

A Low-Latency FIFO for Mixed-Clock Systems*

Tiberiu Chelcea

Steven M. Nowick

Department of Computer Science

Columbia University

e-mail: {tibi,nowick}@cs.columbia.edu

Abstract

This paper presents a low-latency FIFO design that interfaces subsystems on a chip working at different speeds. First, a single-clock domain design is introduced, which is then used as a basis for a mixed-clock version. Finally, the design is adapted to work between subsystems with very long interconnection delays. The designs can be made arbitrarily robust with regard to metastability and clock frequencies.

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 a new low-latency, high-throughput FIFO design which robustly accommodates mixed-clock systems.

Our FIFO design mediates between two interfaces: a *sender* which produces data items and a *receiver* which consumes data items. It is implemented as a circular buffer of identical cells, where each cell communicates with the two systems on common data buses. The input and output behavior of a cell is dictated by the flow of two tokens around the ring: one for enqueueing data and one for dequeueing data. Data items are not moved around the ring once they are enqueue, thus providing the opportunity for low-latency: once a data item is enqueue, it is shortly thereafter available to be dequeue.

We first introduce a basic single-clock FIFO design which is loosely based on a previous asynchronous design in [4]. Then, reusing many of its components, we derive two mixed-clock FIFO designs. These avoid expensive data synchronization: only control signals are synchronized. The first mixed-clock design is a general-purpose one, that can be used to interface any two clock domains. The second one is an extension to accommodate long interconnect delays. It is adaptation to mixed-clock domains of a recent single-clock approach using relay stations [2]. Initial simulation results

in 0.6μ indicate that the designs can be clocked at over 500 MHz.

Related Work. A number of papers in the literature propose FIFOs 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 [5, 6], drift and jitter [5], 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 ([7, 8]). In particular, Seizovic [8] 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 include expensive synchronizers. Furthermore, his design requires the sender to produce data items at a constant rate.

Other designs achieve robust interfacing of mixed-clock systems by temporarily modifying the receiver’s clock ([3, 1, 9]). Synchronization failures are avoided by pausing ([3, 9]) or stretching ([1]) the receiver’s local clock. Each communicating synchronous system is wrapped with asynchronous logic, which is responsible for communicating with the other systems and for adjusting the clocks. Our approach is entirely synchronous and does not change any of the local systems’ clocks, thus avoiding the latency penalties in restarting them.

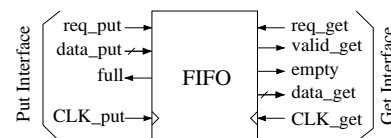


Figure 1: The FIFO’s interfaces

2 Single-Clock Domains

This section introduces our single-clock FIFO design. In the next section, we modify it to handle multiple clocks.

FIFO Overview and Interface. As shown in Fig. 1, a

*This research was funded by NSF Award No. CCR-97-34803.

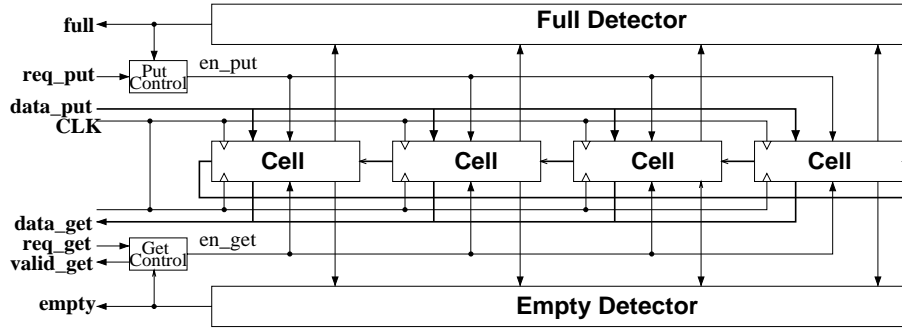


Figure 2: The architecture of the FIFO

FIFO interfaces two subsystems: a *sender* which produces data items and a *receiver* which consumes the data items. The “put interface” consists of a data bus $data_{put}$ (used to send data items), a req_{put} input (requests to enqueue a data item) and a $full$ output (signals when the FIFO is full). The “get interface” consists of a data bus $data_{get}$ (used to receive a data item), a req_{get} input (request to dequeue a data item), a $valid_{get}$ output (used to signal the validity of a data item) and an $empty$ output (indicates when the FIFO is empty). In this design, the two clocks are identical ($CLK_{put} = CLK_{get} = CLK$).

