Project:

MNEMOSENE

(Grant Agreement number 780215)

"Computation-in-memory architecture based on resistive devices"

Funding Scheme: Research and Innovation Action

Call: ICT-31-2017 "Development of new approaches to scale functional performance of information processing and storage substantially beyond the state-of-the-art technologies with a focus on ultra-low power and high performance"

Date of the latest version of ANNEX I: 11/10/2017

D4.6– First report on Initial CIM nano-architecture

Project Coordinator (PC):	Prof. Said Hamdioui Technische Universiteit Delft - Department of Quantum and Computer Engineering (TUD) Tel.: (+31) 15 27 83643 Email: S.Hamdioui@tudelft.nl
Project website address:	www.mnemosene.eu
Lead Partner for Deliverable:	Technische Universiteit Delft (TUD)
Report Issue Date:	22/07/2018

Document History *(Revisions – Amendments)*	
Version and date	**Changes**
10/5/2019	First version initiated by TUD
25/6/2019	Second version incorporating the contributions of all partners
19/07/2019	Final review and editing by TUD

Dissemination Level		
PU	Public	**X**
PP	Restricted to other program participants (including the EC Services)	
RE	Restricted to a group specified by the consortium (including the EC Services)	
CO	Confidential, only for members of the consortium (including the EC)	

European Commission

The MNEMOSENE project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under grant

The MNEMOSENE project aims at demonstrating a new computation-in-memory (CIM) based on resistive devices together with its required programming flow and interface. To develop the new architecture, the following scientific and technical objectives will be targeted:

- Objective 1: Develop new algorithmic solutions for targeted applications for CIM architecture.
- Objective 2: Develop and design new mapping methods integrated in a framework for efficient compilation of the new algorithms into CIM macro-level operations; each of these is mapped to a group of CIM tiles.
- Objective 3: Develop a macro-architecture based on the integration of group of CIM tiles, including the overall scheduling of the macro-level operation, data accesses, inter-tile communication, the partitioning of the crossbar, etc.
- Objective 4: Develop and demonstrate the micro-architecture level of CIM tiles and their models, including primitive logic and arithmetic operators, the mapping of such operators on the crossbar, different circuit choices and the associated design trade-offs, etc.
- Objective 5: Design a simulator (based on calibrated models of memristor devices & building blocks) and FPGA emulator for the new architecture (CIM device combined with conventional CPU) in order demonstrate its superiority. Demonstrate the concept of CIM by performing measurements on fabricated crossbar mounted on a PCB board.

A demonstrator will be produced and tested to show that the storage and processing can be integrated in the same physical location to improve energy efficiency and also to show that the proposed accelerator is able to achieve the following measurable targets (as compared with a general purpose multi-core platform) for the considered applications:

- Improve the energy-delay product by factor of 100X to 1000X
- Improve the computational efficiency (#operations / total-energy) by factor of 10X to 100X
- Improve the performance density (# operations per area) by factor of 10X to 100X

Table of Contents

1. Introduction

Nowadays, in computation systems the communication between the CPU and external memory becomes a bottleneck for the performance as well as energy consumption. One of the promising solution to tackle this challenge is “computation-in-memory (CIM)”; i.e., performing the computing within the memory rather than moving the data to the processing unit. In this field, many researches have already proposed different innovative circuit designs that demonstrate the efficiency of CIM.

In this report, we will present the architecture of our CIM tile and all the nano-instructions that currently supported in this design. We will explain in detail how our compiler translates *application kernels* (*micro-instructions*) into an executable format for our CIM architecture. Moreover, in dealing with an analog memory array as well as the digital periphery (including the controller), synchronization between the analogue and digital time domain becomes crucial and will be discussed in detail. Finally, in order to evaluate our CIM tile, we designed a simulator to estimate the latency and the performance of the CIM architecture to allow for design space exploration.

This report is structured as followed. Section 2 provides high-level information about CIM-tile architecture and all the analog as well as digital components which are employed in our design. Section 3 discusses the implementation of CIM-tile, and shows how our compiler and simulator work together to execute an application kernel. Section 4 evaluates two application kernels and presents some results.

2. CIM-tile architecture

2.1 CIM-tile overview

In order to perform an operation on CIM tile, proper data and voltage levels need to be provided. With the help of periphery circuits, these inputs can be generated and the output of the crossbar of the CIM tile can be captured to translate into the meaningful digital value. Figure 1 depicts the architecture of a CIM tile; it includes the required components and their communication signals which can be a control, digital, or analog data. The operations which can be executed on the crossbar are divided into two categories: 1) write operations and 2) computational operations. The computational operations include read, arithmetic, and logical operations. We will discuss these operations in detail in the following sections.

2.1.1 Write operations

In order to write data to a memristor in the crossbar, we have to specify in which *row* and which *column* the data needs to be written. Therefore, three registers are employed to capture this information.

- *Write Data (WD) register*: The data itself has to be written to the WD register. If the each memristor cell can only store two bits (binary), then the length of WD register is equal to the number of crossbar columns (say *c*). However, if the memristor cells have the capacity of multi-level storage (say *m* bits per cell), then the length of WD register is equal to the number of crossbar columns multiplied by *m*.
- *Write Data Select (WDS) register*: in order to write to specific columns in the crossbar, the information regarding such columns should be stored in WDS register. This register is connected to the control port of tri-state buffers in order to select appropriate columns and make the remaining columns (where we do not want to write the data into them) floating.

Therefore, the length of WDS register has to be equal to the number of columns in the crossbar and each bit of this register indicates whether the data has to be written in its corresponding column or not. Note that other implementations can be used in order to achieve this goal. Since the implementation does not have much effect on the abstract behavior of the architecture, we will not discuss the different types of implementation at this stage.

- *Row Select (RS) register*: To select the proper row, the RS register is employed. Each bit in this register corresponds to one row in the crossbar, and when the bit is active (not active), the corresponding row is selected (not selected). The size of this register strongly depends on the nature of the operation that should be performed, as it will be shown later; but in all cases, the size should be at least the number of rows.

