# Compute-Line based Computational Memory Architecture supporting Binary Logic with two Coloring States

**Driss Azougagh, Ahmed Rebbani and Omar Bouattane**

SSDIA Laboratory, ENSET Mohammedia, Hassan II University of Casablanca
BP 159 Bd Hassan II, Mohammedia Morocco

**Summary**
Processing In Memory (PIM) is in demand more than ever to cope with the growth of Big Data, memory wall and power wall. It eliminates the overhead of data movement between processing unit and memory resulting in high bandwidth, massive parallelism, and high energy efficiency. Most existing PIM works are concentrated on near-memory processing (NMP) and/or in-memory processing (IMP). In this paper we present a compute-line based computational memory architecture (CCMA) supporting in compute-line processing, or simply compute-line, using a controlled inverter (CINV) for pull-down and/or pull-up the line. In one clock cycle, with one single and simple instruction, the compute-line allows multiple pull-downs and/or pull-ups for some bitwise logic computation and multiple writes, simultaneously.
The architecture is easily backward compatible with conventional Static/Dynamic Random Access Memory (SRAM/DRAM) but it has advantage of not using bit-line pre-charging and sensing for read and write operations. It reduces bit-line activities by more than half. When used as an in-memory computing, it also eliminates overhead of data movement demonstrating a great potential to reduce bandwidth and energy consumption. The proposed compute-line is validated and tested to show considerable performance and effectiveness according to the new capabilities offered. This architecture can support a variety of interconnect topologies between multiple compute-lines which will benefit many parallel applications.
***Key words:***
*SRAM Memory, Built-in Computing, compute-line, in-place processing.*

## 1. Introduction

The growth of Big Data, memory wall and power wall are posing unprecedented demand for Processing In-Memory (PIM) [1]. The PIM solutions proposed to move processing logic near the cache [2], [3] or main memory [4], [5]. 3D stacking can make this possible [6]. Compute Caches significantly push the envelope by enabling in- place processing using existing cache elements, where at least one of the operands used in computation has cache locality for data-centric applications. Furthermore, for on-chip processors caches are not necessary. Several Processor-in-Memory architectures have been proposed [7, 8], and some have been implemented [9, 10, 11].

Authors in [12] proposed Compute Caches architecture that uses an emerging SRAM circuit technology, which can be referred to as bit-line computing [13], [14]. A study of logical memories based on Akers logical arrays and generalize these arrays to non-binary symbols was introduced in [15]. Akers logical arrays are particular for symmetric binary and sorting functions. For deep-neural-network (DNN) processors [16], the product-sum (PS) operation predominates the computational workload for both convolution (CNVL) and fully-connect (FCNL) neural-network (NN) layers.
Static/Dynamic Random Access Memory (SRAM/DRAM) has been the predominant technology used to implement memory cell in computer systems [17]. In conventional SRAM (and/or DRAM), bit-lines need to be pre-charged prior any read operation and requires differential sensing or amplification of its voltage levels. To address the challenges of sub-Vt SRAM, in [18], an 8T bit-cell that uses two port topology with 6T storage cell and a 2T read-buffer to isolates the data-retention structure during read-accesses. The two port topology is used to eliminate read Static Noise Margin (SNM) and peripheral assists, controlling Buffer-Foot and VDD to manage bit-line leakage and write errors. The author proposed a low-power non precharge-type two-port SRAM for video processing. In their prior study that saves the charge/discharge power on a read bit-line, a majority logic circuit and data-bit reordering are accommodated to write "1"s in as many as possible [19] as MJ SRAM.
The focus was more on field-programmable gate array (FPGA)-based solutions using a semiconductor device on which designers can reprogram desired digital circuits. In recent years, FPGAs are used in both high-performance systems [20] and embedded systems [21].
Today, in [22], RRAM was viewed as a promising candidate that can meet future storage and computing needs. Author discussed potential computing applications enabled by RRAM devices within both conventional and emerging computing paradigms and introduced a concept of RRAM based Memory Processing Unit (MPU). In their investigation, due to the sneak-path effect and the tradeoff between data retention and endurance, device-level and system level innovations are still needed for large-scale implementation and storage systems. RRAM device tends

Manuscript received June 5, 2019
Manuscript revised June 20, 2019

to be applicable for LUT-based circuitry to store truth table of a Boolean function and for programmable switch to link other CMOS sub-circuits in field-programmable gate arrays (FPGAs).

In our previous work [24], we introduced a bit-line based computational memory architecture (BMCA) that uses select-line to choose in which bit-line to run an operation. It uses a bit-line KEEPER to stabilize the selected bit-line when all memory cells involved in a read operation have their bit value is set to 0. In this work, we tried to get rid of the KEEPER and its overhead imposed. In contrary, in this paper, we introduce a pull up and pull down logics to establish all other combinational logic instead of the complete logical operations NAND, NOR, or a combination of NOT with AND or OR.

We present a compute-line based Computational Memory Architecture (CCMA) that supports either Static or Dynamic Random Access Memory (SRAM or DRAM). The logic read operation is based on either pulling up the bit-line if one of the involved memory cells has its memory state is low or pulling down the bit-line when one of the involved memory cells has its memory state is high. If both cases fail to satisfy the condition, the bit-line keeps its state unchanged. The final bit-line result can be applied to a memory cell by activating its write line. A detailed study and analysis of the CCMA based DRAM is introduced. In the next section we start with an introduction of the compute-line based computational memory architecture. In the third section we describe it functionality and analyze the results. Finally, we conclude with perspective remarks.

