Robust Interfaces for Mixed-Timing Systems with Application to Latency-Insensitive Protocols

Tiberiu Chelcea Steven M. Nowick*
Department of Computer Science
Columbia University
e-mail: {tibi,nowick}@cs.columbia.edu

Abstract

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. The design are then adapted to work between systems with very long interconnection delays, by migrating a single-clock solution by Carloni et al. (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.

1. Introduction

Future VLSI systems will likely be systems-on-a-chip involving many clock domains. 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 FIFO's which robustly accommodate mixed-timing systems.

There are two fundamental challenges in designing systems-on-a-chip: systems operating under different timing assumptions, and long delays in communicating between systems. This paper addresses both of these issues. First, two *basic mixed-timing FIFO's* are introduced which address the first design challenge, for both sync-sync and async-sync interfaces. For the second design challenge, "latency-insensitive protocols" were proposed by Carloni et al. [2]; however, their solution was limited to a single clock domain. In this paper, their solution is generalized to mixed-timing systems: two new *mixed-timing relay stations* are introduced (sync-sync, async-sync), which built on our basic designs.

A particular contribution of this paper are the new relay stations for mixed async/sync interfaces; no previous solution has been proposed which simultaneously solves the above two design challenges: handling mixed asynchronous/synchronous interfaces and accommodating long interconnect delays.

An important theme of our approach is to partition the FIFO's into reusable components. A set of interfaces, both synchronous and asynchronous, is defined; these interfaces can then be glued together to obtain FIFO's 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 FIFO's are especially suitable for high-bandwidth communication: assuming appropriate buffer capacity is used, in steady-state operation the designs have *no synchronization overhead*—each read and write operation 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.

The paper is organized as follows. In Section 2, a top-level view of the two basic FIFO designs is presented, similarities and differences are highlighted, and the critical issues of synchronization and deadlock are raised. In Section 3, the basic mixed-clock FIFO is presented in detail, and simple solutions to both synchronization and deadlock problems are proposed. Reusing some of its components, in Section 4 an asynchronous-synchronous FIFO is introduced. In Section 5, each of the two basic designs is then transformed into a mixed-timing relay-station. Finally, the results for throughput and latency in each design are shown in Section 6, and conclusions are presented in Section 7.

Related Work. A number of papers propose FIFO's and components to handle timing discrepancies between subsystems. Some designs are limited to handling single-clock systems. These approaches have been proposed to handle clock skew [10, 11], drift and jitter [10], and very long interconnect penalties [2].

Several designs have also been proposed to handle mixed-timing domains. One category of approaches attempts to synchronize data items and/or control signals with the receiver, without interfering with its clock ([12, 13]). In particular, Seizovic [13] robustly interfaces *asynchronous* with synchronous environments through a "synchronization FIFO". However, the latency of his design is proportional with the number of FIFO stages, whose implementation includes expensive synchronizers. Furthermore, the design requires the sender to produce data items at a constant rate.

Another solution to robust interfacing of mixed-clock systems is to temporarily pause ([3, 16]) 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.

A few recent FIFO's are closer to our work: mixed-clock and async-async FIFO's from academia, and a mixed-clock FIFO from Intel. A recent mixed-clock FIFO [5] has some similarities in overall architecture to our new design, but has fundamental differences in protocol and implementation. In addition, it has significantly worse latency (three passes through the global signal synchronizers to read data just inserted into an empty FIFO). It also requires the use of a special deadlock detector, which enables the injection of "dummy" data items to restart the FIFO. This design was also modified into a preliminary version of a mixed-clock "relay station", but no relay station FIFO's have previously been proposed for mixed asynchronous/synchronous interfaces. A related asynchronous FIFO design in [4] is re-used in this paper for part of the implementation of the asynchronous interfaces.

