A Metastability-Free Multi-Synchronous Communication Scheme for SoCs

T. Polzer, T. Handl*, and A. Steininger

Vienna University of Technology
Institute of Computer Engineering – Embedded Computing Systems Group
Treitlstrasse 3, A-1040 Vienna, Austria

Abstract. We propose a communication scheme for GALS systems with independent but approximately synchronized clock sources, which guarantees high-speed metastability-free communication between any two peers via bounded-size FIFO buffers. The proposed approach can be used atop of any multi-synchronous clocking system that guarantees a synchronization precision in the order of several clock cycles, like our fault-tolerant DARTS clocks. We determine detailed formulas for the required communication buffer size, and prove that this choice indeed guarantees metastability-free communication between correct peers, at maximum clock speed. We also describe a fast and efficient implementation of our scheme, and calculate the required buffer size for a sample test scenario. Experimental results confirm that the size lower bounds provided by our formulas are tight in this setting.

1 Introduction

Over the last decades, VLSI technology has been dominated by the trends towards shrinking feature sizes, increasing complexity and higher clock rates. The VLSI design style, however, was shaped by the synchronous abstraction and the assumption of relatively low component failure rates that do not match well the above VLSI technology trends: First of all, clock frequencies became so high (and chips so complex) that phase-synchronous clock distribution over the entire chip is a substantial challenge [1]. Chip designers are hence confronted with the erosion of the convenient globally synchronous clock abstraction, which makes GALS (globally asynchronous locally synchronous [2]) systems an attractive alternative. In GALS, the system is partitioned into modules that are compact enough to be designed safely using the synchronous paradigm. Each module is equipped with its own clock source, however, and communication across different clock domains is performed in an asynchronous way.

The second problem with contemporary VLSI technology are increasing error rates [3] due to smaller critical charges, lower voltage swings and high clock and signal frequencies. Hardening techniques at the technology and circuit level [4, 5], and redundancy concepts at the architectural level [6], as known from dependable computing, are hence expected to become widespread in commercial

* This research is supported by the Austrian bm:vit FIT-IT project *DARTS*.

circuits as well. Interestingly, however, even such designs often rely on a single clock source, usually a quartz oscillator, that ultimately constitutes a (silently accepted) single point of failure: Typically, brute-force approaches for providing a robust clock, such as using very strong drivers, are employed here. More sophisticated alternatives like [7, 8] provide very limited fault-tolerance only.

Again, GALS seems to come for a rescue. However, while multiple independent clock sources indeed eliminate the single point of failure, this comes at a high price: The global notion of time that was naturally provided by the synchronous clock does not exist anymore. The modules are running asynchronously to each other, which not only complicates the application design, but also prohibits TMR or voter-based fault-tolerant architectures. Moreover, asynchrony introduces the potential of metastability [9]: Synchronizers must be employed to mitigate these effects at the clock domain boundaries between modules. In order to cope with (slowly) drifting clocks, adaptive synchronizers have been proposed [10, 11]. Although the meantime between metastable upsets can be made arbitrarily large, they cannot be eliminated completely. Moreover, synchronizers tend to degrade performance, since their designs need to be conservative [12].

Therefore, an inter-module communication scheme for GALS that rules out metastability by construction would be an appealing alternative to synchronizer-based asynchronous communication. In this paper, we describe such a communication scheme for GALS systems built atop a multi-synchronous clocking scheme. A multi-synchronous clocking scheme employs independent clock sources that, however, guarantee some known bound on the worst-case synchronization precision, typically in the order of a few clock cycles. One example is our fault-tolerant DARTS clocking scheme for SoC [13]. Although the synchronization precision of multi-synchronous clocking is much worse than that of a conventional synchronous clock, it turns out that its "loose" global synchrony is – in contrast to GALS – sufficient for implementing metastability-free communication at full clock speed, i.e., without reducing the data rate. Implementing this scheme only requires bounded-size FIFO buffers between communicating modules. To the best of our knowledge, there is no related work that targets similar goals.

2 System Architecture

We assume a system consisting of different functional units (FUs) that are internally complex synchronous designs with well defined interfaces to the outside. A typical example is a system-on-chip (SoC) built from IP modules. In the following we will hence treat the FUs as black boxes and concentrate on the inter-FU communication. Although we do not assume the individual FUs and channels to be fault-tolerant, we can actually restrict our attention to the communication between non-faulty FUs, along non-faulty communication channels: If required by the application, fault-tolerance is rather achieved at the architectural level by appropriate replication. Since the behavior of faulty FUs and channels can be disregarded here, we can safely disregard failures in the sequel.