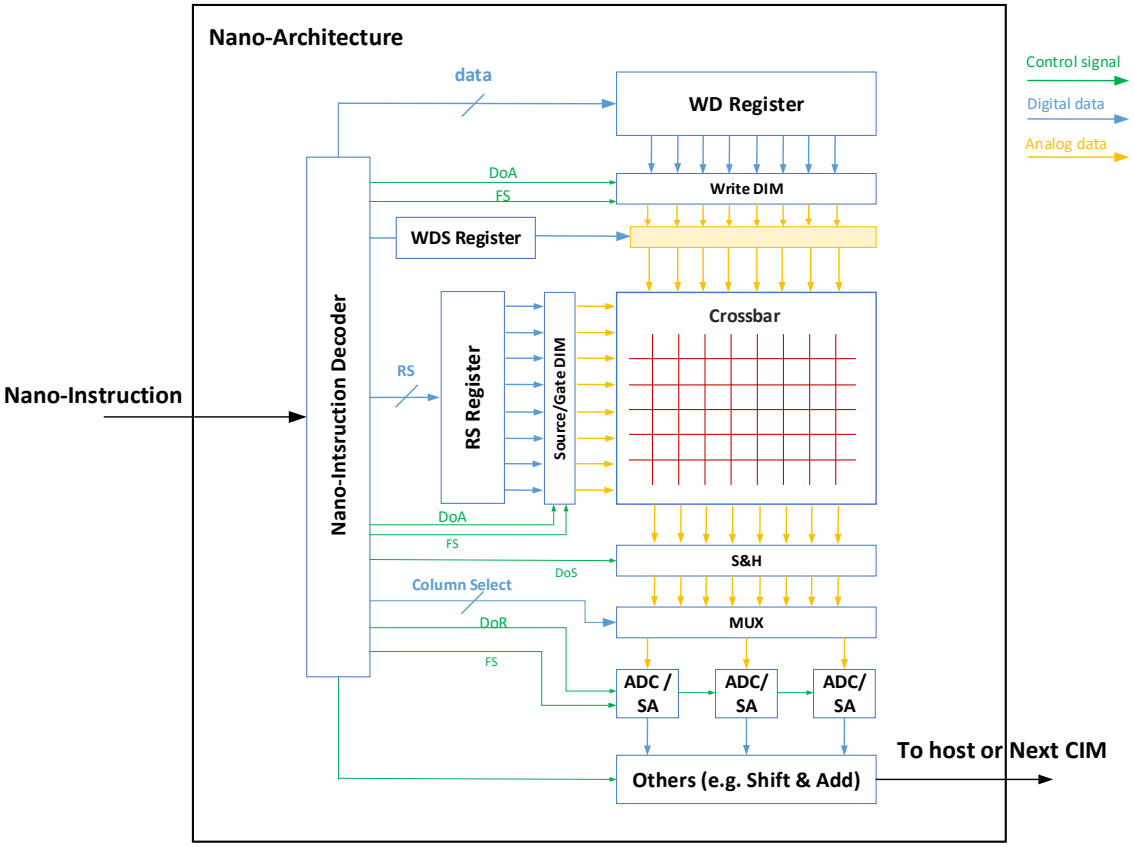

Figure 1: CIM Tile Architecture

It is worth noting that voltages that have to be applied to the memory crossbar depend on crossbar technology (e.g., PCM, RRAM, STT-MRAM) and they are usually different than the voltage used for peripheral circuits (e.g., digital circuits such as registers). Therefore, there is a need for converting the information (signals) from digital to analog domain to drive all inputs of the crossbar (i.e., bit lines, source lines, gate line) , using e.g., a Digital Input Modular (DIM). In the case each memory cell is able to store multi-bits, then Digital to Analog Converter (DAC) or Pulse Width Modulator (PWM) can be employed. When a row is selected, two different voltage levels have to be provided: one for the *source* line and one *gate* line of the target cells/ row; this means two *DIMs* (source and gate) are required to drive the cells in crossbar. On the other hand, when a column is selected, an appropriate voltage is provided via *Write DIM*. Therefore, we need in total three DIMs in this architecture in total.

As depicted in Figure 1, the DIMs receive two control signals: *Function Select (FS)* and *Do-Array (DoA).*

- Since the voltages need to be provided by DIM are different for write and computation operation, the control signal FS is used to guide DIM deliver the right voltage depending on the to-be-performed operation. The signal FS is derived from the nano-instructions which dictates what operations should be executed. Hence, DIMs, based on the information in the RS and WD registers as well as the FS signal, provide the required voltage levels for the crossbar. Once RS, WD, and WDS registers are correctly initialized, and the FS signal is received, then the data can be written into the crossbar.
- In order to do it in a synchronized way, DIMs can apply the voltages to the crossbar only when the "*DoA"* signal is received. Subsequently, the crossbar starts to write the data into the requested location and DIMs have to keep their output voltage the same while the crossbar is busy. DoA is an important signal for timing and pipelining of the architecture. We will give more information about it in the next sections.

2.1.2 Computational operations

In our design, the computational operations are read, addition, multiplication, logical AND, OR, and XOR for which the output is generated outside of the crossbar and has to be read by the periphery circuit in the architecture. For the sake of completeness and to be consistent with other deliverables specially Deliverable D4.4, we will present our terminology for CIM classification again as follows:

- CIM-Array (CIM-A): the computing result is produced within the array. As the memory array is based on memristive devices in our targeted architecture, the output is stored in the form of resistance state.
- CIM-Periphery (CIM-P): the computing result is produced within the periphery. When the periphery is based on CMOS technology, the output is produced in a form of voltage or current, and stored in a form of voltage.

The focus on the project is CIM-P. Depending on either the operands are all stored in the crossbar of only partially stored in the crossbar, two CIM-P sub-classes are defined:

CIM-Pr (all operands are resistive inputs**)**: the computing result is produced within the periphery, while the inputs are *all* resistive stored in the array.

All the computational operations considered in this project are within this subclass, except vector-matrix multiplication. Note that as the data is already stored within the crossbar, no need for use of WD and WDS registers when performing CIM-Pr operations. RS register, however, is employed to select the appropriate rows. As an example, for the read operation, we just need to specify a single row from which the data has to be read. However, if a AND logic operation should be performed, then the two rows (corresponding to the AND operands) should be selected. Note that the intension is to perform bitwise AND operations of the elements of the two rows.

After initializing the RS register, the FS control signal has to determine the type of computational operation. Therefore, write DIM makes the whole columns float and row DIMs for both source and gate line will produce proper voltage levels. Subsequently, similar to the write operation, the DoA signal is issued to start the computation in the crossbar. Despite the write operation, the crossbar generates results which need to be captured by the Sample and Hold unit (S&H). In order to inform the S&H that the data in its input is ready to be captured,

the Do Sample (DoS) signal has to be set. This unit and its control signals are important for the timing of the architecture. We will discuss it in detail in Section 3.3. The S&H unit can be considered as a buffer in the analog domain. When this unit finishes its work, the DoS signal will be reset and the input is disconnected from output until the next DoS.

Since the data which is available on the output of S&H is analog data, we have to use Sense Amplifier (SA) or Analog to Digital Converter (ADC) units to make it usable for the host processor. Because these two units consume much power and area, usually it is not possible to allocate an ADC or SA to each column. Therefore, several columns have to share one ADC or SA. Accordingly, we need analog multiplexer(s) to select columns one by one and connect them to the ADCs. The data for the control part of the multiplexer(s) is provided by the Columns Select (CS) signal. We will discuss its implementation and the length of this signal in Section 3.2. Whenever ADCs convert the data into digital format, then the CS signal would select other columns and this process will continue until all the required columns are read. As mentioned before, if the technology does not allow to have multi-voltage levels on the crossbar rows or if a big number distributed over several crossbars due to the limited values that can be stored in a memristor, then we need digital shift and addition units to calculate the final result. In the current version of this architecture, these units are not implemented. Therefore, we will keep the discussion about their necessity and implementation for the future.

CIM-Ph (input are hybrid inputs): the computing result is produced within the periphery, while the inputs are *partly* resistive stored in the array and *partly* voltage (or current) provided via the periphery.