Finally, a highly-optimized patented (unpublished) mixed-clock FIFO has been developed at Intel [9]. It has a number of similarities to our basic mixed-clock FIFO in overall operation, but has significantly greater area overhead in implementing the synchronization: while our design has only one synchronizer on each of the two global detectors (full and empty), the Intel design has two synchronizers *per cell*.

2. Overview of the New FIFO's

The mixed-timing FIFO's can be best understood by looking first at the underlying common themes across the designs. This section presents an overview of the basic FIFO interfaces, the underlying

*This work was supported by NSF ITR Award No. NSF-CCR-0086036, NSF Award No. NSF-CCR-97-34803, and a gift from Sun Microsystems.

architectures and special features, of the two FIFO's. Later sections present details of the actual FIFO implementations, as well as a detailed solution to the synchronization problem.

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. 1). For space reasons, this paper presents only the sync-sync and async-sync FIFO's. An async-async FIFO has been presented in [4]. A sync-async FIFO has also been designed, and will be described in a forthcoming technical report.

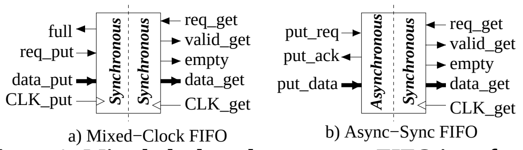
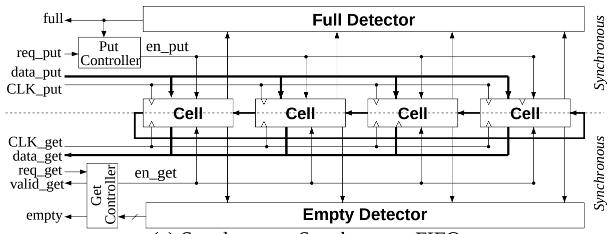


Figure 1: Mixed-clock and async-sync FIFO interfaces

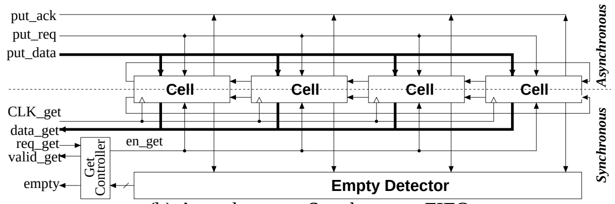
There are two types of synchronous interfaces: a put interface and a get interface. The synchronous interface 1(a) is controlled by CLK_{put}. There are two inputs: req_{put} which controls requests, and $data_{put}$ which is the bus for data items. The *full* output is only asserted when the FIFO is full, otherwise it is deasserted. The get interface 1(b) is controlled by CLK_{get} and a single control input req_{get}. Data is placed on $data_{get}$ output bus, and *empty* is asserted only when the FIFO is empty. For most of this paper, $valid_{get}$ is always asserted during a get operation.

Since the asynchronous interfaces are not synchronized to a clock signal, they are somewhat different. The asynchronous put interface 1(a), much like the synchronous put interface, has two inputs: put_{req} which controls requests, and put_{data}, the bus for data items. However, this interface does not have a *full* output; instead, the interface simply withholds put_{ack} until the operation is completed.

Basic Architecture. Fig. 2 gives a simple overview of the two basic FIFO architectures. Similar architectures can be defined for



(a) Synchronous-Synchronous FIFO



(b) Asynchronous-Synchronous FIFO

Figure 2: Two architectures for mixed-timing FIFO's

async-async ([4]) and sync-async FIFO's.

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 FIFO's 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 FIFO's have a potential for low latency: as soon as a data item is enqueued, it is also available for dequeuing (see Section 6). Secondly, the FIFO's offer the potential for low power: 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.

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 (b) stall the respective interface until it is safe to perform the data operation. In contrast, the asynchronous interfaces do 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.

FIFO Protocols. Now that the various interfaces have been discussed, the behavior of each can be be best understood through some simple simulations (Fig. 3).

