Design of a Refresh-Controller for GC-eDRAM Based FIFOs

Tzachi Noy, *Student Member, IEEE*, and Adam Teman, *Member, IEEE*

***Abstract*— First-in first-out (FIFO) queues are ubiquitous building blocks in modern system-on-chips. Big FIFOs are often realized as static random access memories (SRAMs), and in many cases account for a significant portion of the area and power consumption of integrated circuits (ICs). Gain-cell embedded DRAM (GC-eDRAM) technology is an embedded memory alternative to the pervasive SRAM technology in ICs. It consumes less silicon area and less power than SRAM, but has the drawback of access blockage caused by its periodic data refreshing. In this paper we leverage the unique access patterns implied by the FIFO scheme to design a FIFO realized with GC-eDRAM. We show that such a FIFO is functionally indistinguishable from a FIFO realized with SRAM. The proposed FIFO has no access blockage time due to refresh, and no data integrity issues, and so can be used as an out-of-the-box replacement for FIFOs in existing and future designs, while providing as much as a 2× reduction in both area and power as compared to SRAM.**

***Index Terms*— First-in first-out (FIFO), embedded dynamic random access memory (eDRAM), gain-cells (GCs), retention time, low power, memory availability.**

I. Introduction

FIFO queues are widely used in digital design, and can be found in any type of application, from networking and storage to multimedia and AI applications [1]–[4]. The use case of the FIFOs in these systems is also very broad, from temporary storage lasting only microseconds to much longer periods, such as in low-energy edge sensors, which may need to buffer data for seconds or minutes between transmissions. For many applications, FIFOs can become so large that they dominate the area and power consumption of the system. For example, the first-in first-out (FIFO) comprises 75% of the route area and power for the network-on-chip based architecture in [5]. While queueing theory and network calculus methods are applied to find the minimal size to support all scenarios, in many cases the FIFO size required to support extreme cases is big, while most of the time it is not fully utilized [6]. This results in both area and power costs that could potentially be avoided.

Small FIFOs are usually implemented using flip-flops, but as FIFOs get bigger, the common practice is to use SRAMs, because for big arrays, SRAMs consume less area per bit. In both implementations, area is proportional to the FIFO size, since the extra data that needs to be stored requires extra storage elements to store it. In both flip-flop and static random access memory (SRAM) implementations, power is also correlated to FIFO size, due to the fact that these extra storage elements consume power regardless of the validity of the data they store.

Embedded DRAM (eDRAM) is an implementation option that consumes less area and can consume less power than same size SRAM [7]–[14]. However, eDRAM suffers from two major drawbacks that cause designers to avoid eDRAMs and keep using the costly SRAMs. First, standard eDRAM requires special process steps for fabrication, which make their integration on many ICs both limited and costly. Second, eDRAM requires periodic refresh operations in order to retain the data. During the refresh cycle, the memory is not available for the system to use. While prior work has shown cases where eDRAM can be used as an SRAM alternative for implementing FIFOs [1], [15]–[17], all of these examples eliminated the refresh mechanism by ensuring that the buffers keep the data for periods shorter than the retention time of the memory. To the best of our knowledge, no previous works have proposed using embedded DRAM (eDRAM) for FIFO implementation, when the requirements do not allow the refresh mechanism to be eliminated.

In this paper we address the two aforementioned drawbacks by introducing a novel approach that enables *out-of-the-box* replacement of SRAM with gain-cell embedded DRAM (GCeDRAM) in a FIFO without affecting the system performance. gain-cell embedded DRAM (GC-eDRAM) is a type of eDRAM that is fabricated in a standard logic technology without the need for special process steps, thereby overcoming the first drawback. The second drawback is overcome by applying an algorithmic approach for refreshing the stored data without disturbing the standard operation of the FIFO. The period for which the data is stored in the FIFO is not bounded, as long as the underlying array meets some criterion for data retention cycles (N_{DR}) to array size ratio. We show this criterion to be $N_{DR} \geq 3S - 1$, where S is the FIFO size. Functionally, the FIFO is indistinguishable from an SRAM based FIFO. Furthermore, this criterion should not be considered a strong limitation on the implementation, as any practical GC-eDRAM array will meet it by design.

A hardware controller that implements the proposed refresh algorithm was implemented and tested within a simulation framework to verify functionality and compare power under

Manuscript received December 24, 2019; revised March 31, 2020; accepted May 25, 2020. Date of publication June 4, 2020; date of current version December 1, 2020. This work was supported in part by the Kamin Program of the Israel Innovation Authority under Project 61907, and in part by the Israel Science Foundation under Grant 996/18. This article was recommended by Associate Editor I. Kale. *(Corresponding author: Tzachi Noy.)*

The authors are with the Emerging Nanoscaled Integrated Circuits and Systems (EnICS) Labs, Faculty of Engineering, Bar-Ilan University, Ramat Gan 5290002, Israel (e-mail: tzachi.noy@biu.ac.il; adam.teman@biu.ac.il).

Color versions of one or more of the figures in this article are available online at https://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCSI.2020.2998582

1549-8328 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Bar Ilan University. Downloaded on December 27,2020 at 12:45:15 UTC from IEEE Xplore. Restrictions apply.

various workloads and access patterns. In addition to the straightforward area savings of GC-eDRAM, as compared to SRAM or flip-flops, the proposed algorithm leads to power savings of up to 50%, as compared to an SRAM-based implementation in 28 nm FD-SOI.

Contributions: The main contributions of this paper can be summarized as follows:

- This paper introduces a GC-eDRAM-based implementation of a generic FIFO to provide an area and power efficient alternative to standard SRAM-based implementations without affecting the system performance.
- This is the first presentation of an eDRAM-based FIFO that ensures 100% availability, such that it can be directly swapped with an SRAM-based FIFO without added system complexity. We provide a proof for the limits of the ratio between FIFO size and eDRAM retention time that will ensure the ability to refresh the memory without data loss.
- We propose an algorithm for refreshing an eDRAM-based FIFO that, in addition to ensuring 100% availability, also minimizes the refresh power consumption of the memory under worst case access patterns.

The rest of the paper is organized as follows. Section II provides an overview of FIFOs, eDRAMs, and the motivation to implement the former with the latter. Section III presents features of FIFOs, which can be exploited by the refresh controller in order to guarantee data validity, 100% availability, and to save power. In Section IV, we present a refresh scheme and show that every data item is refreshed on time, while the normal operation of the FIFO is uninterrupted. Section V gives analytical and experimental results of the proposed algorithms, and conclusions are drawn in Section VI.

II. Overview

In this section, we provide a brief introduction to eDRAM and GC-eDRAM, followed by an overview of FIFOs and the motivation to implement FIFOs with eDRAM. The section concludes with an intuitive representation of the state of FIFOs over time that will be used throughout the manuscript to demonstrate the behavior of the refresh algorithm.

A. Embedded DRAM and Gain-Cell Embedded DRAM

Broadly speaking, volatile embedded memories can be divided into two main categories: static random access memory (SRAM), and embedded DRAM (eDRAM), with SRAM the unequivocal dominate technology. SRAM uses a cross-coupled inverter pair to statically retain the stored data as long as a power supply voltage is provided. eDRAM technology, on the other hand, stores data in the form of electric charge on a capacitor and therefore can be implemented with fewer devices. eDRAM can be further divided into two sub categories:

1) Conventional, one-transistor, one-capacitor (1T-1C) eDRAMs, whose basic bitcell is built from a special, high-density, 3D capacitor and a single access transistor.
2) GC-eDRAM, whose basic bitcell is built from 2–4 MOS transistors.

Conventional 1T-1C eDRAMs typically require special process options to build high-density stacked or trench capacitors. Such process options are only available at an extra manufacturing cost and are not readily available for all technology processes. As opposed to this, GC-eDRAMs are fully compatible with baseline digital CMOS technologies and can easily be integrated into any system-on-chip (SoC) at no extra cost. In addition, the GC-eDRAM bit-cell has separate read and write ports, and so, a GC-eDRAM array is two-ported by nature. This feature inherently addresses the requirement of many FIFOs for concurrent read and write operations. While the motivation for implementing FIFOs with eDRAM is true for both 1T-1C and GC-eDRAM, as explained hereafter, the logic compatibility and two-ported nature of GC-eDRAM provide additional advantages. Therefore, the proposed algorithms will focus on this technology.

While the bitcell size of eDRAM is a clear advantage over SRAM, the stored data is unfortunately compromised due to leakage currents, which results in the requirement for a periodic refresh operation. The number of clock cycles following a write, during which data can be safely retrieved, is called the *data retention cycles* and is denoted by N_{DR}. After this period has passed, reads may yield the wrong value, with error probability increasing over time. In this work we consider N_{DR} as a hard limit, such that a row that has not been written to in the last N_{DR} cycles is considered invalid, and in order to ensure data validity, any row holding valid data must be refreshed at least every N_{DR} cycles.

Algorithm 1 describes a naïve approach for refreshing a GC-eDRAM. N_{DR} and S are constants representing the data retention cycles and size of the FIFO respectively. The algorithm is described in a software-like style, with one procedure calling another, such that the execution of the caller is halted until the callee finishes. While this description is used for didactic purpose, the implementation of the algorithm is not intended to run in software on a micro controller, but to be implemented using simple hardware building blocks, such as logic gates and flip-flops. In the algorithm description, an arrow ($\leftarrow$) represents an immediate (blocking) assignment, while a double arrow ($\Leftarrow$) is used to represent a cycle-delayed (non-blocking) assignment. This enables the representation of a memory read command, which only retrieves the data on the following cycle. Therefore, the data read on line 4, is updated in *dout* only after the wait statement on line 5.

