ISSN 2278-3091
Volume 9, No.1.5, 2020
International Journal of Advanced Trends in Computer Science and Engineering
Available Online at http://www.warse.org/IJATCSE/static/pdf/file/ijatcse0991.52020.pdf
https://doi.org/10.30534/ijatcse/2020/0991.52020

# A Compute-line based Computational Memory Architecture using bit-line Keepers: Internals and Analysis

**Driss Azougagh[1], Ahmed rebbani[2], Omar Bouattane[3]**
[1]SSDIA Laboratory, ENSET, Hassan II University of Casablanca, Morocco, azougagh@gmail.com
[2]SSDIA Laboratory, ENSET, Hassan II University of Casablanca, Morocco, a.rebbani@gmail.com
[3]SSDIA Laboratory, ENSET, Hassan II University of Casablanca, Morocco, o.bouattane@gmail.com

## ABSTRACT

The growth of Big Data, memory wall and power wall are posing unprecedented demand for Processing In Memory (PIM). A computational memory architecture supporting in bit-Line processing can be a major key for PIM to eliminate the overhead of moving data from processing unit to memory and vice versa. It promises high bandwidth, massive parallelism, and high energy efficiency. The existing PIM approaches concentrates mostly on near-memory processing (NMP) and/or in-memory processing (IMP). The Compute-line based Computational Memory Architecture (CCMA), or simply (multiple) compute-lines (CLs), represents a different way of approaching in-memory processing (IMP). A compute-line represents a line that carries fine-grained operations using connected memory cells. CL is based on (a selection of) a bit-line for processing elementary logical operations and a bit-line Keeper (KEEPER) for enforcing and stabilizing the outcome results. CCMA is backward compatible with the conventional Static Random Access Memory (SRAM) and can be used for state storing. In contrary to the conventional SRAM, it eliminates the need to pre-charge and sensing bit-line(s) for read and write operations which reduces bit-line activities and support in-place combinational logic which reduces data transfer latency. It introduces a considerable potential to reduce bandwidth and energy consumption by eliminating overhead of data movement when used as an in-memory computing. Moreover, it can easily support any specific interconnect topology between multiple compute-lines for parallel applications by hard wiring their input/output interfaces during chip fabrication. The CCMA designs the KEEPER circuitry so that, in one (or two) clock cycle(s) and through bit-line selection, its can multi-row read bit information from participating memory cells, bitwise logic compute selected operation and multi-row write to targeted memory cells. In this work, toward deep investigation of the CCMA architectures and perspective remarks, CL's capabilities and statistical analysis of running in-place logic operations are presented and showed potential computational and global energy savings.

**Key words :** Compute-line, in-place processing, build-in computing.

## 1. INTRODUCTION

The Computational Memory Architecture (CMA) in [1]–[4] is based on a Compute-Line (CL) with built-in computing capabilities that shares many objectives with Near Data Processing (NDP) including Computing with Memory [5],[6] as in FPGAs, Computing in caches [7] using Computational RAM (CRAM) [8],[9], Near-Memory Processing [10],[11] and Processing In Memory (PIM) [11],[12]. In contrary, CL does not require dedicated computational, pre-charging and/or sensing circuitry logics to conduct a logical operation on locally stored data without any memory copy. Instead, it uses at request [1] or automatically [2]–[4] a bit-line keeper (KEEPER) to keep both bit-lines stable and superposed to each other for each execution of an elementary operation (NAND/NOR/NOT) on the selected data stored locally.

A binary operation (Addition, Multiplication, etc.) can be optimized spatially at the CPU level [13] or temporally at software level [14]. CL architecture follows the temporal computing model to evaluate general functions across multiple cycles and follows the spatial model to execute elementary operations (NAND/NOR/NOT) on large set of data. It supports in bit-line processing bringing parallel computation as close as possible to the location of the stored data. It first selects and sets up an operation, reads bit information from external inputs and/or local memory cells in parallel and simultaneously, stabilizes the bit-lines throughout the controlled [1] or automated [2]–[4] KEEPER, and finally writes the data to external outputs and/or local memory cells in one compute cycle. Moreover, multiple CLs can be interconnected using dedicated topologies with multiple dimensions to accelerate computation and increase locality for specific research areas.

In this work, we introduce internals of CCMA using bit-line keepers and the corresponding analysis of CL. In the next section, we introduce basics summary of CCMA architecture where the concepts of CL and bit-line Keepers are introduced. In the third section, we presented the results of simulating a Full Adder in the CL. It follows an analysis section where CL behaviors are investigated. Finally, we provide a conclusion including some perspective remarks.

## 2. CCMA BASICS

CCMA, as in our previous work [1]–[4], is a pure PIM (or computational memory) architecture that can be configured to perform both state storage and combinational logic. The bit of information can be stored from extern buffer similarly to conventional SRAM or from intern result of applied combinational logic. The in-place computing (or combinational logic) is configurable by control lines (from a sequencer) and the connection between the participating state storages and their corresponding logics on the compute-line.

CCMA has a control unit and a set of *m* selective compute-lines to store data and/or apply logic operations on the stored data. The control unit has extra control column lines (XSL and YSL) used to synchronize participating inputs and select target bit-line for each in-place processing operation. Figure 1 presents an example of selective computational memory with *m* selective compute-lines or simply, now on, we call compute-lines (CLs). For *k* varying from 1 to *m*, each $CL_k$ is composed of a pair of bit-lines $XBL_k$ and $YBL_k$ accompanied with a pair of select-lines $XSL_k$ and $YSL_k$ and contains a set of blocks distributed all along the line.

The CCMA will act as a bit-wise Single Instruction Multiple Data (bit-wise SIMD) if the same command word lines from the control unit are shared by horizontal blocks of the same kind (INPUTs, OUTPUTs, CMCs or KEEPERs). For simplicity and to further save circuitry overhead, one global pair of select-lines XSL and YSL is sufficient and can be shared in an interleaved manner between each two adjacent compute-lines by replacing all $XSL_k$ with XSL and $YSL_k$ with YSL. For correct and functional CCMA, due to dimensioning limitations, the number of CLs (*m*) and the number of blocks per CL (*n*) need to be carefully chosen in order to guarantee stable multiple read/writes from/to CMCs to/from candidate bit-line(s), simultaneously and efficiently.