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. 3(a), when it receives a request on put_{req} and a data item on put_{data}, immediately after the positive edge of CLK_{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. 3c) is enabled by a request on req_{get}, asserted immediately after the positive edge of CLK_{get}. By the end of the clock cycle, a data item is placed on get_{data} together with its validity bit ($valid_{get}$). 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, $valid_{get}$ and *empty* can indicate three outcomes: (a) data item dequeued, more data items available ($valid_{get}$=1, *empty*=0); (b) data item dequeued, FIFO has become empty ($valid_{get}$=1, *empty*=1); (c) FIFO empty, no data item dequeued ($valid_{get}$=0, *empty*=1).

The asynchronous interfaces use 4-phase communication with single-rail bundled data ([8], [15]). Bundled data is a common scheme where a worst-case matched control signal (e.g. put_{req}) indicates when data is valid (put_{data}). The sender starts a put operation (Fig. 3c) by placing a data item on put_{data} and requesting the FIFO to enqueue it on put_{req}. The enqueuing completion is indicated by asserting put_{ack}. The two control wires are then reset to the idle state, first put_{req} and then put_{ack}.

3. Mixed-Clock FIFO

The section now presents in more detail the first of our two basic mixed-timing FIFO's: the mixed-clock (sync-sync) 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 FIFO's. Furthermore, the solution to the synchronization problem is also introduced for this FIFO; the same solution will be applied to all later designs.

The sync-sync FIFO interfaces are shown in Fig. 1a. The FIFO protocol on the put and get interfaces has been described in Fig. 3c, and the architecture of the FIFO was described in Section 2. For the rest of the subsection, the implementation of the basic cell, the empty/full detectors and the external controllers are presented.

3.1 Cell Implementation

A block diagram of an individual cell is shown in Fig. 4. Each cell has 4 interfaces: on the synchronous put interface, the cell receives data on $data_{put}$ and it is enabled on en_{put} to perform a put

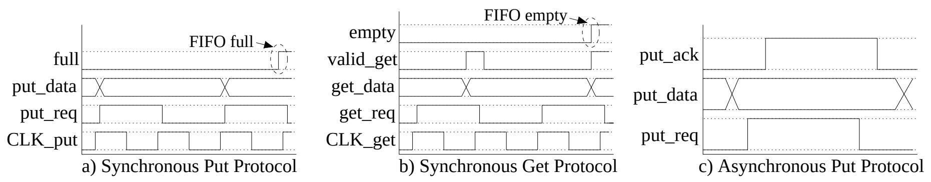


Figure 3: The protocols for each interface

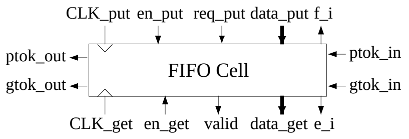


Figure 4: The FIFO cell's interface

operation; req_{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 $data_{get}$, together with its validity bit *valid* (always 1 in this design); the interface is enabled on en_{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 $ptok_{in}$ (put token) and $gtok_{in}$ (get token) from the right cell and passes the tokens on $ptok_{out}$ and $gtok_{out}$ to the left cell.

A detailed implementation of a cell is shown in Fig. 5. The cell's

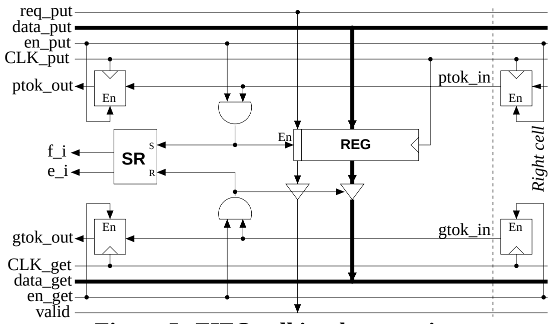


Figure 5: FIFO cell implementation

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 CLK_{put}, and waits for the sender to place a valid data item on the $data_{put}$ bus. A valid data item is indicated to all cells by en_{put}=1, which is the output of the *put controller*.