The algorithm consists of two procedures, the TriggeringLoop and the RefreshLoop. The TriggeringLoop is an infinite loop, waiting $N_{\mathrm{DR}} - (S + 1)$ cycles before it calls the RefreshLoop. It is responsible for the idle periods between the active-refresh periods, during which the memory is available to the system. The RefreshLoop procedure carries out the actual refreshing. When RefreshLoop is called, a *busy* flag is raised, indicating to the system that any access to the memory is currently blocked. It then reads a row, writes back the data received on the following cycle, and continues to read the next row. This is done for all rows, and after the last row is written, the busy flag is dropped. The RefreshLoop is responsible for the active-refresh-periods, and it takes $S + 1$ cycles to complete. The full refresh-cycle

Authorized licensed use limited to: Bar Ilan University. Downloaded on December 27,2020 at 12:45:15 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1 Refresh Algorithm for Random Access Memory

```
1: procedure TRIGGERINGLOOP
2:     loop
3:         wait N_DR − (S + 1) cycles
4:         call REFRESHLOOP

1: procedure REFRESHLOOP
2:     busy ← True
3:     for A ← 0 to S − 1 do
4:         dout ⇐ read(address: A)
5:         wait 1 cycle
6:         write(address: A, data: dout)
7:     wait 1 cycle
8:     busy ← False
```

consists of $S + 1$ active-refresh cycles and $N_{\mathrm{DR}} - (S + 1)$ idle-period cycles, so the full cycle is exactly N_{DR} cycles, which ensures each data is refreshed on time.

This algorithm is intended for GC-eDRAM, which is two-ported and allows simultaneous read and write operations. A single ported eDRAM would require a wait cycle between writing the current row and reading the next. In this case the REFRESHLOOP would take $2S$ cycles, and the triggering condition in TRIGGERINGLOOP would have to be changed accordingly.

B. First In - First Out Queues

A first-in first-out (FIFO) queue is an important building block in any modern SoC. FIFOs are used for many purposes, such as buffering, flow control, clock domain crossing, and delay elements, to name a few. A modern SoC can have anywhere from tens to hundreds of FIFOs with different features and sizes, in some cases comprising a significant part of the SoC [18]–[20]. Optimization of this ubiquitous building block can have dramatic effects.

A FIFO is a hardware queue. Logically, new items join the FIFO on one end, and leave when they reach the other. These ends are often referred to as *head* and *tail*; however, different nomenclature exists regarding these terms. We will use tail to imply the end the items are added to and head for the end they leave from. When referring to the order of the items, we start counting from the head of the FIFO towards the tail, enumerating the item at the head as 1st.

In many hardware implementations of FIFOs, the data items do not move through the queue in the way that packages move in the physical world, but rather the FIFO controller keeps track of where the head and tail of the FIFO are. For a FIFO with a dedicated buffer this is usually done by employing a *read-pointer* that points at the location of the next item to be read from the buffer (i.e., the head of the FIFO), and a *write-pointer* that points at the first available location to which the next incoming item will be written (i.e., the one "just behind" the tail). Every time a write operation is performed, the *write-pointer* is incremented, and similarly, the *read-pointer* is incremented after every read operation.

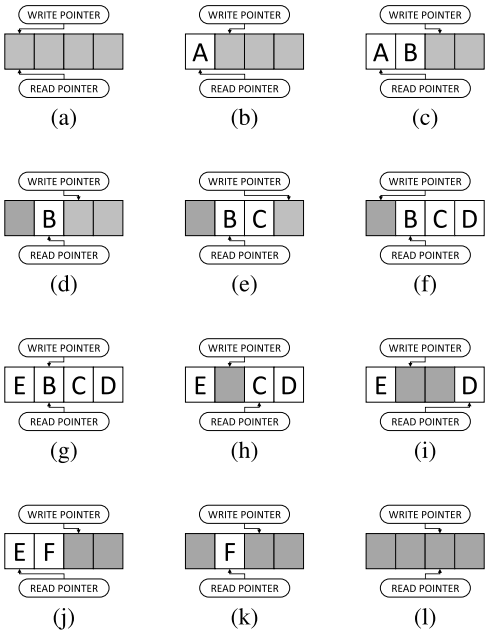


Fig. 1. FIFO operation with *read-pointer* and *write-pointer*. (a) The FIFO is EMPTY, both *read-pointer* and *write-pointer* point at the same location; (b) Item A is written to the location pointed to by *write-pointer* and *write-pointer* points to the next location; (c) Item B is written to the new location pointed to by *write-pointer* and *write-pointer* moves again; (d) Item A is read, *read-pointer* is incremented; (e) Item C is written; (f) Item D is written; (g) Item E is written. *read-pointer* and *write-pointer* point at the same location and the FIFO is FULL; (h) Item C is read; (i) Item D is read; (j) Item D is read out, while at the same time, item F is written. Both *read-pointer* and *write-pointer* are incremented; (k) Item E is read; (l) Item F is read, *read-pointer* and *write-pointer* point at the same location and the FIFO is EMPTY again.

It is important to note that any given buffer has a finite definition of the number of entries it has to store items, and therefore, the FIFO has a finite size. We refer to the maximum number of items that can fit in the FIFO as "fifo size", denoted by S. Once a pointer reaches the last address of the buffer and has to be incremented, it wraps around and starts over by pointing to the first address. If at some point, the *write-pointer* is incremented so that it points at the same location as the *read-pointer*, the FIFO is said to be FULL and no more data can be written before at least one data item is read. Note that the *write-pointer* is not pointing at a free location in this case, but at the location that will become free after the next read, when the item stored in this location will leave the FIFO. In a similar manner, if at some point the *read-pointer* was incremented so that it points to the same location as the *write-pointer*, the FIFO is said to be EMPTY, and no data can be read until new data is written. Similarly, in this case, the *read-pointer* is not pointing to a location holding valid data, but to the location that will become the head once an item will enter the FIFO.

FIFO behavior over time is illustrated in Fig. 1 for a FIFO of size $S = 4$. This illustration shows that once an item enters the FIFO, it never changes location. It shows the cyclic nature of the pointers and also demonstrates the special cases of an EMPTY FIFO (Fig. 1a and Fig. 1l) and a FULL FIFO (Fig. 1g).

Authorized licensed use limited to: Bar Ilan University. Downloaded on December 27,2020 at 12:45:15 UTC from IEEE Xplore. Restrictions apply.

This view of a FIFO is easily implemented with a two-ported memory array, with the described *read-pointer* and *write-pointer* driving the read-address and write-address ports of the memory, respectively. Writing to the FIFO is simply writing to the location pointed at by *write-pointer*, and reading from the FIFO is reading from the location pointed at by *read-pointer*.

C. Motivation for Implementing FIFOs With eDRAM

Replacing SRAMs with eDRAMs in integrated circuit implementations is desired, as in general, an eDRAM is both smaller and in some cases can consume less power than an equivalent SRAM [7], [8], [10], [12].

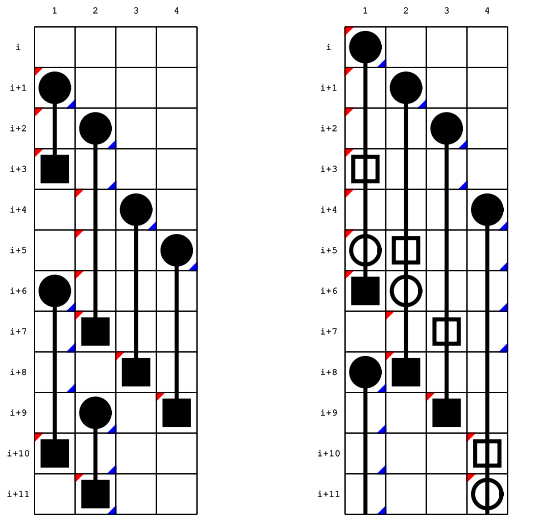
The main drawback of eDRAM compared to SRAM is the need for periodic refresh operations. In addition to the associated power overhead, refresh operations cause the memory to be unavailable to the system for some fraction of the time. However, as we show in this paper, for the purpose of implementing a FIFO, a refresh controller can be devised such that an eDRAM based FIFO is indistinguishable from an SRAM based one, given the eDRAM array meets the relation of data retention cycles versus fifo size, $N_{\mathrm{DR}} \geq 3S - 1$. This relation is derived and proven in Section IV-C.

An additional feature of FIFOs is that they require concurrent read and write operations, which can only be natively achieved with two-ported memories. While in SRAMs, making the array two-ported requires an even larger bit-cell, the GC-eDRAM bit-cell is two-ported by nature. Hence, area savings compared to two-ported SRAM are even more significant. Throughout this manuscript, we refer to a GC-eDRAM based FIFO, but the same should apply to any flavor of two-ported eDRAM.

D. FIFO Visualization Over Time

The FIFO illustrations of Fig. 1 help introduce the underlying mechanisms of a hardware FIFO, depicting the state of the items in the FIFO and the locations of the *read-pointer* and *write-pointer*. In this illustration, each state is represented as a separate drawing, which makes it unsuitable for the depiction of a more complex flow of the FIFO state over time, as required for analysis of FIFO control algorithms. For this purpose we present a novel representation in Fig. 2. By stacking the separate FIFOs that are depicted in Fig. 1, we get a matrix view that concurrently captures the state of the FIFO at every time point. The physical entries of the array are spread horizontally as in Fig. 1, while time is represented on the vertical axis, starting from the top.

In addition to the age and history of each item, by adding a number of symbols, this novel representation also provides the read and write operations that occur over time. The state of the data in the FIFO is illustrated as follows: an empty cell represents an entry that holds no valid data; a filled circle (●) represents an entry being written to; a line represents an entry holding valid data; and a filled square (■) represents an entry being read from. The physical location of data items does not change, and therefore, data items need not be labeled. To further enhance the state representation of the FIFO, colored



(a) FIFO operations (b) Refresh operations

Fig. 2. FIFO state representation over time.

triangles mark the head and the tail of the FIFO, with a red triangle on the top left corner of the cell marking the head, and a blue triangle on the bottom right corner representing the tail.

In the example shown in Fig. 2a, a FIFO of size $S = 4$ is shown for 12 cycles of operation, starting at some point in time (cycle i), during which, the FIFO is empty. The empty state can be immediately recognized, as the topmost row has four empty cells. The triangles for head and tail are not drawn in this row because head and tail are meaningless for an empty FIFO. During cycles $i + 1$ and $i + 2$, two items are written. The first of them is read during cycle $i + 3$, and the second at $i + 7$. During cycle $i + 9$, both a read and a write are issued, represented by a square and a circle on the same row.

