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

D3.1– Initial macro CIM architecture and CIM-ISA

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:	Eindhoven University of Technology (TUE)

Report Issue Date:	31/12/2018

Document History
(Revisions – Amendments)

Version and date	Changes
1.0 08/11/2018	Outline updated
1.1 10/12/2018	Draft prepared
1.2 20/12/2018	Final version

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 agreement No 780215

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

1.1 Motivation

As the data, which is required to be processed, is growing at a galloping pace, the need for brand-new processor architectures, which enjoy new paradigms of computing, becomes vital. Existing processors are based on the Von-Neumann architecture, in which the processing units and memory banks are placed apart from each other. Therefore, every time a data is needed to be processed, it must be fetched from memory, transferred to the processor, and the result, again, must be transferred to memory to be stored back. Considering this behaviour, and given the data which is needed to be processed is significantly large, this paradigm imposes several problems, such as performance reduction, inefficient energy consumption, etc. Various solutions are proposed to address these issues such as adding up to three levels of cache, which are close to the processing units and could operate at a significantly higher frequency rate than main memory, to benefit from spatial and temporal data reuse. Although these kind of modifications have improved former architectures, they fall short to meet the current requirements. There are still several problems which improved architectures cannot address, like insufficient data locality, high latency, limited bandwidth, etc. Hence, the desire for a substantially different architecture to address the aforementioned issues is serious. A possible solution could be an architecture for emerging post CMOS devices, called "memristors." Memristors have attracted computer architects' attention because of promising properties such as non-volatility, dense fabrication, low power consumption, etc.

Despite the fact that, these devices are in their infancy, there are different approaches to make the best use of them. For instance, some use it to implement Boolean logic, while some others use it as an engine to perform vector-matrix multiplication. In both cases, nonetheless, there is a need for a new computer architecture, so that they could operate in harmony with other parts of a system. To develop such an architecture several steps should be taken. For example, since the available memristor-chips are limited in functionality and flexibility, one should emulate (simulate) such a chip to be able to investigate the challenges that arise, and address them. It is crystal clear that, to operate the architecture, an instruction set is required through which the emulator (simulator) knows what it should do and what it should deliver.

In this document, first in chapter 2 we introduce nano-architecture, the overall CIM-tile organization and analogue peripheries. Then, in chapter 3 we explain what the micro-architecture looks like. To do so, we explain PULP and CGRA which will be the bases of the micro-architecture. In chapter 3 the instruction set in different levels (nano and micro) would be described, and then the way through which micro-instructions would be translated to nano-instructions will be explained. Lastly, in chapter five the emulator and simulator which are required to mimic the functionality of CIM-tile will be introduced.

2. Nano Architecture

The nano-instruction set (architecture) (nano-ISA) of a CIM tile directly relates to the control of the CIM tile functionality as well as the functionality of the immediate periphery, which is needed for correct operation. In Section 2.1, we will present the overall organization of the CIM tile and its immediate periphery, and provide a high-level introduction of how the CIM tile (and peripheral circuits) can be controlled. In Section 2.2, more details regarding the analogue periphery will be given and how these can be controlled. In Section 2.3, we present a complete overview of the assumptions regarding the CIM tile and periphery.

2.1 Overall CIM-tile Organization

A CIM tile, comprising a memristor array and peripheral devices, is a versatile device that can both *store data* and *operate on the stored data*. On one hand, it operates like regular memory into which we can **write** data and from which we can **read** data. On the other hand, as the CIM tile comprises an array of memristive devices, **logical bit-wise operations** as well as **vector-matrix multiply operations** can be performed within the memristor array. A conceptual high-level view of the array and its peripheral devices is depicted in **Error! Reference source not found.**.

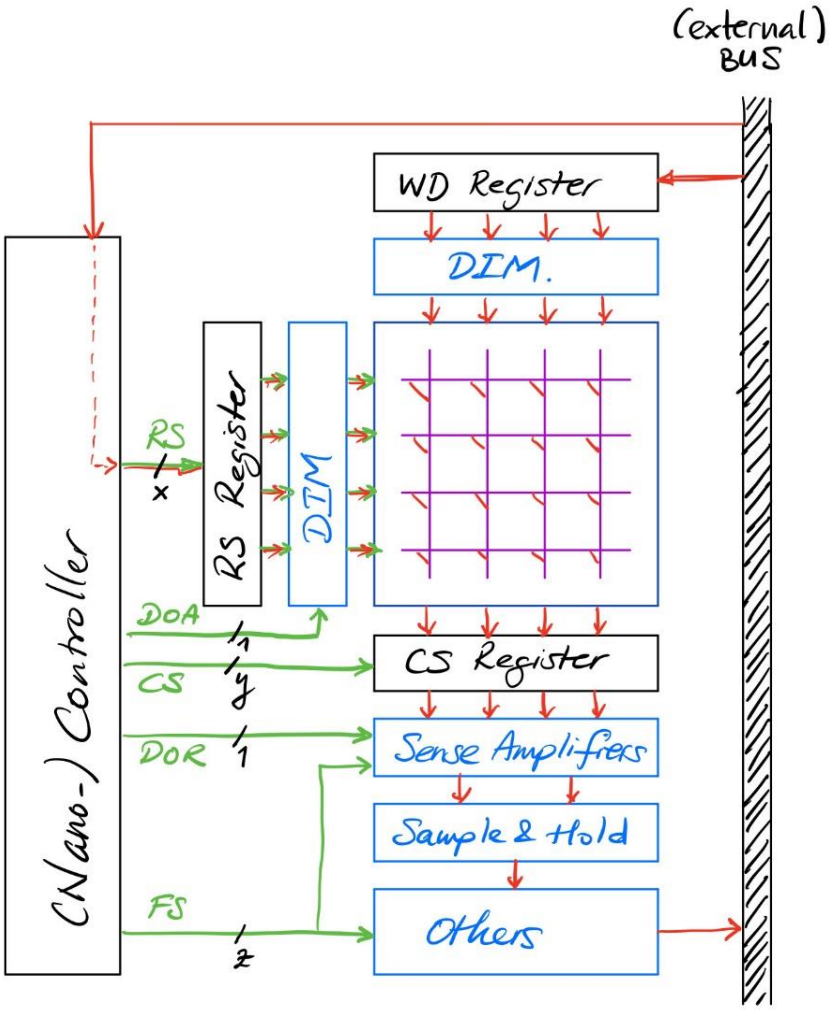

Figure 2-1 Overall CIM tile organization

The peripheral devices include the following:

- Set of digital input modulars (DIMs): the DIMs are used to drive the lines of the memristor array. In case multi-level input is required, a digital-to-analog converter (DAC) or Pulse Width Modulator (PWM) can be used.
- Sense amplifiers: these are used to read out values from the memristor array.
- Sample & hold circuits: these are used to hold the value read from the memristor array.
- Others: in this block, additional functionality can be added, e.g., analog-to-digital converters (ADCs) for Sample-and-hold circuitry or shifters, depending on the 'higher-level' functionality required.

A more detailed description of the peripheral circuits is presented in Section 2.2. For now, only a conceptual understanding is needed to introduce the control signals that are needed for the presented CIM tile. The control of the peripheral devices is performed via the following digital components:

- WD register: this register contains the data that need to be stored in the CIM tile. The content of this register is assumed to be set using the **write data (WD)** signal.
- RS register: this register determines which rows of the array should be activated. It is being set using the **row select (RS)** signals.
- CS register: this register determines which columns of the array should be activated. It is being set using the **column select (CS)** signals.
- Controller: the controller generates the necessary control signal for all the previously introduced peripheral devices, i.e., the **function select (FS)** signals, as well as the RS and CS signals. Furthermore, two activation signals, the **"do array" (DOA)** and the **"do read" (DOR)** signals, are generated as well. The latter two signals activate the necessary operations.

The operation of the CIM tile can divided into two phases:

- Configuration phase: in this phase the precise functionality is set up via the previously mentioned **WD, RS, CS,** and **FS** signals.
- Execution phase: in this phase, the operation (described below) is initiated via the **DOA** and the **DOR** signals.

We envision that using these signals we can accurately describe the high-level functions proposed by the partners in this project. Consequently, we defined these signals as part of our nano-instruction set. The formal definition of the nano-instruction set architecture (nano-ISA) is presented in Section 4.2. Moreover, the separation of any function into these two phases allows for flexibility in setting some of the "WD/RS/CS/FS select"-signals only once and iterate between the phases. More detailed code examples are given in Section 4.3.

We have to note here that the (analogue) circuit details – see Section 2.2 – have been skipped for the sake of simplicity in this introduction. For example, the sharing of sense amplifiers between different columns of the memristor array. However, the nano-ISA is defined to also be able to cope with such details.

Having described the potential of the nano-ISA in controlling a single memristor array, we will illustrate in the following how the same nano-ISA can be used to control larger memristor arrays when they are constructed using smaller arrays. The construction of a larger memristor array can be achieved in the manner depicted in Figure 2-2.

