience $f \leq \left\lfloor \frac{n-2}{3} \right\rfloor$ and hence the number of nodes n. Due to the fact that basic standard cell gates like NAND and NOR, which are used in the sum of products implementation, are typically available only with two and three inputs, hardware effort is additionally increased with increasing number of n. This is true for the product terms as well as for the terminal sum term because increasing numbers n and m result in the need for cascading basic gates. In contrast to the threshold circuits the Tick Generation Module does not suffer from scaling effects since it consists of two basic gates and a single Muller C-Element only. Similarly to the elastic pipelines it benefits from the transistor-level implementation of the Muller C-Element. Table 4.3 lists gate count and area numbers for the involved design units and the Threshold Module block overall.

The comparison of a TG-Alg's components in terms of hardware effort, shown in Ta-

Table 4.4: Hardware effort of a single TG-Alg and its components

	# of basic gates	# of C-Elements	area in $[\mu m^2]$	area in %
Remote Pipeline	-	4	944	0.20
Local Pipeline	-	4	944	0.20
Difference Module	2	1	270	0.06
PCSG	6	-	395	0.08
$+/-$ Counter	8	9	2,553	0.05
11 $+/-$ Counters	88	99	28,083	5.80
$f+1$ circuit	550	-	51,641	10.67
$2f+1$ circuit	1,013	-	176,083	36.39
Tick Generation	2	1	303	0.06
Threshold Modules	3,128	1	455,751	94.19
single TG-Alg	3,218	100	483,862	100.00

ble 4.4, reveals that the sum of product threshold circuit implementation accounts for a substantial part of the entire design. Almost 95% of a TG-Alg's chip area is devoted to the Threshold Modules. The enormous hardware effort reflects the threshold circuits' unfavorable scaling with f and n. In general, the Threshold Modules' predominance in hardware effort allows to give an estimate for the scaling of a TG-Alg's chip area following $\approx \binom{3f+1}{f+1}$. This scaling obviously only applies for the used sum of products approach and would be completely different for other implementation technologies. Analogously to the customized Muller C-Element, an applicable enhancement to reduce the sum of products area effort might be given by an optimized transistor-level implementation. Furthermore, the design alternatives presented in [37] might also provide reasonable options.

4.6 Discussion on algorithm implementations for weaker failure models

In the context of the numbers for gate count and chip area of the Byzantine-tolerant tick generation implementation, a comparison to alternative, less complex approaches seems to be indicated. Algorithms able to cope with (clean/unclean) crashes as well as omission failures have already been introduced in Section 2.3.1. The properties of these algorithms in terms of how many nodes a distributed system has to comprise to be resilient against f failures of a specific type has been shown. Furthermore, the algorithms' complexity has been analyzed in terms of assessing the number of rules and estimating their intricacy. However, what is missing in the treatment of the algorithms for weaker failure models up to this point is an analysis of the underlying distributed system failure model's appro-

Table 4.5: Comparison of Byzantine-, omission- and crash-tolerant algorithms, i.e., Algorithms 8, 6 and 5 in a system with $f = 3$

	Byzantine	Omission	Crash
nodes $n \geq$	$3f + 2$ 11	$2f + 2$ 8	$f + 2$ 5
links	$9f^2 + 9f + 2$ $11 \times 10 = 110$	$4f^2 + 6f + 2$ $8 \times 7 = 56$	$f^2 + 3f + 2$ $5 \times 4 = 20$
Threshold Modules per node	4	2+2 (simple)	2 (simple)
Chip area per node in μm^2	483,862	33,188	9,810
Chip area per node normalized with Byzantine-tolerant design	100%	7%	2%
Chip area entire system in μm^2	5,322,482	265,504	49,050
Chip area entire system normalized with Byzantine-tolerant design	100%	5%	0.9%

priateness for *real* hardware faults. Furthermore, a comparison of the implementation's hardware effort for different approaches might additionally be able to provide valuable insights.

The algorithm analysis of Section 2.3.1 shows that substantial savings in chip area/hardware effort can be expected by relaxing the underlying failure model. More precisely, the reduction of the adverse power of the allowed failures has major impact on the number n of TG-Algs needed to be able to tolerate a fixed number f of failures. Moreover, it has to be noted that less restrictive failure models also result in weaker requirements for node connectivity. Additionally, the number and complexity of the algorithm's rules also profits from the changed failure semantics. Table 4.5 presents the implementation complexity and hardware effort of the omission- and crash-tolerant algorithms (Algorithm 6 and 5) together with the numbers for the Byzantine-tolerant Algorithm 8. This comparison confirms the expected savings when dealing with less restrictive failure models. Considering the analyzed systems which are resilient to $f = 3$ faults the omission-tolerant approach achieves a reduction in hardware effort by a factor of 20 while the crash-tolerant system uses less than $\frac{1}{100}$ of the Byzantine-tolerant design's chip area. As one would expect from Table 4.4, the significant savings are mostly due to the reduced complexity and number of threshold circuits[4]. Especially the implementation of the rules indicated as "simple", representing the algorithm's *at least* 1 conditions, can be performed very hardware efficiently since the functionality translates to a simple OR gate. In addition to the hardware efficiency of the omission and crash algorithms, the appropriateness of the underlying distributed systems' failure models for typically encountered hardware faults is of increased interest. As developed in the following, a severe mismatch between hardware faults like stuck-at or stuck-open faults (cf. Section 1.4.1) and the implemented failure models can be

[4]Using a different implementation technology with improved, e.g., linear, scaling in n like threshold logic [6] would radically change the picture.

identified. A *clean crash* in the setting of a TG-Alg, for instance, can be translated into two main requirements:

(i) Causality: A fault must not produce any extra transitions, or move them ahead in time — every transition experienced by a receiving node must be the result of the algorithm's proper operation.

(ii) Symmetry: The perception of a fault must be the same for all receivers.

When translating requirement (i) to effects on hardware level it can be shown that it is easily violated, e.g., by transients on a communication link which generate extra transitions. Even a stuck-at fault may produce one extra (early) clock transition, thus invalidating condition (i). Requirement (ii) is obviously not fulfilled if the perception of a fault depends on a receiver's specific properties like physical location, threshold or speed. Marginal effects like short glitches or undefined voltage levels created by a fault can be noted as problematic conditions. An open defect in a logic gate or communication link may not only produce both of these, but will definitely cause asymmetric perception if it affects only a branch of the network. These examples make it evident that the assumption of clean crashes is very optimistic in a typical hardware setting, since it rules out many commonly encountered fault scenarios. Unfortunately, relaxing the assumptions to *unclean crash failures* does not help much since requirement (i) remains unchanged, while with respect to (ii) asymmetric perception is now allowed *once* before a faulty link finally turns mute.

The assumption of *omission failures* clearly disburdens of requirement (ii), but it is still hard to argue that even (i) alone can be maintained in a realistic setting. In summary, employing a failure model more restricted than the Byzantine model can generally not be justified for a hardware implementation in practice. To be able to give at least a probabilistic estimation of a system's resilience, a thorough coverage analysis with respect to the used (weak) failure model and expected hardware faults seems to be necessary. Nonetheless, as indicated in Table 4.5, relaxing failure semantics shows the potential for significant savings in terms of chip area and system complexity. The above presented difference in system complexity by a factor of 100 of the Byzantine- and crash-tolerant system suggests that it might be worthwhile to invest in hardening weaker failure models for typically encountered hardware faults. Again a concept from the distributed community might be of help to enforce requirements (i) and (ii). The approach of "simulation" and "failure transformation" for example presented in [81] might provide appropriate means to attain a system tolerant of typical hardware faults on the basis of a failure model weaker than the Byzantine.

4.6.1 Failure transformation

Assume that up to f failures of a certain type A (e.g., Byzantine) might occur in a given distributed system. Then there exist failure transformation algorithms, which provide

services (e.g., broadcast) to an upper layer that are proven to fail in a more restricted way B (e.g., by clean crashes only). Unfortunately generic failure transformation algorithms for turning Byzantine failures into weaker failures typically need to communicate multiple times between distinct nodes p and q to provide a simulation of a single message from p to q at the upper layer. However, analogously to the arguments for conveying only simple zero-bit $\uparrow$ / $\downarrow$ transitions in the Byzantine-tolerant tick generation approach (cf. Section 3.1), the same criteria apply for clock and tick synchronization in general. Thus, a multi-round transformation is definitely not acceptable from a performance point of view. Fortunately, there exists a transformation scheme which turns f Byzantine failures at a lower level of abstraction into f omission failures at the upper level. To be precise, the Byzantine failures are transformed into late timing failures. However, Algorithm 6 introduced in Section 2.3.1 to be omission-tolerant can cope with late timing failures as well. The basic concept of the Byzantine-tolerant tick synchronization approach which is based on failure transformation is depicted in Figure 4.11 and has the following properties:

- the system comprises $2f + 2$ *virtual* supernodes representing the upper layer — omission-tolerant system
- each of the $2f + 1$ supernodes consists of $f + 1$ omission-tolerant subnodes
- at the lower layer each single subnode $p.i$, where p denotes the supernode and $i \in \{1, \ldots, f + 1\}$ the subnode number, executes the transformation algorithm, i.e., Algorithm 9, together with an instance of Algorithm 6.

Figure 4.11 shows an enlarged view of virtual supernode p with identifier $2f + 2$, with three of its subnodes $p.1, p.f$ and $p.f + 1$. It can be seen that $p.1$ is composed of the omission-tolerant algorithm together with an instance of the transformation algorithm — depicted as a set of voters `vnode` 1, ..., `vnode` $2f + 1$.

The actual mapping of Byzantine failures to omissions/late timings is enabled by the $2f+1$ `vnode` voting functions each of which is fed by $f+1$ distinct omission-tolerant subnode outputs of a supernode. In general, the failure transformation Algorithm 9 located at node $p.i$ waits until TICK(k) has been received from all subnodes $q.1, \ldots, q.f + 1$ of a supernode q before it delivers TICK(k) to the omission-tolerant algorithm running at $p.i$. This way each subnode $p.i$ may fail arbitrarily without violating the restrictions of the omission failure model. Moreover the failure mode assumption allows that up to f subnodes may suffer from a fault. By waiting for all $f + 1$ subnodes of a supernode, Algorithm 9 ensures

Algorithm 9 Transformation algorithm at subnode $p.i$

for all supernodes $q \neq p$ execute task:
vnode q**:**
1: **if** received TICK(k) from all nodes $q.j$, $1 \leq j \leq f + 1$ **then**
2: deliver TICK(k) from supernode q to node $p.i$;
3: **end if**

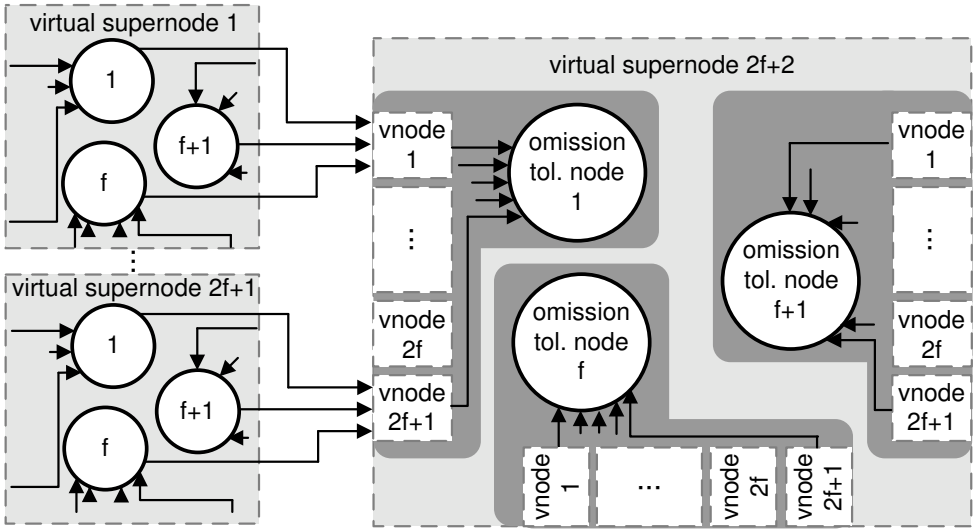


Figure 4.11: Simulation of an omission-tolerant system

that TICK(k) is not delivered to the tick synchronization algorithm running at $p.i$ before it was sent by at least one correct subnode, which makes it compliant to Algorithm 6's assumptions.

While Algorithm 6 can be implemented in hardware by analogous means as the in detail presented Byzantine-tolerant Algorithm 8, it is not self-evident that a feasible implementation exists for the transformation algorithm. A possible implementation of a single `vnode` q process is depicted in Figure 4.12. Analogously to the design of the Byzantine-tolerant algorithm the asynchronous `vnode` implementation has to overcome the problem of voting over unsynchronized subnodes sending anonymous transitions only. Therefore the transformation algorithm needs the possibility to store and align matching ticks with each other. Again, elastic pipelines can be used for this purpose. The actual voting is performed by an $f + 1$-input Muller C-Element which only generates a new tick if it has received the

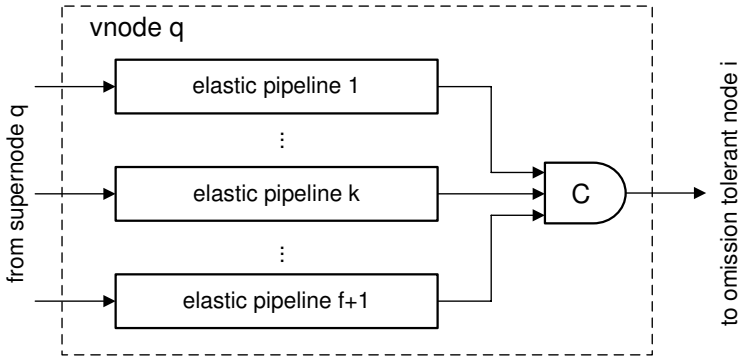


Figure 4.12: Implementation of a single vnode of the transformation algorithm.

Table 4.6: Implementation complexities of Byzantine- and omission-tolerant system

	nodes n	links ℓ
$\mathcal{S}_{Byz}$	$3f+2$	$9f^2+9f+2$
$\mathcal{S}_{Om}$	$2f^2+4f+2$	$4f^4+14f^3+18f^2+10f+2$

respective tick in all of its preceding pipelines. The pipeline size for this approach can be bounded by the difference of the number of ticks received from two correct subnodes of the same supernode, i.e., by the precision π.

The characteristics of the presented failure transformation approach can be summarized by the fact that the advantage of a simpler tick generation algorithm has been traded against an increased number of nodes which have to be augmented by a (simple) transformation algorithm. Recalling the omission-tolerant algorithm's substantial savings in chip area (presented in Table 4.5) an assessment in order to evaluate whether the transformation approach pays off has to be performed. Therefore the two Byzantine tick generation solutions:

- a distributed system running instances of Algorithm 8, further denoted by $\mathcal{S}_{Byz}$, and
- the failure transformation based solution with Algorithm 6 and Algorithm 9, abbreviated by $\mathcal{S}_{Om}$,

are compared with respect to implementation complexity and overall performance.

To be able to tolerate f Byzantine failures, $\mathcal{S}_{Byz}$ requires $n \geq 3f+2$ nodes. The number of links ℓ is, as already presented in Table 4.5, given by $\ell = n(n-1) \geq 9f^2+9f+2$. In case of the transformation-based solution $\mathcal{S}_{Om}$, $n' \geq 2f+2$ must hold for the number of (virtual) supernodes n', from which it follows that the number of subnodes needs to be $n \geq (2f+2)(f+1) = 2f^2+4f+2$. The interconnecting network complexity arises from the requirement that every subnode receives ticks from the subnodes of all supernodes except its own, that is, $(n'-1)(f+1) \geq 2f^2+3f+1$ incoming links yielding an overall link count of $\ell \geq (2f^2+3f+1)(2f^2+4f+2) = 4f^4+14f^3+18f^2+10f+2$. These results are also summarized in Table 4.6.

At first sight with increasing f $\mathcal{S}_{Byz}$ scales far better than $\mathcal{S}_{Om}$. A closer look, however, reveals that dismissing $\mathcal{S}_{Om}$ at this point is not necessarily the optimal choice since the nodes have quite different internal complexity. In the following let A_{Byz} and A_{Om} be a measure for the implementation complexity of a single node of solution $\mathcal{S}_{Byz}$ and $\mathcal{S}_{Om}$, respectively. Let's further denote the complexity of distributed system $\mathcal{S}_{Byz}$ and $\mathcal{S}_{Om}$ tolerating f Byzantine failures with $A^{sys}_{Byz}(f)$ and $A^{sys}_{Om}(f)$, respectively. In this context it has to be noted that different implementation techniques will require different complexity measures. For instance, in the case of the proposed hardware implementations the die size is a natural choice for complexity comparison. Obviously this measure is not suitable for measuring the effort of a software implementation. Thus a comparison can

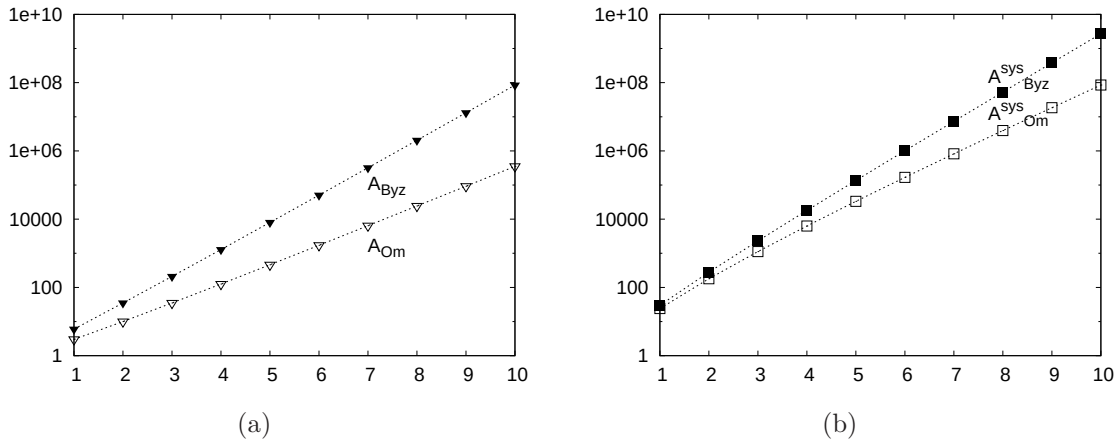


Figure 4.13: Implementation complexities for (a) single Byzantine- and omission-tolerant node and (b) respective systems

only be made with respect to a given target technology. Recalling the implementation complexities of the algorithms presented in Table 4.5, hardware costs of the different implementations for $f = 3$ can be estimated as $A^{sys}_{Byz}(3) = (3f+2)A_{Byz} = 11A_{Byz} \approx 5mm^2$ and $A^{sys}_{Om}(3) = (2f^2+4f+2)A_{Om} = 32A_{Om} \approx 1mm^2$ which shows that $A^{sys}_{Om}(3) < A^{sys}_{Byz}(3)$. The outcome that a system consisting of far more (however, omission-tolerant and hence less complex) nodes[5] is smaller than the Byzantine-tolerant design clearly contradicts the system complexity one would expect from Table 4.6. Hence, the question arises whether the break-even point expected from the analysis above has not been reached at $f = 3$, such that $A^{sys}_{Byz}(f)$ scales better for larger f. It turns out that this is not the case. The reason for this is hidden in the fact that the implementation complexity A_{Byz} and A_{Om} for the different tick generation algorithms depend on f and that $A_{Om}(f) \ll A_{Byz}(f)$. This relation will become more explicit when the implementation characteristics of the Byzantine-tolerant TG-Alg (presented above in Table 4.4) are considered. From Section 4.5 it is known that the Threshold Modules are the main driving force for the implementation complexity of a TG-Alg node. The sum of products' scaling which follows $\binom{n-1}{m}$, e.g., $\binom{3f+1}{f+1}$ outweighs all other building blocks by far. Hence the complexity of the omission- and Byzantine-tolerant node implementations can be approximated by $A_{Om}(f) \propto \binom{2f+1}{f+1}$ and $A_{Byz}(f) \propto \binom{3f+1}{f+1}$, respectively. Figure 4.13 depicts the trends of $A^{sys}_{Byz}(f)$ and $A^{sys}_{Om}(f)$ for $f \in \{1, \ldots, 10\}$. Note that for the complexity trends presented graphs, i.e., $A^{sys}_{Byz} = (3f+2)\binom{3f+1}{f+1}$ multiplicative factors are set to 1. Since the implementation complexity scale is logarithmic, a multiplication by a factor would result in a translation of the curve only. Thus, one can clearly see that $A^{sys}_{Om}(f)$ scales far better than $A^{sys}_{Byz}(f)$.

