ISA and Hardware Design for a Pipelined CIM-tile

Remon F. J. van Duijnen

Technische Universiteit Delft

TU Delft

ISA and Hardware Design for a Pipelined CIM-tile

by

Remon F. J. van Duijnen

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on July 16th, 2020 at 13.00

Student number:	4489683	
Project duration:	November 11th, 2019 – June 1st, 2020	
Thesis committee:	Prof. dr. S. Hamdioui,	TU Delft, supervisor
	Dr. ir. S. Wong,	TU Delft, daily supervisor
	Dr. ir. R. van Leuken,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Preface

This thesis aims to progress work on a novel in-memory computation architecture by contributing towards the instruction set definition and hardware implementation of the architecture. Simulation results and a proof-of-concept hardware implementation have been used to investigate the power consumption, energy consumption and latency of the designed digital circuitry. These characteristics are found to not impose any road-blocks towards progressing the architecture to new levels in future works. This thesis is part of the overarching MNEMOSENE project of which, among many others, the Delft University of Technology is a parter.

The thesis committee for this project consists of prof. dr. Said Hamdioui, dr. ir. Stephan Wong and dr. ir. René van Leuken.

I would like to thank Said Hamdioui for allowing me the opportunity to work on this project, Stephan Wong for his guidance and Mahdi Zahedi for his contribution and many valuable discussions during our close cooperation throughout this project. Furthermore, I would like to express my gratitude towards my friends and family for their continuous support and helping me achieve my goals.

Remon F. J. van Duijnen
Delft, June 2020

Contents

List of Figures

List of Tables

1 Introduction

1.1. Introduction and motivation

Conventional Von Neumann machines inherently separate the processing units from the memory units. This architecture thus requires that data is transferred from the memory units to the processing units for performing computation, and results that should be stored are required to be transferred back to the memory units. Fast technological advances for processing speed have led to processors out-growing the speed at which data can be retrieved from the memory, meaning the bandwidth between the memory and the processor starts to bottleneck the system. This phenomenon is known as the Von Neumann bottleneck [1–3]. Modern computer architectures utilize for example hierarchical memory architectures and pre-fetching schemes to alleviate the Von Neumann bottleneck. However, these methods do not succeed in fully removing the bottleneck. Furthermore, the energy required to retrieve data from the memory is several orders of magnitude higher than the energy required for a single operation within the processor. There is a clear need for a new computing paradigm to further progress modern computer architectures.

One such paradigm is in-memory computing. In-memory computing combats the drawbacks of the Von Neumann architecture by bringing the processing and memory units as close as possible to each other or even combining them into a single unit. Research has shown that certain operations such as Boolean operations and the dot-product are especially susceptible to acceleration using specialized memory arrays, achieving great levels of parallelism and significantly reducing data-traffic and consequently energy consumption. The memory arrays exploit the characteristics of memristive technology to enable the in-memory computation. As Boolean operations and the dot-product can be efficiently supported, specific applications such as data-analytics, signal processing and machine learning can be accelerated using in-memory computing.

The research field is still in an early state and many research projects have been proposed in recent years. One of these projects is called MNEMOSENE [4]. The MNEMOSENE project has five main goals:

1. Develop new algorithmic solutions for targeted applications for CIM architecture.
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.
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.
4. Develop and demonstrate the nano-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.
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.

The thesis project specifically targets objectives 4 and 5, progressing the architecture design of the CiM-tile and demonstrating results obtained by simulation and emulation of the proposed architecture.

1.2. Problem statement

This thesis aims to provide an answer to the following problem:

> Is it possible to create a flexible CIM-tile architecture, capable of exploiting the benefits of in-memory computation to accelerate high-level kernels consisting of a variety of in-memory operations?

The flexibility which is strived towards is aimed at efficiently supporting the currently available technology, while also allowing for near-effortless scaling to support future technologies. According to the initial execution model and in-memory instructions proposed in prior works regarding this novel architecture, the effectiveness of their deployment is analyzed from a hardware perspective. Considering different implementation scenarios and their implications, this thesis reports aims to:

1. Propose a new version of in-memory instructions.
2. Propose and simulate a new organization of crossbar's peripheries.
3. Present an implementation of the tile architecture, written in synthesizable HDL to evaluate the functionality of our tile organization and obtain an estimation of its energy consumption, latency and area usage.

1.3. Methodology

To tackle this problem, a three step road-map is constructed.

- First, the system is designed and simulated on a higher level to be able to properly compare different architecture designs and implementations in terms of energy consumption and performance, keeping potential restrictions from the underlying hardware perspective in mind. As it is widely accepted that in-memory computing is well suited for performing dot-product operations (and thus on a larger scale, matrix-matrix multiplication), the performance of the tile will be benchmarked mainly on a program which executes matrix-matrix multiplication.
- Secondly, a detailed hardware design of the architecture is proposed based on the results obtained from simulation. As this thesis project aims to deliver the first fully functional version of the CIM-tile, and thus also serves as a proof of concept, the design is not targeting specific design corners such as energy efficiency or performance. Instead, different design options are proposed such that the design may be adapted to specific applications at a later stage of the MNEMOSENE project.
- Finally, a version of the architecture is implemented in synthesizable HDL and functionally tested on an FPGA. Numbers regarding power consumption, area usage and latency are obtained from synthesis and implementation results. Just as in simulation, the design is kept parametrized to allow for exploration of different configurations regarding e.g. the sizing of the memory present in the tile.

1.4. Thesis overview

Chapter 2 provides all the prerequisite knowledge on the in-memory computation. In Chapter 3, the prior work the newly proposed architecture is presented and the challenges that should still be overcome are discussed. Chapter 4 will describe the design solutions to these challenges and the results obtained using the simulator. The hardware design will be discussed in Chapter 5. The results of the prototype will be discussed in Chapter 6. Finally, the report will be concluded in Chapter 7.

2

Related Work

This chapter provides a general introduction to in-memory computation. Section 2.1 introduces memristive devices. Circuit-level use of these devices is described in Section 2.2. Section 2.3 describes several proposed in-memory architectures. Finally, Section 2.4 describes work on simulation of the in-memory computing architectures.

2.1. Memristive Devices

Memristive devices, first proposed in 1971 by Leon Chua [5], maintain a relationship between charge and flux elements of a two-terminal passive component. Together with resistors, capacitors and inductors, the memristor as the fourth basic circuit element completes the set of relations between four fundamental elements; voltage, current, charge and flux, as can be seen in Fig 2.1(a). The current-voltage characteristic of a memristive device is a pinched hysteresis loop, as can be seen in Fig 2.1(b). This means the memristive device can be in either of two stable states; the high resistive state(R_H) or the low-resistive state(R_L). Furthermore, the devices have the ability of storing their internal state, effectively remembering its history. In addition to its non-volatility, some large benefits of memristor technologies are their overall great scalability, high memory-density and low leakage power consumption [6–8]. By applying a sufficiently high positive voltage, the memristive device can be switched from its R_H state to its R_L state. Now, the device state represents a logic '1'. In the same way, applying a sufficiently large negative voltage will lead to the device switching from its R_L state to its R_H state, now representing a logic '0'. Using the memristive device in this way allows for binary data storage. However, if the memristive device can be programmed to more than 2 resistance levels, the device can be used to represent more states. In this way, the device can be used to store data consisting of more than one bit [6, 9].

Examples of such technologies are resistive random access memory (ReRAM) [11–15], phase-change-

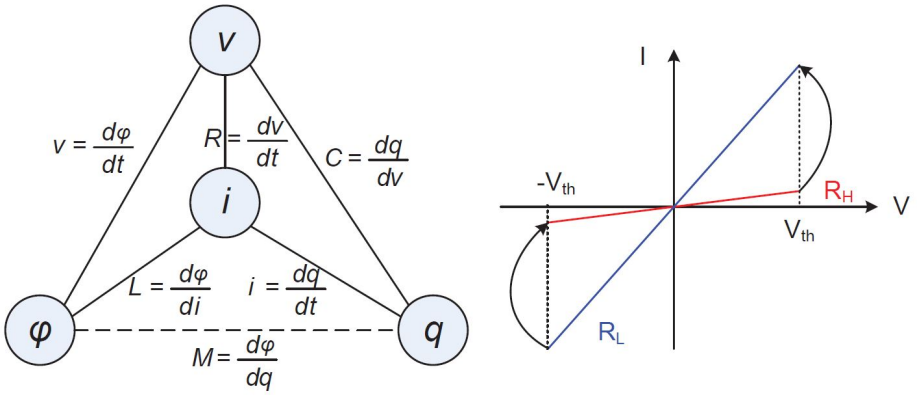


Figure 2.1: Stateful Logic, from Du Nguyen et al.[10]

memories (PCMs) [16–19] and spin-torque-transfer MRAM (STT-MRAM) [20, 21].

- **ReRAM:** The basic idea of ReRAM technology is switching between the low- and high resistance state by changing the conductivity of a normally insulating dielectric [11, 15]. By applying a sufficiently high voltage across the electrodes, a conduction path (also known as a filament) is formed. Because of the presence of this filament, the device is now in its low-resistance state. The filament can then be broken again, resulting in the device returning to its high resistance state. Different material can be used to implement this device. Examples are: HfO_2-based ReRAM [22], TaO_x-based ReRAM [23] and TiO_x-based ReRAM [24].
- **PCM:** This technology is based on changing the physical state of a chalcogenide material [25]. This material can be in either a crystalline or an amorphous state. The two states provide different characteristics in electrical conductivity. While in crystalline state, the devices resistance corresponds to the low resistance state, while the resistance corresponds to a high resistance state if the calcogenide material is in its amorphous state. While PCM has some beneficial properties such as multi-level cell operation possibility and a good switching endurance (order of 10^9), the main drawbacks of this technology are controlling the switching process and the switching speed [10]. PCM technology switches state about 10x slower than ReRAM technology due to the slow crystalline process.
- **STT-MRAM:** This type of non-volatile memory utilizes the angular momentum of charge carriers to change the orientation of of a magnetic layer. This is achieved by passing a current through a fixed magnetic layer, creating a spin-polarized current. This spin-polarized current is then passed through a free magnetic layer, which is stacked on top of the fixed layer, to force the free layer it to change its orientation. If both layers have the same orientation, it is relatively easy for electrons to pass through the stack, meaning the device is now in its LRS. When the layers have opposite orientations, the device is in its HRS. STT-MRAM is capable of very fast switching, but is not as energy efficient as either ReRAM or PCM [10].

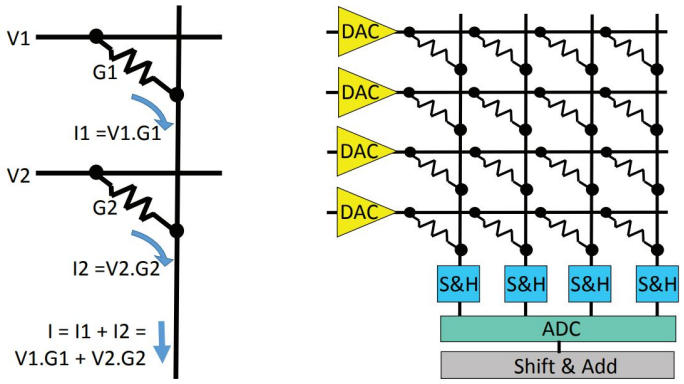
Whether the described technologies should be considered memristors is still open to discussion. There are several arguments in favor of calling these technologies memristors documented by Leon Chua [26]. Others dispute the terminology, arguing that the application of theoretical memristor characteristics to any physically realizable device is questionable [27]. However, the practical use for these technologies has been proven by using them in a multitude of proposed circuit designs and architectures, as will be discussed in Section 2.2 and Section 2.3.

2.2. Memristor-Based computation

The main focus on memristor-based circuits lies within the use of memristive crossbar array structures, allowing for effective computation of dot product operations [28–30]. By changing the resistance of each memristive cell within the crossbar array, these memristive memory arrays can exploit Kirchoff and Ohms laws to allow for in-memory computation. Figure 2.2(a) shows an example of a multiply-accumulate computation performed using memristors. Figure 2.2(b) shows a crossbar-array structure which can be used for these types of operations.

There are multiple metrics on which the memristor-based computation circuits can be classified. One of these metrics is the location of the results after computation. If this computation happens directly within the memory core, consisting of the memory array and peripheral circuits, two classifications can be distinguished; CIM-Array (CIM-A) class computation circuits perform the computation directly within the memory array, while CIM-Periphery (CIM-P) class computation circuits use the array only to store the data and perform the computation in peripheral circuit. Some advantages and disadvantages for both classes are given:

- **CIM-A:** As in this design class both the data storage and computation happen within the memory array, some significant considerations will have to be made for its design. Multiple values can be written at the same time, meaning that the array should be designed for higher currents than the standard write current. This results in e.g. wider data-lines to adequately accommodate these currents. Another disadvantage is the reduced life-time of the memristive devices as their state is changed more frequently as a result of the in-array computation with respect to only changing their state for storing data. However, a distinct advantage of this design class is its ability of using the entire computation-storage bandwidth as both storage and computation happen within the actual memory array.



(a) Multiply-Accumulate operation (b) Vector-Matrix Multiplier

Figure 2.2: Multiply accumulate operation (a), crossbar structure used as vector-matrix multiplier, from Shafiee et al. [30]

- **CIM-P:** The CIM-P class has some distinct advantages and disadvantages with regards to the CIM-A class. For example, as the CIM-P class performs its computation outside of the memory-array, this resulting data has to be fed back into the memory-array if cascaded computation is desired. However, using the memory-array only for data storage comes with some significant advantages. The controller can be simplified as no complex chain of states is required to allow for the different control voltages during computation. Instead, a single set of control voltages can be applied to the memory-array depending on its desired function (e.g. writing to the array, reading from the array etc.). In addition, as the maximum current that will occur in this design class is the write current for writing a single value on each bit-line, the memory array does not require design alteration to allow for the excessively high currents that occur in the CIM-A design class.

A multitude of logic design styles have been proposed using memristive devices [10]. Both the input- and output data can be represented by either a voltage or a resistance. Input and output representation can be combined to arrive at four different classifications, proposed by Du Nguyen et al. [10], voltage-voltage (VV), resistance-voltage(RV), resistance- resistance(RR) and voltage-resistance(VR). An overview of the logic design styles can be seen in Figure 2.3. While logic design styles for hybrid memory and computing using resistance as an output have not been explored yet, there are numerous design styles using voltage as output. Examples of logic design styles that belong to the VV class are Memristor Ratioed Logic [31], current mirror based threshold logic [32] and PLA-like [33]. Examples of RV classed logic design styles are Pinatubo [34] and Scouting Logic [35].

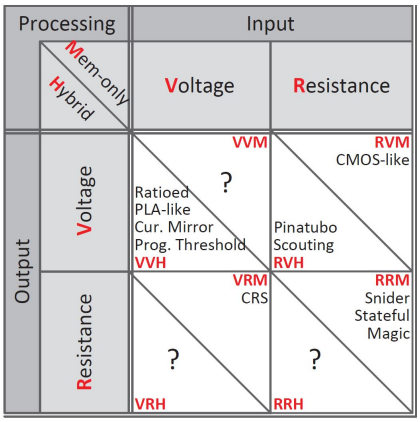


Figure 2.3: Overview of logic design styles, from Du Nguyen et al. [10]

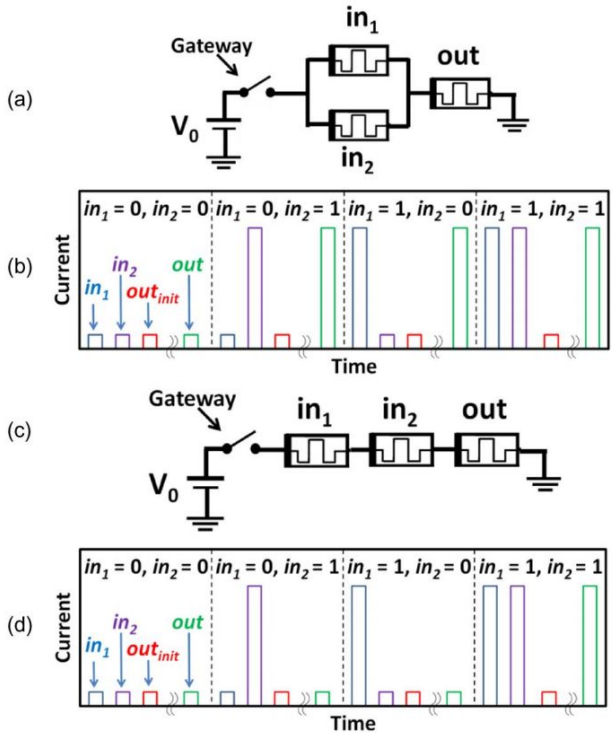


Figure 2.4: Logic gate designs using MAGIC, from Kvatinsky et al. [36]. (a) and (b) show the circuit and results of an OR gate, (c) and (d) show the circuit and results for an AND gate. NOR and NAND gates can be realized by switching the polarity of the output memristor.

- **MAGIC:** An example of CIM-A class circuit design is MAGIC [36]. MAGIC uses multiple memristors in series and parallel connections to create logic gates, storing both the input- and output values as resistances in separate memristors. To perform a logic operation, the output memristor value is set to a pre-defined value. Then, by applying a pulse to the circuit and depending on the input values stored in the input-memristors, this output value either remains at its initial value or is changed to a new value. Different circuit designs can realize different logic operations, as can be seen in Figure 2.4. These logic gates can also be integrated in a crossbar structure [37]. The MAGIC NOR gate satisfies the conditions regarding connectivity and state-representation and can thus be placed within a crossbar array without the necessity for additional components. The setup of the output memristor state is then performed by a regular write operation, and after applying the voltage pulse the output can be read using a regular read operation. In this way, logic computation can be performed directly within the crossbar.

- **Scouting logic:** Scouting logic is an example of CIM-P class circuit design. In Scouting Logic, the read operation is altered in such a way that logic operations are performed. The regular read operations would select the row that should be read by applying the proper read voltage, driving a current trough the resistive element which will be detected by a sense amplifier [35]. By selecting multiple rows at once, the total current that will be sent to the sense amplifier will be determined by the equivalent resistance of the memristors in these rows. This equivalent resistance is thus dependent on the state of the memristors, in other words on the data stored in the memristor. By defining adequate reference currents to the sense amplifier logic operations such as AND, OR and XOR can be performed [10, 35]. This can be seen in Figure 2.5. In a similar fashion, using Ohm's and Kirchoff's law the dot-product functionality can be obtained using memristive devices in a crossbar array [30, 38]. Multiple rows of the crossbar are selecting using the proper voltage, then the sum of the resulting currents through the memristive devices, obeying Kirchoffs's law, can be fed to an ADC. This method of computing the dot-product offers opportunities for very high levels of parallel computation [10], as computation within each of the crossbar columns can be performed in parallel. Computation using this design style is thus performed when reading the data from the memory-array and the results are available in the peripheral

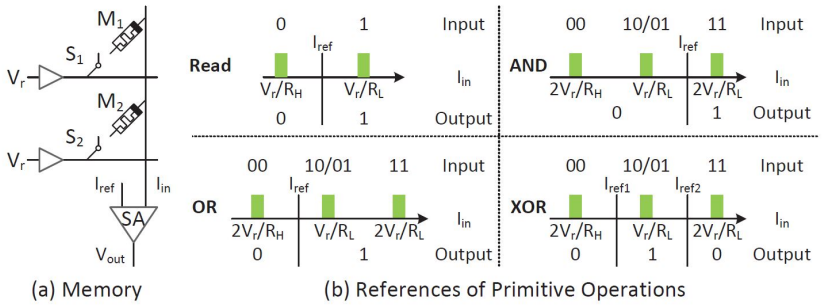


Figure 2.5: Scouting logic main idea, from Xie et al. [35]

circuits of the memory-core. This is thus an example of CIM-P class in-memory computation.

2.3. In-memory Architectures

Considering the CIM-A and CIM-P design classes, a multitude of architecture designs have been proposed [10, 29, 30, 34, 39–52]. This section discusses several architectures using ReRAM memory technology. ReVAMP [39] is an architecture design using the CIM-A computation class, while the PRIME [44] and ISAAC [30] are architectures using the CIM-P design class.

ReVAMP

ReRAM based **V**LIW **A**rchitecture for in-**M**emory com**P**uting (ReVAMP) is the first ReRAM-crossbar based general purpose computing architecture that exploits the crossbar array potential for parallel computing, proposed by Bhattacharjee et al. [39]. As the name suggests, Very Large Instruction Words (VLIW) are used to encapsulate multiple computations in a single instruction. The computation in the VLIW can then be executed in parallel.

The ReVAMP architecture design can be seen in Figure 2.6. It comprises of a traditional Instruction Memory (IM) which is accessed using a Program Counter (PC), an instruction Decoder (ID) and the crossbar array (referred to in bhattacharjee et al. as the Data and Computation Memory (DCM)). In addition, some peripheral circuits and registers surrounding the crossbar array and the IM and ID blocks are used to allow for correct operation and supporting pipelining. The architecture has a three-stage pipeline consisting of Instruction Fetch (IF), Instruction Decode (ID) and Execute (EX) stages. Instructions read from the IM are temporarily stored in the Instruction Register (IR) and made available to the ID that decodes the instruction to output the proper control signals for the Source Select multiplexer, crossbar interconnect and the write circuit. Depending on the control signal provided by the ID, the Source Select multiplexer then sends either the data present at the primary inputs or the data present at the output of the crossbar array to the crossbar interconnect, where the word-line is created. Finally, the write circuits combine the target wordline and the output of the crossbar interconnect to determine the inputs that should be applied to the row and column decoder of the memory array.

This architecture supports an ISA of two instructions. The first instruction is the *Read*-instruction. This instruction reads a word from the memory array into the Data Memory Register (DMR). The data present in this DMR is then available to the source select multiplexer and can thus be used for the next computation. The second supported instruction is the *Apply*-instruction. This instruction specifies is used for the actual computation within the memory-array. Proper benefit of the word-level parallelism is achieved by a technology mapping methodology that extracts the suitable parallelism and generates the VLIW instructions supported by the ReVAMP architecture accordingly.

Some of the advantages of the ReVAMP architecture include:

- Max bandwidth usage and low periphery design effort due to the CIM-A nature.
- Compact crossbar design using only a single device per cell.
- The ability of direct data transfer by copying the resistance values within the crossbar.

The ReVAMP architecture also suffers from some disadvantages. These include:

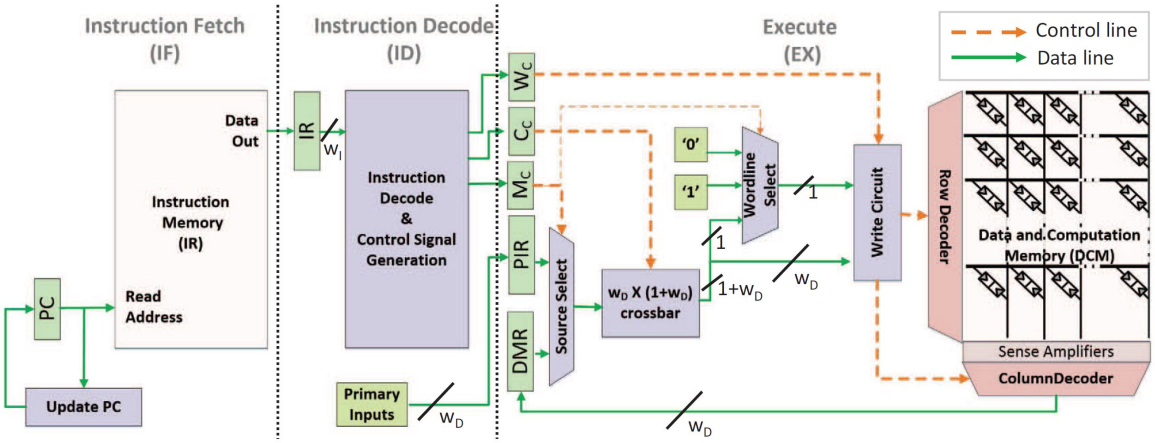


Figure 2.6: ReVAMP architecture, from Bhattacharjee et al. [39]

- High array and controller design effort
- High endurance requirement because of the regular switching when performing computation within the crossbar
- Imbalanced pipeline stages due to the highly complex EX stage.

PRIME

PRIME is a Processing-in-memory architecture for neural network computation in ReRAM-based main memory, proposed by Chi et al. [44]. In contrast to ReVAMP, PRIME is an example of a CIM-P class architecture, performing its computation at in the peripheral circuits at read-out rather than in the memory array itself. The Prime architecture is mainly focused on improving the performance and energy efficiency of NN applications. The implementation of a single input/output layer ANN using a ReRAM crossbar structure can be seen in Figure 2.7.

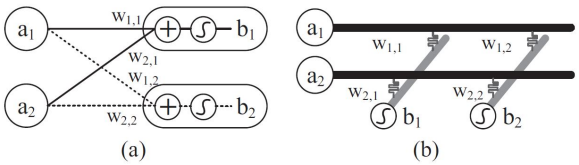


Figure 2.7: Implementation of an ANN with one input/output layer using a ReRAM crossbar, from Chi et al. [44]

The PRIME architecture sub-divides the ReRAM memory into three partitions; the memory-, full function- and buffer sub-arrays. The memory sub-arrays possess only the capability of data storage like conventional memory arrays. While the full function sub-arrays can also function in memory mode, they are also capable of performing computation in their computation mode. The buffer sub-arrays can either be used as conventional memory arrays or as buffers for the full function sub-arrays.

The overall PRIME architecture is subdivided in five parts, shown in Figure 2.8, and consists of a decoder and driver (A), column multiplexer (B), sense amplifier (C), buffer connection (D) and the PRIME controller (E). To allow for the NN computation, the decoders and drivers, column multiplexers and sense amplifier designs of the full function sub-arrays has been appended with some additional components, marked in light blue in Figure 2.8.

A multiplexer in the decoder and driver circuit allows for switching between the memory- and computation modes for the full function array. When computation mode is selected, data from a buffer-array is available and a larger number of input voltage levels becomes available to allow for the NN computation. To store positive and negative weights, two separate crossbar arrays are used. For each set of crossbar arrays, a column multiplexer circuit is appended with an analog subtraction unit and a by-passable sigmoid unit. The analog output is then sensed by sense amplifiers (SA). As NN computation requires higher sensing precision than data-storage, the sense amplifiers have been modified to have a configurable precision between 1 and 8

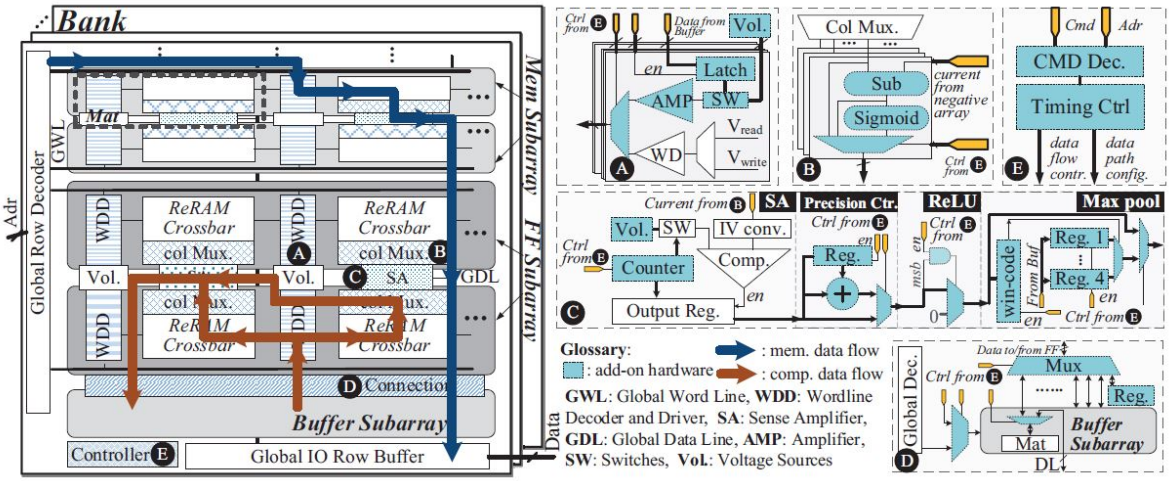


Figure 2.8: PRIME architecture, from Chi et al. [44]

bits. Additional circuitry is used to control the precision, allowing low-precision ReRAM cells to perform high accuracy computation. ReLU and max-pooling functions are also supported using extra circuitry. The connection between the full function- and buffer sub-arrays has been appended with extra buffer and decoders so the full function array to access any location of the buffer, allowing for the random access pattern of the NN computation present in the connection of two layers. The PRIME controller allows for correct operation and use of the extra circuitry by decoding the instructions and generating the appropriate control signals.

The PRIME architecture has some significant advantages such as small area overhead due to the re-use of ReRAM array circuits. However, the architecture solely targets NN applications and is thus not suited for general purpose acceleration of different applications. In addition, the PRIME architecture benefits and suffers from all the CIM-P class benefits and drawbacks, respectively.

ISAAC

Another architecture using CIM-P class approaches to CNN acceleration is ISAAC, proposed by Shafiee et al. [30]. This architecture focuses on the implementation of an inter- and intra-layer pipeline while also proposing solutions for challenges such as the handling of signed arithmetic and achieving high throughput.