2.1 Communication Issues

A fundamental requirement for every communication primitive is that a data item must not be changed by a write operation while being read. It is well known that, in this case, the receiver may remain undecided about the interpretation of the input for an arbitrary time and exhibit arbitrary behavior, such as self-oscillation or undecided output. This undesired effect is known as metastability [9], and causes the need to appropriately align the activities of source and sink.

In principle, the globally synchronous abstraction provides a very efficient solution here: A global clock source is used to clock all individual FUs and co-ordinate their activities. As outlined in the previous section, however, the globally synchronous abstraction is increasingly difficult to maintain, and enormous efforts are hence being made to establish a sufficiently tight co-ordination between sender and receiver at high clock frequencies.

Unfortunately, GALS does not solve this problem: Due to the lack of a common time reference, the communication between FUs needs to be explicitly coordinated by means of handshakes, which severely degrades communication performance. Moreover, a handshake can provide a coordination on a transaction level, but does not solve the synchronization issue at the signal level: The transfer of data across clock domains inevitably introduces the potential for metastability, even during fault-free operation. As already mentioned, synchronizers can only reduce (but not eliminate) this risk and introduce performance penalties [12].

2.2 Multi-Synchronous Clocking

In between the globally synchronous and the GALS clocking schemes there is a "loosely synchronized" scheme termed multi-synchronous clocking [14, 15]. With this approach, all FUs receive a clock of the same frequency on the long run, but with a significant short-term phase jitter that may amount to several clock periods. The worst-case phase deviation between any two FUs' clocks that will ever be encountered is termed the precision (measured in clock ticks), and is usually limited by design. A trivial example of a multi-synchronous solution is a globally synchronous system with significant skew in the clock distribution tree.

Interestingly, there are ways to implement a multi-synchronous clocking scheme also in a fully distributed and fault-tolerant manner: In our DARTS project, we adapted a simple Byzantine fault-tolerant distributed tick generation algorithm introduced in [16] for direct implementation in asynchronous digital logic [17]. Rather than using quartz oscillators, DARTS hence employs a special distributed fault-tolerant ring oscillator.

In [13], it was shown that DARTS clocks guarantee some bounded worst-case precision (π), provided that (1) some relatively uncritical layout timing constraints are met, and (2) that at most f out of the $n \geq 3f+2$ clock instances suffer from arbitrary (Byzantine) failures. Formulas for both the minimum (T^-) and maximum (T^+) clock cycle time have also been determined, as they are needed for determining the required communication buffer size in Sect. 4.2.

3 The Communication Layer

3.1 Fundamentals

The communication scheme traditionally used on top of the globally synchronous paradigm relies on a precision that is considerably better than one single clock cycle: The sender writes data at the active clock transition k, and the receiver reads these data at the next active clock transition $k + 1$. Assuming perfect synchrony, this leaves one clock period for transmission and stabilization of the data. In case of non-perfect synchronization between sender and receiver, however, this interval may shrink, as transition k is determined by the sender's local perception of time, but $k+1$ by the receiver's. Evidently, this simple scheme will no longer work reliably in a multi-synchronous environment, where the synchronization precision may be as large as several clock periods.

In order to reason precisely about such communication schemes for multi-synchronous GALS systems, we employ the notation of precedence described by Lamport in [18]. Informally, the precedence relation $A \rightarrow B$ means that action A must have been finished before action B starts. Given a set P of FUs, termed nodes in the sequel, and denoting the ticks of our multi-synchronous clocking system by C_i^k, where i is the node and k the tick count, we can define the precision π of our system by requiring that

$$\forall i, j \in P, \forall k > 0 : C_i^k \rightarrow C_j^{k+\pi} \tag{1}$$

A simple approach would be to divide the native clock ("microticks") such that the resulting clock ("macroticks") has a precision better than one. The drawback of this approach is the limited throughput. To circumvent it, we suggest implementing metastability-free communication [18] using a pipelined communication scheme directly based on the microticks.

3.2 Pipelined Communication

Let us recall the requirement for metastability-free communication from Sect. 2.1: For the transfer of any given data item, we need to pair write and read transitions such that writing has finished safely before reading starts. In the above synchronous approach, clock transitions of the same direction (rising or falling) are considered indistinguishable. Hence, this pairing is applied strictly via subsequent *alternating* edges. Consequently the phase relation between any two FUs is of central importance and must be maintained within tight bounds.