When there is valid data, the cell performs three actions: (a) it enables register *REG* to latch the data item and also the data validity bit (which is req_{put}), (b) it indicates that the cell has a valid data item (asynchronously sets f_i=1), and (c) it enables the upper left ETDFF (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 ($gtok_{in}$=1) and waits for the receiver to request a data item (en_{get}=1, the output of the *get controller*). When both conditions hold, the cell performs three actions: it (a) enables the broadcasting of the data item on the $data_{get}$ tristate bus and the broadcasting of v_i (the latched req_{put}) on the *valid* tristate bus, (b) indicates that the cell is empty (asynchronously sets e_i=1), and (c) 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.

3.2 Synchronization Issues

A key challenge of designing a mixed-clock FIFO is that of synchronization. Such a FIFO has a highly-concurrent operation: the two 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, the complete solution needs to avoid deadlock, which may occur when using the modified *empty* definition.

Synchronization of Control Signals. Global control signals have to be re-synchronized 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 CLK_{put} and 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 full; 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. 6a-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 solution is 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. 6c): 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.7) to form the global *empty* signal.

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 FIFO's 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. 6c) and the get controller (Fig. 7b) implement exactly the above behavior. The OR gate in the *oe* synchronizer is very important: controlled by en_{get}, it sets the *oe* to a neutral state ("FIFO empty") one clock cycle after a get opera-

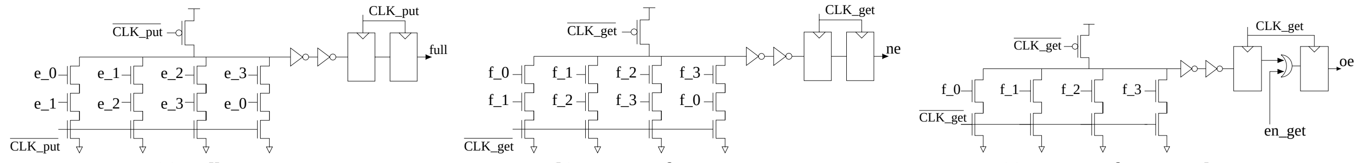


Figure 6: Full and empty detectors for the mixed-clock FIFO

tion takes place. In this case, the "new empty" definition can take precedence in the get controller.

3.3 Put and Get Controllers

Finally, the implementation of the put controller is shown in Fig. 7(a). The controller enables and disables the put operation

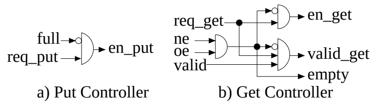


Figure 7: The control for the mixed-clock FIFO

and the movement of the put token in the FIFO: these operations are only enabled when there is a valid data item on $data_{put}$ (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. 7(b) corresponds exactly to these conditions.

4. Async-Sync FIFO

This section now introduces the async-sync FIFO. The design has an asynchronous put interface and a synchronous get interface (see Fig. 1b). The architecture (Fig. 2b) reuses components from the mixed-clock design (Section 3). In particular, the external get controller and empty detector are unchanged; the only components that change are portions of the FIFO cells.

Since the emphasis of this paper is on design reuse, it is useful to consider the reuse of parts of existing cells. Each cell can be divided into 3 distinct parts: a *put part* that deals with the put operation, a *get part* that deals with the get operation, and a *data validity controller (DV)*. The register is split into two parts, one belonging to the put part (the write port), and the other to the get part (the read port). The put and get parts can be designed to interact with a synchronous or asynchronous interface. Then these parts can be glued together with a data validity controller to obtain a cell implementation.

The basic cell in the new async-sync FIFO is designed as follows. The synchronous get part in the mixed-clock cell will be reused as the synchronous half in the mixed async-sync design. The asynchronous put part of [4] will be used for the asynchronous part of the current async-sync cells.

