Robust Interfaces for Mixed-Timing Systems

Tiberiu Chelcea and Steven M. Nowick

***Abstract*—This paper presents several low-latency mixed-timing FIFO (first-in–first-out) interfaces designs that interface systems on a chip working at different speeds. The connected systems can be either synchronous or asynchronous. The designs are then adapted to work between systems with very long interconnect delays, by migrating a single-clock solution by Carloni *et al.* (1999, 2000, and 2001) (for "latency-insensitive" protocols) to mixed-timing domains. The new designs can be made arbitrarily robust with regard to metastability and interface operating speeds. Initial simulations for both latency and throughput are promising.**

***Index Terms*—Digital systems, metastability, synchronization, timing interfaces.**

I. Introduction

FUTURE very large scale integration (VLSI) systems will likely be systems-on-a-chip involving many timing domains [15]. A challenging problem is to robustly interface these domains. There have been few adequate solutions, especially ones providing reliable low-latency communication. The contribution of this paper is the design of low-latency, high-throughput FIFOs (first-in–first-out) interfaces that robustly accommodate mixed-timing systems.

There are two fundamental challenges in designing systems-on-a-chip: systems operating under different timing assumptions, and long interconnect delays in communication between systems. This paper addresses both of these issues. First, four new *mixed-timing* FIFOs are introduced that address the first design challenge, for any combination of synchronous and asynchronous interfaces. Interfaces are assumed to be point-to-point; and for synchronous-to-synchronous interfaces, arbitrary (uncorrelated) clock rates are allowed.

For the second design challenge, "latency-insensitive protocols" were proposed by Carloni *et al.* [3]–[5]; however, their solution was limited to a single-clock domain. In this paper, their solution is generalized to mixed-timing systems: three new *mixed-timing relay stations* are introduced, which build on our basic mixed-timing FIFO designs. No previous solution, of which we are aware, has been proposed that simultaneously solves the above two design challenges: handling mixed asynchronous/synchronous interfaces and accommodating long interconnect delays.

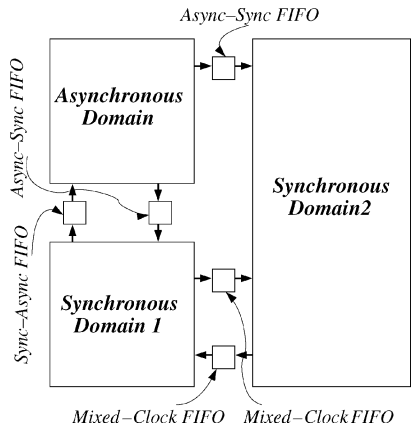

Fig. 1. System with mixed-timing domains.

In addition, as an extension, a hybrid solution is proposed, where the mixed-clock FIFO design is transformed into a "dual-mode" version to handle both arbitrary and equal clock frequencies.

An important theme of our approach is to partition the FIFOs into reusable components, both at the architecture and cell levels. A set of interfaces, both synchronous and asynchronous, is defined; these interfaces can then be glued together to obtain FIFOs which meet the desired timing assumptions on both the sender's and receiver's end. Thus, the design of a mixed-timing FIFO is reduced to combining a few predesigned components.

Our new FIFOs are especially suitable for high-bandwidth communication: assuming appropriate buffer capacity is used and in steady-state operations the designs have *no synchronization overhead*—each read and write can be completed in one cycle. At the same time, the designs are also optimized for low latency, and thus, are suitable for infrequent communication.

As an example, Fig. 1 shows a block-level view of a system-on-a-chip, which consists of several timing domains: one asynchronous domain, and two synchronous domains each operating under a different clock. The three domains communicate with each other; for example, the asynchronous domain both sends and receives data items to/from synchronous domain 1. Connecting the domains directly over wires may lead to potential data synchronization errors in the receiver due to the different timing in each domain. Therefore, the solution is to use *mixed-timing* FIFOs which safely transfer data items between two communicating domains (Fig. 1 shows three types of FIFOs: mixed clock, asynchronous–synchronous, and synchronous–asynchronous). The mixed-timing FIFOs (Fig. 1) are inserted between the sender domain and the receiver domain. For example, in Fig. 1, there are three types of FIFOs:

Manuscript received September 4, 2003; revised February 26, 2004. This work was supported in part by the National Science Foundation under Award CCR-97-34803, in part by the National Science Foundation Information Technology Research (ITR) under Award NSF-CCR-0086036, in part by a gift from Sun Microsystems, Mountain View, CA, and in part by a grant from the New York State Microelectronics Design Center, New York State Office of Science, Technology and Academic Research (NYSTAR). This paper was presented in part at the IEEE 2000 Symposium on Asynchronous Circuits and Systems, and at the 2001 Conference on Design Automation.

T. Chelcea was with Columbia University, New York, NY 10027-6902 USA. He is now with the Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213 USA (e-mail: tibics.cmu.edu).

S. M. Nowick is with the Department of Computer Science, Columbia University, New York, NY 10027 USA (e-mail: nowickcs.columbia.edu).

Digital Object Identifier 10.1109/TVLSI.2004.831476

mixed-clock FIFO (when both the sender and the receiver are synchronous, potentially under arbitrary uncorrelated clock rates), asynchronous–synchronous FIFO (when the sender is asynchronous and the receiver is synchronous), and finally, synchronous–asynchronous FIFO (when the sender is synchronous and the receiver asynchronous). These FIFOs robustly transfer data from one domain to the other, have low latency in communication, and, very important to design reuse, *do not require any modifications to the sender or the receiver domain*.

The paper is organized as follows. The remainder of this section discusses related work in detail, including comparisons and tradeoffs between our approach and some recent approaches. In Section II, a top-level view of the basic FIFO designs is presented, similarities and differences are highlighted, and the critical issues of synchronization and deadlock are raised. In Section III, the mixed-clock FIFO is presented in detail, and simple solutions to both synchronization and deadlock problems are proposed. In Section IV, the asynchronous–asynchronous FIFO is introduced and discussed in detail. Reusing most of the components in the synchronous–synchronous and asynchronous–asynchronous FIFOs, in Section V, the asynchronous–synchronous and synchronous–asynchronous FIFOs are introduced. In Section VI, each basic FIFO design is then transformed into a mixed-timing relay station, to safely handle long interconnect delays. Finally, the results for throughput and latency in each design are shown in Section VII, and conclusions are presented in Section VIII.

A. Related Work

There has been a large body of related work on FIFOs and components to handle timing discrepancies between subsystems. Several solutions have been proposed for the restricted case of single-clock systems. These approaches handle clock skew [21], [25], drift and jitter [25].

A number of approaches have been proposed for interfacing mixed-timing domains, which is also the target of this paper.

Approaches Using Pausible and Stretchable Clocks: One approach for mixed-timing interfaces is to temporarily *pause* [7], [41] or *stretch* [1] the receiver's clock. In contrast to our approach, such designs require asynchronous "wrapper logic" around the receiver and also suffer from penalties in restarting the receiver's clock. In addition, these approaches require the modification of the receiver subsystem, which makes them less suitable for design reuse. An interesting approach by Sjogren and Myers [38] interfaces asynchronous to synchronous domains, and combines a high-speed pipeline with mechanisms for clock pausing in order to achieve very high throughput, without potential metastability failures. However, the latency of their design is still proportional to the number of pipeline stages, and their approach can introduce penalties in starting and stopping the receiver's clock.

Approaches Using Synchronizers: An alternative approach attempts to synchronize data items and/or control signals with the receiver, without interfering with its clock. Seitz [32] uses simple two-latch synchronizers to synchronize data signals. However, this approach suffers from area and throughput penalties compared with ours. Seizovic [33] robustly interfaces *asynchronous* with *synchronous* environments through a "synchronization FIFO." However, the latency of his design is proportional to the number of FIFO stages, where the implementation includes expensive synchronizers. Furthermore, the design requires the sender to produce data items at a constant rate, unlike ours. Sarmenta [31] interfaces mixed-timing systems by introducing complex components that delay the transmission of data items whenever the communication might be unsafe; this approach introduces large area overheads. Finally, Ginosar *et al.* [35] introduces several FIFO synchronizer designs and analyzes their properties. In [20], Ginosar provides an excellent overview of the most common errors in interfacing mixed-clock domains.

Three recent mixed-clock FIFOs are closest to our proposed approach. A recent patent from Intel [24] proposes a highly-optimized mixed-clock FIFO. It has a number of similarities to our basic mixed-clock FIFO in overall structure and operation, but has significantly greater area overhead in implementing the synchronization: while our design only has three synchronizers for the entire FIFO (*independent* of the number n of FIFO cells), the Intel design requires one synchronizer *per FIFO cell* ($n+1$ synchronizers). Thus, while additional cells can be added to our FIFOs without changes to the synchronizers, their design would require new synchronizers to be added. Furthermore, while the Intel design is only applicable for mixed-clock domains, we propose a family of interface circuits to handle arbitrary combinations of synchronous and asynchronous domains (including mixed-clock domains).

Both Pham [30] and Dally [14] also propose mixed-clock FIFOs. In their designs, a RAM is used to store data items, while two counters are used to index the head and the tail of the queue; finally, a comparator is used to generate the global full and empty signals. Compared to our mixed-clock FIFO, Dally's design [14] has significantly greater area overhead: it uses $2n$ synchronizers to synchronize the addresses for the head and tail of the FIFO (where again, n is the number of data items that can be stored in the FIFO). In addition, Dally's design attempts to synchronize the head and the tail on every clock cycle, which leads to the degradation of the FIFO throughput; in contrast, our design only incurs synchronization overhead when the FIFO approaches full or empty. Pham's design [30] is used to interface mixed-clock systems operating at very low frequencies and it *does not synchronize* the global control signals (full and empty). The lack of synchronization makes this design unsuitable for use at higher clock frequencies, where the failure rate would be unacceptable.

Comparison to Chakraborty–Greenstreet's Approach [6]: A recent paper by Chakraborty and Greenstreet [6] proposes a family of interface circuits that mediate between mixed-clock domains. Their approach, published after our initial results [9]–[11], is especially effective at reducing latency penalties. The approach has both advantages and disadvantages compared to our approach, hence a detailed comparison is included below.

Chakraborty *et al.* start with a basic design, controlled by two identical clocks; the latency of this design is two clock cycles. This basic design is then showed to handle clock jitter, while introducing no additional latency penalties in communication. The authors then propose two interesting extensions, which handle rational clock-frequency multiples and plesiochronous

clocks, which exhibit an average latency of only half a clock cycle, with a worst-case penalty of two clock cycles. Finally, the authors also discuss briefly extensions to arbitrary clock frequencies and FIFO interfaces.

There are several tradeoffs between their approach and ours. First, their FIFOs are more restrictive and less robust than ours: their designs are limited to single-clock domains (with jitter) and to special cases of mixed-clock interfaces (rational clock-frequency multiples, plesiochronous clocks). In contrast, our designs can handle arbitrary mixed-clock domains.

Second, in those domains where their designs can be used, our latency penalties may sometimes be larger than theirs. In particular, the worst-case operation of our mixed-clock FIFO occurs when there is a large mismatch in the communication rate between sender and receiver, and the FIFO constantly hits the "empty" (or "full") state, and there is a stalled pending consumer (or producer, respectively). In this case, each time a new data item is enqueued (or dequeued), a new latency penalty is incurred (see Section III). This penalty is further exacerbated if one of the clock rates is very high; in this case, our design may include extra synchronization latches to ensure robust synchronization (see Section III-E).

However, it is important to note that for many applications, our FIFOs do not incur these penalties. For systems where the computation rates for the producer and consumer are roughly equal (i.e., the rates at which data items are produced and consumed), our FIFOs introduce *no communication overhead:* the receiver and sender communicate with different parts of the FIFO concurrently, and each subsystem can put or get its data on *every clock cycle,* without any penalty. In these cases, our designs can tolerate and smooth out a variety of mismatches between domains, which cannot necessarily be handled by Chakraborty's approach. Furthermore, for systems which are not latency-critical, such as the recently-proposed class of *latency-insensitive systems* [3]–[5], our FIFOs fit well and also gracefully handle timing discrepancies that can occur.

Finally, while [6] concentrates only on mixed-clock communication, this paper presents a complete family of interfaces for arbitrary combinations of asynchronous and synchronous domains, as well as further extending the work to handling long interconnect delays.