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

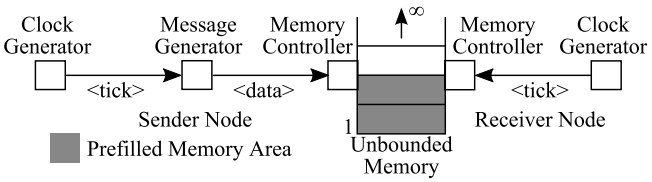

Fig. 1. System model used for the proof

Based on this idea, we can pipeline transmission activities at the microtick level, thereby avoiding the throughput penalty of macrotick-based communication. We simply exploit the precedence given in (1)

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

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

Note that writes and reads can be performed at every microtick here, which maximizes the throughput. As the synchronization precision, however, can be in the order of several microticks, one needs a FIFO buffer in between communicating nodes to avoid data loss. Clearly, minimizing the required buffer size is important, both with respect to costs and communication delay. In the following we will therefore formally derive the respective bound.

4 Formal Model and Proof

4.1 System Model

To be able to prove the correctness of our approach, we first create an algorithmic model of our system. In this model, the nodes are coupled by a single-writer single-reader buffer memory of unbounded size (see Fig. 1). Our proof will reveal that finite buffer size will be sufficient. The behavior of the system is modeled by a sender algorithm (Algorithm 1) and a receiver algorithm (Algorithm 2).

Informal description of the algorithms: In our description, we use the term action for an operation with duration ≥ 0. For action A, $t_s(A)$ and $t_e(A)$, respectively, denote A's start and end time. An action with zero duration is called an event. The following actions and messages can be handled and/or produced by the sender node i:

Algorithm 1 Sender algorithm for node i

1: **on** C_i^k **do**
2: Send $\langle$tick, $k\rangle$ to generate D_i // Simulate Clk Delay
3: **on** D_i^l: l-th receive of any $\langle$tick, $k\rangle$ from node i **do**
4: Send $\langle$data, $l\rangle$ to generate W_i // Simulate Message Delay
5: **on** W_i^m: m-th receive of any $\langle$data, $l\rangle$ from node i **do**
6: $mem(m + \alpha)$:= data // Memory Write Action

Algorithm 2 Receiver algorithm for node j

1: **on** C_j^k **do**
2: Send $\langle$tick, $k\rangle$ to generate R_j // Simulate Clk Delay
3: **on** R_j^l: l-th receive of any $\langle$tick, $k\rangle$ from node j **do**
4: $data := mem(k)$ // Memory Read Event

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

The receiver node can produce/handle the following actions and messages:

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

It is important to note that R_j^k reads memory location k, while W_i^k writes memory location $k+\alpha$. As a consequence of the shifted write index, the memory must be pre-filled with α elements (all zero), simulating that the writes $W_i^{-\alpha+1}, \ldots, W_i^0$ to the memory locations $1, \cdots, \alpha$ have already been finished before the first clock tick $k = 0$ (initial state). A sample execution of tick k for both algorithms can be found in Fig. 2.

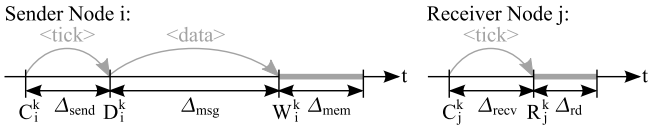

Fig. 2. Execution of tick k

4.2 Prerequisites

We require the clocking scheme to guarantee the following properties:

Assumption 1 (Precision). $\exists \pi : \forall i, j \in P, \forall k \geq 0 : C_i^k \rightarrow C_j^{k+\pi}$

Assumption 2 (Accuracy). $\forall i \in P, k > 0 : \exists T^- = \min_{i,k}(C_i^{k+1} - C_i^k) > 0$

Assumption 3 (Startup). *Before the first clock tick (initial state, $k = 0$), all memories are prefilled with α elements (all zero) and the precision π is zero ($\pi_0 = 0$). This is easy to guarantee in systems with a common reset.*

Assumption 4 (Message Order). *All message channels provide FIFO ordering. Furthermore, the actual delays must be such that every read and write operation is finished before the next one starts. This is always guaranteed if*

$$T^- + \Delta_{\text{send}}(i, k+1) + \Delta_{\text{msg}}(i, k+1) - \Delta_{\text{send}}(i, k) - \Delta_{\text{msg}}(i, k) > \Delta_{\text{mem}}(i, k)$$
$$\text{and } T^- + \Delta_{\text{recv}}(j, k+1) - \Delta_{\text{recv}}(j, k) > \Delta_{\text{rd}}(j, k)$$

For a more in-depth discussion of the delays in real systems see Sect. 5.1.

4.3 Problem Definition and Relation Between Events

Properties of correct operation:

(WR) The write of memory location k must be finished before the read of this location starts ($W_i^{k-\alpha} \rightarrow R_j^k$).

(OV) In case of a bounded-size buffer, the read of an element must be finished before it is overwritten ($R_j^k \rightarrow W_i^{k+\pi+\beta}$, the size of β will be fixed later).

