Timing Domain Crossing using Muller Pipelines

Florian Huemer and Andreas Steininger
Institute for Computer Engineering, TU Wien, Vienna, Austria
{fhuemер,steininger}@ecs.tuwien.ac.at

Abstract—The increasing complexity and modularity of contemporary systems, paired with increasing parameter variabilities, makes the availability of flexible and robust, yet efficient, module-level interconnect instrumental. In this context Globally Asynchronous Locally Synchronous (GALS) systems offer a viable alternative to classical strictly synchronous interconnect. However, with this concept a need for reliable interfaces between different clock and timing domains arises.

In this paper we propose a novel approach for timing domain crossing interfaces which are able to safely transfer (read) tokens (i.e., zero-bit messages) from a synchronous into an asynchronous timing domain and vice versa. At its core the new approach uses a Muller pipeline where read tokens are synchronously written to and read from. Doing this in a way that does not lose any tokens, requires the synchronous timing domain to have exact state information about the fill-level of the Muller pipeline (i.e., is it safe to perform a write or read operation). We thoroughly analyze how this state information can be obtained and propose a simple pipeline sampling model. Furthermore we discuss what timing constraints these concepts rely on and how to deal with metastability that inevitably appears at such interfaces. One possible use of the proposed structures is the control path of metastability that inevitably appears at such interfaces. These concepts rely on and how to deal with metastability that inevitably appears at such interfaces. One possible use of the proposed structures is the control path of metastability that inevitably appears at such interfaces.

Index Terms—FIFO, GALS, timing domain crossing, clock domain crossing, asynchronous

I. INTRODUCTION

Data transfer between storage elements needs to be coordinated. The most common approach for that is the synchronous paradigm, according to which the data transfer is aligned with a global clock whose period is chosen large enough to accommodate for all involved delays. The asynchronous paradigm, in contrast, employs a handshake between sender and receiver. The sender confirms the validity of data by means of a request signal \textit{req}, and the receiver acknowledges the receipt via the \textit{ack} signal\footnote{To keep the explanation simple we restrict it to push channels.}. The concurrent travel of data and \textit{req} causes a race condition. In the bundled data (BD) approach this race condition is solved by the introduction of appropriate delay elements in the \textit{req} path. In the absence of the time grid – as provided by the clock in the synchronous case – asynchronous communication is structured into tokens. The abstraction is that upon the completion of a full handshake cycle a data token is moved from sender to receiver.

Each of these paradigms works very well in a closed domain. However, modern System-on-Chip designs comprise multiple clock domains and/or a mixture of synchronous and asynchronous styles, i.e., multiple timing domains. Hence by the nature of this design approach, data must be able to cross the boundaries of these domains, such that different parts of a system can communicate with each other. The interfaces required for such clock and timing domain crossings require specific care in their conception and design, as metastability becomes a relevant issue there, and they also tend to constitute significant performance bottlenecks. Often FIFO memories are used to sustain throughput in spite of synchronization delays and to facilitate efficient data transfer even in spite of fluctuations in the relative transfer speeds of sender and receiver.

In this paper we present rationale, design and analysis of two building blocks that are fundamental for timing domain crossing: a synchronous/asynchronous interface that allows control data transfer from a synchronous domain into an asynchronous one, and an asynchronous/synchronous interface for the reverse direction. Figure \ref{fig:overview} shows an overview of the respective interfaces considered in this paper. Besides the actual data lines, the synchronous write interface consists of the two control-flow signals. The \textit{write} signal must be asserted (for one clock cycle) to push one data item into the asynchronous domain. The \textit{full} signal indicates that it is not able to process more data items, hence if \textit{full} is active, \textit{write} must not be asserted. Similarly the \textit{empty} signal of the synchronous read interface indicates whether data can be read from the interface. Hence the \textit{read} signal must only be activated if \textit{empty} is deasserted. For each cycle in which \textit{read} is asserted the interface outputs one data item in the next clock cycle. The \textit{full} signal basically implements a back-pressure mechanism, while the \textit{empty} signal constitutes a synchronous request signal. Generating these two flow-control signals typically represents the most challenging function in a timing domain crossing. Our key idea is to employ a Muller pipeline (MPL)\footnote{This research was partially supported by the project ENROL (grant I 3485-N31) of the Austrian Science Fund (FWF).} – a fundamental asynchronous building block that basically implements an asynchronous FIFO for transitions – for transferring the information relevant for flow-control across the domain border.

The asynchronous interfaces are basically BD channels (we only consider 2-phase push channels in this paper) where flow-control is given by the asynchronous handshaking protocol. For the remainder of this paper we refer to FIFO architectures that implement synchronous to asynchronous interfaces (and vice versa) as A/S FIFOs.

The paper is structured as follows: Section \ref{sec:related_work} briefly reviews
some existing work on the topic. Next, Section II revisits the operation of a MPL to finally reflect on how to obtain its fill level through sampling its internal states. Following an outline of our basic approach given in Section VI, the key components of our solution, namely the Desynchronizer and the Resynchronizer, are elaborated in Sections VII and VIII, respectively, along with an analysis on timing and metastability issues. An experimental evaluation of the design is given in Section IX before Section X concludes the paper.

II. RELATED WORK

Later on in this paper we will combine our modules to form a bisynchronous FIFO. While this serves as a compact illustration of our approach, it is not its primary application. There are many other bisynchronous FIFO applications around [2], and we do not claim that ours is better than those, albeit it may be beneficial in specific settings. We present the related work about bisynchronous FIFOs here, since they face similar problems as we have for our A/S FIFOs, namely synchronizing information about the read and write pointers for the FIFO memory across clock domains, such that the flow-control signals (full and empty) can be generated. If a parallel synchronizer is used to directly move these pointers, a special encoding is required such that the respective other side receives a consistent image of the actual value of the pointer. For this purpose Gray code pointers [3] or one-hot (unary) pointers [4] can be employed. Another quite elegant way is the even/odd synchronizer [5]. Here a phase prediction between the read and write clocks is used to avoid long synchronizer chains, which of course contribute to the overall latency of the FIFO.