The ISAAC chip consists of a set of tiles (T) connected using a mesh network. The tiles consist of in-situ multiply-accumulate (IMA) units used for the main computations of the CNN applications. Each of these IMAs consists of several crossbar arrays, ADCs, input- and output registers and shift-and-add (S+A) units. In addition to the IMAs, the tiles pack an eDRAM buffer for storing the input values and, similar to the PRIME architecture, sigmoid and max-pool units are also present. Finally, the tile packs a shift-and-add unit. The chip is connected to other ISAAC chips and external hardware through an I/O interface. The overall chip, tile and IMA architecture can be seen in Figure 2.9.

To obtain high throughput for the CNN application, ISAAC partitions its tiles and IMAs to store weights and perform computation for the different CNN layers. Figure 2.10 gives an example of a tile performing a 2x2 convolution computation on a 6x6 feature map. Output values of the previous layer are stored in buffers and when enough information is available, computation is started. Figure 2.10(A) shows the situation in which enough information is received to start the first 2x2 kernel. Figures 2.10(b) and (c) show the evolution of the computation. Once a set of values has been serviced, the output can be made available to the next layer.

In addition to this inter-tile pipelining scheme, an intra-IMA pipeline is also proposed to improve the individual IMA throughput. The ISAAC architecture focuses on reduces the ADC overhead by using multiple ADCs per IMA. The number of ADCs per IMA is chosen to enable matching the read throughput of the crossbar. Sample and hold circuits are deployed to latch the crossbar outputs, enabling the pipelining. The ISAAC architecture also allows for high accuracy computations using low precision ADCs by sequentially processing each level of bit significance. Here a trade-off between ADC precision and required cycles can be made. For example, a single 1-bit ADC can process a 16-bit input in 16 cycles, while a 2-bit ADC only requires 8 cycles. Another way of reducing the required cycles is by having different IMAs handle sub-sets of the input. The

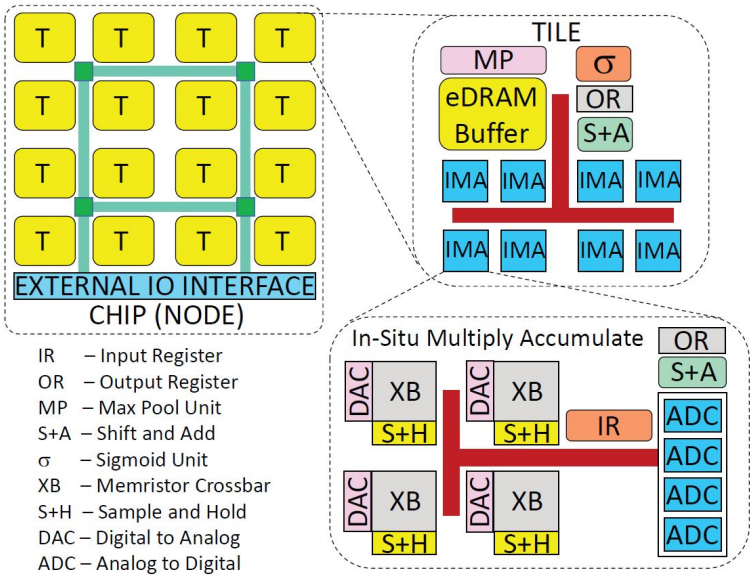


Figure 2.9: ISAAC architecture hierarchy, from Shafiee et al. [30]

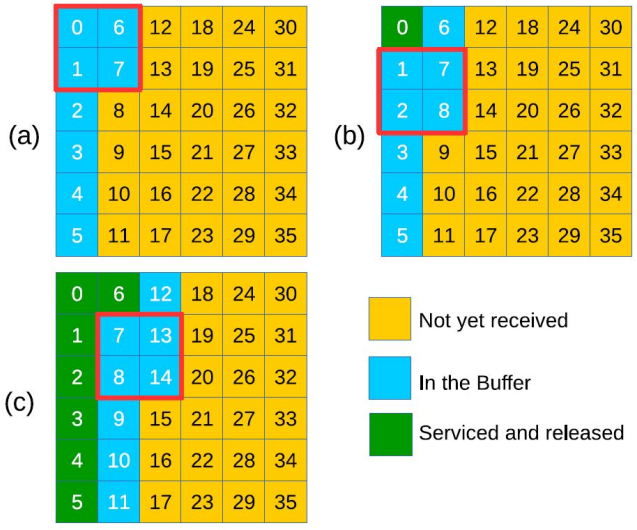


Figure 2.10: The ISAAC pipeline, from Shafiee et al. [30]

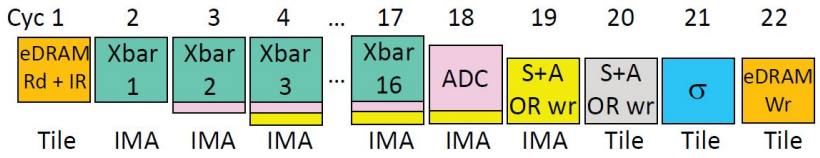


Figure 2.11: The ISAAC intra-tile pipeline, from Shafiee et al. [30]

output of the ADC is then fed to the shift-and-add units or written to the output register. An example of the intra-tile pipelining scheme for an operation in a CNN layer is shown in Figure 2.11. The parallelism of the crossbar, ADCs and shift-and-add circuits can be seen in cycles 3 through 18.

ISAAC is an efficient architecture for CNN applications. Like PRIME, CNN applications are its sole target application. A drawback of the ISAAC architecture is the fact that it is not suitable for in-field training of the network. In addition, the ISAAC architecture benefits and suffers from all the CIM-P class benefits and drawbacks, respectively.

2.4. Simulators

As the architecture design for these architectures consist of many factors influencing the system behaviour, simulators are used to be able to provide accurate results regarding performance and energy usage. A tool called CACTI [53] which is able to provide numbers regarding memory access time, energy and area of non-volatile memories has been around since 2001. Based on this software, some other memory-oriented simulators have been developed like NVSim [54] and NVmain [55]. NVSim provides a circuit-level performance, energy and area model targeting emerging non-volatile memories such as STT-RAM, PCRAM and ReRAM. This tool allows the user to tune circuit parameters such as the array organisation and peripheral circuitry to explore different design options. NVMain, on the other hand, provides a cycle-accurate memory simulator on architectural level. NVMain can be used in conjunction with results obtained from CACTI or NVSim. Two other simulators, MNSIM and a Neuromorphic chip simulator, will be described in more detail.

MNSIM

The first behaviour-level simulator for memristor-based computing, MNSIM, was introduced by Xia et al. [56] in 2018. MNSIM expands on the NVSim and NVMain simulators by allowing non-fixed structures in the peripheral circuits, thus supporting the simulation of computing structures. The architecture simulated by MNSIM can be seen in Figure 2.12. In addition to area, power and latency estimates, this simulation platform presents the user with numbers regarding the accuracy of memristor-based structures, achieving speedups of around 7000x compared to conventional simulation tools like SPICE [57]. This speedup is achieved by using a three step solution to approximate the computation accuracy. First, the non-linear V-I curves of the memristor are approximated by linear curves, simplifying calculations. Then, the interconnects parasitic capacitance and inductance is ignored, effectively modeling the crossbar as a resistance-only device. Finally, the worst-case and average case error rate are calculated. This approximation is thus able to quickly provide a number for the average error and an upper limit to the error. However, the approximation does lead to loss of accuracy with respect to lower-level simulation tools. Another limitation of MNSIM is the lack of simulation support for the memristors dynamic properties.

Neuromorphic chip simulator

Another cycle-accurate, system-level simulator was introduced by Lee et al. [58]. This simulator targets neuromorphic ReRam-based computing chips. The simulation architecture can be seen in Figure 2.13. This simulator framework simulates a neuromorphic chip consisting of many cores. Each core harbors a ReRAM crossbar structure used for computation. Effects due to non-idealities of the ReRAM crossbar, such as write variability and SAF errors [59], are also taken into account by this simulator. These non-idealities are categorized into either static or dynamic effects. Static defects are mainly present due to errors in the fabrication process and do not change over time, while the dynamic defects can change during operations and yield different results every cycle. The two dynamic effects considered in this simulator are random noise and write variability. The noise, resulting in random switching of memristors' states, is modeled by the Monte Carlo approach described by Ambrogio et al. [60]. Write variability is modeled using statistics and data obtained

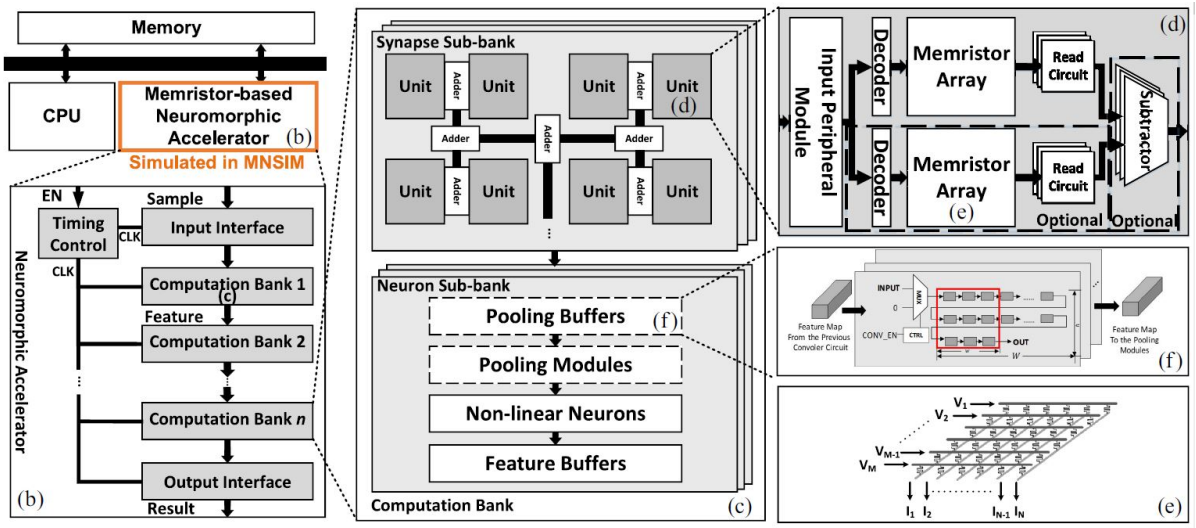


Figure 2.12: MNSIM simulated architecture, from Xia et al. [56]

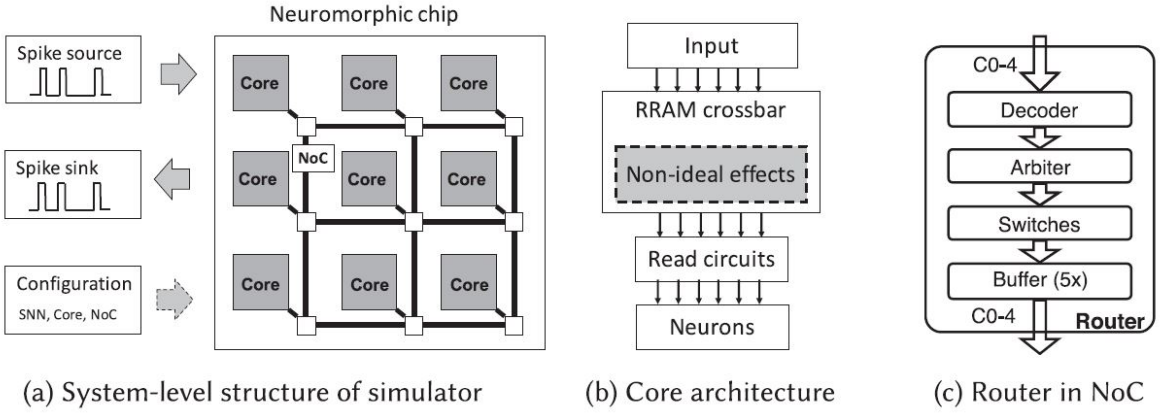


Figure 2.13: Neuromorphic chip simulator architecture, from Lee et al. [58]

from Ambrogio et al. [61]. To alleviate write variability issues, Lee et al. briefly touch upon the idea of having a write verification scheme, reducing the variability at the cost of latency for the write operation.

2.5. Conclusions

This chapter introduced the required prerequisites on in-memory computation. Related works on in-memory computing regarding device level, circuit level and architecture level have been discussed, describing the benefits and drawbacks of each proposed work. In addition, related works on simulation of in-memory computation are discussed.

3

Design proposal

This chapter describes the initial high-level architecture and simulator design. This initial design will be expanded upon during this thesis project to obtain a detailed design for the in-memory computing architecture. Section 3.1 presents the initial architecture design. Section 3.2 highlights the main challenges that should be tackled while aiming for an actual implementation of the architecture

3.1. Prior work

Recall goal 4 and 5 of the MENMOSENE project:

- Develop and demonstrate the nano-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.
- 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.

This section describes the prior works performed on these goals of the MNEMOSENE project. The proposed architecture will be used as a starting point to this thesis project. The overall architecture consist of three levels, dubbed the macro-, micro-, and nano-architectures. The Nano-architecture, also referred to as a CIM-tile, is the main building block of the overall architecture, containing the memristive crossbar and its required peripheral components. While designing this nano-architecture tile, it is assumed that data and instructions are sent to the tile over a data bus. Other interfaces such as a connection network between tiles are considered when designing the micro-architecture. The design proposed in this section was proposed by Zahedi et al. [62]. With the benefits and drawbacks regarding CIM-A and CIM-P endurance, performance and design complexity described in Chapter 2 kept in mind, a CIM-P class tile has been deemed the best candidate for this tile organization. Furthermore, the tile organization and instructions are kept as generic as possible to allow for building many operations using them, creating a general-purpose accelerator.

3.1.1. Nano-Architecture

The architecture of the CIM-tile, shown in Figure 3.1, is based around the memristor crossbar as its main component. Two main operations can be performed in the crossbar which are (i)write and (ii)read/computational. The computational operations that can be supported in the CIM-tile so far are Matrix-Matrix Multiplication (MMM) and vectorized logical AND, OR, and XOR operations. A short explanation of the objective of each component in our CIM-tile architecture concerning the supported operations is provided:

- **Write operation:** The Row Select (RS) and Write Data Select (WDS) registers are used to indicate the location where data has to be written to. In order to provide the actual data to the crossbar, an additional Write Data (WD) register is present in the tile architecture. When not all columns of the crossbar are supposed to be written to, the *WDS* register can be used to select only the desired columns. This also

saves energy and memristor lifetime. For naming consistency, the RS register will be referred to as the Row-Data-Select (RDS) register throughout this thesis report.

- **Read/computational operations:** Despite the write operation in which row selection is just used to indicate to which row the data has to be written, specifically for MMM, it will be employed to feed the actual data of the multiplier to the crossbar. The MMM in is executed by performing multiple dot-product operations based on the principles used by scouting logic, explained in Section 2.2. The results obtained for each dot-product operation are then accumulated to obtain the MMM result. The proposed addition scheme is presented in Section 3.1.5. More detailed information on how MMM performed in the crossbar is provided in [63]. Since the crossbar is an analog circuit, before communicating to this component with either the *RDS* or *WD* registers, the data should be converted to the analog domain using Digital Input Modulars (DIMs). After the crossbar execution is finished, the analog data is sampled and fed into Analog-to-Digital Converters (ADCs) to read the resulting data. Since an ADC is a costly component in terms of energy and area, not all columns have a dedicated ADC and the resulting data should thus be read in multiple sequential operations. To this end, an analog multiplexer is put between the sample&hold component and the ADCs. A Column Select (CS) signal will be used to select the data that should be read. The resulting data read by the ADCs can then be fed into peripheral circuits (e.g. shift&add) and be transferred to either the host or another CIM tile.

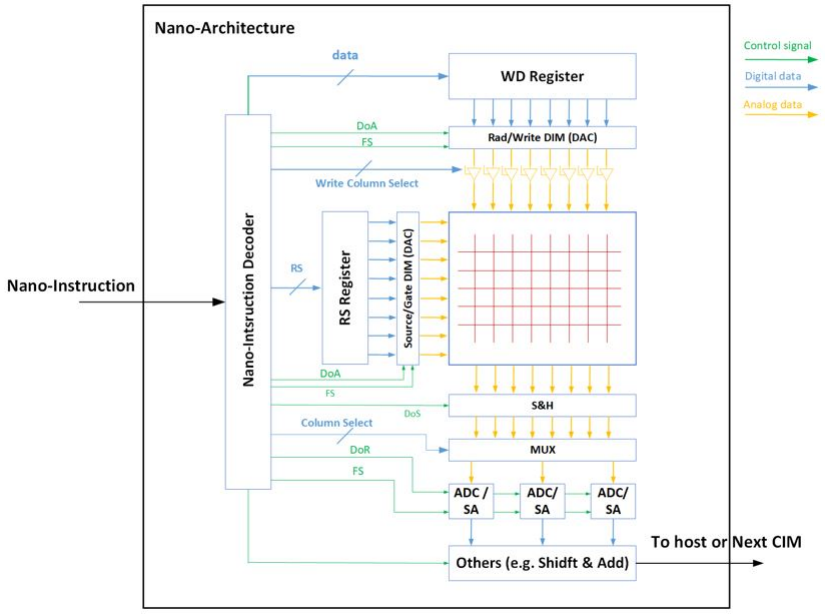


Figure 3.1: Initial nano-architecture

3.1.2. Nano-instructions

The instructions that will be executed within the CIM architecture are, similarly to the architecture design, split up in multiple levels. An application will first be compiled into a set of micro-instructions. Each micro-instruction will then, using a lower-level compiler, be compiled into a set of nano-instructions. These nano-instructions will be executed within the nano-architecture described in Section 3.1.1. An overview of the initial set of nano-instructions and their function is given in Table 3.1.

3.1.3. Compiler

High-level operations should be translated into sequences of nano-instructions which will be executed in the CIM-tile architecture. The compiler takes care of this translation. Translation is thus done at compile-time, allowing for optimizations to be done in the instructions that would not be possible at run-time. An example of the translation of a micro instruction into a set of nano-instructions is given in Appendix A. The compiler can also take into account limitations imposed by technology or the tile-design. For example, the precision of

Nano-instruction	Function
RDS	Fill the Row Data Select register
WD	Fill the Write Data register
WDS	Fill the Mask register
FS	Function selection
DoA	Activate the crossbar
DoS	Activate the sample and hold
CS	Fill the Column Select register
DoR	Activate the ADCs

Table 3.1: Initial set of nano-instructions

the ADCs will determine how many rows can be selected at once. If an application requires to activate more rows than what is supported by ADC resolution, the compiler can then split it into several sections in which the number of rows activated in one shot does not exceed the ADC resolution. The operations supported by the initial version of the compiler are listed in Table 3.2. Any change in tile architecture or configuration can be supported by the compiler resulting in less run-time controller design complexity.

Operation	Desription
Store	Store an MxN matrix at specified location in the crossbar
Read	Read an MxN matrix from specified location from the crossbar
MMM	Perform Matrix-Matrix multiplication between a matrix stored in the crossbar and an externally available matrix
AND	Perform a logical AND operation on a specified set of rows and columns
OR	Perform a logical OR operation on a specified set of rows and columns
XOR	Perform a logical XOR operation on a specified set of rows and columns

Table 3.2: Set of operations supported by the initial compiler

3.1.4. Pipelined architecture

To further improve the CIM-tile throughput, a four-stage pipeline architecture has been introduced. This pipeline divides the tile into the setup stage, the execute stage, the read stage, and the addition stage as can be seen in Figure 3.2. Each set of nano-instructions that correspond to a single complete operation flow through this pipeline is assigned to one of three local program counters, while a single global program counter keeps track of the overall set of instructions.

The purpose of the setup stage is to fill all of the registers surrounding the crossbar with the data required to execute the operations inside the crossbar. The execute stage consists of the actual computation within the crossbar and the sample&hold action required to make the output data available to the read stage. The read stage is responsible for reading the analog data using a set of ADC/SA. Finally, the addition stage performs post-digital processes on the raw crossbar outputs to make them meaningful when performing a matrix-matrix multiplication. Worth noting is that the setup stage and the addition stage are purely digital stages of the pipeline, while the execution stage and read stage both also handle analog data.

This pipelined architecture is fundamentally different from traditional pipeline architectures, where instructions travel through all of the pipeline stages and the latency of each stage should be matched to allow for a balanced pipeline. In the case of the CIM-tile, the instructions serve a purpose in only one of the pipeline stages. The latency of the analog crossbar is assumed to be significantly larger than that of the digital components in the other stages. However, the other three stages require a larger number of nano-instructions

to complete their purposes. This number of instructions combined with the clock frequency of the digital components will result in a latency that should be matched with the crossbar latency to achieve optimum throughput.

This pipeline structure imposes a number of challenges for the CIM-tile design. First, the delay of the analog components differs per function and is data dependent, so a way of knowing when the pipeline should proceed should be implemented. Secondly, in an ideal case the register stage and read stage should finish their functions before the crossbar stage is finished in order for those stages to not impose bottlenecking on the entire pipeline.

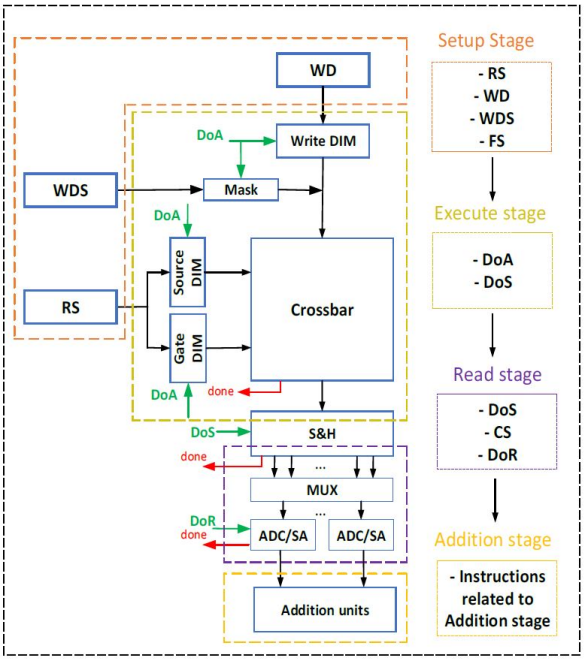


Figure 3.2: Initial pipeline architecture

3.1.5. Efficient addition scheme

In order to efficiently perform an operation like the Matrix-Matrix multiplication on the unsigned integer datatype using the CIM-tile, peripheral circuitry is required to perform addition on the results obtained by the crossbar. The crossbar itself performs the dot-product operation on the multiplier (fed into the crossbar through the *RDS* register), and the multiplicand (stored in the crossbar). Depending on the number of rows that can be selected, the result of this dot-product may be split into several sections. The goal of the peripheral circuit is to perform three types of addition. First, the different sections of the dot-product should be accumulated to acquire the final result of the dot-product operation. These dot-product results correspond to only a single bit-position of the multiplicand. Secondly, the different bit-position results should be added, keeping in mind the proper significance of each position. The result of this second stage is thus the partial result corresponding to a single bit-position of the multiplier. Finally, the results of all bit-positions of the multiplier should be added to arrive at the final result of a dot-product operation on the integer datatype.

Figure 3.3 illustrates the addition scheme proposed by Zahedi et al. [63]. The three addition stages described and the registers required for storing the intermediate results can be seen. The first stages saves the intermediate results of the single-bit position result of the multiplicand in the registers R0, R1 etc. The size of these registers is only dependent on the number of cells per column and is thus equal to $Log_2(crossbarheight)$. As the second stage performs addition on the entire set of bit-positions of the multiplicand, the size of this second intermediate result is $size(multiplicand) + Log_2(crossbarheight)$. However, because the result should be shifted by one position for each iteration of this addition, the least significant bit will never change anymore. By shifting this LSB to the $R2_{temp}$ register at every addition step, the size of the adder and the $R1_{temp}$ register can be kept at $Log_2(crossbarheight)$. The same holds for the third stage, where the adder and $R3_{temp}$ can be kept at size $size(multiplicand) + Log_2(crossbarheight)$, while the $R4_{temp}$ register size

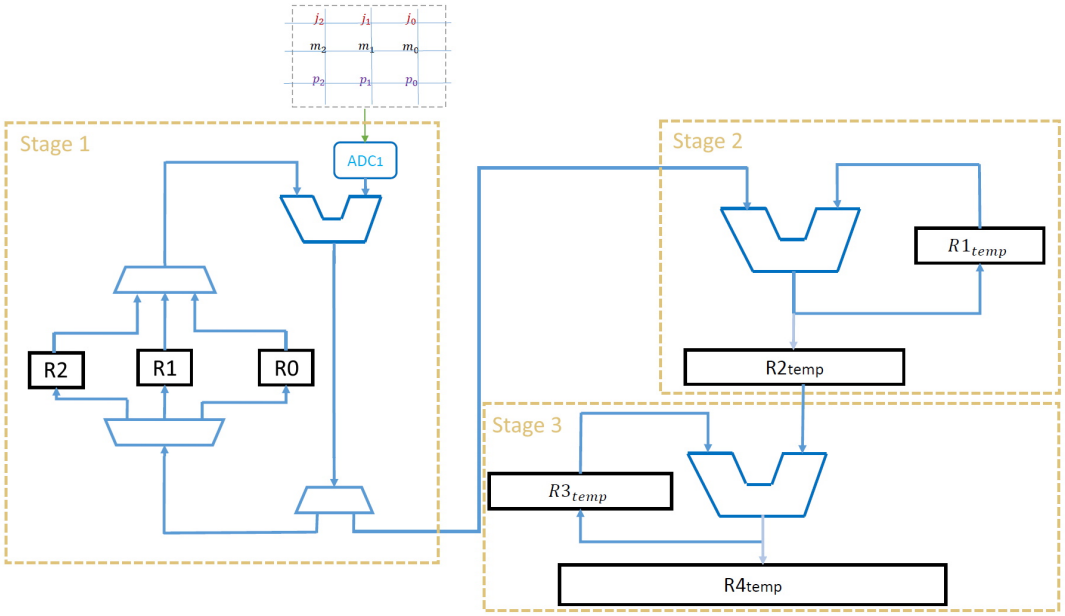


Figure 3.3: Overall addition organization per ADC, from Zahedi et al. [63].

will be $size(multiplier) + size(multiplicand) + Log_2(crossbarheight)$. If the entire multiplicand would be stored in the columns read by a single ADC, the result stored in $R4_{temp}$ would be the final result of the dot-operation. However, if the number of ADCs present in the architecture increases, at some point the result would be spread over multiple of these $R4_{temp}$ registers. In this case, an extra stage is required to accumulate the final result.

3.1.6. SystemC Simulator

To be able to explore different concepts of the CIM-tile architecture, a cycle-accurate CIM-tile simulator has been developed. This simulator is written in systemC to be able to accurately simulate hardware [64, 65]. While it is possible to write systemC code in a way such that it is synthesizable through High-Level Synthesis [66], the simulator code has not been written with this intention. The simulator provides a basis which can be altered and expanded upon to allow for initial testing and exploring of solutions proposed throughout this thesis report. The simulator provides numbers for performance and energy consumption and also allows for tracking of the control signals. The energy consumption of the write operation, read/computational operation and DIM drivers are calculated using equations 3.1, 3.2 and 3.3, respectively. E_{DIM} refers to the energy consumption by the row and column drivers. N_{Ar} and N_c refer to the crossbars activated rows and columns, respectively. E_cell refers to the energy required to read a single cell value from the crossbar. Finally, P_{DIM} and $T_{crossbar}$ refer to the power consumption of the drivers and the latency of the crossbar, respectively.

The main benefit of this new simulator with respect to the simulators described in Section 2.4 is the ability to execute the applications thanks to the detailed execution model and in-memory instructions. Due to this benefit and considering the aforementioned equations, more accurate execution time as well as data-dependent energy number are computed. However, the MNEMOSENE simulator is not yet as advanced as the neuromorphic chip simulator with regards to modeling dynamic effects such as write variability and random noise. A comparison of the different simulators is given in Table 3.3.

$$Write\ \ energy = E_{DIM(source)} + N_{Ac} \cdot (E_{DIM(write)} + E_{cell(write)}) \tag{3.1}$$

$$Read/Computational\ \ Energy = N_{Ar} \cdot (E_{DIM(source)} + N_C \cdot E_{cell(read)}) \tag{3.2}$$

$$E_{DIM(write/source)} = P_{DIM(write/source)} \cdot T_{crossbar(write/read)} \tag{3.3}$$

	Simulation speed	power/energy accuracy	Defect modeling
MNSIM	+	0	0
Neuromorphic	0	+	+
MNEMOSENE	0	++	0

Table 3.3: Comparison of the various in-memory computing simulators.

3.2. Challenges

The prior work done on the proposed architecture serves as starting point for this thesis project. While a high-level architecture has been designed and simulated already, still a number of challenges should be overcome to be able to build an actual hardware implementation of the CIM-tile. This section introduces the main challenges that will be tackled throughout this thesis project.

