Low–Latency Asynchronous FIFO's Using Token Rings*

Tiberiu Chelcea Steven M. Nowick
Department of Computer Science
Columbia University
New York, NY 10027
e-mail: {tibi,nowick}@cs.columbia.edu

Abstract

This paper presents several new asynchronous FIFO designs. While most existing FIFO's trade higher throughput for higher latency, our goal is to achieve very low latency while maintaining good throughput. The designs are implemented as circular arrays of cells connected to common data buses. Data items are not moved around the array once they are enqueued. Each cell's input and output behavior is dictated by the flow of two tokens around the ring: one that allows enqueuing data and one that allows dequeuing data. Two novel protocols are introduced with various degrees of parallelism, as well as four different implementations. The best simulation results, in 0.6μ, have a latency of 1.73ns and throughput of 454 MegaOperations/second for a 4–place FIFO.

1 Introduction

This paper presents several novel designs for asynchronous FIFO's. The designs are built using token passing rings of identical components. The goal is to achieve very low latency, while still maintaining good throughput.

Most existing FIFO's trade higher throughput for higher latency. However, there exist applications where low latency of a data item (delay from input to output in an empty FIFO) is desired. Such FIFO's are used, for example, to interface systems that operate at different speeds, so an asynchronous solution is a natural choice.

There are several existing FIFO designs that target low latency. Some of these use circular buffers with centralized control, while others attempt to modify existing throughput–oriented pipelines to shorten critical paths. A final group of designs uses some form of token–passing techniques and distributed control. Compared to our designs, those in the first two categories suffer from significant overhead in area and delay, while those in the latter category impose greater restrictions on concurrency or have longer critical paths.

Our new FIFO's are implemented using circular structures of identical blocks, called cells. Each cell communicates with the environment on common data buses. The input and output behavior of a cell is dictated by the flow of two tokens around the ring: one that allows enqueuing of data and one that allows dequeuing of data. Data items are not moved around the ring once they are enqueued. Since data is not moved, low latency can be obtained: once a data item is input, it is immediately available for output.

In particular, we present two FIFO protocols, with varying degrees of parallelism, as well as four different implementations. The implementations are synthesized using a variety of circuit styles (handshake circuits, Petri Nets, burst–mode, manual) and decompositions.

Detailed HSpice results are also included for each implementation, to evaluate both latency and throughput. A variety of simulation experiments were performed, with varied FIFO capacity (4- to 16–cell FIFO's) and speed of the environment. The results are quite promising, both in terms of latency and throughput. These results indicate that, even for larger capacity, our bus–based circular FIFO's maintain competitive throughput rates and while allowing very low latency.

Related Work. There are numerous asynchronous FIFO designs presented in the literature. These can be classified them into two broad categories.

Most FIFO designs are targeted to *high throughput*. They can be built from asynchronous pipelines (called *micropipelines* [1]) by removing the stage logic. Recently, a number of optimizations have been proposed [11, 12, 13]. These differ from our designs in two respects: they often have quite *poor latency* (proportional to the length of the queue), and they require *data movement* (which can result in large power consumption). All of these designs have distributed control.

A second class of FIFO designs is targeted to *improving latency* (the time from enqueuing to dequeuing one data item in an empty FIFO). One technique is to use variants of existing flow–through FIFO's, structurally modified such that the datapaths are shortened. An approach in [5] decreases the latency by reducing the number of stages a data item has

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

to travel from input to output (parallel FIFO, tree FIFO and square FIFO). A second approach in [5] is to forward a data item whenever cells in front of it are empty (folded FIFO). In all of these designs, unlike ours, data items are moved. Furthermore, in the first approach, the latency (in terms of number of traversed stages) is still proportional to the number of FIFO stages; in the latter, while the latency is only two stages, these are fairly complex and the critical path includes an arbiter.

Another common technique for low latency, adapted from the synchronous world, is to use a *circular buffer*. Two such designs, by Sutherland [1, 14] and Yakovlev et. al. [15], use *centralized control*. They do not move data; instead, they use counters to keep track of the current storage locations. Unlike ours, these designs suffer from significant complexity, which results in increased delay and area overhead.

Closer to our own work, several designs have been proposed for circular buffer with *distributed control* using *token passing* [16, 19, 18]. Unlike our design, several of these do not allow overlapped writes and reads to the same cell, and thus suffer from increased latency through an unloaded FIFO. In addition, besides concurrency limitations, these designs present latency penalties due to increased datapath length [16] or slow control logic [19] (local control logic is influenced by the state of the current cell as well as the states of the adjacent cells and global FULL/EMPTY signals).

Finally, in a design very similar to ours, Yi [17] has proposed a "Word–Slice FIFO", which uses a circular array of cells, common data buses and token passing. However, there are some notable differences. First, the throughput on the get interface is significantly worse: in addition to the common completion trees and bus broadcasting, his critical path includes 10 gates (3 C–elements, 4 AND's, 1 OR and 2 inverters), while ours includes at most 6 gates (2 C–elements, 1 AOI, 2 inverters and a register read matched delay). Furthermore, while the control latency (in logic gates) is roughly the same for both designs, he has noticeably tighter bundling constraints on the get interface. In an empty FIFO, with a pending get request, our design already enables the read port of the cell using an *early read enable.* In contrast, his design enables the read port only after the new data is safely latched, using a *late read enable*. As a result, due to this late broadcast of the data item, there is less time to meet the bundling requirements. This effect is only exacerbated as the FIFO size increases. Finally, there are some differences in protocol (active vs. passive ports, behavior when the FIFO is full) and in cell interface (he exports of the data validity).

There are also somewhat related designs which are not general–purpose FIFO's. Yantchev & al. [6] present a low-latency *FIFO buffer* for network interfaces. Unlike our design, the buffer's behavior is restricted to writing k data items and then reading them, in strict alternation.