## 2. Computational Memory Architecture

A Compute-line based Computational Memory Architecture (CCMA) is composed of m compute-lines as shown in Fig. 1. The architecture can support two different types of memory cell technologies; a static memory cell (SMC) using two superposed inverters based on MOSFET transistors and a dynamic memory cell (DMC) using Charging/Discharging capacitor as in Fig. 1(f) and (g), respectively. The logic operation is based on either charging selected memory cells for write if one of the memory cells selected to pull-up the compute-line is not charged, discharging selected memory cells for write if one of the memory cells selected for pull-down is charged or do nothing when no memory cell is involved in pull-down/up or no write operation. A detailed analysis of the CCMA based DRAM is introduced. The drawing line in gray colors are only for SMC based architecture needs. Without loose of generality and due to the symmetry in the drawing of the black and gray lines, the attention will be mainly given to the black lines except when the gray ones are explicitly mentioned.

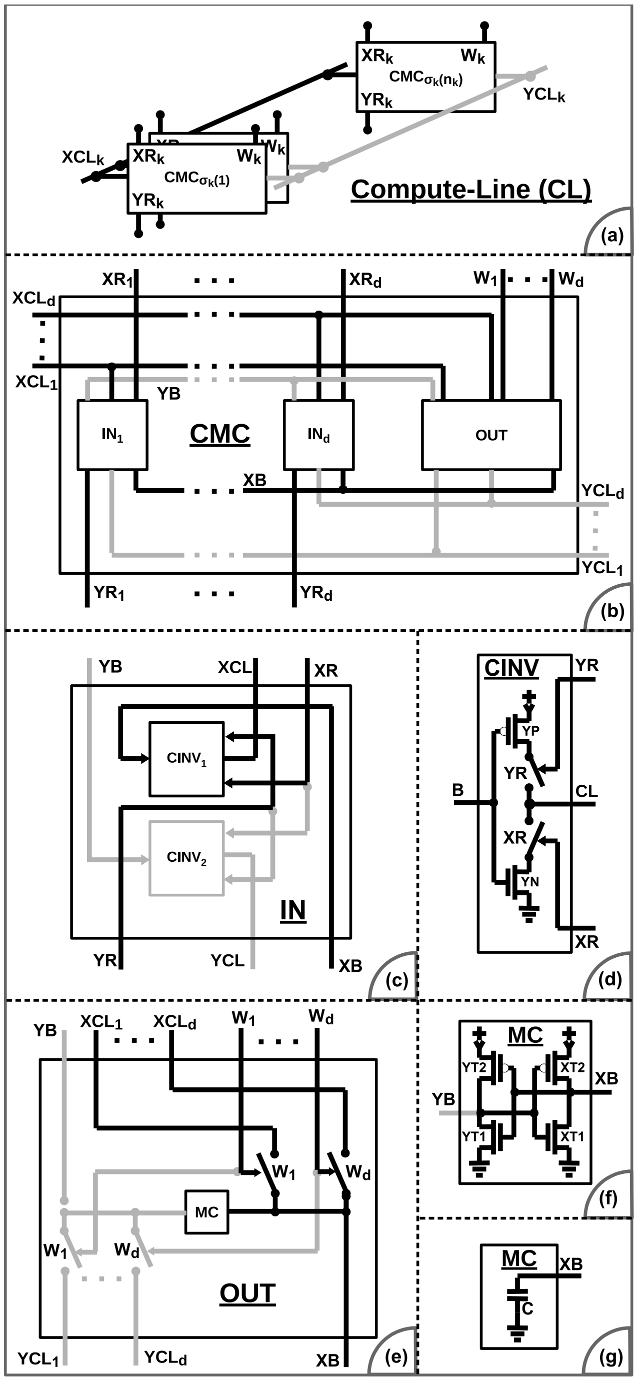

Fig. 1 Bit-Line based Computational Memory Architecture

### 2.1 Compute-line

A compute-line XCL$_k$ (and YCL$_k$) can connect any selected number $n_k$ of computational memory cells (CMCs) based on a selection function σ$_k$, as in Fig. 1(a). The choice of the selection functions ($\sigma_1$,…,$\sigma_m$) determines which topology the CCMA architecture might use.

## 2.2 Computational Memory Cell

In Fig. 1(b), a CMC can be composed of one output (OUT) block and any number *d* of input blocks ($IN_1,\ldots,IN_d$) as required by the chosen CCMA topology. The output block and input blocks share the same internal bit-line XB (and YB in case of SMC). Each input block $IN_i$ shares a compute-line $XCL_i$ (and $YCL_i$) with the output block OUT. For a given CMC, a compute-line $XCL_i$ (and $YCL_i$) with no connection to other CMCs need to be omitted with their corresponding input block $IN_i$ during simulation and/or chip fabrication.

## 2.3 Input Block

An input block IN in Fig. 1(c) is composed of one controlled multi-phase inverter $CINV_1$ (or two $CINV_1$ and $CINV_2$ in case of SMC) shown in Fig. 1(d). It is worth mentioning that in case of SMC, the control line XR of IN is wired to both XR of $INV_1$ and YR of $CINV_2$. Symmetrically, the control line YR of IN is wired to both YR of $CINV_1$ and XR of $CINV_2$. This way, the inverter's phase selected by $CINV_1$ is guaranteed to be the opposite of the other inverter's phase selected by the $CINV_2$.

## 2.4 Output Block

The output block OUT is summarized in Fig. 1(e) and is constructed based on switches connecting local bit-line XB with the compute-lines XCLs using write lines (Ws). As mentioned above, the Memory Cell (MC) of the output block can use either technologies SMC and DMC. In case of SMC, activated write line $W_i$ causes a connection between the bit-lines (XB and its inverse YB) with the compute-lines $XCL_i$ and $YCL_i$, respectively.