For advanced dynamic data-parallelism, specific topology that interconnect any subset of blocks (mainly horizontally) for data-exchange is easily supported by cross wiring their Local INPUTs (LIs) to their Local OUTPUTs (LOs) internally as in Figure 1(b). In addition, each compute-line ($CL_k$) can use proper private select-lines. The presence of the inter-connections between compute-lines can give raise of massive parallel computation on multiple local data, depending on which topology is chosen.

In the following, we concentrated more on studying the basics of KEEPER block and its relation with other blocks, select-lines and bit-lines. In particular, a detailed study and analysis of the functionality and behavior of minimal CL (M-CL) are introduced. These basics are the ones exploited to design the full version of the compute-line supporting built-in computational capabilities in [1] from which other derivations with different optimization objectives and purposes emerged in [2]–[4]. Extending the finding in this work to [2] is straight forward and to [3],[4] requires some study left for future work.

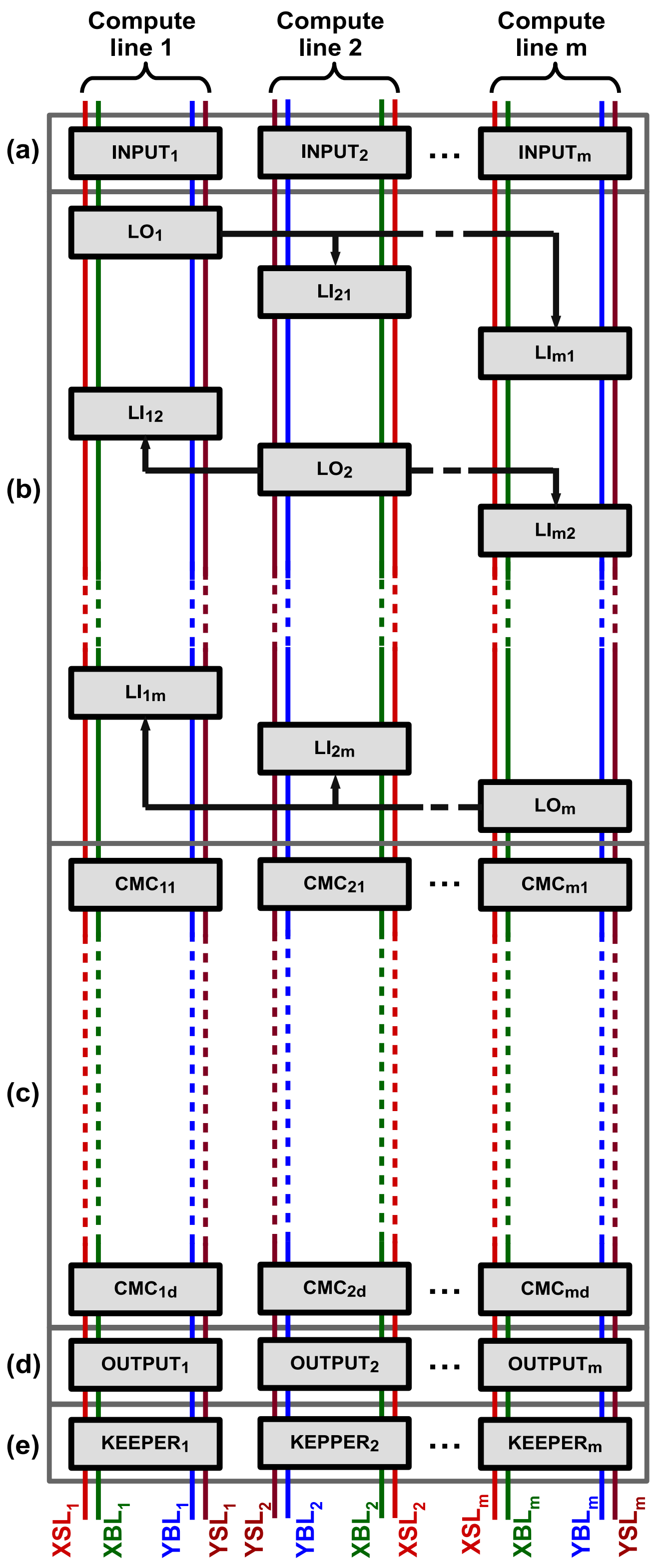

**Figure 1:** CCMA is composed of *m* compute-lines (CLs), each CL has (a) 1×INPUT, (b) *m*×Local Inputs (LIs) and 1×Local Output

## 2.1 Concept of CL

The minimal compute-line architecture (M-CL) in Figure 2 is a subpart of the full symmetrical CL (F-CL) mentioned in [1]–[4]. M-CL supports only NOR/NOT logic operations on the bit-line XBL using only one operation select-line (XSL). Its basic blocks are the minimal INPUT (M-INPUT), OUTPUT (M-OUTPUT) and bit-line KEEPER (M-KEEPER) as in Figure 1(a), (c) and (d), respectively. The minimal computational memory cell (M-CMC) in Figure 1(b) is simply a coupling combination of both M-INPUT and M-OUTPUT where their XI and XO are connected to each other and can act as either M-INPUT, M-OUTPUT or both simultaneously. It is the smallest version of the full CMC with 55% area overhead improvement and requires about three extra nMOS transistors (3T) over the conventional SRAM memory cell with 6T.

The M-INPUT has one nMOS transistor (XN) and one AND logic gate (XG). XN and XG are arranged to pull down the bit-line XBL only when the input XI, the select-line XSL and the read word line XR are all set high. Otherwise, if one of the inputs of the gate XG is 0, the M-INPUT stays passive without interfering with the state of XBL. In the other hand, M-OUTPUT has two nMOS pass transistors (XT and YT) and storage nodes (XB and YB). When the write word line (XW) is activated, the XT and YT play a roll of passing the states in XBL and YBL to XB and YB, respectively. Once the states are stored into the storage nodes, the output (XO) exhibit a steady persistent state for any read by any other internal/external M-INPUTs and/or devices, until XW is activated again and the state of XBL opposite to that of XO.