The idea of token passing has been used successfully in other asynchronous applications. Our token passing approach has similarities to work by both Martin [2] (the permission token only moves when needed) and Ebergen [4] (there is no explicit counter–flowing requests for tokens).

Paper Overview. The paper is structured as follows. Section 2 describes a base protocol for a FIFO cell, and Section 3 describes three different implementations of the base protocol. Section 4 introduces an optimized variant of the base protocol which offers greater parallelism, and also describes an implementation. Finally, HSpice simulation results are presented in Section 5.

2 Base Protocol

This section describes the architecture of the FIFO and discusses the base protocol of each of its cells. The base protocol is used as a template for each design in the paper.

2.1 FIFO Architecture

The FIFO has two interfaces to the environment: a **put** interface for enqueuing data and a **get** interface for dequeuing data. A top level block diagram of the FIFO is presented in Fig. 1. The queue can perform concurrent enqueuing and dequeuing of the data.

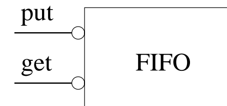

Figure 1: FIFO's interface to the environment

In all the designs, the **put** and **get** channels are passive (i.e. the environment initiates the enqueuing and dequeuing of the data). Throughout the paper, the passive channels are indicated with hollow circles and the active channels with filled circles. However, all designs can be modified to have any combination of passive/active interfaces on the two channels with little effort. The port activity type does not change the base protocol.

At the architectural level, an n–place FIFO consists of a circular array of n identical cells and a special cell called *Starter*. Each cell can store one data item, and it communicates with the environment and with the two neighboring cells (Fig. 2).

Communication with the environment is performed on common global buses. There are two buses: a "put" bus (the environment enqueues some data item, corresponds to the global **put** interface) and a "get" bus (the environment dequeues some data item, corresponds to the global **get** interface).

At any time, after startup, there are two tokens in the array: a PUT token and a GET token. The cell that has the put token

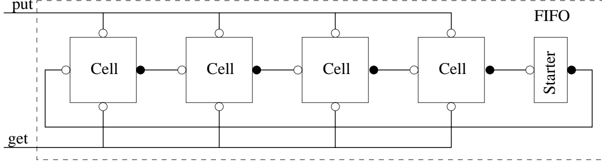

Figure 2: FIFO's internal architecture

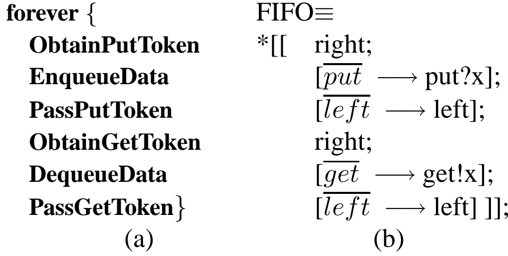

Figure 3: Cell's base protocol

(tail of the queue) has the permission to enqueue the data item from the **put** bus, and the cell that has the GET token (head of the queue) has the permission dequeue its data item onto the **get** bus. The tokens always flow in a counterclockwise direction. The PUT token is *always* ahead of the GET token. This requirement corresponds to normal FIFO behavior: a cell must first enqueue data before dequeuing it.

The *Starter* is a special cell used at startup to inject the two tokens into the ring. Initially, the ring is empty and the starter has both tokens. When requested, it will put the two tokens in circulation: first the PUT token and then the GET token. After that, it will simply pass the tokens from one adjacent cell to the other, performing no other actions.

2.2 Cell: Base Protocol

Informally, the protocol of each cell can be described by the simple program in Fig. 3(a). The cell's behavior is the following. It first requests the PUT token from right. Once it obtains it, it enqueues data when the environment provides it and passes the token to the left. Next it requests the GET token and, when received, dequeues data when the environment requests it and passes the token to the left. The whole cycle starts again by requesting the PUT token from right.

This behavior guarantees:

- *correct sequencing*: since both PUT and GET tokens have first been used by the right cell and then passed to the left neighbor, it follows that the right cell will enqueue data before the left one and it will dequeue data before the left one.
- *no deadlock*: the tokens will freely flow around the ring, because each cell passes tokens to the left once they have been used for a data operation.

The dynamic behavior of a FIFO is illustrated in Fig. 4. The figure presents the initial state of the queue and the resulting states after put and get operations. The PUT and GET tokens are indicated with a star next to the interfaces to the **put** and **get** buses, respectively.

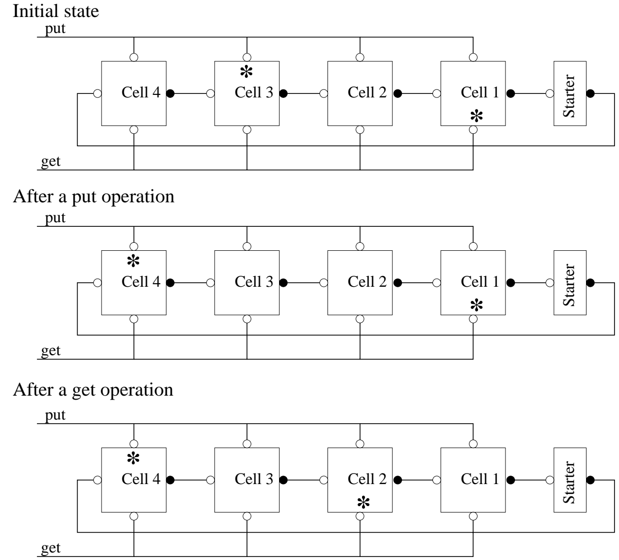

Figure 4: Dynamic behavior of the FIFO

There are two special cases in the FIFO's behavior:

- *FIFO empty*: In this case, the environment can still safely make a *get* request. PUT and GET are in adjacent cells (left and right, respectively). The PUT cell is currently not receiving new data. The GET cell tries to pass the GET token to left, but it is blocked. It will remain blocked until the PUT cell has completely enqueued data and passed the PUT token.
- *FIFO full*: In this case, the environment can still safely make a *put* request. GET and PUT are in adjacent cells (left and right, respectively; the reverse of the previous case). The GET cell is currently not dequeuing stored data. The PUT cell tries to pass the PUT token to the left, but it is blocked. It will remain blocked until the GET cell has completely dequeued data and passed the GET token.