[5] Augmenting each of the $2f^2+4f+2$ omission-tolerant nodes by $2f+1$ `vnode` instances only contributes minimal extra hardware for added elastic pipelines and the voting Muller C-Elements.

The proof that $A_{Byz}(f)$ overwhelms $A_{Om}(f)$ by a factor of $(1+\frac{1}{3})^{f+1}$ was presented in [34]. When both approaches $A^{sys}_{Byz}(f) = (3f+2)A_{Byz}(f)$ and $A^{sys}_{Om}(f) = (2f^2+4f+2)A_{Om}(f)$ are compared, the factor of $(1+\frac{1}{3})^{f+1}$ in algorithm complexity per node outweighs the factor of f in the number of nodes by far. This fact renders solution $\mathcal{S}_{Om}$ more scalable than $\mathcal{S}_{Byz}$, i.e., $A^{sys}_{Byz}(f) \geq A^{sys}_{Om}(f)$. In other words, the savings obtained by the simplification of the algorithm outweigh the overheads of the node replication required to justify the simpler algorithm.

Note, however, that although this result is representative for the standard CMOS implementations of the two presented algorithms, very different results are obtained if non-standard technologies or software implementations are used. As an example Threshold Module implementations following a design style based on "Threshold Logic" [6] would lead to linear scaling of threshold circuits, completely changing the system complexities. In case of software implementation an appropriate complexity measure is code size together with memory usage. The code size of the Byzantine-tolerant and the omission-tolerant algorithm is constant (with respect to n and f) and both only have to store the difference of ticks generated locally and remotely. Hence $A_{Byz}(f) = A_{Om}(f) \in \mathcal{O}(n) = \mathcal{O}(f)$. Thus, in both alternative cases clearly $A^{sys}_{Byz}(f) \leq A^{sys}_{Om}(f)$. On the other hand it also has to be taken into account that connectivity (link count) or performance might be the main measures for complexity and performance. In the light of these different assessment criteria clearly $\mathcal{S}_{Byz}$ will be the solution of choice. $\mathcal{S}_{Byz}$ might also produce ticks at a higher frequency since the frequency is determined by the minimum interconnection delay δ between remote nodes. In case of $\mathcal{S}_{Om}$, ticks sent by p and received by q have to pass through the `vnode` design with an additional pipeline (in front of the voting Muller C-Element) of length π, which increases this delay. In general, an implementation following $\mathcal{S}_{Om}$ might be a viable option for larger values of f and/or n.

Chapter notes

The main focus of this chapter has been on the description of the Byzantine-tolerant tick generation algorithm's functional hardware architecture — corresponding to the algorithmic statements of Algorithm 8. Each of the underlying building blocks have therefore been treated in detail. The hardware implementations introduced in this chapter represent the final results obtained via several design iterations. Each iteration involved hardware adaptations, i.e., redesign of a building block, followed by subsequent algorithmic analyses of the new design. In [39] the basic architecture including a first formal treatment has been presented, while [28] focuses on the hardware implementation aspects of the design. Both papers present early results of the DARTS project. Especially during the mapping to the ASIC library and the subsequent design validation phase some minor adaptations have been made to the hardware design. In addition, it is noteworthy that in the implementation concept phase focus had already been laid on the design of components considered critical. One of these components is the Muller C-Element which is omnipresent in the tick

counters and therefore has significant impact on the performance. Another critical building block is given by the threshold circuits which are considered to be a vital component for all implementations of fault-tolerant algorithms and therefore have been separately assessed in [37].

The substantial area effort of the Byzantine-tolerant tick generation implementation raised the question whether or not weaker failure models — resulting in less complex design — might provide reasonable fault tolerance. In [33] the author analyzed this particular subject yielding the conclusion that a substantial mismatch of distributed systems failure models and hardware related fault models exists. Due to this gap between the two "worlds" of fault and failure modeling, algorithms designed for weak failure assumptions cannot be directly applied in hardware implementations. However, using the Byzantine fault-tolerant implementation's building blocks as a starting point, feasible tick generation alternatives might be derived based on carefully simplified algorithms. The analysis presented in [34] reveals that the threshold circuits' unfavorable scaling in number of nodes n might render transformations of weaker algorithms more efficient than an innately Byzantine-tolerant implementation. However, it has to be considered that the appropriateness of an approach strongly depends on the underlying implementation technology and cost function (e.g., a threshold circuit implementation different to the employed sum of products design might completely change the above presented picture).

Chapter 5

On-chip Evaluation and Measurement Setup

Measure what is measurable, and make measurable what is not so.

Galileo Galilei

THE PROTOTYPE ASIC implementation of the Byzantine-tolerant tick generation approach is partitioned in a way that every TG-Alg node is mapped to a single ASIC — the so-called *Hardware Implemented Tick Synchronization (HITS)* chip. Thus, a tick generation system comprising several TG-Algs, e.g., 8 nodes, consists of a set of 8 chips interconnected on a printed circuit board (PCB). This architecture allows that all links between TG-Alg instances (=TG-Net) can be accessed easily. However, to be able to assess all relevant properties and characteristics of the tick generation approach, additional measurement support has to be added on-chip. When following this strategy it certainly has to be ensured that adding auxiliary hardware and measurement access does not spoil function and/or performance of the tick generation process itself. Any kind of unwanted probing effects have to be minimized by all means to allow for reasonable evaluations of the tick generation approach. Clearly, making internals of a chip accessible from outside has some (at best only minor) influence on the whole design. Therefore the strategy for the prototype evaluation is twofold in the sense that every HITS chip comprises two completely independent TG-Alg instances. One of these TG-Alg designs, further called standard node (stdnode), only provides the bare minimum of interaction with the environment, while the second instance, referred to as experimental node (expnode), is augmented with evaluation specific circuitry and allows enhanced access to TG-Alg internals. This way function and performance of the standard node design are not influenced by probing effects and should therefore deliver unaltered TG-Alg behavior, while the experimental node can be used to artificially generate special operating conditions within a TG-Alg to enable more elaborate investigations.

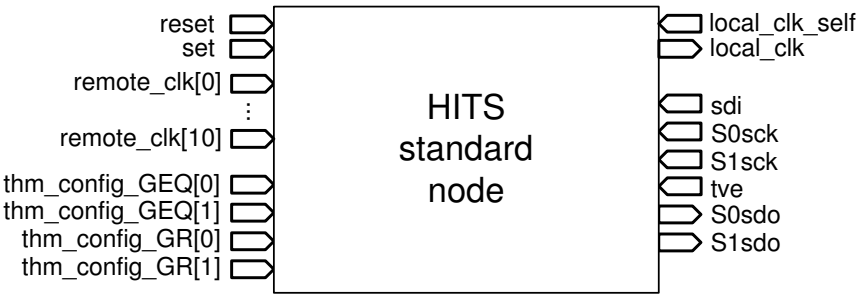


Figure 5.1: Interfaces to the HITS standard node design

5.1 Standard node

As indicated above, the standard node design is targeted at implementing the tick generation core functionality only. That is, as few additional interfaces and/or hardware as possible should be added to the core design since this could possibly influence performance and/or proper operation. Thus, in essence the stdnode implements the circuit blocks presented in Figure 4.1 which have been described in detail in Chapter 4. A few augmentations to the schematic of Figure 4.1, however, had to be made to fully implement all design requirements. At first, input signals enabling the configuration of the Threshold Modules for different threshold values had to be added. Signals `thm_config_GEQ` and `thm_config_GR` implement the switching between different threshold values for the $2f + 1$ and $f + 1$ threshold circuits, respectively. The configuration patterns for these signals are given in Table 5.1. Further enhancements to the standard node design are given by hardware blocks which allow for convenient factory testing of the asynchronous TG-Alg. These test facilities may also be used for evaluation purposes and are therefore treated in more detail in Section 5.1.1. The input and output ports of the standard node design are depicted in Figure 5.1. The `reset` and `set` signals as well as the factory test signals are the only additional interfaces besides the obviously mandatory `local_clk` output and the `remote_clk, local_clk_self, thm_config_GEQ/GR` inputs. The peculiarity of dedicated `reset` and `set` inputs originates in the ASIC library cell of the Muller C-Element (introduced in Section 4.2.1) and allows to choose whether the tick generation starts with TICK($\downarrow$) or TICK($\uparrow$) messages.

Table 5.1: Threshold configuration for (a) $2f + 1$ and (b) $f + 1$ circuits

(a)

`thm_config_GEQ[0]`	0	1	0	1
`thm_config_GEQ[1]`	0	0	1	1
threshold	-	3	5	7

(b)

`thm_config_GR[0]`	0	1	0	1
`thm_config_GR[1]`	0	0	1	1
threshold	-	2	3	4

5.1.1 Test support

Factory test facilities have been added at several points of the standard node design to enable a high test coverage for single stuck-at faults. This test circuitry, however, can also be used for measurement purposes like the evaluation of propagation delays of a TG-Alg's internal units. The strategy of the test approach follows the commonly used scan chain based design. In this context it has to be noticed that as a consequence of the asynchronous (mostly transition signaling based) implementation of the TG-Alg design, no flip-flop registers are available for the use in the scan chain. Hence, to implement the scan approach registers together with the necessary scan logic had to be added artificially. Figure 5.2 depicts the placement of the scan chains **S0** and **S1** within the standard node design. To satisfy the demanded minimal interference with the core TG-Alg design, different types of scan chain elements (scan cells) have been added. The scan cell types are distinguished by their interfacing capabilities. Three types can be identified in the standard node design ranging from a straight observe function, over the possibility to control/alter signals, to a design able to observe as well as control signals. Observe-only scan cells will clearly have the least impact on the TG-Alg design since observe cells do not add additional delays to the signal paths. However, facilitating reasonable test coverage of the factory test requires control over several TG-Alg internal signals. Therefore, scan chain **S1** comprises control scan cells to provide the Pipe Compare Signal Generation circuit with test data. Furthermore, **S0** holds observe/control cells which can in one case be used to observe the operation of the PCSG, while for other test scenarios controlling the input to the threshold circuits. The observe-only scan cells of the **S0** and the **S1** scan chains allow to record the outputs of the remote pipeline and the threshold circuits, respectively. Of special interest for the evaluations and measurements is the fact that in both cases the scan data output signals (`sdo0` and `sdo1`) can be used to assess the respective propagation delays of the remote pipeline as well as the threshold circuits.

5.2 Experimental node

The experimental node TG-Alg design essentially implements the same architecture as the standard node. However, several adaptations have been made, facilitating advanced experiment and measurement support. One of the most noticeable changes is given by the fact that in contrast to the standard node design the local clock feedback is split up into dedicated signals for each of the elastic pipeline pairs — this way enabling full control over the local clock feedback path with the possibility to use distinct propagation delays for each pipeline pair. An example application is the matching of local clock delays with propagation delay of the respective remote clocks. Figure 5.3 gives an overview on all input/output ports of the experimental node design. To improve readability and prevent confusion with the standard node design, all experimental node signal names have a leading `x`, e.g., `xlocal_clk` denotes the local clock output of the expnode design. The remote clock inputs, reset, set and the configuration signals of the threshold circuits share the same

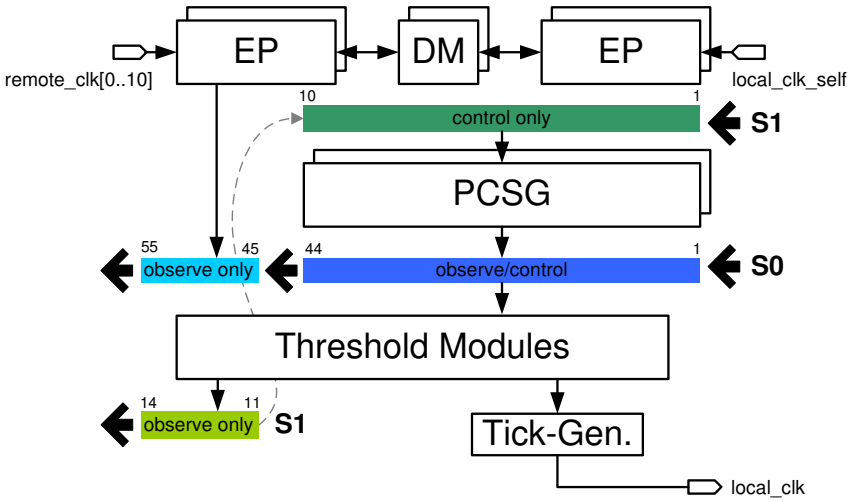


Figure 5.2: Standard node scan chain overview

function as in the standard node. Likewise, the splitting of the `xlocal_clk_self` signal, another general adaptation to the TG-Alg architecture of Figure 4.1, is given by the fact that eight pipeline stages have been used for both the remote and local elastic pipelines. This augmented buffering depth may be used for enhanced precision experiments, dragging TG-Alg clocks artificially apart without losing ticks. Furthermore, in conjunction with the reset/set scan chain presented below (Section 5.2.2) special start-up scenarios can be implemented. Clearly, adding four additional stages per elastic pipeline increases the propagation delay through the TG-Alg, thus performance and correctness constraints had to be reassessed for the modified expnode design timings. Besides the general changes to the TG-Alg design presented above more specific ones, especially facilitating evaluation and measurement access, have also been made. The following sections will introduce and describe the experimental node's special on-chip evaluation support hardware blocks.

5.2.1 Test support

Analogously to the standard node, support for factory testing has also been added to the expnode design. However, due to the fact that non-interference with the TG-Alg design, especially with respect to performance, is not as critical as in the standard node, several adaptations could be applied. Figure 5.4 depicts the resulting scan chain architecture. The previously employed observe-only and control-only scan cells have been replaced by observe/control cells. On the one hand this introduces small delays in the TG-Alg's signal paths, but on the other hand it also enables improved access to the internals of the design. Scan chain **S0** allows to observe and control the fill-level signals GEQ^e/GEQ^o as well as the GR^e/GR^o, while **S1** is able to evaluate and set the threshold gate outputs q_1, q_2, q_3 and q_4 (note that these signals are also mapped to the output ports `xthm_q1, xthm_q2, xthm_q3, xthm_q4`). Furthermore, scan chain **S1** also has read and write access to all

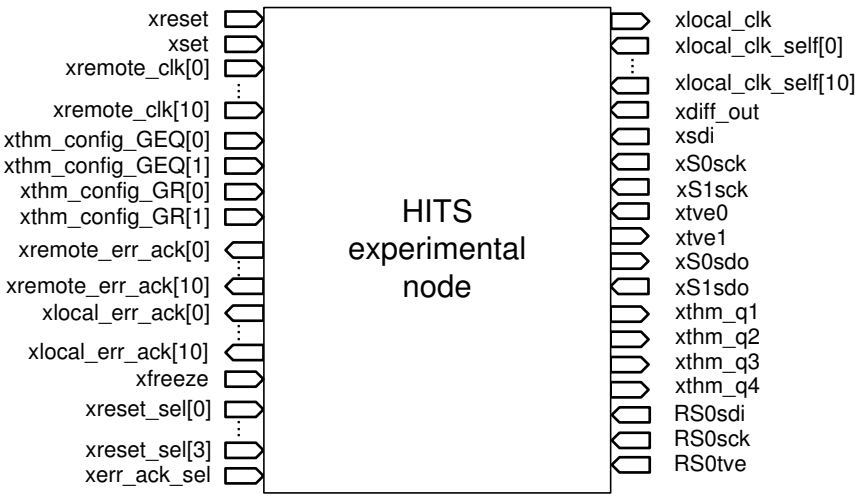


Figure 5.3: Interfaces to the HITS experimental node design

input signals of the PCSG units. This enhanced access to TG-Alg internals becomes very handy when assessing component delays under varying operating conditions. The **RS0** scan chain which is also depicted in Figure 5.4 has a special purpose and is not related to factory testing. In fact this design is responsible for generating customized reset patterns for the Counter Modules and will be further described in the next section.

5.2.2 Reset/set scan chain

As indicated above the reset/set scan chain allows to configure user defined reset/set patterns for the remote and local elastic pipeline as well as the Difference Module initialization. By virtue of these patterns the expnode TG-Alg's +/− Counters can be preloaded with up to eight clock ticks which is especially interesting for speed tests.

A building block closely related to the reset/set scan chain is the reset-selector block. This design unit is responsible for choosing the pipeline pair(s) to be initialized by the reset/set scan chain. In addition, the reset-selector's Counter Module selection also maps the respective `SRemote3` signal to the `xdiff_out` port. The remote pipelines' `SRemote3` signals can be of special interest since the toggling indicates that a tick has been removed from both the local and the respective remote pipelines. Table 5.2 summarizes the configuration patterns for the reset-selector.

5.2.3 Freeze logic

The fact that the whole TG-Alg node is implemented in asynchronous logic makes it hard to obtain a node's current state since it cannot easily be stopped. In a synchronous design one would delay/halt the global clock signal to gather a node's state information. In order

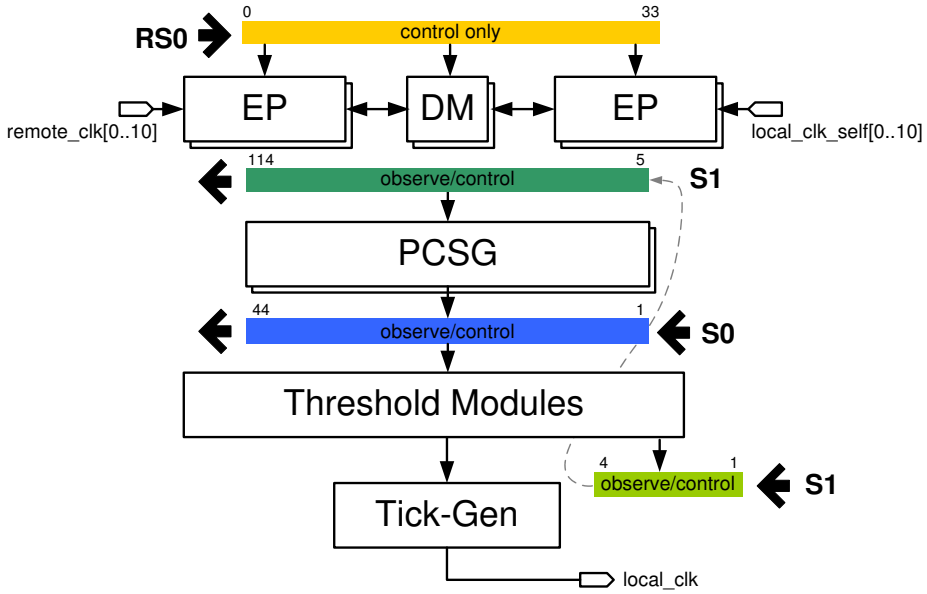


Figure 5.4: Experimental node scan chain overview

Table 5.2: Reset-selector configuration patterns

`xreset_sel`[0...3]				active
[3]	[2]	[1]	[0]	Counter Module
0	0	0	0	unused
0	0	0	1	Counter 1
0	0	1	0	Counter 2
0	0	1	1	Counter 3
0	1	0	0	Counter 4
0	1	0	1	Counter 5
0	1	1	0	Counter 6
0	1	1	1	Counter 7
1	0	0	0	Counter 8
1	0	0	1	Counter 9
1	0	1	0	Counter 10
1	0	1	1	Counter 11
1	1	0	0	unused
1	1	0	1	unused
1	1	1	0	unused
1	1	1	1	all counters

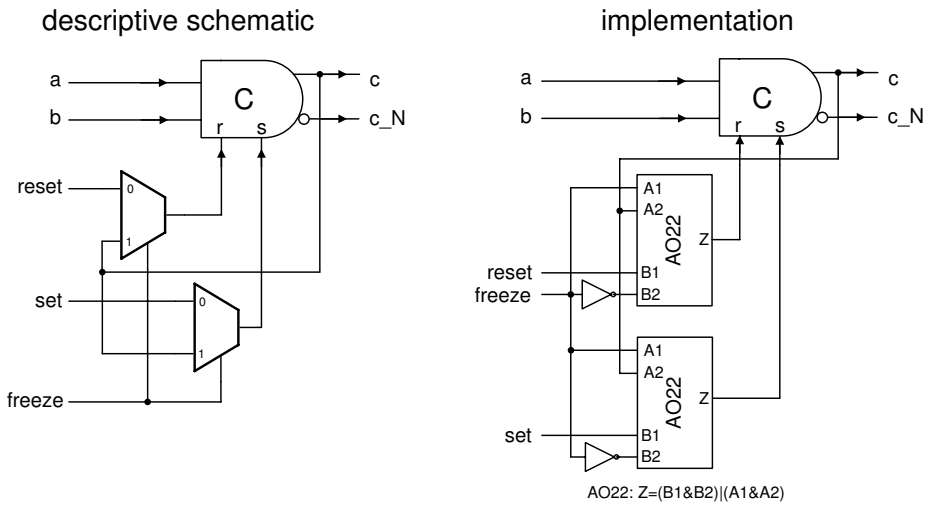


Figure 5.5: TG-Alg halting mechanism via Muller C-Element freeze logic enhancement

to achieve observability and controllability in asynchronous design, the design's self-timed loop has to be broken. To implement such a mechanism able to stop all further processing within the TG-Alg, the expnode design's state holding elements had to be enhanced. The vast majority of a TG-Alg's components are stateless, i.e., pure combinatorial logic. The only state information is held by Muller C-Elements[1]. Hence, to inhibit further tick processing the Muller C-Elements have to be kept in their current state until the state has been gathered. Figure 5.5 shows the circuit which augments all C-Elements of the expnode design, i.e., the Muller C-Elements of the Counter Modules and the one in the Tick Generation Module. The halting mechanism is implemented by taking advantage of the Muller C-Elements' `set` and `reset` inputs. On activation of the `freeze` signal (active high), a logic high at the output of the C-Element is fed back to the `set` and `reset` inputs, this way retaining the high value regardless of input changes. Analogously, a logic low will also be safely stored until the `freeze` signal is deactivated again.