FIFO Protocol. The protocol with the sender is as follows. When the sender wants to enqueue a data item, it places it on $data_{put}$ just after the positive edge of CLK and simultaneously asserts req_{put} . If the data can be accepted (the FIFO is not full), it will be enqueued. When the FIFO becomes full, due to a request on req_{put} , the $full$ output will be asserted just after the next clock edge. At that time, any new pending request from the sender will not be satisfied: the FIFO will freeze the put token and the enqueueing operation, and the new data must be maintained by the sender until $full$ becomes 0.

The protocol with the receiver is more complex. The receiver first issues a request ($req_{get} = 1$) just after the positive edge of CLK . If valid data is available, it will be placed on $data_{get}$ before the next positive clock edge, along with two status bits ($valid_{get}$ and $empty$), and latched by the receiver on that edge.

There are four possible values of $valid_{get}$ and $empty$ in response to a get request. Two cases occur when valid data is dequeued: (1) $valid_{get} = 1$ and $empty = 0$: the FIFO dequeued valid data and is not empty; (2) $valid_{get} = 1$ and $empty = 1$: the FIFO dequeued the last valid data item; it freezes the get token, and any new request from the receiver is ignored until $empty = 0$. The two remaining cases occur when no valid data is available: (3) $valid_{get} = 0$ and $empty = 0$: the FIFO returned a dummy (invalid) data item, and is not empty; or (4) $valid_{get} = 0$ and $empty = 1$: either (a) the FIFO returned no valid data (FIFO is empty and stalled), or (b) it returned a dummy (invalid) data item and has

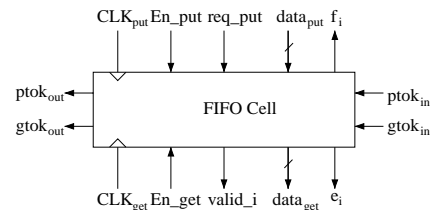


Figure 3: The FIFO cell's interface

become empty. Case 4(a) applies to our single-clock FIFO, when the FIFO is completely empty and cannot process a pending get request. Cases 3 and 4(b) will only be relevant to our mixed-clock FIFO: in this design, both valid and invalid (“dummy”) data items are allowed in the FIFO. The design will be discussed further in Section 3.

FIFO Architecture. Fig. 2 shows the basic architecture of a 4-place FIFO. A FIFO consists of a circular array of identical cells, a *full detector* and an *empty detector*, and control logic for the put operation and get operation. The full and empty detectors observe the state of the FIFO and determine whether the FIFO is full or empty. The input and output behavior of the FIFO is controlled by the flow of two tokens: a *put token* (used to enqueue data items) and a *get token* (used to dequeue data items). Once a data item is enqueued, it is not moved until it is dequeued. Thus the FIFO provides low latency: once a data item is input, it is almost immediately available for output.

The *put controller* enables and disables the put operations. If its output is asserted, the FIFO enqueues one data item and rotates the put token to the left. If it is deasserted, the put interface is stalled. Similarly, the *get controller* enables and disables the get operations.

Tokens move counterclockwise through the array of cells. The cell containing the put token (tail of the queue) has permission to enqueue a data item, and the cell with the get token (head of the queue) has the permission to dequeue its data. The get token is never ahead of the put token. Once a cell has used a token, it will pass it to its left neighbor at the begin-

ning of the next clock cycle, after the respective operation is completed. The token movement is controlled both by interface requests as well as the state of the FIFO (*full* or *empty*), which are combined into the global signals en_{put} and en_{get} .

Cell Implementation. A block diagram of an individual cell is shown in Fig. 3. Each cell communicates with the put interface on $data_{put}$ (to receive data), en_{put} (the request for the put operation from the put controller), and req_{put} (indicates valid data enqueued).¹ Similarly, it communicates with the get interface on $data_{get}$ (used to output a data item), $valid_i$ (indicating the cell contains valid data) and en_{get} (the control for the get operation). In addition, the single clock is tied to both CLK_{put} and CLK_{get} . 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. The state of the cell is communicated to the full detector on e_i (asserted high when the cell is empty) and to the empty detector on f_i (asserted high when the cell is full).