In summary, Chakraborty's approach is highly effective for a more limited domain; our approach at times may incur worse latency penalties (and at times not), but is more general and more robust. An interesting avenue for future research is to try to meld the two approaches. A first step in this direction is a "dual-mode" hybrid design, which we present in Section III-F, which can handle both arbitrary and equal clock frequencies.

Handling Timing Discrepancies due to Long Interconnect: Carloni *et al.* [3] propose a solution to handling arbitrary interconnect delays between sychronous subsystems. They define the notion of a *latency-insensitive system,* and propose channels between the islands which are subdivided by *relay stations*. However, their approach is limited to single-clock domains. In this paper, relay stations are introduced for mixed-timing domains. These are the first proposed solutions which simultaneously address both discrepancies in timing between domains, as well as long interconnect delays between these domains.

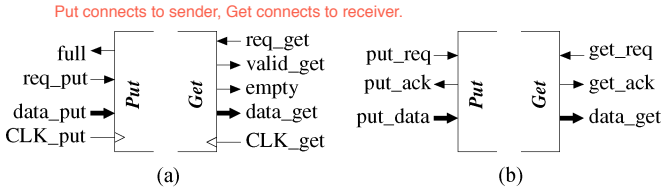

Fig. 2. Synchronous and asynchronous FIFO interfaces: (a) synchronous interfaces and (b) asynchronous interfaces.

Recent Improvements to Our Proposed Approach: Several recent designs published in the past two years have built on the FIFOs presented in this paper and in our earlier publications [9]–[11], and have attempted to further improve them. Moore [26] presents interface circuits between asynchronous and synchronous domains that employ clock pausing. A high-speed ripple FIFO [8] incorporates a novel mechanism (compared to the designs proposed in this paper) to detect almost full/empty; however these FIFOs have a latency proportional to the number of FIFO stages. A recent paper [37] improves the communication behavior of Carloni's basic design (i.e., decouples the input and the output of a subsystem), but does not address the issue of timing-discrepancies between domains. It is interesting to notice that this approach is orthogonal to our proposed relay stations, and can be used with either single-clock or mixed-clock relay stations.

II. Overview of the New FIFOs

The mixed-timing FIFOs can be best understood by looking first at the underlying common themes across all of the designs. This section presents an overview of the basic FIFO interfaces, the underlying architectures and special features, of the mixed-timing FIFOs. Later, in Sections III–V, details of the actual FIFO implementations will be presented, as well as a detailed solution to the synchronization problem.

A. FIFO Interfaces

Each FIFO has two interfaces: a put interface (for the sender) and a get interface (for the receiver). Each interface, in turn, can be either synchronous or asynchronous (Fig. 2). The put and get interfaces can be freely combined to build mixed-timing FIFOs. For example, to build a mixed-clock FIFO, the synchronous put interface is combined with the synchronous get interface. Similarly, to build an asynchronous–synchronous FIFO, the asynchronous put interface is combined with the synchronous get interface.

There are two types of synchronous interfaces: a put interface and a get interface. The synchronous put interface [Fig. 2(a)] is controlled by $\mathrm{CLK_{put}}$. There are two inputs: $\mathrm{req_{put}}$ which controls requests, and $\mathrm{data_{put}}$ which is the bus for data items. The *full* output is only asserted when the FIFO is full, otherwise, it is deasserted. The synchronous get interface [Fig. 2(a)] is controlled by $\mathrm{CLK_{get}}$ and a single-control input $\mathrm{req_{get}}$. Data is placed on $\mathrm{data_{get}}$ output bus, and *empty* is asserted only when the FIFO is empty. For most of this paper, $\mathrm{valid_{get}}$ is always asserted during a get operation.

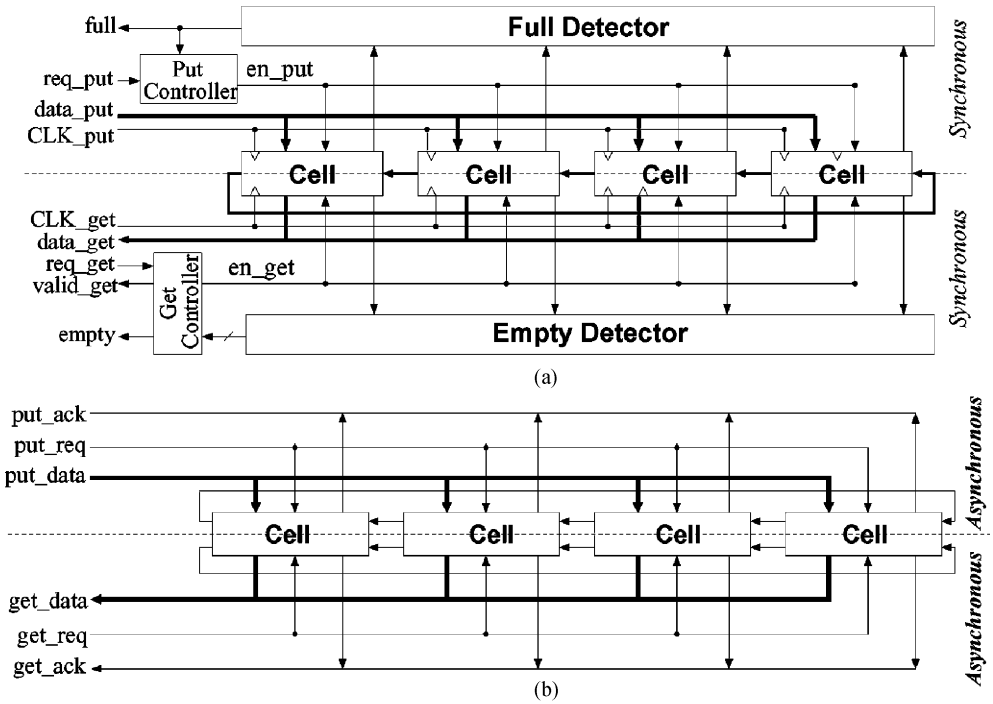

Fig. 3. Two architectures for mixed-timing FIFOs. (a) Synchronous–synchronous FIFO and (b) asynchronous–asynchronous FIFO.

Since the asynchronous interfaces are not synchronized to a clock signal, they are somewhat different. The asynchronous put interface [Fig. 2(b)], much like the synchronous put interface, has two inputs: $\mathrm{put}_{\mathrm{req}}$ which controls requests, and $\mathrm{put}_{\mathrm{data}}$, the bus for data items. However, this interface does not have a *full* output; instead, the interface simply withholds $\mathrm{put}_{\mathrm{ack}}$ until the FIFO becomes non-full. The asynchronous get interface [Fig. 2(b)] has only $\mathrm{get}_{\mathrm{req}}$ as input, which controls the requests for data, and two outputs: $\mathrm{get}_{\mathrm{data}}$, the bus for data items, and $\mathrm{get}_{\mathrm{ack}}$, which indicates the completion of the get operation. This interface does not have an *empty* output; instead, the interface simply withholds $\mathrm{get}_{\mathrm{ack}}$ until the FIFO becomes non-empty.

B. Basic Architecture

Fig. 3 gives a simple overview of the two basic FIFO architectures. Similar architectures can be defined for the asynchronous–synchronous and synchronous–asynchronous FIFOs: the synchronous–asynchronous put/get parts can be freely combined to obtain these mixed asynchronous–synchronous FIFOs.

There are a number of similarities. Each FIFO is constructed as a circular array of identical cells, communicating with the two external interfaces (put and get) on common data buses. The control logic for each operation is distributed among the cells, and allows concurrency between the two interfaces. An important feature of all the circular FIFOs architectures is that data is immobile: once enqueued, it is not moved and is simply dequeued in place.

Two tokens control the input and output behavior of the FIFO: a *put* token is used to enqueue data items, and a *get* token is used to dequeue data items. The cell with the put token is the tail of the queue, while the cell with the get token is its head. Once a cell has used a token for a data operation the token is passed to the next cell. In normal operation, the get token is never ahead of the put token; however, in some special cases, the get token may briefly overtake the put one.

There are several advantages that are common to the proposed architectures. Since data is not passed between the cells from input to output, the FIFOs have a potential for low latency, and as soon as a data item is enqueued, it is also available for dequeuing (see Section VII). Second, the FIFOs offer the potential for low-power and data items are immobile while in the FIFO. Finally, these architectures are highly scalable; the capacity of the FIFO and the width of the data item can be changed with very few design modifications.

C. Empty/Full Detectors and External Controllers

The synchronous interfaces have two additional types of components: *detectors*, which compute the current state of the FIFO, and *external controllers*, which conditionally pass requests for data operations to the cell array. The full and empty detectors observe the state of all cells and compute the global state of the FIFO: full or empty. The output of the full detector is passed to the put interface, while that of the empty detector is passed to the get interface. The put and get controllers filter data-operation requests to the FIFO. Thus, the put controller usually passes put requests, but disables them when the FIFO is full. The get controller normally forwards the get requests, but blocks them when the FIFO is empty.

The asynchronous interfaces do not need such external detectors and controllers. A data operation on a synchronous interface completes within a clock cycle; therefore, the environment does not need an explicit acknowledge. However, if the FIFO becomes full (empty), the environment may need to be stopped from communicating on the put (get) interface. The role of the detectors and controllers is to: a) detect the exception cases and

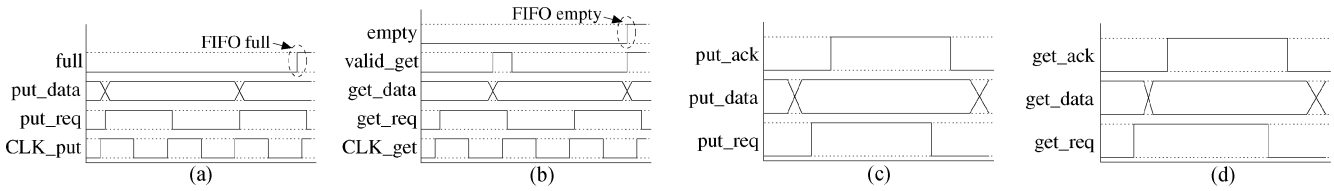

Fig. 4. The protocols for each interface. (a) Synchronous put protocol. b) Synchronous get protocol. (c) Asynchronous put protocol. (d) Asynchronous get protocol.

b) stall the respective interface until it is safe to perform the data operation. In contrast, the asynchronous interfaces do not need such explicit FIFO state signals. When the FIFO becomes full (empty), the put (get) acknowledgment can be withheld indefinitely until it is safe to perform the data operation.

Could this cause mal-functions of the asynch sender module?

D. FIFO Protocols

Now that the various interfaces have been discussed, the behavior of each can be best understood through some simple simulations (Fig. 4).

The synchronous protocols are somewhat more complex than the asynchronous ones, and will be discussed first. The synchronous *put interface* starts a put operation, shown in Fig. 4(a), when it receives a request on $\mathrm{put}_{\mathrm{req}}$ and a data item on $\mathrm{put}_{\mathrm{data}}$, immediately after the positive edge of $\mathrm{CLK}_{\mathrm{put}}$. The data item is enqueued at the start of the next clock cycle. If the FIFO becomes full, then *full* is asserted before the next clock cycle, and the put interface is prevented from any further operation.

A synchronous *get operation* [Fig. 4(b)] is enabled by a request on $\mathrm{get}_{\mathrm{req}}$, asserted immediately after the positive edge of $\mathrm{CLK}_{\mathrm{get}}$. By the end of the clock cycle, a data item is placed on $\mathrm{get}_{\mathrm{data}}$ together with its validity bit ($\mathrm{valid}_{\mathrm{get}}$). If the FIFO becomes empty, that clock cycle *empty* is also asserted, and the get interface is stalled until the FIFO becomes non-empty. Following a get request, $\mathrm{valid}_{\mathrm{get}}$ and *empty* can indicate three outcomes: 1) data item dequeued, more data items available ($\mathrm{valid}_{\mathrm{get}} = 1, \mathrm{empty} = 0$); 2) data item dequeued, FIFO has become empty ($\mathrm{valid}_{\mathrm{get}} = 1, \mathrm{empty} = 1$); and 3) FIFO empty, no data item dequeued ($\mathrm{valid}_{\mathrm{get}} = 0, \mathrm{empty} = 1$).