The KEEPER, select-line(s) and bit-lines play major role in orchestrating in-place operation computing when the operands are locality available. Unlike [15], the CL does not required a pre-charging cycle, sensing amplifiers nor extra separated read-line for read/write operation, instead it exploits bit-lines for read/write and uses extra select-line(s) for operation control. The minimal bit-line KEEPER (M-KEEPER), used in M-CL, is a reduced version of KEEPER with about 30% area overhead improvement. M-KEEPER's role is to keep the bit-line YBL opposed to the bit-line XBL using the controlled inverter (YP and YN) through XG and the command line BK.

When XSL is activated, M-CL senses for any participating M-CMC (M-INPUT) pulling down the bit-line XBL to 0 while the M-KEEPER keeps raising non pulled down XBL to 1 and projecting the opposite state of XBL to YBL, as formulated in the two expressions in (1). Once both bit-lines are stabilized, targeted M-CMCs (M-OUTPUTs) save the resulting state to their storage nodes using the expression in (2).

$$XBL = XSL \times \overline{\sum_{j=1}^{d} XR_j \times XB_j} \quad and \quad YBL = \overline{XBL} \tag{1}$$

$$XB_i = XW_i \times XBL \tag{2}$$

The data transfer medium used between bit-lines and storage nodes has a pair of transistors (XT and YT) for writing and a gate XG and a transistor XN for reading. It has a small latency.

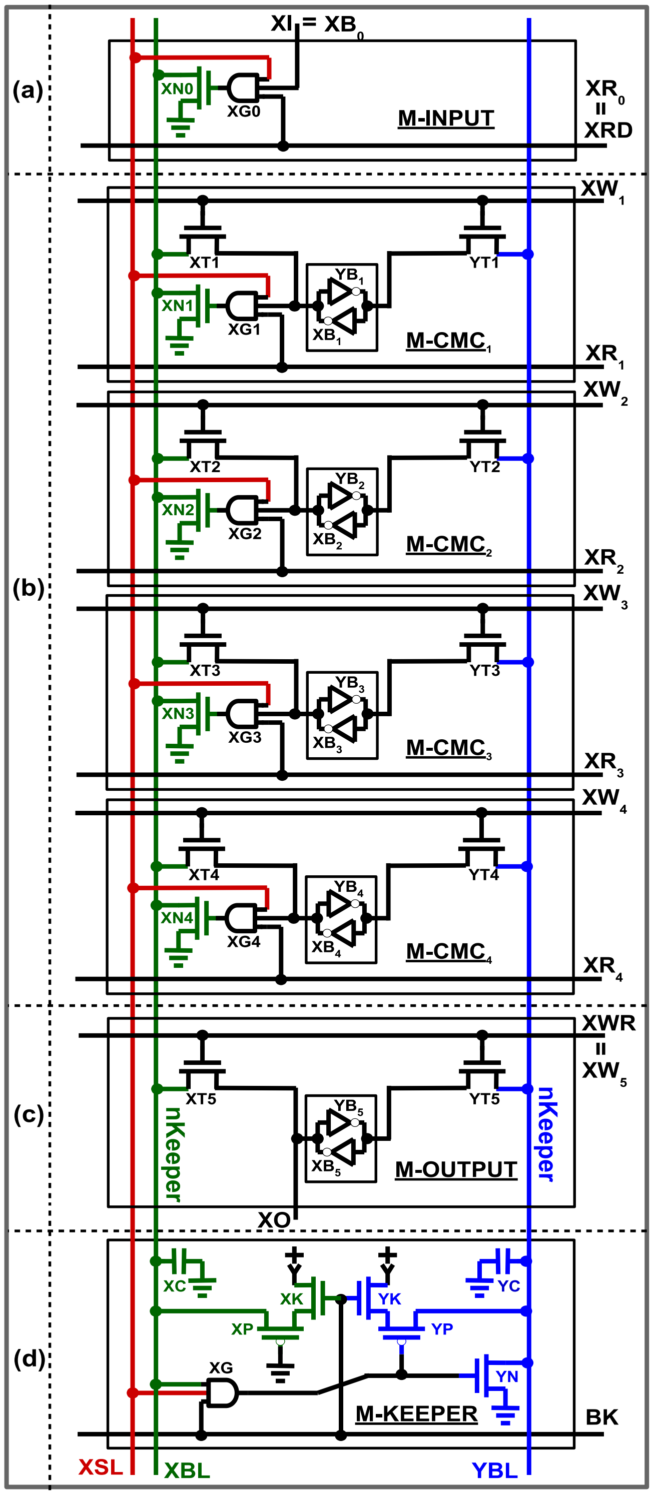

**Figure 2:** Minimal Compute-line (M-CL) with (a) 1×M-INPUT, (b) 4×M-CMC, (c) 1×M-OUTPUT and (d) 1×M-KEEPER
(

## 2.2 Concept of Bit-line Keeper in CL

Prior detailed study of CL, we first introduce two symmetric flavors of bit-line keepers from where the KEEPER architecture is inspired. We present pKeeper and nKeeper circuits as in Figure 3(a) and (b), respectively. It is easy to grasp the symmetry between nKeeper and pKeeper. Hence, in this paper, nKeeper is used as a major model to construct KEEPER as depicted in the M-CL above with the drawings colored with green or blue. The nKeeper has a capacitor $C_n$ connected to an nBit-line *n* and a ground (GND or 0). It uses an nMOS transistor T and pMOS transistor P in serial for passing a degraded or weak 1 to nBit-line *n* and charge the capacitor $C_n$ when commands on *c* is 0 and on *w* is 1. It can support *z* parallel nMOS transistors where each pass transistor $N_j$ is used for passing a strong 0 to the nBit-line *n* and discharge the capacitor $C_n$ when $x_j$ is 1 for some $j \in \{1, .., z\}$.

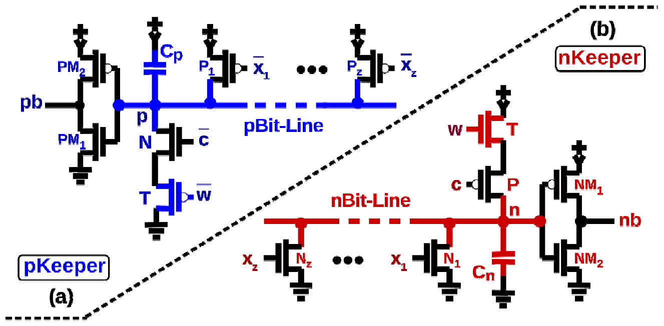

**Figure 3:** Bit-line conception for compute-line using (a) pKeeper with pMOS Transistors to push up pBit-line and (b) nKeeper with nMOS transistors to pull down the nBit-line.