In conjunction with the reset/set scan chain presented before the freeze logic, for instance, allows to force the tick generation system into an otherwise transient state, thus allowing to assess the circuits correctness and general behavior in detail.

5.2.4 Pipeline extension and overflow detection

As mentioned above, the remote and local elastic pipelines have been enlarged to eight stages in contrast to the four stages of the standard node design. These additional stages can be used to evaluate the tick generation approach with (artificially) exceedingly increased jitter. Furthermore, the expnode design provides the possibility to externally extend the pipeline depths, e.g., via an FPGA design. To enable this pipeline exten-

[1]The scan chain flip-flops are not accounted for since they are not part of the tick generation process.

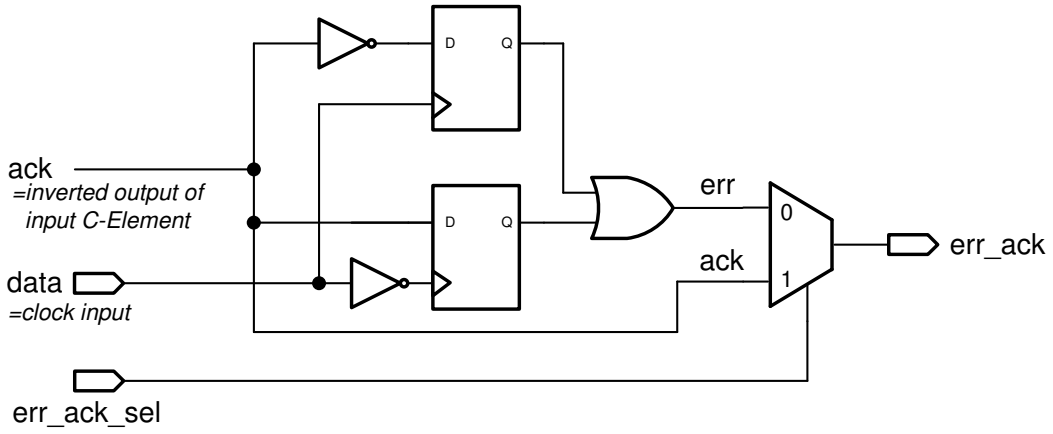


Figure 5.6: Pipeline extension and overflow detection circuit

sion the acknowledge signals of the elastic pipeline inputs — which are not used in the standard node design — are connected to output pins of the HITS experimental node (`xremote_err_ack[0...10]` and `xlocal_err_ack[0...10]`). The specific feature of these output signals is given by the fact that they can be configured to implement two different modes. One configuration maps the input Muller C-Element's acknowledge signal to the output and enables the above described external pipeline extension. In contrast, the other operation mode allows to detect whether or not the respective pipeline has encountered an overflow, i.e., the de-synchronization of the corresponding local and remote clocks is greater than the pipeline depth. The schematic of the pipe-extension/overflow-detection circuit is shown in Figure 5.6. The expnode's `xerr_ack_sel` signal determines whether the pipeline extension or the overflow detection mode is activated. While the pipeline extension is implemented by handing the acknowledge signal to the output port, the overflow detection is more intricate. To illustrate the operation of the overflow detection, the following example execution will be considered. At first it is assumed that the clock input Muller C-Element of the elastic pipeline stores a logic low state and that its pipeline internal input is high, thus enabling the propagation of a ↑-transition at the clock input. If a rising transition arrives at the clock input, the inverted `ack` signal (=low) will get latched into the upper latch leaving the error signal deactivated. In case the pipeline is full after this ↑-transition has arrived, a subsequent TICK(↓) will not be able to pass the Muller C-Element. However, this does not pose a threat since no clock transition has been lost so far. The last low pulse will however be lost if another TICK(↑) arrives at the pipeline input. This pipeline overflow is immediately detected due to the fact that the inverted `ack` signal, which is high, will be latched with the rising edge of the input transition.

Chapter notes

It can be a quite intricate problem to assess whether or not an asynchronous design is operating properly. To enable a high degree of testability and by the same means additionally gain access to vital internals of the HITS chip, an elaborate scan chain approach, presented in detail in [82], has been employed. In the standard node design the introduced test infrastructure provides minimally invasive access to some internals. As a result the standard node design directly reflects the pure tick generation algorithm's operation. Experiments for validation of characteristics predicted by theory do not benefit from the restrictive design. Consequently, to enable the assessment of critical operating conditions of a running DARTS tick generation architecture the experimental node had to be enhanced by dedicated on-chip evaluation support units.

Chapter 6

Experiment Specifications and Theoretical Foundation

> A theory is something nobody believes, except the person who made it.
> An experiment is something everybody believes, except the person who made it.
>
> Albert Einstein

THE IMPLEMENTATION of the Byzantine-tolerant tick generation approach led to the previously presented HITS (Hardware Implemented Tick Synchronization) ASIC. Until now the characteristics of this design have only be assessed regarding hardware effort of the mentioned TG-Alg implementation. Therefore, the assessment strategies presented in this chapter aim at thoroughly characterizing the properties of a running tick generation system. In general, the evaluations of the proposed tick generation scheme have to cover many different aspects and properties of the design. For instance, it starts with the above mentioned validation of worst-case properties and ranges over average case operation modes, to scenarios with varying operating conditions. Given this huge area of interest which has to be covered it is apparent that no single evaluation approach, but rather different specialized schemes have to be employed. Therefore, the experimental assessment of the DARTS tick generation design will be divided into the three main segments: worst-case measurements, average-case measurements and supportive simulations. The worst-case assessments allow to validate bounds for the tick generation system's correctness as well as performance measures via accurately defined measurement scenarios. In the average-case measurements typical operating conditions which might also vary are used for system characterization. Complementary to the first two assessment strategies, the supportive simulations conveniently allow to estimate the tick generation system's behavior under operating conditions which are under full control of the user.

Following the introduced partitioning, confirmation whether or not the theoretical predictions for critical performance and correctness measures (introduced in Section 3.4.3) hold can for instance be performed via measurements. In the context of average-case experiments the assessment of implementation and operation characteristics might be of interest. In particular, stability considerations arise when recalling that the primary goal of the DARTS clocking scheme is to provide conventional synchronous circuits with a fault tolerant clock. On the one hand the asynchronous nature of the TG-Alg implementation allows the design to adapt its operation to varying conditions, thus increasing its robustness. On the other hand this flexibility might be problematic from the synchronous unit's point of view since it is controlled by the adaptive, thus varying TG-Alg clock. Therefore the evaluation of properties which reflect the dynamic behavior of single TG-Algs as well as the whole ensemble of nodes seems to be indicated. In addition, special cases like the booting of the tick generation process as well as numerous fault scenarios must also be covered by measurements and compared to theoretical results and simulations. As a consequence of the broad spectrum of evaluations all relevant properties of the tick generation scheme have to be properly defined in advance, thus enabling systematic measurements of the tick generation's key characteristics.

6.1 Worst-case properties

The employed evaluation strategy for worst-case conditions quite naturally follows the formal analysis of the tick generation approach. Section 3.4.3 introduced the most relevant timing constraints as well as correctness and performance measures. In the subsequent paragraphs scenarios are derived which explicitly force the tick generation system into worst-case operation modes for the parameter under evaluation. This enables a comparison of the theoretically predicted and actually measured parameters. As a consequence of the tick generation scheme's fault-tolerant architecture, i.e., masking of exceedingly slow and/or fast nodes, worst-case conditions, however, can only be attained in scenarios comprising faults. The main purpose of the conducted experiments is to validate certain properties of the hardware design which a priori have been derived by formal proofs. Clearly, the evaluations cannot be exhaustive, however, selected critical points can be assessed in detail and compared to the theoretical results.

6.1.1 Precision

The synchronization precision π of the tick generation system may simply be assessed by measuring the clocks' relative offset (cf. Figure 2.1(b)). However, this evaluation is unlikely to reflect worst-case conditions, thus an appropriate scenario has to be derived and established. Figure 6.1(a) shows the generic setup to statically force a system of $3f+2$ TG-Alg nodes into an operation mode with worst-case precision. The only relevant parameters for this scenario are given by the interconnecting remote delays τ_{rem}. As depicted, a set of

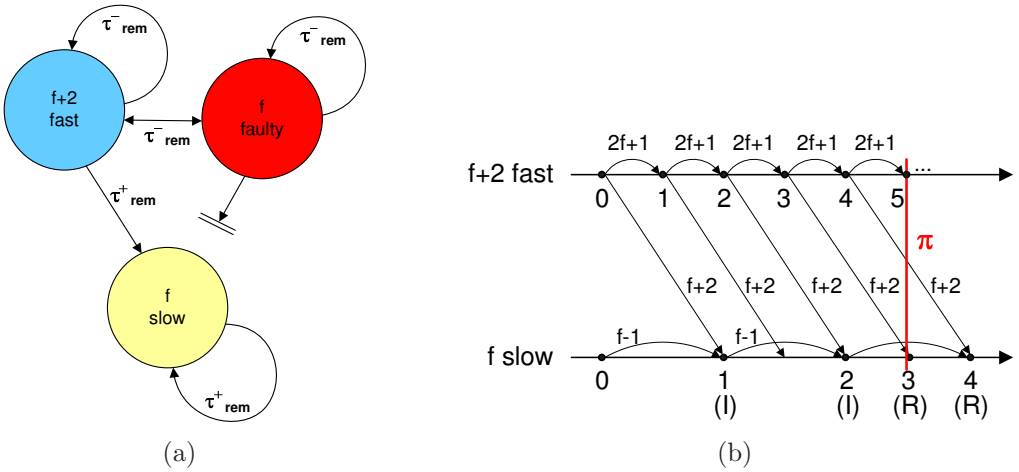


Figure 6.1: Evaluation scenario to attain worst-case precision π

f nodes have to be *faulty* in the way that no TICK(k) messages are delivered to a second set of f *slow* TG-Algs. It further has to be ensured that ticks sent among the set of slow nodes as well as those received from the group of $f + 2$ fast TG-Algs are issued with the maximum remote delay τ^+_{rem}. Connections not explicitly shown in Figure 6.1(a) can be assumed to have delay τ^-_{rem}. More formally speaking, in a system where P denotes the set of all nodes there are three distinct sets of TG-Algs with A comprising the fast nodes, B the slow, and the F faulty ones. The remote delays from p to q in this setup are given by:

$$\begin{aligned}
\tau_{rem}(p \in A, q \in B) &= \tau^+_{rem} \\
\tau_{rem}(p \in F, q \in B) &= +\infty \\
\tau_{rem}(p \in B, q \in B) &= \tau^+_{rem} \\
\tau_{rem}(p \in P, q \in A) &= \tau^-_{rem} \\
\tau_{rem}(p \in P, q \in F) &= \tau^-_{rem}
\end{aligned}$$

To get a better understanding for the reasons why this static evaluation setup represents a valid worst-case scenario for the tick generation system, Figure 6.1(b) depicts an execution trace of the relevant (non-faulty) nodes. As indicated in the trace, it is assumed that all nodes start at approximately the same time by issuing TICK(0). For the example, it is assumed that τ_{rem} alone determines the processing speed of the tick generation system[1]. In the given setup, set A comprises $f + 2$ fast TG-Algs. Together with f fast, but faulty nodes $\in F$, ticks are generated continuously at a rate determined by τ^-_{rem} and according to the algorithm's "Increment Rule" ($=2f + 1$ threshold). Analogously, the f slow TG-Algs $\in B$ also start to issue clock ticks triggered by the "Increment Rule" (I), however, at a

[1]Recall from Section 4.1 that only the ratio Θ of fastest to slowest path determines the algorithm's properties, thus it makes no difference if τ_{rem} or the whole delay of the tick generation path are considered in the experiment scenarios.

period determined by τ^+_{rem}. Thus, group A starts "running away" with τ^-_{rem} while the slow group B "runs behind" with period τ^+_{rem}. Further examining this setup reveals that the slow nodes' flow of issuing ticks at some point changes to the operation mode where the "Relay Rule" (R) takes over tick generation. This switching point is reached when TICK(k) messages arrive at the slow nodes, indicating that the fast remote nodes are ahead by at least one tick, i.e., $k > \ell$, with ℓ being the current local tick number. This way the "Relay Rule" ensures that the system stays in a synchronized state. The maximum offset in time between the first sending of TICK(k) at t_k and the last sending of TICK(k) at t'_k for any pair of correct nodes p, q can be bounded by:

$$|t_k - t'_k| \leq \tau^+_{rem}, \text{ if } f \leq 1$$
$$|t_k - t'_k| \leq (\tau^+_{rem} - \tau^-_{rem}) + \tau^+_{rem}, \text{ if } f > 1$$

These bounds are obviously only estimates based on the simplification that τ_{rem} forms the only relevant timing parameter. For more precise calculations of the maximum clock skew of correct nodes it has to hold that,

$$|t_k - t'_k| \leq T_P + T_{QS}$$

with T_P and T_{QS} being timing parameters derived from the formal analysis of the synchronization properties *Progress* and *Quasi-Simultaneity* (cf. Section 3.4.4 and [39,40]). Finally, by relating the appropriate timing paths to each other the clock skew considerations can be transformed to attain the tick generation system's precision π.

$$\pi = \left\lceil \frac{T_{QS}}{T^-_{first}} \right\rceil + 1$$

6.1.2 Accuracy

In the context of clock synchronization the property named accuracy represents the linear envelope function for the progression of a single clock (cf. Figure 2.1(a)). In other words, it bounds a correct node's minimum and maximum offset in time between the generation of subsequent local ticks. Thus the difference $|t_{k+1} - t_k|$ is bounded by accuracy, with t_k and t_{k+1} denoting the generation instants of TICK(k) and TICK($k+1$) messages, respectively. In contrast to precision π, accuracy denotes an individual local property of every correct TG-Alg. To be able to assess the predictions based on theory for the lower and the upper bound of accuracy (presented in Section 3.4.4) once again appropriate evaluation scenarios have to be identified. However, unlike the assessment of precision π, the accuracy evaluations require dynamic setup configurations to enable worst-case conditions. Analogously to the precision scenario, the accuracy considerations rely on the simplification that τ_{rem} denotes the only parameter influencing the tick generation process. The measurement of the lowest possible rate for generating consecutive ticks, that is, the maximum offset between local TICK(k) and TICK($k+1$) messages, follows a strategy depicted in Figure 6.2.

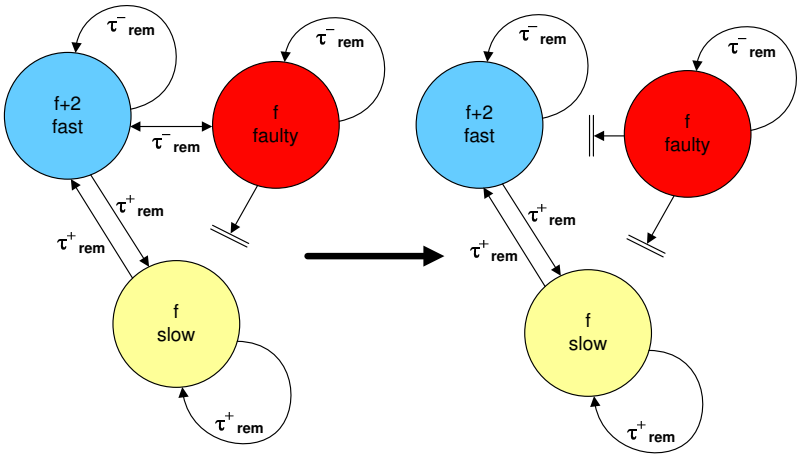


Figure 6.2: Evaluation setup to attain worst-case lower bound for accuracy

Again, similarly to the precision experiment, three disjoint sets of nodes can be identified. Set A consisting of $f+2$ fast TG-Algs, group B comprising f slow nodes and the third set F denoting f faulty nodes (which never send messages to TG-Algs of set B, but apart from that execute the tick generation algorithm). The remote delays of this setup are given below:

$$\begin{aligned}
\tau_{rem}(p \in A, q \in A) &= \tau^-_{rem} \\
\tau_{rem}(p \in A, q \in B) &= \tau^+_{rem} \\
\tau_{rem}(p \in B, q \in B) &= \tau^+_{rem} \\
\tau_{rem}(p \in B, q \in A) &= \tau^+_{rem} \\
\tau_{rem}(p \in P, q \in F) &= \tau^-_{rem} \\
\tau_{rem}(p \in F, q \in B) &= +\infty \\
\tau_{rem}(p \in F, q \in A) &= \tau^-_{rem} \Rightarrow +\infty
\end{aligned}$$

An important detail in this setup is given by the fact that $\tau_{rem}(p \in F, q \in A)$ changes its value from τ^-_{rem} to $+\infty$ during operation, i.e., the faulty nodes of set F stop contributing to the $f+2$ fast nodes' tick generation process. Taking a look at the example shown in Figure 6.3 reveals that at first the tick generation acts similarly to the worst-case precision experiment, leading to the maximum offset between TG-Alg nodes of set A and B. However, when changing the aforementioned delay $\tau_{rem}(p \in F, q \in A)$ from τ^-_{rem} to $+\infty$, the $f+2$ fast nodes from set A cannot make any further progress for the moment. This is because the required $2f+1$ threshold of the "Increment Rule" can no longer be reached without the assistance of the f slow nodes from group B. Thus tick generation in set A is stalled until the slow nodes of set B catch up with the formerly fast nodes. This halting period for TG-Algs in group A starts with TICK(k) being the last tick generated on the fast track (driven by the faulty nodes). The end of the interval is given by the time when

all TICK(k) messages have arrived via the slow links $\tau_{rem}(p \in B, q \in A) = \tau^+_{rem}$, since this enables the generation of the subsequent TICK(k+1).

The assessment of the minimum offset between two consecutive ticks, i.e., the upper bound for the tick generation rate of a single TG-Alg in the system, can be obtained analogously to the lower bound experiments. The setup for a scenario able to force a node of the tick generation system into this worst case is depicted in Figure 6.4, the corresponding example trace is presented in Figure 6.5. As in the above lower bound scenario, the tick generation process first starts to diverge until the maximum offset π between the groups of fast and slow nodes is reached. To facilitate this behavior, a setup identical to the precision experiment forms the starting point of this assessment. A switch to the other set of delays (presented below) can be made once the maximum tick offset is reached, i.e., the slow nodes of set B are lagging behind and generate ticks on behalf of the "Relay Rule" only.

$$\begin{aligned}
\tau_{rem}(p \in A, q \in B) &= \tau^+_{rem} \Rightarrow \tau^-_{rem} \\
\tau_{rem}(p \in F, q \in B) &= +\infty \Rightarrow \tau^-_{rem} \\
\tau_{rem}(p \in B, q \in B) &= \tau^+_{rem} \\
\tau_{rem}(p \in P, q \in A) &= \tau^-_{rem} \\
\tau_{rem}(p \in P, q \in F) &= \tau^-_{rem}
\end{aligned}$$

Changing $\tau_{rem}(p \in A, q \in B)$ from τ^+_{rem} to τ^-_{rem} and $\tau_{rem}(p \in F, q \in B)$ from at first being not connected ($= +\infty$) to τ^-_{rem} provides the formerly slow nodes with TICK() messages arriving at a higher rate. In fact, the switching to fast links almost immediately reduces the clock offset between nodes of set A and B. Moreover, TG-Algs of set B are enabled to generate ticks via the algorithm's "Increment Rule" at a pace determined by the incoming links' delay τ^-_{rem}.