In the approach proposed by Keller et al. [6] the FIFO uses pausible clocks and mutexes to only transmit pointer increment events to the other clock domain, i.e., the actual value of the read/write pointers is never transferred explicitly. This approach is actually somewhat related to the techniques we present in this paper, because we also only transfer pointer increment information across the timing domain boundaries – in our approach in the form of read tokens. A token-based approach is presented in [7]. Our approach differs from that in not requiring mutex elements, and not assuming a ripple FIFO for the data path.

As a matter of fact pausible clocks make the problem of timing domain crossing a little easier because these circuits are by their nature much more related to asynchronous circuits, in that metastability can in principle be converted to additional wait times through the use of mutexes (time-safe rather than value-safe). This property is for example utilized in the A/S FIFOs proposed in [8] and [9].

In contrast to that the A/S FIFO proposed in [10] uses Gray-encoded read and write pointers. The FIFO architecture presented in [11] can be used for bisynchronous and A/S FIFOs. However, this design relies on special, custom-design FIFO cells, that are selected for read and write operation through a token ring based structure.

III. MULLER PIPELINES

This section first gives some background information on MPLs and then analyzes how (consistent) state information can be extracted from them in a synchronous timing domain.

A. Background

An MPL, such as the one shown in Figure 2a, is composed of C gates. Its purpose is to store and transport handshakes from its input (reqi, acki) to its output port (reqo, acko), where each port consists of a request and acknowledgment signal. Handshakes are sequences of request and acknowledgment transitions and can be classified as 2- and 4-phase. A 4-phase handshake is defined by the sequence req↑, ack↑, req↓, ack↓, where the arrow symbols denote rising and falling transitions, respectively. Note that a 4-phase handshake always starts (and ends) with the same logic value on the handshake signals and involves exactly four transitions. In contrast to that, 2-phase handshakes only use two transitions and leave the handshake signals in the opposite logic state. This means that depending on the initial state of the handshake signals, a 2-phase handshake can either be req↑,ack↑ or req↓,ack↓. The MPL is in principle completely agnostic to the handshaking protocol. It is just a matter of interpretation, whether the stored transitions constitute 2-phase or 4-phase handshakes. Input handshakes can also be viewed as tokens that are fed into and travel through (or get stored in) the pipeline. In Figure 2a these tokens travel from top to bottom. In the following we consider only 2-phase handshakes, as these are also used for the circuits proposed in the next sections. Furthermore we will assume that the environment of the MPL will operate the input and output port strictly coherent with the protocol.

B. Pipeline States

The state of an n-stage MPL is defined by the n+2-element vector $x = x_0, ..., x_{n+1}$. The entries $x_1, ..., x_n$ hold the (output) states of all C gates, while input variables $x_0$ and $x_{n+1}$ hold the logic value of the request at the input port (reqi) and the acknowledgment at the output port (acko). The pipeline outputs $ack_o$ and $req_o$ are equal to the nodes $x_1$ and $x_n$, respectively.

MPLs operate according to a simple rule: If the successor and predecessor of some node $x_i$ ($1 \leq i \leq n$) differ in their logic values, $x_i$ (eventually) takes on the value of its predecessor.

2The Muller C-element, or short C gate, is a fundamental gate in asynchronous logic. Its function is to output the logic level seen at its inputs when these match, and to retain the last valid output state otherwise. It can hence also be viewed as an AND gate with hysteresis.
Note that the nodes $x_0$ and $x_{n+1}$ are inputs to the pipeline, thus their values are controlled by the environment. We define an MPL to be in a steady state, when its defining state vector $x$ does not allow for changes to $x_1,...,x_n$ when applying the pipeline operation rule formulated above. Hence the only way to get a pipeline out of a steady state is by toggling one of its inputs ($x_0$ or $x_{n+1}$). A pipeline that is not in a steady state is transitioning. In the following we will define and analyze what it takes for a pipeline in a steady state to be regarded as full or empty.

An $n$-stage pipeline is regarded empty iff all elements of its state vector $x$ have the same logic value, i.e., $x_i = x_{i+1}$. Note that this also includes the inputs: A mismatch between $ack_i$ ($x_{n+1}$) and $req_o$ ($x_n$) indicates that there is still one transition in the pipeline that has not been acknowledged. A mismatch on the input side (i.e., $x_0 = \neg x_1$) indicates a new request that causes a transition to propagate towards the output. More formally, this violates the stability condition from above for $x_1$ and hence our presumption of a stable state. The only possibility to get an empty pipeline out of a steady state is to externally toggle the $req_i$ ($x_0$) input. Without any read being performed at the output, $n$ complete handshakes can be performed on the input side of an empty $n$-stage pipeline. After that the input side can once more toggle the input request ($x_0$), but won’t get an acknowledgment unless the output $x_{n+1}$ is toggled.

An $n$-stage pipeline is full iff there is a strictly alternating 0/1 pattern on the state vector $x$, i.e., $x_i = \neg x_{i+1}$. Again the interfaces are included here: Unless there is an active input request (i.e., $x_0 = \neg x_1$), there is still one “free” position in the pipeline. Likewise, $x_n = x_{n+1}$ would indicate an acknowledged output request, which makes all transitions contained in the pipeline move by one position. This is a dynamic state, as witnessed by the violation of the stability condition for $x_n$. Assuming no further write access, on a full $n$-stage MPL $n+1$-2-phase read handshakes can be performed before the pipeline becomes empty.