Implementing a FIFO using dynamic memory requires periodic refresh of the stored data. A refresh controller may request additional reads and writes, not required for the normal operation of the FIFO, but rather for data integrity. We will refer to these requests as *refresh-read* and *refresh-write*, and for differentiation, we will hereafter refer to the system read and write requests as *fifo-read* and *fifo-write*.

In the proposed visual representation, a *refresh-read* operation is depicted with an empty square (□) and a *refresh-write* operation with an empty circle (◯). Fig. 2(b) shows an example of a FIFO undergoing refresh operations. The refresh occurs during the lifetime of an item. It is first *refresh-read* and later *refresh-written*. In the example, the item in location 1 shows some delay between *refresh-read* and *refresh-write*, the items in locations 2 and 4 experience no delay, and the item in location 3 only experiences *refresh-read*, since it is *fifo-read* (and therefore, evicted from the FIFO) before being *refresh-written*.

Note that during any cycle, only one read operation can occur – either *fifo-read* or *refresh-read*. Similarly, only one write operation can occur during a given cycle – either *fifo-write* or *refresh-write*. Therefore, in this visualization, at most

Authorized licensed use limited to: Bar Ilan University. Downloaded on December 27,2020 at 12:45:15 UTC from IEEE Xplore. Restrictions apply.

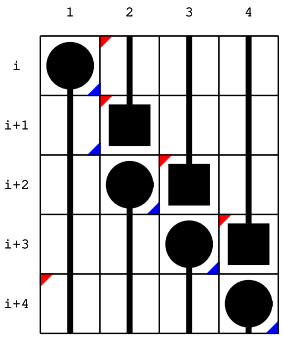


Fig. 3. In a FIFO of size S, the number of *fifo-read* and *fifo-write* operations during the lifetime of a specific data item is bounded by $S-1$. In this example, $S-1$ *fifo-reads* and *fifo-writes* of other items occur during the lifetime of the item in the leftmost column.

one circle and one square (both, either filled or empty) can appear in a single row.

These visualizations will be used in the following sections to introduce and demonstrate the properties of the FIFO and of the proposed refresh controller.

III. FIFO Features Exploitable by the Refresh Controller

In this section, we will describe some features of any finite FIFO which enables the method we describe in Section IV for refreshing a FIFO.

A. Bounded Number of Writes and Reads

A FIFO, being an ordered buffer of finite size, bounds the number of *fifo-reads* and *fifo-writes* the system might issue from the moment a specific item enters the FIFO until it leaves. An item entering the FIFO at the n^{th} position from head, has $n-1$ items ahead of it in the queue, so it will experience exactly $n-1$ *fifo-reads* of these items, before being *fifo-read* itself. Therefore, an item that filled the FIFO will experience the most *fifo-reads* during its lifetime, as compared to other items. The maximum is, therefore, $S-1$ *fifo-reads*.

Analogously, an item reaching the head of the queue, can have at most $S-1$ items queued behind it – all newer. So an item might also experience at most $S-1$ *fifo-writes* of other items during its lifetime.

This bound is visualized in Fig. 3. The number of *fifo-reads* and *fifo-writes*, while the data item in column 1 is valid, are bounded. The next item to be read after 3 reads is the item itself and the 3 write operations following the write of the data item in column 1 make the FIFO FULL. The order and timing of reads and writes might be different than what is shown in Fig. 3, but in any case, the nature of the FIFO bounds the number of reads and writes to $S-1$. We will exploit this fundamental feature of FIFOs to develop a non-blocking refresh controller in Section IV.

B. Notion of Validity

In SRAMs, much of the power dissipation is due to static leakage of the bitcells. In eDRAM, on the other hand, the leakage of the bitcells that causes the decay in the stored data is not taken directly into account in power consumption calculation, because it is the leakage of charges accounted for in write energy. Most of the power consumption in eDRAM is due to read and write operations. However, the storage node leakage indirectly influences the power consumption, because it necessitates refresh operations. The higher the bitcell leakage, the more refresh operations are required per time unit, and hence the higher the power consumption.

In some cases, the size of a FIFO is selected such that the system is guaranteed to work properly even in extreme scenarios. However, these scenarios rarely occur, such that in the average case, the FIFO is far from full. In SRAMs, the power penalty of a larger array is unavoidable, and is the direct result of having a larger number of leaky bitcells. Similarly, in the general case of eDRAMs, the extra rows result in extra refresh operations, which in turn, increase the power consumption.

However, in eDRAM based FIFOs, this does not have to be the case. A FIFO has the notion of which rows hold valid data items and which do not. Only rows between the head and the tail (inclusive) hold valuable data that need to be saved. Therefore, with careful design, the number of refresh operations can be associated with the average fill level of the FIFO and not dictated by the overall size of the memory structure used to implement the FIFO. This can have a dramatic effect on the power consumption in eDRAM based FIFOs, as shown in Section V.

We will exploit this intrinsic feature of FIFOs in the algorithm proposed in Section IV in order to reduce the number of refresh operations. Considering the average fill level of the FIFO as a measure of 'informational-work' done by the FIFO, we get a better proportionality, in the proposed algorithm, between electrical energy and actual 'informational-work' carried out. The more data the FIFO stores, the more power it consumes. Analogously, we can consider the size of the FIFO as 'potential-informational-energy'; so in SRAM-based FIFOs, the electrical energy is proportional to the 'potential-informational-energy' and not the 'informational-work' that was actually done. The more data the FIFO *is able* to store, the more power it consumes, even when very little is actually stored.

C. Strict Ordering

The last feature we want to point out is the ordered fashion of writes and reads in a FIFO. In the general case of a memory – data items are written and read in a random order. Therefore, the age of each item is unknown, unless some timestamp mechanism is used. In a FIFO, on the other hand, the items are known to be written in an ordered fashion, so even though the exact age is unknown, the age relations between the data items are known, including which one is the oldest. We will exploit this feature as well in the algorithm proposed in Section IV.

IV. Refresh Algorithm for a FIFO

In order to directly swap the SRAM memories used for FIFO implementation with GC-eDRAM memories, the FIFO has to adhere to standard interfaces. Specifically, such a

Authorized licensed use limited to: Bar Ilan University. Downloaded on December 27,2020 at 12:45:15 UTC from IEEE Xplore. Restrictions apply.

Algorithm 2 REFRESHLOOP for FIFO

```
1: procedure REFRESHLOOP
2:     refresh-pointer ← read-pointer
3:     refresh-buffer-valid ← False
4:     Ã ← 0
5:     loop
6:         if !fifo-read then
7:             Ã ← Ã + 1
8:         Ã ← min(Ã, 2φ + S − 4)
9:         if fifo-read && read-pointer == refresh-pointer then      ▷ refresh-pointer points to a location being read
10:            refresh-pointer ← refresh-pointer + 1
11:            refresh-buffer-valid ← False
12:            if refresh-pointer == write-pointer then      ▷ stop when refresh-pointer catches up with write-pointer
13:                break from loop
14:        if refresh-buffer-valid && !fifo-write then      ▷ valid data in buffer and no fifo-write
15:            write(address: refresh-pointer, data: refresh-buffer)
16:            refresh-pointer ← refresh-pointer + 1
17:            refresh-buffer-valid ← False
18:            if refresh-pointer == write-pointer then      ▷ stop when refresh-pointer catches up with write-pointer
19:                break from loop
20:        if !refresh-buffer-valid && !fifo-read then      ▷ no data pending to be written and no fifo-read
21:            refresh-buffer ⇐ read(address: refresh-pointer)
22:            refresh-buffer-valid ← True
23:        wait 1 cycle
```

standard interface does not include a "busy" state that tells the system that the memory is undertaking refresh operations. Therefore, the naïve algorithm, presented for simplicity in Algorithm 1, cannot support out-of-the-box replacement of SRAM with GC-eDRAM. In this section, we will introduce a control algorithm that provides this behavior, meaning that the system is never stalled by the algorithm, while it still ensures that all items are guaranteed to be refreshed on time.

Similar to the naïve refresh algorithm for random access memory (Algorithm 1), the proposed algorithm is composed of two procedures. The TRIGGERINGLOOP, is an infinite loop, just like that of Algorithm 1, and is responsible for the idle-periods, during which no refresh is applied. The TRIGGERINGLOOP is essentially waiting for the right time to trigger the REFRESHLOOP procedure. The REFRESHLOOP, similar to the same name procedure in Algorithm 1, is responsible for the active-refresh period.

There are two main differences between the naïve algorithm for random-access memory and the one proposed hereafter (Algorithm 2 and Algorithm 3). First, in the random-access case, the refresh-controller blocks the access to the memory for the system during the active-refresh periods using the *busy* flag. In contrast, the proposed algorithm gives precedence to the system over the refresh-controller, such that the refresh controller reads only on cycles the system does not, and writes only on cycles that the system does not. Therefore, there is no need for a *busy* flag, making it compatible with a standard SRAM interface. The second difference is that in the refresh algorithm for the random-access memory, the refresh is deterministic. The duration of all active-refresh periods is known and constant and takes $S+1$ cycles. The idle periods in that algorithm are also known and constant, and so the full cycle is known and constant. In the FIFO refresh algorithm, on the other hand, neither is constant. Both the active-refresh period and the idle periods change according to the specific pattern of system accesses to the FIFO. Therefore, the refresh-algorithm must consider the access patterns in order to ensure refresh on time, without disturbing the system.