One of the three components is small, but very important: the DV controller, which indicates when the cell is full and when it is empty. In the mixed-clock design, DV was very simple (an SR latch). However, since the mixed async-sync design allows for more concurrency between the writes and reads to the same cell, a new design of DV is required with a more subtle protocol, as will be seen shortly.

Async-Sync Cell Interfaces. The cell communicates on 4 interfaces (Fig. 8): on the put interface to the sender, on the get interface to receiver, on the right one to obtain tokens from the previous cell, and on the left one to pass the tokens to the next cell.

The four interfaces communicate as follows. On the put interface, the cell receives data on put_{data} and request for a put operation on put_{req}; the cell outputs put_{ack_i}, the data operation acknowledgment. The get interface is synchronized by CLK_{get}. There is one input en_{get}, which enables a get operation, and two outputs: a get_{data} bus for data items and f_i, indicating the state of the cell. On the right interface, the cell receives the put token on *we1* and the get token on $gtok_{in}$. The tokens are passed to the left interface: put token on *we* and the get token on $gtok_{out}$.

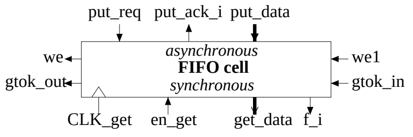


Figure 8: The async-sync FIFO cell interface

Async-Sync Cell Implementation. The cell's implementation consists of the asynchronous part and the synchronous part, plus the new data validity controller, as shown in Fig. 9.

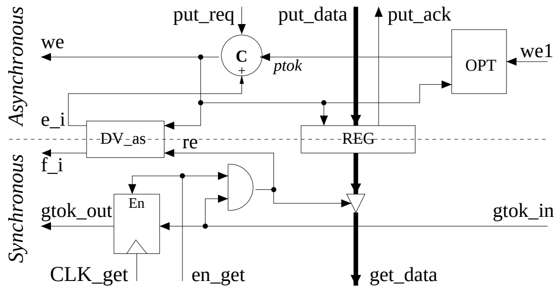


Figure 9: Async-sync cell implementation

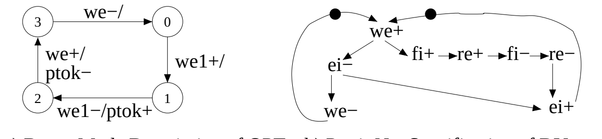
The synchronous part of the cell is identical to the one in the mixed-clock design.

The asynchronous part of the cell is decomposed into several blocks. *ObtainPutToken (OPT)* obtains the respective token from the right interface. It is implemented as a Burst-Mode asynchronous machine (see [7, 4] for details) (Fig. 10a). If the cell does not have the token, *OPT* observes the right cell and waits for the put token (a pulse on *we1*: *we1+*, then *we1-*). At this point, the put token is in the current cell, which enables it for a put operation. When the put operation finishes, the token is sent to the next cell and the cycle resumes. The put operation is controlled by an asymmetric C-element.[1]

The cell's detailed behavior can be understood by simulating a put operation. Initially, the cell starts in an empty state (e_i=1 and f_i=0) and without any tokens. After a two transitions on *we1*, the put token is in the cell (*ptok*=1). When the environment requests a put operation (put_{req}=1), *we* is asserted. This event causes several operations in parallel: the state of the cell is changed to full by DV_{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* (*ptok*=0). When put_{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 synchronous get operation is identical to the mixed-clock FIFO cell's get operation.

The new data validity controller 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 get operation), and f_i (indicating the cell is full—used by the empty detector).

The protocol for DV_{as} is shown as a Petri-Net in Fig.10b ([6]). Once a put operation begins, DV_{as} sets e_i to zero (thus declaring the



a) Burst−Mode Description of OPT b) Petri−Net Specification of DV_as

Figure 10: The async-sync data validity protocol

[1] A C-element has its output at 1 when all its inputs are one; the output becomes 0 when all its inputs become zero. In an asymmetric C-element, some of the inputs (marked with '+') participate only in setting the output of the element to one; their values are irrelevant for the other output transition.