A detailed implementation of a cell is shown in Fig. 4. We

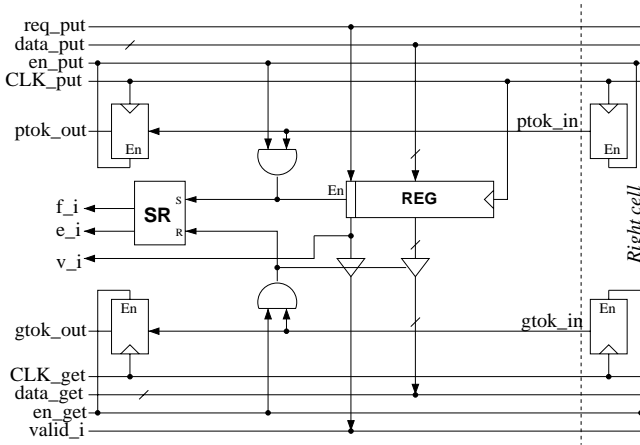


Figure 4: FIFO cell's implementation

will first illustrate the operation of the cell 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 does three actions: (a) it enables the register REG to latch the data item and also a 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 ETDDFF ($en_{put} = 1$) 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

¹In the single-clock scheme, only valid data is enqueued, but the mixed-clock scheme will allow enqueueing of invalid or “dummy” data.

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 several 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 $valid_i$ tristate bus, (b) indicates that the cell is empty (asynchronously sets $e_i = 1$), and (c) enables the lower left ETDDFF to pass the get token. At the beginning of the next clock cycle, the get token is then passed to the left cell.

External Controllers. Now that we have completed the discussion of the FIFO cells, we address the remaining components of Fig. 2: the put and get controllers, and the empty and full detectors.

The implementation of the put and get controllers is shown in Fig. 5. The put controller enables and disables the put op-

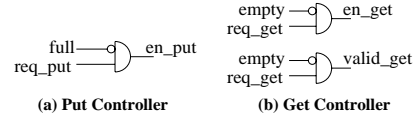


Figure 5: The control for the single-clock FIFO

eration and the movement of the put token in the FIFO: these operations are only enabled when there is a valid data item on $data_{put}$ 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 the FIFO is not empty. The simple AND gate implementations in Fig. 5 correspond exactly to these conditions.

Dynamic logic implementations of the full and empty detectors for a 4-place FIFO are shown in Fig. 6. The full de-

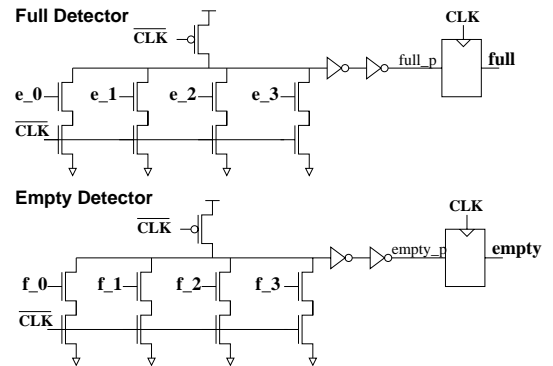


Figure 6: The full and empty detectors' implementation

tor outputs one when there is no empty cell in the FIFO. The empty detector outputs one when there is no full cell in

the FIFO. In the precharge phase, both $full_p$ and $empty_p$ are asserted to one (through two inverters). In the evaluate phase, if no cell is full (all $f_i = 0$), then $empty_p$ remains one, otherwise becomes zero; if no cell is empty (all $e_i = 0$), then $full_p$ remains one, otherwise it becomes zero. These values are latched on the next clock edge to $full$ and $empty$, respectively.

3 Mixed-Clock Domains

In this section we extend our single-clock FIFO to handle mixed-clock domains. We wish to reuse the same FIFO cell implementations and a similar overall architecture. However, since the FIFO is now controlled by two different clocks, we must introduce synchronizers and some modifications to the external logic. Our approach can be made arbitrarily robust in the presence of fast clocks and very different clock speeds.

Overview. Let us now give some intuition motivating the changes needed to handle two clock domains. The previous design will operate correctly with mixed-clock subsystems if the two interfaces are “cooperative”. Interfaces are cooperative if they run at a similar pace and always maintain a few data items of separation in the FIFO. A data item can be safely enqueued into a cell because dequeuing takes place far away from that cell. The global control signals are computed quickly and remain stable.