Algorithm 3 Triggering Loop for FIFO

```
1: procedure TRIGGERINGLOOP
2:     Ã ← 0
3:     loop
4:         if φ = 0 then
5:             Ã ← 0
6:             wait 1
7:         else
8:             if Ã + φ + S ≥ N_DRT then
9:                 call REFRESHLOOP
10:            else
11:                if !fifo-read then
12:                    Ã ← Ã + 1
13:                wait 1 cycle
```

Pseudo-code for the REFRESHLOOP and TRIGGERINGLOOP procedures, which comprise the proposed FIFO refresh-algorithm, is provided in Algorithm 2 and Algorithm 3, respectively. The code is written in software-like style, similar to Algorithm 1, but is to be implemented in hardware. S is the FIFO size and the arithmetic of all pointers is modulo S. ϕ is the fill level of the FIFO. *fifo-write*,

Authorized licensed use limited to: Bar Ilan University. Downloaded on December 27,2020 at 12:45:15 UTC from IEEE Xplore. Restrictions apply.

write-pointer, *fifo-read*, *read-pointer* and ϕ are references to the FIFO controller signals and represent the respective final values for the current clock cycle. The proposed algorithm employs a buffer (*refresh-buffer*) for temporary storage of the data to be refreshed, which enables the insertion of a multi-cycle delay between reading out and writing back the data. As in Algorithm 1, simple arrows ($\leftarrow$) are used for immediate assignments, and double arrows ($\Leftarrow$) are used for delayed assignments of data read from memory, in which the the variable on the left hand side is updated on the cycle following the **read** command.

$\widetilde{A}$ is a global variable, meaning that the same variable is accessible from both the REFRESHLOOP and the TRIGGERINGLOOP, and the final value in the inner loop is observable by the outer loop. Similarly, FIFO fill level ϕ is accessible from both loops, but is updated by the FIFO controller according to *fifo-writes* and *fifo-reads*. We defer detailing the meaning and usage of $\widetilde{A}$ and ϕ to the TRIGGERINGLOOP description in Section IV-D.

In order to prove that refresh-on-time is guaranteed without blocking the system, we will first analyze the REFRESHLOOP of Algorithm 2, assuming that the TRIGGERINGLOOP continuously calls the REFRESHLOOP every time a refresh cycle is finished. We will refer to this variant of the algorithm as the "constant-refresh algorithm", and we will subsequently show that it can be expanded into the TRIGGERINGLOOP of Algorithm 3, while maintaining the refresh-on-time and non-blocking constraints.

A. The REFRESHLOOP Procedure

The goal of the REFRESHLOOP procedure, presented in Algorithm 2, is to sequentially read out the FIFO items between the *read-pointer* and the *write-pointer* and write them back into the FIFO. The refresh-buffer is used to temporarily store a single data item until it is written back. The status of the refresh-buffer is tracked by the *refresh-buffer-valid* flag in Algorithm 2; TRUE if the buffer holds data that needs to be *refresh-written*, and FALSE otherwise.

The basic operation of the REFRESHLOOP is to read out an item from the FIFO and store it in the empty buffer when the system is not reading from the FIFO, and similarly to write the buffered item back to the FIFO when the system is not writing to the FIFO. These two operations are covered by the **if** statements on line 20 and line 14 of Algorithm 2, respectively, applying a **read** and/or **write** operation, if these conditions apply.

The *refresh-pointer* is incremented in two cases. Either, the data from *refresh-buffer* was written (line 16), or the location pointed by the *refresh-pointer* was read, so it need not be refreshed (line 10). In both cases, the increment is followed by *refresh-buffer* being invalidated (lines 11, 17), and a check whether the new *refresh-pointer* is equal to *write-pointer* (lines 12, 18) – a condition indicating all items were refreshed, in which case the REFRESHLOOP ends.

Note that the updates applied to the global variable $\widetilde{A}$ (lines 4, 7, and 8) are used by the TRIGGERINGLOOP, and will be explained in Section IV-D.

B. Correctness of the REFRESHLOOP Algorithm

An algorithm that solves the problem in hand has to have two properties:

1) It must not change the normal operation of the FIFO.
2) It must ensure that all valid items are refreshed on time.

The proof of the first property is straightforward; **write** is applied only under the condition that there is no write request by the FIFO controller and **read** is applied only under the condition that there is no read request. Additionally, data-out arrives at the FIFO controller unmodified on the cycle following a *fifo-read*, while the values of the data-out bus during the cycles following the *refresh-reads* are ignored by the FIFO controller, because it did not request to read. Therefore, from the perspective of the FIFO controller, the refresh controller is transparent, proving the first property.

As for the second property, it is easy to show cases where no refresh controller, even blocking ones, can satisfy the requirement. An array of size S with $N_{\mathrm{DR}} == S/2$, for example, cannot be used to realize a reliable memory, since the time it takes to refresh the array is longer than its retention time. Any refresh scheme needs to have a lower bound on N_{DR} of the underlying memory, or otherwise, it cannot guarantee data integrity.

In the following subsection, we will calculate the upper bound on the age of any item in the FIFO for the proposed algorithm. That is, we will calculate the longest time any data item can endure in the FIFO, from the time it is either *fifo-written* or *refresh-written*, until it is *fifo-read* or *refresh-written*, during the active-refresh period. This upper bound on the age of any item is also the lower bound on N_{DR} for this refresh scheme, for which its data integrity can be guaranteed, because it assures that no data in the FIFO can expire. This means that the second property above (i.e., all valid data items are refreshed on time) is true, if N_{DR} is equal to or greater than this age upper bound.

C. Upper Bound on Item Age in Constant-Refresh Algorithm

In the general case, where N_{DR} might be greater than the bound or when the FIFO is not full, the TRIGGERINGLOOP will add delay cycles between the end of one REFRESHLOOP and the beginning of the next in order to save power. This, we will show later, but for the sake of calculating the bound for N_{DR}, we use the constant-refresh algorithm, where the REFRESHLOOP is triggered back-to-back by the TRIGGERINGLOOP.

An item starts aging once it is written, no matter if it was *fifo-written* or *refresh-written* and continues until the data item is either *fifo-read* or *refresh-written*. Once a *fifo-read* is applied, the item is 'popped' from the FIFO and the stored data has no valid value. Following a *refresh-write*, on the other hand, the data is still valid and therefore, its age is reset and the aging process is restarted. The refresh controller has no control over when the item will be *fifo-read*, so it has to refresh all items before they expire.

Let us first examine the refresh pattern when not interrupted by *fifo-reads* and *fifo-writes*. Fig. 4a shows such a case. Note that in cycle i, location 2 is the head of the FIFO while

Authorized licensed use limited to: Bar Ilan University. Downloaded on December 27,2020 at 12:45:15 UTC from IEEE Xplore. Restrictions apply.

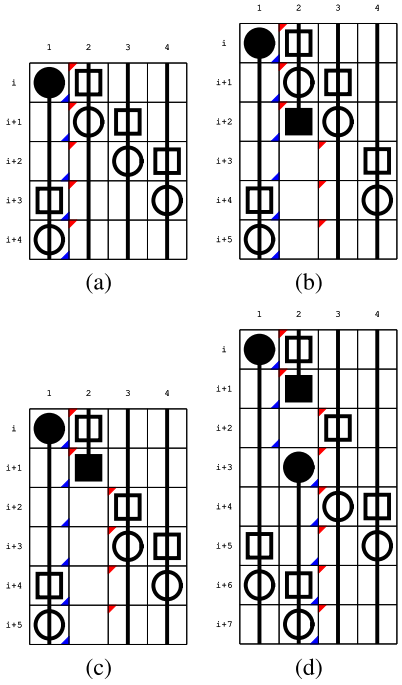


Fig. 4. *fifo-write* to *refresh-write* delay.

location 1 is the tail, so the refresh cycle starts by *refresh-reading* the head and ends with *refresh-writing* the tail during cycle $i + 4$. Also note that the item that was *fifo-written* during the first cycle has no immediate influence on the refresh pattern, because *fifo-write* has effect only when *refresh-buffer-valid* is TRUE (line 14 in Algorithm 2), but the initial value of *refresh-buffer-valid* is FALSE (line 3). This write does eventually influence the refresh cycle, because otherwise, the REFRESHLOOP would have ended one cycle earlier.

Adding a single *fifo-read* during cycles $i+1$ to $i+3$ would have delayed the *refresh-read* in those cycles, and eventually all subsequent refresh operations would be delayed by one cycle. This is demonstrated in Fig. 4b. It shows the same FIFO access pattern as in Fig. 4a with an extra *fifo-read* at cycle $i + 2$, and the resulting refresh access pattern is elongated by one additional cycle. Interestingly, adding the *fifo-read* at cycle $i+1$, instead, would eliminate one *fifo-write*, but would still have the same effect on subsequent refresh operations, as shown in Fig. 4c. Similarly if a *fifo-write* was added to the pattern of Fig. 4c during cycles $i + 3$ to $i + 5$, the *refresh-write* and *refresh-read* would have been delayed on that cycle and for all subsequent refresh operations. This is demonstrated in Fig. 4d.

Note how a *fifo-read* only delays the *refresh-read* that would otherwise occur during the same cycle, while *fifo-write* delays both *refresh-read* and *refresh-write*. This is due to the fact that when a write is pending (*refresh-buffer-valid* == TRUE), *refresh-read* is blocked. Note also, that both *fifo-read* and *fifo-write* delay subsequent refresh operations by one cycle; however, *fifo-write* adds a new item, delaying the end of the REFRESHLOOP by an additional cycle.

Another way to think of it is as a game between the FIFO-controller (playing black), and the refresh-controller (playing white). Every turn, The black selects whether to read and/or to write (following the rules of a FIFO), and the white has to respond. If there exists a strategy for the white to refresh every item on-time for any play the black might choose, then we have a valid solution to the suggested problem. In the strategy suggested in the algorithm, from the moment an item is *fifo-written* until it is *refresh-written*, there can be at most S *refresh-read* and S *refresh-write* operations. With the black doing nothing, this process takes $S + 1$ cycles, as shown in Fig. 4a. Note that Fig. 4a represents the longest period from *fifo-write* to *refresh-write* with black doing nothing. Our assumption that the refresh-controller is constantly in the active-refresh period means it is always refreshing something, and the longest period from *fifo-write* to *refresh-write* is when the *refresh-pointer* is farthest from *write-pointer*. But the black can add *fifo-reads* and *fifo-writes*, each delaying by at most one cycle. As shown above in Section III-A, the black can add at most $S-1$ *fifo-reads* and $S-1$ *fifo-writes* before *fifo-reading* this item. So the time from *fifo-write* to *refresh-write*, might be elongated by no more than $2(S-1)$ cycles. Fig. 5 shows some examples of patterns achieving such a maximal delay.