2.3 Cell: Handshake Behavior

We now concentrate on the detailed handshake behavior of each cell. We first discuss the cell's interfaces and then give a CSP [3] specification of the cell's behavior.

The interface of each cell consists of four channels. Two channels are used to communicate with the environment: **put** (passive, used to receive data from the environment) and **get**

(passive, used to output data to the environment). The other two are used to communicate with the adjacent cells: **right** (active, used to receive the tokens) and **left** (passive, used to pass the tokens)

The flow of the two tokens is multiplexed onto *single* channels. The **right** channel is used first to receive the PUT token and then the GET token. Similarly, the tokens are passed on the **left** channel in the same order. The strict sequence and separation of enqueuing and dequeuing of data guarantees the safe multiplexing of token passing.

The behavior of each cell can is described in a CSP-like syntax, as shown in Fig. 3(b). The CSP program is interpreted as follows. The cell first completes a full handshake on the right channel. The cell then checks the **put** channel for data, using the probe construct [3] (a probe is a boolean function on a channel that is true when there is a pending communication; the start of the communication may have occurred at any point before the probe is checked). When a pending communication is detected, the cell completes a full handshake on **put** and then checks the left channel for a pending request. When a request is detected, the cell completes the full handshake on the left channel. This behavior repeats again, with the communication with the environment on the **get** channel.

The CSP specification leaves some choices for the handshake type and data encoding. In all the designs presented in this paper, we have used 4–phase handshaking for channels, bundled data for data channels and a broad protocol for data validity [8]. The handshake expansion of the above program helps us trace the observable activity (the handshake events) on the cell's channels:

$$\begin{aligned}
*[\quad & right_req\uparrow; [right_ack]; right_req\downarrow; [\neg right_ack];\\
& [put_req]; put_ack\uparrow; [\neg put_req]; put_ack\downarrow;\\
& [left_req]; left_ack\uparrow; [\neg left_req]; left_ack\downarrow;\\
& right_req\uparrow; [\neg right_ack]; right_req\downarrow; [\neg right_ack];\\
& [get_req]; get_ack\uparrow; [\neg get_req]; get_ack\downarrow;\\
& [left_req]; left_ack\uparrow; [\neg left_req]; left_ack\downarrow]
\end{aligned}$$

The *Starter* has a different behavior and its own unique specification. The interface of the *Starter* cell consists of two channels, both used to communicated with the adjacent cells: **left** (passive, used to pass the tokens) and **right** (active, used to request the tokens). Its behavior is described by a CSP program:

$$\begin{aligned}
&\text{STARTER}\equiv\\
&\quad [\overline{left} \longrightarrow left];\\
&\quad [\overline{left} \longrightarrow left];\\
&\quad *[\overline{left} \longrightarrow right; left]
\end{aligned}$$

The starter's behavior is as follows. For the first two requests on its left channel, the *Starter* simply completes the handshake on those channels, corresponding to placing of the two tokens into circulation. The *Starter* then enters an infinite loop in which, for each request on its left channel, it performs a handshake on the right channel and then completes the handshake on the left. This operation corresponds to passing a token from the right cell to the left cell.

The observable events on the starter's interface are given by the following handshake expansion:

$$\begin{aligned}
& [left_req]; left_ack\uparrow; [\neg left_req]; left_ack\downarrow;\\
& [left_req]; left_ack\uparrow; [\neg left_req]; left_ack\downarrow;\\
& *[[left_req];\\
& right_req\uparrow; [right_ack]; right_req\downarrow; [\neg right_ack];\\
& left_ack\uparrow; [\neg left_ack]; left_ack\downarrow]
\end{aligned}$$

3 Base Protocol Implementation

This section presents three distinct implementations of the base protocol. The first one uses handshake circuits [7, 8], the second is synthesized from Petri Nets [9], and the third one uses communicating burst–mode machines [10, 20]. The next section will discuss a more optimized protocol and a corresponding implementation.

3.1 Handshake Circuits

The base protocol can be implemented using handshake circuits [7, 8]. We present both a Tangram program specification, and a corresponding handshake decomposition. Finally, the reader is referred to [8] for details regarding the implementation of the handshake components.

The Tangram programs for both the FIFO cell and the *Starter* are given below:

```
proc cell (put?T & get!T & right & left)
begin
   x : T
| forever do
   right; put?x; left;
   right; get!x; left;
od end

proc starter (left & right)
begin
|  left;
   left;
   forever do right; left od
end
```

The program for the FIFO cell executes an infinite loop. It first performs a communication on the right channel, then inputs a data item (of type T) into internal variable x, and then performs a left communication. This corresponds to the PUT part of the program. The GET part is similar: communication on the right channel, data output from variable x and communication on the left channel.

The *Starter* program initially performs two communications on the left channel and then enters an infinite loop. The loop consists of two communications: a right one followed by a left one[1].

A handshake circuit for FIFO cell is given in Fig. 5. This

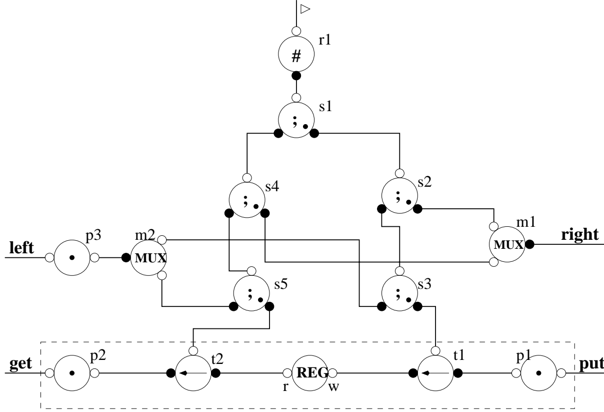

Figure 5: Cell's implementation with handshake circuits

circuit was manually derived from the Tangram program. The topmost component (r1 – "repeater") repeats communication forever with the two–way sequencer s1 (a component that, for each handshake on the passive port, sequences handshakes in turn on its two active ports).