Among computation operations, multiplication is somewhat different than others just in the sense that the data for one of its operands exist in the crossbar and the second one is provided by the RS registers. Accordingly, in the case that the technology support to have *n* voltage levels on the source line of memristors crossbar, then the size of RS is equal to the number of crossbar rows multiplied by *n*. Afterward, all the other components in our architecture are used in the same way as explained in CIM-Pr section. We will provide more in section 3.2.

2.2 Nano-Instruction set architecture (nano-ISA)

In order to execute an operation on the crossbar, several steps need to be performed. A controller is employed for each CIM tile (i.e., one crossbar array and its periphery circuits) to take care of these steps. Each step is considered as a *nano-instruction* that the controller has to fetch (from the nano-instruction memory) and execute. In this section, we will discuss the list of current nano-instructions that enable the targeted CIM operations. *Table 1* shows the summery of the current nano-instructions.

Table 1: List of nano-instructions

Nano-instruction	Opcode	Operands	Purpose
Row Select	RS	Data to fill RS register	Write/Compute
Write Data	WD	Data to fill WD register	Write
Write Data Select	WDS	Data to fill WDS register	Write
Function Select	FS	Control bits to select right operation	Write/Compute
Do Array	DoA	-	Write/Compute
Do Sample	DoS	-	Compute
Columns Select	CS	Data to fill CS register	Compute
Do Read	DoR	-	Compute

- **Row Select (RS):** This instruction is responsible to fill the RS register in order to select the rows that have to be activated for the upcoming operation (e.g. for CIM-Pr operations), and in some cases provides the input data as well (e.g., for CIM-Ph operations). This nano-instruction consists of two parts; opcode which is *RS* and the data used to fill in the register. In the current version of CIM-architecture, the length of "RS" instruction is equal to the size of RS register in addition to the opcode size.

- **Write Data (WD):** Same as row select, we need another nano-instruction to fill in the WD register. The Write Data instruction consist of opcode, which is WD, and the data to be written into the register. As mentioned in Section 2.1, size of WD register depends on the number of levels that memristor cells support. Therefore, the length of this nano-instruction depends on the used device/cell technology.

- **Write Data Select (WDS):** As explained before, this register is responsible for masking columns which should be not selected for write. The instruction consists of the opcode WDS, followed by data to fill in WDS register. Since the length of WDS register is equal to the number of crossbar columns (each bit in the register corresponds to one column), the instruction has a large size; it is equal to the size of the opcode plus the size of WDS register.

  Note that the size of all above nano-instructions (RS, WD and WDS) are quite large since they consist also of the data to fill in these 3 registers. Although there are solutions to solve this problem, for the sake of simplicity we will go with this approach for the first version of simulator and compiler. Later on in the project, we will work out different options and solutions.

- **Function Select (FS):** To perform write and computation operations, different voltage levels need to be applied to the crossbar. Therefore, DIMs have to be configured accordingly. Hence, DIMs have to receive appropriate information regarding the function to-be-executed. The FS nano-instruction is used for this purpose; it provides appropriate information to configure the DIM to deliver the required voltages for the to-be-executed operation. Note also that FS instruction is used also to configure the read circuitry (ADC/SA in Figure 1) for appropriate computing operation. For example (and as shown in Deliverable D4.4), the Scouting logic, and depending on which bit-wise logic operation should be performed, specific configuration (i.e., selecting the correct current references) is needed.

- **Do Array (DoA):** Do Array is an important nano-instruction that activates the DIMs. This clearly separates the "execution" from the configuration phase of the DIMs via the RS, WD, and WDS registers. This will allow for future code scheduling optimizations.

- **Do Sample (DoS):** After a certain delay, the memory array will produce its results to be captured within the peripheral circuits. The *DoS* nano-instruction is used to activate the sample and hold circuitry to capture the results and hold them, until the next Do Sample nano-instruction is issued. This gives time for the ensuing read circuits (e.g., SAs and/ADCs) to read out the results while the array can perform the "next" operation. This conceptual distinction between execution and read-out of results will allow for future code scheduling optimizations.

- **Column Select (CS):** The CS nano-instruction is responsible for correctly connecting the sample and hold circuitry with the read circuits (SAs and/or ADCs). As the number of SAs/ADCs are expected to be fewer than the number of columns in the arrays, the CS nano-instruction will control (analog) multiplexers to allow for sharing of SAs/ADCs among multiple columns. Moreover, the CS nano-instruction can also be used to selectively read-out only specific columns from the array. In the current version, the size of this instruction is equal to the size of the opcode plus the size of data operand, which is assumed to be equal to the number of columns. We will discuss this nano-instruction in more detail in Section 3.

- **Do Read (DoR):** Due to the high power consumption of ADCs, they are activated whenever the analog data is ready on their inputs. Hence, this is done one the DoR is executed. Similar to DoS , this instruction has just an opcode.

2.3 Memory index architecture (IMEC)

The next generation of advanced image and video processing kernels often exhibit a mix of regular and irregular (or data-dependent) memory accesses. (This has already been introduced in the earlier deliverable D1.2.) Moreover, they require data access which goes beyond the immediate local neighbours. Typically, they need a medium-size neighbourhood around the current pixel access. Typical values can be from 7x7 up to 11x11 pixels of 23 bits (in the case of colour images); and these do not directly fit in the local register files, so they need to be accessed from SRAM caches or scratchpad memories. This limits the efficient mapping of these kernels on modern GPUs.

Based on the characteristics of the complex modern multimedia (video, graphics, image etc) signal processing applications, IMEC has proposed a novel memory indexed CIM-P architecture. This architectural technique targets the partly irregular index operations on the multi-dimensional arrays that are common in multimedia applications. This relieves the cores in the CPU or GPU from the complex addressing schemes for the extraction of image data and improves the system efficiency. This architectural idea is described in detail in the other part of the D4.6 **(restricted/confidential version)**.

3. Implementation

3.1 Assumptions and constraints

In this section, all the assumptions made in the current version of the simulator and compiler as well as constraints which comes from technology side are discussed.

- The number of resistance levels supported by a memristor cell/device is assumed to be a variable: This is one of the important constraints with a large impact not only on the control unit, but also on the compiler. This parameter could vary for different memristor technologies. Although this number could be strongly linked also to some characteristics of the crossbar array such as latency and reliability, we here are just interested to develop a generic but parametrized implementation that can be used for any memrisive device. Having multi-level storage per memristive device requires more complex drivers (DIMs) in order to deliver the different required/ appropriate voltages. Moreover, multi-level cell storage implies increase in the size of the WD register, which stores to-be-written data in the memory crossbar. In fact, multi-level storage do not only impact the DIMs and WD registers, but it could requires changes in other registers. Moreover, the loading of this register over a (usually smaller) memory bus can also impact the overall latency. One way to deal with this is to use pipelined operations/ steps. However, this is not integrated in the current version; it will be included in the final version.

- The number of voltage levels that can be used to drive each row line of crossbar is assumed to be only 2: As mentioned in Section 2.1, for multiplication operation, the data for one operand has to be provided into the rows of crossbar. Hence, It could be suitable to have multi-voltage levels to drive the rows. Obviously, this number of voltage levels has direct impact on the size of RS register and the overall performance of the operation. In the current implementation, we assume that the technology just supports *two voltage* levels. More explanation regarding the implementation of multiplication while using only two voltage levels can be found in Section 3.2.