C. Obtaining State Information

Since an MPL is an event driven circuit, care must be taken when extracting (synchronous) state information from it. Figure 2(b) shows how flip-flops can be used to sample the pipeline state vector $x$, effectively transferring its information into the synchronous timing domain. To model the state of the pipeline nodes at the synchronous clock events (i.e., when they are sampled), we use the set $\mathbb{B}_T = \{0, 1, \perp\}$. Consequently $x$ is described as a vector of length $n+2$ over $\mathbb{B}_T$. The values 0 and 1 indicate the normal logic states for the nodes $x_i$ that were stable during the setup/hold (S/H) window of the associated sampling flip-flop. The value $\perp$ indicates that the respective node was transitioning from 0 to 1 (1 to 0) during the S/H window. If a flip-flop samples a transition, it may become metastable. Hence we use a different set of values $\mathbb{B}_M = \{0, 1, M\}$ to model the outputs $x'$ of the sampling flip-flops. The values 0 and 1 again have the usual meaning of logic low and high. The value $M$ denotes a metastable state, and can basically be considered as a wild card for arbitrary behavior of a signal, including late transitions or glitches. This meaning is consistent with the notion of metastability used in metastability containing circuits [12]. Using $\mathbb{B}_M$, Boolean equations can be evaluated normally with slightly extended truth tables for the logic operators.

The flip-flops shown essentially perform the task of a synchronizer, in that they (try to) map the asynchronous signal at their inputs to a stable output value; hence they must be appropriately extended in practice (see later). The function $s$ (Equation (1)) models this operation. It maps a value in $\mathbb{B}_T$ to a set of possible values in $\mathbb{B}_M$, to which the synchronizer might resolve (or fails to resolve, in the case of $M$). Note that sampling a transition can basically cause an arbitrary output value of the synchronizer. The synchronizer can either settle on the old or the new value of the input (0 or 1), or become metastable itself ($M$). Which of the values from $\mathbb{B}_M$ the output actually assumes cannot be predicted for a single instance.

$$s : \mathbb{B}_T \mapsto \mathcal{P}(\mathbb{B}_M), \quad s(x) = \begin{cases} \{0, 1, M\} & \text{if } x \in \{1, \perp\} \\ \{x\} & \text{otherwise} \end{cases} \quad (1)$$

Similarly $s_v$ yields the set of possible values for the a whole vector of length $m$.

$$s_v : \mathbb{B}_T^m \mapsto \mathcal{P}(\mathbb{B}_M^m), \quad s_v(x) = \{y \in \mathbb{B}_M^m | y_i \in s(x_i)\} \quad (2)$$

Ultimately, our goal when evaluating the pipeline state is to retrieve the number $T$ of tokens present. For the stable pipeline we have argued that $T$ equals the number of state changes between consecutive elements along the state vector. So we need to be able to find all instances of 01 or 10 in $x'$ Centering that the propagation delays $\Delta_i^S$ from the individual pipeline nodes to the respective sampling flip-flops (see Figure 2(b)) may vary, one might experience an inconsistent result when taking a sample while several transitions are “in flight”. To safely prevent that, we require that all $\Delta_i^S$ ($0 \leq i \leq n+1$) are equal, i.e., there is no (or negligible) skew between the wires connecting the nodes $x_i$ to the flip-flops.3

3M = M, 0M = 0, 1M = M, M\wedge M = M, all other operations can be reduced to the given rules using De Morgan’s laws.

4In Equations (1) and (2) $\mathcal{P}(A)$ denotes the power set of set $A$. 

---

(a) 4-stage Muller pipeline 
(b) State sampling circuity

Fig. 2: Muller pipeline circuits
In the general case, where no timing constraints are imposed on the MPL, every possible vector in \( \mathbb{B}_T^{v_T} \) can appear at the input of the sampling flip-flops. Under our sampling model this would make it impossible to infer reliable state information about the pipeline, let alone obtaining an exact token count. Consider, for example the state vector \( \uparrow \downarrow \uparrow \downarrow \), representing a single token traveling through an initially empty pipeline. Sampling this vector (and assuming metastability is resolved) may result in 000, 010, 100 or 110 for \( x' \), which obviously correspond to vastly different pipeline states and token counts.

To avoid such problems, we constrain the forward and backward delay of the pipeline stages. Let \( T_S \) and \( T_H \) denote the setup and hold time of the synchronizer/sampling flip-flops as specified in their datasheet. Moreover, \( \Delta_F \) and \( \Delta_B \) denote the forward and backward delay of stage \( i \) in the MPL. For the sake of simplicity we assume equal delays for rising and falling transitions (namely the shorter of both). As indicated in Figure 2, we always measure the delays \( \Delta_F \) and \( \Delta_B \), from and to the nodes \( x_i \). They hence incorporate the delay of the C gate as well as the interconnect and the inverter.

\[
\forall i, 0 \leq i \leq n, \Delta_F^i > T_S + T_H \wedge \Delta_B^i > T_S + T_H \quad (3)
\]

The constraint in Equation (3) guarantees that it can never be the case that two neighboring flip-flops in the synchronizer sample transitions at the same time. This in turn means that neighboring flip-flops cannot both evaluate to random values (or become metastable) at the same time, i.e., if a node \( x_i \) experiences a transition, its neighbors \( x_{i-1} \) and \( x_{i+1} \) must be stable. Moreover, because of the pipeline operation rule, these two stable nodes must have opposite values, i.e., \( x_i = \uparrow \Rightarrow x_{i-1} = 0 \wedge x_{i+1} = 1 \) and \( x_i = \downarrow \Rightarrow x_{i-1} = 1 \wedge x_{i+1} = 0 \) for \( 1 \leq i \leq n \). For the pipeline inputs node \( x_0 \) and \( x_{n+1} \) (which obviously don’t have a predecessor or successor node, respectively) these rules can be simplified to \( x_0 = \uparrow \Rightarrow x_1 = 1 \), \( x_0 = \downarrow \Rightarrow x_1 = 0 \), \( x_{n+1} = \uparrow \Rightarrow x_n = 0 \) and \( x_{n+1} = \downarrow \Rightarrow x_n = 1 \). Figure 3 illustrates this behavior. Note that any variance of \( \Delta_F^i \) between different nodes \( x_i \), basically “enlarges” the S/H window and must be added on the right side of the constraint inequality. Clock skew between the sampling flip-flops has the same effect.