From the theoretical analysis the lower and upper bounds for accuracy can be derived via Equation 3.4. The lower bound part is specified by the slowest possible correct path for generating a tick, i.e., essentially timing path T_P according to the synchrony property *Progress* (cf. Section 3.4.4). In contrast, the upper bound is based on the fastest

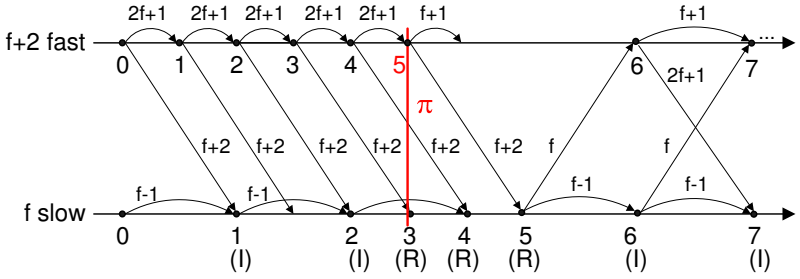


Figure 6.3: Example trace for lower bound for accuracy

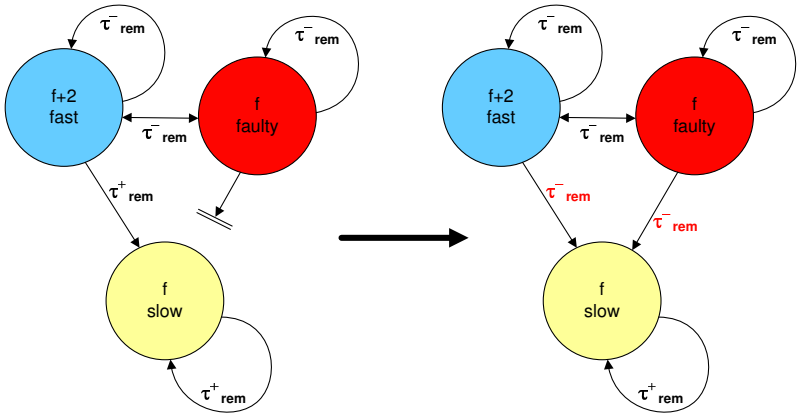


Figure 6.4: Evaluation setup to attain worst-case upper bound for accuracy

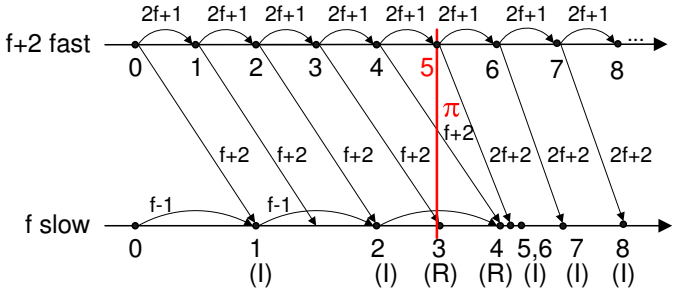


Figure 6.5: Example trace for upper bound for accuracy

remote (T^{-}_{first}) respectively local (T_{min}) tick generation path, with the smaller of the two determining the bound.

6.1.3 Slowest and fastest progress

After the assessment of the local accuracy characteristics this section is concerned with somewhat similar, but global properties. The slowest and fastest progress give system wide bounds for the time between the generation of subsequent ticks TICK(k) and TICK(k+1). In detail, the slowest progress among correct TG-Algs is given by the maximum offset in time between local and global ticks. To further clarify this, let p be the last correct process to send TICK(k) (at time t_k) and q (possibly $p = q$) the last correct process to send TICK(j) (at time t_j), with $j = k + N$ and $N \geq 0$. Then theory in [40] predicts that $t_j - t_k \leq NT_P$ which for the simplified case of predominant remote delays translates to $(\tau^{+}_{rem} - \tau^{-}_{rem}) + (k - j + 1)\tau^{+}_{rem}$. Analogous to the slowest progress, the minimum offset in time between global ticks corresponds to the term fastest progress. For the assessment of the fastest progress let p be the first correct process to send TICK(k) (at time t_k) and

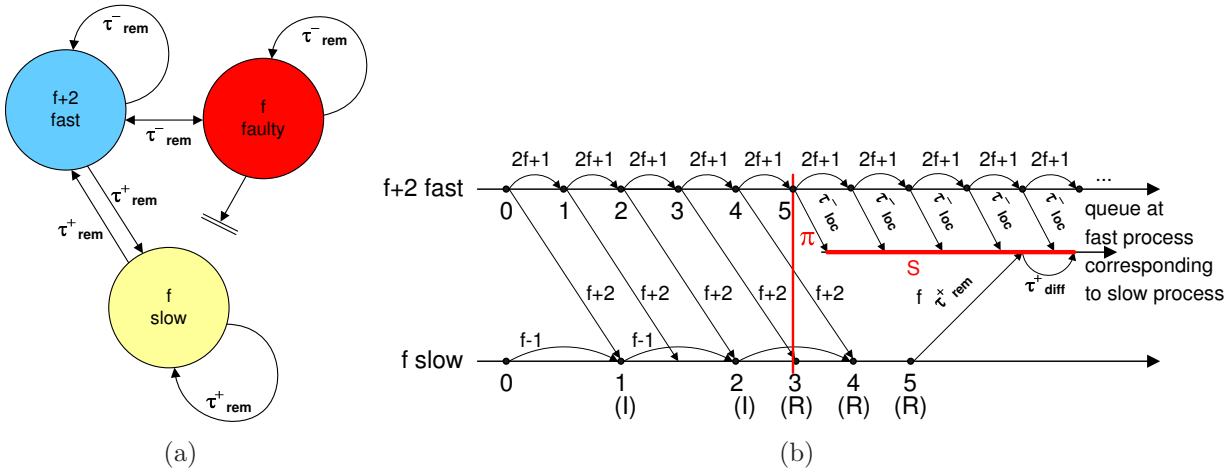


Figure 6.6: Evaluation scenario to assess local queue size constraint

q the first correct process to send TICK($k+1$) (at time t_{k+1}). Then it has to hold that $t_{k+1} - t_k \geq T^-_{first}$ (for the simple case $t_{k+1} - t_k \geq \tau^-_{rem}$).

It should be noted that in the context of the proposed tick generation scheme the considerations regarding slowest and fastest progress for the case of $p = q$ correspond to the above presented accuracy treatment, i.e., the same experiment scenarios apply. In essence the derived accuracy denotes an upper and lower limit for the slowest and fastest progress, i.e., $p = q$ represents the worst case for these properties. In the general case of $p \neq q$ the linear envelope given by the slowest and fastest progress might be significantly smaller than the one defined by accuracy.

6.1.4 Queue size

It is essential for the tick generation process that no clock ticks are ever lost at correct nodes. The remote and local elastic pipelines with their capability to store a fixed number of ticks have to guarantee this property. In Section 3.4.4 bounds for the required pipeline depths depending on several path delays have already been presented. The assessment of these bounds can be performed by artificially forcing TG-Alg nodes into different worst-case conditions. Figure 6.6 presents the setup and an example trace for the evaluation of the local queue size. In this static configuration the fast nodes' local queues which correspond to the slow nodes are filled to a maximum. The delay configuration ensures that set B of f slow TG-Algs lags behind group A of $f + 2$ fast nodes by the maximum precision π. Moreover, for the delivery and processing of TICK() messages from TG-Algs of the slow set B to the fast group A an additional delay of $\tau^+_{rem} + \tau^+_{Diff}$ is added. This delay represents the time from tick generation at the slow node until the removal of the corresponding tick at the fast node's local pipeline.

To be able to evaluate the correctness of theoretical predictions of the remote pipeline size, the setup and scenario shown in Figure 6.7 can be applied. In this configuration a distinctive fast node w with slightly different delay paths than the other fast TG-Algs has to be assumed. In the given setup the f slow nodes' remote pipelines corresponding to node w are filled at the highest possible rate. At the same time these nodes' slow local feedback τ^+_{loc} results in the fact that they lag the maximum precision π behind TG-Alg w.

To assess the exact border case beyond which the queue size will no longer be sufficient the described experiments can be performed iteratively with increased values of Θ, e.g., by step-wise increasing τ^+_{rem}.

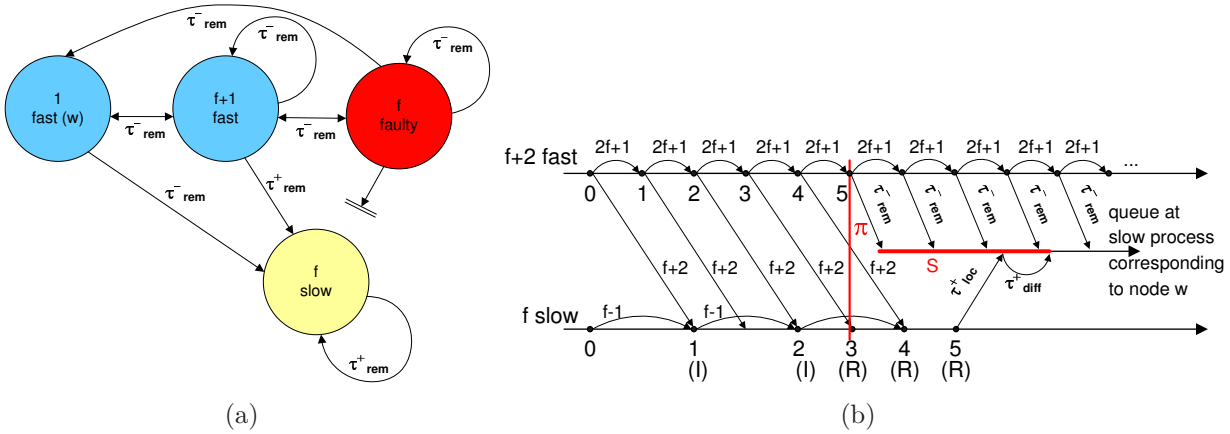


Figure 6.7: Evaluation scenario to assess remote queue size constraint

6.1.5 Booting

The booting constraint has been introduced in Section 3.4.3 and forms a crucial condition for the correct behavior of the tick generation scheme. If a node starts up later than τ^-_{rem} it might have already missed ticks from fast remote nodes. Unfortunately, the anonymous TICK($\downarrow$)/TICK($\uparrow$) messages hide the information of lost ticks. Thus, once a message is lost the corresponding node has to be considered faulty and hence consumes part of the fault tolerance budget. Therefore, partial or substantially skewed booting of the tick generation system's TG-Algs has to be prevented.

Considering the assessment scenarios described in the previous subsections, it can be noticed that all TICK(0) messages are issued at exactly the same instant. If this simplification is removed and nodes start within the admissible interval $[0, \tau^-_{rem}]$ a small shift in some executions has to be accounted for. However, beyond this no additional effect on correctness and performance bounds has to be expected.

6.2 Average case properties

The above presented evaluation scenarios of conservative worst-case considerations are important to verify the predictions from theory with the chip implementation. Nevertheless, the clocking scheme will mostly be operated under less malicious conditions. Thus the assessment of average-case properties of a single TG-Alg as well as the whole tick generation system seems to be reasonable. Especially the evaluation of the *normal* operation mode, i.e., fault-free and under nominal operating conditions, with focus on the asynchronous implementation's sensitivity to parameter variations are of increased interest. Considering the application field of the tick generation scheme, e.g., clocking of replicated synchronous circuits, the main parameters of interest will be the attainable frequency and the respective short and long term stability (jitter).

6.2.1 Operating condition dependence

As indicated above, the TG-Alg implementation follows a fully asynchronous CMOS implementation. Due to this fact a certain degree of operation parameter sensitivity can be expected. In particular, the switching speed of the circuits is likely to be a function of supply voltage. Moreover, digital CMOS circuits are also known to be sensitive to temperature variations. Both effects are typically encountered in normal operation modes. Therefore a thorough characterization of the tick generation scheme regarding voltage and temperature effects seems to be reasonable. The mentioned voltage dependence of a CMOS circuit can be approximated by deriving the delay times for a single gate and essentially boils down to

$$t_{gate} \approx \frac{C_L}{\beta\, V_{DD}}, \tag{6.1}$$

with C_L being the load capacitance, β and V_{DD} representing the CMOS transistors' gain and supply voltage, respectively [91]. Note that the above mentioned temperature dependence of CMOS circuits is hidden inside β. The carrier mobility (electrons and holes) decreases with increasing temperature, thus β decreases, yielding a slowing down of the circuit as temperature rises.

Average frequency: The attainable clock frequency solely relies on switching delays of the asynchronous circuits and interconnection delays of the remote and local clock lines. Using predictions from theory the tick generation scheme's average frequency can be bounded by the earlier introduced synchronization property Progress (P) together with the tick generation path T^{-}_{first}.

$$f_{average} = \left[1/2T_P, 1/2T^{-}_{first}\right] \tag{6.2}$$

The path given by T_P denotes the slowest possible generation of a subsequent tick, while T^{-}_{first} represents the fast remotely triggered tick generation. The required delay

parameters can be extracted from the ASIC design files. Together with delays for the chip interconnect this is sufficient to give a sound estimation of the average clock frequency. For the standard HITS design, $T^{-}_{first} \approx 6ns$ and $T_{min} \approx 6ns$ with an assumed interconnect delay of $1ns$ lead to an expected $f_{average} \approx 71MHz$.

Short term jitter: Short term fluctuations of the frequency and discontinuities in the clock periods are expected from at least two sources. At first, the fact that ↑-transitions and ↓-clock ticks are partially processed via distinct logic blocks results in different propagation delays. Moreover, the above mentioned supply voltage dependence might also introduce additional jitter. Temperature changes are considered to be too slow to yield perceivable short-time effects.

Long term jitter: In the tick generation system's long term operation especially the effect of varying temperature is expected to be noticeable. Self-heating of the TG-Alg chips is anticipated to continuously slow down the tick generation process, although cross-correlation of the measured frequencies with corresponding temperature should allow to mask such effects. This way it might be possible to identify systematic jitter components.

6.2.2 Start-up behavior

As already mentioned in the treatment of the worst case, presented in Section 6.1, the start-up of the tick generation process denotes a very critical point. As long as all nodes boot close enough to each other ($\leq \tau^{-}_{rem}$) theory predicts no consequences to worst-case characteristics. However, in certain configurations it can be expected that varied booting delays result in different clock periods (at least for a few clock ticks). Effects like this might be visible in scenarios similar to the worst-case precision assessment presented in Section 6.1.1. Taking a closer look at the execution trace reveals that the fast nodes continuously operate at the same clock period determined by τ^{-}_{rem}. In contrast to that the group of slow nodes starts with a period depending on τ^{+}_{rem}, generating the first ticks by virtue of the "Increment Rule". After precision π is reached subsequent ticks are triggered via the "Relay Rule" at rate dictated by the fast nodes. Thus the clock periods of some TG-Alg nodes might stabilize only after a settling time which depends on the relation of delay paths and a node's respective start-up time.

6.2.3 Effects of faults

Another obviously critical point in operation of the tick generation is given by the moment in which a fault occurs. Fault effects are likely to lead to perceivable fluctuations in the TG-Algs' frequency. Such fault-induced changes in clock rate might be short term effects only, i.e., increased short term jitter. However, a fault might also be able to influence the mean frequency of a TG-Alg or the whole ensemble, although the effect is bounded by

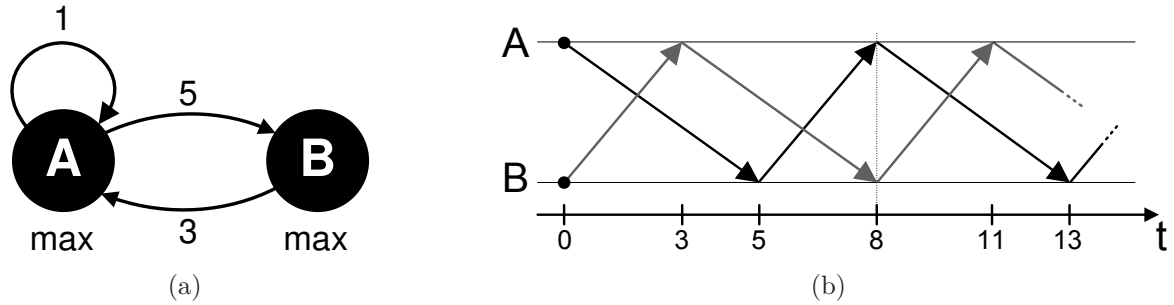
the worst-case measures for slowest and fastest progress as well as accuracy. To illustrate the effect of failing links, consider the assessment scenario for the lower accuracy bound presented in Figure 6.2. In this setup set F of faulty nodes initially provides group A of fast nodes with tick messages at a high rate determined by τ^-_{rem}. Later on these connections are cut ($\tau_{rem} = +\infty$) which yields a large delay before the next tick is generated among the previously fast nodes. Moreover, the mean frequency of the TG-Algs of set A does no longer rely on τ^-_{rem} but on τ^+_{rem}. This can lead to substantial frequency changes depending on the ratio of τ^-_{rem} and τ^+_{rem}.

6.3 Supportive simulation model

In general the proposed tick generation approach's behavior has been assessed in two ways. At first, formal proofs cover worst-case conditions, while secondly, measurements aim at the characterization of typical operation modes. However, for a sound evaluation of the tick generation system this twofold assessment approach still leaves some questions unanswered. The formal analysis cannot provide predictions for average performance measures like the mean frequency. In addition, measurements cannot be performed without environmental influences such as disturbances in the power supply. As a remedy the simulation scheme described below combines reasonable observability with enhanced controllability. This way the simulation framework allows to model and assess the tick generation system with sufficient accuracy to reflect all algorithmic properties. Additionally, in contrast to real measurements, the simulation still provides full control over the tick generation system's environment. To generally introduce the simulation approach consider the following set of example equations:

$$A(k) = max(A(k-1)+1, B(k-1)+3)$$
$$B(k) = max(A(k-1)+5)$$

These kinds of non-linear difference equations which are heavily used in non-linear control theory [44] can quite easily be mapped to message-driven distributed systems. In the



(a) (b)

Figure 6.8: Two-node wait-for-all system (a) graph representation (b) execution trace

particular case of the above presented equations A and B correspond to the nodes of a "wait-for-all" system (the max function ensures that a new message $k+1$ is only triggered if message k has been received on *all* incoming links). As depicted in Figure 6.8(a), the weights of the graph's edges represent the communication delay of the respective link, e.g., every message from node B takes three time units until it arrives at node A. In order to obtain the mean rate for the generation of messages in the presented example, the largest (max) value for the mean cycle weight ξ of all elementary cycles has to be derived. For the graph given above elementary cycles are $A-A$ and $A-B-A$. The mean cycle weight is computed as the sum of the respective cycle's weights divided by the cycle length, e.g., $\xi_{A\text{-}A} = \frac{1}{1} = 1$ and $\xi_{A\text{-}B\text{-}A} = \frac{3+5}{2} = 4$. In the considered system comprising two nodes and three edges only, the resulting mean rate can quite easily be determined. Figure 6.8(b) presents an execution trace corresponding to the system graph and reveals that at time 8 the initial phase between A and B (from time 0) is reached again. Hence, the length of the cycle leading to this point in time is obviously two, yielding a mean message generation rate of $\frac{8}{2} = 4$. Interestingly this equals the above presented cycle $A-B-A$ of the graph. This match is not by chance, but follows from a general theorem in <max,+> algebra. However, to be suitable for the DARTS tick generation scheme, the modeling of the above presented "wait-for-all" system — comprising max and + functions only — has to be enhanced further. This enhancement is inevitable due to the fact that the fault tolerance properties of the DARTS tick generation algorithm relies on threshold functions, i.e., $f+1$-out-of-$3f+1$ and $2f+1$-out-of-$3f+1$ instead of "wait-for-all". The sum of product implementation of the threshold circuits (cf. Section 4.4.1) can quite naturally be translated into a framework of <min,max,+> functions, with min corresponding to the products and max representing the final sum term. In the <min,max,+> representation of the DARTS tick generation approach both the min and the max terms affect mean message generation rate. Thus the previously presented scheme for deriving this rate also needs to be extended. In literature the *Duality Conjecture* [42] has been proposed for solving the problem of computing the mean cycle weight of <min,max,+> systems. The underlying concept is illustrated via the example described below and depicted in Figure 6.9.