- The maximum number of cells that can be written simultaneously is limited: This constraints applies to the write operation. As Figure 2 shows, in order to simultaneously write data to the memristors of the same row, proper voltage level/ driving current need to be applied to each cell. As a consequence, the total current driving all the accessed cells (each via its own bitline) has to follow through the common source line. Since there is a maximum current that each wire can tolerate, fabrication technology limits the number of memristors that can be simultaneously written. Accordingly, this constraint would have effect on the performance and power of the system. Finally, it is the compiler responsibility to generate codes in such a way that this constraint is met; otherwise there is no guarantee on the correct functionality of writing the CIM die.

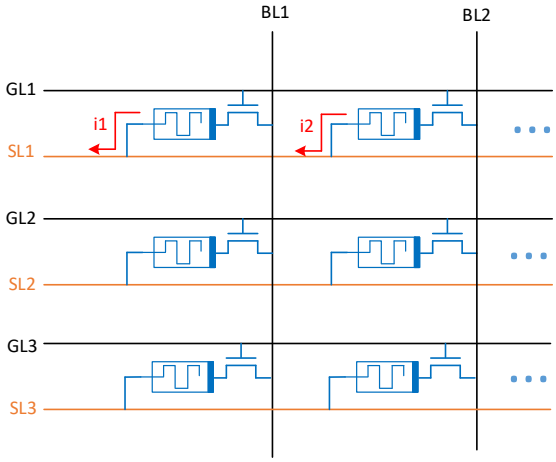

Figure 2: Limitation of write operation

- The precision of ADCs is a limiting factor for the number of rows that can be selected simultaneously: As mentioned in Section 2.1, the add operation allows for summing up the value of several memristors which are in the same column. The important point is that, the result is presented as different voltage or current levels. As the number of rows who are contributing to the addition increase, more voltage or current levels are generated and the ADCs which are responsible to translate these levels into the digital data have to be able to distinguish these levels. The ADC itself is the most power and area consuming unit in the architecture and improving the precision will increase the power exponentially. Therefore, due to the power constraint, and thus limited precision for the ADCs, mostly it is not possible to select all the rows at the same time. This limitation, which again has impact on the performance and power of the system, has to be taken into account by the compiler. It is discussed in more detail in Section 3.2.
- Mapping of data inside the crossbar can be different than other memory units in the system: The data for the WD and RS registers will be provided either from memory in general processor side (DRAM, caches) or other memristor crossbars. In both cases, it is possible that different mappings of data into the memory employed between the destination and source. For instance, in 32-bit general processors one word of data stores contiguously into the memory, but this is not always the same case in the memristor crossbars. Due to the limitations and characteristics of crossbars, storing data in contiguous columns is not always an efficient solution. Therefore, the hardware or compiler should take care of this mapping. For the current implementation, we just assume that the data for RS and WD register is provided in the instructions and it was reordered before based on the crossbar mapping.
- The number of ADCs is usually less than number of columns in the crossbar: Due to the high power and area consumption of ADCs, it is not possible to consider one ADC for each column. Hence, several columns have to share one ADC, which may affect the performance of the system. This constraint imposes modification on the hardware and compiler. In this version of compiler and simulator, this constraint has been taken to account. Therefore, the result that comes out of ADCs maybe needs more processing such as register shifting. More information will be provided in Section 3.3. For sure, both the controller inside the hardware as well as the compiler has to manage it, but we keep it for the next version of the implementation

3.2 Micro-to-Nano compiler

In Section 2.2 we explained all the current nano-instructions that our CIM core supports. In Deliverable D3.1, the list of *micro-instructions* was introduced that acts as the interface between an application specified in higher level programming language and the CIM tile. Consequently, these micro-instructions need to be translated to a sequence of *nano-instructions*, i.e., a nano-program. In this section, we will discuss how this is achieved for the following cases:

- Writing a matrix in the crossbar of the CIM tile
- Reading a matrix from CIM
- Performing Matrix-Matrix Multiplication
- Performing Vector (bit-wise) logical operations (AND,OR,XOR)

Next the above cases are discussed in details.

Writing a matrix in the crossbar of the CIM tile

Using this micro-instruction, a matrix from external memory will be written to the crossbar array. In order achieve this, information regarding the size of matrix and its address both in external memory and crossbar has to be provided. The format of this kernel is shown following.

- *store Addr_Destination i j Addr_Source p q*

The first part of this micro-instruction format is the opcode, by means of which the kernel can be recognized. Subsequently, in order to find out where the data is saved in the external memory, we have to specify the location of the first element (*Addr_Source*) and the size of the matrix (*p and q*). The *Addr_Destination* select which crossbar (if more than 1) the data should be stored in, while row (*i*) and column (*j*) index give the location of the first element in the selected crossbar. Figure 3 illustrates the different parts of this micro-instruction. In order to write this matrix into the crossbar, we have to split the operation into several steps in which just one row of matrix is selected at time to write it into the crossbar. To do that we have to provide some nano-instructions in the right order according to the architecture which was presented before.

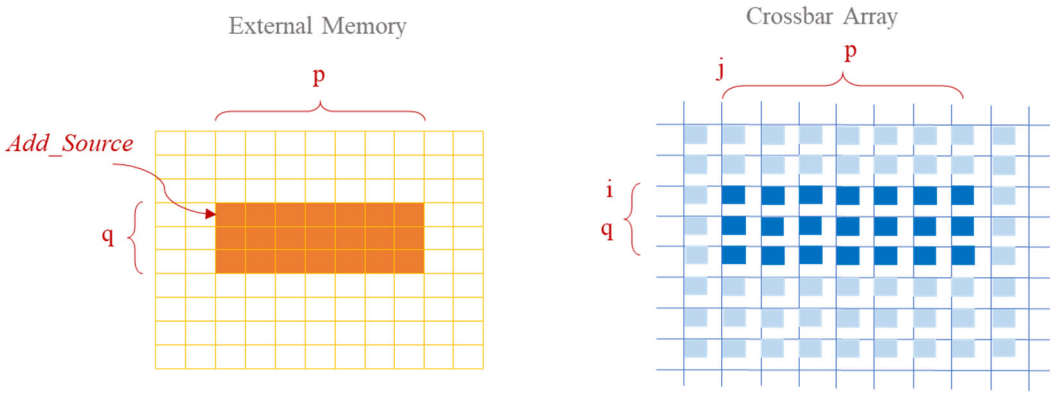

Figure 3: writing a matrix from external memory to the crossbar

The following series of instructions shows how we can perform the write mactrix operation; this is also illustrated by Figure 4

1. RS (Row Select instruction): In the first step, we have to select the first row where the data has to be written by filling the RS register using *RS* nano-instruction, which is responsible for activation and deactivation of rows.
2. WD (Write Data instruction): we need the *WD* nano-instruction to store the data coming from the first row of the matrix into the WD register.
3. *WDS (Write Data Select* instruction): Next, based on the number of columns in the matrix we have to fill the WDS register by issuing the *WDS* nano-instruction.
4. *FS (Function Select instruction):* To generate the suitable voltage levels for the crossbar all the drivers have to be configured using *FS* nano-instruction. In the figure below, the operand WR is used to indicate that it is about write operations; hence, drivers for such operations should be selected.
5. *DoA (Do Array* instruction): at the end, by the issuing the *DoA* nano-instruction the data starts to be written into the requested location of the crossbar.
6. Repeat the steps 1 to 5 for all rows of the matrix.