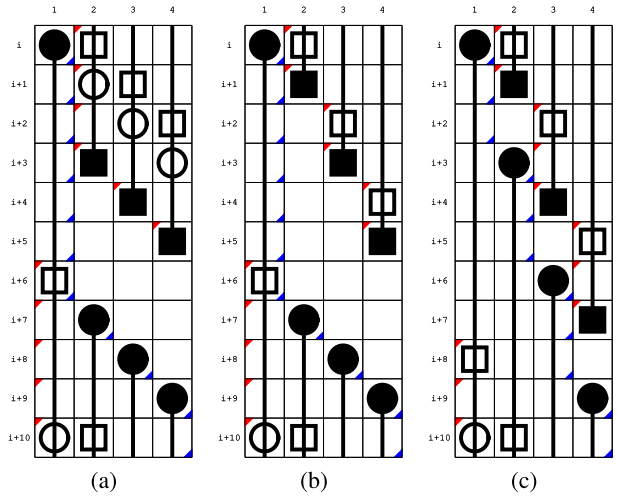


Fig. 5. Examples of maximal *fifo-write* to *refresh-write* delays.

We have found the maximum time from *fifo-write* to *refresh-write*, but similar considerations apply for the time between consecutive *refresh-writes*. The *fifo-write* during the first cycles in all examples of Fig. 4 and Fig. 5, can be replaced by *refresh-write*, and still the refresh pattern will fit the algorithm. Note that this *refresh-write* (that was swapped for the *fifo-write*) finishes a previous REFRESHLOOP, but, as explained with the constant-refresh algorithm, the next REFRESHLOOP begins during the following cycle.

To summarize, our considerations above apply to all data items in the FIFO in a constant active-refresh mode, from the time they are *fifo-written* or *refresh-written* until they are *refresh-written* or *fifo-read*. We found an upper bound on the age of any item in the FIFO during the active-refresh period.

Let A_n be the age of the n^{th} item in the FIFO, then we can write:

$$A_n \leq (S+1) + 2(S-1) = 3S - 1. \quad (1)$$

Authorized licensed use limited to: Bar Ilan University. Downloaded on December 27,2020 at 12:45:15 UTC from IEEE Xplore. Restrictions apply.

Therefore, the suggested constant refresh algorithm meets the second property of refreshing all items before they expire for FIFOs of size S, and N_{DR} adheres to the bound:

$$A_n \leq 3S - 1 \leq N_{\mathrm{DR}}. \tag{2}$$

This means that using the suggested algorithm, an array of size S, which can retain data for (at least) $3S-1$ cycles, can always be used to implement a FIFO of size S.

D. The TRIGGERINGLOOP Procedure

The previous subsections presented and analyzed the REFRESHLOOP algorithm in a constant-refresh variant of the algorithm, where the outer TRIGGERINGLOOP was simplified. It was assumed to trigger the next REFRESHLOOP immediately after finishing the previous one. We now present an energy-efficient TRIGGERINGLOOP algorithm that delays the initialization of the refresh until the latest possible time to lower the refresh frequency and thereby save power, while still guaranteeing data validity.

The TRIGGERINGLOOP algorithm, presented in Algorithm 3, makes use of the two global variables that we disregarded until now: ϕ and $\widetilde{A}$. ϕ is the number of valid items in the FIFO in the current cycle, including any new data item that is written by the FIFO controller during the current cycle. $\widetilde{A}$ is a representation of the age of the oldest item in the FIFO. The proposed algorithm, delays the refresh until $\widetilde{A} + \phi + S \geq N_{\mathrm{DRT}}$. The validity of the algorithm with this delay will be proven in the next subsection.

E. Analysis of the TRIGGERINGLOOP Procedure

The TRIGGERINGLOOP has two objectives: (1) to trigger the REFRESHLOOP so it will ensure data integrity, and (2) to save power by postponing the triggering of the REFRESHLOOP as much as possible. To prove (1) we will find a sufficient condition for the triggering cycle of the REFRESHLOOP which ensures data integrity. In order to save power (objective (2)), the TRIGGERINGLOOP calls the REFRESHLOOP at the very last possible cycle, i.e., when the inequality is met with equality.

When an active-refresh period ends, items are ordered by age, because all items in the FIFO were *refresh-written* during this period, and *refresh-writes* are ordered. During the idle period, the order is kept, because newer items are added in order. Due to the fact that ages are integers, during the idle period we can put an upper bound on A_n, the age of the n^{th} item, given A_1, the age of the oldest. We will mark this bound as $A^{\star}_n$:

$$A_n \leq A_1 - (n-1) \triangleq A^{\star}_n. \tag{3}$$

Let $M_{S,\phi}(n)$ be the maximal time from REFRESHLOOP start, with a fill level of ϕ, until *refresh-write* (or *fifo-read*) of the n^{th} item for a FIFO of size S. To ensure the data integrity of the n^{th} item, it is sufficient to trigger the REFRESHLOOP, while the following condition is met:

$$M_{S,\phi}(n) + A^{\star}_n \leq N_{\mathrm{DR}}. \tag{4}$$

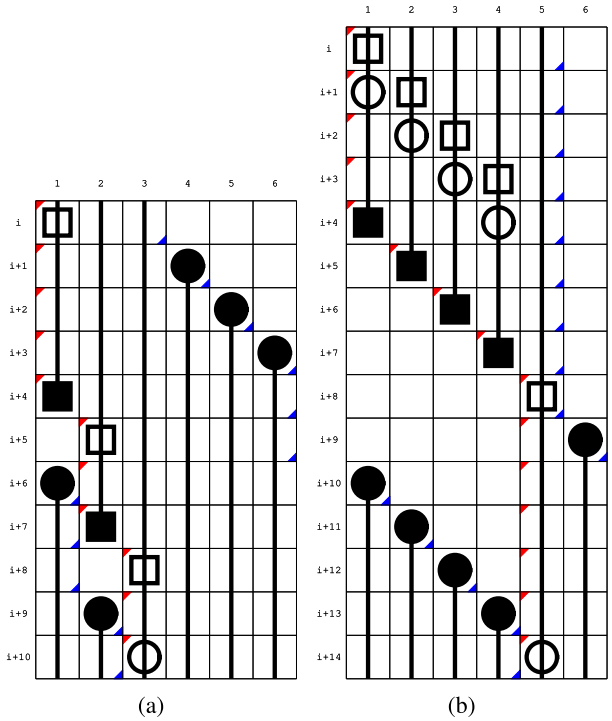


Fig. 6. REFRESHLOOP in a non-full FIFO.

$M_{S,\phi}(n)$ is strictly increasing in n, because items are refreshed in order. Additionally, $M_{S,\phi}(n)$ is integer, so we can write:

$$M_{S,\phi}(n-1) \leq M_{S,\phi}(n) - 1, \tag{5}$$

and from the definition in (3), we have:

$$A^{\star}_{n-1} - 1 = A^{\star}_n. \tag{6}$$

Adding (6) to both sides of (5) provides:

$$M_{S,\phi}(n-1) + A^{\star}_{n-1} \leq M_{S,\phi}(n) + A^{\star}_n. \tag{7}$$

As previously mentioned, (4) is a sufficient condition for starting a REFRESHLOOP such that the n^{th} item is refreshed on time. Its left hand side is non-decreasing, as in (7). Hence, triggering the REFRESHLOOP, such that the last item is refreshed on time ($n = \phi$), ensures that all items in the FIFO are refreshed on time. Accordingly, we can write a sufficient condition for all items to be refreshed on time:

$$A^{\star}_{\phi} + M_{S,\phi}(\phi) \leq N_{\mathrm{DR}}. \tag{8}$$

$M_{S,\phi}(\phi)$ in (8) is the maximal time from the moment a REFRESHLOOP begins for a FIFO of size S with ϕ valid data items, until all those items are *refresh-written* (or *fifo-read*). To calculate $M_{S,\phi}(\phi)$ we will use the same method we previously used in (1) to calculate the bound on the age of items in the constant-refresh algorithm. If no FIFO operations are issued, refreshing ϕ data items takes $\phi + 1$ cycles. Otherwise, there can be at most $\phi - 1$ *fifo-reads* before the ϕ^{th} item is to be refreshed, and at most $S-1$ *fifo-writes*. This is demonstrated in Fig. 6a and Fig. 6b for $\phi = 3$ and $\phi = 5$, respectively, given a FIFO of size $S = 6$.

Authorized licensed use limited to: Bar Ilan University. Downloaded on December 27,2020 at 12:45:15 UTC from IEEE Xplore. Restrictions apply.

Accordingly, the maximum time required by the REFRESHLOOP to finish refreshing all items that were in the FIFO when it started is:

$$M_{S,\phi}(\phi) = (\phi + 1) + (\phi - 1) + (S - 1) = 2\phi + S - 1. \quad (9)$$

As expected, the more items in the FIFO, the longer it could take to complete the REFRESHLOOP. Note that the right hand side of (1) is a special case of (9), with $\phi = S$:

$$A_n \leq M_{S,S}(S) = 3S - 1. \quad (10)$$

So with $M_{S,\phi}(\phi)$, we know that data integrity is guaranteed by triggering the REFRESHLOOP while the following condition is met:

$$A_1 + \phi + S \leq N_{\mathrm{DR}}. \quad (11)$$

And the algorithm uses the complimentary condition for triggering the REFRESHLOOP, substituting A_1 with $\widetilde{A}$:

$$\widetilde{A} + \phi + S \geq N_{\mathrm{DR}}. \quad (12)$$

We now want to show $\widetilde{A}$ is an upper bound on A_1. $\widetilde{A}$ is a global variable, updated in both the TRIGGERINGLOOP and the REFRESHLOOP. In the TRIGGERINGLOOP (Algorithm 3), $\widetilde{A}$ is set to 0 every time the FIFO is empty ($\phi = 0$). When the FIFO is not empty, $\widetilde{A}$ is incremented on every cycle that a *fifo-read* is not issued. This behavior ensures that $\widetilde{A}$ is maintained as an upper bound on the age of the oldest item (A_1), as explained below.