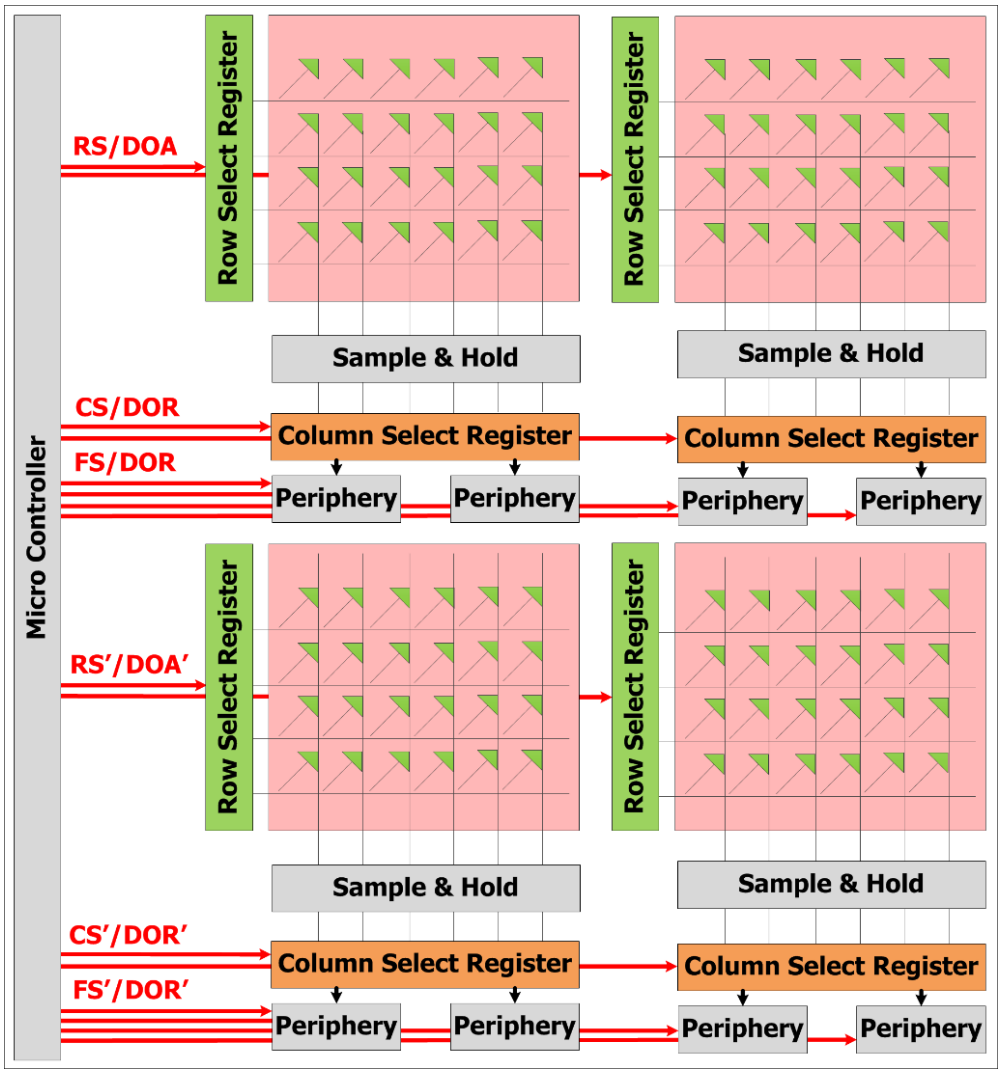

Figure 2-2 Multi-array tile organization

The length of WD, RS, and CS signals are longer (in bits) to cope with the additional rows and columns. However, the peripheral devices are strongly tied to their specific (smaller) memristor array and all operations should remain within the immediate confines of that particular array. Therefore, the DOA, DOR, and FS signals are a combination of the shorter signals controlling the distinct arrays. It must be noted here that operations cannot not cross the boundaries of the memristor arrays. Having a big array like that for more complicated operations, a more complex peripheral device (Periphery) is needed (depicted in Figure 2-3).

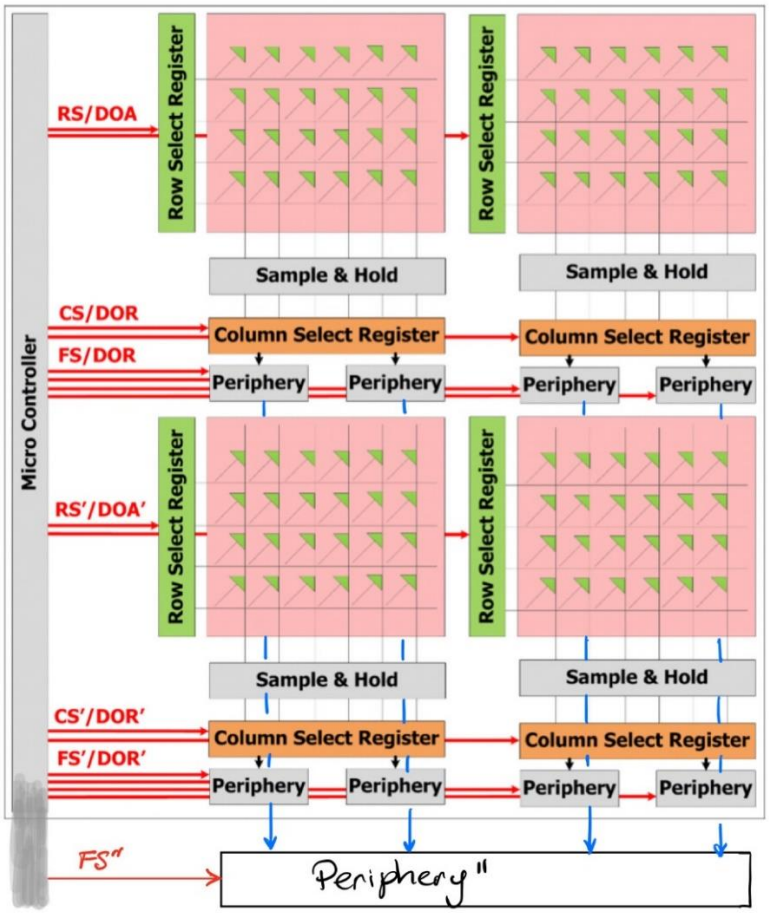

Figure 2-3 Multi-array tile with extra periphery

2.2 Analogue Periphery

The memristor-based CIM tile requires analog circuitry in the periphery to execute logic and arithmetic operations. Figure 2-4 depicts the required analog circuitry to implement all the aforementioned operations within the same CIM tile.

Each circuit, depicted as a blue box in Figure 2-4, is described as following:

Write DIM (Wr DIM):

Thid circuit is required to activate the voltages for the write operation. The write operation can be performed for the whole row in two steps: initialization and writing. During the initialization, the drivers ensure initializing the relevant devices (e.g., the whole row) to the highest resistance state (or lowest resistance state). After that, each device is programmed by applying the necessary voltage to its corresponding column (bit line). The word line of the row being programmed is connected to high voltage, while it is connected to a low voltage (i.e., GND) for the unselected rows. Moreover, the source line of the selected row is connected to a fixed voltage (e.g., GND).

If a write and verify scheme is implemented (in hardware), then the write operation can be performed only to one device at a time.

Write DIMs must be disconnected (high impedance) during reading and computing.

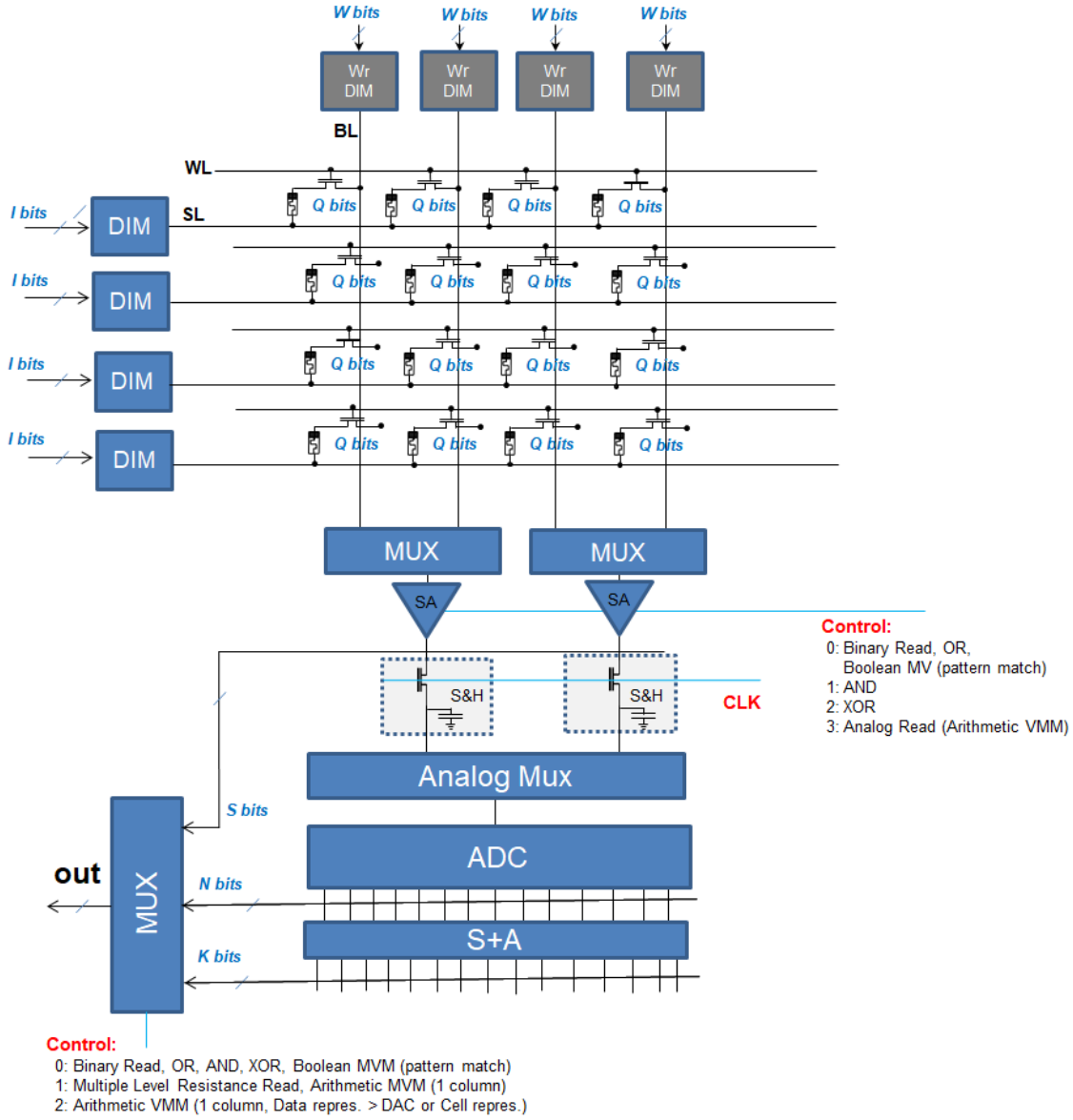

Figure 2-4 Analogue circuitry

Digital Input Modulator (DIM) or Read Driver (RD):

DIM converts a finite precision digital number (i.e., fixed point number) to an analog signal. This conversion is required for reading and computing. DIM can be of two types:

Digital to Analog Convertor (DAC): converts the digital bits to a voltage level.