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

Lemma 1. *Algorithm 1, line 3: $\forall k \geq 1$, it holds that $k = l$ and $D_i^k \rightarrow D_i^{k+1}$.*

Proof. We prove this Lemma by induction.

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

□

Lemma 2. *Algorithm 1, line 5: $\forall l \geq 1$. it holds that $l = m$ and $W_i^l \rightarrow W_i^{l+1}$.*

Proof. The proof is similar to above.

- Induction start ($l = 1$): D_i^1 triggers the first send of a message $\langle$data, 1$\rangle$. By the FIFO property of the links it is also the first message to be delivered and therefore triggering event W_i^1. Since it is the first event the precedence relation is valid.
- Induction hypothesis: Assume the lemma holds for l.
- Induction step ($l \to l+1$): We know that the first l $\langle$data, $m\rangle$ messages trigger the events $W_i^m | m \leq l$. By FIFO order message $\langle$data, $l+1\rangle$ (generated by event D_i^{l+1}) will be the next one delivered, thereby triggering the event W_i^{l+1}. By Assumption 4 we know that $t_\mathrm{s}(W_i^{l+1}) > t_\mathrm{s}(W_i^l) + \Delta_\mathrm{mem}(i,l)$ and therefore W_i^l is finished before W_i^{l+1} is started. Therefore $W_i^l \to W_i^{l+1}$ holds. □

We now define a new relation $\rightsquigarrow$. It is used to model the triggering of events. $A \rightsquigarrow B$ means that event B was triggered by event A. Note that $A \rightsquigarrow B$ implies the precedence relation (A $\to$ B). Using this notation, the trigger dependencies implied by Lemma 1 and 2 read: $C_i^k \rightsquigarrow D_i^k \rightsquigarrow W_i^k$.

Lemma 3. *Algorithm 2, line 3:* $\forall k \geq 1$*, it holds that* $k = l$ *and* $R_j^k \to R_j^{k+1}$.

The proof is equivalent to the one of Lemma 2. In conjunction with Lemma 3, this implies $C_j^k \rightsquigarrow R_j^k$.

4.4 Write-Read Order Proof

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

Lemma 4. $\forall -\alpha + 1 \leq k \leq 0 : t_\mathrm{e}(W_i^k) = 0$

Proof. Follows directly from Assumption 3. □

Lemma 5 gives the latest possible end time for all other write operations.

Lemma 5. $\forall k > 0 : t_\mathrm{e}(W_i^k) \leq t(C_i^k) + \Delta_\mathrm{send}^+ + \Delta_\mathrm{msg}^+ + \Delta_\mathrm{mem}^+$

Proof. We already know that $C_i^k \rightsquigarrow D_i^k \rightsquigarrow W_i^k$. Since D_i^k is triggered by the k-th $\langle$tick, $k\rangle$ message, we get: $t(D_i^k) = t(C_i^k) + \Delta_\mathrm{send}(i,k)$. Since W_i^k is triggered by the k-th $\langle$data, $l\rangle$ message, we get:

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

We know that the event takes $\Delta_\mathrm{mem}(i,k)$ time to finish, so its end time is:

$$t_\mathrm{e}(W_i^k) = t_\mathrm{s}(W_i^k) + \Delta_\mathrm{mem}(i,k) \leq t(C_i^k) + \Delta_\mathrm{send}^+ + \Delta_\mathrm{msg}^+ + \Delta_\mathrm{mem}^+. \ \square$$

We determine the earliest possible time a read operation can start.

Lemma 6. $\forall k > 0 : t_\mathrm{s}(R_j^k) \geq t(C_j^k) + \Delta_\mathrm{recv}^-$

Proof. We already know that $C_j^k \rightsquigarrow R_j^k$. Since R_j^k is triggered by $\langle \mathrm{tick},\ k \rangle$, we link: $t_\mathrm{s}(R_j^k) = t(C_j^k) + \Delta_\mathrm{recv}(j,k) \geq t(C_j^k) + \Delta_\mathrm{recv}^-$. □

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

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

Lemma 7. $\forall k > 0 : t_\mathrm{s}(R_j^k) - t_\mathrm{e}(W_i^{k-\alpha}) \geq 0$