During the TRIGGERINGLOOP, the oldest item in the FIFO is that at the head of the FIFO. Once an item is written into the FIFO, $\widetilde{A}$ starts tracking the age of the first item written, since it is incremented on every cycle. Once a *fifo-read* is applied, the oldest data item is evicted from the FIFO, and therefore, the oldest item is the next item, which is now at the head of the FIFO. $\widetilde{A}$ should now track the age of the new head item, but since the system does not store a time stamp for each FIFO item, there is no information about the age of the new head item, other than that it is newer than the previous (now evicted) head. Therefore, to maintain the bound of the age of the oldest data item in the FIFO, we have to assume the worst case where the new head was written one cycle after the previous head. Conclusively, keeping the value (not incrementing) $\widetilde{A}$ on a *fifo-read* cycle, is sufficient for $\widetilde{A}$ to act as upper bound for the true value of A_1.

In the REFRESHLOOP, $\widetilde{A}$ is set to zero on entrance, and is incremented in a similar manner on cycles during which *fifo-read* is not issued, but is never incremented above $2\phi + S - 4$ (line 8 in Algorithm 2). This maximal value of $\widetilde{A}$ in the REFRESHLOOP arises from considerations similar to the ones above. We know that the oldest item during REFRESHLOOP can be at most $2\phi + S - 4$ cycles old.

We have shown that $\widetilde{A}$ is an upper bound for A_1. This implies that triggering condition in the TRIGGERINGLOOP, is sufficient to guarantee data integrity according to (11), so that the proposed algorithm, consisting of TRIGGERINGLOOP and REFRESHLOOP, guarantees data integrity.

F. Refresh Controller Overhead Estimation

In order to estimate the overhead of the proposed algorithm, we implemented the algorithm as part of a FIFO controller described in Verilog HDL. The hardware implementation, much like the algorithm above, consists of four storage entities, (1) *refresh-pointer*, (2) *refresh-buffer*, (3) age counter ($\widetilde{A}$), and (4) state tracking registers. In terms of area overhead, it can easily be observed that the refresh pointer scales according to $\log(S)$, the refresh buffer scales with the FIFO width (W), and the age counter scales with $\log(N_{\mathrm{DR}})$. The state tracking registers, which are used to track the advancement through the algorithm, are unaffected by the dimensions of the FIFO. So while for a FIFO of size S and width W, the memory array roughly scales according to $O(W \times S)$, the refresh controller area overhead grows much slower, at a rate of $O(W + \log(S))$.[1]

To perceive this relative to an actual FIFO, we synthesized an example FIFO of size $S = 128$ and width $W = 64$. The area of the synthesized controller had a 239 NAND2 equivalent gate count, which is 0.7% of the area of a synthesized SRAM in the target technology. Even if the GC-eDRAM array was 50% smaller than the compiled SRAM, the overhead would still be under 1.5%, which is presumably quite low and the power energy would be very small, as well. With area savings of as much as 50% as compared to an SRAM based FIFO [7], this overhead is indeed worthwhile. Furthermore, as noted above, as the FIFO grows larger, the area (and associated power) overhead of the refresh controller grows at a much smaller rate than the memory array, and therefore, becomes even more negligible. In contrast, for smaller FIFOs the overhead of the refresh controller is more significant, but this is in-line with the general scaling of memories, where for small memories the periphery becomes dominant. Accordingly, very small FIFOs are generally implemented using flip-flops and not SRAM.

V. ANALYTICAL AND EXPERIMENTAL RESULTS

The power consumption of an SRAM based FIFO can be written as:

$$P_{\mathrm{S}} = P^{\mathrm{S}}_{\mathrm{LEAK}} + \lambda F (E^{\mathrm{S}}_{\mathrm{R}} + E^{\mathrm{S}}_{\mathrm{W}}). \quad (13)$$

This term consists of two parts: leakage power, $P^{\mathrm{S}}_{\mathrm{LEAK}}$, which is a constant independent of the access pattern of the system to the FIFO, and dynamic power, which is dependent on the number of writes per second λF, where F is the clock frequency and λ is the ratio of cycles in which *fifo-writes* are issued. Note, that the number of *fifo-writes* and the number of *fifo-reads* are equal for a FIFO, because what goes in must eventually come out. The number of *fifo-writes* per second multiplied by the energy of a single write ($E^{\mathrm{S}}_{\mathrm{W}}$) and a single read ($E^{\mathrm{S}}_{\mathrm{W}}$) thus results in the access pattern dependent part of the power consumption.

For a GC-eDRAM based FIFO with a refresh controller, another term is required to account for the refresh operations. This extra term is dependent on λ_{W} and λ_{R}, the ratio of cycles with *refresh-write* and *refresh-read* operations, respectively. Therefore, we get:

$$P_{\mathrm{D}} = P^{\mathrm{D}}_{\mathrm{LEAK}} + \lambda F (E^{\mathrm{D}}_{\mathrm{R}} + E^{\mathrm{D}}_{\mathrm{W}}) + F(\lambda_{\mathrm{R}} E^{\mathrm{D}}_{\mathrm{R}} + \lambda_{\mathrm{W}} E^{\mathrm{D}}_{\mathrm{W}}). \quad (14)$$

[1]Note that N_{DR} does not increase with FIFO size.

Authorized licensed use limited to: Bar Ilan University. Downloaded on December 27,2020 at 12:45:15 UTC from IEEE Xplore. Restrictions apply.

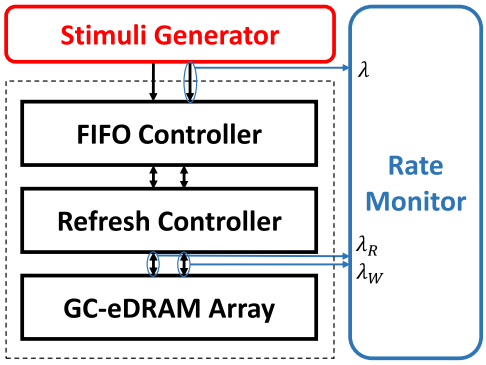


Fig. 7. Test setup for measuring λ_R and λ_W. λ, collected on the top, is the rate of arrivals to the FIFO. λ_R and λ_W, collected at the bottom, are the rate of *refresh-reads* and *refresh-writes*, respectively.

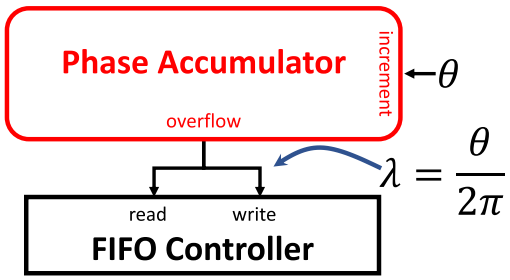


Fig. 8. Stimuli generator for the constant-rate input, constant-rate output scenario. *fifo-reads* and *fifo-writes* occur on the same cycles at which the phase accumulator overflows.

The values of λ_W and λ_R are dependent on the exact sequence of refresh operations. Different arrival and departure sequences to and from the FIFO, yield different refresh sequences, which result in different values for λ_W and λ_R, and hence different energy consumptions. In order to analyze the performance of the proposed refresh controller, we will analyze a subset of the space of all possible access sequences at a time. Arrival and departure patterns can be categorized as in queuing theory. For a specific arrival and departure pair, λ_R and λ_W can be calculated as a function of λ, and so the total energy consumption for the GC-eDRAM based FIFO can be expressed in terms of λ, for a specific arrival and departure pattern. While λ can be analytically derived from the arrival distribution, the refresh operations are dependent on the exact state of the REFRESHLOOP, and can only be analytically derived for very specific scenarios. For the general case, we used a test setup as shown in Fig. 7, where we monitor λ_R and λ_W as a function of λ, for specific stimuli.

We will start by analyzing a simple deterministic case. In order to generate a periodic and deterministic pattern, we generate the read and write requests by the system from the overflow indication of a fixed point phase accumulator [21]. The rate λ, is therefore controlled by the phase increment. Reads and writes are requested by the system at the same rate and on the same cycles in this configuration. This setup is shown in Fig. 8. With this simple setup, we can analytically derive λ_W and λ_R in terms of λ and ϕ (the FIFO fill level), and hence, express the power of a GC-eDRAM based FIFO as a function of these parameters.

Fig. 9 shows analytical results of power (refresh rate) as a function of λ, for several values of ϕ. Fig. 9a and Fig. 9b plot λ_R and λ_W, respectively, while Fig. 9c plots the total power as compared to a reference SRAM. For creating these plots a FIFO of size $S = 128$ and $N_{\mathrm{DR}} = 800$ was chosen as it provides a clear visual representation, however in a typical case, N_{DR} (which is frequency dependent) is expected to be much higher. The periodic nature of *fifo-reads* and *fifo-writes* cause the whole system to have a periodic steady-state behavior with periodic refresh cycles. The time of a refresh cycle in this context is the time from the beginning of one active-refresh period to the next.

The periodic stimuli cause both the active-refresh period and the idle-period to be of almost constant duration, with small deviations due to the differences between the internal state of the controller and the current phase of the accumulator. Over many refresh cycles, the whole system shows a true periodic behavior, which is both an integer number of periods of the stimuli, and an integer number of refresh-cycles. The periodic nature allows us to write λ_R as a ratio N_R/N_C, where N_R is the number of *refresh-reads* in one full refresh-cycle, and N_C is the number of cycles in one full refresh cycle.