3.2.1. Instruction set design and simulation

When compiling a single micro-instruction into a set of nano-instructions, the instruction count can go upwards of 250 thousand nano-instructions. In addition, the initial implementation of several instructions (RDS, WDS, WD, CS) is inefficient as they carry a lot of redundant information, leading to huge instruction sizes and impractical hardware implementation for executing these instructions. This leads to a set of challenges that should be overcome during this thesis project:

- A more data-efficient ISA should be designed. The new iteration of nano-instructions should be implemented in the hardware design, so each of the nano-instructions will have to be investigated for possible challenges in hardware implementation. The addition of new nano-instructions to attain a hardware-feasible set of nano-instructions may be necessary.
- A write-verification scheme should be implemented in the CIM-tile. This will most likely lead to the introduction of additional nano-instructions and possibly changing the tile architecture.

Solutions to these challenges are presented in Chapter 4. The results obtained by simulation of the newly proposed instructions and architecture will be used to explore different architecture options and finally implement the flexible hardware design of the CIM-tile.

3.2.2. Hardware design

Implementing the simulated architecture in hardware imposes a new set of challenges:

- The tile should be capable of correctly executing the newly designed instructions generated by the compiler. This also entails a hardware implementation of the efficient addition scheme design.
- The unconventional pipelining should be supported by the architecture. Results obtained from simulation may be used to decide on the optimum number of pipeline stages and the pipeline configuration.
- An efficient way of feeding data to the tile and retrieving the computation result from the tile should be investigated, keeping in mind potential external overhead.
- The implemented architecture should be tested. However, as there are currently no memristor arrays available, some way of adequately modelling the array is required.

These challenges are investigated and solutions are proposed in Chapter 5.

3.3. Conclusions

This chapter introduced the prior works performed on the novel tile architecture proposed throughout this thesis report. An introduction to the different aspects of the architecture as well as the initial nano-instructions has been given, providing the starting point of this thesis project. An initial high-level architecture has been proposed. The architecture has been investigated for possible pipelining configurations. However, the architecture and its accompanying instructions should be revised from a hardware perspective. Once the instruction set architecture has been revised, the effectiveness of the pipelining should be investigated. In addition, a hardware implementation for the proposed efficient addition scheme should be proposed. A simulation

platform is provided which can be extended to test the effectiveness of different instruction set- and hardware designs. To summarize, the main challenges this thesis project focusses on are:

- Revisit the ISA from a hardware perspective, also making the new instructions more data-efficient.
- Add a write-verification method to the architecture.
- Propose a detailed hardware design for the CIM-tile.
- The tile should support the unconventional pipelining.
- Propose a way of testing the architecture without access to the analog components.

4

Instruction Design and Simulation

This chapter provides solutions to the challenges regarding the ISA implementation and presents simulation results regarding different pipeline configurations obtained by the new version of the simulator. Section 4.1 describes the change of the instruction set and the simulator to allow for hardware implementation of the architecture. Section 4.1.9 presents the results regarding instruction file size reduction and discusses the performance of different pipeline configurations.

4.1. ISA design

The ISA presented in the prior work has been defined with the high-level simulation in mind. As a result, the version of the instructions, compiler and simulator is not adequate for use in a hardware implementation. The instruction set should be defined in a way that it is feasible to implement it in the resulting hardware implementation of the CIM-tile. In general, there are several design aspects that should be kept in mind when designing the ISA.

- First of all, the nano-instructions of an application that are about to be executed should be stored in some memory where they are read sequentially. As the storage of nano-instructions is assumed to happen off-line (before starting the execution of the application), the size of this memory will provide an upper limit to the size of the set of nano-instructions. As the initial version of the compiler uses straightforward but very data-intensive implementations of the row selection, write data, write data selection, and column selection instructions, even a small-sized benchmark is translated to an instruction size between 0.49 and 3.4 MB, depending on the ADC precision and number of ADCs. This dependence on ADC configuration comes from the fact that when either more, or higher precision ADCs are used, less instructions are required to read the result, leading to a smaller overall instruction file size. Key aspects of the ISA design are reducing the number of instructions required per program as well as reducing the size of each instruction individually while maintaining their generality as high as possible.

- Secondly, the CIM-tile, being part of the micro-architecture, will be provided with required data via a databus [67]. The bandwidth of this databus limits the amount of data that can be read per instruction. As there are no hard constraints on the size of this data-bus and the crossbar size yet, the proposed solutions should take into account the fact that the databus might be too small to provide all required data in a single cycle.

- Finally, the initial set of instructions embedded all of the data that is fed to the crossbar in the instruction itself. For real applications, this will not be possible due to the data not being available at compile time. When designing the instructions set, it should thus be taken into account that compile-time unknown data is fed to the CIM-tile at run-time.

This section describes the detailed implementation of all of the instructions that will be supported by the hardware implementation of the CIM-tile. First, assumptions on some of the design parameters are made.

4.1.1. Assumptions

The key parameters that impact the ISA design are:

- Crossbar size.
- Data bus bandwidth from instruction memory or data memory.
- Number of available ADCs (alternatively, the number of columns that each ADC is responsible for).
- Number of cell levels supported by the memristor technology.

In this chapter, a crossbar size of 256x256 is assumed. It should be kept in mind that the crossbar does not necessarily have to be square. For exploring the situation that the bandwidth is too small to fill the registers of the tile in a single instruction, a bandwidth of 32 bits is assumed unless otherwise stated. As the number of columns that can be selected depends directly on the number of available ADCs, this number is a key aspect in the tile performance. In this chapter, it is assumed 32 ADCs are available unless otherwise stated. This implies each ADC is responsible for 8 of the crossbar columns. No hard design choices are made based on the assumptions on crossbar sizing, bandwidth and available ADCs and the effect of altering the values of these parameters is studied when they impact the performance or efficiency of the proposed solution. Finally, it is assumed the memristor cells only support two cell levels, LRS and HRS. Keeping in mind that this may change in the future, the design is made such that it can be expanded for supporting more than two cell levels.

4.1.2. Benchmark exploration

To get an indication regarding the contribution of each nano-instructions to the total instruction file size, a benchmark compiled using the initial version of the micro-to-nano compiler will be explored. All micro-instruction format benchmarks provided by partners of the MNEMOSENE project consist of combinations of storing data in the crossbar and performing matrix-matrix multiplications. The benchmark that will be explored is the 'GEMM' benchmark, consisting of a single matrix-matrix multiplication, preceded by storing the multiplicand matrix in the crossbar. The micro-instructions and an abbreviated version of the set of nano-instructions compiled by the initial version of the compiler can be found in appendix A. As described in Chapter 3, the RDS, WDS, WD and CS instructions will have to be improved in order to be feasible in hardware with potential bandwidth constraints. In addition, these instructions carry a lot of redundant information, which is the main reason for the large instruction file sizes. The size of instruction file in which data for our tile registers such as *RDS*, *WD*, *WDS*, and *CS* is omitted, becomes negligible. The information obtained from the exploration, which is provided in Appendix B, will provide insight to steer towards a new set of nano-instructions. This section focuses on exploring the data-intensive instructions. Table 4.1 shows the contribution of the individual data-intensive instructions to the instruction file size. As this is dependent on the number of available ADCs and the ADC precision, four different configurations are shown. The *RDS* instruction makes up for around 3.1% up to 12.2% of the total data of these data-intensive instructions. *WD* and *WDS* instructions are only executed in the 'store' part of the benchmark, which corresponds to a marginal 0.06% of the total data for low ADC precision, up to a maximum of 4.8% for high ADC precision. This 4.8% is evenly distributed over *RDS*, *WD* and *WDS* instructions, so *WD* and *WDS* both make up for a maximum of 1.6% of the data-intensive instructions. The remaining percentage of the data in the instructions is part of the often executed *CS* instruction, which is 84.6% up to 96.8% of the total data. While all of the mentioned instructions implementations should be changed in order to attain hardware feasibility, the main data reduction can thus be expected in the change of the CS instruction implementation.

#ADCs	ADC bits	file size (MB)	#RDS	#WD/WDS	#CS	%RDS	%WD/WDS	%CS
8	5	13.75	13040	240	409600	3.08	0.06	96.80
8	8	1.74	1840	240	51200	3.44	0.45	95.67
32	5	3.80	13040	240	102400	11.25	0.21	88.34
32	8	0.50	1840	240	12800	12.17	1.59	84.66

Table 4.1: Initial instruction file exploration for the gemm benchmark.

4.1.3. Row Data Selection

Recall the purpose of row selection; fill the RDS register so the appropriate rows of the crossbar will be activated, thus performing the desired crossbar functionality on the correct data present in the crossbar. This is done by feeding either a logic '0' or '1' to the RDS register index corresponding to the row that should be

selected or deselected. The selection of rows can be interpreted in two conceptually different ways. The first interpretation called “Immediate Row selection" is the row selection serving only as a simple selection signal which only signals to which rows the crossbar should apply its functionality. In this interpretation the row selection data is always known at compile time and can thus be stored in the row selection instruction. The second interpretation of row selection called “data-based row selection" is using actual data to select the rows. This latter concept is used when two operands should e.g. be multiplied of which one operand is present in the crossbar, while the other operand should be provided to the tiles RDS register during the computation. This is the case for the VMM operation, used for Matrix-Matrix multiplication. The data required for this data-dependent row selection is, however, not available at compile time. An example of this can be found in the CNN applications, where the result of a matrix-matrix multiplication of one layer is required in the subsequent layers of the network. Because of this compile-time unknown result data, the row selection data can not always be stored in the nano-instruction itself. In this case, an address referring to the location of the stored row selection data should be provided. This data could be stored either in memory external to the CIM-tile or in the crossbar of another CIM-tile. As the final architecture will comprise of multiple instances of the tile-design proposed in this report, some memory management unit present in the chip should handle the data-fetching according to the addresses and feed the data to the CIM-tile. The implementation of this data-feeding for a single CIM-tile is explained in Chapter 5. For now, it is assumed the correct data elements are fed to the CIM-tile without any imposed overhead.

Immediate Row selection

As for this case of row selection the data is known at compile time, it is possible to encode the row selection data, potentially saving instruction size and count. In the most general case, however, the data is completely random, which means there are no opportunities for an efficient encoding. In this case, the data should be simply copied directly into the RDS register. This type of RDS instruction in which a block of data copied to the RDS register will from here on be referred to as RDS-blockwise (RDSb). Under the assumptions stated in Section 4.1.1, this means that (256/32 =) 8 RDSb instructions will have to be executed to fill the entire RDS register. In other cases like sparse 0 presence and sparse 1 presence, however, more efficient ways can be used. For example, if only rows from index 0 to 31 should be selected, this can be done by clearing the RDS register and then executing a single RDSb instruction, provided that there is a way to clear the register and there is a way to make sure the block of 32-bits is inserted at the right location in the RDS register. To allow for clearing the register, a new nano-instruction is introduced; RDS-clear (RDSc), which clears the register by putting all row selection values to ‘0’. To tackle the proper insertion placement of the blocks, the set of crossbar rows is divided into blocks with size equal to the bandwidth and indexed accordingly. This means that the RDS register will be divided in (256/32 =) 8 blocks, which in turn can be indexed using ($Log_2(8)$ =) 3 bits. In the dual case of sparse zero presence, the same solution can be applied provided there is a way to set all RDS register values to ‘1’. To enable this functionality, another nano-instruction is introduced; RDS-set (RDSs). Using this block-wise solution, the RDS register can be filled with any random data in 1 up to 8 instructions. In the worse case scenario, where the data for RDS register is random, the amount of RDSb instructions that will have to be executed is directly dependent on the ADC precision. If the ADC precision is 5 bit or lower (meaning only 32 rows or less can be read at a time), considering the aforementioned assumptions, only a single RDSb instruction will have to be executed, saving 88% of row selection data compared to the initial implementation. The relation between the ADC precision and the data size improvement for row selection can be found in Figure 4.1. Once the ADC precision reaches 8 bits, all rows of the crossbar may be selected during a single operation, meaning 8 RDSb instructions are required and thus no gain in terms of instruction file reduction is achieved. Once the ADC precision drops below 5-bit, meaning the ADC precision is smaller than the available bandwidth, it would be more efficient not to use the entire bandwidth, and an optimum can compromise between the block size and the number of required index bits. The relation between block size and total number of bits required for the RDSb instruction is described by Equation 4.1. Figure 4.2 shows the relation between block size for the RDSb instruction and total number of required bits. While the number of bits per instruction drops as the block size gets smaller, the total number of bits required to write a total of 32 bits to the register significantly increases due to the index bits. It can thus be concluded that, due to this overhead, small block sizes are not desirable even when low-precision ADCs are deployed.

$$\#bits\_total = \#index\_bits + \#data\_bits = log_2\left(\frac{\#crossbar\_rows}{blocksize}\right) + blocksize \tag{4.1}$$

Two other extreme cases of RDS register filling are selecting all rows of the crossbar and selecting only a single row of the crossbar. The first case is likely to happen in real applications, as the problem size may

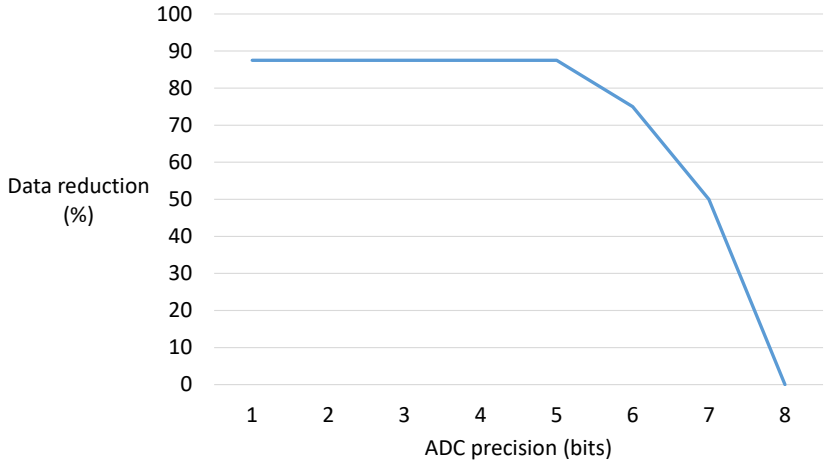


Figure 4.1: RDS instruction data reduction as function of ADC precision

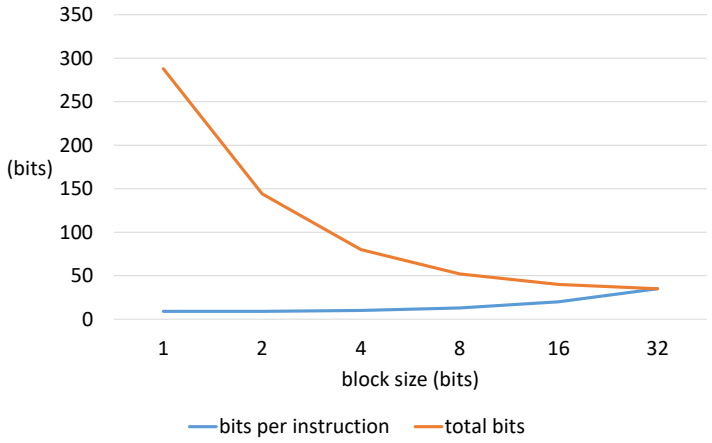


Figure 4.2: Bits required per instruction and total for writing 32 bits as function of block size

greatly exceed the crossbar size, meaning the problem will be split over different tiles for parallel computation or time-multiplexed over a single tile for sequential computation. Selecting all rows is adequately handled by the previously described solutions, as a single RDSs instruction will suffice to select all rows. The extreme case of selecting a single row can be handled by the proposed solution using 2 instructions; one RDSc and one RDSb instruction. This can be potentially cut back to a single instruction by using a different row selection scheme. This solution passes the index of a single row. For a 256 row crossbar, this can be done using $Log_2(256) = 8$ bits. In addition, the entire 32-bit bandwidth can be used to pass $(32/8 =)$ 4 row indices of rows present anywhere in the crossbar in a single instruction. Keeping in mind that the block-wise solution also needs a RDSc or RDSs instruction, this means that this row-wise solution allows for selection of up to 4 rows present anywhere in the crossbar at half instruction count, which is potentially beneficial for small applications or applications showing scattered sparse 1 presence. It should be noted that the potential of this solution is dependent on the available bandwidth and crossbar size. This row-wise solution is, however, greatly inefficient for selecting a large number of rows as the instruction count will scale linearly with the number of rows to be selected, and should thus not be the sole instruction implementation for RDS register filling. In addition, a hardware implementation of this row-wise solution would lead to multiple large decoders (index to single row, so under current assumptions it requires up to four 8-to-256 decoders and the corresponding routing). For these reasons, implementing only the block-wise filling scheme is deemed adequate for row selection under the current assumptions.

data-based row selection
As mentioned, in this case the data that should be fed into the crossbar rows is provided by some memory management unit present on the chip. This means that the tile itself does not have to be aware of any memory addressing. If not all rows can be read due to the ADC precision constraint, the tile will still have to use proper masking on the provided data. Having a single RDS register is insufficient for feeding row-selection data and a data-mask to the crossbar. Instead, the row selection is split into two registers. Similar to the WD and WDS register, an RD register is used to store the multiplier and the RDS register can be used as a masking register using the same block-wise filling scheme to efficiently fill the mask. The way data is presented to the RD register is discussed in Chapter 5. The results from the immediate row-selection also hold true for filling the RDS register when performing data-dependent row selection, with the conceptual difference being that the instructions now provides a mask for the data instead of the selection actual data itself. The tile should be able to indicate when the next set of row data-bits is required. Keeping in mind that in actual hardware the data-elements consisting of e.g. 8 bits are stored in some buffer, a new nano-instruction is introduced, called Row-Data-shift (RDsh). The reasoning behind defining this instruction and the underlying implementation of this buffer is discussed in Chapter 5.

To conclude this section, the block-wise solution is deemed the best stand-alone solution, in itself adequately covering the described cases of row selection. The row-wise solution might provide additional benefit depending on the available data bandwidth, and may thus also be considered for future technology iterations. Depending on ADC precision, the proposed solution offers a data reduction for the row selection up to 87%. As the RDS instruction make up about 11% of the data-intensive instruction in the initial compiler implementation, this corresponds to up to $0.87 \cdot 11 = 9.6\%$ data reduction in the total benchmark.

4.1.4. Column Selection

As mentioned in Section 4.1.2, the initial implementation of the column select instruction makes up for about 88% of the data in the benchmarks set of nano-instructions. The initial implementation offers large flexibility at the cost of a huge instruction size. To decrease this size, some solutions are proposed. Following the assumptions made in Section 4.1.1, the initial implementation of the column selection will execute up to 8 sequential CS/DoR instructions, as can be seen in the compiled set of nano-instructions in Appendix A. Up to a total amount of $(8 \cdot 256 =)$ 2048 data bits are required for the set of CS instructions. This will be taken as a reference to evaluate the performance of the proposed solutions.

- The first proposed solution relies on introducing a new nano-instruction to shift the column selection register bits by 1 position; Column-Select-shift (CSs). Now, a single standard CS instruction consisting of 256 bits will be executed to set the starting point of each ADC. Afterwards, up to 7 CSs instructions can be executed to allow for reading all 8 columns per ADC. In addition, either (i) a set of de-activation bits (1 per ADC) can be appended to the CS instruction to indicate that none of the ADCs columns should be read or (ii) a set of activation bits can be appended to every CS/CSs instruction to indicate whether the ADC should be activated for that specific instruction. variant (i) uses less data, but is less flexible as only consecutive strings of columns can be selected. Variant (ii) allows for selecting any arbitrary set of columns at the cost of higher data usage due to the extra selection bits. Variant (i) would lead to a total of $(256 + 32 =)$ 288 data bits, saving 86% of data compared to the initial implementation. This can be improved further by instead of using 256 bits to directly point to specific columns, using a $(log_2(8) =)$ 3-bit index per ADC. This would lead to a total of $(32 \cdot 3 + 32 =)$ 128 data bits, corresponding to a reduction of 94%. Using the more flexible variant (ii), all possible combinations of columns can be read using a total of $(32 \cdot 8 + 32 \cdot 3 =)$ 352 bits. As at this stage of the architecture design, maintaining selection flexibility is deemd more important than the data efficiency, only variant (ii) will be considered from now on. The relation between number of ADCs and required data bits for this variant can be seen in Figure 4.3, named 'Solution 1'.

- A different solution is passing a single index in the CS instruction, which will set all ADCs to that specific index, and using a set of activation bits to indicate whether an ADC should read the column at that index. Under the current assumptions, that would lead to one 3 bit index followed by 32 activation bits. This solution thus leads to $(8 \cdot 35 =)$ 280 bits. Again, this is dependent on the number of available ADCs. The relation between total required bits and number of available ADCs for this solution can be seen in Figure 4.3 under 'Solution 2'. Once the number of available ADCs drops below 8, the amount

of data required for this solution rises significantly. However, having this few ADCs in the final design would have significant impact on the overall performance, as the number of sequential CS and DoR instructions that should be executed to read the same amount of data then increases. Due to this, it is desired to not have such a low number of ADCs available.

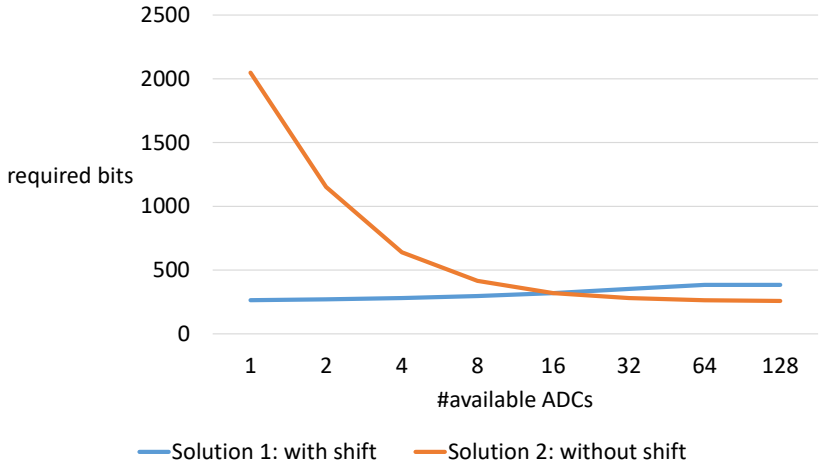


Figure 4.3: data bits required against number of available ADCs.

Comparing the two solutions, the second solution potentially leads to more instructions to be executed than first solution as the ADC indices can no longer be set individually like they could in the CS instruction of the first solution. This is, however, compensated by the fact that an arbitrary index can be passed in each of the CS instructions, meaning that indices can be skipped if none of the columns corresponding to that index have to be read, which could not be done using the shift instruction. Both of these cases are application-dependent. Comparing the amount of data required for both solutions, shown in Figure 4.3, it can be seen that both solutions use the same amount of data if 16 ADCs are available. For larger numbers of available ADCs, the amount of data bits used for the first solution increases, while for the second solution it decreases. In addition, the second solution does not require a shift instruction, thus omitting some controller complexity. Due to these reasons, the second solution is deemed most suitable.

Using the current assumptions and using this solution, the total decrease of data used for the CS instructions is 86.3%. As the CS instructions made up around 88% of the total data-intensive instructions, this proposed solution saves ($0.88 \cdot 86.3 =$) 75.9% of the total data present in the compiled instruction file.

4.1.5. Write Data Selection

The write data selection register serves a similar purpose as the RDS register, but selecting columns instead of rows. However, the write data selection register is only used for selecting the columns, and can not be used for feeding data directly for an operation as was the case for the row selection, described in Section 4.1.3. Due to this, no address based solutions have to be provided for the write data selection. Similar reasoning as for the RDSb solution applies to the write data selection, leading to a block-wise solution for the WDS instructions as well, introducing two new nano-instructions to clear and set the entire WDS register; Write-Data-Select-clear (WDSc) and Write-Data-Select-set (WDSs), respectively.

However, depending on the application the WDS register does not have to be filled for every operation. For example, if a matrix should be written to the crossbar, the set of selected columns does not change for the entire matrix. This means the WDS register has to be filled only once, keeping its values until a write operation should be performed on a different set of columns. In the benchmark example, the WDS register has to be set only once throughout the entire benchmark, meaning the subsequent WDS instructions can be omitted. Introducing the block-wise solution for WDS would lead to a negligible improvement in data-usage and instruction count. This is a good argument in favor of using a more simple implementation, omitting the WDSc and WDSs instructions, thus leading to a less complex controller design. Under the current assumptions, this would mean that the WDS register is always filled by 8 sequential WDS instruction, passing 32 bits per instruction to fill the 256-bit register. Omitting instructions does however impose some constraint on the further design of the instruction set. If instructions that alter the program counter would be introduced, it

may occur that the WDS register should be refilled with proper selection data. Due to this, the block-wise filling and set- and clear instructions are kept in the design for now.

4.1.6. Write Data

As mentioned in Section 4.1.2, the Write data only makes up for a small fraction of the overall data presented in the compiled instruction file (about 0.03% up to 1.2% of the explored benchmark). In addition, it should be kept in mind that this data, which is provided by the WD instruction itself in the initial ISA, is also not necessarily known at compile time. The WD instruction will thus not hold any data. The data is instead presented to the tile via the data bus. The new implementation of the WD instruction thus only has to ensure the data is positioned correctly in the WD register. This can be done by having the WD function as a stripped down version of the block-wise register filling, only providing the index for positioning the data. As the WDS acts like a mask determining which columns are written to the crossbar, The remaining column values do not matter. This means that the WD register does not have to be cleared, so a clear- or set instruction for the WD register is not required.

4.1.7. Write verification

As mentioned in Section 2.4, Write variability can lead to wrong data being present in the crossbar array [61]. To ensure correctness of the data present in the crossbar, A write verification scheme should be added to the design. Essentially, the data stored in a row will be read directly after storing and compared to the data present in the write data register. If a wrong value is read from the crossbar, the data should be re-written. As it can not be determined at compile time if and how many times this rewriting should be executed, it can not be implemented in the compiler directly using the existing nano-instructions. This can be solved by introducing a conditional branch-not-equal (BNE) instruction, branching back to the writing instructions if the contents of the write data register do not match the values read from the crossbar array. However, this means that the current pipeline design should be modified as the data present in the WD register will be overwritten by the next operation before verification can be performed. Two solutions are proposed and their respective benefits and drawbacks are listed.

- **Solution 1:** The first solution is to stall the next pipelined operations until the current write operation has been verified for correctness. This solution has as obvious drawback that the write throughput is lowered due to the lack of pipelining. As the verification operation will have to propagate through all of the pipeline stages, the throughput will be at least more than doubled. The impact of this on the overall performance of a program depends on the execution time of the writing relative to the other operations like vector-matrix multiplication and is thus not only dependent on the architecture (number of ADCs, ADC precision), but also program-dependent. For the $gemm$ benchmark, the impact on overall execution time increase ranges between 0.7% up to 40% depending on the architecture parameters. However, implementation of this solution is relatively simple as the branch instruction and some verification flag will suffice for performing this functionality. It should also be kept in mind that one of the main ways in-memory computation offers benefits is by re-using the data stored in the crossbar. Assuming multiple computations are performed on the stored data (instead of only 1 matrix-matrix-multiplication as in the $gemm$ benchmark and then programming the crossbar again), the impact of this solution on performance drops significantly.
- **Solution 2:** The second solution requires additional hardware and controller complexity to alleviate the impact on write-throughput. By introducing additional registers, the WD and WDS data can be stored until the write verification has been performed, allowing subsequent operations to overwrite the WD and WDS register values. As the verification operation still has to propagate through the entire pipeline, their is still some overhead induced. For the gemm benchmark, the execution time increase ranges from 0.3% up to 20% when using this solution, showing the potential benefit of this solution with respect to solution 1. The impact of this write verification solution on the hardware implementation is, however, significant. As the write operation is now pipelined, out-of-order execution should be supported by the CIM-tile. In addition, the tile controller should be capable of adequately filling the WD/WDS temporary registers.

Both solutions can be improved by using the verification outcome per column to alter the WDS mask, thus only re-writing the wrongly written values, improving power consumption and device lifetime at the cost of extra controller complexity and routing. As the bit error rate is very small, in the order of 10e-6 [61],

Opcode	Operand 1	Operand 2	Function description
RDSb	RDS block index	Masking data	Place masking data into RDS reg at indexed location
RDSc			Clear the entire RDS register
RDSs			Set the entire RDS register
RDsh			Shift the next bits of row data into the RD register
WDb	WD block index		Place write data in WD register at indexed location
WDSb	WDS block index	Masking data	Place masking data into WDS reg at indexed location
WDSc			Clear the entire WDS register
WDSs			Set the entire WDS register
FS	Crossbar function		Select crossbar functionality
DoA			Activate the crossbar function
DoS			Sample the crossbar output
CS	ADC column index	ADC activation bits	Select column to be read by ADC
DoR			Activate the ADC
jal	PC address		Jump to PC at 'address' and save current PC value.
jr			Jump to the PC stored at previous jal instruction.
BNE			Branch to PC stored in branch reg for rewriting
LS			Indicate the last section of column reads
IADD			Activate the third stage of the addition unit
CP			Copy result per ADC to output buffer
AS	Adder selection bits		Select adders for addition between ADCs
CB			Copy addition between ADC result to output buffer

Table 4.2: Overview of the newly proposed ISA.

the benefit of this alteration on current technology and for reasonable sized crossbars would be marginal. As the performance difference between the two solutions gets smaller for larger applications, it is deemed best to first implement the first solution as to not over-complicate the design. To be able to test the write verification scheme, the simulator has been extended to allow for introducing errors due to write-variability.