Pulse Width Modulator (PWM): converts the digital bits to a pulse width.

The length of the converted digital number depends on the size of the DIM. In fact, a 1-bit DIM (driver) can be used with the shift and add (S+A) circuit as will be described later.

Memory Cell:

Each memory cell in the memristor array is composed of one transistor and one resistive device (1T1R). Each 1T1R cell has three lines: source line (SL), word line (WL) and bit line (BL). During reading and computing, input signals are applied to SLs and the output is captured from the BL. In addition, WL of each row can be connected to the SL in the same row.

The number of bits stored in each cell depends on the number of resistance levels that can be represented in the memristive device.

Sense Amplifier (SA):

SA senses the current or voltage value on BL, and converts it to a voltage output. The SA depicted in Figure 2-4 is a current SA that has low input impedance to fix the voltage on BL. Therefore, it mitigates the sneak paths during reading and computing.

The SA supports multiple functionalities indicated by a control signal. This control signal selects one of the following:

1. The reference current needed to execute binary read, scouting OR and Boolean vector-matrix multiplication.
2. The reference current needed to execute Scouting AND.
3. The reference current needed to execute Scouting XOR.
4. The mode of operating as an analog amplifier. The output in this mode is analog, and required to execute the arithmetic vector-matrix multiplication.

Sample and Hold (S&H):

S&H is used as a temporary storage to allow pipelining and sharing the analog to digital convertor (ADC). The ADC operates at higher frequency than the memory array. This allows multiple ADC conversions within the same time required by the memory array. If voltage SA is used instead of current SA, the S&H can be placed before the SAs.

Analog to Digital Convertor (ADC):

ADC is required to convert the analog signal (received from SA) to a digital value. The number of ADC bits depend on:

1. Number of bits per DIM.
2. Number of bits per memory cell.
3. The maximum number of rows activated simultaneously.

Shift and Add (S+A):

S+A is required when the data representation (number of bits) exceeds the representation of DIM or memory cells. Note that it is not used in the examples in Section 4.3.

For example, assume performing arithmetic matrix-vector with 1-bit drivers (1-bit DIM), while the length of each vector element is 2 bits. This can be implemented in two arithmetic matrix-vector multiplication operations. In the first operation, the lowest significant bit of each vector element is applied to its corresponding driver. Subsequently, the result of this operation is stored in the S+A registers. In the second operation, the highest significant bit of each vector element is applied to its corresponding driver. The result of this operation is shifted to the left by one in S+A, and then added to the results of the previous operation.

The number of registers in S+A circuit indicates the maximum length of the output vector, and hence the maximum length of operands supported by the CIM tile.

Analog Multiplexer:

The analog multiplexer is used to share the analog circuits in the periphery. The inputs and outputs of these multiplexers are analog signals. The following multiplexers are required in the periphery:

1. BLs multiplexer: required as the sense amplifier does not fit the pitch of a single memory cell. Therefore, multiple BL share a single SA.
2. SA multiplexers: required as the number of ADC exceeds SA area (or S&H area).

Digital Multiplexer:

The digital multiplexer is required as the output of some operations have different locations. The inputs and outputs of this multiplexer are digital. This digital multiplexer selects one of the following outputs:

1. Digital output from SA:

   The operations captured at this location include: binary read, scouting OR, scouting AND, scouting XOR, and Boolean vector-matrix multiplicationn. The number of bits S (depicted in Figure 2-4) equals the number of SA.

2. Digital output from ADC:

   The operation captured at this location is the arithmetic matrix-vector multiplication, with data representation not exceeding the bits of DIM or memory cell. The number of bits N (depicted in Figure 2-4) equals to the ADC bits.

3. Digital output from S+A:

   The operation captured at this location is arithmetic vector-matrix multiplication, with data representation exceeding the bits of DIM or memory cell. The number of bits K (depicted in Figure 2-4) equals to the number of S+A output bits.

2.3 Assumptions

In sections 2.1 and 2.2, we presented the general overview of a CIM tile, comprising a memristor array and its peripheral devices. In section 2.3.1, we will introduce several parameters (with their typical values) that represent the current state-of-the-art in designing the memristor array and its surrounding periphery. The same parameters will be used in the formal definition of the nano-instructions in section 4.2. Moreover, we will make several assumptions to clearly define the functionality within the CIM tile that will be supported by our nano-ISA. In section 2.3.2, we will highlight several possible future directions (expressed as future optimizations).

2.3.1 Parameters and Assumptions

Parameters

Based on technological possibilities and restrictions, we have defined the following set of parameters and their typical values:

Table 2-1 Parameters and their typical values

Parameter	Meaning	Typical Values	Notes (if any):
Nr	Number of rows (single array)	256	
NcS	Number of columns sharing one sense amplifier	8-16	
NSa	Number of sense amplifiers	16-32	
Nc	Number of columns (single array)	256	Nc = NcS*NSa. We assume that all the SAs are shared by the same number of columns to simplify the controls.
Nm	Number of bits stored in a memristor	1,2,4	A memristor device has 2^{Nm} Resistance levels

As the nano-ISA (section 4.2) is parameterized, we can easily support other designs as the technology progresses resulting in future changes of the typical values.

Assumptions

In the following, we will describe several assumptions that were made in order to not complicate the (initial) definition of the nano-ISA. However, this is not to say that these assumptions cannot be loosened in our future work. The assumptions made so far are:

- Transpose operations within the memristor array (as proposed by IBM) are not supported (yet). The support of the transpose operation will be added in the future.
- Instruction-level parallelism is possible, but not further discussed in this document.
- The data types used by the micro instructions is the same as the nano-instructions. Therefore, no conversion is needed between the two architectural layers.
- Sense amplifier are shared between columns. Consequently, we have to consider the interleaving of data as we envision potential streaming of data, i.e., it would be advantageous to access array elements from the beginning first, and so on. A more detailed discussion on this topic can be found below.

Data interleaving

Due to the sharing of sense amplifiers between columns of the memristor array, it is not possible to access all the elements within a line in a single "cycle". As a consequence, multiple cycles are needed to access the data. We will highlight this behaviour using Figure 2-5 and Figure 2-6

a_0	a_1	a_2	a_3	a_4	a_5
a_6	a_7	a_8	a_9	a_{10}	a_{11}
...	...	...	...	...	...

Figure 2-5 Memory space of Micro-instructions (continues and independent from the number of SAs).

a_0	a_3	a_1	a_4	a_2	a_5
a_6	a_9	a_7	a_{10}	a_8	a_{11}
...	...	...	...	...	...
SA0		SA1		SA2	

Figure 2-6 Memory space of Nano-instructions (interleaved based on the number of SAs).

Assuming a straightforward linear storage (depicted in Figure 2-5), it would result in accessing a0, a2, and a4 in one cycle followed by accessing a1, a3, and a5 in the ensuing cycle. When streaming data, this pattern is not ideal as it would at least require two cycles to access and rearrange the data – the number of cycles will grow if more columns share a single sense amplifier. Therefore, interleaving (as depicted in Figure 2-6) will overcome this issue. In this case, in the first cycle the data elements a0, a1, and a2 are accessed and can be immediately streamed.

While it is beneficial (as illustrated in the previous example) to have interleaving, it does pose a challenge. The challenge does not lie in the technical implementation of this functionality in hardware and it can be completely hidden from the programmer (as is the case in current-day DRAM designs). The challenge lies in effectively integrating this concept into the code examples to make them easily understandable for the reader. For his purpose, we introduced a new notation to describe the long WD, RS, and CS vectors (see section 4.3).

Finally, data interleaving (even though transparent to the programmer) can potentially create the following scenario that the programmer should be aware of. This scenario is highlighted in Figure 2-7.

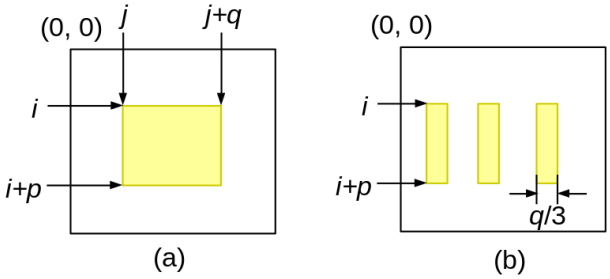

Figure 2-7 (a) a matrix in the Micro-instruction memory space. It starts from (i, j) and has the size of (p, q). (b) The same matrix mapped to the Nano-instruction memory space assuming the array has three SAs. The mapping result is still within row i and i+p. However, in the horizontal dimension, it is divided into three parts.