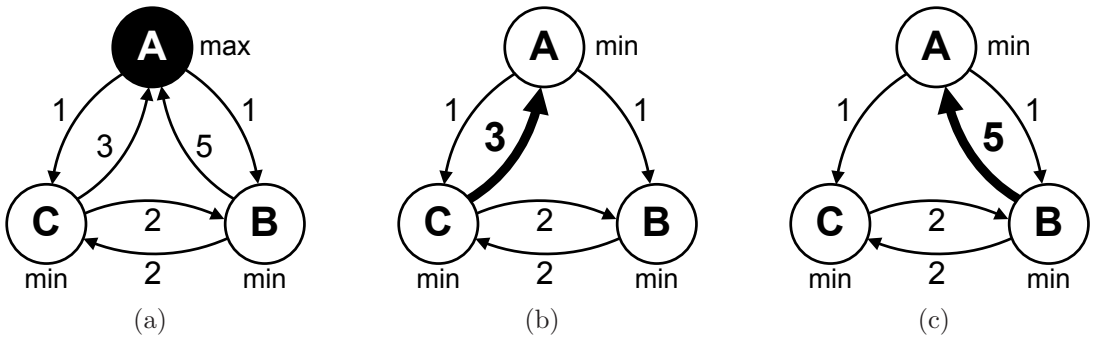


Figure 6.9: <min,max,+> system (a) whole graph (b) first projection (c) second projection

1. All possible projections P_j for the given graph are generated by choosing exactly one incoming edge for each max-node while all other incoming links are dismissed. After this the 1-input max nodes can be replaced by min nodes.

2. For each of these projections the minimum mean cycle weight ξ_{P_j} is determined.

3. The original <min,max,+> graph's mean cycle weight is derived by computing the maximum over the mean cycle weights of all projections.

Unfortunately, this simple algorithm suffers from combinatorial explosion as the number of max-nodes increase. There exist other algorithms for computing ξ, for instance presented in [13], however, their time complexity is still unknown.

In the context of the evaluation of the DARTS tick generation scheme the combinatorial explosion of the before presented algorithm can be handled, at least if the evaluations are limited to a system comprising 5 nodes only. The main interest in these kinds of simulations is given by the fact that it allows to conveniently analyze the tick generation process with configurable properties at a resolution of 1 tick period. Additionally, considering different delay configurations estimates for a system's mean frequency can be easily derived by following the above described <min,max,+> computations.

Chapter notes

In this chapter the most important qualitative and quantitative properties of the DARTS tick generation approach have been pointed out in order to derive appropriate evaluation scenarios for the hardware design presented in Chapter 4. Especially the assessment scenarios for the characterization of the DARTS tick generation approach's worst-case behavior have been presented in detail. Another important area of interest is given by the evaluation of typically encountered / average case characteristics of the DARTS clocks (e.g., mean frequency and jitter). The simulation-based assessment of properties neither covered by formal analysis nor easily accessible by measurements presents a complementary approach to increase the coverage of the evaluations. Additional details to the <min,max,+> simulations of the DARTS clocking scheme have been recently published in [38].

Chapter 7

Evaluation and Measurement Results

There is no such thing as a failed experiment,
only experiments with unexpected outcomes.

Richard Buckminster Fuller

THE INITIAL quote in fact reflects a small, but quite interesting part of the conducted evaluations. Most of the tick generation chips' behavior has been a priori anticipated. However, a few peculiarities of the chip design in conjunction with the measurement setup manifested themselves at first glance as "failed experiments". Most of these unexpected evaluation outcomes later on provided valuable information on the tick generation scheme's characteristics. In general, the measurements of the HITS chips, assembled into a cluster of interacting TG-Alg units, can be seen as the final design validation step. After formal proofs on the algorithm level, followed by simulations on different hardware description levels, the measurements conclude the chain of validation and characterization efforts. Given this integrated assessment approach, the better part of the measurements aimed at the confirmation of properties predicted from theory and simulations. Nevertheless, there have been several characteristics which have not been covered in the aforementioned assessment methodologies. Especially long-term behavior under real-world operating conditions cannot easily be simulated, nor exactly predicted by theory. Therefore, the evaluations provide a holistic approach for assessing worst-case properties — under realistic, thus varying, operation conditions — together with the broad field of typical executions. Last but not least, the insights gained from the detailed assessment may be used as a starting-point for optimizations and enhancements of the tick generation approach.

7.1 Assessing and validating the standard node HITS design

Before going into details with a cluster of tick generation nodes, the initial evaluations are aimed to provide a general view on the HITS ASIC's properties and were therefore made on a single chip. The subsequently presented measurements are, unless stated otherwise, performed at nominal core voltage of 1.8V, room temperature, i.e., 26 to 30°C and the threshold circuits configured for $f = 2$, with all nodes working properly.

7.1.1 Delay validation

From the design verification of the ASIC it is known that all local timing constraints are fulfilled and starting with the assessment of a single chip the global constraints do not pose a threat. Due to the fact that the operation speed of a DARTS tick generation cluster obviously depends on the processing speed of the participating nodes, the assessment of propagation delays and the validation with the numbers obtained from design files is among the most important characterization steps.

In general, there are two distinct paths to initiate the generation of a new tick at the highest possible speed. Either the local pipeline already holds one or more ticks, i.e., is waiting for the corresponding remote tick, or the opposite is true, i.e., the remote pipeline is filled and the next local tick will trigger the generation of a new tick. These paths have already been introduced in Section 3.4.3 with the first one in essence corresponding to T^{-}_{first} and the second one to T_{min}.

Waiting for remote tick: Post-place and route design files for typical operating conditions predict T^{-}_{first} to be approximately $6ns$. Measurements of a similar path showed a delay of $7ns$. This path involves `remote_clk[.]` input pin, next 6 Muller C-Elements[1], the PCSG unit, the Threshold Modules including tick generation and finally `local_clk` output pin.

Waiting for local tick: For the T_{min} path again a delay of $6ns$ has been extracted from the design files. Analogously to the above examination the measured delay is $7ns$. The path is also quite similar comprising the `local_clk_self` input pin followed by 5 Muller C-Elements, the PCSG unit, the Threshold Modules and Tick generation block, ending at the `local_clk` output pin.

The difference of $1ns$ between measurements and design files is mainly due to the fact that the measurement setup does not, unlike the evaluation of the design files, use the shortest path through the threshold circuits.

[1] At first the tick can only propagate through 3 remote Muller C-Elements. Only after the Difference Module has acknowledged the already waiting local tick with the newly arrived one, the remote tick can move on to the final pipeline stage, thus enabling the generation of the next tick.

Table 7.1: Cluster of 8 standard nodes: voltage scaling

core voltage in [V]	avg. frequency in [MHz]	current ASIC U6 in [mA]	current all in [mA]
1.3	38	11.7	100
1.4	43	15.1	126
1.5	47	17.6	150
1.6	50	20.6	178
1.7	52	23.8	204
1.8	54	27.2	233

Based on these path delays the maximum rate of clock signals generated by a HITS standard node can be bounded with $1/2T^{-}_{first}$ and $1/2T_{min}$, respectively, yielding a frequency of $\approx 71MHz$. From theory, however, it is clear that this rate can only be maintained for a short period, i.e., either the remote or the local pipeline has to be full while ticks at the opposite side arrive at T^{-}_{first} or T_{min}, respectively. As soon as the previously filled pipeline gets emptied the clock rate will notably slow down.

7.1.2 Operating condition dependence

After the initial delay considerations using a single chip, a tick generation system composed of 8 fully interconnected standard nodes was assembled (while the experimental nodes were held in a reset state). The customized evaluation board, depicted in Figure 7.1, provides extensive measurement support in order to allow a detailed assessment. It comprises 8 HITS ASICs ($U1$ to $U8$) including individual configuration circuitry, pattern generator and logic analyzer interfaces for test and evaluation purposes as well as additional status and control ports. To have as little influence as possible on the standard nodes' tick generation process in this setup, the central FPGA is not used for signal routing. Thus, all links of the tick generation network (TG-Net) are implemented as point-to-point connections on the printed circuit board (PCB). The first characterization steps for this cluster are targeted to provide insights on the operation condition dependence. Table 7.1 shows the average frequency and the corresponding current drawn by the design. As expected from Equation 6.1 the achieved clock frequency scales proportional to the supplied core voltage. Figure 7.2(a) presents results of detailed measurements in which the applied core voltage has been changed in $10mV$ steps in an interval starting with $1.30V$ and ending at the nominal voltage of $1.80V$. An improved illustration of the measurement data which makes the correlation of voltage and clock frequency more evident is given in Figure 7.2(b). Core voltage and frequency are given in percentages of their respective maximum value. This way it can be observed that a voltage change of 1% yields approximately 1% variation in clock frequency (red line in Figure 7.2(b)). The strong impact of the core voltage on the

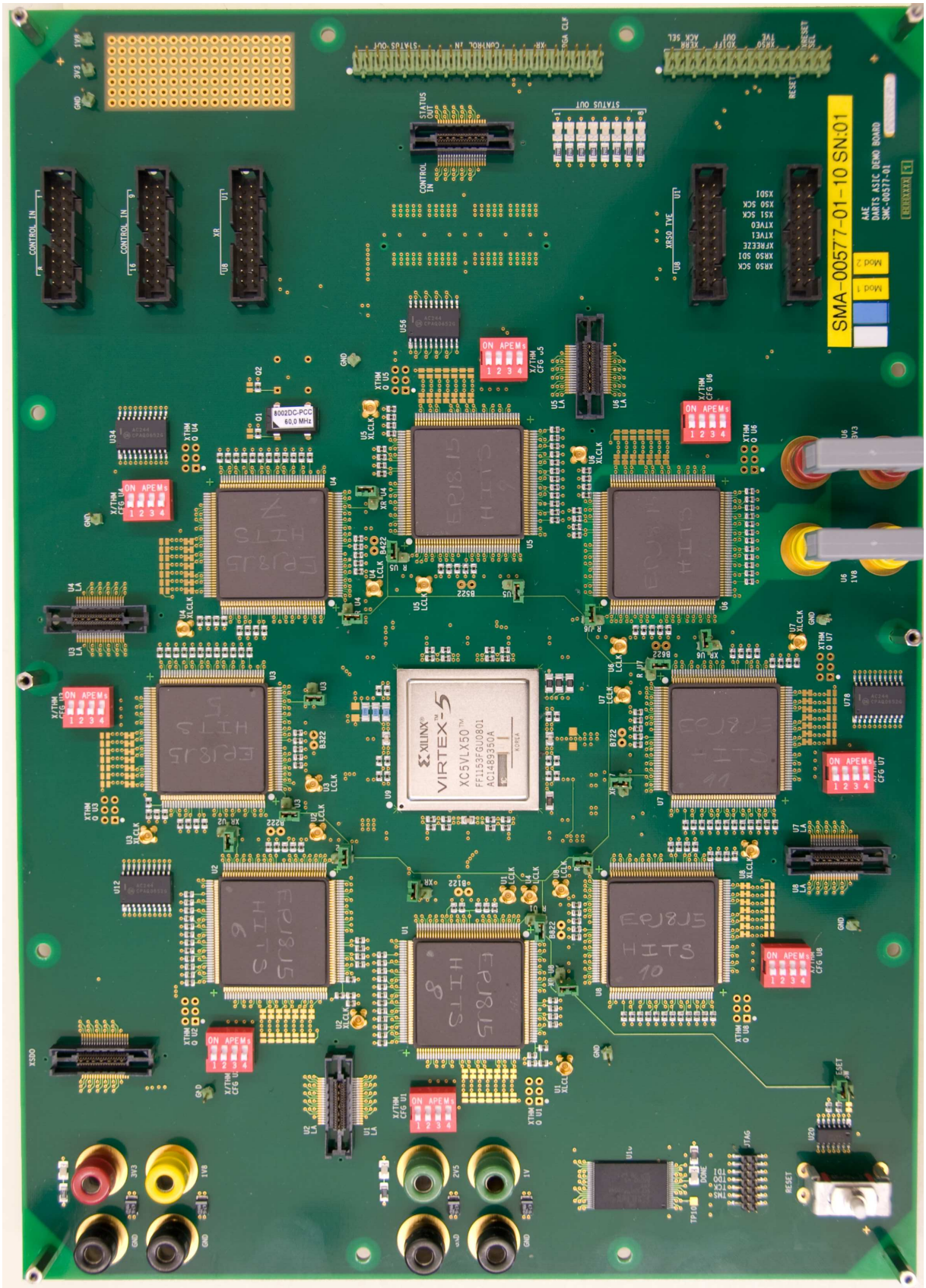
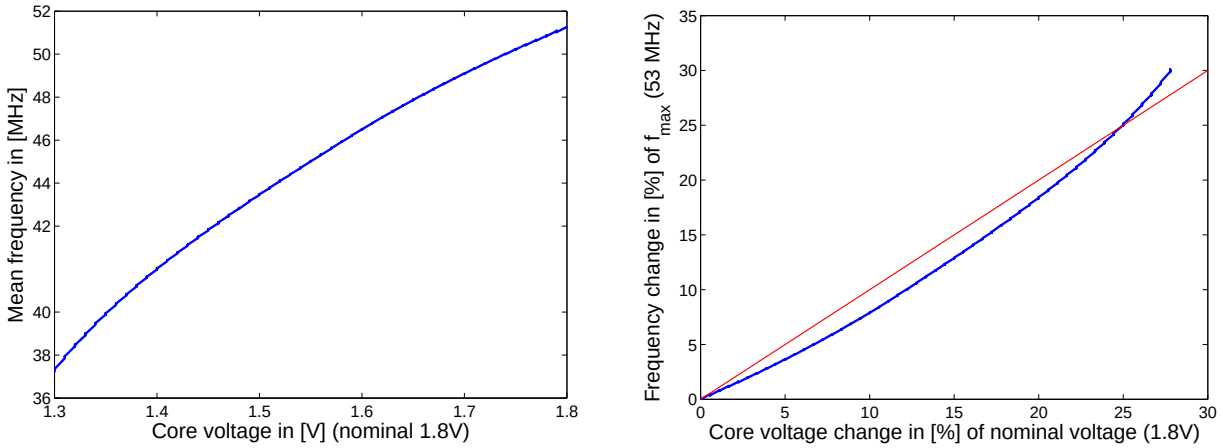


Figure 7.1: DARTS prototype board, comprising 8 interconnected HITS chips



(a) (b)

Figure 7.2: DARTS cluster's mean clock frequency core voltage dependence

operating frequency of the asynchronous tick generation implementation was expected. A second important factor for the design's speed is given by temperature. Again, according to Equation 6.1 the switching speed and propagation delay of CMOS circuits scales indirectly proportional to temperature. This anticipated dependency has also been confirmed by the measurements conducted.

7.1.3 Jitter and stability

The mean frequency and its dependence on core voltage and temperature have already been discussed before. Taking into account that DARTS clocks are designed to provide synchronous circuits with a suitable clock signal, not only the mean frequency but also short term jitter and long term stability are of special interest. Even if the average clock rate of a DARTS clock signal might conform to the specifications of the associated synchronous design there is still potential for problems. A heavily unbalanced duty cycle, for instance, might not be tolerable for synchronous memory interfaces.

The first jitter and stability evaluations are based on short-time measurements with very high resolution (up to 10GS/s) including approximately $40,000$ clock transitions. These measurements aim at characterizing a single node's clock signal of a running DARTS cluster. The obtained results are summarized in Figure 7.3 and described in the following. The clock's measured half periods are presented in the histogram plots shown in Figure 7.3(a), (b) and (c). In the first histogram two cluster points can be identified one at $\approx 8.7ns$ and the other at $\approx 9.8ns$. A separate examination of the *HI* and *LO* clock periods (see Figure 7.3(b) and (c), respectively) reveals the source for the accumulation points in Figure 7.3(a). In general the distributions of the *HI* and *LO* periods correspond with the expected behavior. The difference of the *HI* and *LO* periods' mean values of almost $1ns$

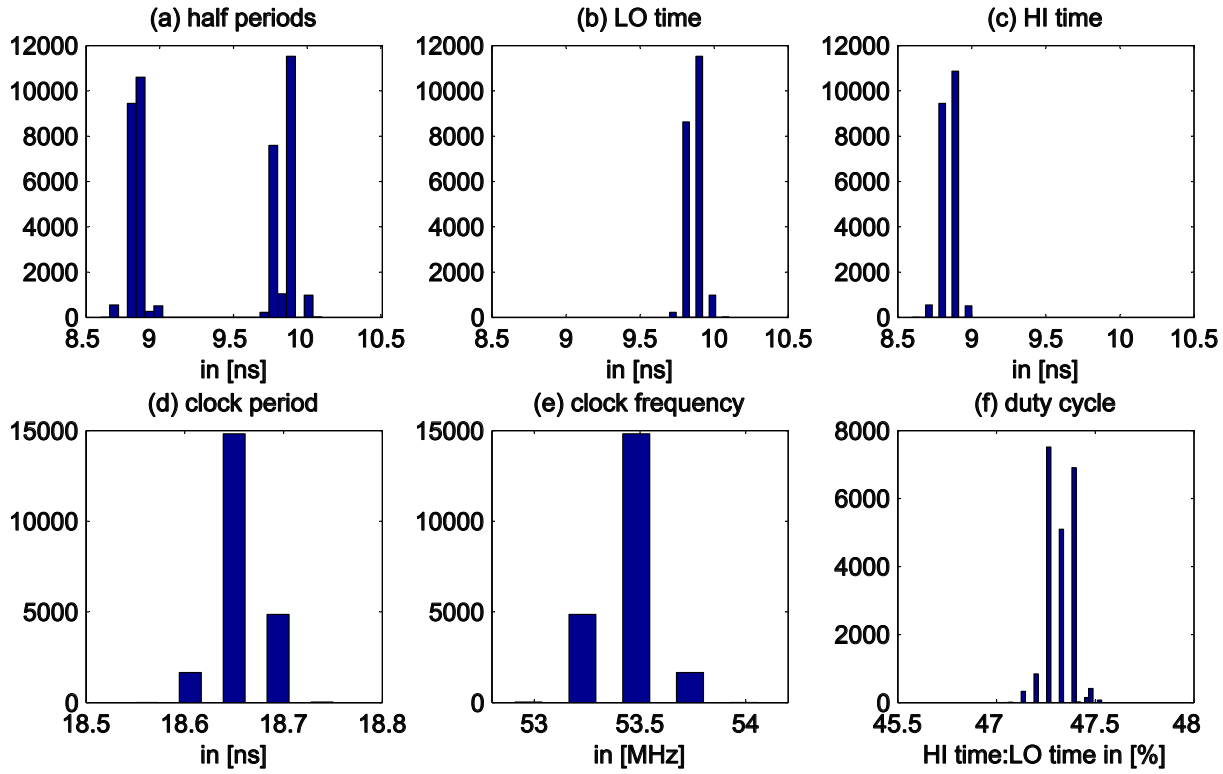
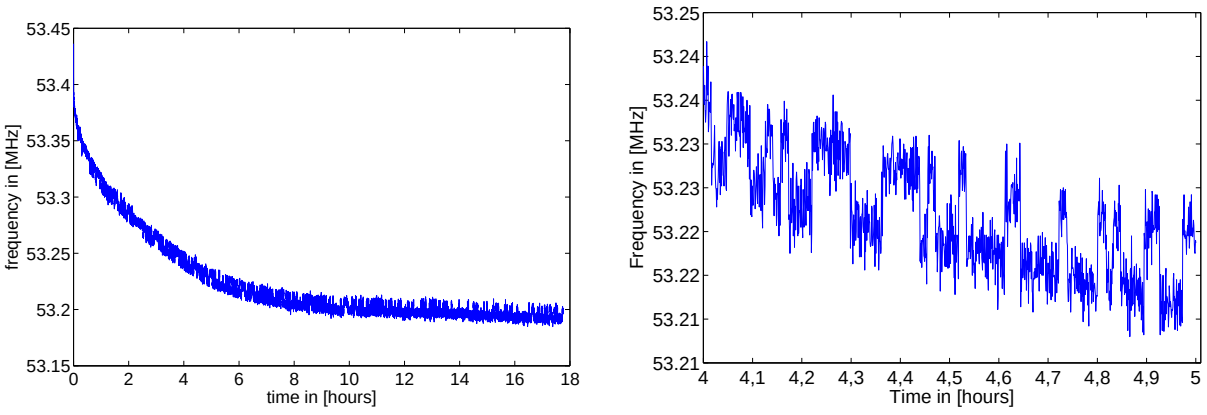


Figure 7.3: Statistical single clock evaluation of a running standard node cluster

can be traced back to slightly different processing speed of the respective clock signals, in particular, the employed Muller C-Elements have different propagation delays for rising and falling transitions. Figure 7.3(d) and (e) present the distribution of clock period and frequency with a mean frequency of $53.4MHz$ and a standard deviation of $0.153MHz$. Again a distinct accumulation point can be identified. As motivated above, the stability of a clock signal's duty cycle might be vital for synchronous circuits. The histogram depicted in Figure 7.3(f) shows that the duty cycle jitters less than 1% point with a mean value of $\approx 47.3\%$ and a standard deviation of 0.28%. The measurements conducted in Section 7.1.2 show that the stability of the clock frequency heavily depends on the stability of the operating conditions. Figure 7.4(a) presents a long term assessment of a node's mean clock frequency with an evaluation interval of more than 17 hours. It can be observed that clock frequency noticeably decreases over time by about $250kHz$. The operating conditions, i.e., core voltage and ambient temperature were not varied in this experiment setup. The measurements start with all nodes in reset state with no activity. Thus no mentionable current is drawn by the HITS chips. As soon as the reset gets deactivated the designs start to draw substantial current which depends on the applied core voltage (cf. Table 7.1). This current contributes to self heating of the running HITS chips and causes the asymptotic decrease of the mean clock frequency depicted in Figure 7.4(a). Figure 7.4(b) shows the same



(a) (b)

Figure 7.4: Long term clock stability (a) 17 hours run, (b) hour four at higher resolution

measurement data where only a time interval of one hour is analyzed more closely. The source for the observed discrete frequency jumps of approximately $10 - 20kHz$ occurring every few minutes could not be pinpointed at first. Even a thorough reassessment of the algorithm and the hardware design was unable to give any explanation for the frequency jumps. However, a closer look at the experiment environment was able to resolve this mystery. A 15-minute snapshot of another frequency measurement including a high resolution trace of the core voltage is presented in Figure 7.5. In this figure it can be observed that a discrete jump of the core voltage is directly followed by a frequency jump. This behavior perfectly fits into the HITS design's voltage dependence presented earlier. In the depicted measurement the voltage changed by $\approx 0.5mV_{rms}$ which led to the aforementioned shift of average frequency by $10 - 20kHz$. The initial cause for the minor voltage change noticeable in the frequency assessment is hidden in the digital power supply (Agilent E3648A) which has quantization steps of $0.5mV_{rms}$. The correctness of this interpretation for the frequency jumps has additionally been confirmed by crosscheck measurements with analog power supplies. In these evaluations overall increased frequency jitter was observed due to the higher level of voltage noise. However, neither discrete steps in the voltage level nor in the mean frequency were encountered.