4.1.8. Jump instruction

Having instructions that manipulate the program counter also allow for further improvement in instruction file reduction by omitting instruction duplicates. This is especially beneficial for the readout-stage instructions as these are, for example when performing a matrix-matrix multiplication, identical for every row and executed many times. In the *gemm* benchmark, executed with low ADC precision, around twelve thousand identical sets of 8 CS and 8 DoR instructions are executed. Assuming it is possible to branch to anywhere in the instruction file, this can be cut down to having only a single set of those instructions present in the file, replacing the other sets with a jump-and-link (jal) instruction. This instruction stores the current program counter and jumps to the set of readout instructions. In order to correctly resume the instructions after branching, a jump-register (jr) instruction should be introduced, which allows for returning to the program counter stored by the jal instruction. These instructions are direct copies of the eponymous instructions found in the MIPS ISA [68]. In the gemm benchmark using low ADC precision, an additional 15% of the total instruction file size is saved. For high ADC precision, the contribution of the read stage to the file size drops and consequently the improvement obtained drops to 12%.

4.1.9. ISA summary and simulation results

This section summarizes the changes and additions to the initially proposed ISA. The initial implementations of the instructions have been revisited from a hardware perspective, taking potential architectural restrictions into account. A write verification scheme is introduced to the architecture, leading to newly introduced instructions. A jump instruction has been implemented to allow for omitting duplicate instruction sections.

Table 4.2 shows the newly proposed set of nano-instructions as well as their operands and a short description for each of the instructions.

The improvements to the ISA discussed throughout this chapter have been implemented in the micro-to-nano instruction compiler. To allow for functionally testing the new ISA, the simulator has been adapted to

#ADCs	ADC precision (bit)	initial file size (MB)	final file size (MB)	reduction (%)
8	5	13.75	0.28	98
8	8	1.74	0.11	93
32	5	3.80	0.28	92
32	8	0.50	0.11	78

Table 4.3: Instruction file results for various configurations.

enable running programs using the instructions retrieved from the compiler. Compiling the gemm benchmark using the improved compiler, a comparison can be made between the initial version of the instructions and the newly defined instructions. The results can be seen in Table 4.3. A file size reduction of 78% up to 98% compared to the initial ISA implementation has been achieved, while also adapting instructions and introducing new instruction to allow for hardware implementation of the architecture.

4.2. Pipeline configuration exploration

Having completed the improved version of the simulator, The effectiveness of different pipeline organizations can explored. Recall the four different pipeline stages presented in Section 3.1.4. Figure 4.4 shows the contribution of each of the four stages to the latency of the system. Clearly, for low numbers of ADCs, the read stage contributes far more to the latency of the system. As the level of parallelism that is achievable in the readout stage is directly determined by the number of available ADCs, it is also not possible to split the read-stage into multiple smaller stages. For these low number of ADC configurations, having a separate pipeline stage for the setup stage and execution stage does not improve the tile performance at all. These stages may then be combined, resulting in a less complex tile controller design as less synchronization between stages is required. With increasing number of ADCs, the readout stage latency becomes smaller. As the latency of the other stages do not scale, their relative contribution to the latency increases. Once the number of available ADCs is 32 or higher, combining the setup and execute stages starts impacting the tile performance.

Partners of the MNEMOSENE project working on the analog technology have provided information on the current technology capability regarding the ADCs used in the CIM-tile. A single ADC per set of 32 columns of the crossbar is possible, which corresponds to 8 ADCs in the 256-column crossbar. This means that for current technology, combining the setup and execution stage does not impose any performance degradation. Having a single ADC per 32-bits also means that, due to the maximum supported datatype size of 32-bit, no addition between ADCs is required. This Means that the AS and CB instructions will never be executed, and thus the latency contribution of the addition stage is near-zero compared to the read-stage. This in turn means that the read-stage and the addition stage can also be combined, again leading to less required synchronization and thus a less complex tile controller. In addition, no registers between the two stages are required, saving hardware. For these reasons, it is decided that the proof of concept implementation of the architecture will consist of a two stage pipeline, and the addition between ADCs will also be omitted. As this design decision may change with evolving technology (increasing number of ADCS), the stage synchronization and addition unit may be extended in future work to allow for three- or four stage pipelining.

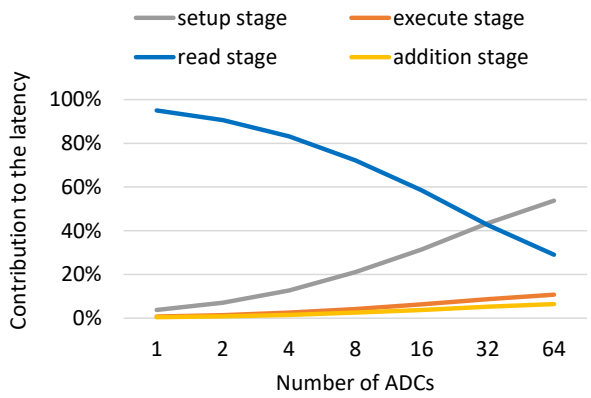


Figure 4.4: Contribution of each pipeline stage to the tile latency for a 256x256 crossbar.

4.3. Conclusions

This chapter has provided the detailed design of the newly proposed nano-instruction set architecture, adapting the initial ISA, compiler and simulator to allow for a hardware implementation to support the ISA and architecture. Results regarding the instruction file size have been presented, ensuring that the nano-instruction format no longer imposes unreasonable constraints on instruction memory size. Depending on the architecture configuration, the newly proposed instruction set achieves up to a 98% reduction in instruction file size for a typical program. Finally, results obtained from the adapted version of the simulator have been used to explore the impact of different pipeline configurations on the CIM-tile performance, allowing for a well-founded decision on the pipelining configuration to be implemented in hardware considering the current technology. For this current technology, it is found that the setup stage and execute stage of the pipeline design can be combined without impacting the performance. In addition, the read-stage and addition stage may be combined as the addition stage contributes very little to the cycle latency of the tile, saving hardware and leading to a less complex synchronization requirement for the tile controller. With evolving technology, the design may be extended without any required alteration of the instruction set. An efficient instruction set has thus been proposed, allowing for scaling towards future technology.

5

System Hardware Implementation

Now that the CIM-tile architecture has been designed and simulated on a high level, the design can be implemented in hardware. As mentioned in Chapter 1, the design is not targeting specific design corners such as power efficiency or high-performance. Instead, this chapter provides different implementations for components of the tile architecture that may allow for adapting the architecture to specific design requirements at a later stage of the MNEMOSENE project. Section 5.1 proposes an extended version of the high-level architecture. Sections 5.2, 5.3, and 5.4 describe the detailed implementation for some of the major components in the tile architecture such as the tile controller and the efficient addition unit. Finally, Section 5.6 proposes a digital model of a memristor array to allow for functional testing of the tile architecture on an FPGA.

5.1. Enhanced tile architecture

As the tile receives its data from the outside memory-management unit, it is preferred to define a clear interface between the 'outside' and the tile. To this end, some buffers can be introduced to the tile architecture. The same holds for the output generated by the tile, which should be processed by the outside unit. A total of three buffers are introduced to the tile architecture; the Write-Data (WD) buffer, Row-Data (RD) buffer and output buffer. For now, it is assumed the sourcing and sinking of this data is performed by an external unit (e.g. memory controller) without inducing any overhead. This means the tile-controller can communicate to the external unit whenever the buffer should be filled or emptied, and this filling and emptying can be performed at any time, provided the outside unit is not servicing another tile. This outside unit will be provided with knowledge on which tile requires what data, and where this data is stored from the micro-instructions. Detailed design of this outside unit is left for future work. The implementation of each of the three buffers will be presented and the implications and impact of these buffers on the performance of the tile will be discussed in Section 5.2.

Regarding the output buffer, executing logical as well as read operations in the crossbar result in the final values without the need of further digital processing in the addition unit. Hence, the addition unit is bypassed and the output can be directly stored in the output buffer. However, for these operations, we consider separate internal register to store the result temporarily before putting it to the output buffer due to the two following reasons. Firstly, as the output of the read operation is also used for write-verification, the contents of the output buffer would also have to be an input to the write verification, leading to a less clean interface to the outside world. Secondly, as the number of available ADCs still leads to multiple cycles being required to read out the result, having a separate internal register allows the tile to start reading the result of the next operation while the outside unit may not have read the contents of the output buffer yet.

The proposed write verification scheme should also be included in the extended architecture. This block of the architecture takes as inputs the contents of the WD and WDS register, as well as the output from the crossbar after it has been sampled. The output produced by the verification is a flag which indicates whether the write operation has been performed successfully or a write-error has occurred. This flag is then fed to the tile-controller to allow for altering the execution flow, repeating the write operation if necessary.

The extended tile architecture can be seen in Figure 5.1. Note that, in order to synthesize the entire tile in an FPGA, the functionality of analog modules such as DIM, crossbar, ADC, and analog MUX are mimicked by their digital models. These models will be discussed in Section 5.6.

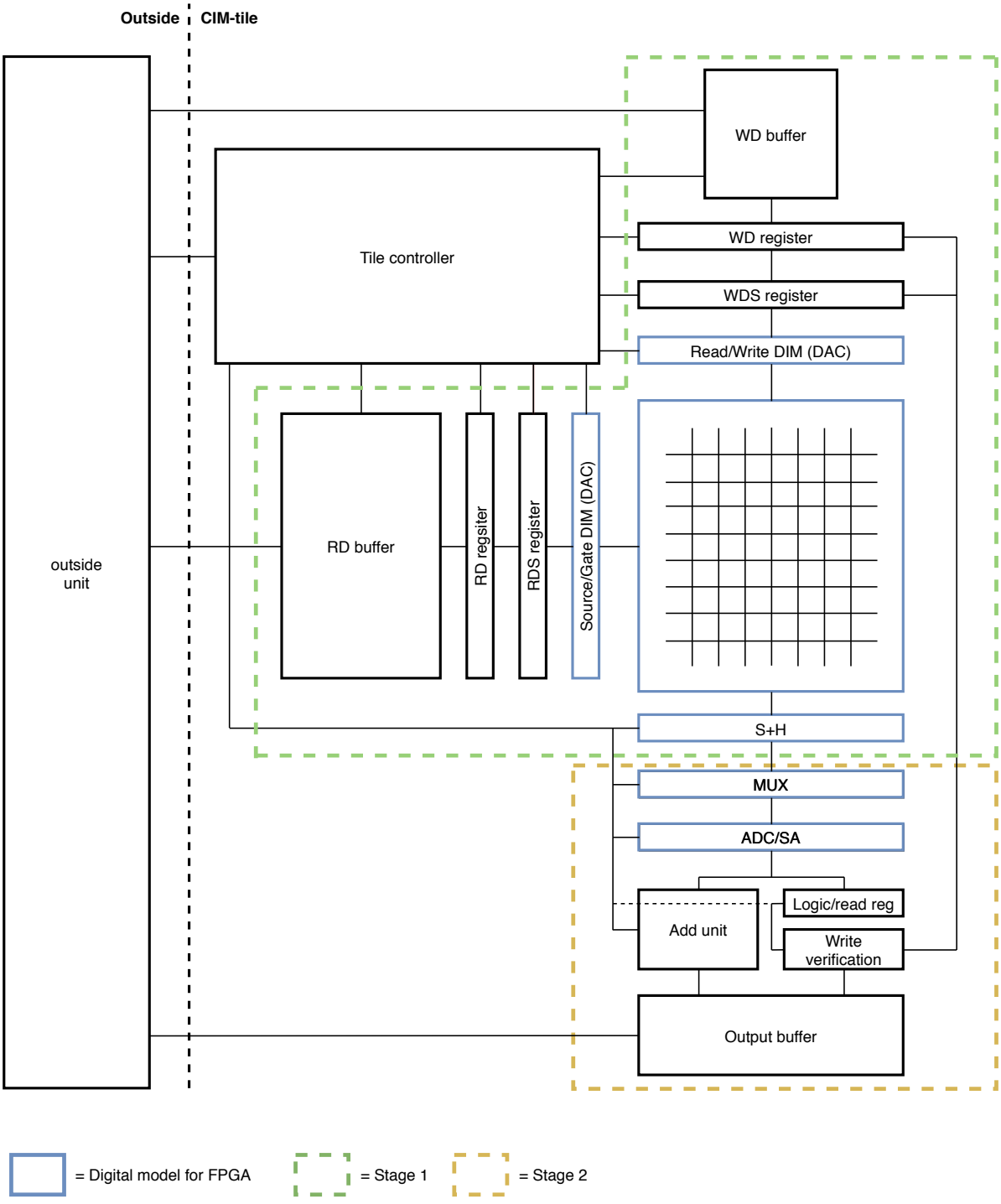


Figure 5.1: The extended tile architecture.

5.2. Data sourcing and sinking

This section discusses the implementation and impact of the newly introduced WD-, RD- and output buffer, as well as their impact on the tile performance.

WD buffer The WD buffer is used to temporarily store write data. While it is possible to allow the outside unit to put the write data directly into the WD register, there are several reasons this is not desired.

- Firstly, having no buffer imposes more stringent timing constraints on feeding the data to the tile, as for the 2-stage pipeline design the contents of the WD register may not be altered until the operation has been finished in the crossbar.
- Secondly, the outside would have direct access to the internal registers of the tile. This means that there is less of a distinction between the tile controller and the outside unit. It is preferred to have a clear interface between the two so the design of the two units may be decoupled.

For these reasons it is decided to always have a buffer for the write data in the design. As seen in the benchmark exploration in Section 4.1.2, the write operation only makes up for a small percentage of the gemm-benchmark. This is, however, highly dependent on the application. For example, in a database application the write operation may make up for a significant portion of the operations, and thus an efficient design of the WD buffer should not be neglected. The two main architecture parameters impacting this buffer are (i) the available bandwidth for writing data from the outside unit to the WD buffer and (ii) the number of columns present in the crossbar (also referred to as crossbar width). Two different scenarios can be distinguished regarding these parameters:

1. The bandwidth is larger than or equal to the crossbar width. In this scenario, the outside unit is able to provide enough data in one cycle to the tile to allow the tile to write data to an entire row of the crossbar. This would mean the WDb instruction may be omitted as no block-wise filling of the WD register is required. However, if in this scenario the WDb instruction is omitted, still some detection is required to indicate when the next set of write-data may be fed to the WD register.
2. The bandwidth is smaller than the crossbar width. In this scenario, the outside unit will require multiple cycles to provide enough data to the tile for the write operation. As the bandwidth is, in this case, smaller than the crossbar width, a new design parameter may be introduced for the minimum size of the WD buffer. The minimum size of the buffer can be chosen to be either:
    - Equal to the available bandwidth.
    - Equal to the crossbar width.

The obvious benefit of choosing the bandwidth as the minimum size is that less registers may be used, which means less area usage and consequently a lower cost. However, this does mean that the buffer is no longer capable of storing enough data to write to an entire row of the crossbar. This leads to a more stringent timing constraint for filling the buffer. Consider the following example, assuming a crossbar width of 256 columns and a bandwidth of 128-bit: filling an entire rows' worth of data to the tile's WD register will require three clock cycles. During the first cycle, the outside unit provides the first 128 bit to the WD buffer. During the second cycle, this data will be put in the correct position of the WD register by the tile controller and the outside unit will write the second 128 bit to the WD buffer. During the third cycle, the second 128 bit can be put in the WD register, meaning the register is full and the tile can thus perform its write operation. It should be noted that the outside unit will have to be available between the first WD register fill and the second one. Now, we consider the case of a WD buffer of size equal to the width of the crossbar (so 256-bit). In this case, the outside controller still requires 2 cycles to provide all data for the single row. However, this data can always be stored in 2 subsequent cycles as the tile does not require to process the first 128 bits before the outside controller can store the second 128 bits into the buffer. It should be evident that the larger WD buffer thus alleviates the timing constraint between the outside unit and the tile controller at the cost of a larger buffer.

Figure 5.2 illustrates how the timing constraint imposed by the smaller WD buffer can impact the overall performance. In this example, the three tiles are all supposed to execute their WD instruction simultaneously (in the example at cycle 6). However, because of the outside unit having to wait until the first WD instruction is executed before writing the second set of data, the WD instruction should be stalled for tiles 2 and

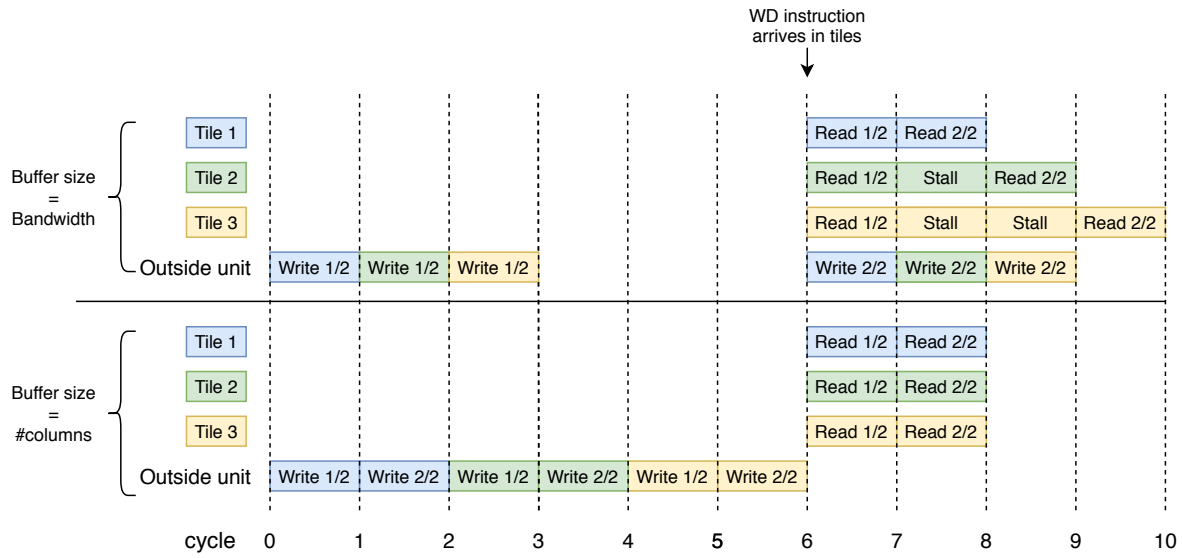


Figure 5.2: Execution flow for three tiles requiring 2 buffer writes and reads for WD buffer size equal to bandwidth or #columns. In this example, all tiles arrive at the WD instruction at cycle 6.

3, resulting in 2 cycles overhead. It should be noted that the actual performance impact is dependent on a number of factors, such as the number of tiles serviced by the outside unit, the application (Number of write operations and their position in the program) and the available bandwidth for writing to the WD buffer, and the cycle latencies of other pipeline stages of the tile. As a number of these factors are unknown at the time of this implementation (e.g. read-stage latency due to unknown number of ADCs), it is deemed best to implement the buffer of size equal to the crossbar-width as to not introduce the aforementioned timing constraint. This is also beneficial for an architecture consisting of multiple tiles as then the timing for servicing each of the tiles also becomes more critical. Exploration of the impact of smaller and larger WD buffer sizes on the performance can be performed using the simulator for any configuration once these factors are known.

Now that the buffer size has been determined, different implementations of the buffer can be considered.

- As the outside unit is aware of the order in which the tile requires the data from the micro-instruction, the buffer may be implemented by a FIFO shift register of which the width is equal to the bandwidth for writing to the buffer and the height is such that the product of width and height equal the size of the crossbar width. This implementation is shown in Figure 5.3a. This solution has two drawbacks. First, as only the top-most row of the buffer can be accessed by the outside unit, a fixed number of shifts is required for the first set of data to arrive at the bottom of the buffer. Secondly, shifting the data through the entire register leads to extra switches in the buffer, leading to energy usage overhead.

- The second solution fills the buffer from the bottom up instead of from the top down, ensuring that the data is always present at the bottom of the buffer. This means no longer a fixed number of cycles is required for filling the buffer. However, this does not entirely solve the issue regarding energy usage overhead and even introduces new challenges such as the outside unit having to provide the data in reversed ordering and the fact that reading and writing can no longer occur at the same time as the first required element is the last to be written to the buffer. In addition, the buffer will now have to support bi-directional shifting, which introduces more hardware and consequently also worsens area usage and energy consumption.

- A third solution, shown in Figure 5.3b, solves both these issues by allowing the outside unit to write to any row of the buffer freely. Using e.g. a counter which increments on a write from the outside unit, decrements on a read from the tile and remains constant when both happen during the same cycle, it can be ensured the data from the outside is written to the bottom-most free row of the buffer. The data present in the filled rows of the buffer is shifted downwards one row when the tile reads a row from the buffer. An obvious drawback of this third solution is the extra hardware that is required to allow for the individually addressable rows of the buffer and the simultaneous reading and writing.

implementation	area	performance	complexity
Bandwidth-sized buffer	+++	0	+++
Crossbar-width sized shift buffer (top down shift)	++	++	+++
Crossbar-width sized shift buffer (bottom up shift)	+	++	+
Crossbar-width sized shift buffer + individual addressing	0	+++	0

Table 5.1: Summary of WD buffer implementations.

The benefits and drawbacks of each of the proposed solutions are summarized in Table 5.1. Taking everything into account, it is deemed best to start by exploring an implementation containing the simple top-down shifting buffer solution. If, after this exploration, the overhead cycles or switching energy prove to be substantial drawbacks to the architecture, the other solutions may be further explored.

RD buffer For buffering the row-data, a different buffer structure should be used. Assuming the tile processes a datatype size larger than 1 bit, the row data is accessed bit-by-bit from LSB to MSB. A parallel-in-serial-out shift register would allow the outside unit to feed the required data elements into the buffer one or multiple elements at a time (depending on the available bandwidth) without having to extract the correct bits from the elements. Then by shifting the elements within the buffer, the buffer allows the crossbar to access the individual bits for each row. In order to fit the data elements, the width of the buffer should be equal to the maximum datatype size supported by the CIM-tile. This is for now assumed to be 32-bit. As for each of the rows an input data-element of the multiplier should be stored, the height of this buffer should be equal to the number of rows present in the crossbar.

Next, different ways the outside controller can access this buffer can be considered. The main consideration is if each of the registers present in the buffer is individually accessible by the outside unit.

- One option is to have a shift buffer similar to the WD buffer. Now, only the top register of the buffer is accessible and the written values are shifted down until the buffer has been filled. This results in a 2D array shift register, shown in Figure 5.4. However, this solution leads to energy consumption overhead due to the shifting when writing and potentially leads to large cycle overhead. For example, when the tile only requires a value to be written to the bottom-most row, this solution would still require shifting the value through the entire register.

- To combat these constraints, the second solution has individually addressable rows in the buffer. In this case, the buffer can be filled in a number of cycles equal to the number of rows that should be fed with data for the particular operation. Figure 5.5 shows the structure of the addressable buffer. Each of the registers corresponding to a row are implemented using parallel-in-serial-out shifting registers. Write-enabling of the registers can be performed by the tile controller or the outside unit for reading or writing, respectively. A drawback of this solution is the fact that the outside now has to communicate not only the data, but also the address to the tile.

Because of the drawbacks regarding cycle and energy overhead for the first solution, individually addressable registers are preferred.

Output buffer The output buffer is used for storing the results of a computation until the result has been read by the outside controller. However, proper sizing of this output buffer is dependent on the tile architecture (number of ADCs and ADC precision). For now, it is assumed the ADCs have sufficient precision to be able to perform computation on all rows of the crossbar. This means that the largest resulting element obtained during a matrix-matrix multiplication can be calculated using Equation 5.1.

$$Max\_element\_size = 2 \cdot max\_datatype\_size + log_2(\#crossbar\_rows) \tag{5.1}$$

For the 32-bit maximum datatype size and 256 crossbar rows, this results in an 72-bit result. Recall the addition scheme presented in Section 3.1.5. In this case, having (256/32 =) eight 72-bit output registers would thus suffice as output buffer. However, this only holds for computation on the 32-bit datatype. If smaller datatypes should also be supported by the tile, the output registers do not suffice anymore. This is due to the fact that, while the result of a single element computation of a smaller datatype is smaller than for the maximum datatype size, more elements can be stored in the crossbar and thus more computation can be

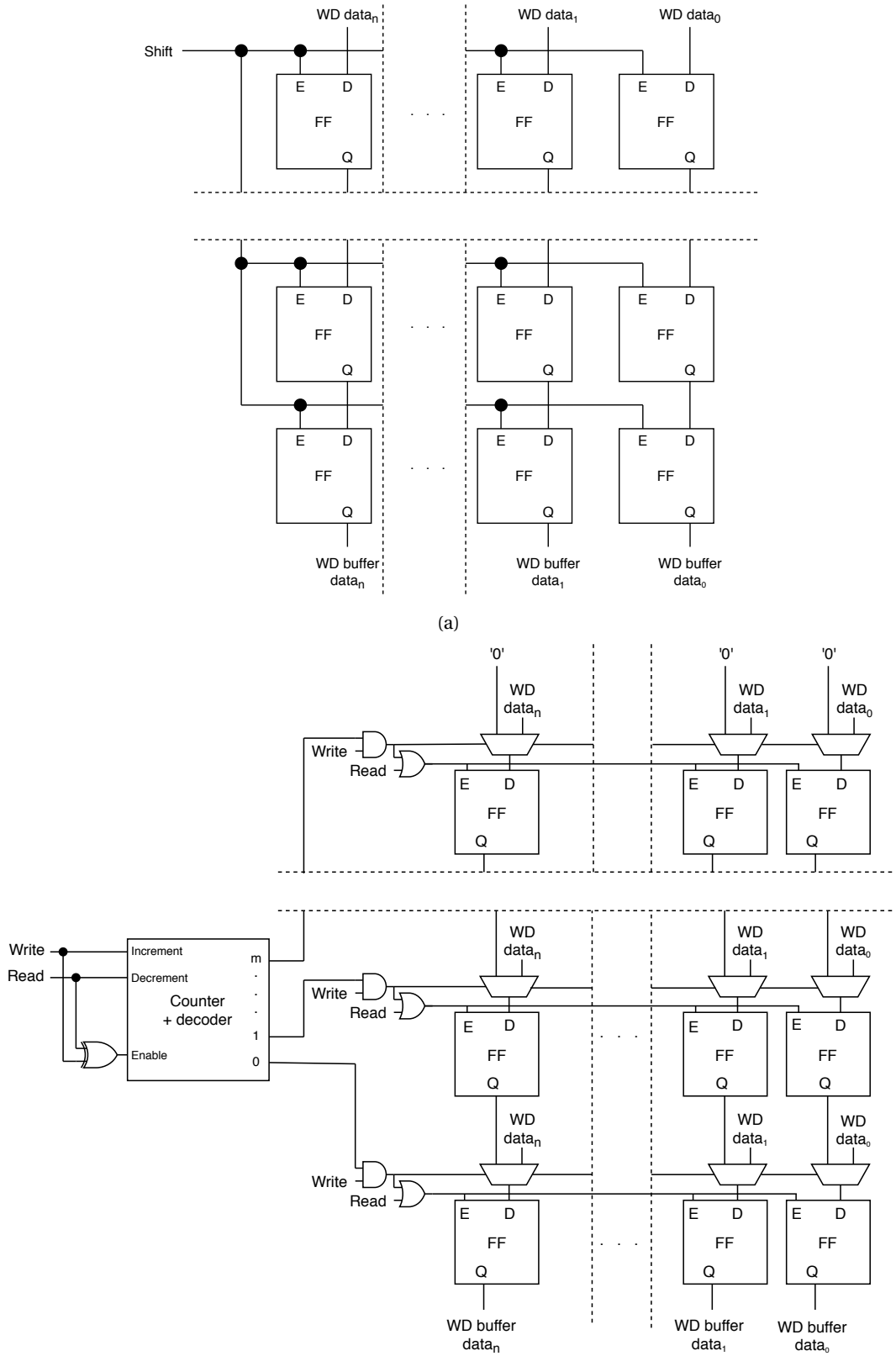


Figure 5.3: The CIM-tile WD buffer using shift (a) and bottom-up with counter for row selection (b).