In reality, however, the two interfaces will often not be cooperative and the FIFO may become empty or full. This situation potentially creates synchronization problems. For example, suppose the FIFO is full. The receiver may grab a data item making the FIFO not full *just when* the sender is reading the FIFO’s state as full and is clocking the $full$ signal. Interestingly, the only problem is for the sender to see a *clean* full signal (i.e, synchronized to its own clock), and not the actual *value* of that signal: if it sees a full FIFO, it conservatively stalls an extra clock cycle; if it sees it as not full, then it correctly proceeds.

Our solution is to introduce synchronization on the global control signals. The $full$ and $empty$ global control signals are now each synchronized to a single clock (CLK_{put} and CLK_{get} , respectively). However, this synchronization in turn introduces additional latencies that may result in data inconsistencies (overflow and underflow). Therefore, we must modify the definitions of “full” and “empty”: we now detect when the FIFO is *heading towards* a full or empty state, and stop the respective interface in time.

There is one final change in the design that is necessitated by the new synchronization scheme: because of the early detection of empty, deadlock may occur. We might stall the receiver when there still is a single valid data item in the FIFO. To avoid deadlock, we allow both valid and invalid (dummy) data items to be enqueued. A dummy data item is enqueued

by the FIFO itself solely to prevent deadlock; once the receiver restarts, no future dummy items are used.

Interestingly, data metastability is not an inherent issue with this FIFO architecture². The two interfaces help each other each other to avoid metastable states: the sender helps the receiver by producing data items and moving the tail of the queue away from the head; the receiver helps the sender by consuming data items and creating space for depositing more data items.

FIFO Architecture. The architecture of our mixed-clock FIFO is shown in Fig. 7. The same exact FIFO cell as in Fig. 4 is used. There are only 4 key differences with respect to our previous design: (a) *two clocks* are now controlling the interfaces (not one clock); (b) *full and empty detectors* are modified to incorporate synchronization; (c) in some special cases deadlock may arise, so a *deadlock detector* is introduced, and (d) the logic inside the *put and get controllers* is modified.

While not shown in Fig. 7, the two clocks now control distinct portions of the FIFO. The passing of the put token is controlled by the sender’s clock (CLK_{put}), and the passing of the get token is controlled by the receiver’s clock (CLK_{get}). The full detector is controlled by CLK_{put} and the empty detector is controlled by CLK_{get} . The newly-introduced deadlock detector is synchronized with CLK_{put} .

Synchronization of Control Signals. We first consider the most important feature of the new FIFO: the synchronization of the two global control signals, $full$ and $empty$. The problem is that each of the signals is clocked by one clock, but its value is effectively controlled by two clocks. Our solution is to re-synchronize the $full$ signal to the sender’s clock, and the $empty$ signal to the receiver’s clock, by adding an extra latch to the output of each of the detectors. These latches delay the output of signals by one clock cycle, which thereby resolves potential metastable states on the two control signals.

Unfortunately, the additional latency on generating $full$ and $empty$ signals may result in FIFO overflow or underflow. For example, when the FIFO becomes full, the sender interface is stalled two clock cycles later; so in the next clock cycle the sender might deposit a new data item, effectively overwriting a un-read data item. Conversely, when the FIFO becomes empty, the receiver interface is stalled two clock cycles later, so in the next clock cycle the receiver might read an empty cell.

As a result, for correct operation, the definitions of full and empty states in the FIFO must be changed: 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 than 2 data items, the FIFO is declared empty: the receiver may then remove the last data item and issue a new (unacknowledged) request,

²Issues with data metastability in the actual implementation are discussed later in the section.

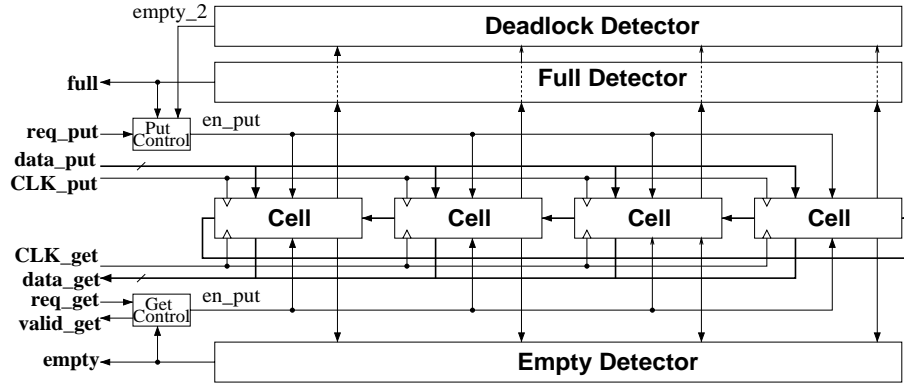


Figure 7: The architecture of the mixed-clock FIFO

before stalling two clock cycles later. Similarly, when there are fewer than two empty cells, the FIFO is declared full: the sender can safely deposit a final data item and issue a new (unacknowledged) request, before being stalled two clock cycles later. The new definitions do not change the protocol with the two subsystems. The only effect is that sometimes the two subsystems see an n -place FIFO as an $n - 1$ place FIFO.