The two handshake calls of s1 (first to s2 and then to s4) correspond to the two parts of the program: first enqueuing data and then dequeuing data. Each part consists of a sequence of three handshakes. For the PUT part, s2 first performs a handshake on the right channel through the MUX m1 (also known as call module; a communication on one of the passive channels determines a communication on the only active channel); it thus obtains the token, and then it calls s3. This sequencer, in turn, first synchronizes with the **put** channel through the passivator p1 (a component that synchronizes communications on its passive ports) and then transfers data to the register through the t1 transferer, thus effectively enqueuing data. It then passes the token by synchronizing with the left channel through the MUX m2 and passivator p3. To end the put cycle, s3 completes the handshake with s2, which, in turn, completes the handshake with s1.

The second handshake on s1 activates a GET, which is very similar, but the communication with the environment is performed on the **get** channel by transfering data from the register.

Interestingly, the behavior of each FIFO cell resembles the behavior of a one–place wagging register [7], a special form of shift register in which input and output operations strictly alternate. In fact, the implementation contains a wagging register (dashed box). One complete cycle of this register consists of a put followed by a get (in contrast, the original wagging register of [7] first outputs data, then inputs data). The rest of the logic is used to control these data operations, synchronizing them with receipt of the appropriate token.

[1] Since Tangram lacks the probe construct of CSP, we first initiate a right communication and then synchronize on the left channel. The role of the starter remains exactly the same, but the observable interface activity changes a bit.

The *REG* handshake component is a storage element, implemented using latches [8] with tri–state outputs. It has two passive ports: w and r (writing data/reading data). The data inputs to the w port are taken from the **put** channel and the data outputs from the r port are placed on the **get** channel. This implementation of the register will be used throughout this paper.

The *Starter* can be similarly decomposed, as shown in Fig. 6. Its behavior is as follows. Upon startup, it sequences

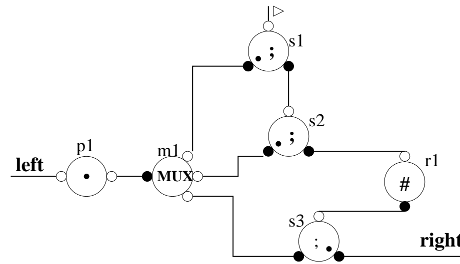

Figure 6: Starter's implementation with handshake circuits

two calls from s1: a call to MUX m1 to synchronize on the left channel and another one to sequencer s2. This latter one first performs a synchronization on the left channel through MUX m1 and then calls repeater r1. This cell henceforth repeatedly calls sequencer s3, which first performs a right communication and then a left one through the MUX. Repeater r1 never acknowledges s2.

The actual implementation of each handshake component can be found in [7] and [8]. We have chosen the components' implementations such that the observable behavior at the interface is the same as the one given by the handshake expansions of the previous section.

3.2 Petri Nets

The behavior of a FIFO cell and of a *Starter* can also be described using Petri Net specifications (not shown here due to space limitations). The specifications closely follow the handshake expansions of Section 2.3. Only the implicit concurrency of the probe construct in the CSP specifications has to be carefully modeled. The specifications were synthesized with Petrify [9] into monolithic circuits (as opposed to networks of communicating controllers).

3.3 Burst–Mode Decomposition

The final implementation of the base protocol is described using burst–mode (BM) machines [10]. We cannot derive a single monolithic burst–mode specification for the cell behavior due to the concurrent activity on the **put** and **get** channels. Instead, we decompose the cell into several interacting machines.

The decomposition for the FIFO cell is given in Fig. 7. It consists of two core BM machines, a datapath component (the register) and two SI C–elements (implementing *Put* and *Get Controllers*[2]). Each component has the following role:

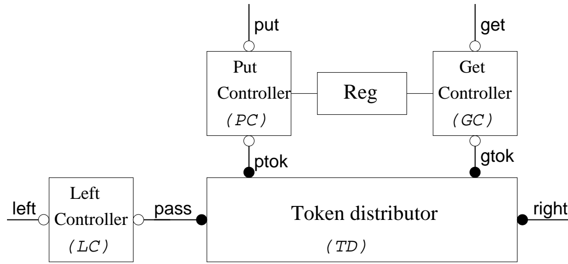

Figure 7: BM decomposition for a FIFO cell

- *Token Distributor (TD)*: This is the core controller of the decomposition. It has two functions: to perform handshaking on the right channel (receive the token) and to alternately enable the *Put Controller* and *Get Controller*. Once the appropriate data operation is complete, *(TD)* passes the token to the left cell by synchronizing with the *Left Controller*.
- *Put/Get Controllers (PC/GC)*: These machines handle the handshaking on the **put**/**get** channels. They are enabled both by requests on the corresponding bus channels and by tokens received from *(TD)* (the PUT token on the *ptok* channel and the GET token on the *gtok* channel). They also control the writing and reading of the data register.
- *Left Controller (LC)*: This machine performs handshaking on the **left** port. It is enabled when it receives a request from the left cell and when *(TD)* tries to pass the token.

The behavior of the interacting machines is as follows. *(TD)* first completes a communication on the right channel, then distributes the token to either *ptok* or *gtok* channels (*ptok* first, *gtok* next) to enable the appropriate data operation. Once the data operation is completed, it then passes the token to the left on the *pass* channel.