The nKeeper might experience contention when both pull-up and pull-down are simultaneously turned on. However, since pull-down pass a strong 0 and pull-up passes a weak 1 to the nBit-line *n*, the strong 0 dominates and discharges the capacitor $C_n$. Hence, the major role of using the transistor T is to reduce the degree of contention since it passes a degraded 1 to the nBit-line *n*. In addition, the transistors T and P are carefully resized in order to regulate the capacitor's charging speed and voltage level by reducing the gate length of T in comparison to that of the rest of transistors in the circuit. An inverter composed of transistors NM1 and NM2 is used to invert the voltage level of the nBit-line. When the gate $x_i$ of the pass Transistor $N_i$ is 1 the nBit-line is pulled down strongly to 0. Thus, a bit-wise logical NOR, NAND and NOT operations can be performed in nKeeper, pKeeper, and both, respectively as summarized in the two expressions in (3). Note that, $n_{new}$ is kind of a reduced form of the expression (1) above.

$$n_{new} = (n_{old} + \overline{w}c)\overline{\sum_{j=1}^{z} x_j} \quad \& \quad p_{new} = p_{old}\,\overline{w\overline{c}} + \overline{\prod_{j=1}^{z} x_j} \qquad (3)$$

Figure 4 shows nKeeper and pKeeper voltage and current waveforms using spice simulation with 45nm PTM model, as cited in [16], for high-performance application (PTM-HP), incorporating high-k metal gate and stress effect(level = 54 and version = 4.0) and *z* is equal to 4. Different transistor models have been tested and they showed similar results with slight but clear noticeable differences. Similar observation was witnessed when the capacitor is changed as well. The voltage level of bit-lines (*n* and *p*) with their inverses (*nb* and *pb*) and the current of their corresponding pass transistors ($N_i$ and $P_i$) are in Figures 4(b) and (a), and 4(d) and (c), respectively. Figure 4(e) presents the control combinations

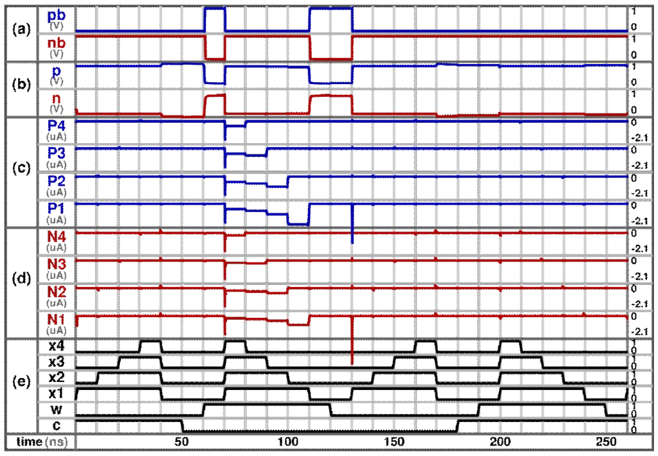

**Figure 4:** The voltage waveforms of nKeeper and pKeeper using spice simulation with 45nm PTM model for (a) voltage inverse of outcome result (b) voltage registered on n Bit-line and pBit-line (c) current traversed pMOS transistors for Pl, P2, P3, and P4 (d) current traversed pMOS transistors Nl, N2, N3 , and N4, and (e) logic control commands applied on c, w, and xz for z ∈{1, .., 4}.

applied to the circuit in gates *c*, *w*, and $x_z$ for all *z*'s. The pattern in time for the control combinations is chosen to show the impact on voltage and current of the circuit as the number of transistors pulling-down the bit-line varies and/or increases.

Vertically, the waveforms are divided into 26 cycles each with a duration of 10ns. In case of nKeeper, the cycles 7 and 12 show the charging of the capacitor due to weak pull-up and in the absence of strong pull-down. The cycles 5, 6, 13, 18, 19, 20, 25, and 26 illustrate how the capacitor's voltage level is preserved and the nBit-line is kept unchanged, in which the nKeeper is said to be in passive state. The remaining cycles presents the nBit-line being pulled-down by at least one of the transistors $N_j$ for $j \in \{1, .., z\}$. Pulling down the nBit-line *n* cause the capacitor $C_n$ to discharge and dominates over the weak 1 passed by T and P to the nBit-line.

The particular noticeable discharging bursts in the waveforms are caused when transiting from phase 7 to 8 and from phase 13 to 14. The first transition causes the capacitor to discharge along 4 transistors and each transistor did registered relatively equal amount of current traversed from drain to source. In the second transition there was only one active transistor N1 and the totality of current discharged did traversed through the transistor and caused a peak pulse current of -5µA. As the number of transistors pulling the nBit-line down increases the line's voltage level becomes more stable and converges faster.

## 3. RESULTS

M-CL can run addition of two *n*-bits binary numbers ( $(a)_n$ and $(b)_n$ ) using only NOT and NOR logic operations as shown in Figure 5. An optional 1-bit carry $c_0$ is initially fetched to the storage nodes XB1. The operands $a_0$ and $b_0$ are fetched to XB2 and XB3, respectively. When $c_0$, $a_k$ and/or $b_k$ are available locally, the iteration becomes shorter, the cycles cyc-c, cyc-a and/or cyc-b are omitted and their storage locations are directly accessed instead of using XB1, XB2 and/or XB3, respectively. Such iteration requires 9 cycles to run Full Adder with sum $s_k$ and carry $c_{k+1}$ stored in XO and XB1, respectively.

Figure 6, adopted from our previous work [1], shows voltage waveforms of running Full Adder (cyc1 through cyc9) on different inputs in M-CL. Common control commands for read, process, and write are shown in Figure 6(c). For a given iteration *k* and a combination *i* ="$c_k|a_k|b_k$" ∈ {0, .., 7}, the Figure 6(*i*) represents the voltage evolution in time of the CL running Full Adder of two bits $a_k$ and $b_k$ with the carry $c_k$.(left side 0/1 digits) with sum $s_k$ and carry $c_{k+1}$ (right side 0/1 digits). Cycles 2, 3, 4, 6 and 7 experience recursive bit-wise NOR operation, where one of the involved operands is also used for storing the outcome result and causing a slight impact on the voltage level of the bit-line XBL before storing the outcome results. However, final result consistency is guaranteed due to the stability enforced by the KEEPER on the bit-lines.