The new implementations of the full and empty detectors, presented in Fig. 8, correspond precisely to the above new

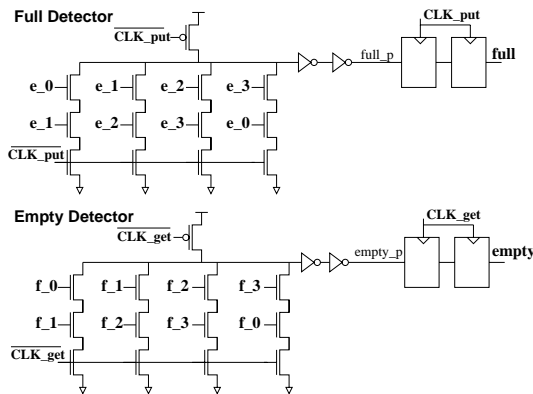


Figure 8: Empty and full detectors, mixed-clock FIFO

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 Prevention. One special case must be now addressed in the new FIFO: the potential for deadlock. Using our new definition of empty (0 or 1 data items), it is possible that the FIFO still contains one valid data item and yet the requesting receiver is stalled.

Our solution is to detect the possibility of deadlock and reactivate the receiver so it can read the stored data item. To do so, we signal to the receiver that the FIFO is “not empty” by injecting a single *dummy data item* (invalid data item). The

dummy data item is injected *only* when the FIFO is “empty” and contains a single valid data item. The empty detector will then see two data items in the FIFO and restart the receiver. (This approach is used even if the receiver is not requesting.)

The solution changes the current implementation in two ways. First, a *deadlock detector* is introduced. Secondly, the put and get controllers must also be changed.

The implementation of the deadlock detector is shown in Fig. 9. It implements the exact condition described above: its

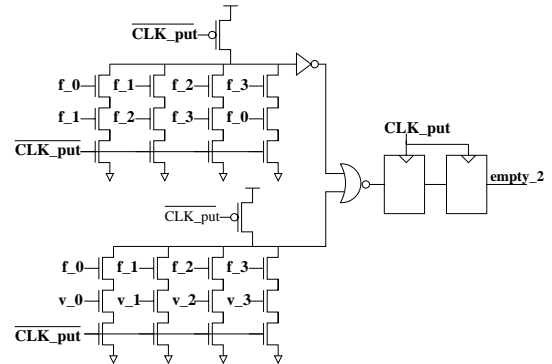


Figure 9: The implementation of deadlock detector

output, $empty_2$, will be 1 if the FIFO is empty and *at least* one cell has a valid data item. The deadlock detector looks for valid data items only in cells that are full and have their data validity bit set (see Fig.4). This prevents the deadlock detector from using stale validity bits: one such bit can become stale because our design never resets the *REG* contents (including the validity bit) after its contents are dequeued.

The output of the deadlock detector controls the injection of dummy data items: $empty_2$ is fed into the put control block which enables the enqueueing of a dummy data item. The detector’s output is in turn synchronized through two ETDF’s controlled by CLK_{put} . Once the dummy data item is injected, no further ones are injected since the output is deasserted (the FIFO is no longer empty).

The put and get control logic must also be modified to handle dummy data items. The put control (Fig. 10(a)) now

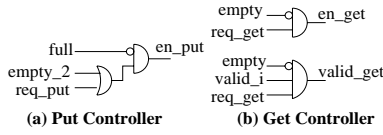


Figure 10: The controllers for the mixed-clock FIFO

enables the enqueueing of data items under two conditions: (a) the data item is valid ($req_{put} = 1$), or (b) a dummy item must be injected for deadlock prevention ($empty_2 = 1$ from the deadlock detector). The get controller is also modified to indicate both valid and invalid data items: $valid_{get}$ is computed from the validity of the current dequeued data item.