2.3.2 Future Optimizations

In section 2.3.1, many assumptions were made to not overcomplicate the initial definition of the nano-ISA. It is not intended that these assumptions remain in place in the future. Actually, they represent opportunities for further extensions and optimizations of the nano-ISA. In this section, we point out several possibilities for future optimizations that were left out for now.

Nano-instruction parallelism. Multiple Micro-instructions can be executed in parallel in the same tile. There are several scenarios:

- Instructions with a same input and the same instruction type: It is possible. For example, the same vector can be applied to two matrices (M1 and M2) in the same array and calculate vector-matrix product in parallel as shown in the figure below. This scenario is similar to SIMD.

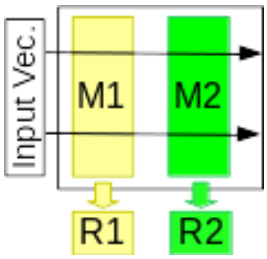

Figure 2-8 Instructions with a same input and the same instruction type

- Instruction with the same input, but different instruction types: It is possible for some cases. For bit-wise logic operations, the inputs are specified by the row register, and the function type is implemented by configuring the sense amplifiers. Therefore, different functions can be assigned to the memristors in the same array. The figure below illustrates a case that AND and OR operations are conducted in parallel. In this case, a row is split into two short vectors, e.g., V1 and V2. However, some instructions (e.g. OR and vector-matrix multiplication) cannot be executed in parallel as they use different ways to specify the inputs.

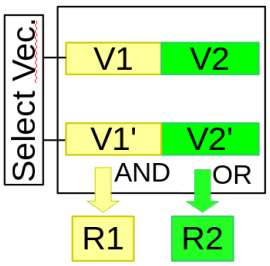

Figure 2-9 Instruction with the same input, but different instruction types

- Instruction with different inputs and instruction types: It is not possible because of conflicts.
- In the configuration phase, when setting up the WD, RS and CS signals, it is possible to actually perform these operations in parallel.

Transpose. We can support transpose operations in the future. To do so, one extra bit should be added to function register to specify the input/output directions.

Data type. Wider data type than the memristor can be supported in the instructions. In this case, multiple devices correspond to one data member, and extra peripheral circuit is required to combine the result of these devices.

Sample and hold. If the operation time of the memristor array is equal or less than periphery, maybe we do no need the sample and hold circuitry.

3. Micro Architecture

What we refer to as micro-architecture describes how we are planning to embed memristor chip which has analogue and digital functionality in contemporary system. Rest of this chapter is organised as followed. In 3.1 one can find how the new functional unit could operate as a new part in harmony with other units and what the big picture of system looks like. In 3.2 we will explain how we are going to consider it as a new functional unit in a coarse grain reconfigurable architecture, whose specifications will be presented as well.

3.1 PULP Cluster

The parallel ultra-low power (PULP) platform provides an efficient communication fabric to integrate CIM accelerators into a heterogeneous multi-cluster system of processing elements (PEs). These PEs consist of general-purpose computing units and application specific hardware accelerators sharing a common interface. The combination of both kinds of processing units gives the possibility to split complex algorithms into parts that map well to the CIM ISA while offloading other parts to general purpose (RISC-V) cores.

3.1.1 Architecture Outline

Each PE in a cluster shares access to the tightly coupled data memory (TCDM), a software-managed multi-banked scratchpad memory. The logarithmic interconnect (LI) provides both an n-to-n connection between memory banks and PEs, and arbitrates in the case of bank conflicts in a combinational fashion. Each PE may contain one or several master ports according to its bandwidth requirements. The LI thus allows high-bandwidth and low latency (single cycle) access to shared memory for intra-cluster communication.

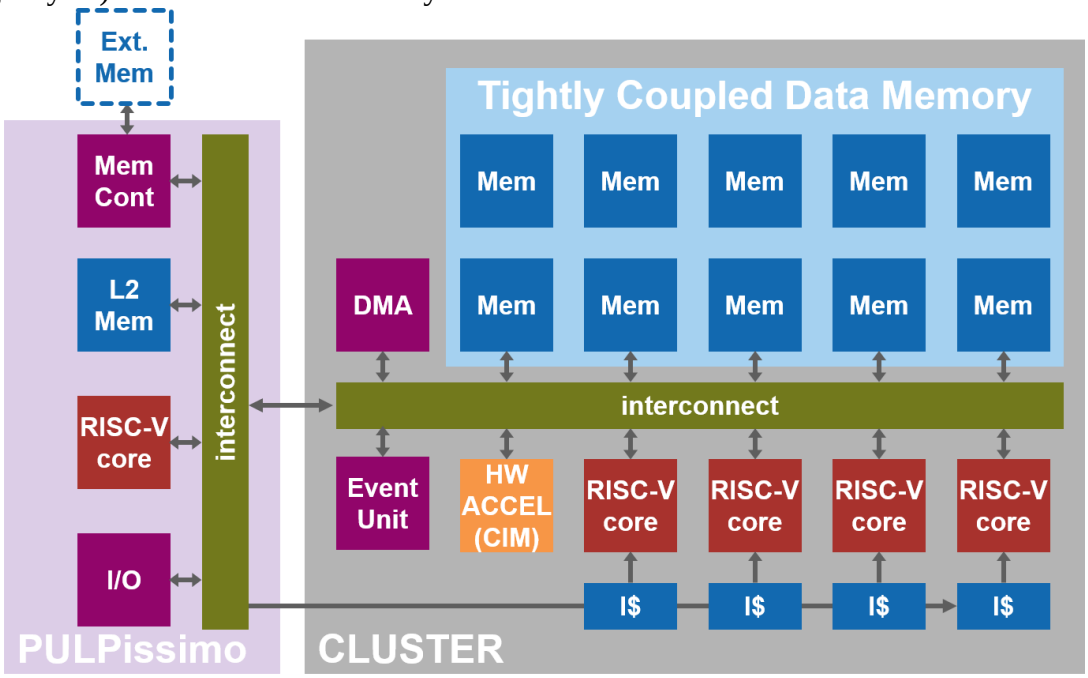

Figure 3-1 The parallel ultra-low power (PULP) platform

For configuration and synchronization purposes all the PEs are also connected to the independent so-called peripheral interconnect based on APB[1] protocol. This secondary interface makes it possible to prepare the CIM accelerator for autonomous operation by writing

1 Advanced Peripheral Bus

to memory-mapped configuration registers that store the desired memory access pattern and CIM instructions to be executed. Upon start or termination of the offloaded task the CIM accelerator may issue events using the event unit to synchronize with the general purpose PEs for post processing or preparation of the next task.
In order to incorporate many CIM accelerators the system can be scaled from a single- to a multi-cluster architecture. For this purpose each cluster is equipped with an optimized direct memory access (DMA) for high-bandwidth inter-cluster communication and access to larger shared L2 memory.
The benefits of the outlined architecture are the tight coupling of CIM accelerators with conventional processor cores to handle complex algorithms in an efficient manner and scalability to high-performance multi-cluster systems incorporating many CIM accelerators.

3.2 CIM Tile Integration

The architecture that we will use for this research is a research architecture from the Electronic Systems group in TUE. It is called CGRA which stands for Coarse Grained Reconfigurable Architecture. This architecture differs from 'normal' CPUs by its ability to reconfigure the processor data-path. This allows design a different processor for every application that one plan to run on it. When one start one's application his or her processor design will be instantiated on the chip and execution of their application will start. Figure 3-2 shows what the CGRA architecture looks like.

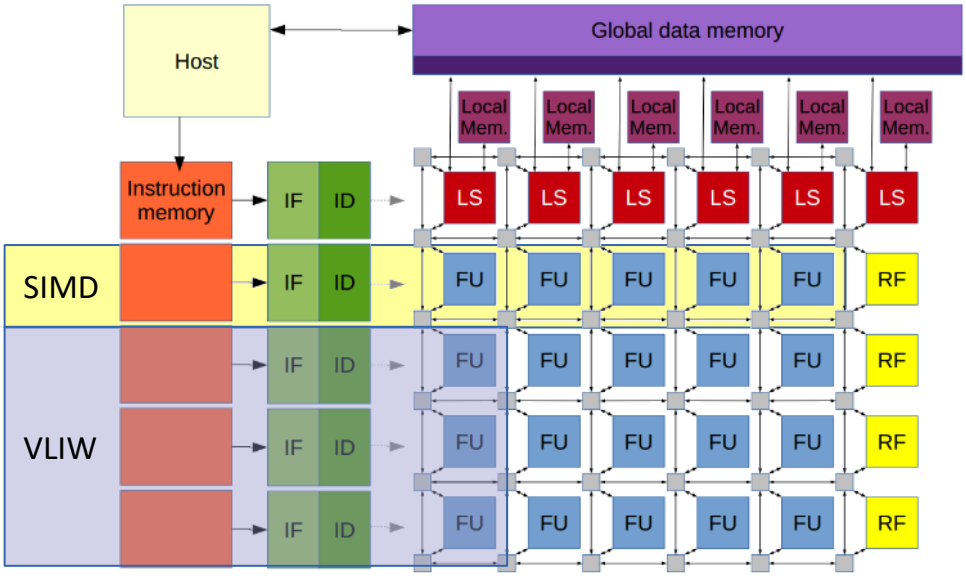

Figure 3-2 Coarse grain reconfigurable architecture (CGRA)

As one can see in the figure there is a grid of functional units (FU) that are connected to small grey boxes. These boxes are called switch-boxes and are used to route data from a specified input to a specified output to make connections between FUs. The routing is static for an application to reduce power consumption. There are several types of FUs:

- **ABU**: This is the Accumulate and Branch Unit. Depending on its configuration it can be used as a multiple-register accumulator (very useful for filtering applications) or as the unit that generates the program counter (PC) and performs branch operations.