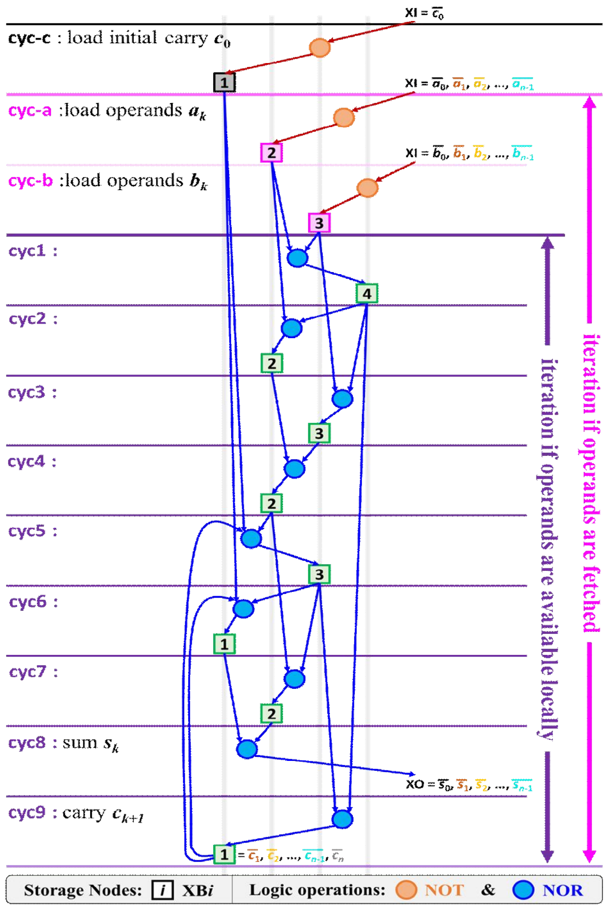
XI = $\overline{c_0}$
cyc-c : load initial carry $c_0$
XI = $\overline{a_0}, \overline{a_1}, \overline{a_2}, ..., \overline{a_{n-1}}$
cyc-a :load operands $a_k$
XI = $\overline{b_0}, \overline{b_1}, \overline{b_2}, ..., \overline{b_{n-1}}$
cyc-b :load operands $b_k$
cyc1 :
cyc2 :
cyc3 :
cyc4 :
cyc5 :
cyc6 :
cyc7 :
cyc8 : sum $s_k$
XO = $\overline{s_0}, \overline{s_1}, \overline{s_2}, ..., \overline{s_{n-1}}$
cyc9 : carry $c_{k+1}$
$= \overline{c_1}, \overline{c_2}, ..., \overline{c_{n-1}}, \overline{c_n}$
iteration if operands are available locally
iteration if operands are fetched
Storage Nodes: XB*i*
Logic operations: NOT & NOR

**Figure 5:** Sequence of operations to iteratively compute n-bits binary addition using Full Adder in M-CL.

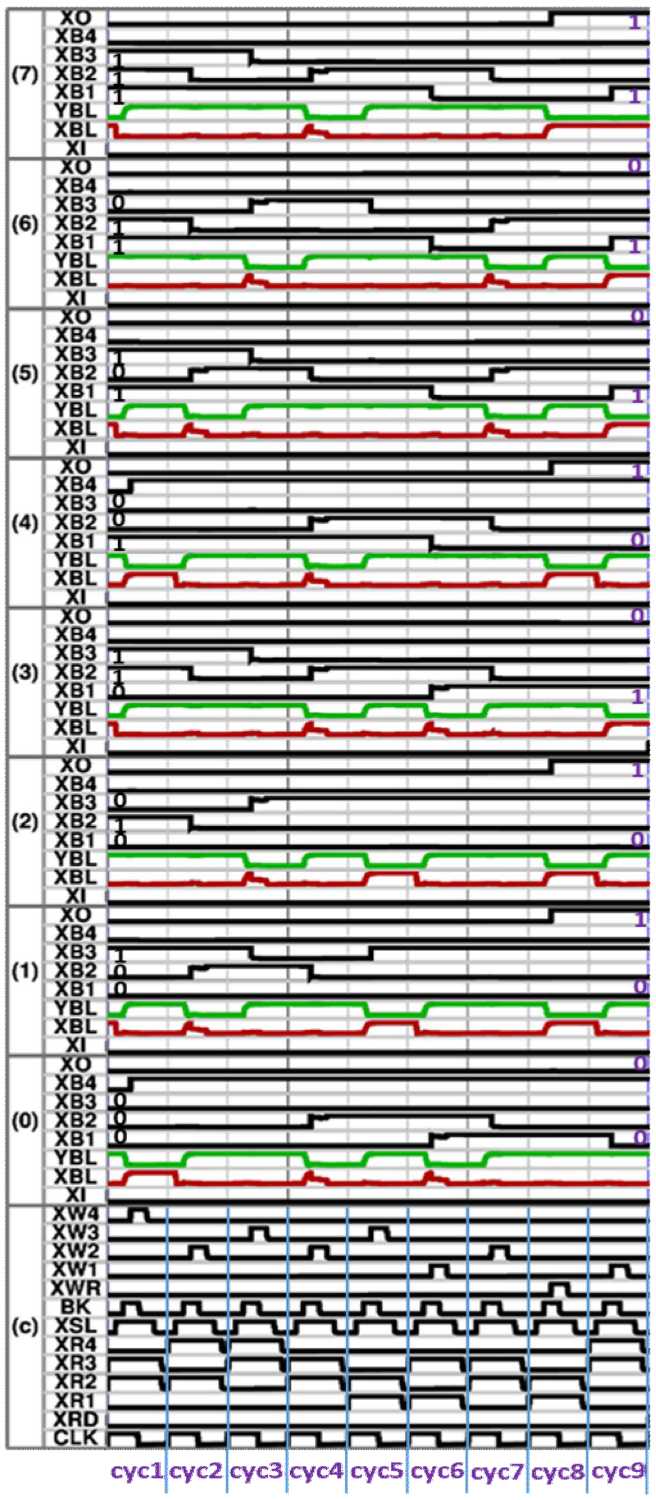

**Figure 6:** Voltage waveforms for simulated Full Adder; (c) control commands & (*i*) XI, XBL, YBL, XB1,XB2, XB3, XB4, XO for a carry $c_k$ and inputs $a_k$ and $b_k$, where *i* = " $c_k|a_k|b_k$" ∈{1, ., 7}.