In addition to the above presented assessment of a single clock signal, the clock signals of the whole ensemble are evaluated in the subsequent paragraphs[2]. The main interest clearly resides in the synchronization of the clock ensemble. Detailed short-term measurement showed that for the fault-free case the ensemble starts with tight synchrony and remains closely synchronized (the small initial offset is due to differences in the propagation of the reset signal). Under normal conditions, i.e., nominal core voltage and room temperature, evaluations yielded initial offsets in the range of $1ns$ to $1.5ns$. In these short-term

[2]Note that evaluations involving the whole cluster of 8 nodes are performed via logic analyzer measurements having a time resolution of $250ps$.

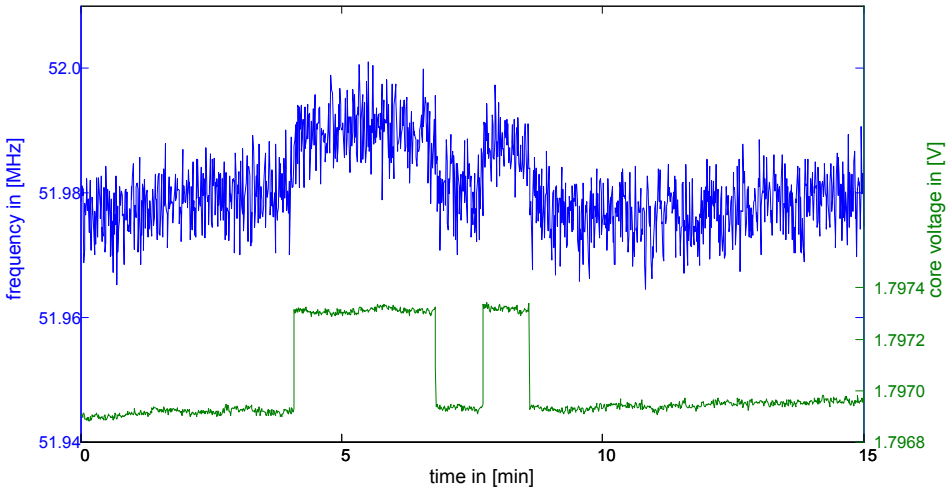


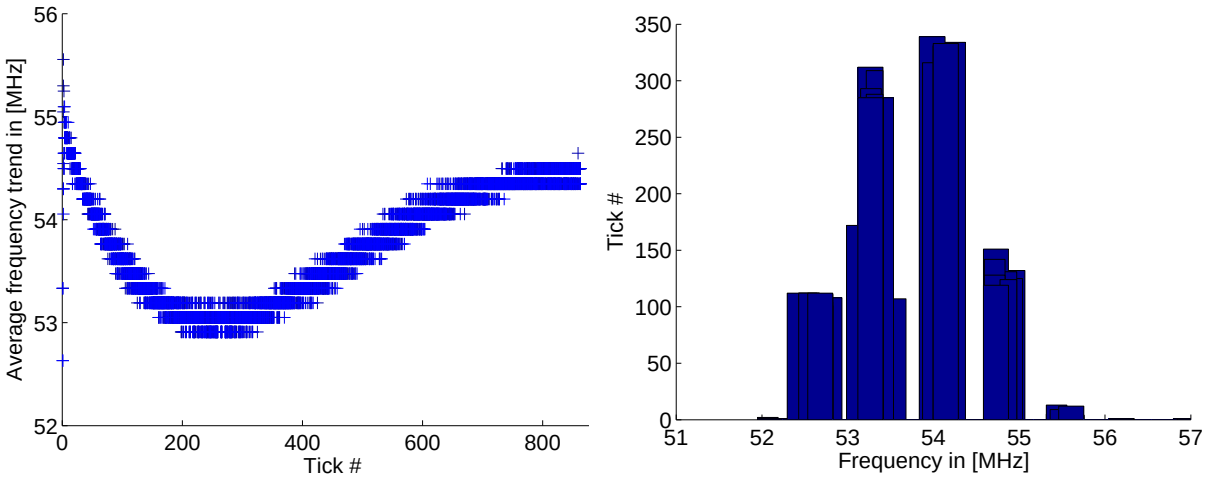
Figure 7.5: Frequency and voltage trace showing power supply variations

measurements the maximum skew among any two TICK(k) clock transitions never exceeded its initial offset of $1.5ns$. Hence, a fault-free clock ensemble running under nominal operating conditions has precision $\pi = 1$. In Figure 7.6(a) all 8 nodes' frequencies of a DARTS cluster (starting from reset state) are depicted[3]. It can be observed that the frequencies of all DARTS clocks change jointly, thus yielding close synchronization. Figure 7.6(b) presents the DARTS clocks' frequency distribution of this short term measurement.

7.1.4 Fault tolerance properties

Up to now all evaluations have assumed TG-Alg nodes operating according to their specification. In contrast, the evaluations presented in this paragraph consider scenarios with faults artificially introduced into a running cluster of 8 standard nodes (which by design should be resilient to $f = 2$ Byzantine faults). In the conducted experiments the consequences of crashing TG-Alg nodes are examined in particular. The node crash scenarios are implemented by resetting one or two nodes of the DARTS cluster. Note that these scenarios do not necessarily have the benign properties of crashes like they are assumed in distributed systems. To substantiate this, recall the discussions from Section 4.6 where it has been pointed out that even stuck-at faults can be outside the scope of the crash fault scenario. An early clock transition, i.e., by changing a clock rail from HI to LOW (stuck-at-0) generated by the activation of a node's reset is already within the class of malicious/Byzantine failures. All combinations of scenarios with one or two nodes crashing yielded Figure 7.7. For each of the reset scenarios the mean frequency before and after

[3]To enhance the expressiveness of the graph the data values actually have been smoothed to compensate for the limited resolution of the logic analyzer. Note that this did NOT affect the general trend but only the magnitude of the frequency changes



(a) (b)

Figure 7.6: Mean frequency (a) trend and (b) histogram of all 8 nodes

the crash has been derived from measurement data. The lines interconnecting these two mean frequency values illustrate the actual drop of the clock frequency. As anticipated, in all 36 reset scenarios the deactivation of nodes leads to a decrease of the mean clock frequency. This slowing down is quite natural since for the non-faulty nodes of the cluster the crashing of nodes implies that one or two of the previously $2f+1$ first node(s) has/have been deactivated. Hence the correct nodes have to wait until tick messages are received from slower nodes which are still up and running, consequently leading to additional delay before the next tick can be generated[4].

7.2 Assessing and validating the experimental node HITS design

As already known from Chapter 5, the experimental node implementation comprises several enhancements for measurement support. This additional circuitry, however, increases the propagation delay of the design. The changes with the most notable impact on the propagation delay are given by the enlarged local and remote pipelines (eight Muller C-Elements each, in contrast to four C-Elements at the standard node), the pipeline extension/overflow detection block and the extensive use of scan cells throughout the design.

[4]Due to small differences in propagation delays and the close synchronization of all clocks, each node's set of $2f+1$ fastest neighbors might be different. This leads to the fact that the reset of each node at least slightly influences the clock overall clock frequency.

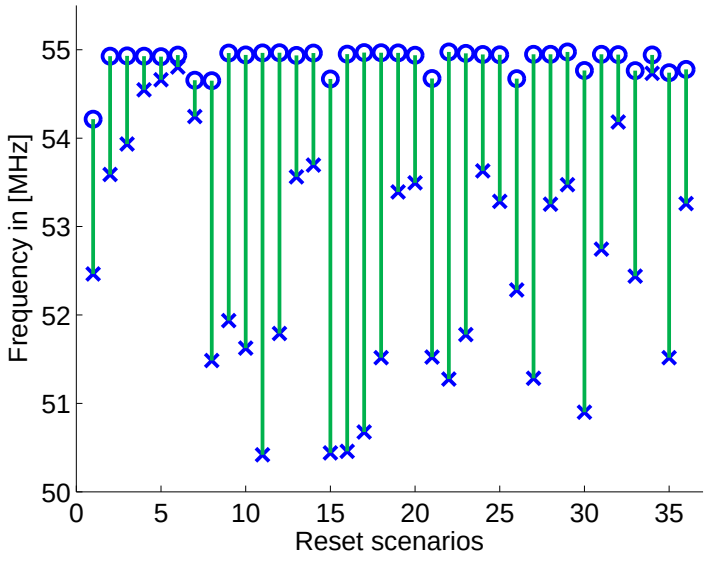


Figure 7.7: Mean Frequency (o) pre and (x) post reset of 1 or 2 nodes

7.2.1 Delay validation

Analogously to Section 7.1.1, the first validation step for the experimental node design is to assess the design's maximum operation speed. Again measurements for delay paths corresponding to T^{-}_{first} and T_{min} have been conducted. These evaluations yielded similar, but certainly increased delay results if compared to the assessment of the standard nodes delays. In detail, the measured delay for the remote path accounts for $9ns$, which is also the case for the local path. The delay values from the design files predicted delays of $7ns$. This mismatch is once again due to the fact that the measured paths did not represent the fastest ones. In contrast to the standard node design, the enhanced accessibility of the expnode design additionally allowed to directly evaluate the disabling path $T_{min,dis}$, which is part of the Interlocking Constraint. The propagation delay of the disabling path is $1ns$ smaller than the enabling path T_{min}, yielding a value of $8ns$. The reason for the decreased delay is given by the involved circuit path which only comprises 6 Muller C-Elements in contrast to the 9 C-Element delays of the enabling path (cf. Figure 3.7). Extracted from the design files, $T_{min,dis}$ adds up to 6ns which is also consistent with the analysis of T_{min}.

7.2.2 Elastic pipeline assessment

It is obvious that the elastic pipelines employed for the local and remote tick queueing are important components of the tick generation path. Therefore a closer look at the performance of these components seems to be indicated. The increased pipeline depth of the experimental node not only allows larger deviation of the nodes' synchrony, but also introduces additional propagation delays for arriving ticks. To assess the maximum

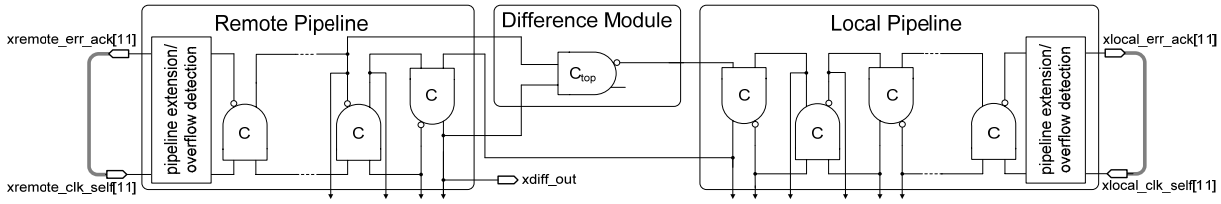


Figure 7.8: Ring oscillator implementation via pipeline pair and Difference-Module

performance of the pipelines — more precisely, the maximum rate for tick queueing and removal — a set of experiments was performed. The experiment setup takes advantage of two special features of the experimental node. First of all, local and remote pipeline extension ports are used to automatically generate new ticks, while secondly, the reset scan chains allow to initialize the elastic pipelines with a certain amount of ticks. For both subsequently described evaluations a single experimental node was used with the remote pipeline's clock input shorted to the respective `xremote_err_ack` signal. Likewise, the corresponding `local_clk_self` signal was also shorted to its respective `xlocal_err_ack` port. Furthermore, the reset-selector unit was configured in a way that `xdiff_out` toggles whenever a tick has been removed from both of the selected pipelines. The setup is depicted in Figure 7.8.

Empty pipelines: In the first experiment both the remote and the local pipeline are empty, i.e., all Muller C-Elements (except the one in the Difference-Module) are initialized to 0. In the logic analyzer trace shown in Figure 7.9 it can be observed that ticks are removed continuously about every $4.5ns$. This delay essentially represents the propagation through 10 Muller C-Elements (8 remote, 1 in the Difference-Module and 1 local) added up with the delays of the clock input driver and the `xdiff_out` buffer. Putting it all together we get a ring oscillator design being capable of generating a sort of clock signal running at a rate of approximately $108MHz$.

Filled pipelines: The second evaluation using the pipelines in a self-feedback configuration takes advantage of the reset/set scan chains to pre-load ticks into the pipelines. The Muller C-Elements of the selected remote/local pipeline pair are initialized with an alternating sequence of TICK($\uparrow$)/TICK($\downarrow$), i.e., both pipelines are completely filled. The resulting logic analyzer trace of this setup is depicted in Figure 7.10. In contrast to Figure 7.9, the removal of ticks starts as soon as reset (`xRSOTVE` in the case where the reset scan chains are used) is released. The first tick is already removed after $1ns$ while the next transitions are processed at approximately $1.75ns$ per tick. Recalling the operation of the Difference-Module from Section 4.3.1, the value of $1.75ns$ accounts for the propagation through the Difference-Module added up with the delays of the 3 Muller C-Elements which directly interact with the Diff-Module's C-Element. Due to the fact that tick generation involves rather slow input and output buffers, the pipelines will get emptied quickly. In this particular case the pipelines are empty

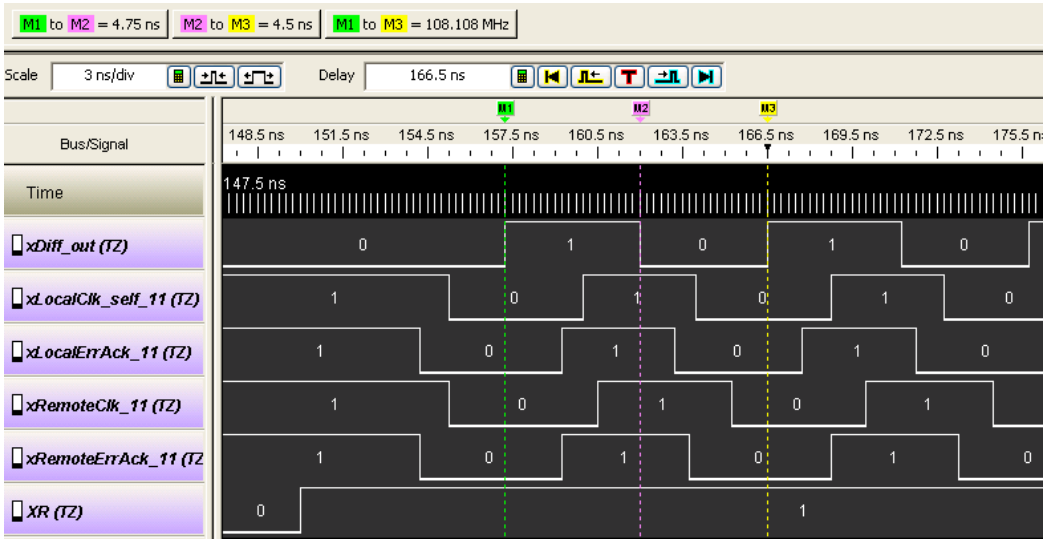


Figure 7.9: Trace of a ring oscillator with uninitialized pipelines

after processing 10 ticks. This circumstance substantially slows down operation to a rate similar to the one presented in the above experiment with initially empty pipelines.

7.2.3 Operating condition dependence

For all system assessments of the experimental node design the demonstrator board presented in Figure 7.1 was used again. However, in contrast to the measurement setup for the standard node, the central Virtex-5 FPGA played a major role for the HITS expnode evaluations. All TG-Net links were routed through the FPGA to provide enhanced observability and control of these clock signals. Additionally, the FPGA enabled the implementation of dedicated high-speed evaluation circuity to more precisely analyze the distributed tick generation system's current operations.

Similar to the analysis of the standard node design, the mean frequency of the experimental node also scales the way that a 1% voltage change leads to a 1% change in clock frequency. The results of these measurements are summarized in Table 7.2.