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 CLK_{get} clock cycle. When the get operation finishes (on the next positive edge of CLK_{get}), DV_{as} sets the cell to "empty" (e_i=1) and the behavior can resume. This asymmetric behavior prevents data corruption by a put operation while a get operation is still taking place.

5. FIFO as a Relay Station

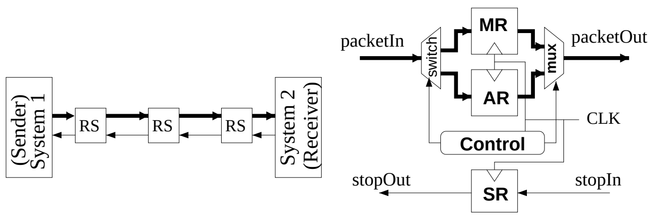
There are two important challenges in designing a chip consisting of several subsystems: the subsystems operate under different timing assumptions (multiple clock speeds, asynchronous vs. synchronous), and there are long communication delays on wires between subsystems. The FIFO designs presented in the previous sections handle various timing assumptions. In this section, these designs are modified to solve the second problem. In particular, the FIFO'ss are transformed into *relay stations*, originally introduced for use only in single-clock systems [2]. In contrast, the proposed new relay stations handle mixed-timing systems.

The section first discusses the concepts and implementation of relay stations. Then, two new mixed-timing relay stations are introduced: sync-sync (mixed-clock) and async-sync.

5.1 Relay-Stations Overview

Relay stations were introduced to alleviate the connection delay penalties between two subsystems operating under the same clock (Fig. 11(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. 11(b). Normally, the packets from the left relay station are passed to the



(a) Insertion of relay stations between systems **(b) Implementation of a relay station**

Figure 11: Relay stations

right relay station. The right relay station also has the possibility 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 MR (main register) 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 un-stalled, it will first send the packet from the main register to the right, and then the one from the auxiliary register.

5.2 Mixed-Clock Relay Station

The new mixed-clock FIFO of Section 3 can now be modified into a special form of *mixed-clock relay station (MCRS)*. Its interface is shown in Fig. 12. The new MCRS simply replaces one of the

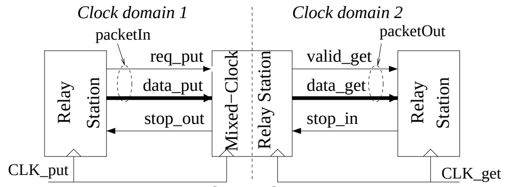
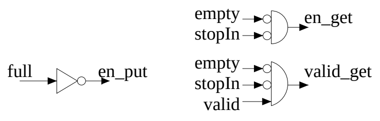


Figure 12: The interface of the relay station FIFO

relay stations, and interfaces between the remaining left and right relay chains (each chain being operated under a different clock).

In contrast to the basic mixed-clock design, 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, req_{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 new relay station can be easily derived from the mixed-clock FIFO by changing only the put and get controllers (Fig. 13). In the mixed-clock FIFO, the put controller enables the enqueuing of



(a) Put Controller (b) Get Controller

Figure 13: The control for the relay station FIFO

valid data items using req_{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. 13(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*.

5.3 Async-Sync Relay Station

Finally, a novel variant is presented: relay stations which handle mixed async-sync interfaces. This design is the first to be proposed which simultaneously solve both critical design challenges: mixed async/sync interfaces and long inter-connect delays.

Basic Architecture. Communication with relay stations between asynchronous and synchronous domains is shown in Fig. 14: the

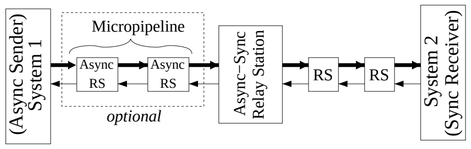


Figure 14: Relay stations between async/sync systems

asynchronous domain sends data packets (possibly through a chain of asynchronous relay stations (ARS), to be discussed below) to the async-sync relay station (ASRS). The packets are then transfered to the synchronous domain, and sent through the chain of synchronous relay stations (SRS) to the receiver.