## 4. ANALYSIS

In voltage waveforms above, the bit-lines did not experienced many charging and discharging of their capacitors during the execution of memory operations (compute operations) nor during the execution of regular fetch/store (read/write) operations. The bit-line XBL, however, experienced in one single cycle a charge and discharge when the same CMC is used for read and write simultaneously. This can happen when the operation is reflexive, where the read and write are applied to the same CMC. This can be avoided by adding extra CMCs, changing the operation sequences of writing the conflicting outcome result elsewhere and copying the outcome bit of information back to the final designated CMC.

For deep analysis, Figure 7 shows all kind of diagrams that can be experienced by (storage nodes of) a written CMC in a compute-line running a logical operation when using M-CL architecture, or simply say all possible/distinct compute cycles. Each row-column $R_iC_j$ shows distinct voltage waveforms of XSL, BK, XW, XBL, YBL and XB for a distinct computation cycle vertically divided into eight phases (ΔT1 through ΔT8) where two phases (ΔT0 and ΔT9) of read word line (XR=1) are omitted due to the space limitation.

The count gain of pulling up or down the bit-lines XBL and YBL is obtained by comparing each bit-line to conventional bit-line using bit-line pre-charging for each possible compute cycle $R_iC_j$, and can be simply summarized into Table 1. In the table, the signs "↑" and "↓" are used with the meaning of pull-up and pull-down, respectively. Preserving and exploiting the previous states of bit-lines XBL and YBL to conduct the current operation does reduce considerably the activities of both bit-lines. For compute cycles in row R7, in the conventional memory, the bit-lines XBL and YBL are pre-charged and since the final result is 1 the XBL will keep its state and YBL will be discharged. Whereas in the proposed CCMA, XBL experiences discharging as well for the three columns C1, C2 and C3 and this is noted by the −1 in the table. However, YBL did not experience any charging and has its pulling-up count gain is 1 for all columns C1, C2 and C3.

Columns ("$C_j$") are classified by the initial values of the bit-lines XBL and YBL before starting computation. Initially, Column C1 has XBL is 0 and YBL is 1, C2 has both XBL and YBL are 0, and C3 has XBL set to 1 and YBL set to 0. The (distinct) compute cycles experienced by a CMC can be distinguished by checking whether its storage nodes were participating in memory operation as an operand or not. In a given compute cycle, we define a memory operation to be directive when each involved CMC is used for either read or write but not for both, as it is the case for the rows R1 through R4. Alternatively, when the read word line (XR) and write word line (XW) of the same CMC are activated, we define the memory operation to be reflexive, as shown in R5 through R7. If the input bit of information read from the CMC is opposite to the resulting outcome to be written to the same CMC, the reflexive operation is called conflictive as in R7, otherwise it is called concordant as shown in R5 and R6. The rows can also be distinguished by the state change experienced by XBL in the given compute cycle. The bit-line is pulled-down in R1, R2, R5, and R6 and passive in R3, R4 and R7.

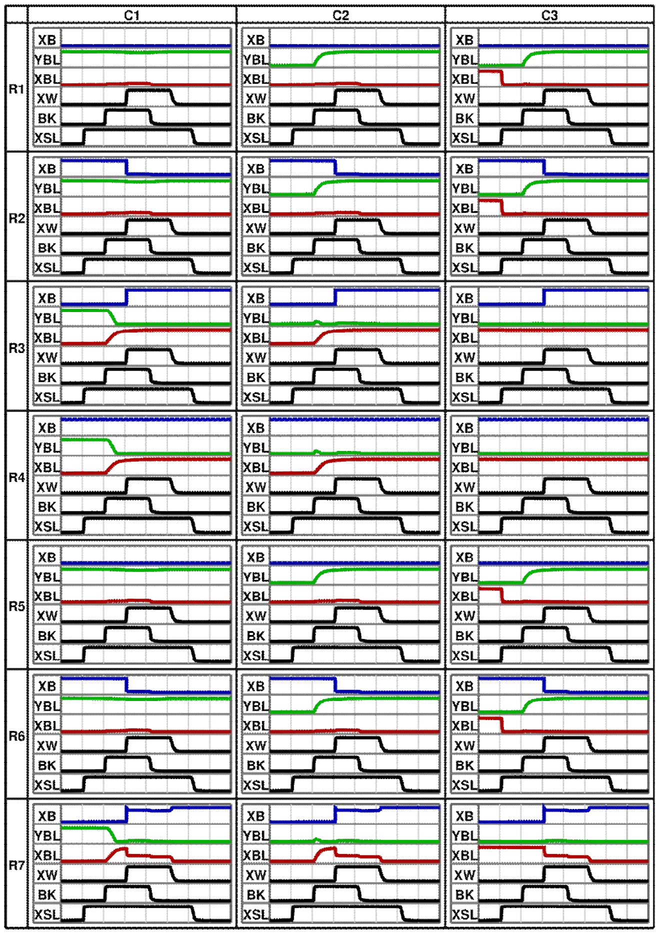

**Figure 7:** Voltage waveforms of all possible/distinct compute cycles $R_iC_j$ , where $i \in \{1, ., 7\}$ and $j \in \{1,2,3\}$, extracted from simulations conducted in the M-CL architecture.