7.2.4 Precision

The worst-case precision π represents one of the most important synchronization properties of the DARTS tick generation approach. One possible scenario to force a tick generation cluster into this worst case has been introduced in Section 6.1.1. Due to the fact that computation and interconnect delays of the DARTS cluster are almost perfectly matched —

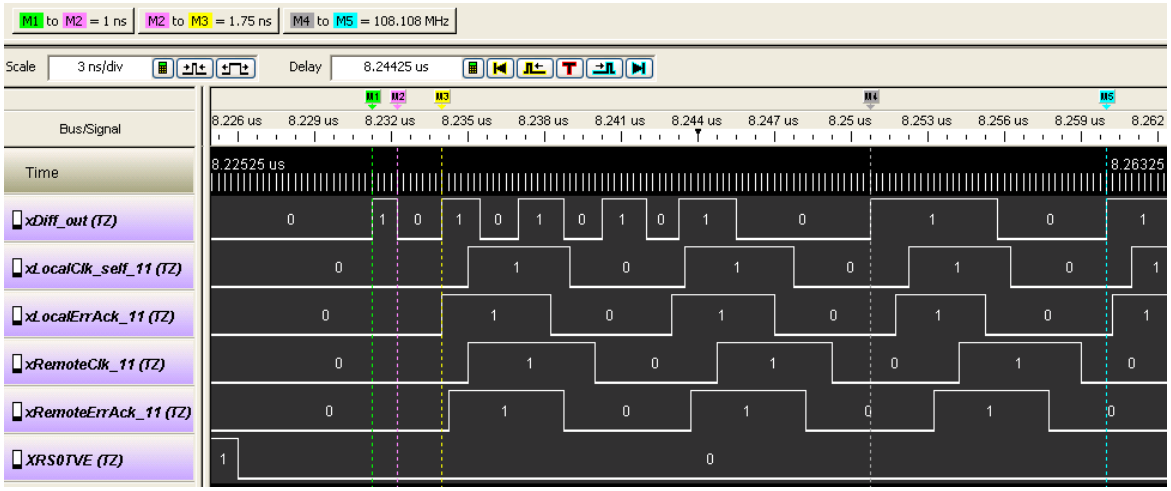


Figure 7.10: Trace of a ring oscillator with initially full pipelines

Table 7.2: Cluster of 8 experimental nodes: voltage scaling

core voltage in [V]	avg. frequency in [MHz]	current ASIC U6 in [mA]	current all in [mA]
1.3	25	8.8	75
1.4	27	10.9	92
1.5	29	12.7	108
1.6	30	14.5	123
1.7	31	16.4	139
1.8	32	18.3	156

which yields a precision $\pi = 1$ — artificially increased τ^+_{rem} delays had to be introduced for selected paths of this scenario. The central FPGA which is already in use for interconnecting the TG-Alg nodes was utilized for the implementation of controllable delay lines for τ^+_{rem}. This way multiple measurements with different parameters for τ^+_{rem} could be performed. Note that, unless stated otherwise, for all expnode measurements $\tau^-_{loc} = \tau^+_{loc} = \tau^-_{rem}$ are approximately equal to $6ns$ (this delay is determined by the input/output buffer and routing delay of the FPGA). As introduced in Equation 3.3, the worst-case precision is mainly based on the proportion of T_{QS} and T^-_{first}. Recall that T_{QS} identifies the time when the last TICK(*k-1*) is generated, while T^-_{first} represents the instant of generation of the first TICK(k). Thus, precision π can be easily validated by measuring the delay T_{QS} and T^-_{first}. Figure 7.11 presents the resulting evaluation results for $T^-_{first} \approx 15ns$ and T_{QS} being scaled step-wise via τ^+_{rem} from $17ns$ to $113ns$. The assessments confirm the predictions based on theory. Additionally it could be observed that due to the fixed delays for $\tau^{+/-}_{rem}$ and $\tau^{+/-}_{loc}$

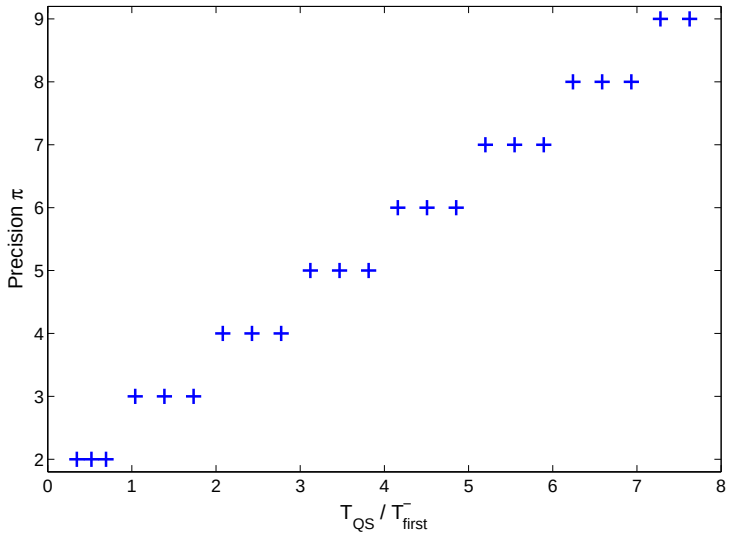


Figure 7.11: Precision vs. fastest to slowest path controlled via the maximum remote delay

the worst case for π manifested itself after only a few clock cycles. As a side note it has to be stated that delays leading to a precision of 9 are outside of the experimental node's specification[5]. The expnode design with elastic pipeline depths of 8 is designed for $\pi \leq 8$. More details on the pipeline depths are given in Section 7.2.6.

7.2.5 Accuracy

The synchronization property accuracy bounds the minimum and maximum tick generation rate for each correct node. Analogously to precision, formally derived bounds for accuracy have already been introduced in Section 3.4.4, Equation 3.4, while the corresponding assessment scenarios for the lower and upper worst-case bounds have been presented in Section 6.1.2. In essence, the implementation of the accuracy assessment followed the same approach as the one employed for the precision experiment. Thus, delays were added for the τ^{+}_{rem} paths to attain improved observability of the worst case. The main difference to the precision measurements is given by the fact that worst-case accuracy can only be achieved by dynamic scenarios, i.e., at a certain point of operation (when maximum precision has already been reached) delays have to be changed according to the scenario's specifications. From Equation 3.4 it is known that T_P and T_{QS} are the relevant terms for deriving the accuracy's lower bound, with T_P denoting the largest time interval between the sending of TICK(k) and TICK(k+1) among all correct nodes. In essence, the tick generation's lower

[5] At $\pi = 9$ slow nodes lose a clock tick. Apart from that, tick generation proceeds at a pace forced by the fast nodes, however, the slow nodes have to be considered faulty after a tick was lost.

bound δ is predicted to be $2T_P + T_{QS}$. A closer look at the formal analysis is given below.

$$\left\lfloor \frac{\delta - (T_{QS} + T_P)}{T_P} \right\rfloor = number\ of\ ticks\ generated\ in\ \delta$$

The scenario for achieving this lower bound aims at maximizing the time interval δ in which no tick is generated by a correct node with

$$\left\lfloor \frac{\delta - (T_{QS} + T_P)}{T_P} \right\rfloor = 0$$

by substituting

$$\delta = 2T_P + T_{QS} - \varepsilon.$$

yields

$$\left\lfloor \frac{2T_P + T_{QS} - \varepsilon - T_{QS} - T_P)}{T_P} \right\rfloor = 0$$

$$\left\lfloor \frac{T_P - \varepsilon}{T_P} \right\rfloor = 0$$

which is obviously true for ε in $(0, T_P]$ with δ being the maximum time value where no tick is generated.

In the evaluations the initially fast nodes $U1, U2, U3$ and $U4$ are running precision π ticks ahead until the faulty nodes $U5$ and $U6$ stop contributing to the fast tick generation process. As a consequence the fast nodes are stalled until the slow nodes $U7$ and $U8$ have finally caught up (cf. Figure 6.3 and Figure 7.12). This catch-up yields a close synchronization of all nodes. However, the new mean frequency of all nodes is now determined by τ^+_{rem} in contrast to τ^-_{rem} as it was before the faulty nodes entirely stopped operation (the slow processing rate is not shown in the trace of Figure 7.12 but was observed in the experiment). The time interval in which no tick is generated by the previously fast nodes was measured to be not larger than $\delta = 120ns$. In the depicted measurement scenario τ^+_{rem} was tuned to $48ns$, in conjunction with $\tau^-_{rem} = 6ns$ this led to $T_{QS} = 42ns$. The assessment of T_P yielded $62ns$, thus, $\delta = 2 \cdot 62ns + 42ns = 168ns$. Comparing the measurement results to the calculations of δ, a notable mismatch can be observed. The predicted bound of $168ns$ holds but is not necessarily tight for the measured scenario.

The assessment of the accuracy's upper bound followed a strategy similar to the one presented above. However, in contrast to the longest time interval without generating a tick this evaluation is concerned with a correct node's fastest generation of successive TICK(k) and TICK(k+1) transitions. The measurement setup again relies on a dynamic switching between two different delay configurations (cf. Figure 6.4). For enhanced observability for all remote clock interconnections, τ^-_{rem} was artificially increased to $\approx 41ns$, while $\tau^-_{loc} = \tau^+_{loc}$ still remained at $6ns$. In this evaluation setup it is ensured that the fastest remotely driven

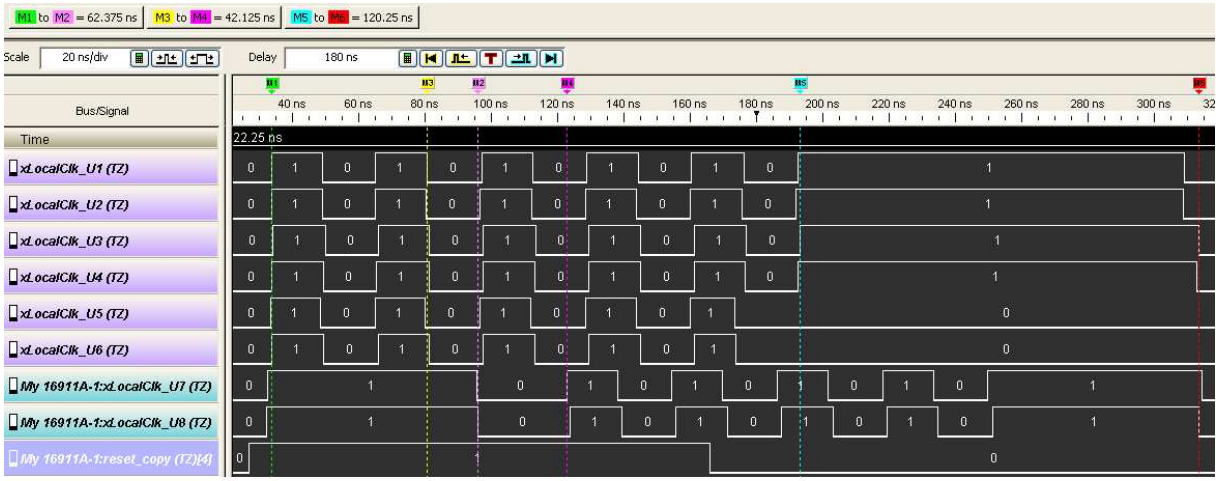


Figure 7.12: Verification measurement for accuracy lower bound

generation of a tick, given by T^{-}_{first}, and the fastest local generation T_{min} are spread apart. In particular, $T^{-}_{first} = 41ns$, while T_{min} is still in the range of $15ns$. From Equation 3.4 it is known that the accuracy's upper bound is given by

$$\min\left\{\left\lceil\frac{\delta}{T^{-}_{first}}\right\rceil + \pi, \left\lceil\frac{\delta}{T_{min}}\right\rceil\right\} \tag{7.1}$$

with δ being the observed interval. For the given measurement setup, obviously T_{min} will be the crucial term for generating ticks at maximum speed which was also observed in several evaluation runs with different settings for τ^{-}_{rem} and τ^{+}_{rem}. Thus, in the above given setup the time between two ticks is never smaller than $T_{min} = 15ns$, i.e., a clock frequency of approximately $33MHz$ represents the upper bound for the given delay configuration.

7.2.6 Queue size

The local and remote elastic pipelines' queue size of 8 stages has already been mentioned in the context of the precision assessment presented in Section 7.2.4. The evaluation setup to generate worst-case conditions for the local pipeline is identical to the initial setup of the lower bound accuracy experiment, however, no delay switching has to be employed. Equation 3.5 gives the formal bound for the local pipeline size, while the respective evaluation scenario has been detailed in Section 6.1.4. The paths between fast and slow nodes in both directions suffer a delay of τ^{+}_{rem}. Thus, tick removal from the fast nodes' local pipelines (corresponding to slow nodes) starts at an instant determined by the delays T_{QS} and τ^{+}_{rem}, while the fast nodes' tick generation inserts ticks into these pipelines at a rate given by T^{-}_{first}. The required local pipeline depth for given values of T_{QS}, τ^{+}_{rem} and

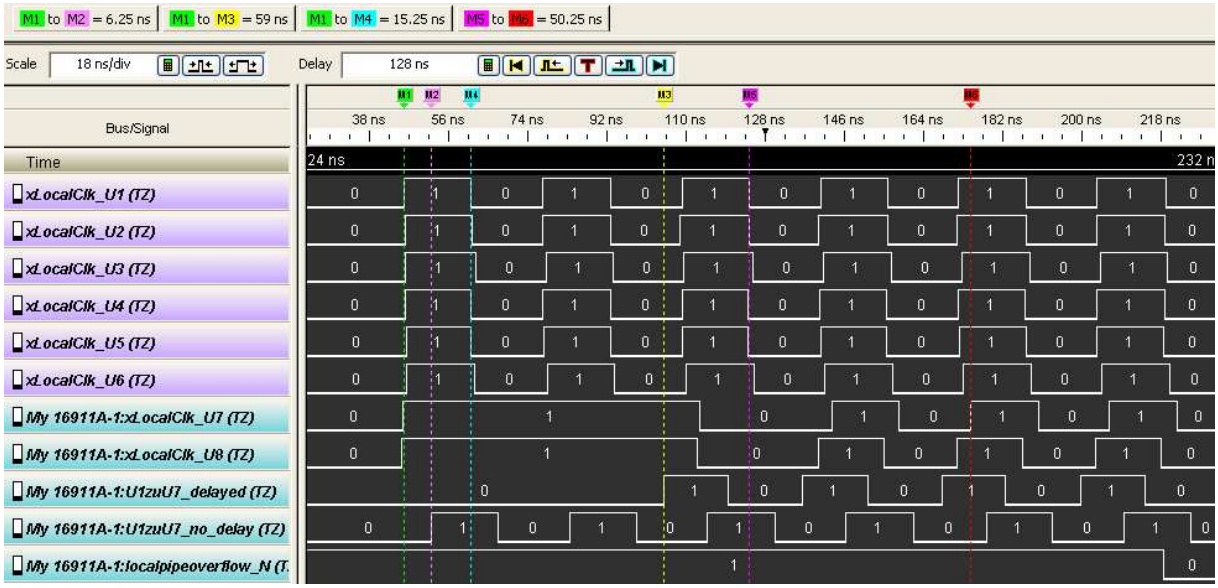


Figure 7.13: Local queue size bound verification

T^-_{first} can therefore be computed by

$$S_{loc} \approx \left\lceil \frac{T_{QS} + \tau^+_{rem} - \tau^-_{loc}}{T^-_{first}} \right\rceil + 2.$$

The logic analyzer trace shown in Figure 7.13 presents a configuration where the overflow detection unit of a fast node's local pipeline indicates (via the active low `localpipeoverflow_N` signal) that a tick has been lost. In a series of measurement campaigns the particular experiment represents the case with $\tau^+_{rem} = 59ns$, which denotes the smallest delay where overflowing local pipelines were observed. The relevant time delay values extracted from the experiment trace are $\tau^+_{rem} = 59ns, T_{QS} = 50ns, \tau^-_{rem} = \tau^-_{loc} = 6ns$ and $T^-_{first} = 15ns$ yielding,

$$S_{loc} \approx \left\lceil \frac{50ns + 59ns - 6ns}{15ns} \right\rceil + 2 = 9.$$

According to the evaluation results the required local queue size of 9 perfectly matches the prediction based on theory.

The delay measurements as well as the validation of the computed bound for the remote queue size have been performed similarly to the previously presented local queue size treatment. The main difference is given by the slightly modified evaluation setup already introduced in Figure 6.7(a). Additionally, it should be noted that in the implemented tick removal strategy the remote elastic pipelines may only buffer one transition less than the

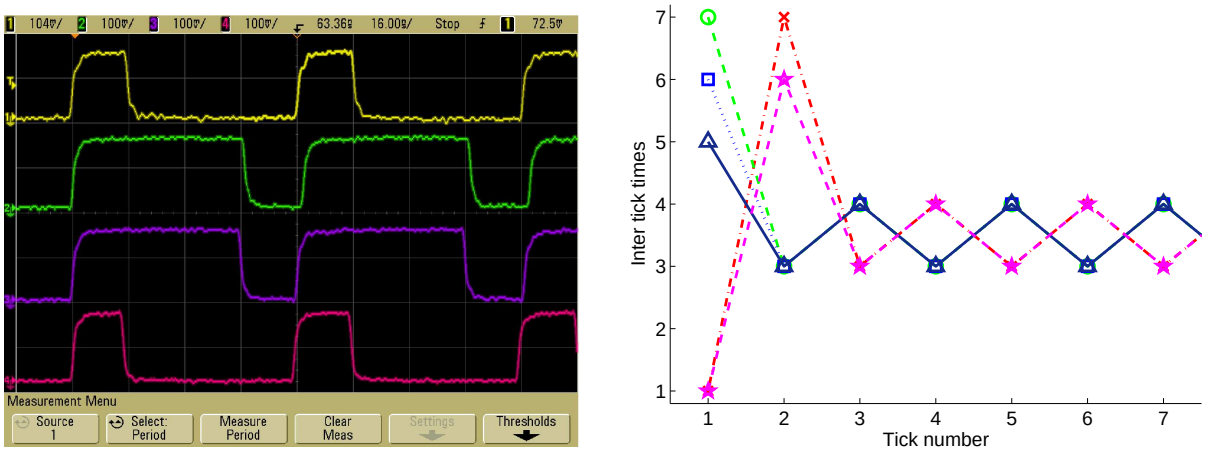


Figure 7.14: DARTS cluster with unbalanced delays (a) oscillations of tick generation periods (b) simulation of settling

number of Muller C-Elements in the pipeline, i.e., 7 instead of 8 ticks may be stored[6]. This reduction in pipeline depth is due to the fact that the remote pipeline's Muller C-Element next to the Difference Module can be seen as part of the Difference Module since it only propagates ticks if they have already been acknowledged by the local side. Taking into account the remote pipeline's decreased size the predictions based on theory have been validated again. The evaluations of the before presented worst-case scenario for the remote pipeline size showed by stepwise increasing T_{QS} that a pipeline depth of 7 Muller C-Elements is sufficient for $T_{QS} < 105ns$ with $\tau^{-}_{rem} = 6ns$ and $T^{-}_{first} = 15.5ns$.

7.2.7 Oscillations and start-up behavior

After the assessment of several average- and worst-case properties, the evaluations conclude with the analysis of the DARTS ensemble's sensitivity to unbalanced interconnection delays, i.e., $\tau^{+}_{rem} \neq \tau^{-}_{rem}$. It is well known that asynchronous circuits in general may exhibit varying processing delays in the execution of repetitive tasks. Figure 7.14(a) presents this kind of oscillation in tick generation speed in the example of an unbalanced 5-node DARTS cluster (6 out of the 20 remote links have been configured to suffer from excessive τ^{+}_{rem} delays).

In the context of delay-insensitive circuits (="wait-for-all" systems) Burns [11] modeled these oscillation effects using a suitable <max,+> representation of the circuit components. In the fault-tolerant DARTS framework, however, <max,+> is not sufficient since $f+1$ and $2f+1$ threshold functions cannot be represented. As a consequence, <min,max,+> algebra (cf. Section 6.3) has been used as foundation for the conducted simulations. The

[6]This implementation specific peculiarity is not covered by Equation 3.6 and thus has to be explicitly taken into account in the pipeline size assessment.

resulting MATLAB-models allow to predict the duration of every single clock period of a 5-node DARTS cluster. Measurements with different delay configurations confirmed the appropriateness and validity of the <min,max,+> representation. Furthermore, the simulations enabled a detailed analysis of a DARTS cluster's start-up behavior.

In the numerous measurements with initially all nodes in a reset state the DARTS nodes start generating the first tick closely synchronized to each other (maximum measured skew of $1.5ns$). In scenarios where no artificial delays had been added into the TG-Net all nodes stayed in tight synchrony with each other. However, in static scenarios with unbalanced interconnection delays (delays and operating conditions assumed to be constant during operation) it could be observed that the tick generation rate varied before it finally stabilized after a few ticks, cf. the local queue size measurement scenario presented in Figure 7.13. The particularly interesting observation is that the length of the settling period in which the DARTS system stabilizes, as well as potential oscillations of the clocks' periods only depend on the system's interconnection and processing delays. In order to get quantitative and qualitative estimations for the settling time and the characteristics of the oscillations, the <min,max,+> simulation model could be used again. Simulation results for a 5-node system are presented in Figure 7.14(b). The simulation trace reveals that after the concurrent generation of the first transitions the nodes split up into two groups. After some initial settling time (in the given example at TICK(3)) these two cliques further on operate with stable but phase shifted clock periods. If compared to the measurement results shown in Figure 7.14(a) it can be observed that simulations match the real-world behavior of DARTS clocks quite well. The oscilloscope snapshot confirms the two aforementioned clock cliques shown at channel 1 and 4, as well as channel 2 and 3. It should be mentioned that in addition to the presented scenarios with (small) initial prefix (=settling time) and oscillation of the clock period, setups without continuously oscillating periods are most likely to be encountered. In fact, setups where the mismatch of the propagation delay will only be small represents the typical case, which provides stable clock periods.

Table 7.3: Characteristics of HITS tick generation

parameter	value	comment	equation
technology	$180nm$	radiation tolerant process with custom Muller C-Element	-
average frequency	$53MHz$ $33MHz$	standard node experimental node	6.2
precision π	1 up to 8	measured typical configuration with balanced intercon. delays expnode design with heavily unbalanced delays	3.3
voltage dependence		1% voltage change yields 1% freq. change	6.1
fault tolerance	2 Byzantines	in a system of 8 nodes	$n \geq 3f + 2$

Chapter notes

The assessment of the most important characteristics of the HITS ASICs has been the main focus of this chapter and parts of the results are presented in Table 7.3. Besides the measurements of DARTS clocks' operating condition dependence, high effort has been put into the verification of the system's bounds predicted by theory. Regarding synchronization properties and queue size bounds, it can be concluded that none of the computed theoretical bounds have been violated, however, the tightness of some predictions could be further improved. In general, the tick generation scheme complies with the expected characteristics and therefore emphasizes the feasibility of the fault-tolerant clocking approach. However, some of the measured properties, e.g., the non-perfect stability under varying operating conditions, will require enhancements before synchronous circuits may be reliably clocked with DARTS clocks. As a consequence the evaluation of the DARTS tick generation scheme not only constitutes the final design validation step, it also provides the foundation for adaptations and improvements.