Asynchronous Relay Stations: Implementation. In principle, *no relay stations* need to be inserted in the asynchronous communication channels (Fig. 14): the async-sync relay station can communicate directly with the asynchronous domain. However, in practice, the designer needs to address both (a) correctness and (b) performance issues in their designs, so inserting ARS's may be desirable.

There are two common asynchronous communication styles: dual-rail and single-rail bundled data [8]. 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 ARS's 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 ARS's may be desirable (as in Carloni) to limit the wire lengths between stages to short hops. Also, as in the dual-rail case, inserting ARS's 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* ([15],[14]). Unlike the synchronous data packets, the asynchronous ones do not need a validity bit: the presence of valid data

Version	Throughput												Latency					
	8-bit data items						16-bit data items						8-bit data items					
	4-place		8-place		16-place		4-place		8-place		16-place		4-place		8-place		16-place	
	put	get	put	get	put	get	put	get	put	get	put	get	Min	Max	Min	Max	Min	Max
Mixed-Clock	565	549	544	523	505	484	505	492	488	471	460	439	5.43	6.34	5.79	6.64	6.14	7.17
Async-Sync	421	549	379	523	357	484	386	492	351	471	332	439	5.53	6.45	6.13	7.17	6.47	7.51
Mixed-Clock RS	580	539	550	517	509	475	521	478	498	459	467	430	5.48	6.41	6.05	7.02	6.23	7.28
Async-Sync RS	421	539	379	517	357	475	386	478	351	459	332	430	5.61	6.35	6.18	7.13	6.57	7.62

Table 1: Simulation results

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.

Async-Sync Relay Stations: Implementation. Finally, the async-sync relay station (ASRS) can be derived by modifying the async-sync FIFO (Section 4). The new interfaces are shown in Fig. 15. Note that the asynchronous interface is identical and supports the

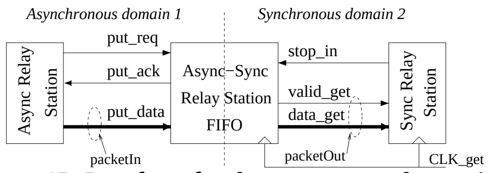


Figure 15: Interfaces for the async-sync relay stations

same communication protocol as the asynchronous interface in the FIFO's 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. The synchronous interface supports the same communication protocol with the respective synchronous interface as in the mixed-clock relay stations, including a data validity bit.

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

The async-sync relay station operates as follows. At its asynchronous interface, the relay station only enqueues data items when they are presented. 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, the right interface receives valid data packets except when the ASRS is empty.

The implementation for the ASRS get controller is shown in Fig. 16. The get controller enables an explicit get operation (en_{get}=1)

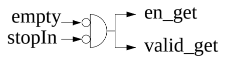


Figure 16: The new ASRS get controller

when it is not stopped from the right (*stopIn*=0) and when the relay station is not empty (*empty*=0) (as in a MCRS). The validity signal, $valid_{get}$, is invalid if either the relay station is stopped or it is empty.

6. Results

To validate the new FIFO's, each was simulated using both commercial and academia tools. The circuits were designed using both library and custom components and simulated using Cadence HSpice in 0.6μ HP CMOS technology, at 3.3V and 300K. Burst-Mode controllers were synthesized using Minimalist[7] and the Petri-Net controller was synthesized using Petrify[6]. Since all simulations are pre-layout, special care was taken in modeling the control and data global buses. Appropriate buffering was inserted into the control buses put_{req}/en_{put} and get_{req}/en_{get}. In addition, put_{ack} was generated through a tree of OR-gates that merges individual acknowledgments into a single global one. For the get_{data} tri-state buses, wiring and environmental delays were modeled by inserting capacitive loads.