The asynchronous interfaces use a four-phase communication with single-rail bundled data [19], [39] on both put and get interfaces (discussed below in this order). Bundled data is a common scheme where a worst-case matched control signal (e.g., $\mathrm{put}_{\mathrm{req}}$) indicates when data is valid (e.g., $\mathrm{put}_{\mathrm{data}}$). The sender starts a *put operation* [Fig. 4(c)] by placing a data item on $\mathrm{put}_{\mathrm{data}}$ and requesting the FIFO to enqueue it on $\mathrm{put}_{\mathrm{req}}$. The enqueuing completion is indicated by asserting $\mathrm{put}_{\mathrm{ack}}$. The two control wires are then reset to the idle state, first $\mathrm{put}_{\mathrm{req}}$ and then $\mathrm{put}_{\mathrm{ack}}$. The receiver starts a *get operation* [Fig. 4(d)] by requesting the FIFO on $\mathrm{get}_{\mathrm{req}}$ to dequeue a data item. The FIFO places the data item on $\mathrm{get}_{\mathrm{data}}$ and indicates on $\mathrm{get}_{\mathrm{ack}}$ that the data item can be read by the receiver domain. The two control wires are then reset to the idle state, first $\mathrm{get}_{\mathrm{req}}$ and then $\mathrm{get}_{\mathrm{ack}}$.

III. Mixed-Clock FIFO

This section now presents in more detail the first of the two new basic mixed-timing FIFOs: the mixed-clock (synchronous–synchronous) FIFO. The design is not only useful in itself but also provides a basis for understanding all the remaining FIFO designs. Components from this design are also reused in the other FIFOs. Furthermore, the solution to the synchronization problem is also introduced for this FIFO; the same solution will be applied to all later designs.

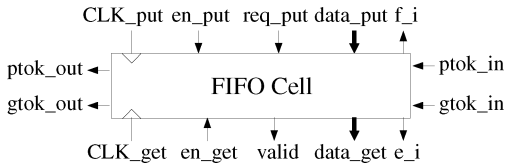

Fig. 5. Mixed-clock FIFO cell's interface.

The synchronous–synchronous FIFO interfaces are shown in Fig. 2(a). The FIFO protocol on the put and get interfaces has been described in Fig. 4(a)–(b), and the architecture of the FIFO was described in Section II. For the rest of the section, the implementation of the basic cell, the empty/full detectors and the external controllers are presented. The latency of the FIFO is analytically derived. Finally, some issues in the robustness of communication are also addressed; the section concludes with an extension to the basic design that can improve FIFOs latency under variations in clock frequencies.

A. Cell Implementation

A block diagram of an individual cell is shown in Fig. 5. Each cell has four interfaces: on the synchronous put interface, the cell receives data on $\mathrm{data}_{\mathrm{put}}$ and it is enabled on $\mathrm{en}_{\mathrm{put}}$ to perform a put operation; $\mathrm{req}_{\mathrm{put}}$ indicates data validity (it is always 1 in this design). The cell communicates with the *full detector* on e_i, which is asserted high when the cell is empty. On the get interface, the cell outputs data on $\mathrm{data}_{\mathrm{get}}$, together with its validity bit *valid* (always 1 in this design); the interface is enabled on $\mathrm{en}_{\mathrm{get}}$. The cell communicates with the *empty detector* on f_i, which is asserted high when the cell is full. Each cell receives tokens on $\mathrm{ptok}_{\mathrm{in}}$ (put token) and $\mathrm{gtok}_{\mathrm{in}}$ (get token) from the right cell and passes the tokens on $\mathrm{ptok}_{\mathrm{out}}$ and $\mathrm{gtok}_{\mathrm{out}}$ to the left cell.

A detailed implementation of a cell is shown in Fig. 6. The cell's behavior is illustrated by tracing a *put operation* and then a *get operation*. Initially, the cell starts in an empty state ($e_i = 1$ and $f_i = 0$) and without any tokens. The cell waits to receive the put token from the right cell on the positive edge of $\mathrm{CLK}_{\mathrm{put}}$, and waits for the sender to place a valid data item on the $\mathrm{data}_{\mathrm{put}}$ bus. A valid data item is indicated to all cells by $\mathrm{en}_{\mathrm{put}} = 1$, which is the output of the *put controller*.

When there is valid data, the cell performs three actions: 1) it enables register *REG* to latch the data item and also the data validity bit (which is $\mathrm{req}_{\mathrm{put}}$); 2) it indicates that the cell has a

SR - Set/Reset Latch

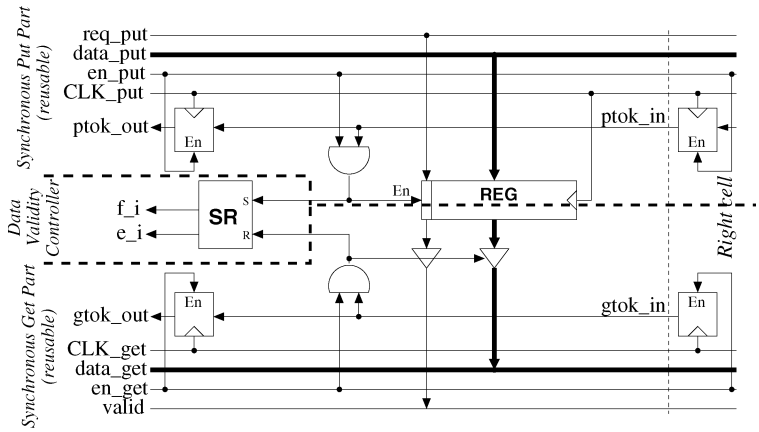

Fig. 6. Mixed-clock FIFO cell implementation.

valid data item (asynchronously sets $f_i = 1$); and 3) it enables the upper left ETDFF ($\mathrm{en_{put}} = 1$) to pass the put token to the left cell. On the positive edge of the next clock cycle, the data item and validity bit are finally latched and the put token is passed to the left cell.

The behavior for dequeuing data is quite similar. The cell waits to receive the get token ($\mathrm{gtok_{in}} = 1$) and waits for the receiver to request a data item ($\mathrm{en_{get}} = 1$, the output of the *get controller*). When both conditions hold, the cell performs three actions: 1) it enables the broadcasting of the data item on the $\mathrm{data_{get}}$ tristate bus and the broadcasting of v_i (the latched $\mathrm{req_{put}}$) on the *valid* tristate bus; 2) indicates that the cell is empty (asynchronously sets $e_i = 1$); and 3) enables the lower left ETDFF to pass the get token. At the beginning of the next clock cycle, the get token is then passed to the left cell.

The mixed-clock FIFO cell can be divided into three distinct parts (Fig. 6): a reusable *synchronous put part* that deals with the put operation, a reusable *synchronous get part* that deals with the get operation, and a *data validity controller*. The register is split into two parts, one belonging to the put part (the write port), and one belonging to the get part (the get port). The put and get parts will be reused in the mixed asynchronous/synchronous FIFOs, while the data validity controller is unique to this mixed-clock FIFO.

B. Synchronization Issues

A key challenge of designing a mixed-clock FIFO is that of synchronization. Such a FIFO has a highly-concurrent operation: the put and get interfaces may change and read the state of the FIFO concurrently under two different clocks. Therefore, *synchronizers* must be added to the two global control signals (*full* and *empty*). Unfortunately, the added delays through the synchronizers may cause overflow/underflow in the FIFO. A simple solution is to modify the definitions of *full* and *empty*, to anticipate imminent full and empty states. However, a complete safe solution also needs to avoid deadlock, which may occur when using the modified *empty* definition. The proposed safe solution is derived below through a series of steps, each presented in turn; the subsection ends with a discussion of the FIFO correctness when it enters a metastable state.

Synchronization of Control Signals: Global control signals have to be resynchronized to the interfaces' clocks. The problem is that the state of the FIFO (full/empty) is manipulated by the two interfaces, while it is read by only one of them (*full* by the put interface, *empty* by the get interface). A simple and robust solution is to use synchronizing latches. The current designs use only a pair of synchronizing latches; however, for arbitrary robustness, the designer might use more than two. The synchronizers are added to the output of the full and empty detectors, and are controlled by $\mathrm{CLK_{put}}$ and $\mathrm{CLK_{get}}$, respectively.

Modification of Full and Empty Detectors: The added latencies through the synchronizers may cause the FIFO to overflow or underflow. For example, when the FIFO becomes empty, the receiver interface is stalled two clock cycles later; so in the next clock cycle the receiver might request and read an empty cell. A similar problem arises when the FIFO becomes full.

The solution to the over/underflow problem is to change the definitions of *full* and *empty*. The FIFO is now considered "full" when there are either 0 or 1 empty cells left, and it is considered "empty" when there are either 0 or 1 cells filled. Thus, when there are fewer that 2 data items, the FIFO is declared empty; the receiver may then remove the last data item and issue a new unanswered request, before stalling two clock cycles later. (A similar behavior applies for the full case.) The new definitions do not change the protocol with the two systems. The only effect is that sometimes the two systems see an n-place FIFO as a $n-1$ place one.

The new implementations of full and empty detectors, presented in Fig. 7(a)–(b), correspond precisely to the above new definitions: the FIFO is full when there are no two *consecutive* cells empty, and the FIFO is empty when there are no two *consecutive* cells full.

Deadlock Avoidance: Unfortunately, the early detection of empty, in some cases, may cause the FIFO to deadlock. Using the new empty definition (0 or 1 data items), it is possible that the FIFO still contains one data item, but the requesting receiver is still stalled.

The final safe solution is, therefore, to use a *bi-modal* empty detector. The detector, in addition to computing the "new empty" definition (*ne*), also computes the "true empty" one [*oe*, see Fig. 7(c)]: the FIFO is empty when there are zero full cells left. The two empty signals are then synchronized with the receiver clock and combined through an AND gate (Fig. 8) to form the global *empty* signal. The FIFO dynamically alternates

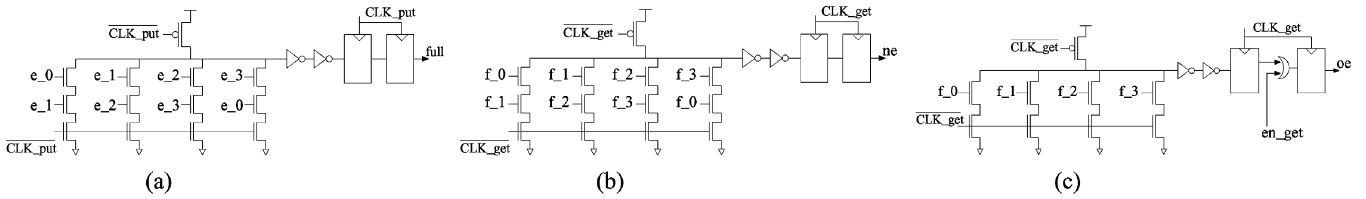

Fig. 7. Full and empty detectors for the mixed-clock FIFO: (a) full detector; b) detector for "new" empty; and (c) detector for normal empty.

between the two different empty detectors depending on recent access patterns, thus avoiding both synchronization failure and deadlock.

The intuition behind the bi-modal detector is as follows. If there have not been any recent gets—for at least one clock cycle—*oe* dominates. This is especially important when there is only one data item in the FIFO: the get interface needs to receive it, so *oe* is used to indicate the FIFOs state (not empty). However, when the get interface has just removed a data item, *ne must* be used to indicate the state, in order to prevent the FIFO underflow, which the synchronization delays for *oe* might cause.

The two empty definitions produce the same result in all but one case: when there is *exactly* one data item in the FIFO. Suppose that the get interface has just removed the next-to-last data item in the FIFO. If in the current clock cycle there is another get request, the request is satisfied and *ne* will stall the get interface in the next clock cycle (it will assert "FIFO empty"). However, if there is *no* get request, then *oe* will dominate in the next clock cycle (it will assert "FIFO not empty"), allowing a subsequent get request to be satisfied. Whenever the last data item is dequeued, *ne* again immediately dominates and stalls the get interface on time. At this point, no further gets will be satisfied, so *oe* again will be used to indicate the FIFO's state.

The "true empty" detector [Fig. 7(c)] and the get controller [Fig. 8(b)] implement exactly the above behavior. The OR gate in the *oe* synchronizer is very important: controlled by $\mathrm{en_{get}}$, it sets the *oe* to a neutral state ("FIFO empty") one clock cycle after a get operation takes place. In this case, the "new empty" definition can take precedence in the get controller.