- **IMM**: The immediate unit can be used to generate values on the data network, these values typically are constants used in the application.
- **ALU**: These perform typical operations such as addition, subtraction, shifting and comparison.
- **MUL**: These are multiplier units
- **RF**: These are register files and contain 16 registers in which you can store variables. Register file operations in the CGRA are explicit.
- **LSU**: The Load Store Units are used for reading and writing data from/to the memories. Each LSU has its own local scratch-pad type memory and a connection to the global memory that all LSUs can access.

Connecting functional elements via switch-boxes is quite similar to how a FPGA works. The main difference with the CGRA is the much bigger building blocks; a FPGA typically uses building blocks of 1 (or sometimes a few) bits that implement simple logical operations (and, or, sometimes a 1-bit full-adder) while the CGRA uses much larger bit-widths that perform more complex operations. This reduces the amount of configuration memory-bits required, which is good for energy.

These are also Instruction Decode (ID) and Instruction Fetch (IF) units. These units can be connected to one or more FUs and control their operation, this allows us to execute programs on the CGRA (which is another difference to FPGAs). By connecting multiple FUs to the same ID a vector processor can be constructed, since all FUs will now perform the same operation (DLP). If multiple IDs are used to control the FUs a VLIW style processor is constructed (ILP). Any mix between ILP and DLP can be made in order to allow for efficient application mapping.

One to make his or her CGRA instance, should change the architecture description. This is an XML file containing a description of which FUs are present and how they are connected. The functional units in the current CGRA support up to 4 inputs and up to 2 outputs. The CGRA toolchain can produce a .dot file of the architecture description which visualizes the block and the connections in between. (Figure 3-3)

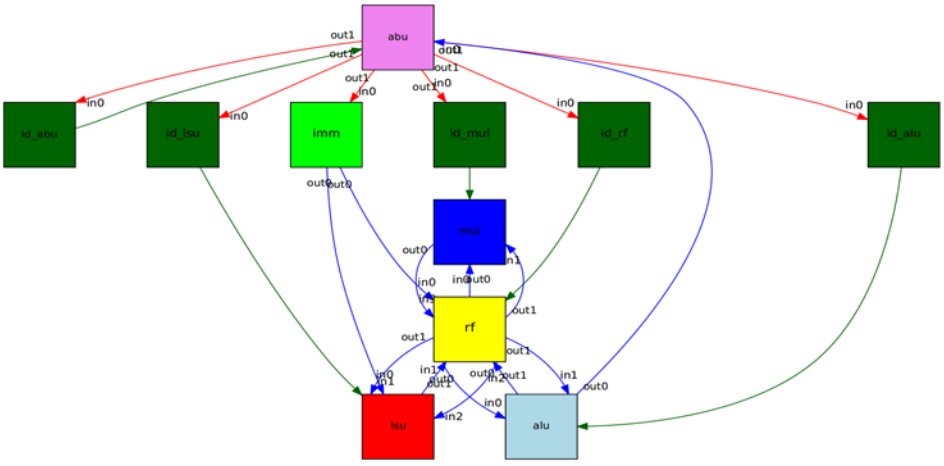

Figure 3-3 Picture of a resulted .dot file for an application

Just specifying the architecture description that one wants to use is of course not sufficient, some code has to be written as well. Since the compiler for this architecture is in very early

stages of development programming this architecture has to be done in our own assembly dialect, which we call PASM (Parallel Assembly).

It should be noted that the emulator, which would be described in details in chapter 4, could be used both in PULP and in CGRA.

4. Instruction Set

In section 4.1 and 4.2 we formally introduce the micro- and nano-instruction set, respectively. In section 4.3, we present several examples how the micro-instructions can be translated into nano (-instruction) programs.

4.1 Micro-ISA

These instructions are developed to represent for to cover applications that would be run on the memristor-chip including all required arguments.

4.1.1 Defining Variables

Note: Variables contained in the non-volatile memristor array have a **_nv** postfix, boolean variables have a **b_** prefix, **a** stands for array, **t** is a tensor, **v** is a vector.

```
int i, j, k, p, q, r, e;

nv int4 v_nv[], a_nv[][], t_nv[][][];

int4 v[], a[][], t[][][];

nv bool b_a_nv[][];

bool b_v[], b_a[][];
```

4.1.2 Store and Read

Note: Depending on the physical structure of the memristor array, we will implement one of the -1d, -2d, or -3d cases for store and read. The others will be emulated through that native instruction. For example, a 2d memristor array will only do 2d operations and 1d operations are then converted into (p,1) array operations. For 2d and 3d operations, the input data provided of size **p*q** or **p*q*r** needs to fit within the input vector buffer. This is to be ensured by the programmer.

- *storing **a** of size **p, q** to **a_nv** starting from **i, j***

  ```
  store_into_nv (a_nv, i, j, a, p, q);
  ```

- *read a from j, i to j+q, i+p*

  ```
  read1d_from_nv (&v, v_nv, i, p);
  ```

- *read **a** from **i, j** to **i+p, j+q***

  ```
  read_from_nv (&a, a_nv, i, j, p)
  ```

4.1.3 VMM (Vector Matrix Multiply)

Note: The matrix-matrix operation can be used to compute the matrix-vector result by setting e=1. Only when we can find a more efficient way to implement the matrix-vector operation does it make sense to include that specific form separately.

- *multiply int4 vector **v** of size **(p, 1)** with a window of matrix **a** from **(i, j)** to **(i+p, j+q)***

  matvec_nv_v_int4 (&v, a_nv, i, j, p, q, v);

- *multiply int4 matrix **a** of size **(p, e)** with a window of matrix **a** from **(i, j)** to **(i+p, j+q)***

  matmat_nv_v_int4 (&a, a_nv, i, j, p, q, a, e);

4.1.4 Pattern Matching

Note: The matrix-matrix operation can be used to compute the matrix-vector result by setting e=1. Only when we can find a more efficient way to implement the matrix-vector operation does it make sense to include that specific form separately.

- *multiply binary vector **b_v** of size **(p, 1)** with a window of matrix **b_a_nv** from **(i, j)** to **(i+p, j+q)***

  matvec_nv_v_binary (&b_v, b_a_nv, i, j, p, q, b_v);

- *multiply binary vector **b_a** of size **(p, e)** with a window of matrix **b_a_nv** from **(i, j)** to **(i+p, j+q)***

  matmat_nv_v_binary (&b_a, b_a_nv, i, j, p, q, b_a, e);

4.1.5 Boolean Logic

Note: The boolean logic operations assume a 2d memristor array

- *and-ing elements **p..p+q** of row **i** and row **j** of matrix **b** and return a vector of **q** elements*

  and_nv_nv (&b_v, b_a_nv, i, j, p, q);

- *or-ing elements **p..p+q** of row **i** and row **j** of matrix **b** and return a vector of **q** elements*

  or_nv_nv (&b_v, b_a_nv, i, j, p, q);

- *xor-ing elements **p..p+q** of row **i** and row **j** of matrix **b** and return a vector of **q** elements*

  xor_nv_nv (&b_v, b_a_nv, i, j, p, q);

4.2 Nano-ISA

In this section, we will formally introduce the nano-instruction set that is needed to control the CIM tile introduced in section 2.1. As we are dealing with several possible CIM tile organizations, we have chosen certain parameters to characterize them – see Table 2-1 (section

2.3). These parameters are used in the definitions of the nano-instructions. Finally, we purposely opted at this stage to omit the definition of the opcodes as we expect more (specialized) instructions to be defined in the future and, therefore, not fix the opcode space yet.

Row Select (RS)

The RS instruction is intended to control which row of the memristor array should be active. Depending on whether a multi-level input or binary input is required, there are two instructions formats. The single-level/single-bit format is given as follows:

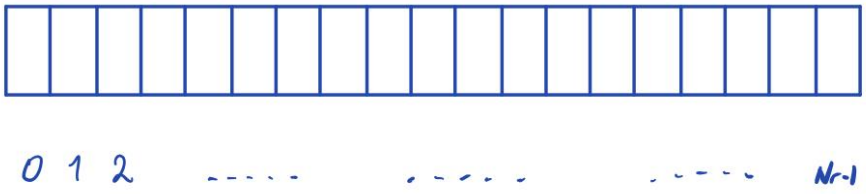

Figure 4-1 Single-bit RS instruction

Each bit corresponds to one row of the array and it is assumed that the enumeration of the rows starts at the top, being zero (0), downwards to row (Nr-1).

The multi-level/multi-bit format is given as follows:

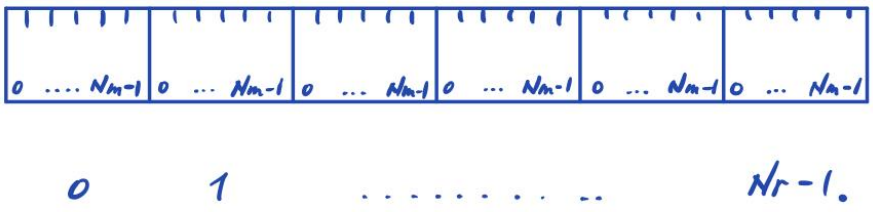

Figure 4-2 Multi-bit RS instruction

The 0-th set of bits correspond to row 0 (at the top of the array).