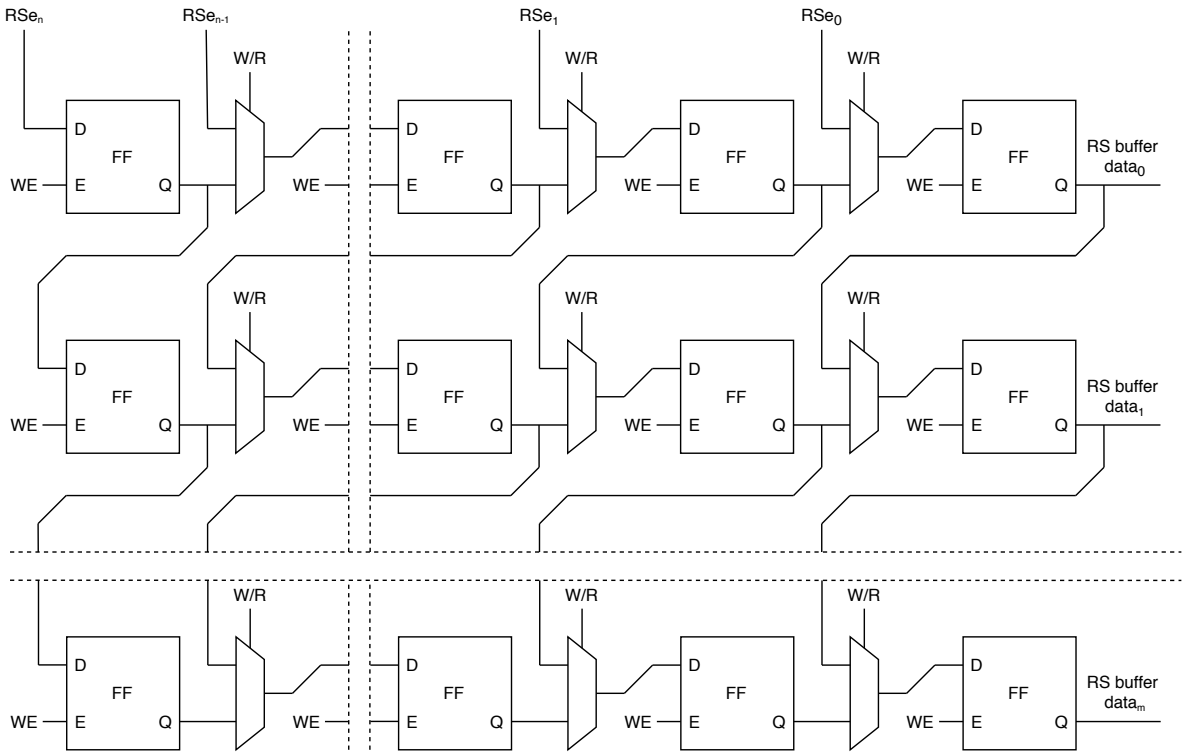


Figure 5.4: Implementation of the CIM-tile RDS buffer input shifting version

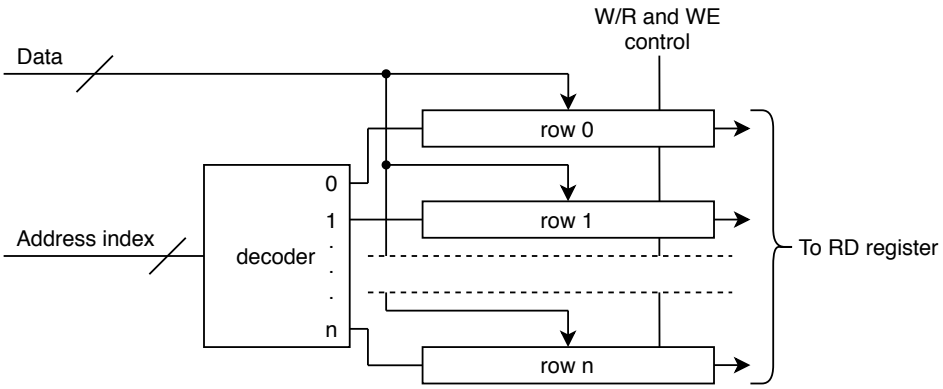


Figure 5.5: Implementation of the CIM-tile RDS buffer addressable version

datatype size	output element size	output element size · #elements stored in array
32	72	576
24	56	560
16	40	640
8	24	768
4	16	1024
2	12	1536
1	8	2048

Table 5.2: Required output register sizes per element computation and for performing computation on entirely filled crossbar per datatype, assuming 256x256 crossbar.

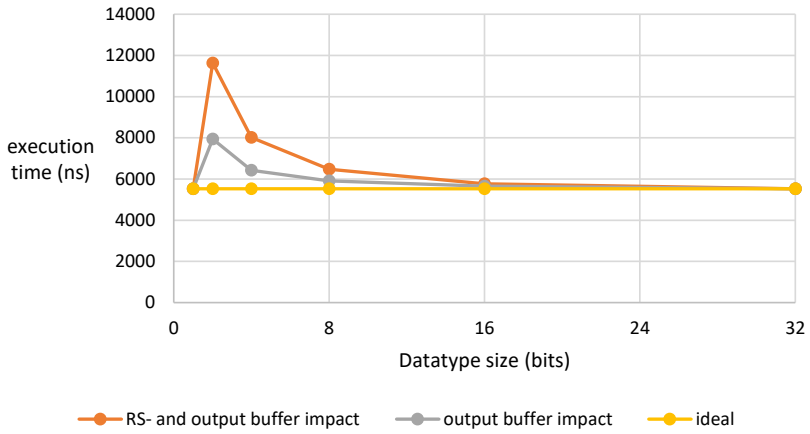


Figure 5.6: Impact of time-multiplexing on execution time for different datatype sizes.

performed in parallel. Table 5.2 shows the number of bits required per element computation as well as the number of bits required when performing computation on the entire crossbar filled with elements for different datatype sizes. It can be clearly seen that the eight 72-bit registers no longer suffice when performing computation on smaller datatypes.

A potential solution to supporting smaller datatypes is to time-multiplex the computations of elements which have a shared ADC and addition scheme. This leads to a loss in parallelism and thus potentially also leads to overall performance degradation. Figure 5.6 shows the execution time of the gemm-benchmark for different datatype sizes when executed on a tile-architecture designed for supporting a maximum datatype size of 32-bit. In the ideal case, no time-multiplexing is required and consequently the RD buffer does not impose additional overhead. The impact of time-multiplexing is shown, labelled as 'output buffer impact'. Finally, the impact of time-multiplexing and the additional overhead induced by the RD buffer due to refilling the buffer for each of the iterations is labelled 'RD- and output buffer impact'. It can be seen that performance degradation is acceptable down to 8-bit datatypes due to the read-out stage of the pipeline bottlenecking. For even smaller datatype sizes, the loss of parallelism takes a clear toll on the execution time. The single bit datatype, however, can be treated as a special case as only a single computation has to be performed before sequentially reading the output of all columns, thus not impacting performance by time-multiplexing. In addition, while the execution time of computation on some smaller datatypes rises, the total amount of output data generated by these computations is exceptionally higher than for the higher datatypes. This can be seen in Figure 5.7. While the input bit-throughput thus lowers for smaller datatypes, the output bit generation increases. The results presented in Figures 5.6 and 5.7 have also been explored for a tile-architecture designed for supporting a maximum datatype size of 16-bit. These results can be found in Appendix C.

5.3. Addition scheme design

The addition scheme described in Section 3.1.5 allows for efficient addition of the crossbar output for MMM operations [63]. Most of the addition scheme can be implemented like proposed by Zahedi et al. However, a

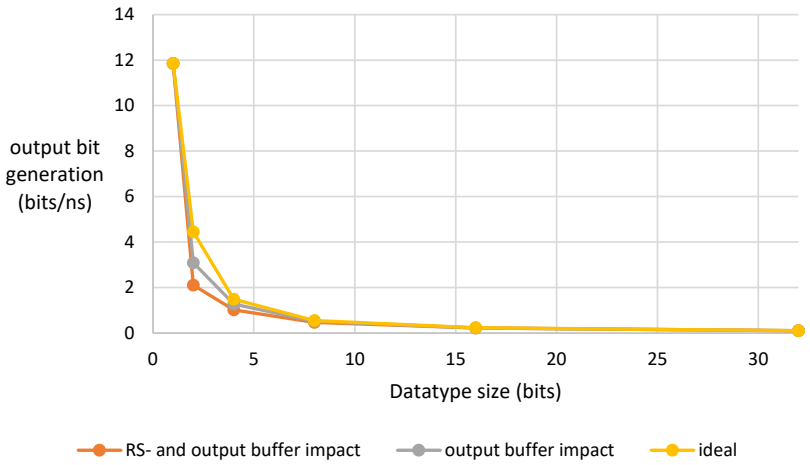


Figure 5.7: Output generation per time for different datatype sizes.

hardware implementation for the position shifting and LSB storing performed in the intermediate registers has not yet been proposed. As the number of shifts that should be performed is dependent on the datatype that is being processed, a regular shift register would not suffice as the result would not be positioned correctly in the intermediate or final register. For example, if the maximum supported datatype size is 32-bit, the $R2_{temp}$-register size would be $(8+32) = 40$ bits. However, if we perform computation on an 8-bit datatype, the LSB will be shifted only 7 times, meaning that the result will be stored in position 39 down to 23 instead of positions 15 down to 0 in the register. The same issue occurs for the $R4_{temp}$ register. Three solutions to proper result placement are proposed.

- The first solution relies on always shifting enough times to ensure proper result positioning. As the size of the registers is fixed, this can be implemented by having a counter keep track of the number of times that the result has been shifted. Once the computation has finished (indicated by the IADD, CP and CB instructions for the different intermediate registers), the result can be shifted the remaining number of times. The high level implementation can be seenin Figure 5.9a. While this solution offers low implementation complexity, it compromises the performance of the addition scheme heavily for smaller datatypes as the addition scheme now has a fixed latency, rather than a latency scaling linearly with the datatype size. The overhead cycles would add directly to the latency of the second pipeline stage, which is already the critical stage due to the read-out part of the stage. Assuming the maximum supported datatype size is 32-bit, Figure 5.8 shows the number of overhead cycles this solution imposes on each element computation for different numbers of ADCs (corresponding to different temporary register sizes). For each ADC configuration there is an optimal datatype size for which there is no cycle overhead due to the addition scheme. This corresponds to the datatype size where the result of the computation fits exactly in the $R4_{temp}$ register, and can then be directly copied to the output buffer. For smaller datatype sizes, the overhead comes from the extra shifts required in the $R2_{temp}$- and $R4_{temp}$ register. For larger datatype sizes, the overhead is due to the required addition between different $R4_{temp}$ registers.

- The second solution proposed does not require any overhead shifting, by inserting the data in the correct starting position of the intermediate register. The datatype that is being processed is used to select the proper position in which the data should be inserted in the intermediate register. After the number of shifts required for the addition, the data will be in the correct position in the register without any overhead cycles. While this solution does not compromise the addition scheme performance at all, it leads to huge amounts of multiplexing required to select the proper starting position if any random datatype should be supported by the CIM-Tile. To get rid of this unreasonable multiplexing, a constraint can be set on the datatypes that the tile can process. By having the tile only process datatypes which are multiples of bytes, only 4 different starting points in the shift register are required to support up to 4-byte datatypes. Figure 5.9b illustrates the principle of this solution. The 'adder config' signal is used to set up the multiplexing to select the proper starting point. All registers storing lower-significant bits than the starting point are set to shift, while all register storing higher-significant bits remain at '0'.

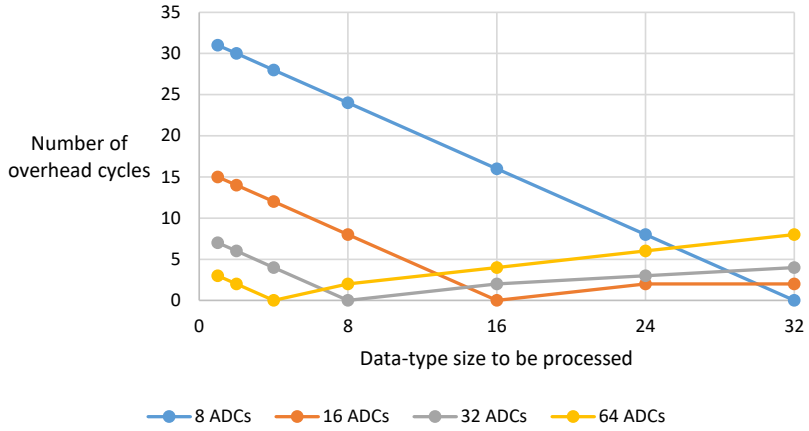


Figure 5.8: Overhead cycles imposed by the addition scheme for different #ADCs.

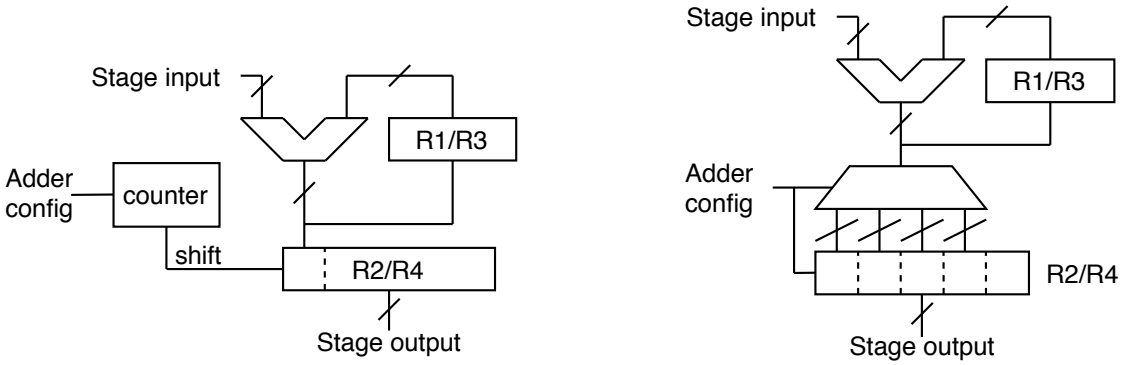


Figure 5.9: Addition scheme stage implementation using counter (a) and using a flexible starting point (b).

- If the constraint on the datatypes turns out to be a large issue for future applications, the third solution may be considered. This solution allows for computation on all random datatypes (thus omitting the constraint imposed by solution 2) without imposing large numbers of overhead cycles by using a hybrid design of the first and second solution. If, for example, computation should be performed on a 10-bit datatype, the multiplexing of solution two can set the starting point in the shift register at 16. Then, after computation, the shifting described in solution 1 can be used to shift the remaining 6 positions. This solution offers the possibility of processing any datatype with a maximum of 7 overhead shifts at the cost of more complex control. As there are, at the time of writing this report, no reasons to believe that the constraint on datatypes is an issue for applications, the second solution is deemed sufficient for the hardware implementation. For every solution, it is required that the addition unit is set up properly to perform computation on the desired datatype. Depending on how the tiles will be deployed, this can be done by either setting up the addition unit prior to executing a program, or allowing for change of the set-up at run-time with the help of the compiler by introducing a new nano-instruction.

In the initial version of the simulator, IADD, CP and CB instructions were used to copy the remaining values of the intermediate registers and activate the next addition stage. As these instructions thus activated registers in two of the stages, two cycles would be required for the instructions. It should be noted that, for each of the proposed implementations, the copying of the remaining value no longer required as these bits are readily available in the registers after the final iteration. These instructions are now thus only used to activate the next addition stage and thus the instruction can be executed in a single cycle.

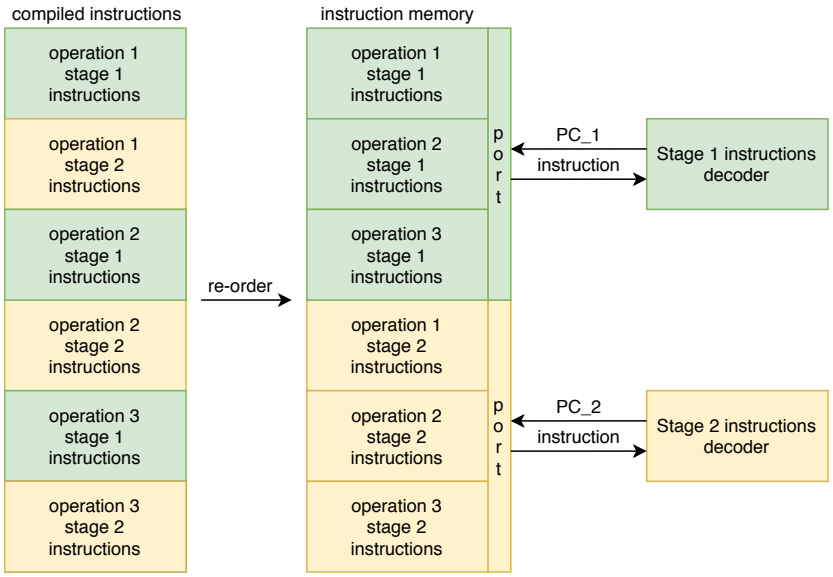


Figure 5.10: Instruction ordering to allow for two separate decoders executing their respective set of instructions.

5.4. Nano-controller design

This section described the design of the tile controller. The controller serves two main purposes, namely 1) executing the nano-instructions from the compiled program and 2) communicating with the outside unit when the WD/RD buffers need to be filled or the output buffer should be read.

Due to the unconventional pipeline structure described in Section 3.1.4, multiple instructions should be executed in parallel in a single clock cycle. To this end, two instruction decoders are required. As the different stages correspond to separate, non-overlapping subsets of the ISA, the two decoders do not have to support all different instructions, but just the instructions that correspond to their respective pipeline stage.

The instructions produced by the compiler will be re-ordered in the instruction memory to group all instructions corresponding to a specific pipeline stage, allowing the two decoders to run through their respective parts of the program without having to jump throughout the entire instruction memory. This re-ordering for three operations (e.g. VMM) is illustrated in Figure 5.10 for the case of two pipeline stages. Besides, Some synchronization between the two stage decoders is required to ensure proper program execution. This is achieved by introducing a stall detection unit which monitors what instructions are being performed by the controller parts for each stage, stalling stages when necessary. In addition, the stall-detection can be used to stall pipeline stages when the crossbar is still busy performing its computation or when either of the input buffers is empty or the output buffer is full and the stage requires these buffers for its current instruction. Monitoring the state of the buffers is performed by the buffer management block, which also communicates the state of the buffers to the outside unit so it can re-fill the input buffers or read the output buffer. The top-level design of the controller for the two-stage pipeline can be seen in Figure 5.11.

- **Stage 1 controller:** The first stage of the pipeline design is responsible for the set-up of the tile registers and the execution of the operation in the crossbar. This controller stage has to perform three tasks.
  - Update the program counter to the correct subsequent instruction
  - Provide the index and data operands of the instructions to the correct registers
  - Provide the correct control signals for the digital and analog circuits

  As the number and size of the operands differs for the instructions present in the ISA, a lookup table (LUT) is used to store the size of each of the instructions. The opcode of the instructions is used to select the proper LUT value. In addition, a separate register called the branch register is present in the controller. This register is used to branch back to the lastly executed function select instruction which selected the 'write' operation. This is used to re-perform a write when the write verification returns false. The operands of the instructions are fed into the correct registers using a demux of which the

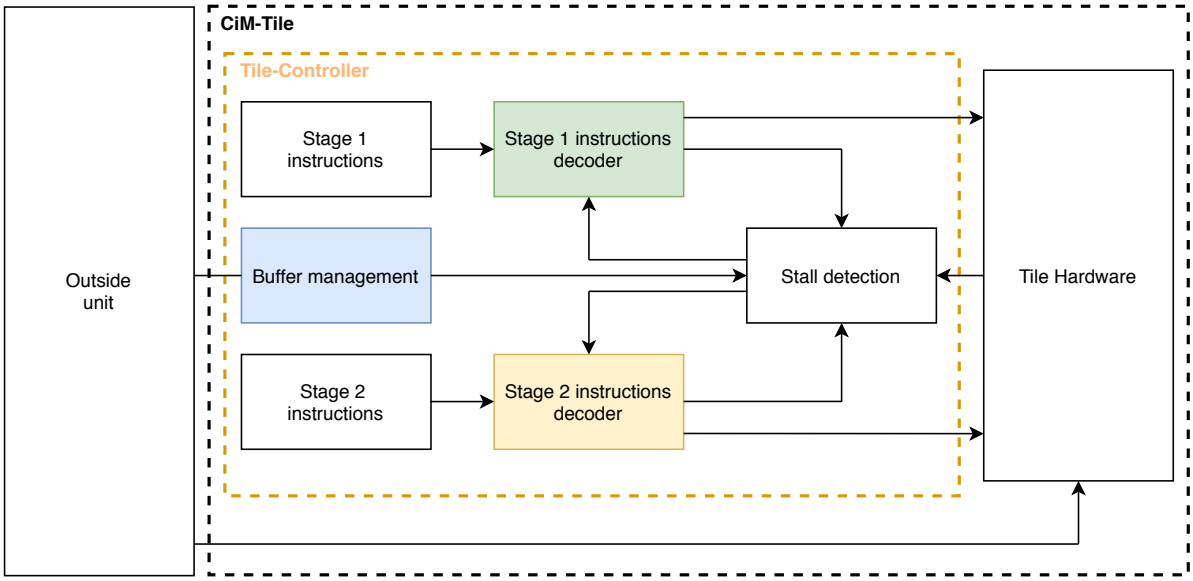


Figure 5.11: Block diagram of the tile controller for a two-stage pipelined tile.

select signal is again deduced from the opcode. Finally, the control signals for activation of each of the registers in the first stage are also generated from the information carried by the opcode. An overview of the

- **Stage 2 controller:** The second stage of the pipeline is responsible for reading out the results from the crossbar and performing addition for VMM operations, as well as placing the output of the computation in the output buffer. The stage 2 controller has to execute the same three tasks as the stage 1 controller, but differs regarding the program counter updating. In addition to the PC LUT and branching register, the PC can be updated directly from the jal instruction operand as well as returning to the linked PC value when the jr instruction is executed.

- **Stall detection:** The stall detection uses the information provided by the opcode of the instruction at the current PC of each of the stages to synchronize the stages to allow for proper program execution. As any operation starts with a set of instructions corresponding to the first stage, this stage is allowed to start executing instructions right after initiating a program execution. The second stage, however, is stalled until the results from the crossbar execution have been sampled and are thus available to the second stage. To achieve this, the second stage is stalled until the DoS instruction has been performed. Then, the first stage may start its next batch of instructions, but is stalled on executing the DoS instruction if the second stage has not yet finished its set of instructions for the operation. In addition, the stage 1 instructions that require either the RD- or WD-buffer to contain valid data should be stalled if this data is not yet present in the buffers. The same hold for instructions that require the activation of the output buffer when the result stored in the output buffer should still be read by the outside unit. The first stage is also stalled when the crossbar has not yet finished its execution. Finally, the first stage is stalled during a write verification as explained in Section 4.1.7. To summarize:

  Stage 1 is stalled when:

  - DoS should be executed while stage 2 has not yet finished its operation instructions.
  - The crossbar has not yet finished its computation.
  - RDsh should be executed while the RD buffer does not contain valid data yet.
  - WD should be executed while the WD buffer does not contain valid data yet.
  - When stage 2 is performing a read for write verification.

Stage 2 is stalled when:

- Stage 1 has not yet finished its operation instructions (indicated by DoS)
- A value should be written to the output buffer while the current values present in that buffer should still be read by the outside unit.

- **Buffer management:** The buffer management block sends signals to the stall detection and the outside unit providing information on whether there is valid data present in each of the buffers. The way the state of the buffers is monitored differs for each of the three buffers.

  - The simplest case is the output buffer. The contents of this buffer are valid after an instruction has been executed that activates the output buffer. This is either the CP or CB instruction for an architecture without or with the addition between ADCs stage of the addition unit, respectively. For the read and logic operations, the output buffer is activated after the final read has been performed. The outside unit sends a signal to the buffer management once the output buffer has been read, indicating the next computation result can be stored in the buffer once finished.
  - The state of the WD buffer can be tracked using a counter, indicating how many valid elements are present in the buffer. Once there are no more valid elements in the buffer, the buffer management will set up a flag indicating that any WD instruction will have to be stalled. Once new elements are provided by the outside unit, the flag is reset.
  - Keeping track of the RD buffer state is the most complex out of the three buffers, as the number of valid bits present in the buffer depends on the datatype that is being processed as well as the number of data copies that are put in the buffer to support the time-multiplexing described in Section 5.2. As this is directly deduced from the micro-instructions which are already being processed by the outside unit to provide the correct data-elements, the outside unit can provide the tile with the number of valid data bits directly when filling the RD buffer. As the RSsh instruction explicitly indicates when the contents of the buffer are shifted, the buffer management can now indicate when the buffer is empty.

5.5. Other peripherals

The remaining architecture components that should be implemented are

- The RD and WD register
- The RDS and WDS register
- The write verification circuitry.

As the RD buffer sizing is chosen such that it is able to hold all required data elements and shift them on the RDsh instruction, the LSB column of the RD buffer can be directly used as the RD register. While this would save registers, it means that the RD buffer can only be refilled once the last bit of the input elements has been processed by the tile. To allow the outside unit to start filling the buffer while the tile is still processing the last element, the separate RD register has been kept in the design for now. In addition, having the separate RD register helps define a clear interface between the outside and the CIM-tile internals.

The WD register can be implemented using a de-multiplexer to guide the data from the WD buffer into the correct register block according to the block-wise register filling scheme. As the de-multiplexer essentially decodes the select signal provided by the WD instruction, the decoded version of the index may be directly used to enable the correct block of registers. The implementation of the WD register can be seen in Figure 5.12.

The RDS and WDS registers can both be implemented similarly to the WD register, with the only difference being that the data fed into the demux is now coming from the RDSb and WDSb instructions instead of the WD buffer. In addition, the select registers present in the blocks require a global set/reset to support the RDSc, RDSs, WDSc and WDSb instructions. The data that is finally fed into the crossbar is then obtained by performing the logical AND on the data from the RD/WD register and the masking bits of the RDS/WDS register. As it is also required that rows can be selected directly using the masking bits (immediate row selection), the RDS register allows for bypassing of the data from the RD register when either a write-, read- or logical operation is executed.

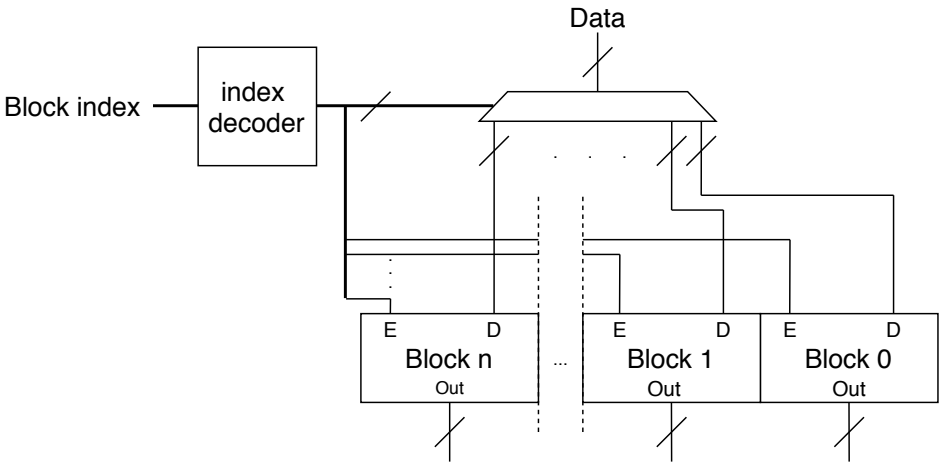


Figure 5.12: Implementation of the WD register.

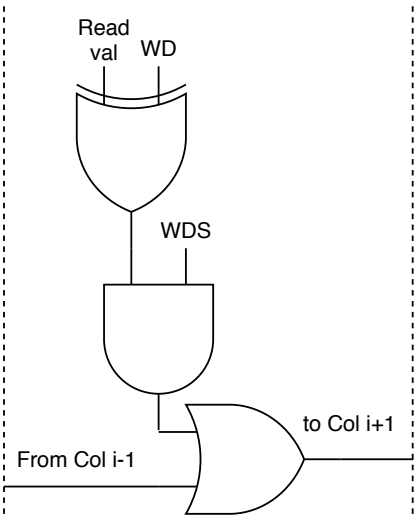


Figure 5.13: Implementation of the write verification block per column.

Recall the write verification scheme proposed in Section 4.1.7. Essentially, the write verification block only has to compare the crossbar output (stored in the read/logic register) to the data present in the WD register, setting the write verify flag when one or more bits do not match. This comparison can be made by using a logical XOR gate. As only the output of the columns to which values were written should be verified, the output of the comparison is combined with the masking data from the WDS register through an AND gate. Finally, the the logical OR operation can be used on the output produced for each of the columns to generate the verification flag, resulting in a logical '1' when a writing error has occured. The implementation schematic can be seen in Figure 5.13

5.6. Test setup/Crossbar model

As the actual crossbar array is, at the time of this design project, not yet available, a model for the crossbar should be designed to enable proper testing of the CIM-tile hardware design. This model will have to satisfy a set of requirements to ensure that the model is adequate for testing purposes. These requirements are based on the fact that the crossbar functionality should be mimicked by the model, but its intrinsics such as latency and energy consumption do not have to be captured accurately as this is already done by the simulator.

The crossbar model must be able to:

1. Store bit-values in an array as provided by the controller

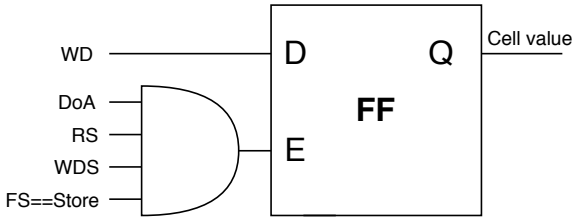


Figure 5.14: Model of a single memristor cell

2. Provide cell-values at its output when a read operation is performed

3. Perform addition of values stored in each column to support matrix-matrix multiplication

4. Perform the logic operations AND, OR & XOR

In addition to these requirements, it is preferred that the modelled crossbar latency is lower than that of the other pipeline stages. If this is the case, all different scenarios of specific stage latencies can be tested by artificially increasing the latency of the model or stages.