Metastable States: In general, the solution to synchronize the global control signals to their respective clock signals will not avoid metastability states, which result in incorrect readings of these signals on the respective interfaces. However, the particular implementation of the Full and Empty controllers guarantees that, even if these signals are incorrectly read, the FIFO does not perform an illegal operation. During every clock cycle, the detectors first set the global state to full/empty during the positive phase of the clock, and then, during the negative phase of the clock compute the actual state. Thus, the global full/empty signals can enter a metastable state *only* on a transition from "full"/"empty" to "not full"/"not empty." That is, the input to the synchronizing latches will not be changing arbitrarily, but may only be changing monotonically. Therefore, at the output of the synchronizer, an incorrect reading of these signals will result in stalling the interfaces for one clock cycle. In addition, should the synchronizer produce an intermediate value between 0 and 1, a simple filtering gate [32] can be attached to produce a clear 0/1 to the full/empty detector. This behavior, while clearly detrimental to performance, will not cause the FIFO to overflow or underflow.

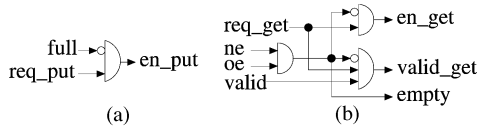

Fig. 8. The external controllers for the mixed-clock FIFO: (a) put controller and (b) get controller.

C. Put and Get Controllers

Finally, the implementation of the put controller is shown in Fig. 8(a). The controller enables and disables the put operation and the movement of the put token in the FIFO: these operations are only enabled when there is a valid data item on $\mathrm{data_{put}}$ ($\mathrm{req_{put}}$ asserted) and the FIFO is not full.

The get controller enables and disables the get operation and the movement of the get token in the FIFO: they are only enabled when there is a request from the receiver and at least one of the empty detectors (*ne* and *oe*) indicates the FIFO is not empty. The simple implementation in Fig. 8(b) corresponds exactly to these conditions.

D. Latency Analysis

Having presented the basic design of the mixed-clock FIFO, this section provides an analysis of its latency. Latency is defined as the time it takes from enqueuing to dequeuing a data item through an *empty FIFO,* when the get interface is pending (i.e., it is stalled, while actively requesting a data item). In this case, the FIFO exhibits worst-case behavior, because it needs to be restarted. (In contrast, this restart overhead would not be exhibited if the FIFO were not empty, and instead operating in steady-state.)

For the mixed-clock FIFO, latency is not uniquely defined; instead it is an interval whose minimum (min) and maximum (max) values depend on the relative phases of the put and get clocks. In the following, the conditions for these two values are derived as a function of the two clock periods.

To derive the latency expression, it is useful to look at the critical path of the FIFO. A data item is placed on the global buses by the put interface, and the cell with the put token changes its state from empty to full in the first half of the put clock cycle. This signal goes to the empty detector. Eventually, during the negative phase of the next clock cycle, the empty detector computes the FIFO state as "not empty." The output of the detector is synchronized through a pair of latches, adding one get clock cycle. Finally, the get interface can now dequeue the data item and store it in one extra clock cycle. It is easy to see that the

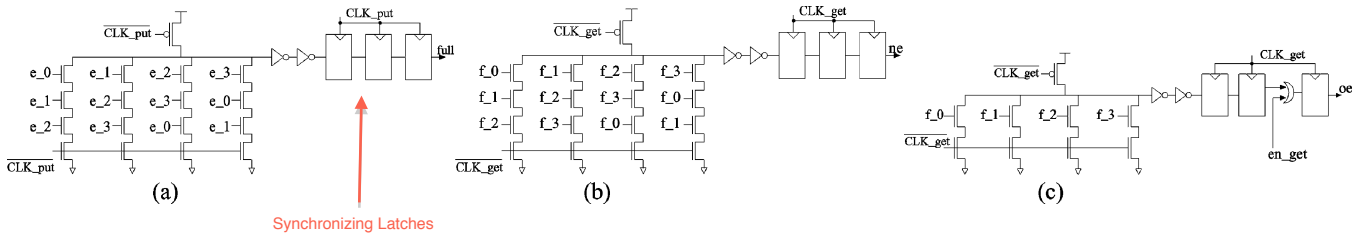

Fig. 9. Robust full and empty detectors for mixed-clock FIFO: (a) full detector; (b) detector for "new" empty; and (c) detector for normal empty.

delay is variable, based on phase difference between two clock cycles.

only unknown is the time elapsed from changing the state of a cell until the start of the computation in the empty detector; this delay is determined by the relative phases of the put and get clock cycles.

The "min" and "max" expressions for latency are derived as follows. The put clock period is T_{put}, and the get clock period is T_{get}. For the "min" expression, the negative phase of the get clock is aligned with the negative phase of the put clock: a change in the state of a cell (enabled by a put operation) immediately triggers the computation of the FIFO state as "not empty" (performed during the negative phase of CLK_{get}). Thus, based on the discussion of the critical path, the minimum latency is: $L_{\min} = 0.5T_{\text{put}} + 2.5T_{\text{get}}$. For the "Max" latency, the negative phase of the Get clock is aligned with the positive phase of the Put clock: a change in the state of a cell has to wait the entire positive phase of the Get clock until it triggers the computation in the Empty detector. Thus, the maximum latency is $L_{\max} = 0.5T_{\text{put}} + 3T_{\text{get}}$.

The above analysis of the "min" and "max" conditions form the basis of the experimental results presented in Section VII.

E. Robustness Issues

The FIFO design developed in this section works under many operating conditions. However, there are some cases where it might fail. The first case is due to large discrepancies between the sender's and receiver's clock frequencies. The second case is when the FIFO interfaces operate at very fast clock frequencies, and metastable states on control signals cannot be solved in only one clock cycle. Below, we present solutions to these problems that make the FIFO design more robust.

Large Differences in Clock Speeds: The mixed-clock FIFO design uses a small optimization which may cause malfunction in extreme cases: early signaling of full/empty for a cell. As shown in Fig. 6, a cell's full/empty value is set *asynchronously* in the SR latch during the clock cycle, but the data operation is only completed on the next clock edge. This optimization may cause problems when the relative clock rates of the two interfaces are very different.

The potential for error can be illustrated through a simple simulation. Suppose there are exactly two data items in the FIFO and the sender sends a new data item. At the beginning of the clock cycle, the tail cell is set to full, but the actual data will only be latched at the end of the clock cycle. However, if the receiver's clock frequency is more than three times that of the sender, the receiver can quickly read the two intervening valid data items and access then the tail cell. In this case, it can then read the stale data from this cell and complete the get operation, *before* the sender has completed its clock cycle and latched the new data.

A simple and robust solution is to modify the full or empty detectors (whichever controls the faster interface) to stop the faster interface earlier. Suppose that the receiver is much faster than the sender. The definition of empty is modified as follows: the FIFO is empty when there are less than three full cells left. In this case, the receiver will stall before reading the last item, so no stale data will be dequeued.

Fast Clock Speeds: It is desirable to have high *mean time between failures* (MTBF) in the FIFO operation. At low-clock frequencies, using two latches for synchronizing the global control signals is usually sufficient for good MTBF, but at higher clock frequencies, the MTBF may be inadequate.

The synchronization of global control signals can be made arbitrarily robust. To do so, more latches can be added to whichever global control signal (or signals), *full* and *empty*, has poor MTBF. However, the added latencies through the extra latches may introduce underflow/overflow in the basic mixed-clock FIFO.

The solution to the overflow/underflow problem is to change the definitions of "FIFO full" and "FIFO empty." Suppose that synchronizing the global control signals ("new full," "new empty," and normal empty) require k latches. Therefore, to prevent overflow, the FIFO is considered "full" when there are fewer than k empty cells in the FIFO. To prevent underflow, the FIFO is considered "empty" when there are fewer than k full cells in the FIFO.

Fig. 9 shows the implementations of the three detectors for $k = 3$ (i.e., there are three synchronizing latches at their output). The implementations of "new full" and "new empty" detectors have now changed: the FIFO is full when there are no *three* consecutive empty cells, and the FIFO is empty when there are no *three consecutive* full cells. Notice that the implementation of the normal empty detector is not changed.

These simple modifications to the basic mixed-clock FIFO can make the synchronization of global control signals arbitrarily robust. However, this solution may have a negative impact on the throughput of the get interface. A simple simulation will illustrate this problem. Suppose the FIFO has two data items and that there have not been any recent gets (i.e., *ne* indicates "FIFO empty," *oe* indicates "FIFO not empty," and the get interface is not stalled). When the get interface removes a data item, *ne* takes precedence and the FIFO will stall the get interface for the next clock cycle. After this stall cycle, *oe* takes precedence (indicating "FIFO not empty") and the get interface can safely remove the last data item. Thus, the throughput on the get interface is now decreased: two data items are dequeued

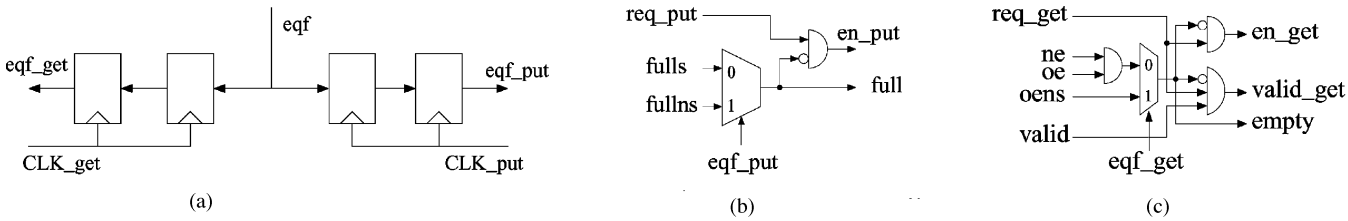

Fig. 10. New components for the dual-mode FIFO. (a) eqf synchronizer. (b) Dual-mode put controller. (c) Dual-mode get controller.

in three clock cycles, whereas with the basic mixed-clock FIFO, these two data items can be dequeued in two clock cycles.

F. Handling Variations in Clock Frequencies

The basic mixed-clock FIFO assumes that the clock frequencies for the put and get interfaces are different and un-correlated. However, if the two frequencies change over time, the mixed-clock FIFO cannot take advantage of this behavior and improve its performance. In this subsection, a simple extension to the basic design is presented. Now, the FIFO works in a *dual-mode*: in the *basic mode* (uncorrelated put/get frequencies), its behavior is exactly similar to that of the basic FIFO; in the *special mode*, the two clock frequencies are equal and have a zero phase, and FIFO greatly decreases its latency. Switching from one mode to the other is done by the environment, and signaled to the FIFO on an extra input signal (eqf), asynchronous to the two clock frequencies.

The basic FIFO can be modified into the dual-mode FIFO by changing only the put and get controllers, and by adding one extra simple component. These changes are shown in Fig. 10. The first component [Fig. 10(a)] simply synchronizes the globally asynchronous signal eqf to the two interfaces (put and get) through a pair of latches. The eqf signal is produced by the environment: when $\mathrm{eqf} = 0$, the two clock frequencies are different and uncorrelated, and the FIFO works in the basic mode; when $\mathrm{eqf} = 1$ the two clock frequencies are equal and with zero phase, and the FIFO works in the special mode.

The changes to the "put" and "get" controllers are straightforward. Compared with the old implementations (Fig. 8), the new implementations [Fig. 10(b)–(c)] contain one extra Mux. The Mux is controlled by the synchronized versions of eqf ($\mathrm{eqf}_{\mathrm{put}}$ and $\mathrm{eqf}_{\mathrm{get}}$, respectively). For the "put" controller, $\mathrm{eqf}_{\mathrm{put}}$ selects between the synchronized output of the full controller [*fulls* – after the pair of synchronizing latches Fig. 7(a)] and the unsynchronized output of the full detector [*fullns* – after the first synchronizing latch Fig. 7(a)]. For the "get" controller, $\mathrm{eqf}_{\mathrm{get}}$ selects between the output of the dual-mode empty detector and the unsynchronized output of the "normal empty" detector [*oens* – the output of the first synchronizing latch in Fig. 7(c)]. In both cases, the outputs of the Mux's are used as the global full/empty signal to control the put/get operations, respectively.