We have to note that a zero (in the single-bit format) and series of zeros (in the multi-bit format) will disable the corresponding row of the array. In the case that the multi-bit format is being used, it is expected that the hardware contains a DIM in front of the memristor array.

Column Select (CS)

The CS instruction is intended to control which column should be written to (from the write data (WD) buffer) or read out by the sense amplifiers or sample-and-hold circuitry. The format of the instruction is given as follows:

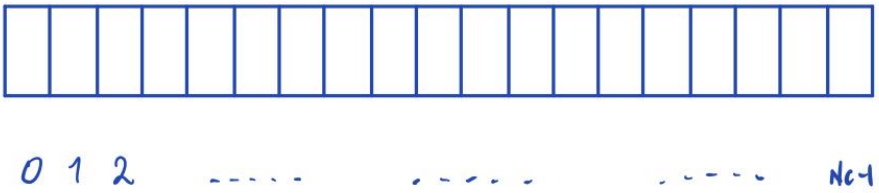

Figure 4-3 CS instruction

Write Data (WD)

The WD instruction simply contains the data that should be written to the write data (WD) buffer. In conjunction with the column select (CS) instruction, specific columns can be chosen to be written into. Given that the memristor array potentially can contain multi-level storage elements, there are two instruction formats. The single-bit format is given as follows:

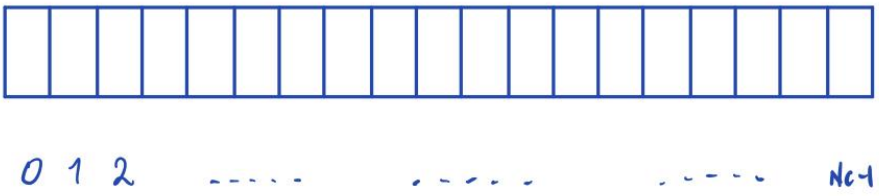

Figure 4-4 Single-bit WD instruction

The multi-level/multi-bit format is given as follows:

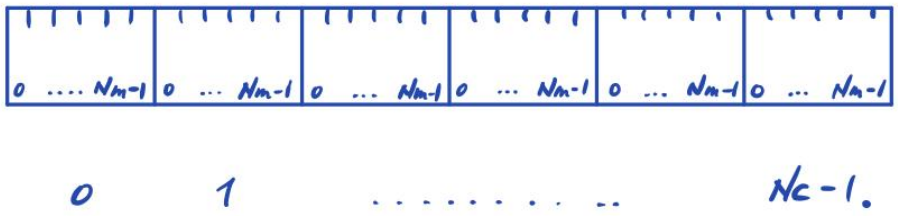

Figure 4-5 Multi-bit WD instruction

Function Select (FS)

The FS instruction is defined as a series of bit containers to control various peripheral devices, surrounding the memristor array. Moreover, it is expected that the functionalities of these devices will grow, which will consequently add more bits per container. As there is no fixed implementation, the number of containers nor the number of bits within the containers can be predefined. Therefore, we only present an example instruction format for the FS instruction (based on Figure 2-4 in section 2.2) as follows:

Figure 4-6 FS instruction

It is expected that the FS instruction can be fixed at a later stage of the project as more fixed designs will be presented and for each fixed design a fixed FS instruction format can be defined.

Do Array (DOA)

The DOA instruction controls when voltages should be applied to the memristor array. This instruction will be encoded in the opcode, but it does not have any operands.

Do Read (DOR)

The DOR instruction controls the operation of the sense amplifiers to start reading their inputs. This instruction will be encoded in the opcode, but it does not have any operands.

Possible optimizations

The lengths of some of the nano-instructions are quite long and it is expected that in future versions, the lengths of these instructions can be reduced. We have to note here that the initially proposed nano-instructions are purposely long in order to maintain full flexibility and avoid making any choices regarding the size of the array and the number of and organization of the peripheral devices. However, as mentioned before, the lengthy nano-instructions are issued in the configuration phase and it is not expected that these are frequently issued.

We envision the following optimizations that can reduce the length of the instructions:

- Certain operations only require several rows and, therefore, the encoding of these rows alone could reduce the instruction (format) length. A consequence could be that a reset instruction is needed to reset the row select register.
- The read-out circuitry will in many cases cycle through the columns connected to the sense amplifiers or sample-and-hold circuits. In these cases, it makes little sense to issue the full CS instructions in each cycle. We, therefore, expect to include shift instructions to perform these operations. Cycling through the rows (or inputs applied to the rows) is another expected operation that would benefit from a shift operation.
- By introducing a restriction on the starting index of data to be aligned with the sense amplifiers, we can also opt to access the columns with only an offset. Some of the code examples in section 4.3 demonstrate this possibility. Consequently, we can use *NcS* bits instead of *Nc* bits and with a binary encoding reduce it further to $\log_2 NcS$ bits.
- As mentioned before in the definition of the FS instruction, different choices for the periphery can enlarge or reduce the needed control bits.

4.3 Translation Examples (from micro to nano)

The utilization of the CIM tile and the nano-instructions are fully hidden from the user/programmer as they only issue micro-instructions to the CIM tile. At this stage of the project, we rely on pre-compiled (by hand, as no compiler exists yet) nano-codes to schedule all the (nano)-operations within the CIM tile. As the tools mature, the underlying nano-architecture can be made visible as it happened with RISC and VLIW machines in the past.

Before we can demonstrate how the previously defined micro-instructions (section 4.1) are translated to the nano-instructions, we need to define a new notation to describe the content of the long vectors used in the WD, RS, and CS signals. Since the vectors are expected to be

sparse, i.e., containing many zeroes and only a few non-zero elements, the notation is defined as follows:

- n represents the length of the vector
- m represents the set of values a vector element can take, e.g., [0,1] when m=2
- vector $\underline{v}_{n,m} = (i_x)$, with i ∈ [0, …, n-1] and x ∈ [0, … m-1]:
    - i denotes the location of the non-zero element
    - x denotes the value at location i
    - For example: $\underline{v}_{256,2} = (30_1, 40_1, 50_1)$ represents a vector of length 256 with non-zero elements at positions 30, 40, and 50. Moreover, the non-zero element is 1.
- A run of non-zero elements (if only ones) are represented as follows: $<20,30>_1$. This means that at position 20 through 30 (inclusive) all values are 1 and the remaining elements within the vectors are zeroes.
- A more generic notation is: $<20,30>_{\underline{\mathbf{v}}}$ with vector **v** representing the vector of non-zero values to be placed within the positions from 20 to 30. Vector **v** comprises of the following elements: (v0, v1, v2, ….) and should contain at least enough elements to "fill" all the positions. In case the vector is shorter, we can opt to pad with zeroes or leave the content unchanged. The choice is left as future work.
- A left/right shift is represented by "-s"/"+s", respectively, with s representing the number of positions to be shifted.
    - For example: $\underline{v}_{256,2} = (<30,40>_1+2) = (<32,42>_1)$
    - For example: $\underline{v}_{256,2} = (<30,40>_1, 50_1)-4 = (<26,36>_1, 46_1)$
- Vectors with a regular repetition of non-zero values can be represented as follows: $\underline{v}_{n,m} = (<i,i+p \mid \text{mod } x=y>_1)$, in which the locations for which hold "k mod x=y" where k ∈ [i, …, i+p] contain a '1'.
    - For example: $\underline{v}_{256,2} = (<24,40 \mid \text{mod } 8=0>_1) = (24_1, 32_1, 40_1)$
    - For example: $\underline{v}_{256,2} = (<24,40 \mid \text{mod } 8=1>_1) = (25_1, 33_1)$

In order to support interleaving, we can extend our "periodic" notation to include a range of values. In the following two examples, we assume that the content of vector **v** = (v0, v1, v2, …). By "looping" through the k-values, we selectively determine the locations to place the elements from vector **v**. Two examples are given below:

- $(\{k=[0...3]\}<20,31|\text{mod } 4=k>_{\underline{\mathbf{v}}}) = (20_{v0}, 21_{v3}, 22_{v6}, 23_{v9}, 24_{v1}, 25_{v4}, 26_{v7}, 27_{v10}, 28_{v2}, 29_{v5}, 30_{v8}, 31_{v11})$
- $(\{k=[0,1]\}<20,31|\text{mod } 4=k>_{\underline{\mathbf{v}}}) = (20_{v0}, 21_{v3}, 24_{v1}, 25_{v4}, 28_{v2}, 29_{v5})$

In addition, the interleaving requires a slight remapping in the enumeration of the array column indices (as perceived at the micro level compared to actual memristor array column indices). Without interleaving, the enumeration of the columns as perceived by the programmer and the enumeration of the memristor arrays is the exactly the same, as highlighted in the figure below. We assumed here: Nc=9.