## 2.5 Controlled Inverter

The controlled multi-phase inverter CINV can have up to 4 phases; de-active when both switches XR and YR are off, pull-down when only the switch XR is on, pull-up when only the switch YR is on and invert when both XR and YR are on.

## 2.6 Dynamic Compute-Line

For illustration, Fig. 2 shows an example of one simple single compute-line based on DMC where each computational memory cell CMC has one input block composed of one controlled inverter, one output block and one dynamic memory cell (a single capacitor). For deep study and analysis of the CCMA architecture, we choose DMC based technology due to its complicated behaviors when capacitors are charging and discharging and it includes most cases of the limited and deterministic behaviors of SMC based technology. To apply input data to the circuit during simulation, we introduced an input IN line with three logical states 0, 1 and high impedance. This line only affect and alter the value stored in XB0. In the next section, we will show that we can inject whatever data to the circuit without using the introduced IN line.

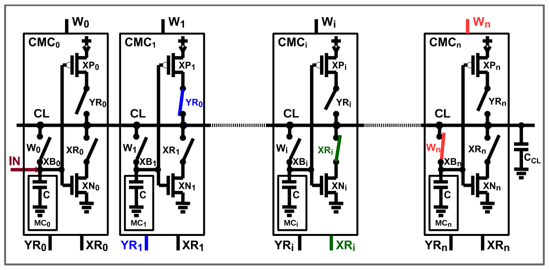

Fig. 2 Dynamic Compute-Line

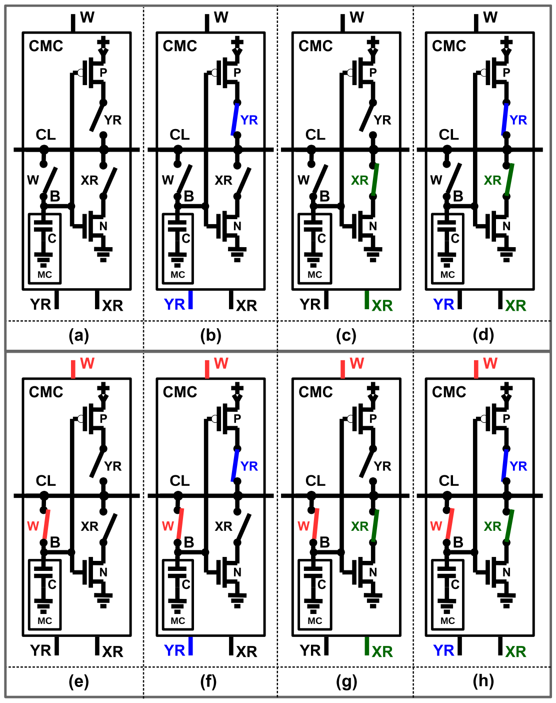

Fig. 3 CMC control commands; eight possible combinations (a) ~ (h).

All the eight possible states of controlling a CMC are summarized in Fig. 3. Among these states, it is worth mentioning that the last three cases Fig. 3(f), (g) and (h) might cause instable outcomes and conflicts if the total capacitor of the compute-line XCL is not large enough to cancel the recursive effect given that all read lines is always applied before any write line.

## 3. Functionality & Results

A compute-line operation is achieved by using pull-down and/or pull-up as basic operations with multiple input operands. The compute-line operation is complete and can compute all bit-wise basic logic operations including binary ones such as NOT, NOR and NAND. If we define 1 and 0 as stables states for a MC, in case of DMC, the stability of the bit information is achieved when the capacitor can sustain its charge or discharge level sufficiently higher or lower than NMOS or PMOS transistor thresholds, respectively, for a considerable period of time (at least hundreds of microseconds).

If no computational memory cell is pulling-up/down the compute-line during an operation then the new voltage level of the compute-line can be theoretically determined using the formula in Eq. 1, where λ is the ratio between the capacitances $C_{CL}$ and C of the compute-line and a memory cell's capacitor, respectively. However, the real and exact voltage level of the compute-line using this formula for existing (semi-conductor) technologies is not practical. Therefore, The CMC controller role is to avoid any non-deterministic state that might cause either conflict between pull-downs and pull-ups and/or between multiple compute-line writes. The search for the right combination of commands for each time we wanted to execute a set of operations is the role of the Compute-line controller.

$$V_{CL-new} = \frac{\lambda * V_{CL-old} + \sum_{i=0}^{n} W_i * V_{XB_i}}{\lambda + \sum_{i=0}^{n} W_i} \tag{1}$$

### 3.1 Compute-Line Controller

A timing diagram for the proposed dynamic compute-line with two different setups and an identical scenario is shown in Fig. 4. The difference choice of setups affects only the computation's speed and in some cases the stability of the compute line. The controller need to pre-compute which simultaneous combination of pull up and pull down is allowed to avoid collision. The second setup uses extra four computational memory cells ($CMC_9$ through $CMC_{12}$) as a compute-line's capacitor buffer by activating their write line (W9 through W8) for all the time while executing compute-line operations during the simulation. The smaller the compute-line's capacitor the smaller the time consumed to execute compute-line operations. The first setup, without using a capacitor buffer, registers 26% reduction in the total execution of the same scenario compared to the second setup.