We will first analyze the case of $\lambda = 0$, where no data items enter or leave the FIFO. In this case, the refresh controller will require ϕ *refresh-reads* in order to refresh the whole FIFO once, so we can write:

$$N_R|_{\lambda=0} = \phi. \tag{15}$$

The time from the beginning of one refresh cycle to the next is $N_{\mathrm{DR}} - S - \phi$, because no *fifo-reads* are issued in our case and $\widetilde{A}$ is incremented on every cycle, so $N_C = N_{\mathrm{DR}} - S - \phi$ cycles. This means that for a FIFO with $\lambda = 0$, the percentage of cycles with *fifo-reads* is given by:

$$\lambda_R|_{\lambda=0} = \frac{\phi}{N_{\mathrm{DR}} - S - \phi}. \tag{16}$$

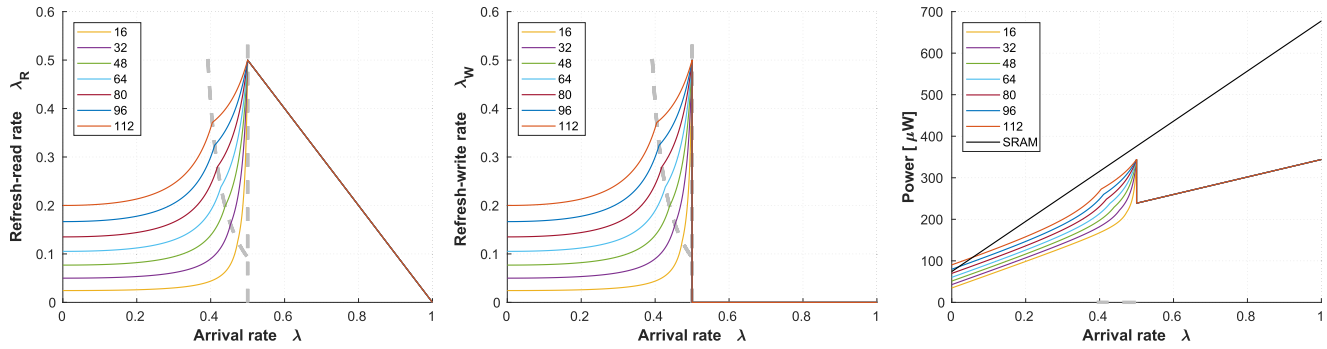
As λ increases, both N_R and N_C increase. N_R is increased because the number of *refresh-reads* per refresh cycle increases, as new items arriving during the active-refresh period will be refreshed as well. Note, however, that as λ increases, the rate of the refresh decreases. The refresh controller operates on cycles the FIFO controller does not, so the rate of refresh in the active-refresh period is $1 - \lambda$ and will be hereafter denoted as $\bar{\lambda}$. Let n be the number of cycles it will take the refresh controller to refresh from head to tail (active-refresh stage). The refresh pointer advances at a rate of $\bar{\lambda}$, and it has to refresh both the items already in the FIFO at the beginning of the active-refresh stage (ϕ) and the ones arriving during the stage (λn). Obviously, for $\lambda \geq \frac{1}{2}$, refresh will never end, but for the case of $\lambda < \frac{1}{2}$ we can write:

$$\bar{\lambda} n = \phi + \lambda n. \tag{17}$$

We can then rearrange (17) to get the number of cycles it will take:

$$n = \frac{\phi}{1 - 2\lambda}, \tag{18}$$

Authorized licensed use limited to: Bar Ilan University. Downloaded on December 27,2020 at 12:45:15 UTC from IEEE Xplore. Restrictions apply.



(a) Refresh-read rate (b) Refresh-write rate (c) Total power

Fig. 9. Power as a function of arrival rate for several fill levels compared to SRAM.

and therefore express N_R as a function of λ :

$$N_R|_{\lambda<\frac{1}{2}} = \bar{\lambda}n = \frac{\bar{\lambda}\phi}{1-2\lambda}. \tag{19}$$

N_C is also affected by the change in λ. For low values of λ, the triggering condition is the same, but the rate at which $\widetilde{A}$ is incremented is slower. Recall that $\widetilde{A}$ is only incremented on cycles with no *fifo-read*, so it too is incremented at a rate of $\bar{\lambda}$. We can therefore write:

$$N_C^{(1)} = \frac{(N_{\mathrm{DR}} - S - \phi)}{\bar{\lambda}}, \tag{20}$$

and now combine (19) and (20) to calculate λ_R :

$$\lambda_R^{(1)} = \frac{N_R}{N_C} = \frac{\bar{\lambda}^2\phi}{(1-2\lambda)(N_{\mathrm{DR}} - S - \phi)}. \tag{21}$$

This calculated value of λ_R perfectly matches the simulation results graphs for small values of λ (Fig. 9a). As λ increases, the active stage takes longer to complete. At the point where $\widetilde{A}$ becomes $2\phi + S - 4$ during the active-refresh period, the limiting condition of the REFRESHLOOP (Algorithm 2) kicks in and $\widetilde{A}$ stops incrementing until the active stage ends. In that case, (20) does not hold anymore. We can calculate N_C for this case as the sum of the cycles of the active-refresh period and the number of cycles of the idle period. We already found the number of cycles of the active-refresh period in (18). The number of cycles for the idle period is the number of cycles it takes for $\widetilde{A}$ to count from $2\phi+S-4$ to $N_{\mathrm{DR}}-S-\phi$ at a rate of $\bar{\lambda}$:

$$\begin{aligned} N_C^{(2)} &= \frac{\phi}{1-2\lambda} + \frac{(N_{\mathrm{DR}} - S - \phi) - (2\phi + S - 4)}{\bar{\lambda}} \\ N_C^{(2)} &= \frac{\phi}{1-2\lambda} + \frac{N_{\mathrm{DR}} - 2S - 3\phi + 4}{\bar{\lambda}}. \end{aligned} \tag{22}$$

So for this region, we can write:

$$\lambda_R^{(2)} = \frac{N_R}{N_C} = \frac{\bar{\lambda}^2\phi}{\lambda\phi + (1-2\lambda)(N_{\mathrm{DR}} - 2S - 3\phi + 4)}. \tag{23}$$

The third region is for $\lambda \geq \frac{1}{2}$. For these cases, the active-refresh phase never ends because $\lambda \geq \bar{\lambda}$. This means that for this region, *refresh-reads* will be issued on every cycle with no *fifo-read*, so we can write:

$$\lambda_R^{(3)} = \bar{\lambda}. \tag{24}$$

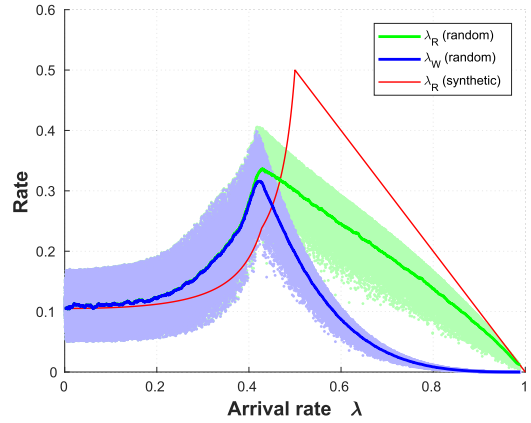
Fig. 9b shows similar results for λ_W. Regions (1) and (2) are similar to the results for λ_R. The same arguments apply; for $\lambda < 0.5$; every item *refresh-read* is *refresh-written*, except for, in some cases, the very last item, so $\lambda_W \leq \lambda_R$, but the difference is negligible. For region (3), $\lambda_W = 0$. The data items that are being *refresh-read*, are always *fifo-read* on the next cycle in case $\lambda \geq 0.5$, so no item is ever *refresh-written*. This is again a result of the strict timing of reads and writes done in this example.

Fig. 9c shows the total power consumption for SRAM and GC-eDRAM according to (13) and (14). Power numbers for both SRAM and GC-eDRAM are based on 28 nm FD-SOI implementations taken from [12].[2] For most fill levels, the power consumption is better for GC-eDRAM than for SRAM. Only for very low values of λ with high fill levels, the power consumption of SRAM based FIFO is slightly better. On the other hand, for higher values of λ, the power savings of GC-eDRAM based FIFOs become significant. In fact, once $\lambda > 0.5$, which is a highly-probable scenario for extensive periods in many FIFOs, the buffered data is read out before a refresh operation is required, essentially nullifying the drawbacks of using a dynamic memory implementation.

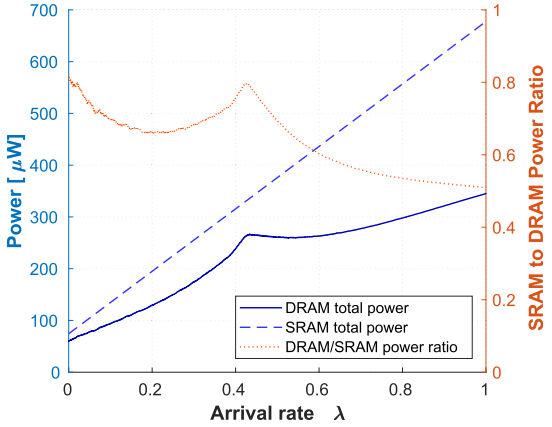
The scenarios discussed so far were synthetic, in order to simplify the mathematical analysis. To demonstrate more realistic scenarios, simulations with random inter-arrival and service times (G/G/1) were run. The results are shown in Fig. 10a for the case of $\phi = 64$. The red line shows the analytic results for the deterministic case discussed above. The light green and light blue dots show the measured λ_R and λ_W (respectively) over 100,000 random simulations. The green and blue lines show the sliding median windows over the samples. The shape of the graph for the random case is similar to the shape of the deterministic case. At $\lambda = 0$ both have the same value, because in this case, the random and the deterministic scenarios are the same. Both graphs rise with increasing rate as λ is increases until they reach some maximum, at which point they start to decrease as λ grows. Note that in the deterministic case, λ_W dropped abruptly to zero at $\lambda = 0.5$ (see Fig. 9b) since at this point any item *refresh-read* was *fifo-read* on the following

[2] $P^{\mathrm{S}}_{\mathrm{LEAK}} = 9.07\,\mathrm{nW}$; $E^{\mathrm{S}}_{\mathrm{READ}} = 0.255\,\mathrm{pJ}$; $E^{\mathrm{S}}_{\mathrm{WRITE}} = 0.498\,\mathrm{pJ}$
$P^{\mathrm{D}}_{\mathrm{LEAK}} = 3.29\,\mathrm{nW}$; $E^{\mathrm{D}}_{\mathrm{READ}} = 0.133\,\mathrm{pJ}$; $E^{\mathrm{D}}_{\mathrm{WRITE}} = 0.263\,\mathrm{pJ}$

Authorized licensed use limited to: Bar Ilan University. Downloaded on December 27,2020 at 12:45:15 UTC from IEEE Xplore. Restrictions apply.