While in the special mode, the global "full"/"empty" signals are available to the respective interface one clock cycle later. Thus, when the FIFO becomes full, the put interface is stopped in the next clock cycle; when the FIFO becomes not full, the FIFO is able to enqueue a data item immediately in the next clock cycle. Similarly, when FIFO becomes empty, the get interface is stalled immediately; when the FIFO becomes not-empty, the get interface is able to dequeue a data item the very next clock cycle after the data item is enqueued. As a consequence, while in special mode, the latency of the FIFO is exactly *one clock cycle*, greatly improved when compared with the basic mode.

IV. Asynchronous–Asynchronous FIFO

The mixed-clock FIFO of the previous section interfaces two arbitrary synchronous domains operating under different clocks. The asynchronous–asynchronous FIFO is at the opposite extreme: the FIFO interfaces domains that do not have clocks (are asynchronous). This section discusses this asynchronous–asynchronous FIFO. The interfaces and the architecture of this FIFO have been presented in Section II. The remaining part (the design of an asynchronous–asynchronous FIFO cell) is now presented.

A. Asynchronous–Asynchronous FIFO Cell

The implementation of an asynchronous cell (Fig. 11) consists of several Burst-mode machines synthesized using the Minimalist CAD tool (implementing the Obtain Put Token (OPT) and Obtain Get Token (OGT) controllers) [17], [18][27], , a machine [data validity (DV) controller] synthesized using the Petrify CAD tool [13], two asymmetric-C elements [13], and one register. The cell's behavior is illustrated by tracing a put operation and then a get operation. Initially, the cell starts in an empty state ($\mathrm{valid} = 0$) and without tokens.

The cell enqueues a data item as follows. After a two transitions on *we1*, the put token is in the cell ($\mathrm{ptok} = 1$). When the environment requests a put operation ($\mathrm{put}_{\mathrm{req}} = 1$), *we* is asserted. This event causes several operations in parallel: the state of the cell is changed to full by $\mathrm{DV}_{\mathrm{as}}$, register *REG* is enabled to latch data, and the cell starts to send the put token to the left cell and to reset *OPT* ($\mathrm{ptok} = 0$). When $\mathrm{put}_{\mathrm{req}}$ is deasserted, *we* is then deasserted. This event completes the passing of the put token to the left cell. The cell is now prepared to start another put operation once the data in *REG* is dequeued.

The cell dequeues a stored data item in a similar fashion. After two transitions on *re1*, the get token is in the cell ($\mathrm{gtok} = 1$). The register then immediately outputs its data onto the global get data bus, *even before* the get interface requests it. When the environment does request a get operation ($\mathrm{get}_{\mathrm{req}} = 1$), *re* is asserted. This event causes the cell to acknowledge the get operation and to start sending the get token to the left cell. When $\mathrm{get}_{\mathrm{req}}$ is deasserted, *re* is deasserted. This event causes several operations in parallel: the cell completes the four-phase handshake on the get interface, it completes sending the put

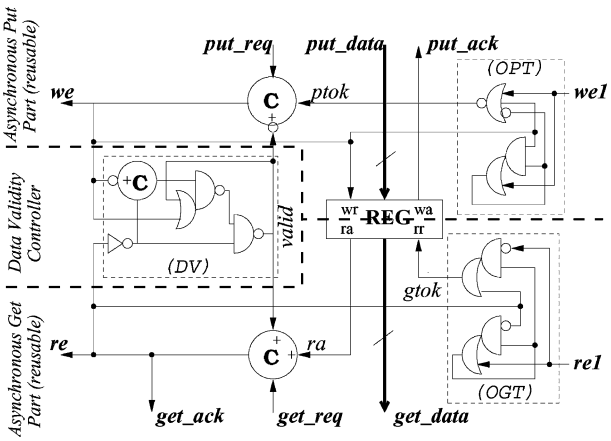

Fig. 11. Asynchronous–asynchronous FIFO cell implementation.

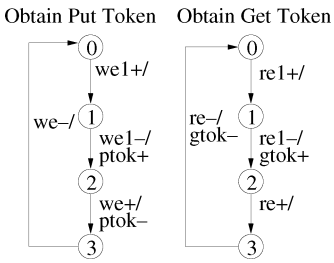

Fig. 12. Burst-mode specifications for OPT and OGT.

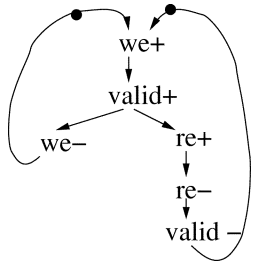

Fig. 13. STG specification for DV.

token, OGT is reset ($\mathrm{gtok} = 0$), and the data validity controller changes the state of the cell to "empty" ($\mathrm{valid} = 0$).

The asynchronous–asynchronous FIFO cell can be divided into three parts: a reusable *asynchronous put part* that deals with the put operation, a reusable *asynchronous get part* that deals with the get operation, and a *data validity controller*. The register is split into two parts, one belonging to the put part (the write port) and one belonging to the get part (the read port). The put and get parts will be reused in the design of the mixed asynchronous/synchronous FIFOs, while the data validity controller is unique to this FIFO.

The three controllers work as follows. OPT will enable PC by raising *ptok* as soon as the cell on the right has finished setting and then resetting *we1* which is the *write enable* signal from the right cell (i.e., once the right cell has latched its data, the current cell is enabled for the put operation). This pulse on *we1* therefore indicates a completed token passing. OPT will dessert *ptok* when the current cell starts to latch new data (*we* is high).

OGT is very similar; it is controlled by the *read enable* signals from the right cell (*re1*) and the current cell (*re*).

The BM specifications [17], [18], [27], [28] for OPT and OGT are shown in Fig. 12. A BM specification is a finite-state machine Mealy-type specification, which consists of a set of states, a set of arcs, and a unique starting state [28], [27]. An arc is labeled with an input burst (a set of transitions on the input signals), followed by an output burst (a set of transitions on the output signals, possibly empty). A BM machine waits for a complete input burst to arrive; transitions may come in any order and at any time. Once the complete input burst has arrived, the output burst is generated, and the machine moves to the next specification state. For example, for the OPT specification, the machine starts in state "0," and waits for a rising transition on *we1* ($\mathrm{we1}+$, where $+$ indicates a rising transition); once this arrives, the machine simply moves to state "1" since the output burst is empty. In state "1," the machine waits for the input burst $\mathrm{we1}-$ (where $-$ indicates a falling transition), after which it generates the output burst $\mathrm{ptok}+$. All BM controllers were synthesized with the MINIMALIST [17] CAD tool.

The DV controller has two inputs, and an output which indicates when data is valid. This output (called *valid*) is asserted when *we* (write enable) is asserted, and the output is deasserted when *re* is deasserted (after being previously asserted).

The signal transition graph (STG) specification [12], [13] for DV is shown in Fig. 13. STGs are interpreted Petri Net specifications, where each event corresponds to an actual signal transition. An STG specification consists of vertices and arcs. The vertices correspond to signal transitions on inputs and outputs. The arcs correspond to causal relations between transitions: a signal transition can fire (i.e., can be received if it's an input or can be produced if it's an output) iff all signal transitions with arcs leading to this signal transition have fired. In addition, an STG has a set of initial markings (the black dots in Fig. 13), which indicate which transitions are initially enabled to fire. For example, the DV specification indicates that, initially, only a rising transition on *we* can be received. Once this transition happens, DV generates $\mathrm{valid}+$ (a rising transition on *valid*), which, in turn, enables two other transitions ($\mathrm{we}-$ and $\mathrm{re}-$) to be received concurrently. The $\mathrm{we}-$ transition partially enables $\mathrm{we}+$, while $\mathrm{re}+$ leads to two more transitions: $\mathrm{re}-$ and $\mathrm{valid}-$; only after these last two transitions occur, $\mathrm{we}+$ is enabled again and the machine can resume its operation. The DV controller was synthesized using Petrify [13].

Both PC and GC are implemented with asymmetric C-elements: the output *we* of PC becomes 1 when every input is 1, but is reset when *put_req* and *ptok* are 0; the output *re* of GC will be 1 when all inputs are 1, but it will reset whenever *get_req* is 0.

There is one more optimization at the implementation level. It is desirable to drive the output get data bus as soon as possible due to the increased load on it. For this, the output of the register is enable to write data to the bus *as soon as* a cell has the get token, even if no *get_req* has been issued! The acknowledgment *ra* from the register is then used to enable GC. This is the reason why GC is implemented with an asymmetric C-element (otherwise deadlock might occur).

Timing Constraints: The above implementation of the asynchronous–asynchronous FIFO cell is not speed-independent

(SI) [19], [32]; that is, the FIFO may fail if the gates can have arbitrary gate delays (and assuming the wires have zero delay). There are two types of timing constraints that have to be met to make the implementation work correctly. The first category of timing constraints contains the *fundamental mode* timing constraints [32] for the OPT and OGT BM machines: the next input must arrive only after the circuit is stable. Similarly, DV is not speed independent because it was synthesized with the .slowenv Petrify option, which models *fundamental mode* constraints. These timing constraints are very easily met. Increasing the number of cells does not affect the timing constraints since they are localized in the controllers that communicate only with the adjacent cells.

The circuit also presents a *pulse-width* timing constraint on *we*. The pulse width of *we* high must be greater than the time for machine DV to process the *we* high input. The timing constraint, expressed in terms of critical paths, is: $\delta_{\mathrm{we}\uparrow \rightarrow \mathrm{valid}\uparrow \mathrm{feedback}} < \delta_{\mathrm{we}\uparrow \rightarrow \mathrm{put\_ack}\uparrow \rightarrow \mathrm{put\_req}\downarrow \rightarrow \mathrm{we}\downarrow}$. The constraint is easily met since the longer delay involves a path through the environment, as well as a reasonably long internal cell path (several gates). In fact, as the number of FIFO cells increases, it is easier to meet this constraint because the path through the environment becomes longer.

The design has been formally verified for *conformance equivalence* using the trace theory verifier AVER [16]. The timing constraints are modeled as constraints on the environment: the environment is observing the internal signals in the implementation and is producing an event only after the trace corresponding to the shortest path in the timing constraint has occurred.

V. Mixed Asynchronous/Synchronous FIFOs

Each of the previous two FIFO, i.e., synchronous–synchronous and asynchronous–asynchronous, have a very modular design. As a consequence, the asynchronous–synchronous and synchronous–asynchronous FIFOs can be easily built by reusing parts of the synchronous–synchronous and asynchronous–asynchronous FIFOs: the only part that changes in the new FIFOs is one small component in each cell implementation.

At the architectural level, the previous FIFOs consist of two reusable parts: a put interface and a get interface. The two reusable parts may be combined to design a new mixed-timing FIFO: combining the synchronous put interface with the asynchronous get interface results in a synchronous–asynchronous FIFO; combining the asynchronous put interface with the synchronous get interface results in an asynchronous–synchronous FIFO. In particular, for the synchronous interfaces, the external put/get controllers and the full/empty detectors are reused without any change.

Interestingly, each existing FIFO cell also can be divided into three distinct parts: a reusable *put part* that deals with the put operation, a reusable *get part* that deals with the get operation, and a *data validity* (*DV*) *controller*. Therefore, cells for the two new FIFOs can be designed by simply combining the appropriate put/get reusable parts, and designing only a *new DV* controller. For example, an asynchronous–synchronous FIFO cell is build

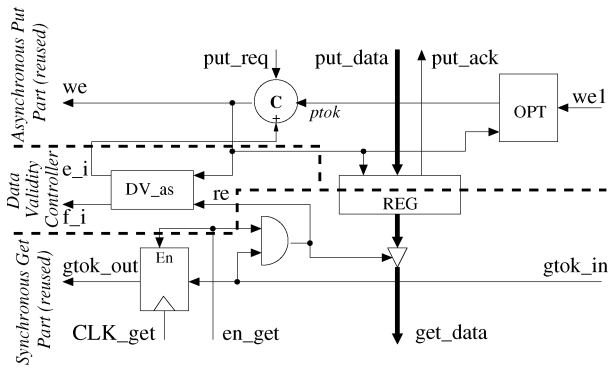

Fig. 14. Asynchronous–synchronous cell implementation.

by combining the asynchronous put part (Fig. 11) of the asynchronous–asynchronous FIFO cell and the synchronous get part (Fig. 6) of the mixed-clock FIFO.

In the following, each mixed asynchronous/synchronous FIFO cell is discussed in turn. Since most of the components in the cell's implementation have been already presented, only the data validity controller (unique for each FIFO type) is presented in detail.