FIFO Robustness. 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 when it operates at very fast clock frequencies, and metastable states on control cannot be solved in one clock cycle. The second case is due to large discrepancies between the sender’s and receiver’s clock frequencies. Below, we present solutions to these problems that make the FIFO more robust.

Fast clock speeds. In general, it is desirable to have high mean time between failures (MTBF). At low clock frequencies, using two latches for synchronization is usually sufficient for good MTBF, but at higher clock frequencies the MTBF may be inadequate. We can make the synchronization arbitrarily robust by adding more latches to whichever global control signal (or signals), $full$ and $empty$, has poor MTBF. For each added latch, we must declare the FIFO full when there is one *more* empty cell left and declare the FIFO empty when there is one *more* full cell left, thus modifying the full or empty logic accordingly.

Large differences in clock speeds. Our design uses a small optimization which may cause malfunction in extreme cases: early signaling of full/empty for a cell. As shown in Fig. 4, 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.

To see the problem, 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 clock frequency of the receiver 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.

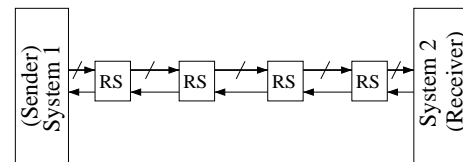
A simple and robust solution is to modify the full or empty logic (whichever controls the faster interface) to stop the faster interface earlier. Let us suppose that the receiver is much faster than the sender. The definition of empty is modified as follows: the FIFO is empty when there are less than three full cells left. In this case, the receiver will stall before reading the last item, so no stale data will be dequeued³.

4 FIFO as a Relay Station

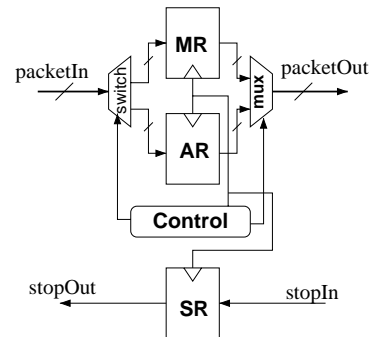
There are two important challenges in designing a chip consisting of several subsystems: the subsystems operate at different clock speeds, and there are long communication delays on wires between subsystems. The FIFO design presented in the previous section handles different clock speeds. We now modify our design to solve the second problem too. For this, we use the idea of *relay stations*, first introduced in [2] for use only in single-clock systems. Here, we adapt them to mixed-clock systems.

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 system 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



(a) Insertion of relay stations between systems



(b) Implementation of a relay station

Figure 11: Relay stations

³If empty logic is modified, then the deadlock detector must be modified similarly.

the 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.

Our mixed-clock FIFO can be modified into a special form of mixed-clock relay station. Its new interface is shown in Fig. 12. The new FIFO simply replaces one relay station,

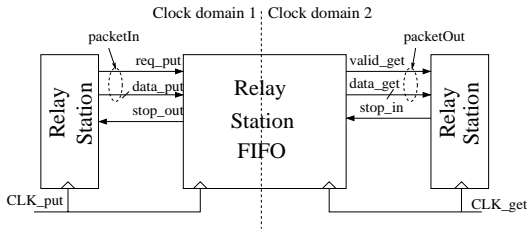


Figure 12: The interface of the relay station FIFO

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

In contrast to the mixed-clock design, the new FIFO *always* passes (valid or invalid) 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. Thus, unlike our 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 FIFO simply stops the put interface: the *full* signal is used as *stopOut* to the left interface.

The new FIFO can be easily derived from our mixed-clock design of Section 3 using small modifications. The put and get controllers need to be changed. In the mixed-clock FIFO, the put controller enables both the injection of dummy data items and the enqueueing of valid ones (thus using both *empty₂* and *req_put* signals), while in the relay-station design (Fig. 13) it simply allows both dummy and valid data items

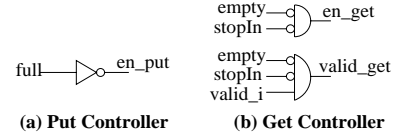


Figure 13: The control for the relay station FIFO

to pass through. In addition, even compared with our basic single-clock design, where the put controller only enables insertion of data items on demand (i.e. using the *req_put* input), our 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, our new get controller enables continuous dequeuing of data items, unlike our 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*.