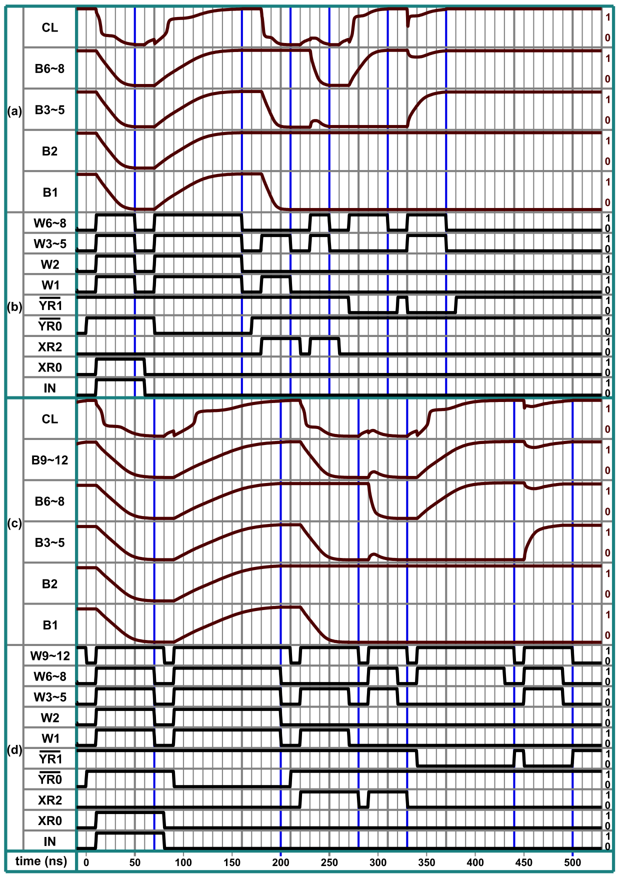

Fig. 4 Dynamic compute-line behavior.

### 3.2 Pull-down/up Combinatory Logic

The pull-down, pull-up and write functions used to compute any non-conflicting compute-line operation is summarized in Eq. 2 and 3, respectively. Without loose of generality, a series $(a_i)_k$ is used to start from 0 for a total of $k$+1 values and can be updated for convenience to start from 1 or to end up to $k$-1. A program, function or a peace of code can be composed of any set of such pull-down/up-write instructions. Any set of such instructions need to be off-line studied in advance for all possible input data to avoid any conflict that might lead to inconsistent results.

$$CL = CL \downarrow (XR_i * B_i)_n \uparrow (YR_i * B_i)_n \tag{2}$$

$$\text{where}$$

$$x \downarrow (a_i)_k = x * \overline{\sum_{i=0}^{k} a_i} \quad \text{and} \quad x \uparrow (a_i)_k = x + \overline{\prod_{i=0}^{k} a_i}$$

$$B_i = \overline{W_i} * B_i + W_i * CL \quad \text{for} \quad i \in \{0,...,n\} \tag{3}$$

As summarized in table 1, for any given *a* and *b* we have 16 combinatory logics $O_0$ through $O_{15}$. Each combination $O_i$ can be calculated using one or more pull-down/up functions.

The operations $O_0$ and $O_{15}$ can be used to write 0 or 1 to any CMC as an input data instead of any extra circuitry (as in the input IN inserted to the computational memory cell $CMC_0$ introduced above).

Table 1: Binary logic operations using pull-down and pull-up operations.

| a | 0 | 0 | 1 | 1 | Logical function | Instructions (load ‘a’ first) | cycles | space | Instructions (load ‘b’ first) | cycles | space |
|---|---|---|---|---|---|---|---|---|---|---|---|
| b | 0 | 1 | 0 | 1 | | | | | | | |
| $o_0$ | 0 | 0 | 0 | 0 | $a\,\bar{a}$ , $b\,\bar{b}$ | $x=a; x=x\downarrow a;$ | 1 | 1 | $x=b; x=x\downarrow b;$ | 1 | 1 |
| $o_1$ | 1 | 0 | 0 | 0 | $\bar{a}\,\bar{b}$ | $x=\bar{a}; x=x\downarrow b;$ | 2 | 1 | $x=\bar{b}; x=x\downarrow a;$ | 2 | 1 |
| $o_2$ | 0 | 1 | 0 | 0 | $\bar{a}\,b$ | $x=\bar{a}; y=\bar{b}; x=x\downarrow y;$ | 3 | 2 | $x=b; x=x\downarrow a;$ | 1 | 1 |
| $o_3$ | 1 | 1 | 0 | 0 | $\bar{a}$ | $x=\bar{a};$ | 1 | 1 | $x=\bar{a};$ | 1 | 1 |
| $o_4$ | 0 | 0 | 1 | 0 | $a\,\bar{b}$ | $x=a; x=x\downarrow b;$ | 1 | 1 | $x=\bar{b}; y=\bar{a}; x=x\downarrow y;$ | 3 | 2 |
| $o_5$ | 1 | 0 | 1 | 0 | $\bar{b}$ | $x=\bar{b};$ | 1 | 1 | $x=\bar{b};$ | 1 | 1 |
| $o_6$ | 0 | 1 | 1 | 0 | $\bar{a}\,b+a\,\bar{b}$ | $x=y=a; y=y\uparrow b; x=x\downarrow b; x=x\uparrow y;$ | 3 | 2 | $x=y=b; y=y\uparrow a; x=x\downarrow a; x=x\uparrow y;$ | 3 | 2 |
| $o_7$ | 1 | 1 | 1 | 0 | $\bar{a}+\bar{b}$ | $x=\bar{a}; x=x\uparrow b;$ | 2 | 1 | $x=\bar{b}; x=x\uparrow a;$ | 2 | 1 |
| $o_8$ | 0 | 0 | 0 | 1 | $a\,b$ | $x=a; y=\bar{b}; x=x\downarrow y;$ | 2 | 2 | $x=b; y=\bar{a}; x=x\downarrow y;$ | 2 | 2 |
| $o_9$ | 1 | 0 | 0 | 1 | $a\,b+\bar{a}\,\bar{b}$ | $x=y=a; y=y\downarrow b; x=x\uparrow b; x=x\downarrow y;$ | 3 | 2 | $x=y=b; y=y\downarrow a; x=x\uparrow a; x=x\downarrow y;$ | 3 | 2 |
| $o_{10}$ | 0 | 1 | 0 | 1 | $b$ | $x=b;$ | 0 | 1 | $x=b;$ | 0 | 1 |
| $o_{11}$ | 1 | 1 | 0 | 1 | $\bar{a}+b$ | $x=\bar{a}; y=\bar{b}; x=x\uparrow y;$ | 3 | 2 | $x=b; x=x\uparrow a;$ | 1 | 1 |
| $o_{12}$ | 0 | 0 | 1 | 1 | $a$ | $x=a;$ | 0 | 1 | $x=a;$ | 0 | 1 |
| $o_{13}$ | 1 | 0 | 1 | 1 | $a+\bar{b}$ | $x=a; x=x\uparrow b;$ | 1 | 1 | $x=\bar{b}; y=\bar{a}; x=x\uparrow y;$ | 3 | 2 |
| $o_{14}$ | 0 | 1 | 1 | 1 | $a+b$ | $x=a; y=\bar{b}; x=x\uparrow y;$ | 2 | 2 | $x=b; y=\bar{a}; x=x\uparrow y;$ | 2 | 2 |
| $o_{15}$ | 1 | 1 | 1 | 1 | $a+\bar{a}$ , $b+\bar{b}$ | $x=a; x=x\uparrow a;$ | 1 | 1 | $x=b; x=x\uparrow b;$ | 1 | 1 |