Figure 4 depicts a sample code generated by the compiler for this operation to be performed in a 32*32 crossbar.

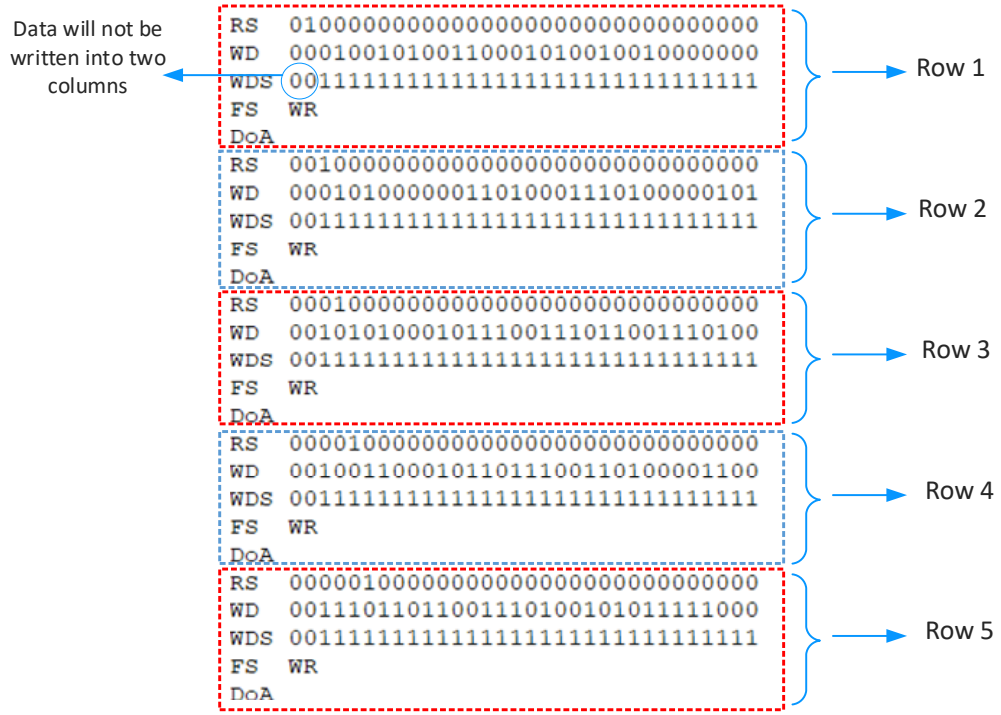

Figure 4: Sequence of nano-instructions for writing a matrix into crossbar

Reading a matrix

In this micro-instruction, the value stored inside a matrix of memristors has to be read. Similar to the previous nano-instruction, the location and dimension of the matrix need to be specified in the nano-instruction format which is written below:

- *read Addr_Destination Addr_Source i j p q*

In the case that we have several host processors, the data read from the crossbar has to be sent to the processor which asks for it. For this purpose the *Addr_Destination* is used. Although the architecture and communication between the general processing unit and CIM core have not been established yet, the term *Addr_Destination* provides this information for us. In our first version of the simulator, we ignore external communication. Therefore, these terms are ignored for now.

Similar to the write operation, to read a matrix from crossbar we have to split it into several steps in which just one row of the matrix should be read. The process is as follows and is illustrated in Figure 5 generated by the compiler reading two rows:

1. RS: activate the desired row one at a time.
2. FS: configure the drivers to generate the appropriate voltage levels needed for reading operation. Note that as it is about the read operation (indicated by RD in the figure).
3. *DoA:* execute the read.
4. *DoS (Do Sample instruction):* The results received from the crossbar have to be read and hold in a wide enough register using *Do-Sample* nano-instruction. Thereafter, these analog results should be converted to digital and send to the outside world. As usually the number of ADCs integrated in a CIM is smaller than the max bandwidth of CIM, results should be (analog) multiplexed, which is the job of the next instruction.
5. *CS (Column Select instruction):* selects part of the obtained results to be send to the ADCs.
6. *DoR (Do Read)*: execute read by activating ADC to generate the digital output.
7. Repeat steps 5 and 6 till all the sampled data is send to the digital output latches.
8. Repeat steps 1 to 7 till all rows are read.

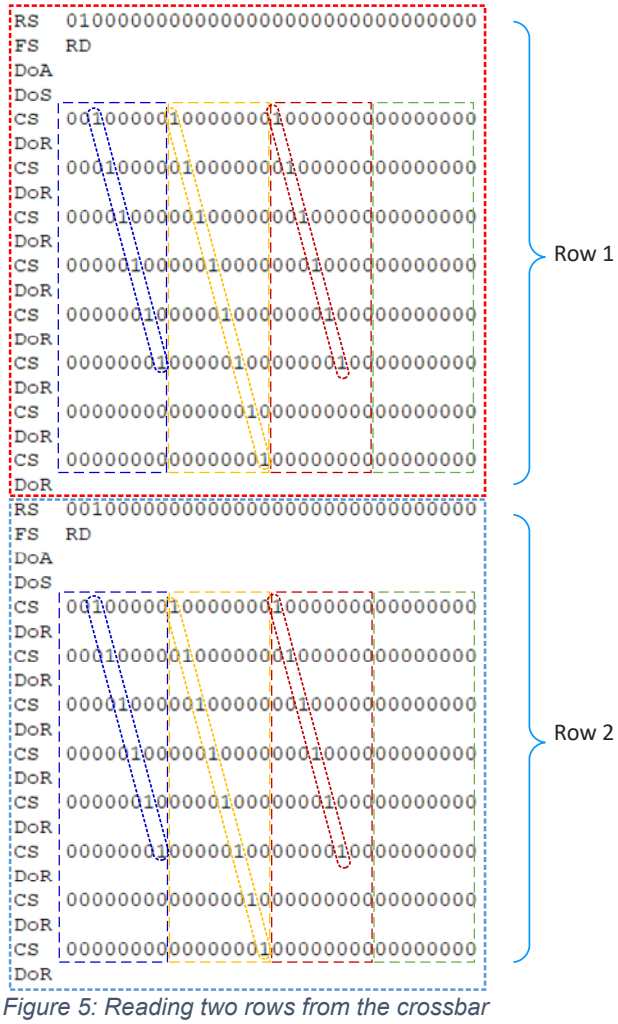

Figure 5: Reading two rows from the crossbar

Error! Reference source not found. shows how the compiler groups all the columns and how each group share one ADC. Therefore, in each group, only one column can be activated in a moment. Accordingly, the length of nano-instructions for reading one row depends on the number of ADCs and number of columns, which has to be read.