**Table 1:** Number of transitions gained per cycle of bit-lines XBL and YBL over to the conventional bit-line BL and BL, respectively.

| | | R1 | R2 | R3 | R4 | R5 | R6 | R7 |
|---|---|---|---|---|---|---|---|---|
| XBL↑ | C1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| | C2 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| | C3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| XBL↓ | C1 | 1 | 1 | 0 | 0 | 1 | 1 | -1 |
| | C2 | 1 | 1 | 0 | 0 | 1 | 1 | -1 |
| | C3 | 0 | 0 | 0 | 0 | 0 | 0 | -1 |
| YBL↑ | C1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | C2 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| | C3 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| YBL↓ | C1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | C2 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| | C3 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |

The initial value of the target storage node XB (0 or 1) also classifies the rows. XB is 1 for R2, R4 and R6 and 0 for the rest. After an operation is applied, the select-line XSL directs the participating CMCs (INPUTs) with values 1 to pull down the bit-line XBL. The only possible pull down that changed the content of bit-line XBL are registered in rows R1, R2, R5, and R6 for column C3. The KEEPER introduces a sharp distinction between both bit-lines when BK is set to 1. As it can be seen from rows R3, R4, and R7 for columns C1 or C2, after select-line XSL is activated, KEEPER charges the capacitor XC if the bit-line XBL is not pulled-down by any of the participating CMCs (INPUTs). Then, it enforces the distinction by projecting the inverted value of XBL to YBL.

In row R7, we can witness a side effect of reflective operations on the bit-line XBL as the CMCs are selected for both read and write at the same time. At write operation, XW is set to 1, the bit-line YBL converges faster to strong 0 while XBL is charging. Writing bit-lines values back to one of the CMCs while participating as an operand causes conflict. This conflict occurs when the storage node XB is rising to 1 and the corresponding pass transistor $XN_j$ (for some $j \in \{1, .., d\}$) starts pulling-down the bit-line XBL that was charging. As long as YB is strongly enough pulled down to 0, XB reaches a voltage level of stability sufficiently enough to converge to the correct outcome result at the end of the operation.

Each two successive operations might differ in the sources INPUTs (CMCs) as well as the destinations OUTPUTs (CMCs). However, the last state of bit-lines left by the first operation should be the same as the initial state of the bit-lines used by the second operation. Therefore, the cyclic directed graph that connect all possible/distinct compute cycles can be summarized by seven inference rules; R1 → C1, R2 → C1, R5 → C1, R6 → C1, R3 → C3, R4 → C3, and R7 → C2. The rule "$R_i \rightarrow C_v$" means that any compute cycle ($R_iC^*$) in the row $R_i$ can be followed by any compute cycle ($R^*C_v$) in the column $C_v$. We define $P(R_uC_v|R_iC_j)$ to be the probability distribution for experiencing the compute cycle in $R_uC_v$ right after completing the compute cycle in $R_iC_j$. The $P(R_uC_v|R_iC_j)$ is 0 when $R_i \rightarrow C_v$ is not among the seven rules above.

In this setup, our comparison will focus on activities of the bit-lines instead of the results stored by CMCs. For CL with *m* CMCs, and logical operations having *i* operands and *o* target outputs in one compute cycle, the occurrence distribution is summarized in Table 2. Note that, as the number of operands *i* raises the chance of pulling down XBL increases. And, when the number of CMCs *m* is larger than the number of operands *i* and target outputs *o*, the chance of experiencing a reflexive operation becomes small. For M-CL with 4 CMCs conducting operation with 2 operands and one output, the chance of "XB = 1" is half, a passive XBL is one fourth, and reflexive is half. For this example, the occurrence/experience frequency of each compute sequence "$R_i \rightarrow C_v$" of operations is following fairly a uniform distribution as summarized in Table 3. If all distinct compute cycles occur following this probability distribution, the improvement of bit-lines charging/discharging in comparison to the conventional bit-lines can be summarized in Table 4. The results approximates the findings in [1], where the bit-line charging was reduced by about 68% and bit-line activity by 60%. High occurrence frequencies are dominated by the column C1 and approved by the voltage waveforms.

**Table 2:** Probability distribution for an operand XB=1, a passive bit-line selection (XBL↑) and reflexive, for M-CL with *m* CMCs and operations with *i* operands and *o* outputs.

| X | XB = 1 | XBL↑ | reflexive |
|---|---|---|---|
| p(X) | $\frac{1}{2}$ | $\frac{1}{2^i}$ | $1-\frac{(m-i)!(m-o)!}{m!(m-i-o)!}$ |

**Table 3:** Occurrence frequency of each compute cycle $R_iC_j$ in percentage (%) for $i \in \{1, .., 7\}$ and $j \in \{1, 2, 3\}$.

| | C1 | C2 | C3 | | Total |
|---|---|---|---|---|---|
| R1 | 15.23 | 1.17 | 2.34 | | 18.75 |
| R2 | 15.23 | 1.17 | 2.34 | | 18.75 |
| R3 | 5.08 | 0.39 | 0.78 | | 6.25 |
| R4 | 5.08 | 0.39 | 0.78 | | 6.25 |
| R5 | 15.23 | 1.17 | 2.34 | | 18.75 |
| R6 | 20.31 | 1.56 | 3.12 | | 25.00 |
| R7 | 5.08 | 0.39 | 0.78 | | 6.25 |
| | | | | | |
| Total | 81.25 | 6.25 | 12.50 | | 100.00 |

**Table 4:** Improvement (in %) of bit-lines charging/discharging XBL and YBL for each $R_iC_j$.

| | | R1 | R2 | R3 | R4 | R5 | R6 | R7 | | Total |
|---|---|---|---|---|---|---|---|---|---|---|
| XBL↑ | C1 | 15.2 | 15.2 | 0.0 | 0.0 | 15.2 | 20.3 | 0.0 | | 66 02 |
| | C2 | 1.2 | 1.2 | 0.0 | 0.0 | 1.2 | 1.6 | 0.0 | | 5.08 |
| | C3 | 2.3 | 2.3 | 0.8 | 0.8 | 2.3 | 3.1 | 0.8 | | 12.50 |
| | *tot* | 18.8 | 18.8 | 0.8 | 0.8 | 18.8 | 25.0 | 0.8 | | 83.59 |
| XBL↓ | C1 | 15.2 | 15.2 | 0.0 | 0.0 | 15.2 | 20.3 | -5.1 | | 60.94 |
| | C2 | 1.2 | 1.2 | 0.0 | 0.0 | 1.2 | 1.6 | -0.4 | | 4.69 |
| | C3 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | -0.8 | | –0.78 |
| | *tot* | 16.4 | 16.4 | 0.0 | 0.0 | 16.4 | 21.9 | –6.3 | | 64.84 |
| YBL↑ | C1 | 15.2 | 15.2 | 5.1 | 5.1 | 15.2 | 20.3 | 5.1 | | 81.25 |
| | C2 | 0.0 | 0.0 | 0.4 | 0.4 | 0.0 | 0.0 | 0.4 | | 1.17 |
| | C3 | 0.0 | 0.0 | 0.8 | 0.8 | 0.0 | 0.0 | 0.8 | | 2.34 |
| | *tot* | 15.2 | 15.2 | 6.3 | 6.3 | 15.2 | 20.3 | 6.3 | | 84.77 |
| YBL↓ | C1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | | 0.0 |
| | C2 | 0.0 | 0.0 | 0.4 | 0.4 | 0.0 | 0.0 | 0.4 | | 1.17 |
| | C3 | 0.0 | 0.0 | 0.8 | 0.8 | 0.0 | 0.0 | 0.8 | | 2.34 |
| | *tot* | 0.0 | 0.0 | 1.2 | 1.2 | 0.0 | 0.0 | 1.2 | | 3.52 |