A. Asynchronous–Synchronous FIFO

The asynchronous–synchronous FIFO interfaces asynchronous and synchronous domains, where the asynchronous domain is the sender and the synchronous domain is the receiver. The asynchronous–synchronous FIFO is built from reusable interfaces and components from the two previous basic FIFOs. Only one small component in each cell is unique to this type of FIFO, and will be discussed in more detail.

At the interface level, the asynchronous–synchronous FIFO consists of the asynchronous put interface and the synchronous get interface (Fig. 2). Architecturally, the FIFO reuses the asynchronous put part of the mixed-clock FIFO [Fig. 3(b)] and the synchronous get part [Fig. 3(a)] of the asynchronous–asynchronous FIFO. In particular, the empty detector and the get controller are reused without any change.

The basic cell of the asynchronous–synchronous FIFO (Fig. 14) consists of the asynchronous put part of the asynchronous–asynchronous FIFO cell (Fig. 11), the synchronous get part of the mixed-clock FIFO cell (Fig. 6), and a new data validity controller.

The new data validity controller ($\mathrm{DV_{as}}$) indicates when the cell contains a data item; it thus controls the put and get operations. It accepts as inputs *we* (which signals that a put operation is taking place) and *re* (which signals that a get operation is taking place). Its outputs are e_i (indicating the cell is empty, allowing the next put operation), and f_i (indicating the cell is full – used by the empty detector).

The protocol for $\mathrm{DV_{as}}$ is shown as an STG [13] in Fig. 15. Once a put operation begins, $\mathrm{DV_{as}}$ sets e_i to zero (thus declaring the cell not empty), and f_i to one (thus enabling a get operation). After a get operation begins ($re+$), the cell is declared "not full" ($f_i = 0$) asynchronously, in the middle of the $\mathrm{CLK_{get}}$ clock cycle. When the get operation finishes (on the next positive edge of $\mathrm{CLK_{get}}$), $\mathrm{DV_{as}}$ sets the cell to "empty" ($e_i = 1$) and the behavior can resume. This asymmetric behavior prevents data

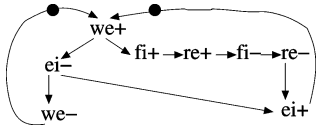

Fig. 15. STG specification of $\mathrm{DV_{as}}$.

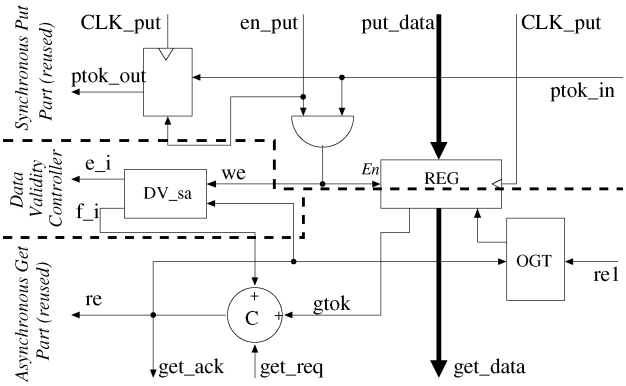

Fig. 16. Synchronous–asynchronous cell implementation.

corruption by a put operation while a get operation is still taking place.

B. Synchronous–Asynchronous FIFO

The synchronous–asynchronous FIFO interfaces synchronous and asynchronous domains, where the synchronous domain is the sender and the asynchronous domain is the receiver. The synchronous–asynchronous FIFO is built from reusable interfaces and components from the two previous basic FIFOs. Only one small component in each cell is unique to this type of FIFO, and will be discussed in more detail.

At the interface level, the synchronous–asynchronous FIFO consists of the synchronous put interface and the asynchronous get interface (Fig. 2). Architecturally, the FIFO reuses the synchronous put part of the mixed-clock FIFO [Fig. 3(a)] and the asynchronous get part [Fig. 3(b)] of the asynchronous–asynchronous FIFO. In particular, the full detector and the put controller are reused without any change.

The basic cell of the synchronous–asynchronous FIFO (Fig. 16) consists of the synchronous put part of the mixed-clock FIFO cell (Fig. 6), the asynchronous get part of the asynchronous–asynchronous FIFO cell (Fig. 11), and a new data validity controller.

The new data validity controller ($\mathrm{DV_{sa}}$) indicates when the cell contains a data item; it thus controls the put and get operations. It accepts as inputs *we* (which signals that a put operation is taking place) and *re* (which signals that a get operation is taking place). Its outputs are e_i (indicating the cell is empty – used by the full detector), and f_i (indicating the cell is full, allowing the next get operation).

The protocol for $\mathrm{DV_{sa}}$ is shown as an STG [13] in Fig. 17. Once a put operation begins, $\mathrm{DV_{sa}}$ sets e_i to zero (thus declaring the cell not empty), and f_i to one (thus enabling a get operation). After a get operation completes ($re+$ followed by $re-$), the cell is declared not full ($f_i = 0$ and $e_i = 1$). In parallel with the get operation, the cell completes the put operation (we−). After the put operation completes *and* the cell becomes empty, a put operation may start again. This asymmetric behavior prevents data corruption by a put operation while a get operation is still taking place.

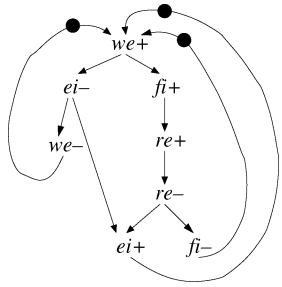

Fig. 17. STG specification of $\mathrm{DV_{as}}$.

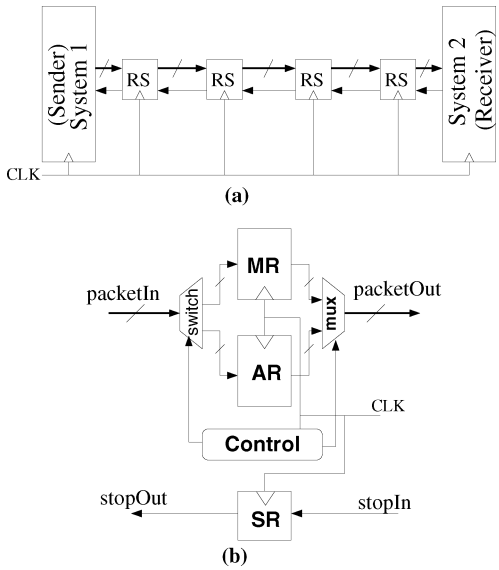

Fig. 18. Single-clock relay stations: (a) insertion of relay station between systems and (b) implementation of a relay station.[3].

VI. FIFO as a Relay Station

This paper has so far presented a set of basic solutions for interfacing mixed-timing domains. Now, the issue of long communication delays on wires between mixed-timing subsystems is also addressed. A previous, single-clock solution (called *relay stations*) to this problem [3] is presented in Section VI-A, since it constitutes the basis of the new mixed-timing relay stations. These new designs are then presented in Section VI-B–D.

A. Single-Clock Relay Stations

Relay stations were introduced to alleviate the connection delay penalties between two subsystems operating under the same clock [Fig. 18(a)]. After placement, the systems may be connected by very long wires, on which a signal takes several clock cycles to travel. The solution is to break the long wires into segments corresponding to clock cycles, and then insert a chain of relay stations which act like a FIFO sending packets from one subsystem to another.

The implementation of a relay station is given in Fig. 18(b). Normally, the packets from the left relay station are passed to the right relay station. The right relay station also has the possibility

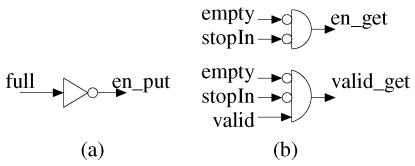

Fig. 19. The control for the relay station FIFO. (a) Put controller. (b) Get controller.

to put counter pressure on the data flow by stopping the relay stations to the left. Each relay station has two registers: one is used in normal operation and one used to store an extra packet when stopped.

A relay station works as follows. In normal operation, at the beginning of every clock cycle, the data packet received on *packetIn* from the left relay station is copied to main register (MR) and then forwarded on *packetOut* to the right relay station. A packet consists of a data item and a valid bit which indicates the validity of the data in the packet. If the receiver system wants to stop receiving data, it raises *stopIn*. On the next clock edge, the relay station raises *stopOut* and latches the next packet to the auxiliary register. When the relay station is unstalled, it will first send the packet from the main register to the right, and then the one from the auxiliary register.

B. Mixed-Timing Relay Stations

The new mixed-timing FIFOs can now be modified into mixed-timing relay stations (MCRS).

There are several differences between the operation of a FIFO and that of a relay station. In contrast to the basic mixed-clock FIFO, the new MCRS *always* passes valid data items from left to right: there are no active requests on either interface. Instead, the get and put interfaces can only actively stop, or interrupt, the continuous flow of data items. The get interface reads a data item from the FIFO on every clock cycle; its only possibility to stop the flow is to assert *stopIn*. Similarly, the FIFO always enqueues data items from the put interface. Enqueued data may now be either valid or invalid. Thus, unlike previous designs, $\mathrm{req}_{\mathrm{put}}$ is used *solely* to indicate data validity, being treated as part of *packetIn* and not as a control signal. When it becomes full, the MCRS simply stops the put interface: the *full* signal is used as *stopOut* to the left interface.

The mixed-clock relay station is derived from the mixed-clock FIFO through a process of *renaming* and *hiding* wires on the interfaces. On the put interface, the FIFO interface wires are renamed as follows. The $\mathrm{req}_{\mathrm{put}}$ wire becomes the validity bit in the incoming packet. The *full* signal is renamed to *stopOut*. On the get interface, the *empty* signal is no longer exported. The rest of the wires are renamed as follows: $\mathrm{req}_{\mathrm{get}}$ becomes *stopIn*, while $\mathrm{valid}_{\mathrm{get}}$ becomes the validity bit for the outgoing packet.

Internally, the mixed-clock relay station is built from the mixed-clock FIFO by changing *only* the put and get controllers (Fig. 19). In the mixed-clock FIFO, the put controller enables the enqueueing of valid data items using $\mathrm{req}_{\mathrm{put}}$ signals, while in the relay station design it simply allows valid data items to pass through. In addition, the new put controller continuously enqueues data items unless the FIFO becomes full. Thus, the put controller is simply an inverter [see Fig. 19(a)]. In a similar fashion, the new get controller enables continuous dequeuing of data items, unlike the mixed-clock FIFO where dequeuing was done on demand. Dequeuing can only be interrupted if the FIFO becomes empty or the get interface signals it can no longer accept data items by asserting *stopIn*.

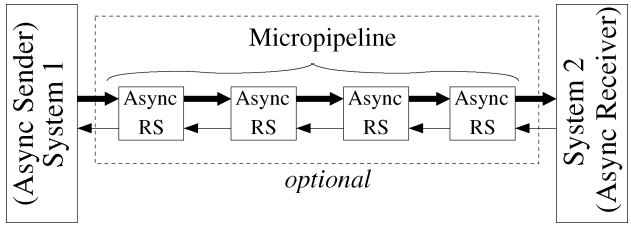

Fig. 20. Asynchronous relay stations.

C. Asynchronous Relay Stations

In principle, *no relay stations* need to be inserted in the asynchronous communication channels (Fig. 20): the two asynchronous domains can communicate directly over long wires. However, in practice, the designer needs to address both 1) correctness and 2) performance issues in their designs, so inserting asynchronous relay stations (ARS) may be desirable.

There are two common asynchronous communication styles: dual-rail and single-rail bundled data [19]. The dual-rail style encodes both the value and validity of each data bit on a pair of wires; therefore, communication between systems is robust with respect to arbitrary wire delays, so that no relay stations are needed. However, to meet performance goals, insertion of ARSs may be desirable to increase the throughput. The second communication style (single-rail bundled data) has timing assumptions between the single-rail data itself and the worst-case bundling control wire (called a "bundling constraint"). In this case, a chain of ARSs may be desirable (as in Carloni) to limit the wire lengths between stages to short hops. Also, as in the dual-rail case, inserting ARSs can increase the throughput on the interfaces.

A chain of asynchronous relay stations can be directly implemented by using a standard asynchronous FIFO called a *micropipeline* [36], [39]. Unlike the synchronous data packets, the asynchronous ones do not need a validity bit: the presence of valid data packets is signaled on the control wires and an ARS can wait indefinitely between receiving data packets. Therefore, a micropipeline implements the desired ARS behavior.