Matrix-Matrix Multiplication

In this part, we will explain how two matrices, one inside and the other one outside the crossbar, can be multiplied using our CIM architecture. The only difference from traditional matrix-matrix multiplication is that instead of multiplication of rows to columns, columns from first matrix would be multiplied to columns of the second matrix. The instruction format for this micro-instruction is as following:

- *MMM Addr_Destination Add_ExtMem p e Addr_Cross i j q*

The first part of this micro-instruction format is the opcode. Similar to the previous micro-instructions, in order to find out where the data is saved in the external memory, (*Addr_ExtMem*) is used and the size of the matrix is given by (*p and e*). The *Addr_Destination* select which crossbar (if more than 1) the data should be stored in, while row (*i*) and column (*j*) index give the location of the first element in the selected crossbar. Moreover, the number of rows in both matrixes has to be the same, but they can have a different number of columns. Hence, the size of the matrix in the crossbar is presented by (*p and q*). Figure 6 illustrates the different parts of this micro-instruction.

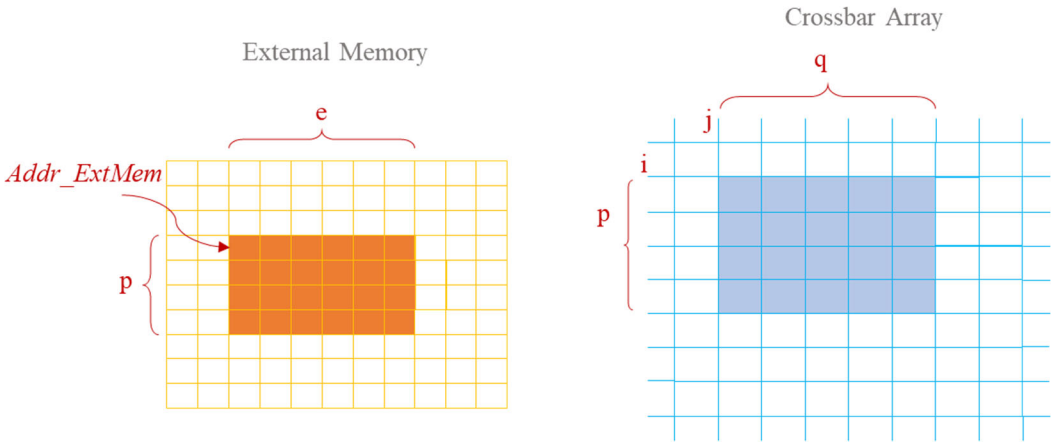

Figure 6: Multiplication of two matrixes

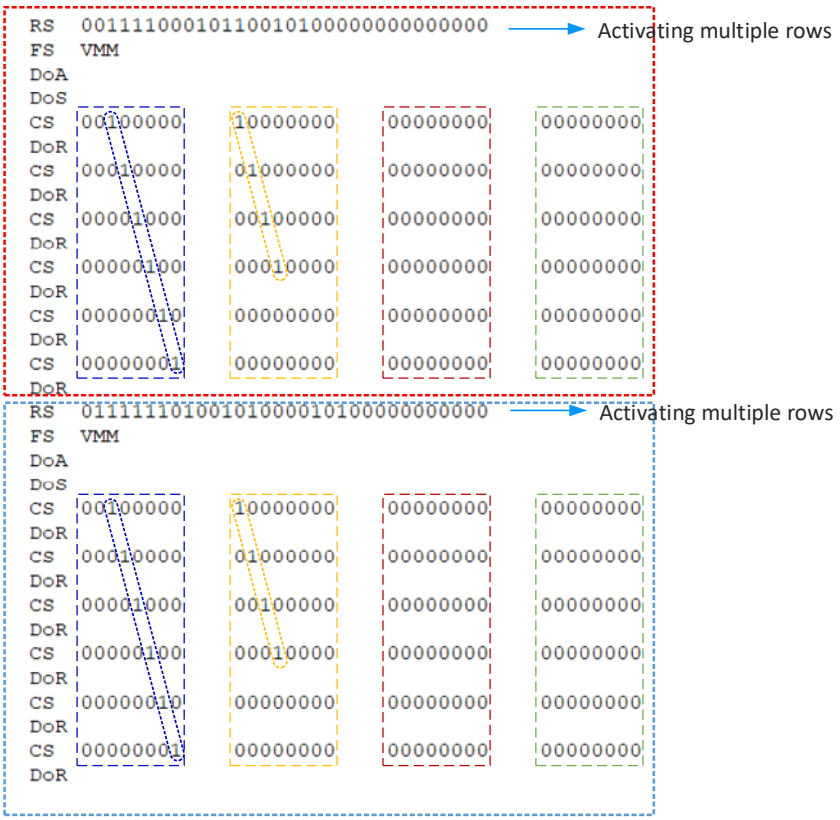

Figure 7: The code generated by compiler for multiplying two columns to an array inside the crossbar

To perform this multiplication, we need to divide it into several steps in which one column from external memory would be multiplied to whole the array in the crossbar. The only difference from read operation is that here we use the *RS* register not only for activating of rows in the crossbar, but for providing data to the rows as well. In our current simulator version, since we assume that drivers for source line can just generate two voltage levels, the data for each row inside the *RS* register should be a binary value. In other words, the data for the crossbar rows can be interpreted as activation or deactivation information as well. The sequence of nano-instructions to multiply two columns from external memory to matrix inside the crossbar is as follows and illustrated in Figure 7.

1. RS: despite the read operation in which only one bit of RS should have the value “1”, here multiple rows can be activated at the same time.
2. FS: configure the drivers and sense amplifiers to generate the appropriate voltage levels required for vector-matrix multiplication (VMM).
3. *DoA:* execute the multiplication in the crossbar.
4. *DoS (Do Sample instruction):* as explained before, the results received from the crossbar have to be captured and hold using sample and hold circuit when this nano-instruction is issued.
5. *CS (Column Select instruction):* selects part of the obtained results to be sent to the ADCs.
6. *DoR (Do Read)*: activating ADC to generate the digital output. In Figure 7, ADCs which are allocated to the last two groups of columns are always powered off, since the values of those columns are not needed to be read.
7. Repeat steps 5 and 6 till all the sampled data is send to the digital output latches.
8. Repeat steps 1 to 7 till all columns of the matrix in the external memory are multiplied.

Vector logical operation (AND,OR,XOR)

Our CIM architecture can also perform logical operations inside the array. In this architecture, both operands of operations, which can be a vector, are stored inside the array. The location and length of these two operands have to be provided in the micro-instruction which is written as follows:

- *logical_X Addr_D Addr_S i j p q*

where *X* can be *AND*, *OR*, or XOR. Addr_D is the address in which the result of the operation would be sent into it. Addr_S is the index of crossbar where operands are stored inside it which is used when there are multiple crossbars. Besides, i and j indicate the rows inside the crossbar where the first and second operands of the operation are stored. Finally, p and q show the columns starting point and the length of operands, respectively. The sequence of nano-instructions is the same as read operation with the small difference in RS register where more than one row should be activated. Moreover, based on the type of logical operation the reference current for sense amplifier has to be changed, which is performed by FS nano-instruction.

3.3 Nano-simulator

The proposed CIM architecture is generalized and capable of targeting different technologies with different configurations of the peripheral devices/circuits. Timing and power estimates of the modules within the CIM architecture are obtained from low-level models produced by other partners in the MNEMOSENE project. These numbers configure specific parameters of the simulator. In this manner, we can easily perform design space exploration between different applications and different technologies.