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

- $1 \leq k \leq \alpha$: $t_\mathrm{s}(R_j^k) - \underbrace{t_\mathrm{e}(W_i^{k-\alpha})}_{=0 \text{ by Lemma 4}} = t_\mathrm{s}(R_j^k) \geq 0.$
- $k > \alpha$:

$$\begin{aligned}
&t_\mathrm{s}(R_j^k) - t_\mathrm{e}(W_i^{k-\alpha}) \geq t(C_j^k) + \Delta_\mathrm{recv}^- - t(C_i^{k-\alpha}) - \Delta_\mathrm{send}^+ - \Delta_\mathrm{msg}^+ - \Delta_\mathrm{mem}^+ \\
&= \underbrace{t(C_j^k) - t(C_i^{k-\pi})}_{\geq 0 \text{ by Assumption 1}} + \underbrace{t(C_i^{k-\pi}) - t(C_i^{k-\alpha})}_{\geq (\alpha-\pi)T^- \text{ by Assumption 2}} + \Delta_\mathrm{recv}^- - \Delta_\mathrm{send}^+ - \Delta_\mathrm{msg}^+ - \Delta_\mathrm{mem}^+ \\
&\geq \left(\pi + \left\lceil \frac{\Delta_\mathrm{send}^+ + \Delta_\mathrm{msg}^+ + \Delta_\mathrm{mem}^+ - \Delta_\mathrm{recv}^-}{T^-} \right\rceil - \pi \right) T^- + \Delta_\mathrm{recv}^- - \Delta_\mathrm{send}^+ - \Delta_\mathrm{msg}^+ - \Delta_\mathrm{mem}^+ \\
&\geq T^- \frac{\Delta_\mathrm{send}^+ + \Delta_\mathrm{msg}^+ + \Delta_\mathrm{mem}^+ - \Delta_\mathrm{recv}^-}{T^-} + + \Delta_\mathrm{recv}^- - \Delta_\mathrm{send}^+ - \Delta_\mathrm{msg}^+ - \Delta_\mathrm{mem}^+ = 0. \ \square
\end{aligned}$$

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

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

elements, no element is read before it is written.

4.5 Bounded Buffer Size

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

As in the previous section, we will show the start and end time, respectively, for the read and write operations. To determine the required buffer size, we need the earliest possible start time of all write operations (excluding the first α writes, since they are prefilled).

Lemma 8. $\forall k > 0 : t_\mathrm{s}(W_i^k) \geq t(C_i^k) + \Delta_\mathrm{send}^- + \Delta_\mathrm{msg}^-$

Proof. We already know that $C_i^k \rightsquigarrow D_i^k \rightsquigarrow W_i^k$. Since D_i^k is triggered by the k-th $\langle \mathrm{tick}, k \rangle$ message, we get: $t(D_i^k) = t(C_i^k) + \Delta_\mathrm{send}(i,k)$. Since W_i^k is triggered by the k-th $\langle \mathrm{data}, l \rangle$ message, we get:

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

In addition to the start of the write operations, we need the latest possible end time of the read operations.

Lemma 9. $\forall k > 0 : t_\mathrm{e}(R_j^k) \leq t(C_j^k) + \Delta_\mathrm{recv}^+ + \Delta_\mathrm{rd}^+$

Proof. We already know that $C_j^k \rightsquigarrow R_j^k$. Since R_j^k is triggered by the k-th $\langle \mathrm{tick}, k \rangle$ message, we get: $t_\mathrm{s}(R_j^k) = t(C_j^k) + \Delta_\mathrm{recv}(j,k) \leq t(C_j^k) + \Delta_\mathrm{recv}^+$. Knowing that a read operation finishes within $\Delta_\mathrm{rd}(j,k)$, we get:

$$t_\mathrm{e}(R_j^k) = t_\mathrm{s}(R_j^k) + \Delta_\mathrm{rd}(j,k) \leq t_\mathrm{s}(R_j^k) + \Delta_\mathrm{rd}^+ \leq t(C_j^k) + \Delta_\mathrm{recv}^+ + \Delta_\mathrm{rd}^+. \ \square$$

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

Lemma 10. *There are always less or equal than $\pi+\alpha+\beta$ unread elements in the buffer (i.e., $\forall k \geq 0 : t_\mathrm{s}(W_i^{k+\pi+\beta}) - t_\mathrm{e}(R_j^k) \geq 0$) with $\beta = \left\lceil \frac{\Delta_\mathrm{recv}^+ + \Delta_\mathrm{rd}^+ - \Delta_\mathrm{send}^- - \Delta_\mathrm{msg}^-}{T^-} \right\rceil$.*

Proof. We have to distinguish two cases:

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

$$\begin{aligned} &t_\mathrm{s}(W_i^{k+\pi+\beta}) - t_\mathrm{e}(R_j^k) \geq t(C_i^{k+\pi+\beta}) + \Delta_\mathrm{send}^- + \Delta_\mathrm{msg}^- - t(C_j^k) - \Delta_\mathrm{recv}^+ - \Delta_\mathrm{rd}^+ \\ &= \underbrace{t(C_i^{k+\pi+\beta}) - t(C_j^{k+\beta})}_{\geq 0 \text{ by Assumption 1}} + \underbrace{t(C_j^{k+\beta}) - t(C_j^k)}_{\geq \beta T^- \text{ by Assumption 2}} + \Delta_\mathrm{send}^- + \Delta_\mathrm{msg}^- - \Delta_\mathrm{recv}^+ - \Delta_\mathrm{rd}^+ \\ &\geq \left\lceil \frac{\Delta_\mathrm{recv}^+ + \Delta_\mathrm{rd}^+ - \Delta_\mathrm{send}^- - \Delta_\mathrm{msg}^-}{T^-} \right\rceil T^- + \Delta_\mathrm{send}^- + \Delta_\mathrm{msg}^- - \Delta_\mathrm{recv}^+ - \Delta_\mathrm{rd}^+ \\ &\geq \Delta_\mathrm{recv}^+ + \Delta_\mathrm{rd}^+ - \Delta_\mathrm{send}^- - \Delta_\mathrm{msg}^- + \Delta_\mathrm{send}^- + \Delta_\mathrm{msg}^- - \Delta_\mathrm{recv}^+ - \Delta_\mathrm{rd}^+ = 0. \ \square \end{aligned}$$

Theorem 1. *For $\alpha = \pi + \left\lceil \frac{\Delta_\mathrm{send}^+ + \Delta_\mathrm{msg}^+ + \Delta_\mathrm{mem}^+ - \Delta_\mathrm{recv}^-}{T^-} \right\rceil$, a sufficient FIFO buffer size is given by*

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

5 VLSI Implementation

The layout of a single FU, consisting of application logic, communication subsystem, and (DARTS) clocking subsystem, is shown in Fig. 3. As usual with GALS systems, we implemented communication subsystem and application logic according to the classic synchronous paradigm, using standard development tools.

The **transmitter** operates as a synchronous peripheral slave device of a controller implemented in the FU's application logic. Data to be transmitted is passed via an 8-bit register interface. In case no data is available from the controller, the transmitter is responsible for inserting idle patterns. In any case, the data is serialized and line coding is applied. To guarantee small skew between clock and data line, a matched (parallel) routing of the PCB traces for data and clock is employed. (Similar constraints are found in all modern high speed source synchronous systems like DDR memory). To relax the timing margins as far as possible, we introduce a 180° phase shift between data and clock.

The **memory element** is integrated into the receiver node. Its read port is controlled by the receiver clock, whereas the write port is under the control of the sender node. Recall that this buffer memory effectively compensates the (bounded) clock skew between sender and receiver clock, and thus ensures metastability-free data communication. As shown in Sect. 4, this can indeed be accomplished with a sufficiently large buffer size. The memory element itself is implemented as a ring buffer with individual address pointers for input and output. Since the only potential for metastability, namely a simultaneous access to the same address [18] is ruled out by construction, it can be written to and read from independently.

The **receiver** operates as a synchronous peripheral slave device for a controller unit implemented in the receiving FU. It takes the data out of the buffer using its own (i.e. its controller's) local clock. After de-serializing and decoding the data, it supplies them to its controller via a memory-mapped 8-bit interface.

5.1 Mapping the Model to the Implementation

The correctness of the algorithms presented and proved correct in Sect. 3 and 4 depends on Assumptions 1–4, which must be guaranteed by the underlying system. In our case, bounded precision and accuracy are inherently guaranteed by our DARTS clocking scheme, and the synchronous start is ensured by a system-wide reset (including DARTS). Finally, Assumption 4 is a timing condition known from classic synchronous systems. It bounds the uncertainty of

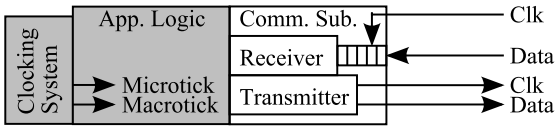

Fig. 3. Layout of a functional unit

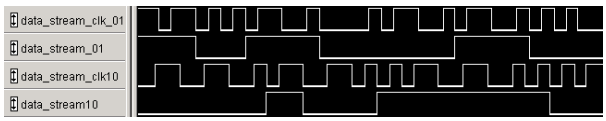

Fig. 4. Example communication within the test system

the involved delays and relates it to the clock period. Since delay uncertainties are caused by manufacturing variabilities and varying operation conditions only, guaranteeing Assumption 4 is easily accomplished by means of standard timing analysis tools.