Two metrics have been simulated for each design: *latency* and *throughput*. 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. The throughput and latency have been computed for different FIFO capacities (4/8/16 cells) and data widths (8/16 bits).

The results for maximum throughput are given in Table 1. For synchronous interfaces, the throughput is expressed as the maximum clock frequency with which that interface can be clocked. Since the asynchronous interfaces do not have a clock, the throughput is given in MegaOps/s (the number of data operations the interface can perform in a second). 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 put interfaces are slower than their synchronous counterparts because of the increased complexity of the asynchronous put protocol.

Latencies through empty FIFO's are shown only for designs with 8-bit data items (Table 1). 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 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. 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 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*).

Throughput and latency results are quite good for a bus-based design. As expected, both throughput and latency decrease as the FIFO capacity and the datapath width increase. Throughput tends to be higher for synchronous interfaces than for asynchronous ones.

7. 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. The design are then adapted to work between systems with very long interconnection delays, by migrating a single-clock solution by Carloni et. al. (for "latency-insensitive" protocols) to mixed-timing 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 four FIFO designs presented in this paper (two basic designs, and two mixed-timing relay stations) provide a powerful and effective set of solutions for the challenges of future systems-on-a-chip.

8. REFERENCES

[1] D.S. Bormann, P.Y.K. Cheung, "Asynchronous Wrapper for Heterogenous Systems", Proc. ICCD'97, pg. 307-314.
[2] L. Carloni, K. McMillan, A. Saldanha, A. Sangiovanni-Vincentelli, "A Methodology for Correct-by-Construction Latency Insensitive Design", ICCAD'99.
[3] D.M. Chapiro, "Globally-Asynchronous Locally-Synchronous Systems", PhD Thesis, Stanford University, Oct. 1984.
[4] T. Chelcea, S. Nowick, "Low-Latency Asychronous FIFO's using Token Rings", IEEE ASYNC'00 Symp., pp. 210-220.
[5] T. Chelcea, S. Nowick, "A Low-Latency FIFO for Mixed-Clock Systems", IEEE Wkshp. on VLSI'00, pp. 119-128.
[6] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, A. Yakovlev, "Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *IEICE Transactions on Information and Systems*, Vol. E80-D, Number 3, pp. 315-325, March 1997.
[7] R.M. Fuhrer, S.M. Nowick, M. Theobald, N.K. Jha, B. Lin, L. Plana, "MINIMALIST: An environment for Synthesis, Verification and Testability of Burst-Mode Asynchronous Machines," CUCS-020-99, Columbia University, Computer Science Department, 1999.
[8] S. B. Furber, "Asynchronous Design", Proc. of Submicron Electronics, Il Ciocco Italy, pp. 461-492, 1997.
[9] J. Jex, C. Dike, K. Self, "Fully Asynchronous Interface with Programmable Metastability Settling Time Synchronizer", Patent No. 5,598,113 (Jan. 28, 1997).
[10] R. Kol, R. Ginosar, "Adaptive Synchronization for Multi-Synchronous Systems", ICCD'98, pp. 188-198.
[11] M. R. Greenstreet, "Implementing a STARI Chip", ICCD'95, pp. 38-43.
[12] C. L. Seitz, "System Timing", Introduction to VLSI Systems, Ch. 7, Addison-Wesley Pub. Co., 1980.
[13] J. Seizovic, "Pipeline Synchronization", IEEE ASYNC'94 Symp., pp. 87-96.
[14] M. Singh, S.M. Nowick, "MOUSETRAP: Ultra-High-Speed Transition-Signaling Asynchronous Pipelines", ACM TAU-00 Workshop, Austin, TX (Dec. 2000).
[15] I. E. Sutherland, "Micropipelines", Communications of the ACM, 32(6), pp. 720-738, June 1989.
[16] K.Y. Yun, R.P. Donohue, "Pausible Clocking: A First Step Toward Heterogeneous Systems", ICCD'96, pp. 118-123.