The simulator reads the nano-instruction file line by line. Each instruction is fetched, decoded, and executed by the digital controller inside the simulator and whenever its execution is completed the next instruction will be started. The CIM tile consists of digital and analog components which need to be synchronized with each other. Accordingly, our nano-instructions divided into two groups which are responsible to fill in the data into the digital registers and controlling the analog circuits. To ensure our system is synchronized, we have to capture the latency of analog circuit, otherwise, we may lose data if the speed of fetching instructions is greater than the latency of analog circuits.

As can be observed in Figure *8*, the crossbar, the sample and hold unit, and the ADCs are the three analog circuits in our design which are controlled by *Do Array*, *Do Sample*, and *Do Read* nano-instructions, respectively. These three nano-instructions have to be fetched and decoded as other instructions, but their execution times depend on the latency of their corresponding analog circuits. This posed a challenge in the design of our simulator. We devised two solution. First, we can implement counters inside the digital control unit to capture the latency of these three circuits. For instance, when we issue *DoA* signal, the counter starts to count and whenever it reaches the predefined value, next nano-instruction can be executed. Second, the analog circuits are designed in such a way that they can generate a done signal internally whenever the final results are ready. In this case, we need a complex and smart

circuit, but in the simulator both options can be supported since it just mimic the abstract analog behavior of each component in the design.

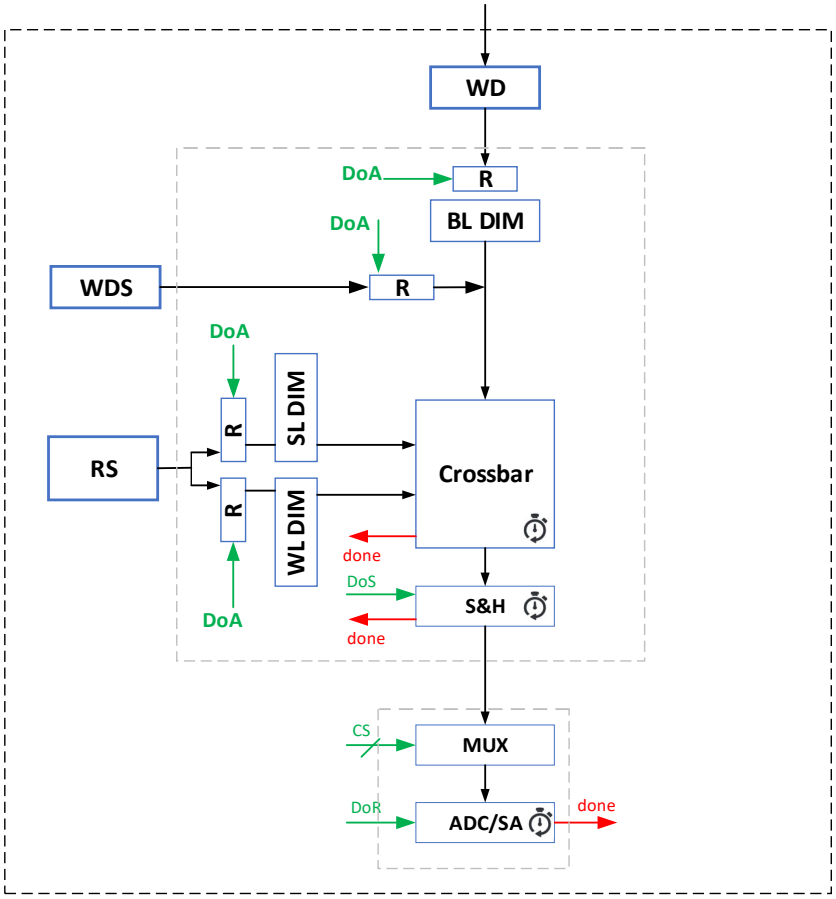

Figure 8: Synchronization of analog and digital circuits using counter or done signal

3.4 Potential Future Work

In this section, we present several possible directions for our future work in the design of our simulator and nano-compiler.

- As already discussed, a nano-instructions can be executed whenever the previous one is finished. Therefore, one potential improvement is to make the architecture pipelined in order to enhance the performance of the applications. To support pipelining, both compiler and simulator have to be updated.
- New components such as *Shift and Accumulate* can be added to the periphery in order to improve the capability of architecture. As new components are added, new nano-instructions are needed to be defined.
- In the current version of our architecture, all the data required by registers are included in the nano-instruction. Since the length of the registers is quite big, we need a big instruction memory as well. Therefore, a new way of accessing data has to be proposed.

4. Evaluation

In this section, we will investigate the impact of clock frequency and number of ADCs on the performance of the system using write and read operation. These are preliminary results as the evaluation is still ongoing. First, a brief overview on the simulation set up is provided, thereafter the results are presented.

4.1 Simulation setup

In this section, we will our initial results extracted from the simulator to clarify the behavior of the CIM architecture. To obtain these results, arbitrary micro instructions run in our CIM tile contain 256*256 crossbar. Furtheremore, we assume that the memristor devices can only store binary values and just 2 voltage levels can be used for the crossbar source lines as well. Therefore, the length of RS, WD, and WDS registers, as well as CS, are 256 bits. The simulation model consists of the 1T1R array using Resistive RAM (RRAM) technology. The targeted technology is RRAM using a 1T1R configuration. The delay parameters were derived from SPICE models of 40nm CMOS technology. The simulation parameters regarding delay of analog circuits are summarized in Table 2.

Table 2: Analog circuits delay

	Delay (ns)
Crossbar write/read latency	100
S&H delay	1
ADC delay	2

4.2 Simulation result

For the following results, we consider that fetching and execution of each nano-instruction take only one clock cycle that they are parameterized. Furthermore, we assume that the width of the data bus is large enough to fill in each of the registers in one clock cycle.

Figure 9 depicts the impact of clock frequency on the latency of the write operation. In this case, 20 rows from external memory have to be written into the crossbar. We can see that:

- since the latency of analog circuits is not dependent on the digital clock frequency, the relation between clock frequency and total delay is not linear.
- decreasing clock frequency in some ranges (e.g. from 2 to 1GHz) does not have much overhead on the total latency, which means without losing much performance we can gain power.

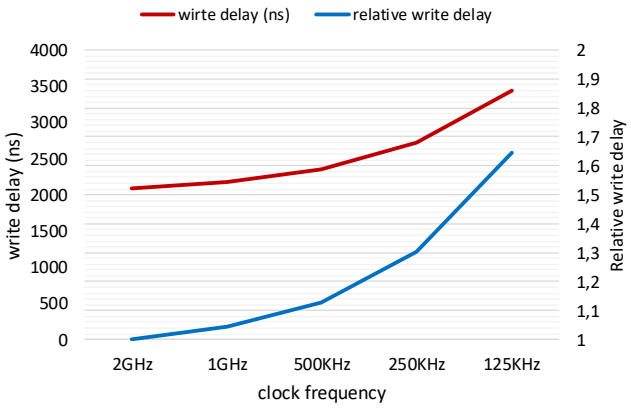

Figure 9: Effect of clock frequency on latency of write operation

Figure 10 demonstrates this conclusion in a different way. This figure depicts the impact of clock frequency on the contribution of each part of the architecture over total latency. Based on this, the latency of the crossbar has the biggest portion over the total latency of write operation by far. At some point, by increasing the clock frequency, some parts of the CIM tile which works with the digital clock have almost no contribution to the latency of the system. Therefore, increasing the clock frequency after some points will not have a much positive effect on the performance; rather, increases the power consumption. In conclusion, finding the optimal digital clock frequency based on the crossbar technology and application requirements is crucial.