In summary, if we assume that metastability is resolved, every token, i.e., every 01 or 10 transition in the state vector \( x' \) can be identified reliably. For the token detection it does not matter to which value metastable flip-flops resolve, since the total number of state changes stays the same in both cases (recall that transitions must be enframed by two opposite stable values). The only exception to this are transitions on the pipeline inputs \( x_0 \) and \( x_{n+1} \) (e.g. \( 100 \) can be sampled as \( 000 \)).

Now let’s reason about the possible states an \( n \)-stage pipeline can have under our sampling model. Let \( X^m \subset \mathbb{B}_T^{v_T} \), \( m = n + 2 \) denote the set of possible values the vector \( x \) can take under the timing constraints formulated above. Starting with the set \( \mathbb{B}_T^{v_T} \) we can simply remove all states that violate one of the implications formulated above to hence obtain \( X^m \).

To construct \( X^m \), for some given \( m \), in a recursive way the following approach can be used: Given \( X^{m-1} \) the set \( X^m \) is generated by appending a new element to the front of the vectors in \( X^{m-1} \) and considering the possible values this new element can take based on the the exclusion rules formulated above. From a circuit view this means the original input request \( x_0 \) is now driven by a newly added C gate whose inverted input is connected to \( x_1 \). The new input request of the extended pipeline is then given by this C gate’s non-inverted input. For all \( x \in X^{m-1} \) where we have \( x_0 = \uparrow \) (i) a new vector \( z \) is added to \( X^m \), where \( z_0 = 1 \) (0) and \( z_i = x_{i-1} \) for \( 1 \leq i \leq n + 1 \). All other symbols for \( z_0 \) are forbidden by the exclusion rules. If \( x \in X^{m-1} \) starts with a constant value, new three vectors are added to \( X^m \). The only value that is forbidden for \( z_0 \) now is \( \uparrow \) if \( x_0 = 1 \) and \( \downarrow \) if \( x_0 = 0 \).

From this recursive construction it is now possible to derive a suitable recurrence relation that yields \( v_m = |X^m| \), i.e., the number of possible state vectors for a pipeline with \( n \) stages (i.e., C gates). Let \( v^C_m \) and \( v^T_m \) denote the number of state vectors (for a given vector length \( m \)) that start with a constant (i.e., 0 or 1) and transitional (i.e., \( \uparrow \) or \( \downarrow \)) value at \( x_0 \), respectively (thus \( v_m = v^C_m + v^T_m \)). Using the recursive construction of \( X^m \), Equation (4) can be established.

\[
v_m = 3v_{m-1} - v^T_{m-1} \quad (4)
\]

From the discussion above it is also clear that \( v^T_{m-1} = v^C_m \) must hold, because for all vectors in \( X^m \) starting with a constant value in \( x_0 \) a single vector starting with either \( \uparrow \) or \( \downarrow \) is added to \( X^{m-1} \). After some simple transformations using the mentioned equalities, we arrive at the recurrence relation shown in Equation (5).

\[
v_m = 2v_{m-1} + v_{m-2} \quad (5)
\]

Now it remains to establish suitable initial values for \( v_1 \) and \( v_0 \). Since \( m = n + 2 \), at first glance \( v_1 \) and \( v_0 \) don’t make much sense in the context of an actual pipeline circuit. However, setting \( v_0 = 0 \) and \( v_1 = 4 \) yields the value 8 for \( v_2 \), which is the expected number of states for a pair of handshake signals (i.e., a “zero-stage” MPL). Similarly, \( v_3 \) yields 20, which again matches the expectation for a pipeline consisting of a single C gate. For that matter \( v_1 = 4 \) can be viewed as the number of states a single wire can have.

IV. PROPOSED APPROACH

In essence, we propose two basic modules, a synchronous/asynchronous interface which we will call Desynchronizer in the following, and an asynchronous/synchronous...
interface called Resynchronizer. These two components form the cornerstones for our A/S FIFO architecture and can, if connected together, also be used to implement a bisynchronous FIFO. We will leverage the knowledge from Section III about the possible pipeline behaviors under the imposed restrictions for those implementations.

A. Problems and Requirements

The fundamental problem with timing domain crossing interfaces is to compensate for the lack of a handshake on the synchronous side, by means of full or empty status flags. The generation of these flags necessarily involves combining status information from both timing domains, which creates the demand for synchronization. A very well known instance of this problem is the comparison of read and write pointers in bisynchronous FIFOs outlined in Section II. While the use of synchronizers can easily solve this, synchronizers, in the general case, do create a delay that makes the validity of the status flags questionable: Knowing that the asynchronous domain was ready to receive data, e.g. two clock cycles ago, does not imply that this is still the case when this information is received (i.e., after the synchronizer delay). This seems to be a very fundamental issue that cannot be eliminated, but rather requires appropriate consideration. In our attempt to provide an efficient approach for the latter, let us pin down the requirements more precisely:

(R1) Validity of the full flag: If the full flag is inactive at any clock cycle, then the asynchronous domain must indeed be able to accept a data item at the next active clock edge.

(R2) Validity of the empty flag: If the empty flag is inactive at any clock cycle, then the asynchronous domain must indeed be able to deliver a valid data item at the next active clock edge.

(R3) Maximum Throughput: It shall be possible to move a data stream over the interface at a sustained rate of one data item per clock cycle.