When enabled, *(PC)* will synchronize and complete the four–phase handshaking on the **put** channel and on the *ptok* channel, and also controls the register to enable data latching. The *Get Controller* works similarly, but it enables the register to output data.

[2]We will shortly show that these can actually be regarded as valid BM machines.

The BM specifications for *(PC)* and *(GC)* are given in Fig. 8. These specifications assume the cell has the appropriate PUT/GET token. However, usually a cell does not have the appropriate token. In this case, it will simply observe the changes on *put_req* (or *get_req*) without producing any output.

The BM specification for *(LC)* is given in Fig. 9 and the BM specification for *(TD)* is given in Fig. 10.

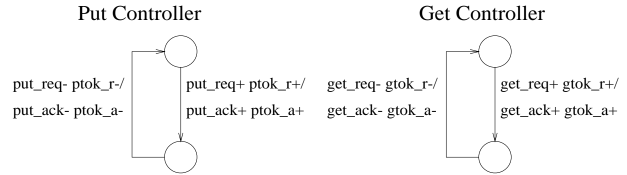

Figure 8: BM specs for Put and Get controllers

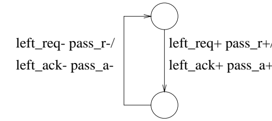

Figure 9: BM specification for the Left Controller

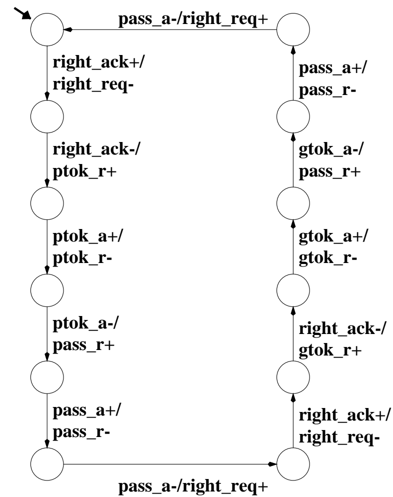

Figure 10: BM specification for the Token Distributor

The specifications for *(PC)*, *(GC)* and *(LC)* can be implemented as C-elements. For *(PC)* once the *input burst* (a concurrent change in the inputs that determine output and/or state changes [10]) *put_req+ ptok_r+* arrives, the controller responds by acknowledging on both channels in the *output burst* (a concurrent change in the outputs caused by the input

burst). Symmetrically, the input burst *put_req- ptok_r-* causes the end of the handshakes by lowering the acknowledges on both channels.[3]

The specification for *(TD)* is more complex. The circuit starts with a full handshake on the right channel, followed by a full handshake on the *ptok* channel. Once the handshake is complete, it executes a full handshake on the *pass* channel. Similar handshaking is used to handle the *gtok* channel.

The register is controlled by *(PC)* and *(GC)*. The *put_ack* output of *(PC)* is fed to the write enable port of *REG*. The resulting *write acknowledge* signal is used as an acknowledgment on the **put** channel. Similarly, *get_ack* from *(GC)* is used as read enable for the register, and the *read acknowledge* is used as an acknowledgment for the **get** channel.

Finally, the BM specification for the *Starter* is straightforward (Fig. 11). All BM machines were synthesized using the MINIMALIST [20, 21] CAD package.

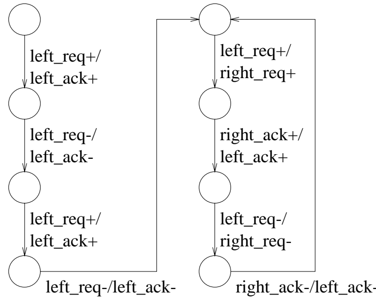

Figure 11: BM specification for the *Starter*

4 Optimized Protocol

In the previous two sections we have discussed the base protocol and its implementation. In this section, we concentrate on ways to improve the performance of our FIFO, both in terms of latency and throughput. Increased performance is obtained by allowing more concurrency both at the program level (parallelizing operations) and the architectural level (overlapping return-to-zero phases with active phases).

4.1 High–Level Protocol

The high–level optimized protocol is given by a Petri Net in Fig. 12. There are two types of parallelization. The first one allows a data operation and the subsequent passing of a token to be overlapped, the token passing operation starting after the data operation starts (the dotted boxes in the figure indicate that token passing starts only after the data operation *begins*, but then they are concurrent). The second parallelization allows the concurrency between dequeuing of data and obtaining the PUT token, and the concurrency between enqueuing data and obtaining the GET token.

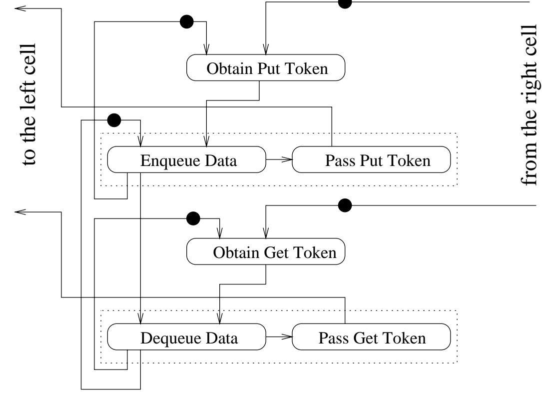

Figure 12: The optimized protocol

The newly introduced concurrency has two effects:

- *higher throughput*: the number of critical actions performed for a data operation is decreased from 3 to 2: obtain token, then do data processing; the resulting token passing is performed in parallel with data processing.
- *lower latency*: the number of critical actions between enqueuing and dequeuing decreases from 2 to at most 1. If *ObtainGetToken* is fast and is complete by the end of enqueuing data, the cell can begin dequeuing data immediately. Otherwise it waits until the GET token is obtained.

The new protocol changes the interface of each cell. There are still two channels communicating with the environment; however, there can no longer be only one channel each on the left and right interfaces. Since obtaining the PUT and GET tokens can be performed in parallel, the actions cannot be multiplexed on single **right** or **left** channels. The new design therefore has two channels on both the left and the right interfaces.