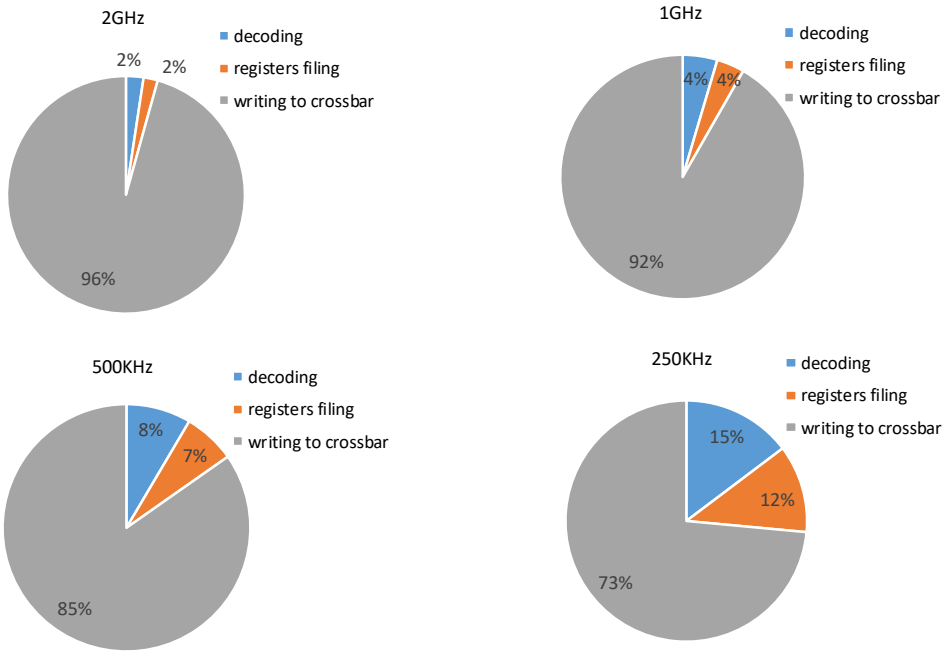

Figure 10: Contribution of each part in CIM tile to the latency of write operation

We performed an experiment for a read operation in which a matrix with 20 rows by 256 columns is read from the crossbar. Despite the write operation, all the analog circuits in the architecture need to be used in this operation, which imposes a bigger latency on the system. In

Figure 11, similar to the write operation, we can see the effect of clock frequency on the latency of the system, which ends up to the same conclusion explained before. The result shown in this figure achieved based on 4 numbers of ADC for the whole crossbar.

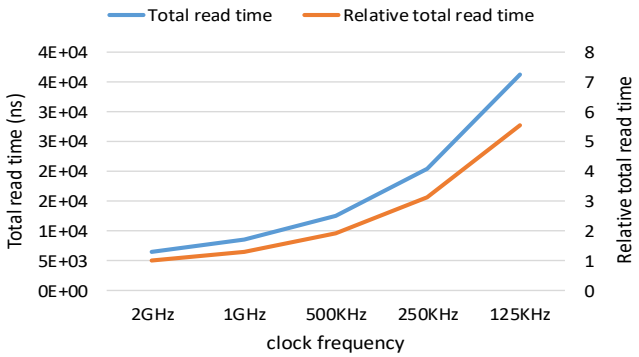

Figure 11: Effect of clock frequency on latency of read operation

Figure 12 demonstrates the impact of ADC numbers on the total latency of the system in 2GHz clock frequency. This figure clearly shows how big the effect of ADC numbers is on the performance of the system. As we increase the number of ADCs, the contribution of this part on the total latency will be decreased (see Figure 13). Accordingly, at some point (e.g. 8 number of ADCs) increasing the number of ADCs, provides just small improvement on the total latency, which can be unacceptable if power and area overhead imposed by this unit are taken into account. In conclusion, employing maximum number of ADCs to gain high performance is not always a good idea and the designer has to find the proper number based on the application constraints as well as system characteristics from device to architecture level.

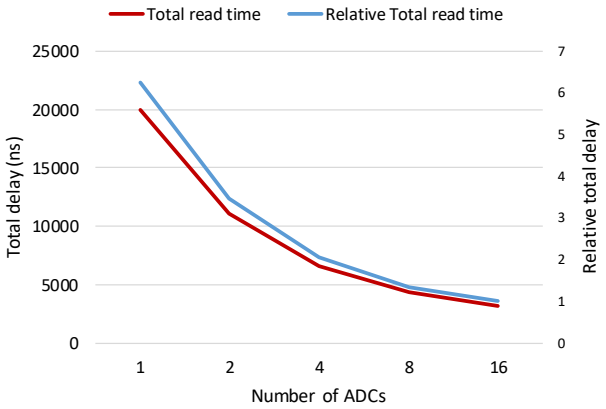

Figure 12: The effect of number of ADCs on the total latency

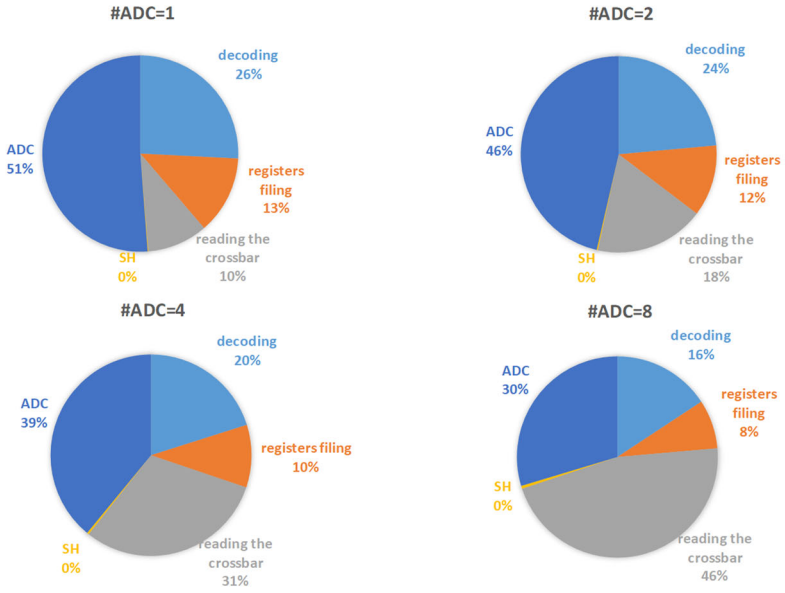

Figure 13: Contributions of each part in CIM tile to the total latency in 2GHz clock frequency