(R1) is essential to prevent losing data through overflow while (R2) prevents reading stale data from an empty FIFO. While (R1) and (R2) are mandatory for any reasonable timing domain interface, they do not prevent overly conservative activation of the full and empty flags. (R3) is introduced to counterbalance this by also considering performance. Note that our definition of (R3) only applies to the data throughput, it does not restrict the latency. We chose to do so, since, having FIFO applications in mind, the efficient transfer of single data items may not be the primary concern anyway.

B. Key Solution Principle

The core of our proposed FIFO architectures is an MPL. For each new data item, written to the write port of the FIFO, a token is placed into this pipeline. For each of these tokens the read port may perform exactly one read operation on the FIFO, removing the token from the pipeline. In this sense, the MPL is used as an asynchronous up/down counter, representing the fill-level of the FIFO. The tokens themselves don’t carry any information, only their total number stored in the pipeline is relevant. The actual data must be stored in some shared memory, and each port of the FIFO must maintain its own memory pointer to it. Since fundamentally the MPL is an asynchronous structure, reading from and writing to it from the asynchronous domain is natural and straightforward; and with full or empty flags in place on the synchronous sides for establishing flow-control, maintaining a correct token count remains feasible.

Unlike conventional approaches where read and write pointer must be related in some way to obtain information on the fill level, the token count in our up/down counter directly reflects the fill level. Consequently the pointers for the shared memory on the read and write port of the FIFO can be maintained independently and need never be mapped into the respective other timing domain, as no comparison is required – this is a significant advantage of the up/down counting underlying our approach. For every write operation, one token is added to the MPL and the write pointer is incremented, and upon every read one token is removed and the read pointer incremented. Of course, the pipeline depth must be adjusted to the number of locations in the FIFO memory.

Now the remaining challenge is to generate the full and empty flag the synchronous sides rely on to comply with (R1) and (R2). To this end, we have to overcome three problems:

First, reading the fill status of an immanently event-driven MPL is non-trivial. We have already elaborated on that in Section III.

Second, the status must be moved across the timing domain boundary from asynchronous (where the MPL is located) to synchronous (where the flag is needed). In principle, the full or empty flag can be generated on the asynchronous side, in which case only the flag as such needs to cross the timing domain. While this seems conceptually very attractive, we have decided to move the flag generation to the synchronous domain, as this simplifies the implementation as well as eases the reasoning for why the proposed circuits work correctly. Even though, as a consequence, a number of pipeline status signals now need to cross the timing domain, only one of them is actually critical for metastability, as will become clear later on.

Third, as already mentioned, synchronizing an asynchronous event to a fixed clock causes a delay. This delay in obtaining the pipeline state information needs to be appropriately accounted for.

To handle these issues we designed the De- and Resynchronizer circuits. These two components basically evaluate a synchronized state of a certain number of pipeline stages. For the Desynchronizer the sub-pipeline comprises the first stages after the circuitry with which the synchronous side places a read token into the pipeline. The obtained information is then used to generate the full flag for the synchronous domain. Consequently the Resynchronizer must look at the last few stages to generate the empty flag. As a consequence of (R3) the number of pipeline stages that are synchronized must be appropriately matched to the synchronizer delay, which
is usually chosen with a specific mean-time between upset (MTBU) requirement in mind.

Although we consider FIFOs an important application and use one to illustrate our approach, the focus of this paper will be on the two interface blocks, the Desynchronizer and the Resynchronizer. With these components in place, the remaining parts of a FIFO implementation (memory cells and memory pointers) can each operate in a single timing domain and are hence relatively easy to implement. We will give a brief implementation example in Section VII.

V. DESYNCHRONIZER

An overview of the Desynchronizer circuit is shown in Figure 4. It can be seen that the single flip-flops from Figure 2a have been replaced by n-flop synchronizers, where n can be chosen according to the reliability demands of the circuit, i.e., the desired MTBU. This synchronizer array reads the state vector \( x = x_0, ..., x_{n+1} \) of the first \( n \) stages of the MPL used to buffer the read tokens and produces the synchronized state vector \( x' \). We need to sample exactly \( n \) stages to be able to compensate for the synchronizer delay of \( n \) cycles. This will become more clear later on. The bottom-most C gate in the figure is not part of this \( n \)-stage (sub-) pipeline; it is, however, relevant for the pipeline timing constraints. Since this C gate produces the input acknowledgment \( x_{n+1} \) to the \( n \)-stage (sub-) pipeline, it determines \( \Delta_n \) as indicated in the figure. It also represents the first C gate in the subsequent pipeline portion that is used to buffer read tokens. Recall that the total pipeline depth (in terms of token capacity) must be equal to the number of memory locations in the FIFO (which may be significantly larger than \( n \)).

The synchronous write port uses a toggle flip-flop to place \( x \) to the pipeline\(^3\). At every active clock edge where write is high the input request to the pipeline \( (x_0) \) is inverted, hence generating a new token. As it is not possible to directly observe the output acknowledgment \( x_1 \) of the pipeline, because of metastability concerns, the input port must assume that (i) whenever full is low the pipeline can accept a token and that (ii) this write operation completes within one clock cycle. In order for this design to work properly, we have to impose timing constraints (Equation 6) on the minimum clock period \( T_c \) (i.e., the maximum frequency) the synchronous port can be operated with.

\[
T_c > t_{CO} + \sum_{0 \leq i \leq n} \Delta_i^F + \Delta_i^S + T_S \bigwedge T_c > \max_{0 \leq i \leq n} \delta_i^B \quad (6)
\]

Here, \( t_{CO} \) denotes the clock-to-output delay of the toggle flip-flop and \( T_S \) again denotes the setup time of the sample flip-flops. Moreover, \( \delta_i^B \) denotes the delay from node \( x_i \) back to the input of the previous C gate (including the inverter). This is illustrated in the figure for \( \delta_n^B \). The constraint ensures, that given an empty pipeline a token that is generated during a write operation must have left the \( (n\text{-stage}) \) pipeline within one clock cycle (of course assuming that the subsequent logic at the asynchronous side is ready to receive this token). This means that the synchronizer will again sample a constant pattern and the full flag will stay deasserted. This is required to sustain a continuous data stream, as demanded by (R3).