4.2 Cell's Architecture

A block diagram of the cell's architecture is given in Fig. 13. The cell is decomposed into several blocks. *ObtainPutToken (OPT)* and *ObtainGetToken (OGT)* obtain the respective tokens from the right interface, *PutController (PC)* and *GetController (GC)* perform handshaking on the respective **put** and **get** channels and also pass the respective token to the left

[3]Interestingly, the BM specs of Fig. 8 can be shown to be complete and valid specifications of the desired controller's behavior, regardless of whether the cell has a token. In a BM specification, if a proper subset of the input burst is received, any BM implementation must be hazard–free, and the circuit will not have any state/output change. This partial input burst may therefore be asserted and deasserted in turn, arbitrarily, under fundamental mode timing assumptions, and the implementation will be guaranteed to behave correctly. Therefore, any BM implementations synthesized from the above specifications will have the desired behavior.

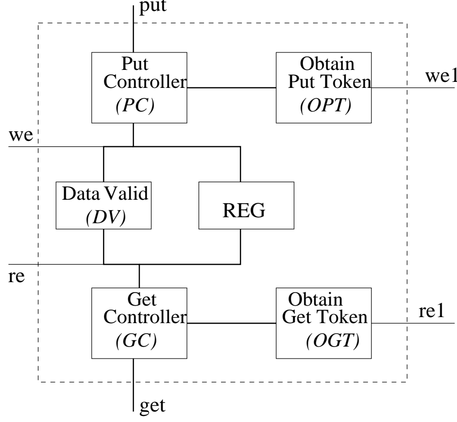

Figure 13: Cell Decomposition (optimized protocol)

cell, and *DataValid (DV)* indicates when the *REG* has valid data (after enqueuing) and when not (after dequeuing).

(PC) is enabled when three conditions have all occurred: the cell has the PUT token, there is a request from the environment on the **put** channel to enqueue data, and data in the cell is not valid (i.e. what was previously there has been dequeued). Once *(PC)* is enabled, it latches data in the *REG* by asserting *we*, communicates to *(DV)* that data is valid, and starts sending the PUT token to the left cell and a put acknowledgment to the environment. At the end of the handshake on **put**, *(PC)* makes the latches opaque, finishes sending the PUT token to the left, and tells *(OPT)* to request the PUT for the next put operation.

(GC) is similarly enabled by three conditions: the presence of the GET token, a request from the environment on the **get** channel, and data validity (i.e. a data item was enqueued). When *(GC)* is enabled, it outputs data from the *REG* onto the **get** bus by asserting *re* and starts sending the GET token to the left. The register will acknowledge to the environment when data is output. At the end of the *get* handshake, *(GC)* tells *(DV)* that data is invalid, finishes sending the GET token to the left, makes the latches opaque and tells *(OGT)* to obtain the next GET token from the right.

There are two points worth mentioning here:

- newly–enqueued data is marked valid *as soon as* the active phase of enqueuing is over, which means that the active phase of dequeuing can be overlapped with the return to zero of enqueuing;
- newly–dequeued data is marked invalid only at the *end* of the dequeuing return to zero. This prevents the cell from overwriting its data while the contents of the *REG* are being output.

The new cell specification has been formally verified to implement a FIFO. Using a trace theory verifier, AVER [14], it was shown that the composition of three cells is *conformation–equivalent* to a FIFO of capacity three: a collection of three cells can be used in lieu of a 3–place FIFO and will have the same observable behavior. The verification can easily be repeated for any number of cells.

4.3 Cell's Implementation

The actual implementation of a cell (Fig. 14) consists of several BM machines synthesized using MINIMALIST (implementing *(OPT)*, *(OGT)*) [20], a machine (*(DV)*) synthesized using Petrify [9], and some small hand–designed components. Note that the left and right interfaces consist of single wires and not of channels. This optimization makes our implementation non–SI, but it increases its speed.

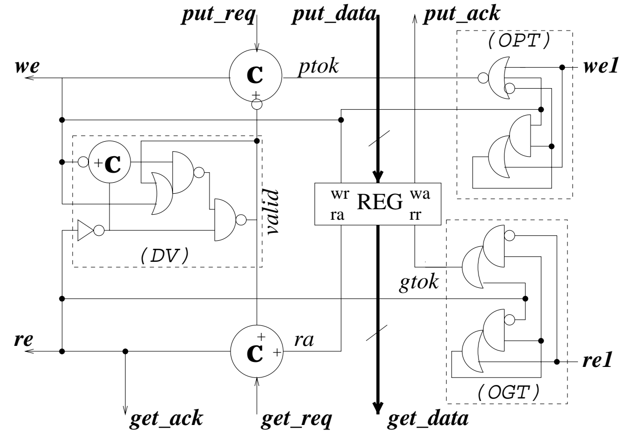

Figure 14: Cell implementation (optimized protocol)

The BM specifications for *(OPT)* and *(OGT)* are shown in Fig. 15. The STG specification for *(DV)* is shown in Fig. 16.

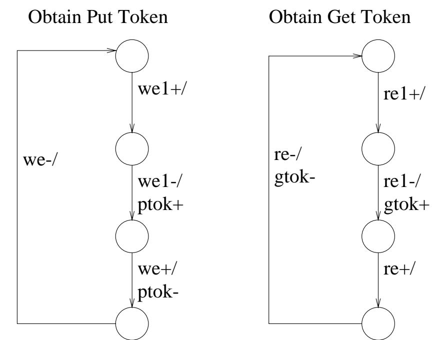

Figure 15: BM specifications for *(OPT)* and *(OGT)*

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.

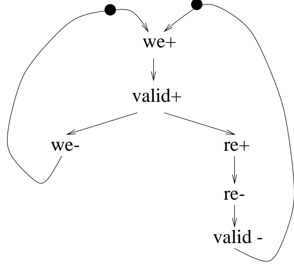

Figure 16: STG specification for *(DV)*

(OPT) will deassert *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 *(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).