A single 2-level memristor cell can be modelled using a single flip-flop, storing a single data-bit. When performing a store operation, the Flip-flops enable input may be driven by the logical AND of the function-select signal to detect a store operation, the Row-Select and Write-data-Select signals to select the proper cells to write the data to and the DoA signal to only enable the Flip-flop for a single clock cycle. The data input of the Flip-flop should then be fed with the proper write data provided by the WD register. Figure 5.14 shows the implementation of a single crossbar cell.

For the required functionalities of the crossbar, first a solution for the column value addition is proposed. Here, all values stored in the cells present in a specific column that are selected by the RDS/WDS registers should be added and presented as a single value at the output of the crossbar model, essentially modelling the sum of currents fed into the SA's in the actual crossbar chip. The model can be implemented to perform this addition in a single clock cycle by having lots of addition units, essentially forming an n to $log_2(n)+1$ reduction scheme, where $n+1$ is the number of cells per column. This first solution implemented for 6 cells can be seen in Figure 5.15a. It should be evident that this kind of circuit for more plausible crossbar column sizes, e.g. 256 cells per column, leads to highly complex routing and high latency for the single addition. Highly contrasting this first solution, the addition may also be performed a single bit at a time by having a parallel-in-sequential-out shift-register across the entire crossbar column and performing the addition using a single adder at the output of the crossbar. This solution provides simple implementation in terms of routing and low latency at the cost of a high number of required cycles per addition. This solution is shown in Figure 5.15b.

While the sequential crossbar model adequately captures the crossbar functional behaviour, its high cycle latency imposes challenges to testing of the architecture. For example, a 256x256 implementation of the crossbar model has a latency 256 cycles, which is much higher than the readout stage of the pipeline for any reasonable ADC configuration. This means that the execution time for running applications during functional testing increases significantly and the tile controller is not subject to all practical scenarios in terms of stall detection. In addition, when the CIM-architecture is extended to consist of multiple CIM-tiles, the overhead imposed by the crossbar model allows the outside unit to take more time for filling the tile buffers, thus also portraying an unrealistic scenario. To combat these problems, hybrids of the two solutions may be explored to find an optimum between design complexity and latency. These hybrids are constructed by using $n:(ceil(log_2(n))+1)$ bit reduction schemes on sub-sets of cells and then storing the resulting bits in a multi-bit PISO shift-register. This way, the required number of cycles is reduced by a factor n while maintaining a relatively low-complexity implementation provided that n does not become too large. Cases for $n \leq 32$ will be considered. The explored solutions will include only reduction schemes for cell numbers that are powers of 2. A crossbar size of 256x256 will be assumed for this exploration. For easy comparison, the components used in the addition scheme are represented by the area and delay units presented in Table 5.3.

Now, a total of six different (hybrid) solutions can be considered. Figure 5.16 presents the latency per cycle against the number of required cycles to perform the addition for the different solutions. Using the results from this exploration, a solution may be selected such that the crossbar model does not bottleneck the system.

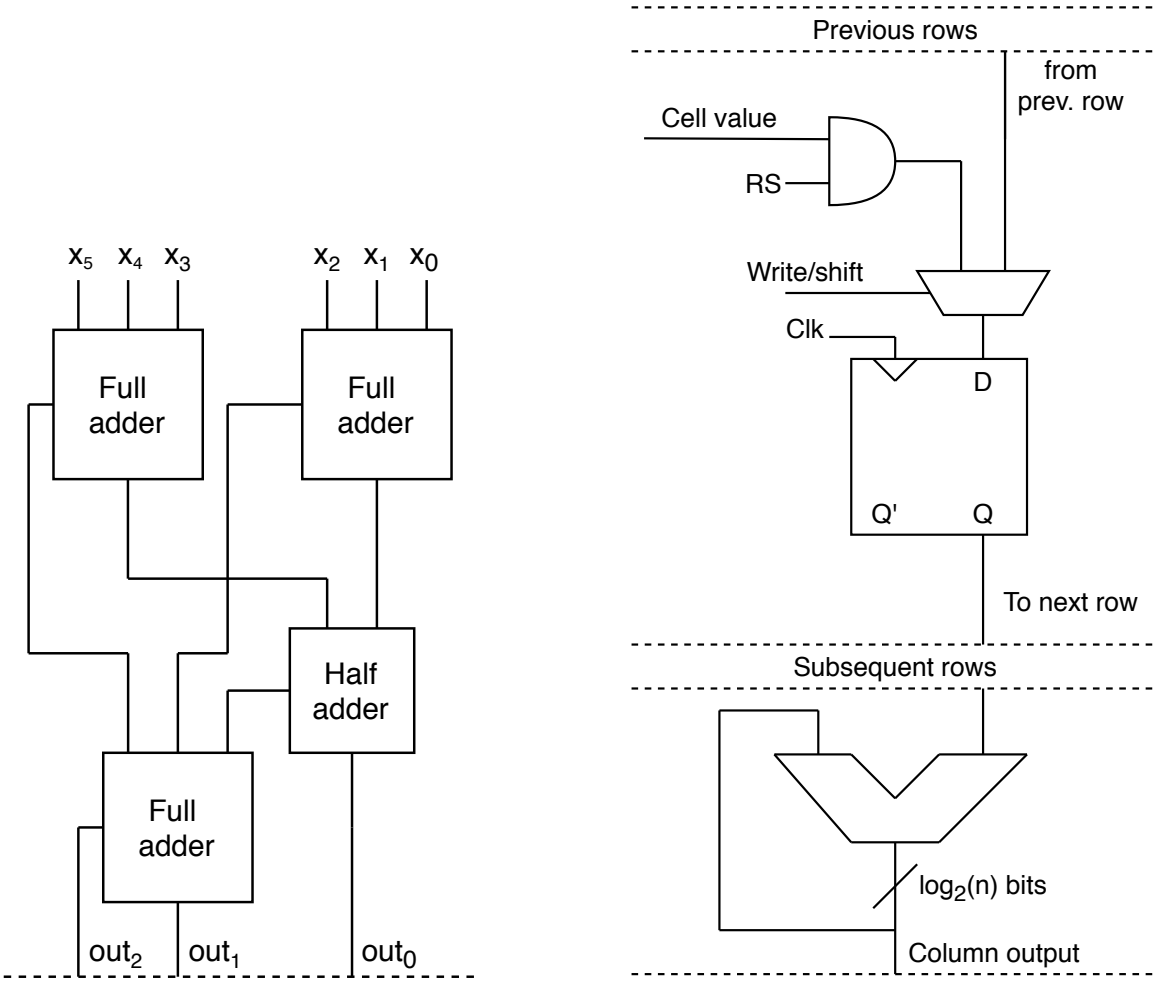


Figure 5.15: Single-cycle addition for 6 cells (a) and n-cycle addition for n cells using a PISO shift-register (b)

Component	Delay (units)
Full adder	4
Half adder	2
D flip-flop	5
2-input MUX	2

Table 5.3: Area/delay model for the components used in the addition scheme

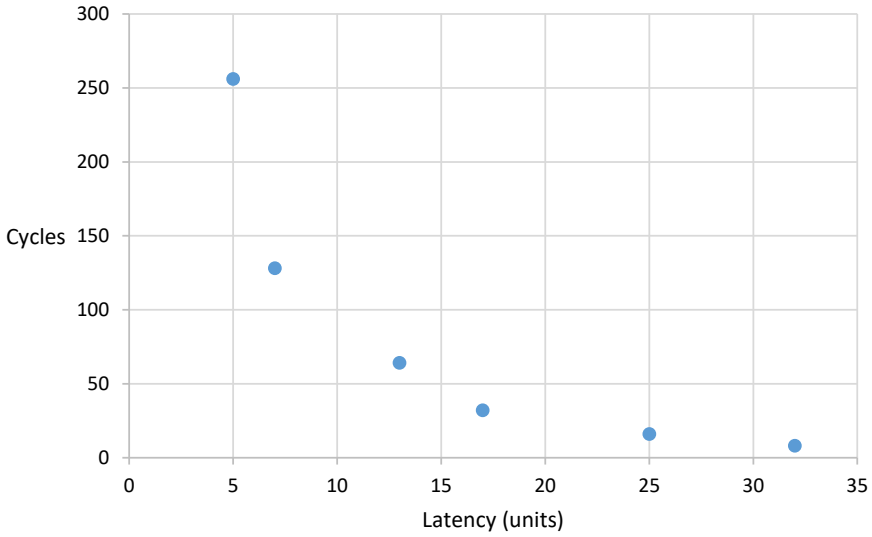


Figure 5.16: Latency of the addition scheme vs required cycles to perform addition over all cells

Next, the logical operation functionality of the crossbar should be implemented. As the column sum is readily available, it can also be used to determine the outcome of certain logical operations. The OR operation can be performed by checking if the column sum is greater than 0, which can be implemented by taking the OR of all column sum bits. The XOR operation result can be read directly from the column sum LSB, as the XOR operation essentially shows whether an even or odd number of 1's are present in the column sum. The AND operation, however, required some additional information about the number of rows that have been selected. The AND operation should output a logical '1', if and only if the column sum equals this number of selected rows. To implement this, an additional column in the crossbar model can be introduced for storing and summation of the RDS values. Checking whether the number of selected rows equals the column sum can then be performed by taking the XOR of each pair of bits between the column sum and the RDS sum, followed by an OR gate on all XOR outcomes. This method would, however, introduce a lot of hardware per column for this final check. To be more area-efficient, the AND-operation may also be performed sequentially during the shifting in the crossbar model. This can be done by introducing an extra register besides the column addition unit, which checks the current cell value against its corresponding RDS value. Once a selected row holds a '0' as its cell value, the register value will be set to '1' until it is reset by the next operation. This register value can then be used to indicate the output of the logical AND operation. Figure 5.17 shows the column model with the additional AND register. Figure 5.18 shows the implementation of the logic OR/XOR operations as well as how the correct crossbar output is selected depending on the desired operation.

Finally, some control is required to ensure correct functionality of the crossbar model. Once the crossbar receives the DoA instruction, it should perform its column addition and select the proper output dependent on the Function Select register contents. After the crossbar has finished its computation, a done signal should be outputted to the tile-controller, indicating the output is ready to read and the next computation can be performed. To this end, a crossbar-model controller has been implemented, using a counter to keep track of the number of shifts that has to be performed and generating the proper control signals and done signal depending on this counters value. The completed top level diagram of the finalized crossbar model can be seen in Figure 5.19.

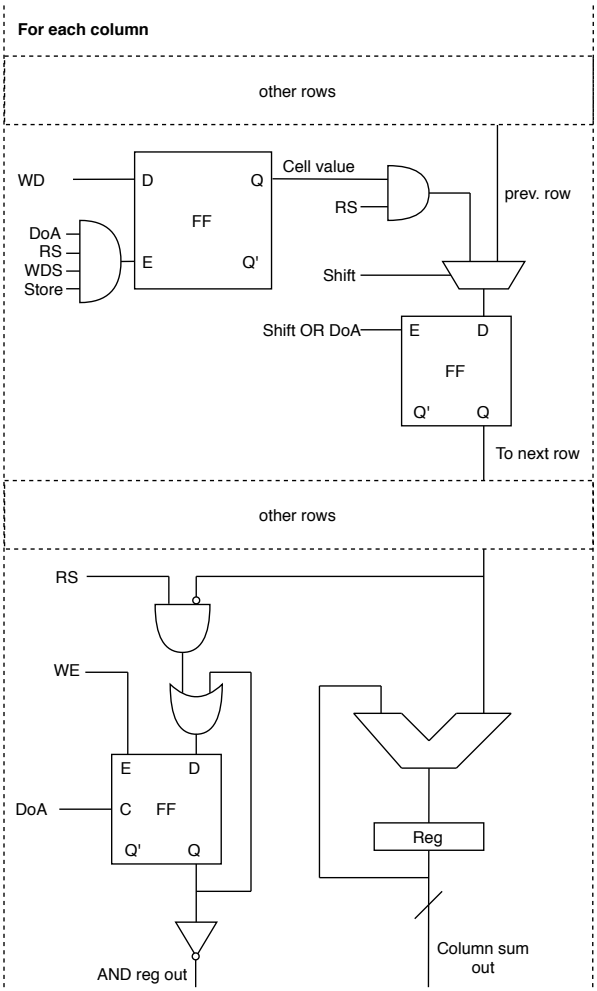


Figure 5.17: The extended crossbar column model

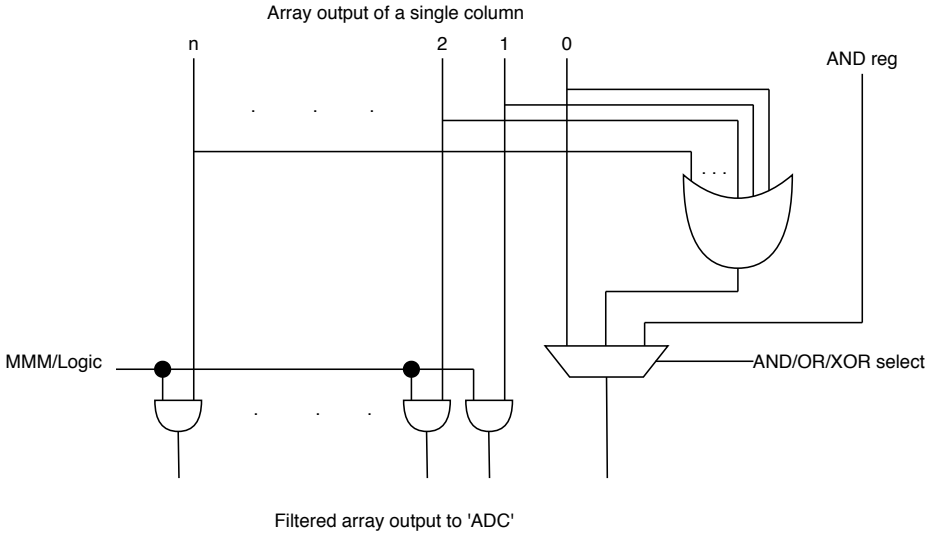


Figure 5.18: OR/XOR implementation and output selection

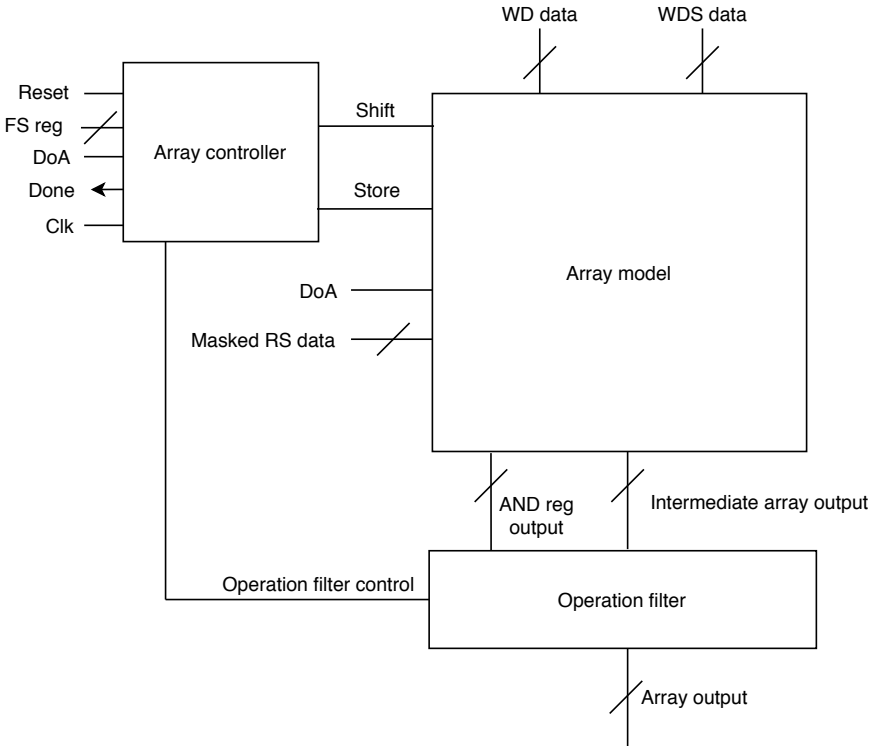


Figure 5.19: Crossbar model top level overview.

5.7. Conclusions

In this chapter, solutions have been proposed to the hardware challenges for the tile architecture. A buffer structure has been proposed for sourcing and sinking data and the impact on tile performance for different data-types has been explored, ensuring that the buffer structure does not impose large performance degradation on the architecture. Various different buffer structures across the spectrum of performance and efficiency have been proposed for the WD and RD buffer. As a detailed implementation of the outside unit and studying the effect of that unit servicing multiple tiles is left for future work, the buffers structures that will be implemented in the proof-of-concept hardware have been chosen to be a reasonable trade-off between performance, complexity and energy consumption. Furthermore, three implementations of the efficient addition scheme have been proposed, describing the situations in which each of them can be effectively deployed in the tile architecture. The addition scheme version that will be implemented imposes no cycle overhead at the cost of restricting the size of the datatypes to be multiples of bytes. If, in future versions of the architecture, it is preferable to have the tile service any random datatype size, the proposed hybrid addition scheme may be implemented in the architecture. Following the decision on the pipeline configuration from Chapter 4, a design for the CIM-tile controller supporting two stage pipelining has been proposed. The synchronization used to enable the pipelining can be easily extended to allow for more pipeline stages once this becomes beneficial for the tile performance with evolving technology. The tile architecture has thus been revisited from a hardware perspective, supporting the proposed instruction set architecture . To allow for functional testing of the design implementation without using actual memristor crossbar arrays and the required analog peripherals, a digital model capable of adequately capturing the memristor crossbar behaviour has been proposed.

6 Results

To be able to provide accurate numbers regarding the design area, power consumption and latency, the design has been implemented in synthesizable HDL. As many of the design parameters might change in the near future (e.g. number of available ADCs), the HDL has been written in a parametrized way. This also allows for exploring and directly comparing the characteristics for different configurations of the design. Section 6.1 discusses the capabilities and restrictions of the HDL design. Section 6.2 presents results obtained from synthesis and implementation of different configurations. Section 6.3 discusses possible improvements to the design. This chapter is concluded in Section 6.4.

6.1. HDL implementation details

The compiler and simulator allow for swiftly changing many design parameters such as the crossbar size, number of available ADCs and many more. Ideally, this software could be synthesized into hardware using a High-Level-Synthesis (HLS) tool such as Vivado. However, the initial version of the simulator which this project expanded on, had not been written with HLS in mind, resulting in non-synthesizable SystemC code. In order to achieve synthesizability, the simulator would have to be rebuilt entirely. In the interest of time, it is decided to build a separate HDL platform, directly implementing the design presented in Chapter 5. This HDL can then be synthesized and implemented on FPGA to serve as a proof-of-concept. Having a direct implementation without being able to change configuration parameters like in simulation would, however, not allow for comparison between different configurations in terms of area, power consumption and latency. To this end, The HDL has been written in a parametrized way similar to the simulator. The parameters that can be directly changed are:

1. Crossbar size.
2. Maximum supported datatype size, and consequently the RD buffer width.
3. Number of available ADCs.
4. Block-size for the block-wise masking registers filling.
5. Bandwidth for WD data communication from the outside unit, and consequently the WD buffer width.

Parameters 1, 2 and 3 have direct impact on the sizing of some of the major components of the architecture. To elaborate, the crossbar size has direct impact on all register and buffer sizes, the maximum supported datatype size affects the RD- and output buffer as well as the addition unit, and the number of available ADCs affects the structure of our addition unit. These parameters will thus be the main nubs to change during the results exploration in Section 6.2, while parameter 4 and 5 are mainly introduced to allow for easy connectivity once the outside unit and instruction memory are designed in future work. As properly allowing for all kinds of parameter corner cases is much more time consuming and difficult in HDL than in software, some constraints have been imposed on the parameters. The constraints put on the parameters are:

– All parameters must be powers of 2 (excluding 2^0).

- The crossbar size must be square.
- The number of columns served by each ADC must not drop below the maximum-datatype size (discussed in Section 4.1.9).
- The bandwidth for fetching row data is equal to the maximum supported datatype size.

It is worth noting that the HDL has been written such that it allows for relatively simply alleviating the aforementioned constraints in future work. For now, the HDL platform capabilities allow for exploring different configurations. Synthesis- and analysis tool Vivado is used to generate accurate timing and area reports for the design. In addition, the vectorless power analysis is used to provides an estimate for power consumption.

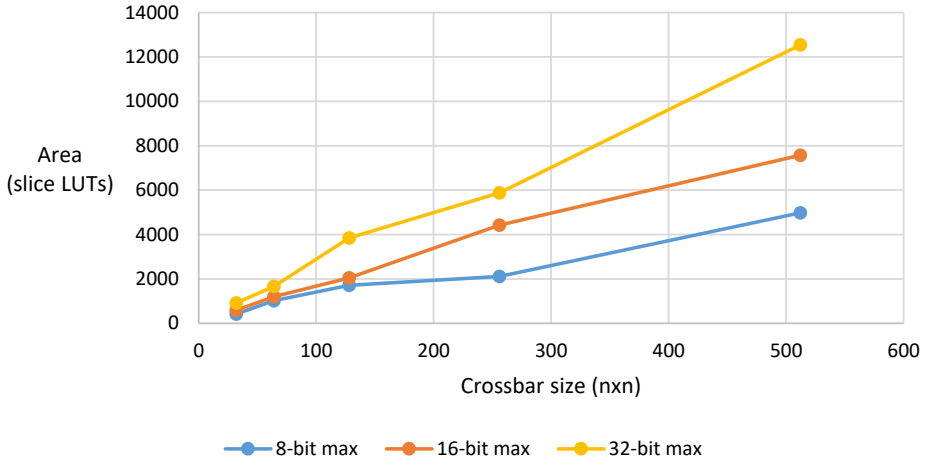
6.2. Synthesis and implementation results

Exploration of different configurations will be performed by changing the crossbar size and number of available ADCs for designs allowing for different maximum datatype sizes. The different maximum datatype sizes that will be explored are 8-bit, 16-bit and 32-bit. First, the effect of scaling the crossbar while maintaining one ADC for each set of 32 columns can be studied. In theory, most digital components of the CIM-tile architecture such as the buffers, registers and the addition unit, scale linearly with the crossbar size as long as a constant ratio between the crossbar size and the number of ADCs is maintained. Figures 6.1a and 6.1b show the area and power consumption retrieved from reports generated by Vivado during synthesis and implementation. These results include only the digital design of the CIM-tile, thus excluding the models for the crossbar, ADC, S+H and DIMs. The linear increase in area usage and power consumption for each of the three different datatype configurations can be seen. In addition, the effect of supporting different datatypes can be seen. Clearly, supporting a larger datatype requires a larger RD buffer as well as a larger addition unit, resulting in more area usage and larger power consumption.

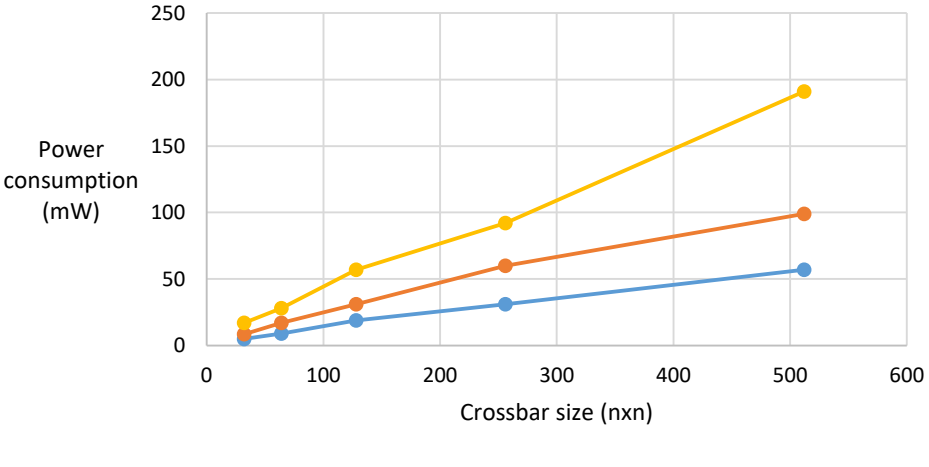
Figure 6.1c shows the maximum clock frequency the configurations can be run at without any timing errors. It can be seen that up to a crossbar size of 256x256, the maximum achievable clock frequency for each of the configurations is around 95MHz. For a crossbar size of 512x512, the RD buffer starts bottlenecking the design. It can be seen that the larger RD buffer required for supporting the larger datatypes leads to a larger impact on clock frequency. Another part of the design which is running close to its limits regarding frequency is the first stage of the addition unit. Recall that in this stage there are registers present to store the output of the ADC for each of the columns the ADC serves. (De-)multiplexing is used to select which registers should be selected for feeding into the stage-1 adder. It is evident that, when the number of ADCs decreases and each ADC thus serves an increasingly large number of columns, the multiplexing network also scales up. Under the assumption of at least 1 ADC per 32 columns, the maximum clock frequency this addition stage can handle is 104 MHz. It is good to realize that as evolving technology allows for more ADCs in the architecture in the future, the stage becomes smaller and thus less complex mutliplexing is required. In addition, if sufficient ADC precision allows for selecting all of the rows of the crossbar in one shot, the first addition stage is not required at all. This should be kept in mind when deciding on the final sizing of the CIM-tile at a later stage of the MNEMOSENE project.

To be able to pinpoint which parts of the design contribute most to the area and power consumption, the hierarchical reporting from Vivado is used. Figure 6.2 shows the typical contribution of each of the major consumers of the design for the three different maximum supported datatype sizes. It can be seen that, when supporting larger datatypes, the contribution of the RD-buffer becomes increasingly dominant. Another observation is the fact that the WD-buffer contribution to the overall power consumption is only marginal, confirming the assumptions made in Section 5.2 which lead to the use of the top-down shifting WD buffer.

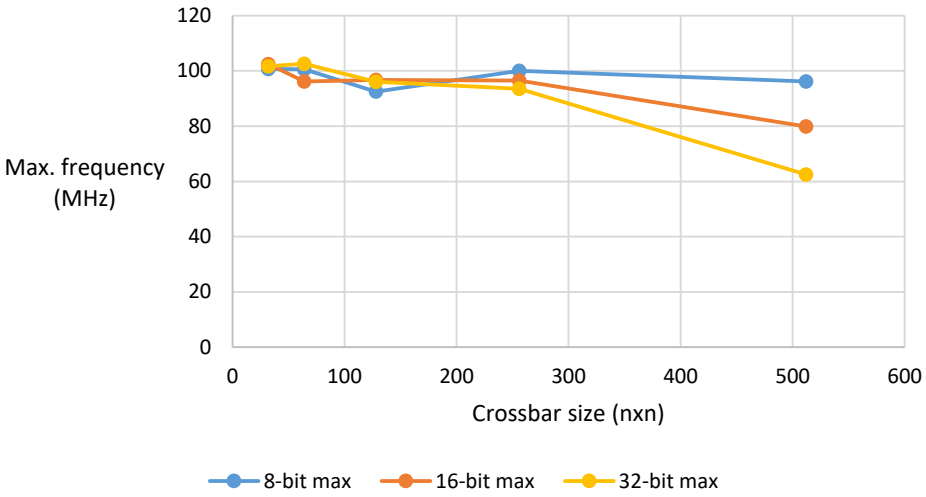
The energy and power consumption of the digital components of the CIM-tile can now be directly compared to the energy and power consumption of the analog components. During the duration of this thesis project, the analog components have been studied as part of the MNEMOSENE project [62], providing an accurate model for the analog components regarding energy and power consumption. Equations eqs. (6.1) to (6.4) are used to calculate the energy and power consumption of the crossbar for reading/computation and writing. R_{rc} represents the resistance value of the cell located at row 'r' and column 'c' in the crossbar. $P_{DIM_{read/write}}$ correspond to the power consumption for reading and writing, respectively. The parameter $activation_{r/c}$ is a binary value indicating whether a row or column has been activated or not. Finally, $V_{(read/write)}$ is the crossbar read or write voltage, and $I_{(write)}$ is the crossbar write current. The values for



(a)



(b)



(c)

Figure 6.1: Implementation results regarding Area (a), power consumption (b) and max frequency (c) vs crossbar sizes for 1 ADC per 32 columns.