D. Mixed Asynchronous–Synchronous Relay Stations

Finally, two novel relay station variants are presented in this section: relay stations which handle mixed asynchronous–synchronous interfaces. These designs are the first to be proposed which simultaneously solve both critical design challenges: mixed asynchronous–synchronous interfaces and long interconnect delays.

Basic Architecture: Communication with relay stations between asynchronous and synchronous domains is shown in Fig. 21. An asynchronous domain sends data packets (possibly through a chain of asynchronous relay stations) to the ASRS. The packets are then transferred to the synchronous domain, and sent through a chain of SRS to the receiver [Fig. 21(a)].

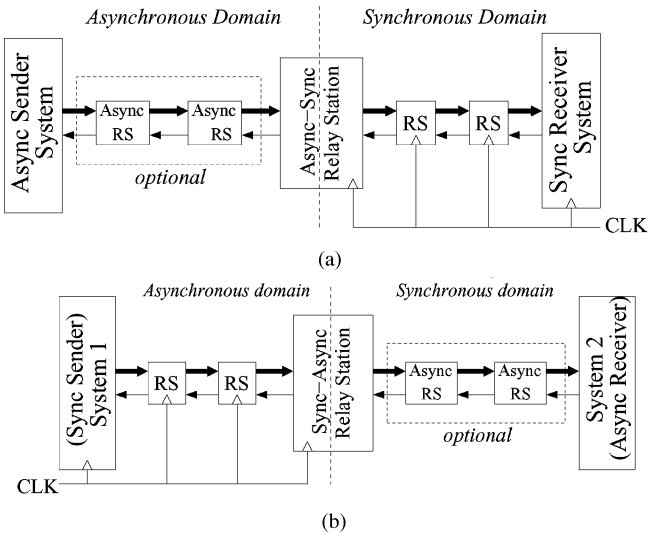

Fig. 21. Mixed asynchronous–synchronous relay stations. (a) Asynchronous–synchronous relay station. (b) Synchronous–asynchronous relay station.

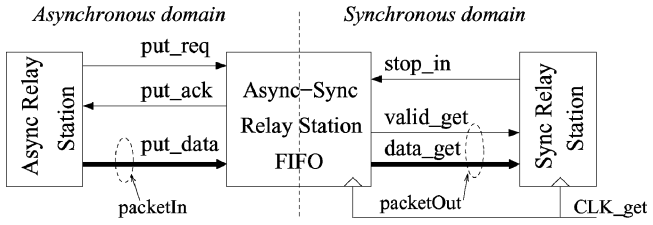

Fig. 22. Interfaces for the ASRS.

Similarly, a synchronous domain sends data packets to the SARS through a chain of SRS. The packets are then transferred to the asynchronous domain and sent (possibly through a chain of asynchronous relay stations) to the receiver [Fig. 21(b)].

The implementations of the two new types of relay stations (asynchronous–synchronous and synchronous–asynchronous) are now discussed in turn.

ASRS: Implementation: The ASRS can be derived by modifying the asynchronous–synchronous FIFO. The new interfaces are shown in Fig. 22. Note that the asynchronous interface is identical and supports the same communication protocol as the asynchronous interface in the FIFOs counterpart. This holds because these interfaces exactly match the micropipeline interfaces. There is no need for an explicit validity bit since data is enqueued only when requested. However, the synchronous interface supports the same communication protocol with the respective synchronous interface as in the mixed-clock relay station, including a data validity bit.

At the architectural level, the ASRS reuses unmodified most of the components of the asynchronous–synchronous FIFO. In fact, the only changes in the ASRS design are in the *get controller* (see below).

The ASRS operates as follows. At its asynchronous interface, the relay station only enqueues data items when they are present. However, on the synchronous interface, the ASRS must output a data item on every clock cycle (either valid, or invalid when it is empty), unless it is stopped by the right relay station. Thus,

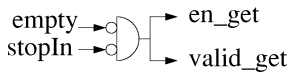

Fig. 23. New ASRS get controller.

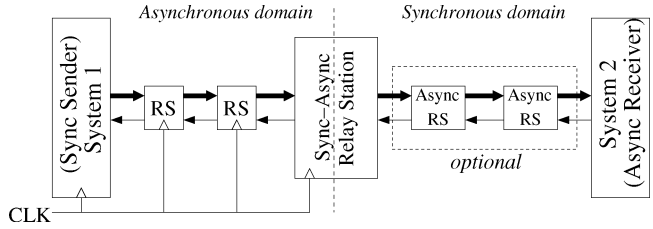

Fig. 24. Interfaces for the SARS.

the right interface receives valid data packets except when the ASRS is empty.

The implementation for the ASRS get controller is shown in Fig. 23. The get controller enables an explicit get operation ($\mathrm{en_{get}} = 1$) when it is not stopped from the ($\mathrm{stopIn} = 0$) and when the relay station is not empty ($\mathrm{empty} = 0$) (as in the mixed-clock relay station). The data packet validity signal, $\mathrm{valid_{get}}$, is invalid if either the relay station is empty or it is stopped.

SARS: Implementation: The SARS can be derived from the asynchronous–synchronous FIFO. The new interfaces are shown in Fig. 24. Notice that the asynchronous interface is identical and supports the same communication protocol as the asynchronous get interface of the asynchronous–asynchronous FIFO. Also, the synchronous interface is identical with the synchronous interface of the mixed-clock relay station.

At the architectural level, the SARS reuses unmodified most of the components of the asynchronous–synchronous FIFO. The only part that changes is the put controller. The changes in the put controller are exactly the same as for the mixed-clock relay station: the two synchronous interfaces support the exact same protocol. Thus, the implementation of the put controller [Fig. 19(a)] in the mixed-clock relay station can be reused in this design without any modifications.

E. Optimized Relay Station Implementations

The basic mixed-timing relay stations are derived from their FIFO counterparts by reusing most of the FIFO components. In fact, these relay stations are designed through simple changes on the interfaces (renaming and eliminating interface wires) and small modifications in the put and get controllers.

The basic mixed-timing relay station designs can be further optimized for area and for power. These optimizations are enabled by the relay station's communication protocol. First, the "latency insensitive" nature of communication enables area saving by simplifying the empty detectors in the new relay stations. Second, the presence of both valid and invalid data packets in communication allows for power savings: only the valid data packets can now be enqueued in the relay station, while the invalid data packets are simply ignored.

In the following, the two optimizations are discussed for the mixed-clock relay station. However, it is important to notice that these optimizations can be applied to the mixed asynchronous/synchronous relay stations without any change.

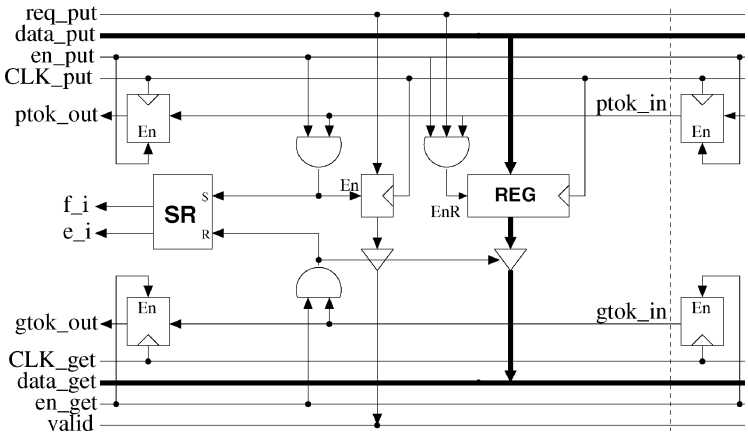

Fig. 25. Optimized mixed-clock relay station cell.

1) Optimization for Area: The latency-insensitive protocol that the mixed-clock relay station obeys allows for reductions in area. The intuition is as follows. The mixed-clock relay station receives data packets every clock cycle (unless full). Therefore, when the relay station is *empty* there is no danger of deadlock: the receiver domain constantly sends data packets and the get interface is eventually started. As a consequence, the relay station *does not need* a bi-modal empty detector: the new empty detector suffices to ensure correct relay station operation. The area reduction comes from eliminating the detector for normal empty.

The correctness of the computation in the two domains (sender and receiver) is not affected by this optimization. However, when the relay station becomes empty, the observable trace of data packets is now changed. When using the bi-modal empty detector, the latency of a valid data packet through an empty relay station is one sender clock cycle (to enqueue the data packet) and two receiver's clock cycles (to synchronize the global empty). When using just the new empty detector, the same latency is *two* sender clock cycles (to enqueue *two* data packets) and two receiver's clock cycles (synchronization of global empty). Thus, with the proposed optimization, the receiver domain receives one extra invalid data packet, before starting to receive the data packets from the sender.

2) Optimization for Power: In normal communication, the mixed-clock relay station passes both valid and invalid data packets. However, the invalid data packets are simply discarded by the receiver domain. Therefore, the exact value of a data item in an invalid data packet is irrelevant. The mixed-clock relay station can then simply *not latch* invalid data items, thus allowing for power savings. It is important to notice that the validity bit in an invalid data packet still needs to be latched; otherwise, the receiver domain might incorrectly receive invalid data packets falsely labeled as valid.

At the architectural level, the only change in the mixed-clock relay station occurs in the basic cell implementation (Fig. 25). This implementation is derived from the mixed-clock FIFO cell implementation. The only difference between the two is in the *synchronous put part*: the register is now split in two parts, each part controlled by a separate AND gate. The first part, consisting of a single ETDFF, enqueues the validity bit whenever the put interface is enabled to enqueue ($\mathrm{en_{put}} = 1$) and the cell has the put token ($\mathrm{ptok_{in}} = 1$). The second part, consisting of n ETDFFs (where n is the width of a data item), enqueues a data item when the put interface is enabled to enqueue, when the cell has the put token, and when *the data packet is valid* ($\mathrm{req_{put}} = 1$). Thus, whenever an invalid data packet is to be enqueued in the relay station, the cell with the put token overwrites only the validity bit, leaving the stale value of the data item unchanged.

The put part in the optimized mixed-clock relay station cell can be reused without any change in the SARS.

VII. Results

To validate the new FIFOs, each design was simulated using both commercial and academic tools. The circuits were designed using both library and custom components and simulated using Cadence HSPICE in 0.6-μm HP CMOS technology, at 3.3 V and 300 K. Burst-mode controllers were synthesized using MINIMALIST [17] and the Petri-Net controller was synthesized using Petrify [13].

Since all simulations are pre-layout, special care was taken in modeling the global control and data buses. Appropriate buffering was inserted into the control buses $\mathrm{put_{req}}/\mathrm{en_{put}}$ and $\mathrm{get_{req}}/\mathrm{en_{get}}$. In addition, $\mathrm{put_{ack}}$ was generated through a tree of OR-gates that merges individual acknowledgments into a single global one. For the $\mathrm{get_{data}}$ tristate buses, wiring and environmental delays were modeled by inserting capacitive loads.

Two metrics have been simulated for each design: *latency* and *throughput* (Table I). Latency is the delay from the input of data on the put interface to its appearance at the output on the get interface in an empty FIFO. Throughput is defined as the reverse of the cycle time for a put or get operation, and it is reported in megahertz for the synchronous interface, and megaoperations per second for the asynchronous interfaces. The throughput results are consistent with the FIFO designs. The synchronous get interfaces are slower than the synchronous put interface because of the complexity of empty detector. The asynchronous interfaces are slower than their synchronous counterparts because of the increased complexity of the asynchronous put protocol.

The experimental setup for latency is as follows: in an empty FIFO, the get interface requests a data item. After the FIFO is stable, the put interface places a data item. The latency is computed as the elapsed time between the moment when the put data

TABLE I
SIMULATION RESULTS

FIFO Type	Throughput				Latency (ns)			
	4-place		16-place		4-place		16-place	
	put	get	put	get	Min	Max	Min	Max
Mixed-Clock	565	549	505	484	5.43	6.34	6.14	7.17
Async-Async	423	454	359	367	1.73		2.29	
Async-Sync	421	549	357	484	5.53	6.45	6.47	7.51
Sync-Async	565	454	505	360	1.95		2.44	
Mixed-Clock RS	580	539	509	475	5.48	6.41	6.23	7.28
Async-Sync RS	421	539	357	475	5.48	6.41	6.23	7.28

bus has valid data to the moment when the get interface retrieves the data item and can use it. Latency for a FIFO with a synchronous receiver is not uniquely defined (see also Section III-D for a detailed discussion). Latency varies with the exact moment when data items are safely enqueued in a cell. If the data item is enqueued by the put interface immediately after the positive edge of $\mathrm{CLK_{get}}$, then latency is increased (column *Max* in the table). If the data item is enqueued right before the empty detector starts computation, then latency is decreased (column *Min*).