## 3.3 Case Study: Full Adder

The proposed case study, implements a Full adder in a compute line. This arithmetic operation is the most used in any computation unit today. For demonstration purpose, a Full adder is implemented in the proposed compute-line. It takes two input operands and one carry, and returns an output and a carry. It can be easily extended to support addition of integers.

A number of entries required to show all combination of three operands, as in the full adder, is 256 entries. Due to space limit we choose not to investigate on such a table. Instead, we implemented a scanner program that can give us the minimum number of compute-line read-write instructions required to achieve a given output. The result summary is in the Fig. 5. All possible combinations of three operands can be checked in the vertical bit-wise alignment between the following three hexadecimal values 0x55, 0x33 and 0x0F. Applying a bit-wise full adder of this three value we get 0x69 as an output and 0x17 as a remainder.

While scanning, in Fig. 5(a), the only possible first occurrence of 0x17 is at the fifth operation after the first 3 initial operations reserved to load 0x55, 0x33 and 0x0F in the given order. The first occurrence of 0x69 tokes two more operation in comparison to the first occurrence of 0x17. When we scanned for the first occurrence of 0x17 given that 0x69 is occurred Fig. 5(b) we saved one operation in comparison to scanning for 0x69 given 0x17 Fig. 5(c). The drawing in Fig. 5(a) is repeated in the Fig. 5(b) and (c) for comparison purpose only.

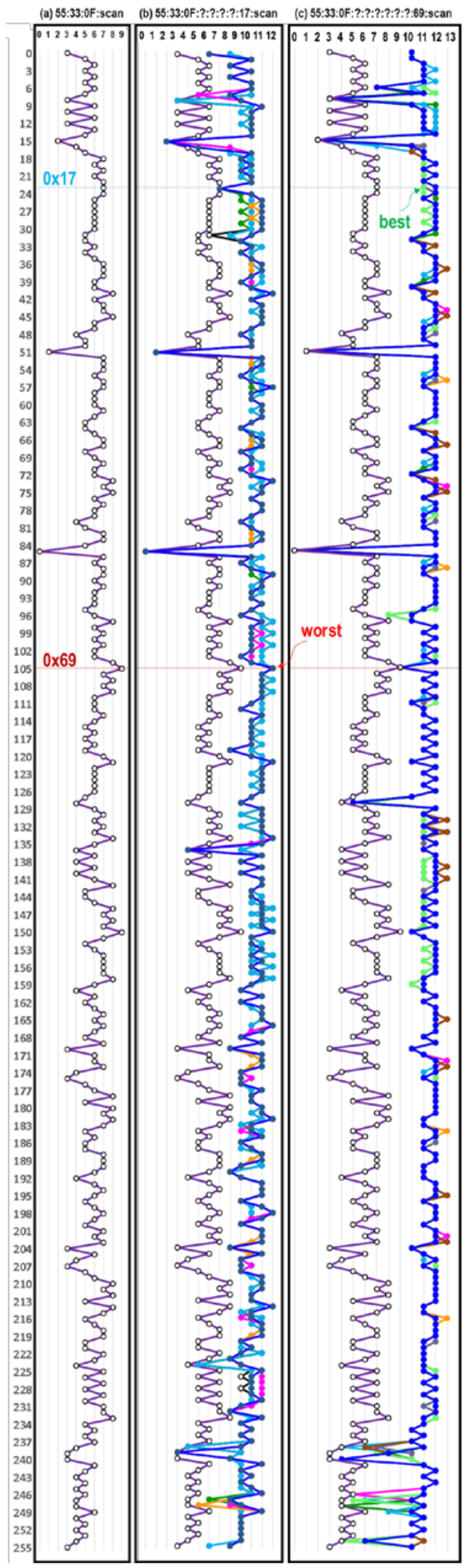

Fig. 5 pull-down/up scan based on fixed input data 55:33:0F.