Chapter 8

Conclusions and Future Work

A conclusion is the place where you got tired of thinking.

Harold Fricklestein

THE WORK presented in this thesis ranges from nanoscale VLSI chip considerations to high-level distributed algorithm design. In the light of the continuous downsizing in chip technology and the accompanied reliability issues, the two at first glance very dissimilar fields of computer engineering have been identified to have several properties in common. In order to cope with robustness issues of modern VLSI circuits — in particular focusing on the clock signal — a fault-tolerant clocking scheme based on distributed algorithms has been designed. Thereby, the development process of the HITS ASICs greatly benefited from theoretical results provided by the distributed systems community. The focus of this thesis is placed on the design and implementation of the hardware block required to implement the underlying tick generation algorithm. Additionally, great efforts have been made to assess the implementation's properties and to validate the synchronization properties and implementation characteristics predicted by theory. The main, tangible result of the conducted work is given by the operational tick generation scheme which relies on a set of 8 interconnected HITS ASICs. Furthermore, the output from extensive evaluation campaigns provides valuable characterization data for the HITS chips as well as the whole DARTS clocking scheme.

The achievements presented in this thesis can be seen as proof of concept for successfully adopting certain results from the distributed systems community to solve problems in VLSI design. However, besides the general feasibility of the DARTS clocking scheme, a large potential for improvements has to be mentioned. Furthermore, the applicability of bounded but none-standard synchrony of the DARTS clocks still has to be shown. A list of the most important topics for further improvements and investigations is presented below.

Scaling with number of nodes n: The clocking scheme's requirement of fully connected TG-Algs yields a quadratic growth of the number of links with n. For large systems, e.g., $n > 20$, the interconnection effort might no longer be feasible. Additionally, increasing numbers of n have substantial impact on complexity of the hardware design. In particular, the employed sum of products threshold circuit architecture does not facilitate upscaling the number of nodes. Hence, both issues, but most importantly the ASIC design's unfavorable scaling with n should be considered in future generations of the DARTS clocking scheme.

Recovery after transient failures: As soon as a node has suffered from a fault it is considered faulty until the whole DARTS system is restarted. Obviously, this implicit mapping of transient fault effects to permanent failures is a non-ideal property of the clocking scheme. Hence, measures for fault detection and online reintegration of nodes seem to be crucial to be able to achieve reasonable mission times.

Increasing clock speed: The maximum clock speed of the presented first generation HITS chip of $\approx 55MHz$ can be considered quite good as it has not been extensively optimized and involves board-level communication. However, besides minor improvements due to design optimizations, e.g., of the threshold circuit implementation, more pronounced speed-up can be expected from conceptional refinements. Recent work [19,20] on this topic yielded promising results for increasing clock speed. The proposed enhancements rely on the pipelined execution of the tick generation algorithms, i.e., circuit and wire delays are used as pipeline for clock transitions thus allowing multiple ticks being in transition at the same time.

Metastability considerations: It has been ensured by design that correct DARTS nodes will never suffer from metastability issues. However, a fault might be able upset a Muller C-Element. The non-zero probability of metastability propagation has been investigated in [35]. This metastability analysis accounts for the fact that metastability theoretically has the potential for catastrophic fault propagation throughout the DARTS system. Fortunately, the analysis confirmed the expectation that Muller C-Elements have synchronizing properties and therefore lead to metastability decay. However, besides the qualitative evidence of the metastability decay a quantitative assessment still has to be conducted for the DARTS clocking scheme.

Reliable communication scheme: In order to fully benefit from the fault-tolerant clocks provided by DARTS, a suitable communication framework has to be employed. This framework has to be capable of coping with potentially substantial offsets between clocks, without losing clock speed due to synchronization approaches like clock dividers. A suitable communication scheme achieving operation at the full DARTS clock speed has recently been proposed in [72].

Bibliography

[1] E. Anceaume, C. Delporte-Gallet, H. Fauconnier, M. Hurfin, and J. Widder. Clock synchronization in the byzantine-recovery failure model. In *International Conference On Principles Of DIstributed Systems OPODIS 2007*, LNCS, pages 90–104, Guadeloupe, French West Indies, Dec. 2007. Springer Verlag.

[2] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd ed.)*. John Wiley & Sons, Inc., Apr. 2004.

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

[4] B. Barak, S. Halevi, A. Herzberg, and D. Naor. Clock synchronization with faults and recoveries (extended abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 133–142, Portland, Oregon, United States, 2000. ACM Press.

[5] R. Baumann. Soft errors in advanced computer systems. *IEEE Design & Test of Computers*, 22(3):258–266, May-June 2005.

[6] V. Beiu, J. M. Quintana, and M. J. Avedillo. VLSI Implementations of Threshold Logic – A Comprehensive Survey. *IEEE Transactions on Neural Networks*, 14(5):1217–1243, Sept. 2003.

[7] M. Biely, G. Fuchs, and M. Fuegger. Clock synchronization in the crash-recovery failure model. Research Report 1/2008, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-2, 1040 Vienna, Austria, 2008.

[8] D. L. Black. On the existince of delay-insensitive fair arbiters: Trace theory and its limitations. *Distributed Computing*, 1:205–225, 1986.

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

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

[11] S. M. Burns. *Performance analysis and optimization of asynchronous circuits*. PhD thesis, Pasadena, CA, USA, 1991.

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

[13] Y. Cheng and D.-Z. Zheng. A cycle time computing algorithm and its application in the structural analysis of min-max systems. *Discrete Event Dynamic Systems*, 14(1):5–30, 2004.

[14] B.-R. Choi, K. Park, and M. Kim. An improved hardware implementation of the fault-tolerant clock synchronization algorithm for large multiprocessor systems. *IEEE Transactions on Computers*, 39(3):404–407, Mar. 1990.

[15] C. Constantinescu. Trends and challenges in VLSI circuit reliability. *IEEE Micro*, 23(4):14–19, July 2003.

[16] F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3(3):146–158, 1989.

[17] J. Dama and A. Lines. GHz asynchronous SRAM in 65nm. In *15th IEEE Symposium on Asynchronous Circuits and Systems, 2009. ASYNC '09*, pages 85–94, May 2009.

[18] M. Delvai. *Design of an Asynchronous Processor Based on Code Alternation Logic – Treatment of Non-Linear Data Paths*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, 2005.

[19] A. Dielacher, M. Fuegger, and U. Schmid. How to speed-up fault-tolerant clock generation in VLSI systems-on-chip via pipelining. In *Proceedings of the 27th ACM Symposium on Principles of Distributed Computing (PODC'08)*, page 423. ACM Press, Aug. 2008. An extended version is available as RR 15/2009, Institut für Technische Informatik, TU-Wien, http://www.vmars.tuwien.ac.at/documents/extern/2571/techreport.pdf.

[20] A. Dielacher, M. Fuegger, and U. Schmid. How to speed-up fault-tolerant clock generation in VLSI systems-on-chip via pipelining. Research Report 15/2009, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2009. http://www.vmars.tuwien.ac.at/documents/extern/2571/techreport.pdf.

[21] D. Dobberpuhl et al. A 200-MHz 64-bit dual-issue CMOS microprocessor. *IEEE J. Solid-State Circuits*, 27(11):1555–1567, Nov. 1992.

[22] R. Dobkin, R. Ginosar, and C. Sotiriou. Data synchronization issues in GALS SoCs. pages 170–179, April 2004.

[23] D. Dolev, J. Y. Halpern, and H. R. Strong. On the possibility and impossibility of achieving clock synchronization. *Journal of Computer and System Sciences*, 32:230–250, 1986.

[24] S. Fairbanks. Method and apparatus for a distributed clock generator, 2004. US patent no. US2004108876.

[25] S. Fairbanks and S. Moore. Self-timed circuitry for global clocking. In *Proceedings of the Eleventh International IEEE Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 86–96, Mar. 2005.

[26] K. Fant and S. Brandt. Null convention logic(tm): a complete and consistent logic for asynchronous digital circuit synthesis. In *Proceedings of the International Conference on Application Specific Systems, Architectures and Processors*, pages 261–273, Aug. 1996.

[27] M. Favalli and C. Metra. TMR voting in the presence of crosstalk faults at the voter inputs. *IEEE Transactions on Reliability*, 53(3):342–348, Sept. 2004.

[28] M. Ferringer, G. Fuchs, A. Steininger, and G. Kempf. VLSI Implementation of a Fault-Tolerant Distributed Clock Generation. *IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT2006)*, pages 563–571, Oct. 2006.

[29] M. Fischer, N. Lynch, and M. Merritt. Easy impossibility proofs for the distributed consensus problem. *Distributed Computing*, 1(1):26–39, 1986.

[30] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

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

[32] W. Friesenbichler, T. Panhofer, and M. Delvai. Improving fault tolerance by using reconfigurable asynchronous circuits. *IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*, Apr. 2008.

[33] G. Fuchs. Implications of VLSI fault models and distributed systems failure models – a hardware designer's view. In B. Charron-Bost, S. Dolev, J. Ebergen, and U. Schmid, editors, *Fault-Tolerant Distributed Algorithms on VLSI Chips*, number 08371 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2009. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[34] G. Fuchs, M. Fuegger, U. Schmid, and A. Steininger. Mapping a fault-tolerant distributed algorithm to systems on chip. In *11th Euromicro conference on Digital System Design Architectures, Methods and Tools (DSD'08)*, pages 242–249, Parma, Italy, September 2008.

[35] G. Fuchs, M. Fuegger, and A. Steininger. On the threat of metastability in an asynchronous fault-tolerant clock generation scheme. In *15th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'09)*, Chapel Hill, N. Carolina, USA, May 2009.

[36] G. Fuchs, M. Fuegger, A. Steininger, and F. Zangerl. Analysis of constraints in a fault-tolerant distributed clock generation scheme. *3rd International Workshop on Dependable Embedded Systems (WDES'06)*, Oct. 2006.

[37] G. Fuchs, J. Grahsl, U. Schmid, A. Steininger, and G. Kempf. Threshold Modules – Die Schlüsselelemente zur Verteilten Generierung eines Fehlertoleranten Taktes. In *Proceedings of the Austrian National Conference on the Design of Integrated Circuits and Systems (Austrochip 2006)*, pages 149–156, Vienna, Oct. 2006.

[38] M. Fuegger, G. Fuchs, U. Schmid, and A. Steininger. On the stability and robustness of non-synchronous circuits with timing loops. *3rd Workshop on Dependable and Secure Nanocomputing*, Jun. 2009.

[39] M. Fuegger, U. Schmid, G. Fuchs, and G. Kempf. Fault-Tolerant Distributed Clock Generation in VLSI Systems-on-Chip. In *Proceedings of the Sixth European Dependable Computing Conference (EDCC-6)*, pages 87–96. IEEE Computer Society Press, Oct. 2006.

[40] M. Fuegger, U. Schmid, G. Fuchs, A. Steininger, G. Kempf, and M. Sust. Fault-tolerant distributed tick generation in VLSI systems-on-chip. Research Report 53/2009, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-2, 1040 Vienna, Austria, 2009.

[41] R. Ginosar. Fourteen ways to fool your synchronizer. *International Symposium on Asynchronous Circuits and Systems*, page 89, 2003.

[42] J. Gunawardena. Cycle times and fixed points of min-max functions. In *11th International Conference on Analysis and Optimization of Systems*, pages 266–272. Springer, 1994.

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

[44] B. Heidergott, G. J. Olsder, and J. von der Woude. *Max plus at work*. Princeton Univ. Press, 2006.

[45] R. Höller, M. Horauer, G. Gridling, N. Kerö, U. Schmid, and K. Schossmaier. SynUTC - high precision time synchronization over Ethernet networks. In *Proceedings of the 8th Workshop on Electronics for LHC Experiments (LECC'02)*, pages 428–432, Colmar, France, Sept. 9–13, 2002.

[46] A. Hopkins, I. Jr. Smith, T.B., and J. Lala. FTMP – a highly reliable fault-tolerant multiprocess for aircraft. In *Proceedings of the IEEE*, volume 66, pages 1221–1239, Oct. 1978.

[47] International technology roadmap for semiconductors, 2007.

[48] W. Jang and A. J. Martin. SEU-tolerant QDI circuits. In *Proceedings 11th Int'l Symposium on Asynchronous Circuits and Systems (ASYNC'05)*, pages 156–165, 2005.

[49] R. M. Kieckhafer, C. J. Walter, A. M. Finn, and P. M. Thambidurai. The MAFT architecture for distributed fault tolerance. *IEEE Transactions on Computers*, 37:398–405, Apr. 1988.

[50] H. Kopetz. *Real-Time Systems, Design Pinciples for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

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

[52] L. Lamport. The mutual exclusion problem: Part I–the theory of interprocess communication. *Journal of the ACM*, 33(2):313–326, 1986.

[53] L. Lamport. Arbitration-free synchronization. *Distributed Computing*, 16(2/3):219–237, September 2003.

[54] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, Jan. 1985.

[55] J. Lundelius and N. Lynch. A new fault-tolerant algorithm for clock synchronization. In *Proc. 3rd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 75–88, August 1984.

[56] A. Martin, M. Nystrom, K. Papadantonakis, P. Penzes, P. Prakash, C. Wong, J. Chang, K. Ko, B. Lee, E. Ou, J. Pugh, E.-V. Talvala, J. Tong, and A. Tura. The lutonium: a sub-nanojoule asynchronous 8051 microcontroller. pages 14–23, May 2003.

[57] A. J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1:226–234, 1986.

[58] A. J. Martin. Limitations to delay-insensitivity in asynchronous circuits. Technical report, California Institute of Technology, Pasadena, CA, USA, 1990.

[59] A. J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In *AUSCRYPT '90: Proceedings of the sixth MIT conference on Advanced research in VLSI*, pages 263–278, Cambridge, MA, USA, 1990. MIT Press.

[60] M. S. Maza and M. L. Aranda. Analysis of clock distribution networks in the presence of crosstalk and groundbounce. In *Proceedings International IEEE Conference on Electronics, Circuits, and Systems (ICECS)*, pages 773–776, 2001.

[61] M. S. Maza and M. L. Aranda. Interconnected rings and oscillators as gigahertz clock distribution nets. In *GLSVLSI '03: Proceedings of the 13th ACM Great Lakes symposium on VLSI*, pages 41–44. ACM Press, 2003.

[62] M. S. Maza and M. L. Aranda. Analysis and verification of interconnected rings as clock distribution networks. In *GLSVLSI '04: Proceedings of the 13th ACM Great Lakes symposium on VLSI*, pages 312–315. ACM Press, 2004.

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

[64] P. Miner, P. Padilla, and W. Torres. A provably correct design of a fault-tolerant clock synchronization circuit. pages 341–346, Oct 1992.

[65] G. Moore. Progress in digital integrated electronics. *Technical Digest IEEE International Electron Devices Meeting*, pages 11–13, 1975.

[66] J. Muttersbach, T. Villiger, and W. Fichtner. Practical design of globally-asynchronous locally-synchronous systems. pages 52–59, 2000.

[67] C. J. Myers. *Asynchronous Circuit Design*. John Wiley & Sons, Inc., 2001.

[68] S. D. Naffziger, G. Colon-Bonet, T. Fischer, R. Riedlinger, T. J. Sullivan, and T. Grutkowski. The Implementation of the Itanium 2 Microprocessor. *IEEE Journal of Solid-State Circuits*, 37(11):1448–1460, Nov. 2002.

[69] S. M. Nowick and C. W. O. Donnell. On the existence of hazard-free multi-level logic. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 109–120. IEEE Computer Society Press, May 2003.

[70] M. Omana, D. Rossi, and C. Metra. Fast and low-cost clock deskew buffer. In *19th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT 2004)*, pages 202–210, Oct. 2004.

[71] M. Omana, D. Rossi, and C. Metra. Low Cost Scheme for On-Line Clock Skew Compensation. In *Proceedings of the IEEE VLSI Test Symposium*, pages 90–95, May 2005.

[72] T. Polzer, T. Handl, and A. Steininger. A metastability-free multi-synchronous communication scheme for fault-tolerant SoCs. Research Report 10/2009, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2009.

[73] P. Ramanathan, K. Shin, and R. Butler. Fault-tolerant clock synchronization in distributed systems. *Computer. IEEE Computer Society Press*, 23(10):30–42, Oct. 1990.

[74] Restle et al. The clock distribution of the power4 microprocessor. In *IEEE International Solid-State Circuits Conference ISSCC, Digest of Technical Papers.*, volume 2. IEEE, 2002.

[75] D. Rossi, M. Omana, F. Toma, and C. Metra. Multiple transient faults in logic: an issue for next generation ICs? In *20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT 2005)*, pages 352–360, Oct. 2005.

[76] N. Seifert, P. Shipley, M. Pant, V. Ambrose, and B. GiII. Radiation-induced clock jitter and race. In *Proceedings 43rd Annual IEEE International Reliability Physics Symposium*, pages 215–222, 17-21, 2005.

[77] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. *Proceedings of International Conference on Dependable Systems and Networks, DSN*, pages 389–398, 2002.

[78] M. L. Shooman. *Reliability of Computer Systems and Networks*. Wiley, 2002.

[79] B. Simons, J. Lundelius-Welch, and N. Lynch. An overview of clock synchronization. In B. Simons and A. Spector, editors, *Fault-Tolerant Distributed Computing*, LNCS 448, pages 84–96. Springer Verlag, 1990.

[80] J. Sparsø and S. Furber. *Principles of Asynchronous Circuit Design*. Dimes, 2001.

[81] T. K. Srikanth and S. Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, Apr. 1987.

[82] A. Steininger, T. Handl, G. Fuchs, and F. Zangerl. Testing the hardware implementation of a distributed clock generation algorithm for SoCs. *IEEE East-West Design and Test International Workshop*, pages 59–64, Sept. 2006.

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

[84] J. Teifel and R. Manohar. Highly pipelined asynchronous FPGAs. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 133–142, New York, NY, USA, 2004. ACM.

[85] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2000.

[86] C. Uri. Terabit crossbar switch core for multi-clock-domain SoCs. In *Proceedings of the 15th Symposium on High Performance Chips (HOT CHIPS)*, page 102ff, 2003.

[87] K. van Berkel. Beware the isochronic fork. *Integr. VLSI J.*, 13(2):103–128, 1992.

[88] D. VanAlen and A. Somani. An all digital phase locked loop fault tolerant clock. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 3170–3173, June 1991.

[89] N. Vasanthavada and P. Marinos. Synchronization of fault-tolerant clocks in the presence of malicious failures. *IEEE Transactions on Computers*, 37(4):440–448, Apr. 1988.

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

[91] N. H. E. Weste and K. Eshraghian. *Principles of CMOS VLSI design: a systems perspective*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1985.

[92] J. Widder. *Distributed Computing in the Presence of Bounded Asynchrony*. PhD thesis, Vienna University of Technology, Fakultät für Informatik, May 2004.

[93] J. Widder and U. Schmid. The Theta-Model: Achieving synchrony without clocks. *Distributed Computing*, 2009. (to appear).

[94] L. Wissel, S. Pheasant, R. Loughran, C. LeBlanc, and B. Klaasen. Managing soft errors in ASICs. *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 85–88, 2002.

Curriculum Vitae
Gottfried Fuchs

Date of Birth:	March, 15^{th}, 1978
Place of Birth:	Oberpullendorf, Austria
1984–1988	Elementary school Deutschkreutz
1988–1991	Secondary school Oberpullendorf
1991–1992	Secondary school BRG Wien XXIII
1992–1998	Technical High School Mödling, Curriculumn Electronic and Telecommunication Engineering with focus on Computer Engineering
1998–2004	Master Curriculum Computer Science at the Vienna University of Technology, with focus on Computer Engineering Graduation with distinction
2004–2005	Bachelor Curriculum Computer Science Management at the Vienna University of Technology
2004–2009	PhD Curriculum Computer Engineering at the Vienna University of Technology
2004–2009	Research assistant at the Vienna University of Technology, Institute for Computer Engineering, Embedded Computing Systems Group
2005	Master Curriculum Computer Science Management at the Vienna University of Technology Graduation with distinction