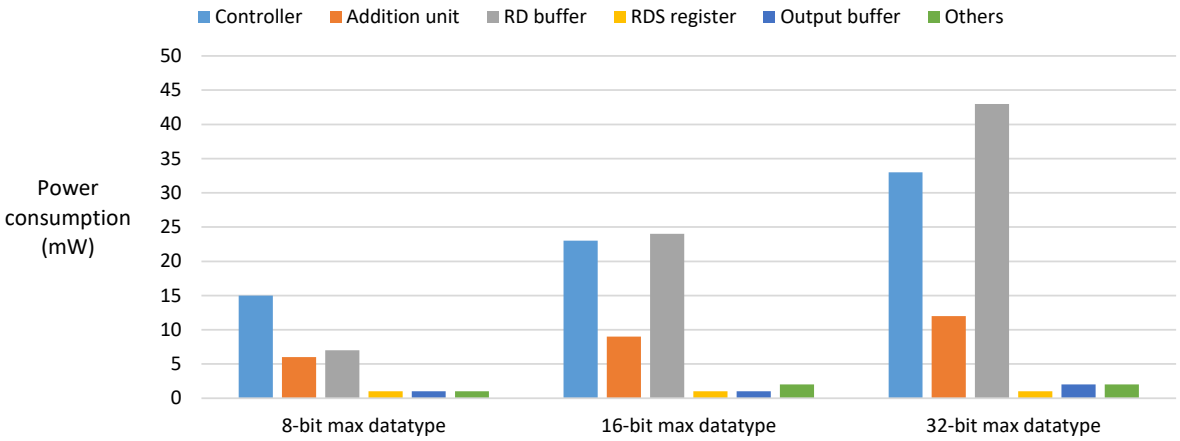


Figure 6.2: The contribution of each of the components to the power consumption.

Component	Parameters	Spec		
		ReRAM	PCM	STT-MRAM
Memristive devices	cell levels	2	2	2
	LRS	5k	20k	5k
	HRS	1M	10M	10K
	read voltage	0.2V	0.2V	0.9V
	write voltage	2V	1V	1.5V
	write current	100 μA	300 μA	200 μA
	read time	10 ns	10 ns	10 ns
	write time	100 ns	100 ns	60 ns
Crossbar	structure	1T1R		
	num. columns	256		
	num. rows	256		
S&H	number	256		
	hold time	9.2 ms		
	latching energy	0.25 pJ		
	latency	0.6 ns		
DIM		read DIM		write DIM
	number	256		256
	power	3.9 μW		3.9 μW
ADC	power	2.6 mW		
	precision	8 bits		
	latency	1.2 GSps		

Table 6.1: Value of parameters used for the computations [62]

the parameters used in the equations can be seen in Table 6.1. The characteristics of the digital and analog components are compared for a 256x256 crossbar and 8 available ADCs.

$$P_{(read,compute)} = \sum_{r=1}^{\#rows} \left(\left[\sum_{c=1}^{\#columns} \left(\frac{V^2_{(read)}}{R_{rc}} \right) + P_{DIM_{read}} \right] * activation_r \right) \tag{6.1}$$

$$P_{(write)} = \sum_{c=1}^{\#columns} \left(\left[V_{(write)} * I_{(write)} + P_{DIM_{write}} \right] * activation_c \right) \tag{6.2}$$

$$E_{(read,compute)} = P_{(read,compute)} * T_{Xbar(read,compute)} \tag{6.3}$$

$$E_{(write)} = P_{(write)} * T_{Xbar(write)} \tag{6.4}$$

Power consumption To compare the power consumption of the analog components and the digital circuitry and retrieve a general power requirement for the CIM-tile, the worst case power consumption of the crossbar array is considered for different technologies. This worst case scenario assumes all rows and columns

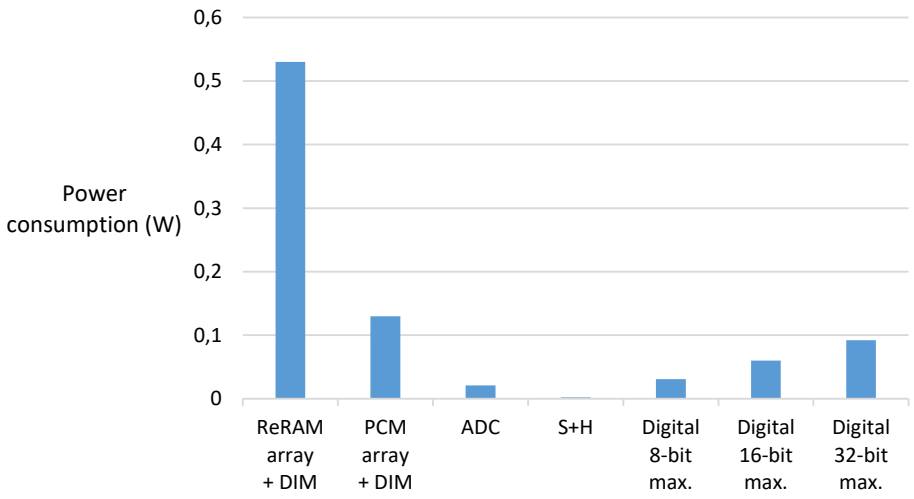


Figure 6.3: Comparison of the power consumption of various analog components and the digital circuitry for different maximum supported datatype sizes.

are activated, and all cells are in their low-resistance state. Figure 6.3 shows a comparison of the power consumption for the analog and digital components of the architecture. The power consumption of an STT-MRAM crossbar array for the read/computation operation is around 20 times larger than the ReRAM crossbar array due to the high read voltage. The STT-MRAM array has been omitted from Figure 6.3 for better readability of the remaining components. In addition to this worst-case scenario, a more typical case for the crossbar and DIM power consumption can be obtained by assuming random data present in the crossbar and RD buffer. In this case, effectively only half of the rows is activated during a computational operation, leading to halving the power consumption. In addition, for ReRAM and PCM technology, 50% of the cells have a negligible contribution to the power consumption due to the high resistance of their HRS leading to again halving the overall power consumption of the array. For STT-MRAM, the power consumption contribution of cells in HRS states is only halved, leading to only a 25% reduction due to the HRS cells. To summarize:

- For ReRAM and PCM technology, the typical power consumption is around 4 times lower than the worst case power consumption portrayed in Figure 6.3. In this typical case, the PCM array contributes less to the total power consumption than the digital circuits for 16-bit and 32-bit maximum datatype size configurations.

- For STT-MRAM technology, the typical power consumption is around 2.7 times lower than the worst case power consumption. In this case, the majority of the power consumption is still due to the analog array.

Energy consumption By embedding Equations eqs. (6.1) to (6.4) into the simulator, accurate numbers for energy consumption taking actual data and analog component activation into account can be obtained. Again, the 'GEMM' benchmark is used to obtain these numbers. Figure 6.4 shows a comparison of the energy consumption of the analog components and the digital circuitry for executing the benchmark at a clock frequency of 90MHz. It can be seen that, while the power consumption of the digital circuitry is a lot lower than that of the crossbar arrays, the energy consumption for running the benchmark is significantly higher. This is due to the fact that the analog components are not activate during the entire time the benchmark program is running. It should be kept in mind that the presented power and energy figures of the digital circuitry are only representative for the actual FPGA implementation of the circuitry. An ASIC implementation of the circuitry may lead to one or even several orders of magnitude lower power consumption [69, 70]. This means that an ASIC implementation of the digital CIM-tile circuitry is expected to be in the same order of magnitude as the analog circuitry regarding these characteristics. Detailed investigation of an ASIC implementation of the design is left for future work.

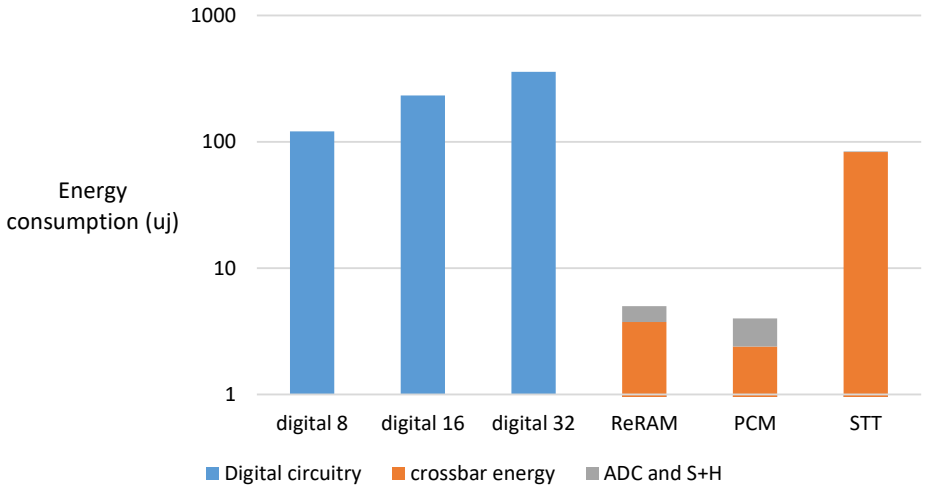


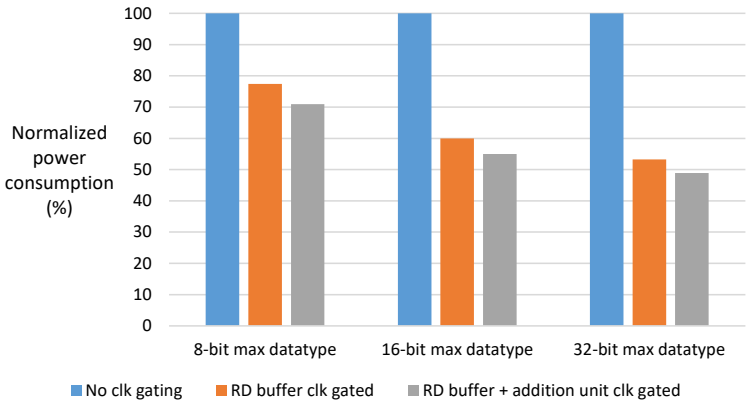
Figure 6.4: Energy comparison of Digital circuitry and analog components for executing the GEMM benchmark @90MHz.

6.3. Possible improvements

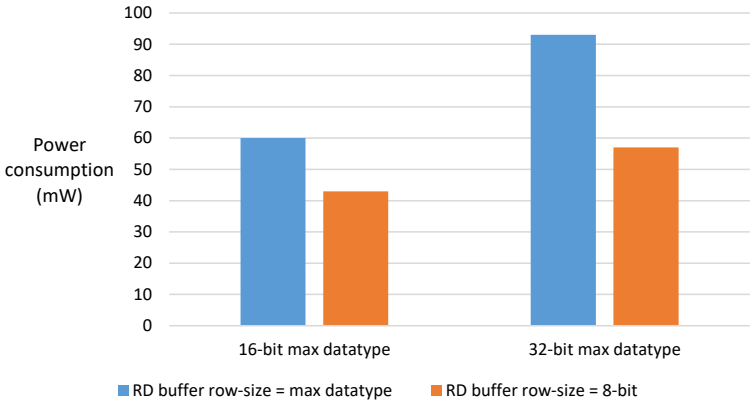
While there are no hard requirements on the power consumption or area for the proof-of-concept version of the CIM-tile implementation delivered by this thesis project, some avenues to explore for potential improvements regarding these characteristics are proposed in this section.

As one of the major contributors to area and power consumption, the RD buffer should be kept in mind for potential improvement when progressing the CIM-architecture. One avenue that may be explored to save dynamic power consumption is clock gating of this buffer. The reported percentage of the the RD-buffer power consumption spent on the clocking network is around 50%, which corresponds to around 23% of the total power consumption of the digital circuitry of the CIM-tile. As the buffer only requires activation during an RDsh instruction or while the outside unit is writing values into the buffer, the clocking network of the buffer may be gated for anywhere between 25% and nearly 100% of the execution time, depending on the application. Note that, in practice, 100% will not occur for any programming containing at least one VMM operation. Clock gating the addition unit may also be explored to save dynamic power consumption, although the theoretical improvement to be made is a lot smaller than for the RD buffer as the addition unit is around a quarter of the size of the RD buffer when supporting 32-bit, while also being activate for a larger percentage of the execution time. Figure 6.5a shows the theoretical improvement obtained by clock gating for different maximum supported datatype sizes. As expected, the improvement is highest for the designs supporting larger datatypes. It should be noted that this improvement does not come for free, as clock gating also leads to some trade-offs. Firstly, extra hardware is required for the gating. However, the enable conditions of the registers may be combined with the clock gating, which saves area. Secondly, the design should be made robust against unwanted triggers due to glitching. As this RD buffer also acts as part of the interface between the tile and the outside unit, this also effects the design of the outside unit. Finally, it should be kept in mind that extra care has to be taken when inserting scan technology for testing the ASIC implementation after fabrication. Investigation of the impact of clock gating the RD buffer on the outside unit and detailed implementation of this clock gating is left for future work.

Another possible area and power consumption improvement avenue that may be explored is sizing down the RD buffer for larger datatypes. As datatypes are assumed to be multiples of bytes in this current implementation, an option would be to size the RD buffer to a single byte, requiring the outside unit to refill the buffer multiple times during the processing of a single set of elements. Figure 6.5b shows the area/power improvement for the the designs supporting 16-bit- and 32-bit max datatypes. A potential drawback of this alteration is the extra requirements imposed on the outside unit, which will have to be investigated during the design of this unit.



(a)



(b)

Figure 6.5: Theoretical power consumption improvements due to clock gating (a) and scaling down the RD-buffer (b)

6.4. Conclusions

In this chapter, the HDL implementation of the proposed design has been discussed, presenting its capabilities and constraints. The HDL platform has been used to explore different configurations of the CIM-tile for relevant parameters, reporting numbers on area requirement, power consumption and latency for each of the configurations. It has been shown that these characteristics scale linearly with the crossbar size, provided the number of ADCs and consequently the addition unit also scale accordingly. This means that, as the energy consumption of the crossbar array increases quadratically when scaling up the array, the contribution of the digital circuitry to the energy consumption drops. The energy- and power consumption of the digital circuitry have been compared to that of the analog components of the architecture for a 256x256 crossbar and 8 avaiable ADCs. While the power consumption of the digital circuitry is smaller than that of the analog components, its contribution to energy consumption is larger due to the analog components not constantly being activated. To improve power- and energy consumption some avenues of improvement have been proposed which allow for 30% up to 50% improvement for the digital circuitry. While there is room for improvement, it can be concluded that the characteristics of the digital circuitry do not impose any road-blocks on progressing the CIM-architecture. In addition, the presented results provide valuable information that may be used in future work to make well-founded decisions on trade-offs such as the number of CIM-tiles versus the individual tile size.

7 Conclusion

This chapter will conclude the thesis report. Section 7.1 provides a summary of the report. Section 7.2 revisits the problem statement and elaborates on the main contributions of this thesis. Finally, Section 7.3 discusses the limitations of the design and work to be done on the project in the near future.

7.1. Summary

Chapter 2 introduced the required prerequisites on in-memory computation. Related works on in-memory computing regarding device level, circuit level and architecture level have been discussed, describing the benefits and drawbacks of each proposed work. In addition, related works on simulation of in-memory computation are discussed.

Chapter 3 introduced the prior works performed on the novel tile architecture proposed throughout this thesis report. An introduction to the different aspects of the architecture as well as the initial nano-instructions has been given, providing the starting point of this thesis project. An initial high-level architecture has been proposed. The architecture has been investigated for possible pipelining configurations. However, the architecture and its accompanying instructions should be revised from a hardware perspective. Once the instruction set architecture has been revised, the effectiveness of the pipelining should be investigated. In addition, a hardware implementation for the proposed efficient addition scheme should be proposed. A simulation platform is provided which can be extended to test the effectiveness of different instruction set- and hardware designs. To summarize, the main challenges this thesis project focuses on are:

- Revisit the ISA from a hardware perspective, also making the new instructions more data-efficient.
- Add a write-verification method to the architecture.
- Propose a detailed hardware design for the CIM-tile.
- The tile should support the unconventional pipelining.
- Propose a way of testing the architecture without access to the analog components.

Chapter 4 provided the detailed design of the newly proposed nano-instruction set architecture, adapting the initial ISA, compiler and simulator to allow for a hardware implementation to support the ISA and architecture. Results regarding the instruction file size have been presented, ensuring that the nano-instruction format no longer imposes unreasonable constraints on instruction memory size. Depending on the architecture configuration, the newly proposed instruction set achieves up to a 98% reduction in instruction file size for a typical program. Finally, results obtained from the adapted version of the simulator have been used to explore the impact of different pipeline configurations on the CIM-tile performance, allowing for a well-founded decision on the pipelining configuration to be implemented in hardware considering the current technology. For this current technology, it is found that the setup stage and execute stage of the pipeline design can be combined without impacting the performance. In addition, the read-stage and addition stage may be combined as the addition stage contributes very little to the cycle latency of the tile, saving hardware and leading to a less complex synchronization requirement for the tile controller. With evolving technology, the design may be extended without any required alteration of the instruction set. An efficient instruction set has thus been proposed, allowing for scaling towards future technology.

In Chapter 5, solutions have been proposed to the hardware challenges for the tile architecture. A buffer structure has been proposed for sourcing and sinking data and the impact on tile performance for different data-types has been explored, ensuring that the buffer structure does not impose large performance degradation on the architecture. Various different buffer structures across the spectrum of performance and efficiency have been proposed for the WD and RD buffer. As a detailed implementation of the outside unit and studying the effect of that unit servicing multiple tiles is left for future work, the buffers structures that will be implemented in the proof-of-concept hardware have been chosen to be a reasonable trade-off between performance, complexity and energy consumption. Furthermore, three implementations of the efficient addition scheme have been proposed, describing the situations in which each of them can be effectively deployed in the tile architecture. The addition scheme version that was implemented imposes no cycle overhead at the cost of restricting the size of the datatypes to be multiples of bytes. If, in future versions of the architecture, it is preferable to have the tile service any random datatype size, the proposed hybrid addition scheme may be implemented in the architecture. Following the decision on the pipeline configuration from Chapter 4, a design for the CIM-tile controller supporting two stage pipelining has been proposed. The synchronization used to enable the pipelining can be easily extended to allow for more pipeline stages once this becomes beneficial for the tile performance with evolving technology. The tile architecture has thus been revisited from a hardware perspective, supporting the proposed instruction set architecture . To allow for functional testing of the design implementation without using actual memristor crossbar arrays and the required analog peripherals, a digital model capable of adequately capturing the memristor crossbar behaviour has been proposed.

Finally, in Chapter 6, the HDL implementation of the proposed design has been discussed, presenting its capabilities and constraints. The HDL platform has been used to explore different configurations of the CIM-tile for relevant parameters, reporting numbers on area requirement, power consumption and latency for each of the configurations. It has been shown that these characteristics scale linearly with the crossbar size, provided the number of ADCs and consequently the addition unit also scale accordingly. This means that, as the energy consumption of the crossbar array increases quadratically when scaling up the array, the contribution of the digital circuitry to the energy consumption drops. The energy- and power consumption of the digital circuitry have been compared to that of the analog components of the architecture for a 256x256 crossbar and 8 avaiable ADCs. While the power consumption of the digital circuitry is smaller than that of the analog components, its contribution to energy consumption is larger due to the analog components not constantly being activated. To improve power- and energy consumption some avenues of improvement have been proposed which allow for 30% up to 50% improvement for the digital circuitry. While there is room for improvement, it can be concluded that the characteristics of the digital circuitry do not impose any roadblocks on progressing the CIM-architecture. In addition, the presented results provide valuable information that may be used in future work to make well-founded decisions on trade-offs such as the number of CIM-tiles versus the individual tile size.

7.2. Contributions

This thesis aimed to demonstrate a sub-part of a novel CIM-architecture as part of the MNEMOSENE project, called the CIM-tile. The design consists of digital peripherals used in conjunction with a digital model of a memristor array and analog peripherals to allow for functional testing of the design. The architecture that was initially proposed for the CIM-tile has been revisited from a hardware perspective, and implementations of each of the architecture components have been proposed. The goal of creating a flexible architecture capable of exploiting the benefits of in-memory computation has been achieved. The main contributions of this project are:

- A new version of the in-memory nano-instructions has been proposed, revisiting the initial instructions from a hardware perspective.
- The CIM-tile simulator has been altered from a hardware perspective and extended to allow for the newly proposed ISA.
- Simulation results have been used to make well-founded decisions regarding restrictions on an initial proof of concept version of the CIM-tile in hardware.
- Implementations for each of the architecture components have been proposed and compared.
- A parametrized HDL platform has been built, allowing for exploration of the design in actual hardware.

– A digital model of a memristor array has been proposed and integrated into the HDL platform to allow for functional testing of the design.

– Figures regarding area, power consumption and latency of the design have been presented.

7.3. Future work

While the goal of this thesis project has been achieved, the development of the overall in-memory computation architecture as part of the MNEMOSENE project is not yet finished. Some restrictions of the HDL platform have been discussed in Section 6.1. Due to hardware platform restrictions and the large area requirement of the digital crossbar model, the design has not been functionally tested for larger configurations. However, the configurations that have been tested still encapsulate all different corner cases regarding the design parameters. One of the main future steps is integrating the proposed digital circuitry and an actual analog memristive crossbar array once the technology is available.

Some proposed steps that will help progress the CIM-architecture are:

- Extend the HDL platform to alleviate restrictions presented in Section 6.1.
- Explore the CIM-tile improvement avenues proposed in Section 6.3.
- Integrate the proposed digital circuitry with actual analog memristive technology.
- Perform exploration of the design for implementation in ASIC, and compare the characteristics to the results obtained for the FPGA implementation described in this thesis.
- Progress the overall CIM architecture by creating a detailed design of the outside unit, taking into account the effect of the unit servicing multiple CIM-tiles. This also entails extending the simulator and HDL platform to perform exploration similar to the one described for the CIM-tile in this thesis report.

The work presented in this thesis report is believed to be of great value to the execution of the listed future works. Different solutions to certain architectural challenges have been proposed for foreseen applications and situations. However, as there are still many uncertainties regarding evolving technology and potential unforeseen situations, it is advised to keep an open mind about the proposed solutions.

A

Initial micro2nano gemm

The nano-instructions shown are the result of compiling the gemm benchmark using 32 8-bit ADCs

Micro instructions for gemm benchmark

```
store &B[0][0] 0 0 240 220 220
MMM &A[0][0] 0 0 200 220 240 240 220
```

Abbreviated set of nano-instructions for gemm benchmark

```
RS + 256 data bits
WD + 256 data bits
WDS + 256 data bits
FS WR
DoA
END
```

repeat above block to write all 240 rows

```
RS + 256 data bits
FS VMM
DoA
DoS
CS + 256 data bits
DoR
CS + 256 data bits
DoR
CS + 256 data bits
DoR
CS + 256 data bits
DoR
CS + 256 data bits
DoR
repeat CS and DoR another 7 times
END
```

repeat above block untill MMM is done, in this case it takes 1600 instances of this block (with occasional LS, IADD, CP instructions)

B

‘gemm’ Benchmark Exploration

#ADCs	ADC bits	file size (MB)	#instructions	#RS	#WD	#WDS	#CS	#FS	#DoA	#DoS	#DoR	#LS	#IADD	#CP	#AS	#CB	#END
8	5	13.75	888041	13040	240	240	409600	13040	13040	12800	409600	1600	1600	200	0	0	13040
8	8	1.74	115241	1840	240	240	51200	1840	1840	1600	51200	1600	1600	200	0	0	1840
32	5	3.80	271641	13040	240	240	102400	13040	13040	12800	102400	1600	1600	200	0	0	13040
32	8	0.50	38441	1840	240	240	12800	1840	1840	1600	12800	1600	1600	200	0	0	1840

Table B.1: Initial instruction file exploration for the gemm benchmark. This table shows individual instruction counts in the gemm benchmark.

#ADCs	ADC bits	file size (MB)	%RS	%WD	%WDS	%CS	%FS	%DoA	%DoS	%DoR	%LS	%IADD	%CP	%AS	%CB	%END
8	5	13.75	1.47	0.03	0.03	46.12	1.47	1.47	1.44	46.12	0.18	0.18	0.02	0	0	1.47
8	8	1.74	1.60	0.21	0.21	44.43	1.60	1.60	1.39	44.43	1.39	1.39	0.17	0	0	1.60
32	5	3.80	4.77	0.09	0.09	37.42	4.77	4.77	4.71	37.42	0.59	0.59	0.07	0	0	4.77
32	8	0.50	4.79	0.62	0.62	33.30	4.79	4.79	4.16	33.30	4.16	4.16	0.52	0	0	4.79

Table B.2: Initial instruction file exploration for the gemm benchmark. This table shows the instructions’ contribution percentage to the total number of instructions.

C

RS/WD/Output buffer results

The impact of the RS, WD and output buffer on the performance of the CiM-tile has been explored and presented in Chapter 5. This appendix presents the same figures when the maximum supported datatype is 16-bit instead of 32-bit. Figures C.1 and C.2 show the execution time and output generation for different datatype sizes, respectively. It can be seen that the trend of the graphs are equal to the ones presented for 32-bit maximum datatype size.

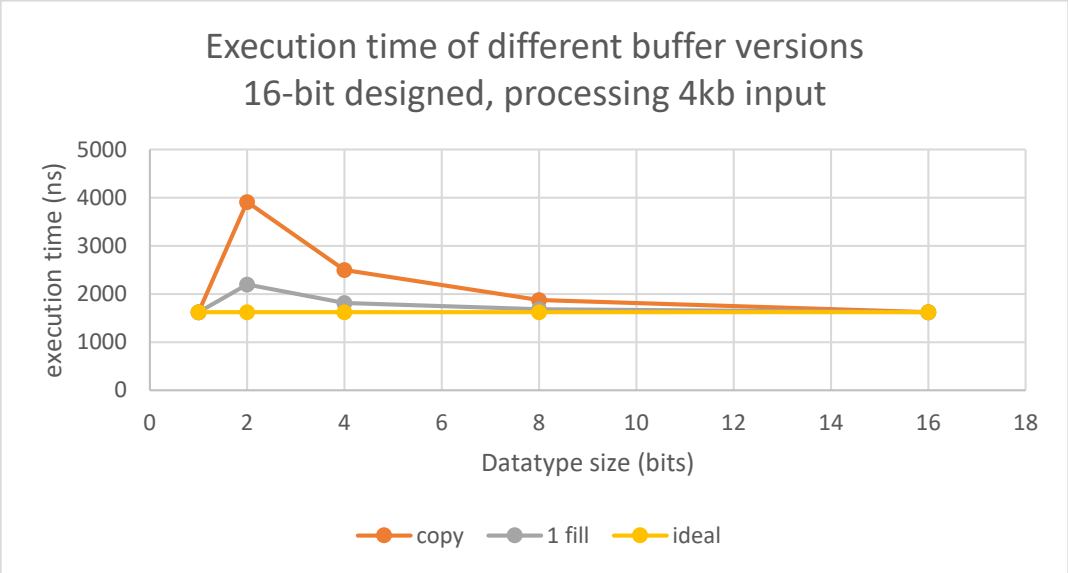


Figure C.1: Impact of time-multiplexing on execution time for different datatype sizes.

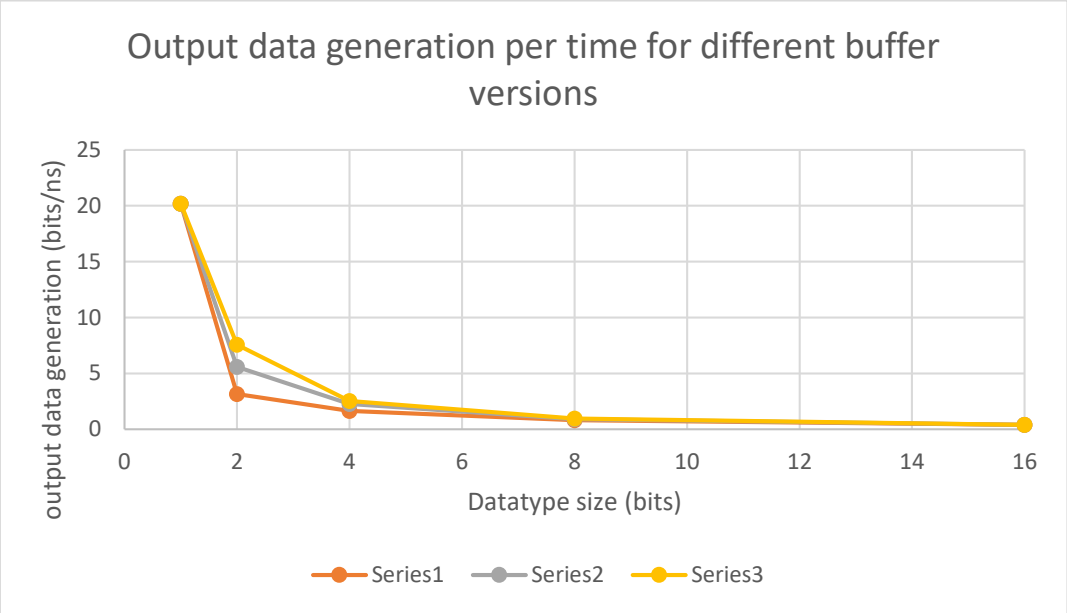


Figure C.2: Output generation per time for different datatype sizes.

Bibliography

[1] J. von Neumann, "First draft of a report on the edvac," June 1945.

[2] M. Godfrey and D. Hendry, "The computer as von neumann planned it," *IEEE Annals of the History of Computing*, vol. 15, pp. 11–21, 1993.

[3] H. Thimbleby, "Modes, wysiwyg and the von neumann bottleneck," *IEE Colloquium on Formal Methods and Human-Computer Interaction*, Feb 1988.