The scanning revealed 20 sets of 8 operations with distinct paths (and 34 sets with repeated paths and less efficiency) that generate 0x17. For the result 0x69, the scanner detected 40 possible sets of 10 operations with distinct paths (28 sets with repeated paths and less efficient) where 4 cases registered an average minimum 5 of bit flips for read-lines and compute-line combined.
Given 40 sets of 0x69, scanning for 0x17 generated 348 distinct paths with repeated ones reached 2714. The list below shows 6 best candidate paths having an average bit flip count equal to 5:

(s1) 55 33 0F 08 FE F6 80 EE 60 69 7F 17
(s2) **55 33 0F 08 FE F6 80 EE E0 69 1F 17**
(s3) 55 33 0F 08 FE F6 80 EE E0 69 7F 17
(s4) **55 33 0F EF 80 90 FE 88 F8 69 01 17**
(s5) 55 33 0F EF 80 90 FE 88 F8 69 07 17
(s6) 55 33 0F EF 80 90 FE 88 F9 69 01 17

Two sets S2 and S4 are chosen to introduce the functions involved to get the resulting path as shown in table 2. The reason behind this choice is to show the power and the flexibility the compute-line it can provide. Function with 3 operands is used in the 5th line. Inverse function can be seen in the 11th line for S2. Line 10 shows a safe appearance of both pull-down and pull-up functions for the given sequence of functions from line 4 through 9. If this sequence of functions changes the line 10 might not be safe to compute. Different sequence of functions might generate different paths as it is the case for S2 and S4.

Table 2: set of functions involved to compute a full adder for S2 and S4.

| | S2 | S4 |
|---|---|---|
| 1 | $a = 0x55$ | $a = 0x55$ |
| 2 | $b = 0x33$ | $b = 0x33$ |
| 3 | $c = 0x0F$ | $c = 0x0F$ |
| 4 | $x_1 = c \downarrow (a,b) = 0x08$ | $x_1 = c \uparrow (a,b) = 0xEF$ |
| 5 | $x_2 = x_1 \uparrow (a,b,c) = 0xFE$ | $x_2 = x_1 \downarrow (a,b,c) = 0x80$ |
| 6 | $x_3 = x_2 \downarrow (x_1) = 0xF6$ | $x_3 = x_2 \uparrow (x_1) = 0x90$ |
| 7 | $x_4 = x_3 \downarrow (a,b) = 0x80$ | $x_4 = x_3 \uparrow (a,b) = 0xFE$ |
| 8 | $x_5 = x_4 \uparrow (a,b) = 0xEE$ | $x_5 = x_4 \downarrow (a,b) = 0x88$ |
| 9 | $x_6 = x_5 \downarrow (c) = 0xE0$ | $x_6 = x_5 \uparrow (c) = 0xF8$ |
| 10 | $v = x_6 \downarrow (x_4) \uparrow (x_3) = 0x69$ | $v = x_6 \downarrow (x_3) \uparrow (x_4) = 0x69$ |
| 11 | $x_7 = v \downarrow (x_6) \uparrow (x_6) = 0x1F$ | $x_7 = v \downarrow (x_4) = 0x01$ |
| 12 | $r = x_7 \downarrow (x_1, x_6) = 0x17$ | $r = x_7 \uparrow (x_1, x_6) = 0x17$ |

The temporary variables $x_1$ through $x_7$ are used to keep the intermediate values required to compute the sum and the remainder of a full adder. The number of these variables can be reduced by reusing the ones that are not referred any further and for this examples we can save up to two temporary locations without overwriting a, b and c.

## 3.4 Compute-Line instruction

For simplicity, we generalized all instructions into one formula Eq. 4 where *x*, *y* and *w* are alphabet letters and $(i)_{n_x}$, $(j)_{n_y}$ and $(k)_{n_w}$ are, respectively, series of index values to the involved computational memory cells. An empty series has it corresponding letter omitted from the instruction line and example codes in the following will illustrate clearly the compute-line instruction usage.

$$x \quad (i)_{n_x} \quad y \quad (j)_{n_y} \quad w \quad (k)_{n_w}; \tag{4}$$

Given an instruction, the controller will pull-down all computational memory cells with indices in the series (*i*), pull-up all computational memory cells with indices in the series (*j*), and write results into all computational memory cells with indices in the series (*k*). When only the letter *x* is present in the instruction, the compute-line can be discharged if any of the $CMC_i$ has a charged capacitor. Symmetrically, when only the letter *y* is present in the instruction, the compute-line can be charged if any of the capacitor of $CMC_j$ is in a discharged state. Alternatively, when only the letter *w* used for the instruction, a balance between capacitors in the selected $CMC_k$ will take place as well as the compute-line's capacitor due to the length of the wiring. For a period of time, as explained above, the compute-line can be adjusted to have a different capacitance if the write line of a set of CMCs is kept set to 1 during all the period. Therefore, the time to run an instruction depends mainly on the number of values in the series ($k_w$).

## 3.5 Results

If the three input operands (a,b, and c) are already loaded into $CMC_0$, $CMC_1$ and $CMC_2$, respectively, the instruction sets for the above two sets of operations S2 and S4 are introduced as in Table 3. For spice simulation, we used 45nm PTM model, as cited in [25], for high-performance application (PTM-HP), incorporating high-k metal gate and stress effect (level=54 & version=4.0). The associated Timing diagram of S2 is shown in Fig. 6.