The occurrence expectation for a compute cycle that execute a reflexive and conflictive operation is about 6.25% which is negligible. If the distinct compute cycles are independent and follow a uniform distribution then in a long run the ratio of writes that modifies a CMC will take half of the operations. However, as the number of pull down are more frequent, the calculated ratio of any bit-line experiencing a change is about 57.29% and the chances of charging XBL is 32.29%.

## 5. CONCLUSION

In this paper we introduced our prior work on compute-line based computational memory architecture (CCMA) for in-place storage processing based on compute-line (CL) concept using KEEPER. The KEPPER keeps both bit-lines stays distinguished from each other. CCMA, a memory-like structure with built-in computing capabilities, supports basic (and complete) binary operations on multiple local and/or remote data simultaneously. The architecture does extensively support computation and data locality. We introduced Minimal CL (M-CL) and showed how data movement can be reduced to almost null while executing built-in operations. Our contribution are investigating CCMA, studying its behavior/functionality and illustrating promising perspectives. Compiling real program needs to take into consideration the above finding. Generally, depending on the chosen application, CCMA with a suitable connection topology between its compute-lines can achieve massive parallelism.

## REFERENCES

1. D. Azougagh, A.Rebbani, and O. Bouattane. **Computational Memory Architecture Supporting in Bit-Line Processing**, International Journal of Computer Science and Network Security, vol. 18, no. 7, July 2018.
2. D. Azougagh, A.Rebbani, and O. Bouattane. **An Enhanced Bit-line Keeper for Computational Memory Architecture**, International Conference on Systems of Collaboration Big Data, Internet of Things & Security (SysCoBIoTS), Casablanca, pp. 1-5, December 2019. doi: 10.1109/SysCoBIoTS48768.2019.9028042.
3. D. Azougagh, A.Rebbani, and O. Bouattane. **A power Saving Bit-line Keeper for Computational Memory Architecture**, International Conference on Electrical and Information Technologies (ICEIT), Rabat, pp. 1-6, March 2020. doi: 10.1109/ICEIT48248.2020.9113299.
4. D. Azougagh, A.Rebbani, and O. Bouattane. **Bit-line Keeper based Computational Memory Architecture: varieties and comparisons**, 1st International Conference on Innovative Research in Applied Science, Engineering and Technology (IRASET), Meknes, pp. 1-6, Morocco, 2020. doi: 10.1109/IRASET48871.2020.9092329.
5. P. K. Meher, **LUT Optimization for Memory-Based Computation**, in IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 57, no. 4, pp. 285-289, April 2010. doi: 10.1109/TCSII.2010.2043467.
6. M.M.A. Basiri and S.K.N. Mahammad, **Memory Based Multiplier Design in Custom and FPGA Implementation**, In: El-Alfy ES., Thampi S., Takagi H., Piramuthu S., Hanne T. (eds), Advances in Intelligent Informatics. Advances in Intelligent Systems and Computing, Springer, Cham, vol. 320, pp. 253-265, 2015. doi: 10.1007/978-3-319-11218-3_24.
7. S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, **Compute caches**, in IEEE International Symposium on High Performance Computer Architecture (HPCA), Austin, TX, pp. 481–492, Feb 2017. doi: 10.1109/HPCA.2017.21.
8. M. Sayed and W. Badawy, **A new class of computational RAM architectures for real-time MPEG-4 applications**, The 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications, Calgary, Alberta, Canada, pp. 328-332, 2003. doi: 10.1109/IWSOC.2003.1213057.
9. Z. Chowdhury et al., **Efficient In-Memory Processing Using Spintronics**, in IEEE Computer Architecture Letters, vol. 17, no. 1, pp. 42-46, January-June 2018. doi: 10.1109/LCA.2017.2751042
10. S. F. Yitbarek, T. Yang, R. Das and T. Austin, **Exploring specialized near-memory processing for data intensive operations**, Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, pp. 1449-1452, 2016.
11. S. Khoram, Y. Zha, J. Zhang, and J. Li, **Challenges and Opportunities: From Near-memory Computing to In-memory Computing**, In ACM Proceedings on International Symposium on Physical Design (ISPD '17). ACM, New York, NY, USA, pp. 43-46, March 2017. doi: 10.1145/3036669.3038242
12. J. Ahn, S. Yoo, O. Mutlu and K. Choi, **PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture**, ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA), Portland, OR, pp. 336-348, 2015. doi: 10.1145/2749469.2750385
13. M.Siva Kumar, Syed Inthiyaz, J. Dhamini, A.Sanjay, U.Chandu Srinivas, **Delay Estimation of Different Approximate Adders using Mentor Graphics**, International Journal of Advanced Trends in Computer Science and Engineering (IJATCSE, Vol. 8, No. 6), pp. 3584-3587, November-December 2019, doi: 10.30534/ijatcse/2019/141862019.
14. Lavanya Maddisetti, Ranjan K. Senapati, JVR Ravindra, **Training Neural Network as Approximate 4:2 Compressor applying Machine Learning Algorithms for Accuracy Comparison**, International Journal of Advanced Trends in Computer Science and Engineering (IJATCSE, Vol. 8, No. 2), pp. 221-215, March-April 2019, doi: 10.30534/ijatcse/2019/17822019
15. 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

16. **Predictive technology model**, in 2018-2019, [Online]. Available: http://ptm.asu.edu/