(a) Measured λ_R and λ_W for random patterns with $\phi = 64$. The red plot shows λ_R of the synthetic case as in Fig. 9a.



(b) Power consumption of SRAM vs. GC-eDRAM for random patterns. The dotted line shows DRAM/SRAM power ratio.

Fig. 10. Simulation results for 100,000 random access patterns with $\phi = 64$.

cycle, so it had no time to be *refresh-written*. When a random access is applied, the trend of fewer *refresh-write* operations continues, but in this case, some items do get *refresh-written*, due to the stochastic nature, and so a non-zero level of λ_W is maintained.

The average power consumption of the FIFO was measured from the simulations used to plot Fig. 10a and is shown as a function of λ in Fig. 10b. In this plot, the SRAM power increases linearly with arrival rate, as expected. The GC-eDRAM power follows a similar trend for low values of λ – albeit with 20%–30% lower power consumption than SRAM. However, once λ surpasses 0.4, the GC-eDRAM power increases at a much lower rate, resulting in higher power savings compared to SRAM, reaching as much as 50% at $\lambda = 1$.

VI. Conclusion

First-in First-out queues are a fundamental building block in integrated circuits, and often account for large amounts of silicon real-estate and system power-consumption. Gain-cell embedded DRAM is a high-density, low-power embedded memory solution, which is further compatible with FIFO implementation, as it is inherently two-ported and can lead to power savings proportional to the actual 'informational-work' that is done by the FIFO. In this paper, we propose a control algorithm to ensure that GC-eDRAM refresh operations are hidden from the external user, thereby enabling out-of-the-box replacement of SRAM or flip-flops with GC-eDRAM in FIFOs. The algorithm correctness was shown, including the extraction of bounds for FIFO size versus retention time of the GC-eDRAM array to ensure data integrity. The proposed algorithm was implemented in a hardware controller and simulated under both deterministic and random access patterns, showing close correlation between the analytical and the measured behavior of the FIFO providing as much as 50% power savings, as compared to an SRAM implementation in a 28 nm FD-SOI technology.

Acknowledgment

The authors would like to thank Dr. O. Keren for her invaluable assistance and contribution to the manuscript.

References

[1] K. Lee, S.-J. Lee, and H.-J. Yoo, "A practical method to use eDRAM in the shared bus packet switch," in *Proc. Global Telecommun. Conf. (GLOBECOM)*, vol. 3, Nov. 2002, pp. 2303–2307.

[2] S. Sariga and C. Nandagopal, "An area efficient network on chip architecture using high performance pipelines FIFO technique," in *Proc. IEEE Int. Conf. Electr., Instrum. Commun. Eng. (ICEICE)*, Apr. 2017, pp. 1–5.

[3] F. Hassen and L. Mhamdi, "A scalable multi-stage packet-switch for data center networks," *J. Commun. Netw.*, vol. 19, pp. 65–79, Feb. 2017.

[4] W. Choi, K. Choi, and J. Park, "Low cost convolutional neural network accelerator based on bi-directional filtering and bit-width reduction," *IEEE Access*, vol. 6, pp. 14734–14746, 2018.

[5] P. Ou *et al.*, "A 65nm 39GOPS/W24-core processor with 11Tb/s/W packet-controlled circuit-switched double-layer network-on-chip and heterogeneous execution array," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2013, pp. 56–57.

[6] C. Soviani and S. A. Edwards, "FIFO sizing for high-performance pipelines," in *Proc. Int. Workshop Logic Synth. (IWLS)*, May 2007. [Online]. Available: http://www.iwls.org/iwls2007/, doi: 10.7916/D8RB7F0B.

[7] P. Meinerzhagen, A. Teman, R. Giterman, N. Edri, A. Burg, and A. Fish, *Gain-Cell Embedded DRAMs for Low-Power VLSI Systems-on-Chip*. Cham, Switzerland: Springer, 2018.

[8] A. Teman, P. Meinerzhagen, A. Burg, and A. Fish, "Review and classification of gain cell eDRAM implementations," in *Proc. IEEE 27th Conv. Elect. Electron. Eng. Israel*, Nov. 2012, pp. 1–5.

[9] R. Giterman, A. Teman, P. Meinerzhagen, L. Atias, A. Burg, and A. Fish, "Single-supply 3T gain-cell for low-voltage low-power applications," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 1, pp. 358–362, Jan. 2016.

[10] A. Teman, G. Karakonstantis, R. Giterman, P. Meinerzhagen, and A. Burg, "Energy versus data integrity trade-offs in embedded high-density logic compatible dynamic memories," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*. San Jose, CA, USA: EDA Consortium, 2015, pp. 489–494.

[11] N. Edri, P. Meinerzhagen, A. Teman, A. Burg, and A. Fish, "Silicon-proven, per-cell retention time distribution model for gain-cell based eDRAMs," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 63, no. 2, pp. 222–232, Feb. 2016.

[12] R. Giterman, A. Fish, A. Burg, and A. Teman, "A 4-transistor nMOS-only logic-compatible gain-cell embedded DRAM with over 1.6-ms retention time at 700 mV in 28-nm FD-SOI," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 65, no. 4, pp. 1245–1256, Apr. 2018.

[13] R. Giterman, A. Fish, N. Geuli, E. Mentovich, A. Burg, and A. Teman, "An 800-MHz mixed-V_T 4T IFGC embedded DRAM in 28-nm CMOS bulk process for approximate storage applications," *IEEE J. Solid-State Circuits*, vol. 53, no. 7, pp. 2136–2148, Jul. 2018.

[14] R. Giterman, R. Golman, and A. Teman, "Improving energy-efficiency in dynamic memories through retention failure detection," *IEEE Access*, vol. 7, pp. 27641–27649, 2019.

Authorized licensed use limited to: Bar Ilan University. Downloaded on December 27,2020 at 12:45:15 UTC from IEEE Xplore. Restrictions apply.

[15] G. Kang, W. Choi, and J. Park, "Embedded DRAM-based memory customization for low-cost FFT processor design," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 12, pp. 3484–3494, Dec. 2017.
[16] Y. S. Park, D. Blaauw, D. Sylvester, and Z. Zhang, "Low-power high-throughput LDPC decoder using non-refresh embedded DRAM," *IEEE J. Solid-State Circuits*, vol. 49, no. 3, pp. 783–794, Mar. 2014.
[17] W. Choi, G. Kang, and J. Park, "A refresh-less eDRAM macro with embedded voltage reference and selective read for an area and power efficient Viterbi decoder," *IEEE J. Solid-State Circuits*, vol. 50, no. 10, pp. 2451–2462, Oct. 2015.
[18] P. P. Pande, C. Grecu, A. Ivanov, and R. Saleh, "Design of a switch for network on chip applications," in *Proc. Int. Symp. Circuits Syst. (ISCAS)*, vol. 5, May 2003, p. 5.
[19] P. Wielage, E. J. Marinissen, M. Altheimer, and C. Wouters, "Design and DfT of a high-speed area-efficient embedded asynchronous FIFO," in *Proc. Design, Autom. Test Eur. Conf. Exhib.*, Apr. 2007, pp. 1–6.
[20] A. Sheibanyrad, A. Greiner, and I. Miro-Panades, "Multisynchronous and fully asynchronous NoCs for GALS architectures," *IEEE Design Test Comput.*, vol. 25, no. 6, pp. 572–580, Nov. 2008.
[21] H. Hikawa and S. Mori, "A digital frequency synthesizer with a phase accumulator," in *Proc. IEEE . Int. Symp. Circuits Syst.*, vol. 1, Jun. 1988, pp. 373–376.

Tzachi Noy (Student Member, IEEE) received the B.Sc. and M.Sc. degrees in electrical engineering from Bar-Ilan University, Ramat-Gan, Israel, in 2007 and 2019, respectively. He is currently pursuing the Ph.D. degree in electrical engineering with the Emerging Nanoscaled Intergrated Circuits and Systems (EnICS) Labs, Bar-Ilan University, under the supervision of Dr. A. Teman. His research interests include EDA optimization heuristics, hardware implementations for low-power machine learning applications, design of very large-scale integration circuits, and high-performance computer architectures.

Adam Teman (Member, IEEE) received the Ph.D. degree in electrical and computer engineering from Ben-Gurion University (BGU), Be'er Sheva, in 2014 under a Kreitman Fellowship. He worked as a Design Engineer with Marvell Semiconductors from 2006 to 2007, with an emphasis on Physical Implementation. From 2014 to 2015, he was a Post-Doctoral Researcher with the Telecommunications Circuits Lab (TCL), École Polytechnique Fédérale de Lausanne (EPFL), Switzerland, under a Swiss Government Excellence Scholarship. In 2015, he joined the Faculty of Engineering, Bar-Ilan University (BIU) in 2015 as a Tenure Track Senior Lecturer at the Department of Electrical Engineering and as the Co-Director of the Emerging Nanoscaled Integrated Circuits and Systems (EnICS) Labs Research Center. He has authored over 80 scientific articles and 8 patent applications, and has participated in over 15 IC tapeouts. He is the coauthor of the recently published book *Gain-Cell Embedded DRAMs for Low-Power VLSI Systems-on-Chip*, available from Springer. His research interests include embedded memories, energy efficient circuit design, hardware for artificial intelligence, hardware acceleration, and methodologies for physical implementation. Dr. Teman is a member of the technical and review boards of several conferences and journals. He has been honored with Teaching Excellence recognitions at both BGU and BIU and has received multiple awards for outstanding research, including the Wolf Foundation Scholarship and the Intel Prize. Since 2020, he has been awarded the Krill Prize for outstanding young researchers and the BIU Rector's prize for outstanding research. He is an Associate Editor with the *Microelectronics Journal*.

Authorized licensed use limited to: Bar Ilan University. Downloaded on December 27,2020 at 12:45:15 UTC from IEEE Xplore. Restrictions apply.