The actions/events and messages introduced in the algorithmic model in Sect. 4 are mapped to the implementation as follows: The messages $\langle \text{tick}, k \rangle$ are just clock ticks. The delays $\Delta_{\text{send}}(i,k)$ and $\Delta_{\text{recv}}(j,k)$ are the times needed for the ticks to reach the sender output element and the read port of the memory element, respectively. The message $\langle \text{data}\ ,l \rangle$ is implemented as the data and clock transition running on the serial communication link between the sender and receiver. The corresponding delay $\Delta_{\text{msg}}(i,k)$ is the sum of the involved output-, line- and input-delays and the memory setup time. Obviously the simple signal lines that convey all these messages respect FIFO order. The duration of the write operation $\Delta_{\text{mem}}(i,k)$ is the time needed for the new data to reach the output element of the memory (including all combinational logic on its path) after the clock edge and the setup time needed for the output element. $\Delta_{\text{rd}}(j,k)$ is mapped to the hold time of the memory output element.

5.2 Performance and Efficiency

A typical operation example of our communication subsystem is shown in the logic analyzer trace in Fig. 4. It was generated by means of our test system (see Sect. 6.1) using the random clock emulation mode with a precision of 4. The clock traces reveal that our approach also works under very unfavorable clocking conditions: The highly variable phase relation between the clocks would definitely upset any traditional (phase-)synchronous system, and would also violate the feasible operating conditions of synchronizer-based solutions like [10, 11].

Our implementation achieves a (gross) data rate of 1 Mbps/MHz, since one data bit is transferred with every active clock edge. This leads to a data rate of 24 Mbps for our test system clock of 24 MHz, which uses a single data rail. Multiplying this throughput by using parallel data lines is straightforward. For a system with a clock frequency of 100 MHz, for example, using 16 parallel data lines for one clock line would result in a data rate of 1.6 Gbps.

6 Experiments

We now verify the claim that the buffer size derived in Sect. 4 (a) is sufficient for fault-free and metastability-free operation with clocks showing a precision of several clock cycles, and (b) gives a reasonably tight lower bound.

6.1 Test System

To evaluate our communication scheme under controllable and possibly very unfavorable conditions, we have implemented a test system. It consists of 3 Xilinx Virtex-4 FPGAs and a host PC. One of the FPGAs acts as a global test controller. The other two FPGAs host target FUs that exchange messages. They are randomly generated at the host PC and downloaded to both target FPGAs, such that the receiver of a message can check its correctness. If an error is detected, communication stops until the test controller has re-initialized the test system.

6.2 Clock Emulation

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

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

Worst case precision clock emulation: In this mode, the two clock signals are deliberately kept as far apart as the precision allows. This way we can check whether the system can indeed operate under unfavorable conditions. At the same time, chances are high that the communication will fail if the buffer size is too small, which gives an indication of whether the size calculated in Sect. 4 is a tight lower bound. To actually generate such worst case clock scenarios, we artificially stop one clock while the other one runs at its full frequency. As soon as the precision limit is reached, the stopped clock is speeded up to its full frequency again.

Random clock emulation: This mode is used to assess the performance of the system under continuously changing relative clock speeds. An example was already shown in Fig. 4. Clock frequencies are varied over time here, by utilizing a set of "clock primitives" (each one with a different frequency) that can be used by the host PC when constructing an emulation schedule: Clock primitives are randomly assembled into two different sequences that represent the clock traces of the two target FUs. Of course, care is taken to keep the emulated clocks within the precision limits.

6.3 Test Conditions

For the experiments reported here, we used a clocking system with a precision of 4. The emulation base clock (emulation resolution) was set to 48 MHz. This leads to a mean clock frequency of 24 MHz for the random emulation mode, and a clock frequency of 24 MHz for the worst-case emulation mode. The test system

was implemented in a way that minimizes the required buffer size according to Theorem 1: The last term is zero, while the middle term evaluates to 1. This leads to a required buffer size of $2\pi + 1 = 9$.

6.4 Performed Tests and Test Results

Recall that the purpose of our experiments is to verify whether the calculated buffer size is (a) sufficient and (b) minimal. Since it is of course not possible to exhaustively emulate all possible clock relations, we can not *prove* the absence of any buffer overruns for a buffer size 9. We can, however, check the failure-free operation under adverse conditions during some period in order to increase the confidence in our modeling. Furthermore, by reducing the buffer size below the calculated limit, we can easily test the hypothesis (b).

The actual test runs were executed 5000 times for each buffer size and each clock emulation type. For each run, we calculated new clock traces for the emulation. If no failure has been encountered after 2 seconds of observation time, the run was considered fault-free. The resulting histogram, showing the error percentage, has been calculated after all the test runs have been completed.

Figure 5 presents the collected test results. First of all, the random emulation mode did not produce any error in case of a buffer size of eight, whereas the worst case emulation mode did. This indicates that the typical failure probability is actually very low, and becomes visible within limited observation time only if worst case conditions are artificially established. For smaller buffer sizes, the expected failures could be observed frequently. Therefore, our results give a good confirmation of our hypotheses (a) and (b) and, hence, of our analytical results.