VIII. CONCLUSIONS

This paper presents several low-latency mixed-timing FIFO designs that interface systems on a chip working at different speeds. The connected systems can be either synchronous or asynchronous. These designs are also adapted to work between systems with very long interconnect delays, by migrating a single-clock solution by Carloni *et al.* (for "latency-insensitive" protocols) to mixed-timing domains; this is the first proposed solution to handling both mixed-timing domains and long latencies between these domains. The new designs can be made arbitrarily robust with regard to metastability. Initial simulations for both latency and throughput are promising. Taken together, the seven FIFO designs presented in this paper (four mixed-timing FIFOs and three relay stations) provide a powerful and effective set of solutions for the challenges of future systems-on-a-chip.

Recently, our proposed mixed-clock FIFOs have been used in several extensive studies of GALS architectures [23], [22], [34]. Iyer and Marculescu [23] have incorporated latency and power models of the mixed-clock FIFO into the Wattch simulator [2], and have demonstrated that a GALS implementation of an Alpha 21 264 processor with multiple clock domains: 1) does not show a better power/performance cost when simply introducing our FIFOs in the communication between domains, but 2) when the multiple-clock implementation is augmented with voltage scaling, this cost function for the GALS design is vastly improved over the single-clock implementation. Hassoun and Alpert [22] have proposed a novel algorithm for optimal routing with minimum latency between two mixed-clock domains; they have used the proposed mixed-clock FIFO in the estimations of communication overheads. Finally, in [34], the proposed set of FIFOs is used to evaluate the performance of a novel algorithm for the dynamic control of the frequency and voltage of a multiple clock domain architecture.

In the future, the authors want to pursue two main directions. First, the authors want to write synthesizable Verilog models for each of the FIFO designs presented in this paper, and adapt the mixed-clock FIFO to a standard communication protocol (Open Core Protocol [29]) for on-chip communication between mixed-timing domains. Second, the authors are very interested in integrated the presented designs with the lower-latency solutions of [6]. This integration will prove beneficial for both design families: the latency of our proposed designs can be significantly reduced, while the family of designs from [6] can be adapted to handle any combination of asynchronous/synchronous domains, as well as to reduce its latency for steady–state communication.

ACKNOWLEDGMENT

The authors acknowledge useful discussions with L. Carloni and Prof. A. Sangiovanni-Vincentelli at University of California at Berkeley.

REFERENCES

[1] D. S. Bormann and P. Y. K. Cheung, "Asynchronous wrapper for heterogeneous systems," in *Proc. Int. Conf. Computer Design (ICCD'97)*, 1997, pp. 307–314.
[2] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *Proc. Int. Symp. Computer Architecture (ISCA'00)*, 2000, pp. 83–94.
[3] L. Carloni, K. McMillan, A. Saldanha, and A. Sangiovanni-Vincentelli, "A methodology for correct-by-construction latency insensitive design," in *Proc. Int. Conf. Computer Aided Design (ICCAD'99)*, 1999, pp. 309–315.
[4] L. P. Carloni and A. L. Sangiovanni-Vincentelli, "Performance analysis and optimization of latency-insensitive systems," in *Proc. Design Automation Conf. (DAC'00)*, 2000, pp. 361–367.
[5] L. P. Carloni, K. L. McMillan, and A. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Trans. Computer-Aided Design*, vol. 20, pp. 1059–1076, Sept. 2001.
[6] A. Chakraborty and M. R. Greenstreet, "Efficient self-timed interfaces for crossing clock domains," in *Proc. 9th IEEE Int. Symp. Asynchronous Circuits Systems (ASYNC'03)*, 2003, pp. 78–88.
[7] D. M. Chapiro, "Globally-asynchronous locally-synchronous systems," Ph.D. dissertation, Stanford University, Stanford, CA, 1984.
[8] W. S. Coates and R. J. Drost, "Congestion and starvation detection in ripple FIFO's," in *Proc. 9th IEEE Int. Symp. Asynchronous Circuits and Systems (ASYNC'03)*, 2003, pp. 36–45.
[9] T. Chelcea and S. Nowick, "Low-latency asynchronous FIFO's using token rings," in *Proc. IEEE Symp. Asynchronous Circuits and Systems (ASYNC'00)*, 2000, pp. 210–220.
[10] ——, "A low-latency FIFO for mixed-clock systems," in *Proc. IEEE Workshop on VLSI'00*, 2000, pp. 119–128.
[11] ——, "Robust interfaces for mixed-timing systems with application to latency-insensitive protocols," in *Proc. Design Automation Conf. (DAC'01)*, 2001, pp. 21–26.
[12] T.-A. Chu, "Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications," Ph.D. dissertation, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge, MA, 1987.
[13] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *IEICE Trans. Inform. Syst.*, vol. E80-D, no. 3, pp. 315–325, Mar. 1997.
[14] W. J. Dally and J. W. Poulton, "Digital systems engineering," in *Synchronization*. Cambridge, U.K.: Cambridge Univ. Press, 1998, ch. 10, pp. 462–513.
[15] G. de Micheli, "Assembling and SOC: Communication architectures and protocols," presented at the Design Automation Conf. (DAC'03), Tutorial Session, Anaheim, CA, 2003.
[16] D. Dill, S. M. Nowick, and R. Sproull, "Specification and automatic verification of self-timed queues," *Formal Methods Syst. Des.*, vol. 1, pp. 29–60, 1992.

[17] R. M. Fuhrer, S. M. Nowick, M. Theobald, N. K. Jha, B. Lin, and L. Plana, "*MINIMALIST: An environment for synthesis, verification and testability of burst-mode asynchronous machines*," Columbia Univ., Dept. of Computer Science, New York, Tech. Report CUCS-020–99, 1999.
[18] R. M. Fuhrer and S. M. Nowick, *Sequential Optimization of Asynchronous and Synchronous Finite-State Machines: Algorithms and Tools*. Norwell, MA: Kluwer , 2001.
[19] S. B. Furber, "Asynchronous design," in *Proc. Submicron Electronics*, Il Ciocco, Italy, 1997, pp. 461–492.
[20] R. Ginosar, "Fourteen ways to fool your synchronizer," in *Proc. 9th IEEE Int. Symp. Asynchronous Circuits and Systems (ASYNC'03)*, 2003, pp. 89–97.
[21] M. R. Greenstreet, "Implementing a STARI chip," in *Proc. Int. Conf. Computer Design (ICCD'95)*, pp. 38–43.
[22] S. Hassoun and C. J. Alpert, "Optimal path routing in single and multiple clock domain systems," *IEEE Trans. Computer-Aided Design*, vol. 22, pp. 1580–1588, Nov. 2003.
[23] A. Iyer and D. Marculescu, "Power-performance evaluation of globally asynchronous, locally synchronous processors," in *Proc. 29th Int. Symp. Computer Architectures*, 2002, pp. 158–168.
[24] J. Jex, C. Dike, and K. Self, "Fully asynchronous interface with programmable metastability settling time synchronizer," Patent 5 598 113, 28, 1997.
[25] R. Kol and R. Ginosar, "Adaptive synchronization for multi-synchronous systems," in *1998 IEEE Int. Conf. Computer Design (ICCD'98)*, Oct. 1998, pp. 188–189.
[26] S. Moore, G. Taylor, R. Mullins, and P. Robinson, "Point to point GALS interconnect," in *Proc. 8th Int. Symp. Asynchronous Circuits and Systems (ASYNC'02)*, 2002, pp. 69–75.
[27] S. M. Nowick and D. L. Dill, "Synthesis of asynchronous state machines using a local clock," in *Proc. Int. Conf. Computer Design (ICCD'91)*, 1991, pp. 192–197.
[28] S. M. Nowick, "Automatic Synthesis of Burst-Mode Asynchronous Controllers," Stanford Univ., Tech. Report CSL-TR-95–686, 1993.
[29] Open Core Protocol Home Page (2004). [Online]. Available: http://www.ocpip.org/home
[30] G. N. Pham and K. C. Schmitt, "A high throughput, asynchronous, dual port FIFO memory implemented in ASIC technology," in *Proc. Annual IEEE Int. ASCI Conf. and Exhibition*, 1989, pp. P3-1.1–1.4.
[31] L. F. G. Sarmenta, G. A. Pratt, and S. A. Ward, "Rational clocking," in *Proc. ICCD'95*, Oct. 1995, pp. 217–228.
[32] C. L. Seitz, *System Timing, Introduction to VLSI Systems*. Reading, MA: Addison-Wesley, 1980, ch. 7.
[33] J. Seizovic, "Pipeline synchronization," in *Proc. IEEE Symp. Asynchronous Circuits and Systems (ASYNC'94)*, 1994, pp. 87–96.
[34] G. Semeraro, D. H. Albonesi, S. G. Dropsho, G. Magklis, S. Dwarkadas, and M. L. Scott, "Dynamic frequency and voltage control for a multiple clock domain microarchitecture," in *Proc. 35th Int. Symp. Microarchitecture*, Nov. 2002, pp. 356–367.
[35] Y. Semiat and R. Ginosar, "Timing measurements of synchronization circuits," in *Proc. 9th IEEE Int. Symp. Asynchronous Circuits and Systems (ASYNC'03)*, 2003, pp. 68–77.
[36] M. Singh and S. M. Nowick, "MOUSETRAP: Ultra-high-speed transition-signaling asynchronous pipelines," in *Proc. IEEE Int. Conf. Computer Design (ICCD'01)*, Austin, TX, 2001.
[37] M. Singh and M. Theobald, "Generalized latency-insensitive systems for single-clock and multi-clock architectures," in *Proc. Design, Automation and Test Eur. (DATE'04)*, Mar. 2004, pp. 1008–1013.
[38] A. E. Sjogren and C. J. Myers, "Interfacing synchronous and asynchronous modules within a high-speed pipeline," *IEEE Trans. VLSI Syst.*, vol. 8, pp. 573–583, Oct. 2000.
[39] I. E. Sutherland, "Micropipelines," *Commun. ACM*, vol. 32, no. 6, pp. 720–738, June 1989.
[40] D. Wingard, "Micronetwork-based integration for SOC's," in *Proc. Design Automation Conf. (DAC'01)*, 2001, pp. 673–677.
[41] K. Y. Yun and R. P. Donohue, "Pausible clocking: A first step toward heterogeneous systems," in *Proc. Int. Conf. Computer Design (ICCD'96)*, 1996, pp. 118–123.

Tiberiu Chelcea received the B.S. and M.S. degrees from the Politehnica University of Bucharest, Romania, in 1995 and 1996, respectively, and the Ph.D. degree in computer science from Columbia University, New York, in 2004.

He is currently a Postdoctoral Fellow with the Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA. His research interests include asynchronous circuits, design techniques for mixed-timing digital systems, synthesis methods from high-level languages down to asynchronous systems.

Steven M. Nowick received the B.A. degree from Yale University, New Haven, CT, and the Ph.D. degree in computer science from Stanford University, Stanford, CA, in 1993. His Ph.D. dissertation introduced an automated synthesis method for locally-clocked asynchronous state machines.

Currently, he is an Associate Professor of Computer Science and Electrical Engineering at Columbia University, NY. His general research interests include asynchronous circuits, computer-aided digital design (CAD), low-power and high-performance digital systems, and logic synthesis. His current research is on CAD tools, as well as design methods, for asynchronous and mixed-timing digital systems.

Dr. Nowick received a National Science Foundation Faculty Early Career (CAREER) Award in 1995, an Alfred P. Sloan Research Fellowship in 1995, and a National Science Foundation Research Initiation Award (RIA) in 1993. He received Best Paper Awards at the 1991 International Conference on Computer Design and at the 2000 IEEE Asynchronous Symposium. He was also the recipient of two medium-scale National Science Foundation ITR Awards in 2000. He is currently an Associate Editor of the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS and IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN of ICAS. In 2002, he was Program Chair of the ACM International Workshop on Logic and Synthesis. He was Program Cochair of the IEEE Async Symposium in 1994 and 1999, and will be General Chair in 2005.