Table 3: set of functions involved to compute a full adder for S2 and S4.

| | S2 | S4 |
|---|---|---|
| 4 | x 0 1 w 5; | y 0 1 w 5; |
| 5 | y 0 1 2 w 3; | x 0 1 2 w 3; |
| 6 | x 5 w 6; | y 5 w 6; |
| 7 | x 0 1 w 4; | y 0 1 w 4; |
| 8 | y 0 1 w 3; | x 0 1 w 3; |
| 9 | x 2 w 7; | y 2 w 7; |
| 10 | x 4 y 6 w 3; | x 6 y 4 w 3; |
| 11 | x 7 y 7 w 6; | x 4 w 6; |
| 12 | x 5 7 w 4; | y 5 7 w 4; |

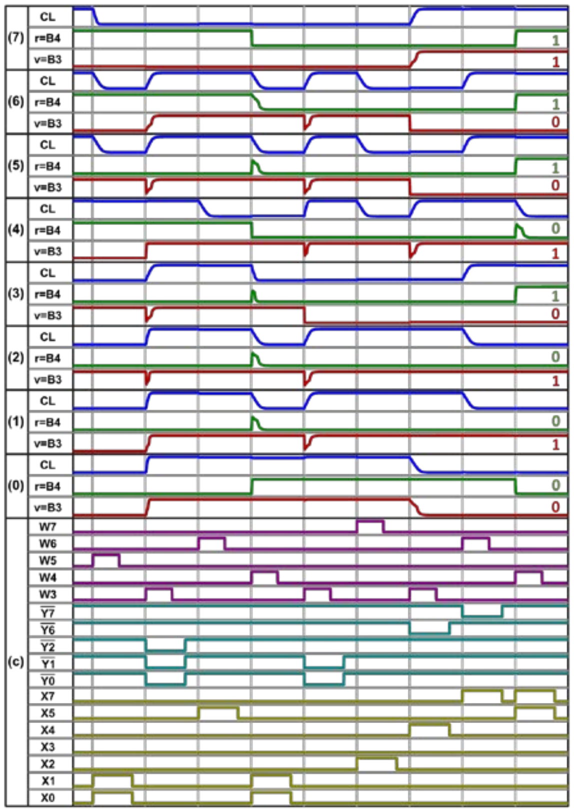

Fig. 6 Timing diagram of full adder in a dynamic compute-line.

The simulation gives correct results and confirms with the study and the analysis presented so far. The compute-line have promising potential to change the way program execution might be conducted.

## 4. Conclusion

A generalized compute-line based computational memory architecture that supports both common (static and dynamic) memory technologies was presented. In order to deeply investigate the capabilities of the compute-line, we choose the dynamic version due to its rich possibilities and it covers most of the issue the static one has. We showed the potential of using compute-line to compute any combinatory logic operation with different ways. This can unleash different real-time programming optimizations.
We showed that 50% of total compute-line activities can be saved during computation if the CMCs' stored data are taken in consideration while choosing the right set of the instructions to execute the code. Reducing the complexity of the instruction sets into a single and simple one helped a lot to study the behavior of the compute-line. Translating existing codes to the compute-line language and scaling the architecture to adopt complex topologies are scheduled for future work.

## References

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," SIGARCH Comput. Archit. News, vol. 23, no. 1, pp. 20–24, Mar. 1995. [Online]. Available: https://doi.org/10.1145/216585.216588
[2] P. A La Fratta and P. M Kogge, "Design enhancements for in-cache computations," Workshop on Chip Multiprocessor Memory Systems and Interconnects, 01 2009. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.577.4395{\&}\\rep=rep1{\&}type=pdf
[3] F. Duarte and S. Wong, "Cache-based memory copy hardware accelerator for multicore systems," IEEE Transactions on Computers, vol. 59, no. 11, pp. 1494–1507, Nov 2010. [Online]. Available: https://doi.org/10.1109/TC.2010.41
[4] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent ram," IEEE Micro, vol. 17, no. 2, pp. 34–44, Mar 1997. [Online]. Available: https://doi.org/10.1109/40.592312
[5] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization," in 2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), ser. MICRO-46. New York, NY, USA: ACM, Dec 2013, pp. 185–197. [Online]. Available: https://doi.org/10.1145/2540708.2540725
[6] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA). ACM, June 2015, pp. 336–348. [Online]. Available: https://doi.org/10.1145/2749469.2750385
[7] C. E. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick, "Scalable processors in the billion-transistor era: Iram," Computer, vol. 30, no. 9, pp. 75–78, Sep 1997. [Online]. Available: https://doi.org/10.1109/2.612252
[8] M. Oskin, F. T. Chong, and T. Sherwood, "Active pages: a computation model for intelligent memory," in Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235). Washington, DC, USA: IEEE Computer Society, Jun 1998, pp. 192–203. [Online]. Available: https://doi.org/10.1109/ISCA.1998.694774
[9] D. G. Elliott, W. M. Snelgrove, and M. Stumm, "Computational ram: A memory-simd hybrid and its application to dsp," in 1992 Proceedings of the IEEE Custom Integrated Circuits Conference, May 1992, pp. 30.6.1–30.6.4. [Online]. Available: https://doi.org/10.1109/CICC.1992.591879
[10] D. G. Elliott, M. Stumm, W. M. Snelgrove, C. Cojocaru, and R. Mckenzie, "Computational ram: implementing processors in memory," IEEE Design Test of Computers, vol. 16, no. 1, pp. 32–41, Jan 1999. [Online]. Available: https://doi.org/10.1109/54.748803
[11] J. C. Gealow and C. G. Sodini, "A pixel-parallel image processor using logic pitch-matched to dynamic memory," IEEE Journal of Solid-State Circuits, vol. 34, no. 6, pp. 831–