7 Conclusion and Future Work

Multi-synchronous clocking is an attractive alternative to globally synchronous clocking in modern high-speed VLSI circuits, such as complex SoCs, which also allows to avoid the single point of failure usually represented by a central clock source. In this paper, we have shown how to employ the loose synchrony provided by multi-synchronous GALS system for implementing a high-speed pipelined communication scheme that is metastability-free by construction. It employs a bounded-size FIFO buffer for compensating the skew between the sender and

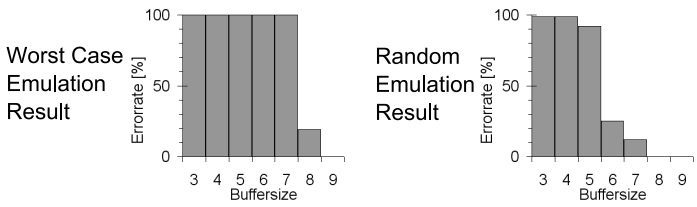

Fig. 5. Results of the experiments

receiver clock. We derived a reasonably tight lower bound for the required buffer size, and provided a formal proof of correctness and freedom of metastability. Furthermore, we have described an efficient implementation of our communication scheme, and experimentally demonstrated its feasibility using a custom test system. Part of our future work will be devoted to extensions of our approach, in particular, flow control and timing error detection.

Acknowledgments: Our thanks go to Matthias Fuegger for guiding us through the formal proofs and to Ulrich Schmid whose ideas initiated this work.

References

1. Metra, C., Francescantonio, S.D., Mak, T., Ricco, B.: Implications of clock distribution faults and issues with screening them during manufacturing testing. IEEE Transactions on Computers **53**(5) (May 2004) 531–546
2. Chapiro, D.M.: Globally-Asynchronous Locally-Syn-chronous Systems. PhD thesis, Stanford Univ. (October 1984)
3. Constantinescu, C.: Trends and challenges in VLSI circuit reliability. IEEE Micro **23**(4) (July 2003) 14–19
4. Borkar, S.: Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. IEEE Micro **25**(6) (November 2005)
5. Mavis, D., Eaton, P.: SEU and SET modeling and mitigation in deep submicron technologies. In: Proceedings 45th Annual IEEE International Reliability physics symposium. (April 2007) 293–305
6. Nicolaidis, M.: Design for soft error mitigation. IEEE Transactions on Device and Materials Reliability **5**(3) (Sept. 2005) 405–418
7. Fairbanks, S.: Method and apparatus for a distributed clock generator (2004)
8. Maza, M., Aranda, M.: Interconnected rings and oscillators as gigahertz clock distribution nets. In: Proceedings of the 13th ACM symposium on VLSI. (2003)
9. Kleeman, L., Cantoni, A.: Metastable behavior in digital systems. IEEE Design & Test of Computers (December 1987) 4–19
10. Ginosar, R., Kol, R.: Adaptive synchronization. In: Computer Design: VLSI in Computers and Processors, 1998. ICCD '98. (1998)
11. Panades, I.M., Greiner, A.: Bi-synchronous FIFO for Synchronous Circuit Communication Well Suited for Network-on-Chip in GALS Architectures. In: Proc. 1st Int. Symp. on Networks-on-Chip (NOCS07), IEEE CS Press (2007)
12. Dobkin, R., Ginosar, R., Sotiriou, C.: Data synchronization issues in gals socs. In: Asynchronous Circuits and Systems (ASYNC'04). (2004)
13. Fuegger, M., Schmid, U., Fuchs, G., Kempf, G.: Fault-Tolerant Distributed Clock Generation in VLSI Systems-on-Chip. In: Proc. of the 6th European Dependable Computing Conference (EDCC-6), IEEE Comp. Soc. Press (October 2006)
14. Teehan, P., Greenstreet, M., Lemieux, G.: A survey and taxonomy of gals design styles. Design & Test of Computers, IEEE (2007)
15. Semiat, Y., Ginosar, R.: Timing measurements of synchronization circuits. In: Asynchronous Circuits and Systems, 2003. Proceedings. 9th Int. Symp. on. (2003)
16. Widder, J., Schmid, U.: Achieving synchrony without clocks. Research Report 49/2005, Technische Universität Wien, Institut für Technische Informatik (2005)
17. Ferringer, M., Fuchs, G., Steininger, A., Kempf, G.: VLSI Implementation of a Fault-Tolerant Distributed Clock Generation. IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT2006) (October 2006) 563–571
18. Lamport, L.: Arbitration-free synchronization. Distrib. Comput. **16**(2-3) (2003)