Finally, note that a relay-station FIFO can never deadlock: the put interface enqueues a data item (valid or invalid) on every clock cycle, meaning that the FIFO, if “empty”, becomes “non empty” immediately. After the synchronization through the empty detector, the get interface is thus able to read the stream of data items. Until it is restarted, the get interface simply “reads” void data items. In summary, since the get interface *always* restarts, the deadlock detector is removed from the design.

5 Results

In order to evaluate the performance of the various FIFO designs presented in this paper, the circuits were implemented in 0.6 μ HP CMOS technology and simulated at 3.3V and 300K using HSpice (©Avant! Corporation).

We have simulated the behavior of a FIFO of capacity 8, with a data item of width 8. Special care has been taken to make the simulations realistic. For *en_put*, *req_put*, *en_get* and *data_put*, we have used buffering to drive the capacitance of all cells. For *valid_i* and *data_get*, in addition to the load provided by the appropriate latching/result logic, we have modeled the load contributed by long wires. Since we this is a pre-layout simulation, we can only estimate the wire load, which was assumed to be 2 inverters per cell.

The results for the maximum clock frequencies at which each circuit can be clocked are presented in Table 1. All of

Version	CLK_{in}	CLK_{out}
Single Clock	584 MHz	
Mixed Clocks	523 MHz	554 MHz
Relay Station	562 MHz	552 MHz

Table 1: Maximum clocking frequencies for new FIFO’s

the designs can be clocked to over 500MHz. The somewhat lower performance of the mixed-clock design is due to the increased complexity of the *empty* and *empty₂* logic. Although the mixed-clock and the relay station designs are very similar, the performance of the latter is better due to the elimination of the *empty₂* circuitry and the decreased complexity of the put controller.

6 Conclusions and Future Work

This paper has presented a new low-latency FIFO design which interfaces between subsystems with different clock speeds. The design is based on the idea of token passing. Our solution is developed from a basic single-clock version. Our mixed-clock design does not need data synchronization: it only synchronizes on a few global control signals. A further extension to the newly-introduced relay stations is able to interface between systems with long interconnect delays and different clock speeds.

In steady-state operation, our mixed-clock FIFO maintains both high throughput and complete robustness. The receiver can dequeue and the sender can enqueue data items on each respective clock cycle. Furthermore, in steady state, our design has *zero* probability of synchronization failure. In contrast, Seizovic [8] always synchronizes data through a pipeline synchronizer, which always has a non-zero probability of failure. Likewise, the approach using stretchable and pausable clocks ([1, 3, 9]) may result in throughput penalties, even in steady-state, due to local clock adjustments.

One area of future work is to further improve the FIFO's latency. When the FIFO is empty, our design has moderate overhead in restarting. When a valid data item is inserted, our design requires deadlock detection, insertion of a dummy token and reactivation of the get interface before that item can be removed. Furthermore, the injection of a single dummy token may in some cases delay the read of the next inserted valid data item. We believe that new variants of deadlock prevention may reduce these overheads.

We also plan to adapt our mixed-clock design to interface asynchronous and synchronous subsystems, by combining it with our earlier asynchronous FIFO presented in [4].

Acknowledgments.

The authors would like to thank Montek Singh, Michael Theobald and Stephen Unger for helpful discussions and suggestions about our design, and for help with the simulations.

References

- [1] D.S. Bormann, P.Y.K. Cheung "Asynchronous Wrapper for Heterogeneous Systems", Proc. International Conference on Computer Design (ICCD), IEEE Computer Society Press, 1997, 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 Asynchronous FIFO's using Token Rings", International Symposium on Advanced Research in Asynchronous Circuits and Systems, April 2000.
- [5] R. Kol, R. Ginosar, "Adaptive Synchronization for Multi-Synchronous Systems", 1998 IEEE International Conference on Computer Design (ICCD'98), pp. 188-189, Oct. 1998.
- [6] M. R. Greenstreet, "Implementing a STARI Chip", Proc. International Conf. Computer Design (ICCD), pp. 38-43, IEEE Computer Society Press, 1995.
- [7] C. L. Seitz, "System Timing", Introduction to VLSI Systems, Ch. 7, Addison-Wesley, 1980.
- [8] J. Seizovic, "Pipeline Synchronization", Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, pp. 87-96, November 1994.
- [9] K.Y. Yun, R.P. Donohue "Pausible Clocking: A First Step Toward Heterogeneous Systems", Proc. International Conference on Computer Design (ICCD), IEEE Computer Society Press, Oct. 1996.