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. We wish to drive the output **get** data bus as soon as possible due to the increased load on it. For this, we enable writing to the bus (i.e. enabling a read of *REG*) *as soon as* a cell has the GET token, even if no *get_req* has been issued! The acknowledgement *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).

For this design we do not need an explicit *Starter*. We can simply modify a typical cell's implementation (Fig. 14) to initially contain both tokens. To do so, we add reset logic to initialize *ptok*, *gtok* and the internal state variables for *(OPT)* and *(OGT)* to one.

4.4 Timing Constraints

The above implementation of the FIFO cell is not speed–independent: 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 for the *(OPT)* and *(OGT)* burst–mode machines: the next input must arrive only after the circuit is stable. Similarly, *(DV)* is not QDI or SI 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:

$$\delta_{we\uparrow \rightarrow valid\uparrow\ fedback} < \delta_{we\uparrow \rightarrow put_ack\uparrow \rightarrow put_req\downarrow \rightarrow we\downarrow}$$

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 non–SI cell implementation can be formally verified for conformance to the cell specification using either the Firemaps [22] or ATACS [23] tools. The former tool is not publicly available; the later one required time bounds on the paths through the logic and neither of the authors were proficient in using it. Instead, we have verified conformance by changing the cell specification and using trace theory (AVER). 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.

All BM controllers were synthesized using MINIMALIST [20], and the *(DV)* controller was synthesized using Petrify [9].

5 Results

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

We performed four sets of experiments. Two sets were conducted on FIFO's of capacity 4 and two sets on FIFO's of capacity 16. All FIFO's had data of width 8. Since the delays through the environment are critical for the cycle time, we have also explored two distinct assumptions about the environment: for each FIFO length, we consider both a "slow" and a "fast" path through the environment. The "slow" path has 3 inverters, roughly corresponding to latching of data and placing a new data item on the data input buses. For the "fast" path through the environment we have only one inverter. This is probably an unrealistic path. However, we wanted to test the maximum performance of our circuits assuming an ideal environment.

Special care has been taken to make the simulations realistic. We have used the following models for each global signal:

- *put_req; get_req; put_data*: For the 16–place FIFO appropriate buffering is used for each of these signals to

drive the capacitance of all cells. For the 4–place FIFO, buffering is not needed because, for each design, the load of each signal is only two transistors per cell.[4]

- *put_ack; get_ack*: An explicit tree of OR gates is used to merge the individual acknowledges from each cell into a global acknowledge to the environment.
- *get_data*: Modeling of this global data bus requires special attention. The bus is driven by tri–state buffers. Therefore, both the load contributed by the environment and by the long wires must be modeled. We have assumed that the load provided by the environment is 2 inverters (roughly corresponding to a latch). Since these are pre-layout simulations, we do not know the length of data wires, so we have simply assumed a wire capacitance of 2 inverters per cell.

Two metrics have been simulated for each design: *latency* and *throughput*. Latency is the delay from the input of a data item to its presence at the output (i.e. from $put_req \uparrow$ to $get_ack \uparrow$) in an otherwise empty FIFO. Throughput is defined as the inverse of the cycle time for a get (put) operation.

The results for latency are presented in Table 1. The experiments are labeled: E4F/E4S (4–place FIFO with fast/slow environment) and E16F/E16E (16–place FIFO with fast/slow environment). The best latency (1.73ns) was obtained for the 4–place FIFO, under the optimized protocol. However, for the 16–place FIFO a good latency was still obtained (~2.3ns).

Latency (ns)	Base			Opt.
	Hndsh Circ	Petri Net	BM	
E4S	13.76	12.54	7.94	1.73
E4F	13.75	12.54	7.81	1.73
E16S	14.32	13.01	8.52	2.30
E16F	14.13	12.90	8.41	2.29

Table 1: The latency of the FIFO designs

The results are consistent with our earlier analysis of latency at program level. The better results for the optimization are mostly due to allowing overlap between the active phases of **put** and **get** operations in the *same* cell. Also, as is expected, the latency decreases when the capacity of the FIFO is increased. This is due to the introduction of broadcasting and the increased depth of the acknowledgment tree.

The throughput (in MegaOps/sec) for each design is presented in Table 2. The best throughtput is obtained with the 4–place FIFO with a fast environment (454 MegaOps/sec). For a 16–place FIFO, the results are still good (~350 MegaOps/sec).

As expected, it is observed that the throughput decreases as the FIFO capacity increases. Also, a slow environment is seen to slow down the FIFO's. However, for the base case, using handshake circuits and Petri Nets implementations, the internal paths are longer than those through the environment for all experiments, so for the above experiments the throughput decrease due to slow environment is not observable.

[4] The only exception is the implementation of the Base Protocol with Petri Nets.

Throughput (MegaOps/sec)		Base			Opt.
		Hndsh Circ	Petri Net	BM	
E4S	put	185	200	200	404
	get	161	208	172	427
E4F	put	185	200	204	423
	get	162	216	175	454
E16S	put	175	190	191	335
	get	161	196	164	348
E16F	put	179	195	192	359
	get	162	202	167	367

Table 2: The throughput of the FIFO designs

In general, the table indicates that the throughput of **put** is generally larger than that of **get**: when data is output, the loads attached to the output data wires increase the cycle time. However, there are two exceptions. For the Base Protocol, using the Petri net implementation, the load for *put_req* is larger than that for *get_req*, so the cycle time for **put** is longer. Also, in the optimized protocol, *early* data output is allowed even before a **get** request has been received, so by the time a request arrives, data is already output. Therefore, the **get** cycle time is reduced.