Hence, for every clock cycle in which a token is written to the pipeline (and the pipeline does not start to fill up) the state of all nodes \( x_i \) must toggle and the new C gate states must propagate to the respective sampling flip-flops (first inequality). Furthermore, there must be enough time such that all C gates in the pipeline are again armed for the next (opposite) transition, i.e., the next token (second inequality). If the pipeline starts to fill up, because tokens are not removed fast enough at the asynchronous side, the state vector \( x \) sampled by the synchronizer will be different from the all-zero/all-one vector, which indicate an empty pipeline. In particular the first token that is not able to leave the pipeline within one clock cycle results in a state vector where the value of \( x_n \) and \( x_{n+1} \) will be different.

After \( n \) cycles the sampled status vector \( x \) will have propagated through the synchronizer and appear as \( x' \) at its output. Because of this latency the MPL must be able to absorb another \( n \) read tokens without overflowing after the first token that was not able to leave the pipeline within one clock cycle.

As a consequence full is activated as soon as the pipeline is recognized not being completely empty, i.e., already when it holds a single token. Therefore it is possible that the full signal is asserted in some cycle \( c \) although there were no write accesses for the last \( n-1 \) cycles. Obviously, this is pessimistic and does not fully utilize the pipeline in every case, but it is safe with respect to (R2) and allows for a very simple, fully combinational implementation of the logic to generate the full flag. Moreover, \( n \) will typically be as low as 3, so the sub-pipeline is not very deep.

As shown in Figure 4 the full signal is driven by the inverted output of the constant pattern detector \( \text{full} = \neg \text{cp} \), whose implementation is defined by Equation 7.

\[
\text{cp}(x') = \left( \bigwedge_{0 \leq i \leq n} x_i' \right) \lor \left( \bigwedge_{0 \leq i \leq n+1} \overline{x_i'} \right) \quad (7)
\]

\[^3\]The proposed principle would also work with 4-phase handshakes, however, this would mean among other things that the MPL would need to be twice as long.
Using this approach it is guaranteed that whenever full is asserted it stays asserted until the pipeline is empty, i.e., there are no tokens left in the circuit that could lead to further transitions on x.

Finally, the risk for metastability on the full signal must be evaluated. To this end we leverage the findings from Section III and examine all possible pipeline states given by the set Xn−2.

Note that writes to the MPL are only performed synchronously, thus the flip-flop chain at x0 will never sample a transition and hence cannot become metastable. So for our analysis we don’t have to consider those vectors x ∈ Xn−2 where x0 ∈ {↑, ↓}. This only leaves two cases for x, that when sampled and converted to x’ = s0(x), can lead to an arbitrary value at the full signal (i.e., 0, 1 or M). In particular these are constant patterns with a transition at the last node x_{n-1} (x_{i|n-1} = 1, x_{n-1} = ↑ or x_{i|n-1} = 0, x_{n-1} = ↓). These situations occur when a token is just about to leave the pipeline but didn’t quite make it. Hence, in our application it is irrelevant to which stable value the synchronizer resolves the transition. Either the pipeline appears non-empty for another cycle, which is a safe state since writes are not allowed, or it is considered empty, which is also fine because the token was just about to leave the pipeline anyway. Consequently M is the only critical output, and the overall MTBU of the circuit is hence determined solely by the synchronizer on x_{n-1}. Metastability on all the other nodes is always masked. This indicates that it is beneficial to use flip-flops with good metastability resolving capabilities on this signal to minimize the risk of an upset – the remaining “synchronizer” chains are just needed for timing alignment of the paths. They do not at all contribute to the MTBU, and cheaper implementations may be used. This is in contrast to other schemes, like the Gray encoding, where all paths need to have good metastability resolution, even though only one specific path is relevant at a time (i.e., for a given count value).

VI. RESYNCHRONIZER

An overview of the Resynchronizer circuit is shown in Figure 5. It can be seen that its general structure is closely related to that of the Desynchronizer. However, now the last n stages of the token buffer pipeline are sampled by n-flip synchronizers to transfer the pipeline status across the timing domain boundary and to ultimately deduce the FIFO fill level. Again, we need to sample n stages, in order to compensate for the synchronizer latency. Similar to the Desynchronizer here the top-most C gate is not considered part of this pipeline. It is included in the figure as it determines ΔB. The toggle flip-flop is now used to control the input acknowledgment, i.e., to remove tokens from the pipeline.

For the minimum clock period Tc, a similar constraint applies as for the Desynchronizer (Equation (8)).

\[ T_c > T_{CO} + \sum_{0 \leq i \leq n} \Delta^B + \Delta^S + T_S \land T_c > \max_{0 \leq i \leq n} \delta^F_i \quad (8) \]

The variable \( \delta^F_i \) denotes the delay from the node \( x_i \) to the non-inverting input of the next C gate. This constraint ensures that after a read from a full pipeline the pipeline is again indicated as full for the next clock cycle, if there was already a new token pending on the input port (\( req \neq ack \)). To accomplish this all values \( x_i \) must invert their logic state within one clock cycle, which means that the bubble (free space) created by removing the token through the read must ripple up the whole pipeline from \( x_{n-1} \) to \( x_0 \). In addition, this constraint is more than sufficient to ensure that a read operation completes within one clock cycle.

Since it must be guaranteed that the read signal is only asserted if there is at least one token in the pipeline, care must be taken when generating the empty signal. As can be seen from Figure 5 there are two conditions for the deassertion of empty, either (i) \( ap \) or (ii) \( one \) must be asserted.