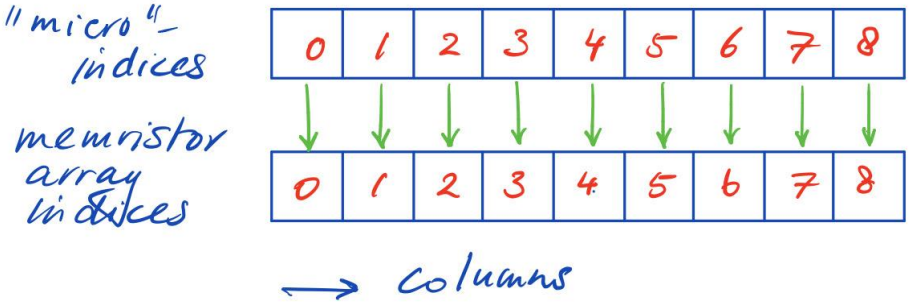

Figure 4-7 Memristor array indices mapping to micro-instruction indices

However, with interleaving the "consecutive" column indices assumed at the micro-level are actually remapped as follows: (with Nc=9, NcS=3, and NSa=3)

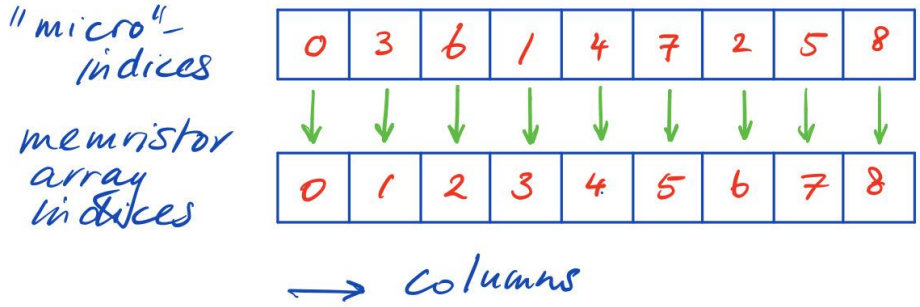

Figure 4-8 Memristor array indices mapping to nano-instruction indices

It should be clear that the remapping does not reduce nor add more memory locations as assumed at the micro-level. As explained in section 2.3, the remapping will allow for parallel accesses to columns (0, 3, 6) - which corresponds to micro-level columns (0,1,2) - in order to speed up the access to the data stored at those locations. We have to note that the remapping is fully transparent to the programmer. He/she is not aware of this. However, it is used in the examples (later on) in this section and a clear understanding of this remapping is needed to understand the code examples.

Finally, the nano-programs presented in this section are fully unrolled and do not require any form of sequencing. However, in order to shorten the code size in the presented examples, we introduce the pseudo-code of the utilized for-loop:

- for i from x to x+p

  <action>[i]

  end for

Unrolling of this loop results in the <action> being performed p times with its parameter i, where i=x, x+1, ..., x+p-1

It is expected that in the (near) future, we might need to introduce sequencing (nano-) instructions, but that is out of the scope for this deliverable and part of future work.

In the following, we will present translations of all the previously introduced micro-instructions (see section 4.1).

store_into_nv (a_nv, i, j, a, p, q)	
*storing **a** of size **p, q** to **a_nv** starting from **i, j***	
Non-interleaved code	Interleaved code
Pre-requisites: 1. $i + p <= N_r$ 2. $j + q <= N_c$	Pre-requisites: 1. $i + p <= N_r$ 2. $j + q <= N_c$ 3. $j = N_j * N_{Sa}$ 4. $q = N_q * N_{Sa}$ 5. N_j and N_q are integers
Code: for x from i to i+p **RS** (x_1) **WD** ($<j,j+q-1>_{\underline{ax}}$) **FS** "write" **CS** ($<j,j+q-1>_1$) **DOA** end for	Code: for x from i to i+p **RS** (x_1) **WD** ($\{k=[N_j..(N_j+N_q-1)]\}<0, N_c-1\|\text{mod } N_{cS}=k>_{\underline{ax}}$) **FS** "write" for y from N_j to $(N_j + N_q)$ **CS** ($<0, N_c-1\|\text{mod } N_{cS}=y>_1$) **DOA** end for end for

read_from_nv (&a, a_nv, i, j, p)	
*read **a** from **j, i** to **j+q, i+p***	
Non-interleaved code	Interleaved code
Pre-requisites: 1. $i + p <= N_r$ 2. $j + q <= N_c$	Pre-requisites: 1. $i + p <= N_r$ 2. $j + q <= N_c$ 3. $j = N_j * N_{Sa}$ 4. $q = N_q * N_{Sa}$ 5. N_j and N_q are integers
Code: for x from i to i+p **RS** (x_1) **FS** "read" **DOA** **CS** ($<j,j+q-1>_1$) **DOR** end for	Code: for x from i to i+p **RS** (x_1) **FS** "read" **DOA** for y from N_j to $(N_j + N_q)$ **CS** ($<0, N_c-1\|\text{mod } N_{cS}=y>_1$) **DOR** end for end for

matvec_nv_v (&v, a_nv, i, j, p, q, v)	
*multiply vector **v** of size **(p, 1)** with a window of matrix **a_nv** from **(i, j)** to **(i+p, j+q)***	
Non-interleaved code	Interleaved code
Pre-requisites: 1. i + p <= Nr 2. j + q <= Nc	Pre-requisites: 3. i + p <= Nr 4. j+ q <= Nc 5. *j = Nj * NSa* 6. *q = Nq * NSa* • *Nj* and *Nq* are integers

Code: **RS** (<i,i+p-1>$_v$) **FS** "matrix_multiply" **DOA** **CS** (<j,j+q-1>$_1$) **DOR**	Code: **RS** (<i,i+p-1>$_v$) **FS** "matrix_multiply" **DOA** for *x* from *Nj* to (*Nj + Np)* **CS** (<0, *Nc*-1|mod NcS=*x*>$_1$) **DOR** end for

matmat_nv_v (&a, a_nv, i, j, p, q, a, e)
smultiply int4 matrix a of size (p, e) with a window of matrix a from (i, j) to (i+p, j+q)
This micro-instruction can be performed by utilizing the "matvec_nv" instruction multiple times. Consequently, we can utilize the previously introduced nano-program multiple times. for *x* from *0* to *e* matvec_nv_v (&a[x], a_nv, i, j, p, q, a[x]) end for

and_nv_nv (&b_v, b_a_nv, i, j, p, q)	
*and-ing elements **p..p+q** of row **i** and row **j** of matrix **b** and return a vector of **q** elements*	
Non-interleaved code	Interleaved code
	Pre-requisites: 7. *p = Np * NSa* 8. *q = Nq * NSa* • *Np* and *Nq* are integers
Code: **RS** (i_1, j_1) **FS** "and" **DOA** **CS** (<p,p+q-1>$_1$) **DOR**	Code: **RS** (i_1, j_1) **FS** "and" **DOA** for *x* from *Np* to (*Np + Nq)* **CS** (<0, *Nc*-1|mod NcS=*x*>$_1$) **DOR** end for

or_nv_nv (&b_v, b_a_nv, i, j, p, q)	
*or-ing elements **p..p+q** of row **i** and row **j** of matrix **b** and return a vector of **q** elements*	
Non-interleaved code	Interleaved code
	Pre-requisites: 9. *p = Np * NSa* 10. *q = Nq * NSa* • *Np* and *Nq* are integers
Code: **RS** (i_1, j_1) **FS** "or" **DOA** **CS** (<p,p+q-1>$_1$) **DOR**	Code: **RS** (i_1, j_1) **FS** "or" **DOA** for *x* from *Np* to (*Np + Nq)* **CS** (<0, *Nc*-1|mod NcS=*x*>$_1$) **DOR** end for

xor_nv_nv (&b_v, b_a_nv, i, j, p, q)	
*xor-ing elements **p..p+q** of row **i** and row **j** of matrix **b** and return a vector of **q** elements*	
Non-interleaved code	Interleaved code
	Pre-requisites: 11. $p = Np * NSa$ 12. $q = Nq * NSa$ • Np and Nq are integers
Code: **RS** (i_1, j_1) **FS** "xor" **DOA** **CS** (<p,p+q-1>$_1$) **DOR**	Code: **RS** (i_1, j_1) **FS** "xor" **DOA** for x from Np to $(Np + Nq)$ **CS** (<0, Nc-1|mod NcS=x>$_1$) **DOR** end for

5. CIM Tile Emulator and Simulator

5.1 Emulator

5.1.1 Emulation Necessity

So far there is no commercial memristor-chip which one can buy and use it as a stand-alone system or as a part of other processing unit, like a co-processor. Even if such a chip is produced, as a result of its mixed signal nature, it surely needs a surrounding logic which can make connection between the chip and the system. To design the logic, define its functionality, explore its requirements, while the chip is under development, we definitely need an emulator (simulator).

Having an emulator and designing its surrounding logic, we would be able to test a system which includes a potential memristor chip. Doing so, we can get some good enough approximation of the possible outcomes when various kernels are being run on it. We believe the estimation of performance, energy consumption, and consumed area would be fairly close to what should be expected, since we will try to add as much as details possible during the process of designing the emulator.

Finally, if we have an emulation of the memristor chip, we would be able to explore what possible challenges, which the system may face and we have not taken them into account, could be. Consequently, we could do research on the ways through which possible issues could be addressed.

5.1.2 Emulator Properties

The first thing one should know about the emulator is that although the meristor chip has a both analogue and digital functionality, the emulator would be a digital implementation. It would be built on an FPGA which is purely digital.

The emulator should resemble the exact same functionality and support the same (micro- and nano-) instruction set. By resembling, we mean the external signal of the emulator and memristor must be identical. Meanwhile, if there is any internal states which would affect the external signals, they must be imitated exactly in the same fashion.