The area for each implementation (expressed in number of transistors) is given in Table 3 for the 4–place FIFO. In adition to the FIFO area (which includes the *Starter*, where appropriate), we only indicate the area of the cell's control part, since the same register is used for every implementation. The register uses 292 transistors, including delay matching and buffering. Again, the design for the optimized protocol has the smallest area, which is expected due to the simplicity of its controllers.

Area	Base			Opt.
	Hndsh Circ	Petri Net	BM	
Cell cntrl	164	167	180	89
FIFO	1914	1983	1968	1545

Table 3: The area of the FIFO designs (# transistors)

The above results indicate that the best implementation is for the optimized protocol, which has very low latency, while maintaining a good throughput for a bus–based design. Also, of all the presented designs, it has the smallest area.

6 Conclusions

This paper has presented and analyzed some novel FIFO designs based on the idea of token passing. The goal was very

low latency while still maintaining high throughput. The base protocol implementation was improved by manipulating the parallelism at the program level and also by introducing architectural optimizations. In particular, the design for the optimized protocol shows the best results with respect to both throughput and latency.

Our designs also have potential for reduced power consumption. It is well known that data immobility may help in saving power. For example, Sparsø [24] cited immobile operands as one key decision in his low–power design. However, for bus–based systems the relation between data immobility and power consumption is not always clear. van Berkel [7] made an argument that there is little power reduction in a data–immobile bus–based design, since broadcasting may increase power consumption. However, we feel that, overall, our bus–based designs have the potential for reduced power consumption.

Acknowledgments

The authors would like to thank Montek Singh for suggesting ideas that led to the optimized protocol and for help with the simulations. We also thank Michael Theobald and Steve Unger for helpful discussions and insights about our designs, and to anonymous reviewers for pointing out several related designs and encouraging us to include formal verification and timing constraints.

References

[1] Ivan Sutherland, "Micropipelines," *Communications of the ACM*, 32(6):720-738, June 1989.

[2] Alain Martin, "The Design of a Self-timed Circuit for Distributed Mutual Exclusion," *Proceedings of the 1985 Chapel Hill Conference on VLSI*, pp. 245-260, Computer Science Press, 1985.

[3] Alain Martin, "Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits," *Developments in Concurrency and Communication*, UT Year of Programming Series, pp. 1-64, Addison-Wesley, 1990.

[4] Jo C. Ebergen, "Arbiters: An Exercise in Specifying and Decomposing Asynchronously Communicating Components," *Science of Computer Programming* 18(1992), pg. 223-245.

[5] Erik Brunvand, "Low latency self-timed flow-through FIFOs," *Advanced Research in VLSI*, pp. 76-90, IEEE Computer Society Press, 1995.

[6] J. T. Yantchev and C. G. Huang and M. B. Josephs and I. M. Nedelchev, "Low-Latency Asynchronous FIFO Buffers," *Asynchronous Design Methodologies*, pp. 24-31, IEEE Computer Society Press, May 1995.

[7] Kees van Berkel and Martin Rem, "VLSI Programming of Asynchronous Circuits for Low Power," *Asynchronous Digital Circuit Design, Workshops in Computing*, pp. 152-210, Springer-Verlag, 1995.

[8] Ad Peeters, "Single–Rail Handshake Circuits," PhD. Thesis, Eindhoven Technical University, 1996.

[9] J. Cortadella and M. Kishinevsky and A. Kondratyev and L. Lavagno and 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.

[10] Steven M. Nowick, "Automatic Synthesis of Burst-Mode Asynchronous Controllers," PhD Thesis, CSL-TR-95-686, Stanford University, Department of Computer Science, 1993.

[11] S.B. Furber, J. Liu, "Dynamic logic in four–phase micropipelines," *Proc. of ASYNC'96*. IEEE Computer Society Press, March 1996.

[12] R. Kol, R. Ginosar, "A doubly–latched asynchronous pipeline," *Proc. of ICCD'96*, pg. 706-711, October 1996.

[13] Charles E. Molnar, I.W. Jones, W.S. Coates, J.K. Lexau, S.M. Fairbanks, I.E. Sutherland, "Two FIFO Ring Performance Experiments" *Proceedings of the IEEE*, 87(2), pp. 297-307, February 1999.

[14] D. Dill, S.M. Nowick, R. Sproull, "Specification and Automatic Verification of Self–Timed Queues," *Formal Methods in System Design* 1:29-60(1992).

[15] A.V. Yakovlev, A.M. Koelmans, L. Lavagno, "High-Level Modeling and Design of Asynchronous Interface Logic", *IEEE Design and Test of Computers*, Spring 1995.

[16] A.V. Yakovlev, "Concurrency Models for Designing Interface Logic in Distributed Systems", *Technical Report 285, Univ. of Newcastle upon Tyre*, November 1989.

[17] K.K. Yi, "The Design of a Self–Timed Low Power FIFO Using a Word–Slice Structure", *M.Phil Thesis, Univ. of Manchester*, September 1998.

[18] T.-A. Chu, "Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications", *PhD Thesis, MIT Laboratory for Computer Science*, Number MIT/LCS/TR-393, June 1987.

[19] M. Kishinevsky, A. Kondratyev, A. Taubin, V. Varshavsky "Concurrent Hardware: The Theory and Practice of Self–Timed Design", Willey and Sons, 1993.

[20] 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.

[21] R.M. Fuhrer, "Sequential Optimization of Asynchronous and Synchronous Finite-State Machines: Algorithms and Tools," PhD Thesis, Columbia University, May 1999.

[22] R. Negulescu, "A Technique for Finding and Verifying Speed-Dependences in Gate Circuits", *Proc. International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU)*, December 1997.

[23] Chris J. Myers, "Computer Aided Synthesis and Verification of Gate-Level Timed Circuits", *PhD Thesis, Stanford University*, October, 1995.

[24] L. S. Nielsen, J. Sparsø, "A Low-power Asynchronous Data-path for a FIR Filter Bank," *Proc. of Async'96*. IEEE Computer Society Press, March 1996.