Let’s first consider the former condition. Whenever there is an alternating pattern detected on the vector \( x' \), we know that the pipeline was full \( n \) cycles ago. This means that, even if we assume the worst case, namely that \( n \) read operations have been performed in the last \( n \) cycles, there must still be one token left to read in the pipeline. Hence, whenever there is an alternating pattern on \( x' \), empty can be safely deasserted, allowing for a read operation in the next cycle. As long as there is a constant stream of new tokens arriving at the input side of the pipeline (\( req \) and \( ack \)), the \( (n\text{-stage}) \) pipeline will always stay full and the synchronous side can perform continuous reads (requirement (R3)). This behavior is illustrated by the timing diagram in Figure 6 for \( n = 2 \). For the sake of clarity we added \( x_{n-1} \) as an individual signal trace, as it represents the synchronously generated input acknowledgment to the pipeline. The signal \( x_n \) holds the intermediate value of the flip-flop (synchronizer) chain. The pipeline starts out in a full state where \( x = x' = 0101 \), which would allow for three read operations. After the first read another token immediately enters the pipeline in the same clock cycle, which leaves the pipeline in the state \( x = 1010 \) (red state label). Note that if no new token would enter, the state of the pipeline after the read would be 0010. Because of the new token entering the pipeline in total four consecutive read operations are possible.

If the design would just use the \( ap \) signal to generate the empty signal, the Desynchronizer would already work fine for
streaming applications. However, this way the situation may arise that there are tokens left in the pipeline (i.e., data left in the FIFO), that cannot be read because their number is too small to trigger the alternating pattern detector. Consider, for example the case where only a single token arrives. This token would not be detected, and could hence not be read, until further tokens arrive to fill up the pipeline.

To eliminate this issue, we introduced condition (ii), i.e., the one signal. The one signal basically indicates whether there is an active, i.e., unacknowledged, output request \( x_{n+1} \), which means that there is at least one token left in the pipeline. For that matter it can be viewed as a full indicator for the 0-stage sub-pipeline formed by the output request and input acknowledgment signals. Since the state vector \( x \) corresponds to the state of the pipeline \( n \) cycles ago, it is only safe to interpret the output of the XOR gate if there have not been any reads during this time. For this purpose the constant pattern detector checks if the input acknowledgment to the pipeline has changed during the last \( n \) cycles.

Now it only remains to analyze the effects of metastability on the pipeline state information signals one and ap and therefore the empty signal; specifically which scenarios can lead to a violation of (R2) or metastability propagation.

Similar to \( x_0 \) in the Desynchronizer, the flip-flop at \( x_{n+1} \) can never sample a transition and thus can not become metastable. Hence, knowing that \( x_{n+1} \in \{0,1\} \), the remaining condition for the one signal to produce a stable output is that \( x_n \in \{0,1\} \) must hold. Unfortunately, for the case where \( x_n \in \{1,1\} \) we may still get an arbitrary output (i.e., 0, 1 or \( M \)) at \( x'_n \) and consequently at one. Let’s analyze the consequences of that:

- In the case of an \( M \) at \( x'_n \), metastability propagation to the empty output can ultimately cause the circuit to fail. This situation may arise, when a transition is traveling through the MPL, but did not quite make it to the bottom-most C gate. In such a case we must rely on the synchronizer to minimize the probability that this value actually appears at its output.
- If the synchronizer internally resolves metastability at \( x'_n \) to the same logic value as held by \( x_{n+1} \), one stays low, which is a safe outcome because read operations are forbidden.
- If, on the other hand, \( x'_n \) takes the opposite value of \( x_{n+1} \), one will be asserted. However, since the transition of the C gate at \( x_n \), that caused this value occurred \( n \) cycles ago, we can be certain that it has settled in the meantime and a token can successfully be read from the pipeline.

For generating the ap signal, we have the alternating pattern detector in place whose basic function is represented by the left product term in Equation (9) (the symbol \( \oplus \) denotes an exclusive or operation): The pipeline is full when all neighboring nodes have opposite logic states.

\[
ap(x') = \left( \bigwedge_{0 \leq i \leq n} x'_i \oplus x'_{i+1} \right) \wedge \left( \bigwedge_{0 \leq i \leq n-1} x'_i \oplus x'_{i+2} \right)
\]

Unfortunately, this function alone is not sufficiently resilient to metastability at any \( x'_i \). To improve that, we have added the second (right) term in Equation (9) which states that every node must have the same logic state as the nodes next to its direct neighbors. While this term is logically redundant, it allows for metastability masking on all \( x'_i \) for \( 1 \leq i \leq n \) for the following reason: From the discussion about the possible values for \( x \), we know that transitions (i.e., \( \uparrow \) or \( \downarrow \)) at \( x_i \) must always be framed by two opposite stable values (i.e., \( x_{i+1} = \neg x_{i-1} \)). This means that even if one (or more) of the signals \( x'_i \) for \( 1 \leq i \leq n \) evaluate to \( M \) because the synchronizer was not able to resolve metastability, \( M \) would not be able to propagate to the ap signal, since, due to the second term, its stable neighbors will make ap evaluate to a stable 0 in those cases. Hence, regarding the ap signal metastability is only an issue for node \( x_0 \). The overhead for the metastability containing alternating pattern detector can be estimated by a factor of two, when compared to a non-metastability containing version.

An analysis of the set of possible vectors for \( x \) reveals that there are only two cases that can lead to an ambiguous value at ap (i.e., vectors \( x \) where \( \{ap(y) | y \in s_v(x)\} = \{0, 1, M\} \)). These cases are alternating patterns starting with a rising or falling transition on \( x_0 \) instead of 1 or 0, respectively, i.e., \( x_0, x_1 = \uparrow \) 0 and \( x_0, x_1 = \downarrow \) 1, whereas \( x_i = \neg x_{i-1} \) for \( i > 1 \). However, for the same reasoning as presented for the one signal, both of these cases are unproblematic, as long as the synchronizer manages to resolve metastability.