Additionally, it must meet the same timing constraints in which the real memristor chip would operate. There might be some differences in the operation time because of the different nature of the chip and the emulator; nonetheless, the result must be ready in the same time frames. As an instance, when a memrsitor chip of size 256 by 256 does vector-matrix multiplication it might approximately take 1 micro second. However, this vector-matrix multiply consist of 2^{16} multiplies and accumulation. This may consume all the available resources on an FPGA and yet it could not do that much operation in the same time frame. In such cases the most naive solution is to do the whole operation in a lock step which may take more than 1us.

It should be able to support all the data types on which the real memristor chip could operate. So far this includes binary and fixed-point 4 bit signals.

Finally, the whole emulator must be developed in a parametric fashion since due to the probable progresses in the memristor chip plenty of feature might change. As an instance, so far the available memristor crossbars usually are not bigger than 256 by 256. One of the challenges which prevents the researchers to go further and make the crossbar bigger is IR drop. In future this might be addressed and the crossbar dimensions go bigger. Therefore, the emulator should be able to keep up with such changes.

5.1.3 Early Draft of Possible Emulator

Considering all just explained and having the following figure, a big picture for what might a CIM tile look like, we need to design an emulator. Since we are going to make the emulator on CGRA platform and then use it in pulp architecture, we consider four inputs and two outputs for the CIM tile. As one can see there, input and output buffers at the start and end, respectively, which will be used for a couple of purposes. Since we have many inputs for memristor crossbar, e.g. 256, and having that many ports is not feasible due to the cost of routing, and also the constraints of CGRA, we need an input buffer to feed in data and store them on it. Same applies for output buffer as well. This makes sense, also because the memristor chip, as mentioned before, generates results in 1us, and the CGRA operates at 300MHz. Having this in mind, we could collect the data at a higher frequency rate and again deliver them through output buffer in the same fashion. Additionally, if one does the calculation based on the clock frequency of CGRA and memristor-chip, we could fill in input buffer (output buffer) as big as 4.8KB (2.4KB).

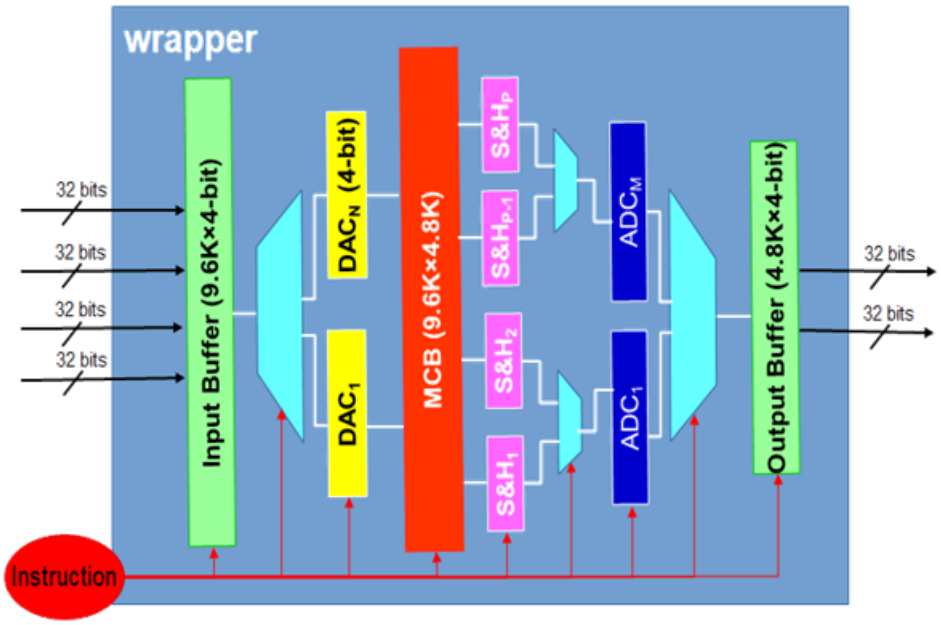

Figure 5-1 A block-diagram of a possible CIM tile

The simplest way to look at the emulator is to consider the memristor-chip as a calculator and the surrounding logic in a wrapper. (Figure 5-2)

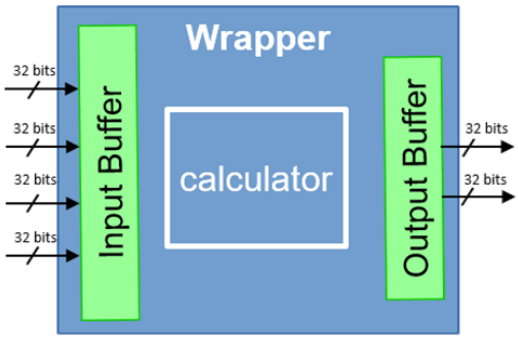

Figure 5-2 Simplest representation of possible emulator

We have tried to figure out how calculator should emulate the memrsitor-chip and what the surrounding logic should be. Firstly there must be some multiplexers to determine where the inputs should go. If inputs are memristor values, they should be guided to weight memory; if those are the values on which some operation must be done, they should be stored in input buffer to be fed to the calculator. There are some counters on upper-left corner which will calculate the addresses of the memristors which will take part in the calculation. For storing memristor values we have considered block memories which would preserve the values (weight memory in the figure.) We have assumed that each memristor crossbar could be configured to operate in different modes (as mentioned in section 2.1 and 2.2.) Consequently we have put decoders before the units which would be responsible for performing the operation. The possible operations, so far, are 4-bit vector-matrix multiply, 1-bit vector-matrix multiply, and Boolean logic operation. All these operations and their corresponding applications have been discussed in details in previous reports. For vector-matrix multiplies, since we could not do all the multiplication and accumulations in one shot, the 4(1)-bit MAC units are followed by accumulators. The produced result should go through an encoder to be stored on the output

buffer and be delivered to the system. Definitely, we would need finite state machines to orchestrate the whole tile.

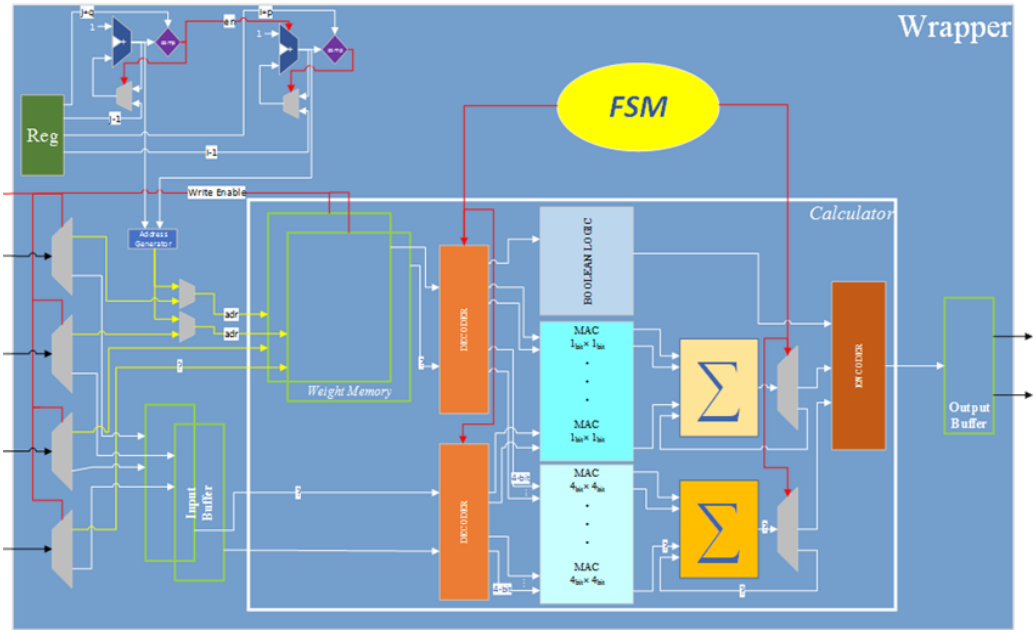

Figure 5-3 Possible design for the emulator

5.2 Simulator

With the newly defined nano-instructions, it is now possible start defining the contours of the simulator capable of functionally and cycle-wise simulate the CIM tile. Moreover, with the right input provided by the technology partners, it should also be possible to determine the energy consumption of the CIM tile and timing, i.e., the cycle time.

We can define the requirements of the simulator as follows:

1. Perform cycle-accurate simulation of the nano-instructions: Using the timing information, for the different operations within the CIM tile, provided by the technology partners, we can define the cycle time of the device. Subsequently, the number of cycles can be determined by executing the nano-programs, e.g., examples were given in section 4.3
2. Simulate the functionality of the CIM tile: the functionality of the memristor array, as well as the periphery, can be modelled and simulated. They perform the functionality as defined by the nano-instructions. After execution of the nano-programs, the correctness can be verified by looking at the output and content of the memristor array.
3. Allow for parameterization of the most common designs used in this project: The functional modelling (in bullet 2) can be parameterized based on actual designs specified by the MNEMOSENE partners. Within the simulator, these designs can be functionally verified as well as the corresponding nano-programs.
4. Capable of interfacing with higher-level simulators: The perception of the CIM tile as an accelerator is captured in the micro-instruction set. More precisely, the micro-instructions are translated into nano-programs. It is possible to utilize this interface as well to interface with, e.g., the CGRA simulator from TUE. In this manner. “higher-level” simulators can be connected to the envisioned CIM tile simulator.

5. Use output from low-level models to derive timing and power/energy consumption results: The accuracy of the simulator can be enhanced during the project when more designs are proposed and more accurate models (of the memristor array) are utilized.

The starting point of the simulator is the design depicted in Figure 2-1. A more detailed representation of the internal data structures and representations will be defined at a later stage in this project.