839, Jun 1999. [Online]. Available: https://doi.org/10.1109/4.766817

[12] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), Feb 2017, pp. 481–492. [Online]. Available: https://doi.org/10.1109/HPCA.2017.21

[13] S. Jeloka, N. B. Akesh, D. Sylvester, and D. Blaauw, "A 28 nm configurable memory (tcam/bcam/sram) using push-rule 6t bit cell enabling logic-in-memory," IEEE Journal of Solid-State Circuits, vol. 51, no. 4, pp. 1009–1021, April 2016. [Online]. Available: https://doi.org/10.1109/JSSC.2016.2515510

[14] M. Kang, E. P. Kim, M. s. Keel, and N. R. Shanbhag, "Energy-efficient and high throughput sparse distributed memory architecture," in 2015 IEEE International Symposium on Circuits and Systems (ISCAS), Lisbon, Portugal, May 2015, pp. 2505–2508. [Online]. Available: https://doi.org/10.1109/ISCAS.2015.7169194

[15] E. Yaakobi, A. A. Jiang and J. Bruck, "In-memory computing of akers logic array," in 2013 IEEE International Symposium on Information Theory, July 2013, pp. 2369–2373. [Online]. Available: https://doi.org/10.1109/isit.2013.6620650

[16] W. Khwa et al., "A 65nm 4kb algorithm dependent computing-in-memory sram unit-macro with 2.3ns and 55.8tops/w fully parallel product-sum operation for binary dnn edge processors," in 2018 IEEE International Solid - State Circuits Conference - (ISSCC), Feb 2018, pp. 496–498. [Online]. Available: https://doi.org/10.1109/ISSCC.2018.8310401

[17] A. Valero, S. Petit, J. Sahuquillo, P. LÃ¸pez, and J. Duato, "Design, performance, and energy consumption of edram/sram macrocells for l1 data caches," IEEE Transactions on Computers, vol. 61, no. 9, pp. 1231–1242, Sept 2012. [Online]. Available: https://doi.org/10.1109/TC.2011.138

[18] N. Verma and A. P. Chandrakasan, "A 256 kb 65 nm 8t subthreshold sram employing sense-amplifier redundancy," IEEE Journal of Solid- State Circuits, vol. 43, no. 1, pp. 141–149, Jan 2008. [Online]. Available: https://doi.org/10.1109/JSSC.2007.908005

[19] H. Noguchi, Y. Iguchi, H. Fujiwara, Y. Morita, K. Nii, H. Kawaguchi, and M. Yoshimoto, "A 10t non-precharge two-port sram for 74video processing," in IEEE Computer Society Annual Symposium on VLSI (ISVLSI '07), March 2007, pp. 107–112. [Online]. Available: https://doi.org/10.1109/ISVLSI.2007.2

[20] A. Putnam et al., "A reconfigurable fabric for accelerating large-scale datacenter services," in 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), June 2014, pp. 13–24. [Online]. Available: https://doi.org/10.1109/ISCA.2014.6853195

[21] J. Viktorin, P. Korcek, T. Fukac, and J. Korenek, "Network monitoring probe based on xilinx zynq," in 2014 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), ser. ANCS '14. New York, NY, USA: ACM, Oct 2014, pp. 237–238. [Online]. Available: https://doi.org/10.1145/2658260.2661769

[22] M. A. Zidan and W. D. Lu, "Rram fabric for neuromorphic and reconfigurable compute-in-memory systems," in 2018 IEEE Custom Integrated Circuits Conference (CICC), April 2018, pp. 1–8.

[23] P. Kim, Jaeyoung; Mazumder, "A robust 12t sram cell with improved write margin for ultra-low power applications in 40nm cmos," Integration, the VLSI Journal, vol. 57, 03 2017. [Online]. Available: https://doi.org/10.1016/j.vlsi.2016.09.008

[24] Driss Azougagh, Ahmed Rebbani, and Omar Bouattane, "Computational Memory Architecture Supporting in Bit-Line Processing", IJCSNS International Journal of Computer Science and Network Security, VOL.18 No.7, July 2018

[25] "Predictive technology model." [Online]. Available: http://ptm.asu.edu/

**Driss Azougagh** received his B.S. degree in Computer Science in 1995 from Mohamed ben Abdellah University, Fes, Morocco. He received his Master degree in Computer Science in 2002 from Korea Advanced Institute of Science and Technology, Deajeon, Korea. He is a Ph.D. student at the University Hassan II Mohammedia, ENSET Institute. His research is focused on computer architecture. His research interests include (Massively Distributed and Parallel) Computer Architecture and Processing.

**Ahmed Rebbani** received the B.S. degree in Electronics in 1988 the M.S. degree in Applied Electronics in 1992 from the ENSET Institute, Mohammedia, Morocco. He received the DEA diploma in information processing in 1997 from the Faculty of Sciences Ben Msik, Casablanca, Morocco. He is now teacher and researcher at the Hassan II University of Casablanca, ENSET Institute Mohammedia. His research is focused on Internet of things and renewable energy.

**Omar Bouattane** has his Ph.D. degree in 2001 in Parallel Image Processing on Reconfigurable Computing Mesh from the Faculty of Science Ain Chock, CASABLANCA, Morocco. He has published more than 30 research publications and brevets in various National, International conference proceedings and Journals. His research interests include Massively Parallel Architectures, cluster analysis, pattern recognition, image processing and fuzzy logic.