**VII. Results**

This section presents a prototype implementation of the presented circuits to demonstrate the viability and practicality of our approach. Furthermore, we briefly analyze the impact of metastability on the MTBU of the circuits.

**A. Prototype**

For the prototype implementation we combined the Re- and Desynchronizer circuits to implement a bisynchronous FIFO in a Field Programmable Gate Array (FPGA). Figure 7 shows an overview of this design. We used \( n = 3 \) synchronizer flip-flops and a FIFO depth of \( d = 16 \) elements.

The Altera Cyclone IV FPGA we used features logic cells containing a 4-input LUT and a flip-flop. The interconnect of this device is capable of using the LUT’s and flip-flop’s output simultaneously. Hence, the 2-input C gates constituting the MPL, are implemented in a single LUT, using one input for the feedback path and one as a reset input, to
ensure the correct start-up of the circuit. The output of the LUT is directly connected to the flip-flop in the same logic cell, which results in a small and more importantly uniform delay for this connection across the whole pipeline (\(\Delta^S\) in Figure 2b). Besides the explicit placements of the MPL and the sampling flip-flops, no further manual interventions were necessary. Actual values for the S/H times are not available in the datasheet of our target FPGA. We hence experimentally verified that Equation (3) is not violated. The MPL connecting the Re- and Desynchronizer must accommodate for the size of the (dual-clocked) memory, which means that we need \(d - 1\) stages in total. Recall that a full \(n\)-stage MPL contains \(n + 1\) tokens (Section III).

The bisynchronous FIFO is embedded in a hardware test-bench that generates pseudo-random input data using a linear feedback shift register (LFSR) and measures the data throughput. Furthermore, the read and write signals can also be controlled using LFSRs to simulate sporadic access to the FIFO (of course considering the full and empty signals). Both FIFO sides maintain a counter that records the total number of data items that passed through the FIFO. Matching these two counters reveals if data has been lost or falsely generated (requirements (R1) and (R2)). The read side also automatically checks whether the read data matches the expected value.

The read and write clocks were generated using uncorrelated external clock sources. We tested a wide range of different read/write clock speed combinations and operation modes (maximal throughput and sporadic access). These experiments verified that the FIFO indeed works as expected and that the throughput of a data stream is only limited by the slower one of the read and write clocks, fulfilling requirement (R3). We were able to measure maximum operation frequencies of up to 275 MHz for both the read and write port. To put this in number into context: The maximum performance for block RAMs (i.e., M9K blocks) in the used FPGA (speed grade C7) is 274 MHz [13]. This means that our FIFO allows to utilize the full performance of the used memory block and that the timing constraints imposed on the MPL don’t affect the maximum operation frequency in this case. In fact another simple experiment showed that the Re- and Desynchronizer could even handle higher frequencies. This is not unexpected, as an MPL stage can operate very fast, and for our timing constraints (Equations (2) and (8)) only a few stages become effective, namely as many as we have synchronizer stages.

Note that this holds even if the FIFO is much deeper.

The external clock generators also allowed us to test the circuit under the influence of significant clock jitters (in the range of tens of MHz). Unsurprisingly the circuit proved to be very robust against such disturbances, because the only thing that must be ensured is that the minimum clock period \(T_c\) is still sufficiently long.

The example implementation also shows that our design uses fairly standard components and even maps quite nicely to FPGAs, even though the C gate is somewhat handcrafted. However, if a different pipeline style would be chosen, that does not use any special gates (e.g. Mousetrap [14]) we could even get rid of those.

B. MTBU Considerations

The prototype implementation is not intended to analyze the metastability behavior of the presented circuits, because the related properties are easy to assess analytically with the usual approach for \(n\) flip-flop synchronizers [15]. The only parameter that must be determined for that is the actual rate of transitions at the input of the (relevant) flip-flop chains. Clearly, this data rate strongly depends on the actual use case of the circuit and its environment. However, as a pessimistic limit one can use the maximum data (token) rate.

For the Desynchronizer metastability can only occur if the pipeline starts to fill up because tokens cannot be removed (i.e., read) fast enough on the asynchronous side. Hence the data rate for the MTBU estimation is basically the rate with which this happens.

For the Resynchronizer, we have a similar situation. As long as the pipeline is kept full, metastability is not an issue because the synchronizer flip-flops always sample stable input signals. Hence metastability can only occur during the time the pipeline fills up and empties.

So normally one will come along with a very low \(n\) (like e.g. 3), which keeps the latency of the circuit within bounds and also yields a low minimum requirement for the FIFO depth.

VIII. Conclusion

In this paper we have presented a synchronous to asynchronous and an asynchronous to synchronous interface that are both based on the use of an MPL, intended for implementing A/S FIFOs. More specifically we employ the MPL to convey tokens across the timing domain boundary. These tokens provide both, sender and receiver with all necessary information for their local memory pointer management and flow-control. Reducing the information exchange to tokens in an MPL allows for a systematic and efficient metastability handling at the boundary. Although still synchronizers are mandatory to cope with metastability, which is in general inevitable at timing domain boundaries, we have confined the critical components to a single synchronizer path per direction. In addition our proposed concept pipelines data transfers and thus handles the synchronizer latency in the transmission of data streams without negatively impacting the throughput.
resulting in a sustained data rate of one data item per clock cycle.

Moreover, the proposed circuits also allow for a configuration as a bisynchronous FIFO, which we have used to experimentally verify our approach and confirm its robust operation, high throughput and complete flexibility with respect to the frequencies on both synchronous sides (as long as the lower limit for $T_c$ is satisfied).

Potential improvements comprise an optimization of the interface performance for non-blocked data, as well as an implementation based on other pipeline styles that can be realized with standard CMOS library components alone (i.e., without Muller C-elements).

REFERENCES