[4] "Mnemosene - computation-in-memory architecture based on resistive devices," 2020, http://www.mnemosene.eu.

[5] L. Chua, "Memristor, the missing circuit element," *IEEE Transactions on Circuit Theory*, vol. 18, no. 5, pp. 507–519, Sep 1971.

[6] S. Yu and P.-Y. Chen, "Emerging memory technologies: Recent trends and prospects," *IEEE Solid-State Circuits Magazine*, vol. 8, no. 2, pp. 43–56, 2016.

[7] Sung Hyun Jo, T. Kumar, S. Narayanan, W. D. Lu, and H. Nazarian, "3d-stackable crossbar resistive memory based on field assisted superlinear threshold (fast) selector," in *2014 IEEE International Electron Devices Meeting*, Dec 2014, pp. 6.7.1–6.7.4.

[8] B. Govoreanu, G. S. Kar, Y. Chen, V. Paraschiv, S. Kubicek, A. Fantini, I. P. Radu, L. Goux, S. Clima, R. Degraeve, N. Jossart, O. Richard, T. Vandeweyer, K. Seo, P. Hendrickx, G. Pourtois, H. Bender, L. Altimime, D. J. Wouters, J. A. Kittl, and M. Jurczak, "10×10nm2 hf/hfox crossbar resistive ram with excellent performance, reliability and low-energy operation," in *2011 International Electron Devices Meeting*, Dec 2011, pp. 31.6.1–31.6.4.

[9] Y. Kim, X. Fong, K.-W. Kwon, M.-C. Chen, and K. Roy, "Multilevel spin-orbit torque mrams," *IEEE Transactions on Electron Devices*, vol. 62, no. 2, pp. 561–568, 2014.

[10] H. Nguyen, J. Yu, L. Xie, M. Taouil, S. Hamdioui, and D. Fey, "Memristive devices for computing: Beyond cmos and beyond von neumann," *International conference on VLSI-SoC*, October 2017.

[11] H. Lee, P. Chen, T. Wu, Y. Checn, C. Wang, P. Tzeng, C. Lin, F. Checn, C. Lien, and M. Tsai, "Low power and high speed bipolar switching with a thin reactive ti buffer layer in robust hfo2 based rram," *IEEE International Electron Devices Meeting*, December 2008.

[12] S. Hsu and W. Zhuang, "Electrically programmable resistance cross point memory," June 2001.

[13] M. Lee and I. Baek, "Methods of programming non-volatile memory devices including transition metal oxide layer as data storage material layer and devices so operated," October 2004.

[14] K. Campbell, J. Moore, and T. Gilton, "Single-polarity programmable resistance-variable memory element," August 2002.

[15] M. Lanza, "A review on resistive switching in high-k dielectrics: A nanoscale point of view using conductive atomic force microscope," *Materials*, pp. 2155–2182, 2014.

[16] W. HokenMaier, D. Labrecque, R. Jurasek, V. Butler, C. Scoville, A. Willey, S. Loeffler, Y. Li, and S. Sharma, "A 90nm 32-mb phase change memory with flash spi compatibility," *IMW*, May 2014.

[17] R. Bez, P. Cappelletti, G. Servalli, and A. Pirovano, "Phase change memories have taken the field," *IEEE International Memory Workshop*, May 2013.

[18] L. Xu, Y. Xie, and Y. Lin, "High-reliable multi-level phase change memory with bipolar selectors," *IEEE International Conference on ASIC*, pp. 1011–1014, October 2009.

[19] S. Lai, "Current status of the phase change memory and its future," *IEEE International Electron Devices Meeting*, pp. 255–258, December 2003.

[20] S. Bhatti, R. Sbiaa, A. Hirohat, H. Ohno, S. Fukami, and S. Piramanayagam, "Spintronics based random access memory: a review," *Materials today*, vol. 20, no. 9, pp. 530–548, 2017.

[21] X. Guo, E. Ipek, and T. soyata, "Resistive computation: avoiding the power wall with low-leakage, stt-mram based computing," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, 2010.

[22] K. Higuchi, K. Miyaji, K. Johguchi, and K. Takeuchi, "50 nm hfo2 reram with 50-times endurance enhancement by set/reset turnback pulse & verify scheme," *ICSSDM*, pp. 1001–1002, 2011.

[23] Z. Wei, Y. Kanzawa, K. Arita, Y. Katoh, K. Kawai, S. Maraoka, S. Mitani, S. Fujii, K. Katamayama, M. Iijima, T. Mikawa, T. Ninomiya, R. Miyanaga, Y. Kawashima, K. Tsuji, A. Himeno, T. Okada, R. Azuma, K. Shimakawa, H. Sugaya, T. Takagi, R. Yasuhara, K. Horiba, H. Kumigashira, and M. Oshima, "Highly reliable taox reram and direct evidence of redox reaction mechanism," *IEEE International Electron Devices Meeting*, December 2008.

[24] D. Strukov, G. Snider, D. Stewart, and R. Williams, "The missing memristor found," *Nature*, pp. 453–459, May 2008.

[25] R. Bez, "Chalcogenide pcm: a memory technology for next decade," *IEEE International Electron Devices Meeting*, pp. 89–92, December 2009.

[26] L. Chua, "Resistance switching memories are memristors," *Applied Physics A*, vol. 102, no. 4, pp. 765–783, March 2011.

[27] P. Meuffels and R. Soni, "Fundamental issues and problems in the realization of memristors," July 2012.

[28] M. Lebdeh, U. Reinsalu, H. Nguyen, S. Wong, and S. Hamdioui, "Memristive device based circuits for computation-in-memory architectures," *ISCAS*, 2019.

[29] S. Hamdioui, L. Xie, H. Nguyen, M. Taouil, K. Bertels, H. Corporaal, and H. Jiao, "Memristor based computation-in-memory architecture for data-intensive applications," *Design, Automation And Test in Europe*, pp. 1718–1725, 2015.

[30] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. Strachan, M. Hu, R. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analo arithmetic in crossbars," *ACM/IEEE ISCA*, June 2016.

[31] S. Kvatinsky, N. Wald, G. Satat, A. Kolodny, and U. Weiser, "Mrl - memristor ratioed logic," *International Workshop on CNNA*, October 2012.

[32] G. Rose, J. Rajendran, H. Manem, R. Karri, and R. Pino, "Leveraging memristive systems in the construction of digital logic circuits," *IEEE*, vol. 100, pp. 2033–2049, 2012.

[33] J. Borghetti, Z. Li, J. Straznicky, X. Li, D. Ohlberg, W. Wu, D. Stewart, and S. Williams, "A hybrid nanomemristor/transistor logic circuit capable of self programming," *PNAS*, vol. 106, no. 6, pp. 1699–1703, February 2009.

[34] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," *ACM/EDAC/IEEE DAC*, vol. 53, June 2016.

[35] L. Xie, H. Nguyen, J. Yu, A. Kaichouhi, M. Taouil, M. AlFailakawi, and S. Hamdioui, "Scouting logic: A novel memristor-based logic design for resistive computing," *IEEE ISVLSI*, pp. 177–181, 2017.

[36] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Magic—memristor-aided logic," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 11, pp. 895–899, Nov 2014.

[37] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Memristor-based material implication (imply) logic: Design principles and methodologies," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 10, pp. 2054–2066, Oct 2014.

[38] M. Hu, H. Li, Q. Wu, and G. Rose, "Hardware realization of bsb recall function using memristor crossbar arrays," *ACM/EDAC/IEEE Design Automation Conference*, pp. 498–503, 2012.

[39] D. Bhattacharjee, R. Devadoss, and A. Chattopadhyay, "Revamp: Reram based vliw architecture for in-memory computing," *Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE*, pp. 782–787, 2017.

[40] P.-E. Gaillardon, L. Amar, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. De Micheli, "The programmable logic-in-memory (plim) computer," *Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE*, p. 427–432, 2016.

[41] R. Ben Hur and S. Kvatinsky, "Memristive memory processing unit (mpu) controller for in-memory processing," *ICSEE*, pp. 1–5, 2016.

[42] A. Siemon, S. Menzel, R. Waser, and E. Linn, "A complementary resistive switch-based crossbar array adder," *IEEE journal on emerging and selected topics in circuits and systems*, vol. 5, no. 1, pp. 64–74, 2015.

[43] D. Fujiki, S. Mahlke, and R. Das, "In-memory data parallel processor," *In Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems. ACM*, pp. 1–14, 2018.

[44] C. Ping, L. Shuangchen, X. Cong, T. Zhang, Z. Jishen, L. Yongpan, W. Yu, and X. Yuan, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," *ACM/IEEE*, pp. 27–39, 2016.

[45] L. Yavits, S. Kvatinsky, A. Morad, and R. Ginosar, "Resistive associative processor," *Computer Architecture Letters (CAL)*, 2015.

[46] J. Yu, L. Xie, M. Taouil, and S. Hamdioui, "Memristive devices for computation-in-memory," *in Design, Automation and Test in Europe (DATE)*, 2018.

[47] A. Subramaniyan, J. Wang, E. Balasubramanian, D. Blaauw, D. Sylvester, and R. Das, "Cache automation," *In Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 259–272, 2017.

[48] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," *arXiv preprint*, 2018.

[49] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," *In High Performance Computer Architecture (HPCA)*, pp. 481–492, 2017.

[50] T. Finkbeiner, G. Hush, T. Larsen, P. Lea, J. Leidel, and T. Manning, "In-memory intelligence," *IEEE micro*, vol. 37, no. 4, pp. 30–38, 2017.

[51] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. Kozuch, O. Mutlu, P. Gibbons, and T. Mowry, "Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology," *In Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 273–287, 2017.

[52] S. Li, D. Niu, K. Malladi, H. Zheng, B. Brennan, and Y. Xie, "Drisa: A dram-based reconfigurable in-situ accelerator," *In Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 288–301, 2017.

[53] S. J. E. Wilton and N. P. Jouppi, "Cacti: an enhanced cache access and cycle time model," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 5, pp. 677–688, May 1996.

[54] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 7, pp. 994–1007, July 2012.

[55] M. Poremba and Y. Xie, "Nvmain: An architectural-level main memory simulator for emerging non-volatile memories," in *2012 IEEE Computer Society Annual Symposium on VLSI*, Aug 2012, pp. 392–397.

[56] L. Xia, B. Li, T. Tang, P. Gu, P. Chen, S. Yu, Y. Cao, Y. Wang, Y. Xie, and H. Yang, "Mnsim: Simulation platform for memristor-based neuromorphic computing system," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 5, pp. 1009–1022, May 2018.

[57] L. Nagel and D. Pederson, "spice (simulation program with integrated circuit emphasis)," *Memo ERL-M382, University of California, Berkeley*, April 1973.

[58] M. Lee, Y. Cui, T. Somu, T. Luo, J. Zhou, W. Tang, W. Wong, and R. Goh, "A system-level simulator for rram-based neuromorphic computing chips," *ACS Transactions on Architecture and Code Optimization (TACO)*, vol. 15, no. 4, p. 64, 2019.

[59] F. Clegg and E. McCluskey, "Algebraic properties of faults in logic networks," *Stanford University*, 1970.

[60] S. Ambrogio, S. Balatti, A. Cubeta, A. Calderoni, N. Ramaswamy, and D. Ielmini, "Statistical fluctuations in hfox resistive-switching memory: Part ii—random telegraph noise," *IEEE Transactions on Electron Devices*, vol. 61, no. 8, pp. 2920–2927, Aug 2014.

[61] S. Ambrogio, S. Balatti, A. Cubeta, A. Calderoni, N. Ramaswamy, and D. Ielmini, "Statistical fluctuations in hfox resistive-switching memory: Part i - set/reset variability," *IEEE Transactions on Electron Devices*, vol. 61, no. 8, pp. 2912–2919, Aug 2014.

[62] M. Zahedi, M. Lebdeh, S. Wong, and S. Hamdioui, "Mnemosene: Tile organization and simulator for memristor-based computation in-memory," *submitted to MICRO, to be published*, 2020.

[63] M. Zahedi, "Efficient organization of digital perphery to support integer datatype for memristor-based cim," *To be published*, 2020.

[64] K. Einwich, "Introduction to the systemc ams extension standard," *IEEE international symposium on DDECS*, April 2011.

[65] T. Wieman, C. Schröder, B. Bhattacharya, and T. Vanthournout, Band Jeremiassen, "An overview of open systemc initiative standards development," *IEEE Design & Ttest of Computers*, vol. 29, pp. 14–22, April 2012.

[66] D. O'loughlin, A. Coffey, F. Callaly, D. Lyons, and F. Morgan, "Xilinx vivado high level synthesis: Case studies," *ISSC/CIIT*, June 2014.

[67] Delft Univeristy of Technology and Eindhoven University of Technology, "Initial macro cim architecture and cim-isa," December 2018.

[68] C. Price, *MIPS IV instruction set*. MIPS Technologies inc., 1995.

[69] I. Kuon and J. Rose, "Measuring the gap between fpgas and asics," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, 2007.

[70] A. Amara, F. Amiel, and T. Ea, "Fpga vs. asic for low power applications," *Microelectronics Journal*, vol. 37, pp. 669–677, 2006.

This page is intentionally left blank

CIM-tile execution model and hardware implementation

NOTE: This is a draft for a paper proposing our CIM-tile architecture. The final contents of the paper depend on whether the previous paper gets accepted. Also some additional work outside the scope of this thesis may be included such as showing ASIC-implementation results. As such, the abstract and conclusion have been left intentionally blank for now.

***Abstract*—TO BE WRITTEN**

I. INTRODUCTION

Conventional Von Neumann machines inherently separate the processing units from the memory units. This architecture thus requires that data is transferred from the memory units to the processing units for performing computation, and results that should be stored are required to be transferred back to the memory units. Fast technological advances for processing speed have led to processors out-growing the speed at which data can be retrieved from the memory, meaning the bandwidth between the memory and the processor starts to bottleneck the system. This phenomenon is known as the Von Neumann bottleneck. Modern computer architectures utilize for example hierarchical memory architectures, pre-fetching schemes, and parallel computing to alleviate the Von Neumann bottleneck. However, these methods do not succeed in fully removing the bottleneck. Furthermore, the energy required to retrieve data from the memory is several orders of magnitude higher than the energy required for a single operation within the processor. There is a clear need for a new computing paradigm to further progress modern computer architectures.

One such paradigm is in-memory computing. In-memory computing combats the drawbacks of the Von Neumann architecture by bringing the processing and memory units as close as possible to each other or even combining them into a single unit. Research has shown that certain operations such as Boolean operations and the dot-product are especially susceptible to acceleration using specialized memory arrays, achieving great levels of parallelism and significantly reducing data-traffic and consequently energy consumption. The memory arrays exploit the characteristics of memristive technology to enable the in-memory computation. As Boolean operations and the dot-product can be efficiently supported, specific applications such as data-analytics, signal processing and machine learning can be accelerated using in-memory computing.

The research field is still in an early state and many research projects have been proposed in recent years. One of these projects is called MNEMOSENE [1]. In previous works [2], an execution model for a novel CIM-tile has been proposed. This execution model had been designed with simulation in mind and should still be revisited from a hardware perspective. The instruction set requires alteration to attain hardware-feasibility. In addition, hardware design and implementations of the components present in the architecture should still be presented.

The contributions of this paper are the following:

- We propose a CIM-tile architecture and execution model based on our previous model [2].
- We develop our fully parameterized simulator and compiler to support the refined architecture and in-memory instructions.
- Our proposed CIM-tile architecture synthesised and implemented in FPGA and ASIC for evaluation.
- we provide a comparison in terms of area, energy, and latency for digital and analog components which gives a good insight to designers working in other levels.

II. RELATED WORKS

This section provides preliminary knowledge on memristive devices and their deployment in computational circuits. In addition, architectures utilizing in-memory computation are discussed. Finally, different simulation environments for memristive architectures are discussed.

A. Memristive devices

Memristive devices maintain a relationship between charge and flux elements of a two-terminal passive component. Together with resistors, capacitors and inductors, the memristor as the fourth basic circuit element completes the set of relations between four fundamental elements; voltage, current, charge and flux. The current-voltage characteristic of a memristive device is a pinched hysteresis loop, as can be seen in Fig 1(a). This means the memristive device can be in either of two stable states; the high resistive state(R_H) or the low-resistive state(R_L). Furthermore, the devices have the ability of storing their internal state, effectively remembering its history. In addition to its non-volatility, some large benefits of memristor technologies are their overall great scalability, high memory-density and low leakage power consumption [3]–[5]. By applying a sufficiently high positive voltage, the memristive device can be switched from its R_H state to its R_L state. Now, the device state represents a logic '1'. In the same way, applying a sufficiently large negative voltage will lead

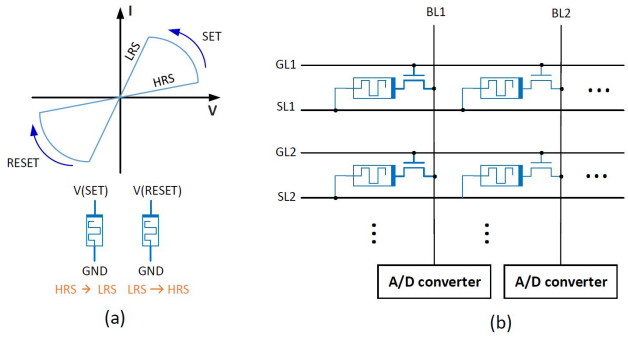


Fig. 1: Memristor behavior (a) and 1T1R crossbar structure (b).

to the device switching from its R_L state to its R_H state, now representing a logic '0'. Using the memristive device in this way allows for binary data storage. However, if the memristive device can be programmed to more than 2 resistance levels, the device can be used to represent more states. In this way, the device can be used to store data consisting of more than one bit [3], [6].

Examples of such technologies are resistive random access memory (ReRAM) [7]–[11], phase-change-memories (PCMs) [12]–[15] and spin-torque-transfer MRAM (STT-MRAM) [16], [17].

B. Memrsistor-based computation

The main focus on memristor-based circuits lies within the use of memristive crossbar array structures, allowing for effective computation of dot product operations [18]–[20]. Figure 1(b) portrays the use of memristors in a crossbar array structure. By changing the resistance of each memristive cell within the crossbar array, these memristive memory arrays can exploit Kirchoff and Ohms laws to allow for in-memory computation.

There are multiple metrics on which the memristor-based computation circuits can be classified. One of these metrics is the location of the results after computation. If this computation happens directly within the memory core, consisting of the memory array and peripheral circuits, two classifications can be distinguished; CIM-Array (CIM-A) class computation circuits perform the computation directly within the memory array, while CIM-Periphery (CIM-P) class computation circuits use the array only to store the data and perform the computation in peripheral circuit. Some advantages and disadvantages for both classes are given:

- **CIM-A:** As in this design class both the data storage and computation happen within the memory array, some significant considerations will have to be made for its design. Multiple values can be written at the same time, meaning that the array should be designed for higher currents than the standard write current. This results in e.g. wider data-lines to adequately accommodate these currents. Another disadvantage is the reduced life-time of the memristive devices as their state is changed more frequently as a result of the in-array computation with respect to only changing their state for storing data. However, a distinct advantage of this design class is its ability of using the entire computation-storage bandwidth as both storage and computation happen within the actual memory array.
- **CIM-P:** The CIM-P class has some distinct advantages and disadvantages with regards to the CIM-A class. For example, as the CIM-P class performs its computation outside of the memory-array, this resulting data has to be fed back into the memory-array if cascaded computation is desired. However, using the memory-array only for data storage comes with some significant advantages. The controller can be simplified as no complex chain of states is required to allow for the different control voltages during computation. Instead, a single set of control voltages can be applied to the memory-array depending on its desired function (e.g. writing to the array, reading from the array etc.). In addition, as the maximum current that will occur in this design class is the write current for writing a single value on each bit-line, the memory array does not require design alteration to allow for the excessively high currents that occur in the CIM-A design class.

C. In-memory architectures

Considering the CIM-A and CIM-P design classes, a multitude of architecture designs have been proposed [19]–[36].

ReVAMP is the first ReRAM-crossbar based general purpose computing architecture that exploits the crossbar array potential for parallel computing, proposed by Bhattacharjee et al. [21]. In this architecture. Very Large Instruction Words (VLIW) are used to encapsulate multiple computations in a single instruction. The computation in the VLIW can then be executed in parallel. PRIME is a Processing-in-memory architecture for neural network computation in ReRAM-based main memory, proposed by Chi et al. [27]. In contrast to ReVAMP, PRIME is an example of a CIM-P class architecture, performing its computation at in the peripheral circuits at read-out rather than in the memory array itself. The Prime architecture is mainly focused on improving the performance and energy efficiency of NN applications. Another architecture using CIM-P class approaches to CNN acceleration is ISAAC, proposed by Shafiee et al. [20]. This architecture focuses on the implementation of an inter- and intra-layer pipeline while also proposing solutions for challenges such as the handling of signed arithmetic and achieving high throughput.

D. In-memory Simulators

As the architecture design for these architectures consist of many factors influencing the system behaviour, simulators are used to be able to provide accurate results regarding performance and energy usage. A tool called CACTI [37] which is able to provide numbers regarding memory access time, energy and area of non-volatile memories has been around since 2001. Based on this software, some other memory-oriented simulators have been developed like NVSim [38] and NVmain [39].

NVSim provides a circuit-level performance, energy and area model targeting emerging non-volatile memories such as STT-RAM, PCRAM and ReRAM. This tool allows the user to tune circuit parameters such as the array organisation and peripheral circuitry to explore different design options. NVMain, on the other hand, provides a cycle-accurate memory simulator on architectural level. NVMain can be used in conjunction with results obtained from CACTI or NVSim. However, the aforementioned simulators do not allow for cycle-accurate simulation capable of tracking control signals and data present in any of the tile components. This also affects the accuracy of obtained characteristics such as energy consumption.

III. Overall CIM-tile Architecture

This section discussed the extended CIM-tile architecture, based of the previously proposed architecture. First, the changes made to the ISA are presented. Then, the hardware architecture is discussed. Finally, the simulation environment used for initial testing the architecture and obtaining results regarding the power- and energy consumption is discussed.

A. ISA

The two main reasons for revisiting the ISA are the following:

- Attain hardware feasibility considering constraints such as communication bandwidth.
- Achieve a more data-efficient ISA.

In addition, as previous works have presented error-rates due to write variability in memristive cells [40], some write verification scheme may be implemented to ensure data-correctness. While a large portion of the ISA may remain unchanged, the row selection, write-data, write-data selection and column selection instructions should be revisited. Actual data can no longer be embedded into the instructions as this data is not necessarily available at compile time and these instructions' implementations are not efficient. For now, it is assumed the data is available in some buffer. This buffer structure will be discussed in more detail in Section III-B. To achieve the goals regarding the revisited ISA, the following changes to these instructions are proposed:

Row selection: Instead of the row selection passing actual data for MMM, it now passes a mask, similar to the write-data mask. Immediate row selection is still allowed for read, write and logical operations. In addition, once a mask has been put in the RS register, row selection instructions only have to be executed once the mask actually changes.

Write data: Embedded data is removed from the instruction. Instead, the instruction now only signals that data present in the input buffer has to be moved to the actual WD register.

Write data selection: Similarly to the row-selection, unnecessary duplicates are omitted from the instruction file at compile time.

Column selection: This is now performed by passing a global index which indicates which of the ADCs input columns should be selected by the analog multiplexer. In addition, an activation bit per ADC is used to allow for deactivating an ADC is a certain column should not be read.

Jump instructions: As the read stage often performs identical sets of instructions (e.g. when performing a MMM, the same set of columns has to be read for every dot-product operation), a jump instruction is introduced to save a large portion of the instruction file size. Similarly to MIPS [41], jump-and-link (jal) and jump-register (jr) instructions are introduced, allowing for re-using the same block of instructions for every read.

Block-wise register filling: Instead of having a single instruction for filling the registers, a set of multiple instructions may offer a more data-efficient approach. For example, by splitting the the row-selection register into smaller blocks, the register can be efficiently filled with a mask by first setting the entire register to all 0's or 1's, and then changing only the incorrect blocks. Using this approach, it is not necessary to use a bit for every single row when filling the register, unless all blocks contain 1's and 0's simultaneously. To allow for this approach, the RDS instruction is changed for a set of three instructions, Row-Data-Select-clear (RDSc), Row-Data-Select-set (RDSs) and Row-Data-Select-block (RDSb). The set and clear instructions are used to set the register to all 0's or all 1's, respectively. The RDSb instruction is used to change the contents of one of the blocks to any arbitrary mask that is passed by the instruction. To select the proper block, the RDSb instruction also passes some index bits. This principle also applies to the WDS and WD instructions. For the WD instructions, however, the set and clear instructions are not required due to the WDS register masking the write-data.

Write verification: To allow for write verification, the ISA should be able to branch back to the write instructions if the output read from the crossbar does not match the contents of the WD register. To this end, a Branch-Not-Equal instruction is introduced to the ISA. If the write verify returns false, this instruction branches both of the program counters to the PC stored in their branch register.

Considering all the aforementioned changes to the ISA, we end up with the new ISA portrayed in Table I. Compilation results have shown up to 98% reduction in file size with respect to the previous ISA when compiling a generic benchmark consisting of storing data in the crossbar and performing MMM.

B. Architecture

The improved architecture presented throughout this section can be seen in Figure 3. As the tile receives its data from some outside unit, it is preferred to define a clear interface between the 'outside' and the tile. To this end, some buffers can be introduced to the tile architecture. The same holds for the output generated by the tile, which should be processed by the outside unit. A total of three buffers are introduced to the tile architecture; the Write-Data (WD) buffer, Row-Data (RD) buffer and output buffer.

WD buffer: In addition to serving as interface, this buffer is also used to alleviate the timing constraints of presenting data

Opcode	Op 1	Op 2	Function description
RDSb	Index	Mask	Place 'Mask' into RDS reg at 'Index'
RDSc			Clear the entire RDS register
RDSs			Set the entire RDS register
RDsh			Shift RD buffer contents
WDb	Index		Copy data to WD reg. at 'Index'
WDSb	Index	Mask	Place 'Mask' into WDS reg at 'Index'
WDSc			Clear the entire WDS register
WDSc			Set the entire WDS register
FS	Function		Select crossbar functionality
DoA			Activate the crossbar function
DoS			Sample the crossbar output
CS	Index	Activation	Select column to be read by ADC
DoR			Activate the ADC
jal	Address		Jump to 'Address' and store PC.
jr			Jump to the PC stored in return reg.
BNE			Branch to PC stored in branch reg.
LS			Indicate the last section of column reads
IADD			Activate third stage of the addition unit
CP			Copy result per ADC to output buffer
AS	Selection		Select adders for addition between ADCs
CB			Copy addition between ADC to output

TABLE I: Overview of the newly proposed ISA.

to the CIM-tile when the bandwidth is insufficient to transfer all required data in one shot. The buffer has been sized such that an entire row of data can be stored inside it, meaning that during the entirety of the write operation, the next set of data can be stored in the buffer.

RD buffer: As the data should be fed into the crossbar bit-by-bit from LSB to MSB during a MMM operation, this buffer has been implemented using a PISO register per crossbar row. By having each of the registers sized to fit the maximum supported datatype size (e.g. 32-bit), the buffer only has to be filled once for each of the element dot-product operations.

Output buffer: This buffer is used only for temporarily storing the result data until the data has been processed or stored elsewhere. As the output of the crossbar should be stored in this buffer, its size determines the amount of parallel computation that can be performed.

The initial design exploration performed in [2] has shown that, for current technology, a two stage pipeline configuration will lead to significant performance increase while a three or four stage pipeline only leads to a small additional increase. For this reason, a two stage pipeline has been implemented. As this pipeline, unlike conventional pipelines, requires simultaneous execution of instructions in each of the stages, a novel controller design is proposed to support the architecture.

The controller top-level design can be seen in Figure 2. It consists of two decoders to execute two instructions simultaneously. Each of the decoders is dedicated to one of the pipeline stages, meaning that they only have to support the instructions associated with their respective stage. To ensure correct program execution, some synchronization between the two stage decoders is required. This is done by monitoring the instructions executed in each of the stages and stalling stages when necessary. In addition, the state of the analog components is monitored, stalling execution when e.g. the crossbar has not yet finished its current operation. Finally, the controller monitors the states of each of the buffers, setting up flags to the 'outside' when either the input buffers should be filled or the output buffer should be read.

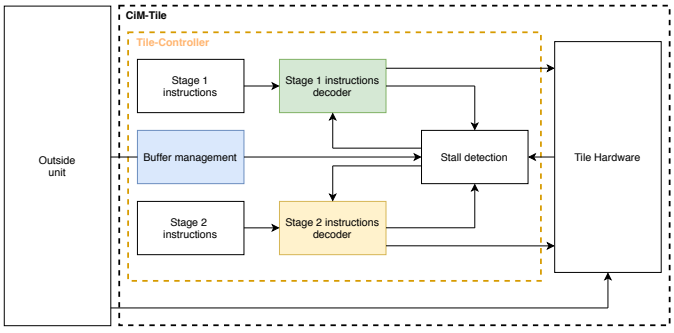


Fig. 2: The CIM-tile controller for supporting 2-stage pipelining.

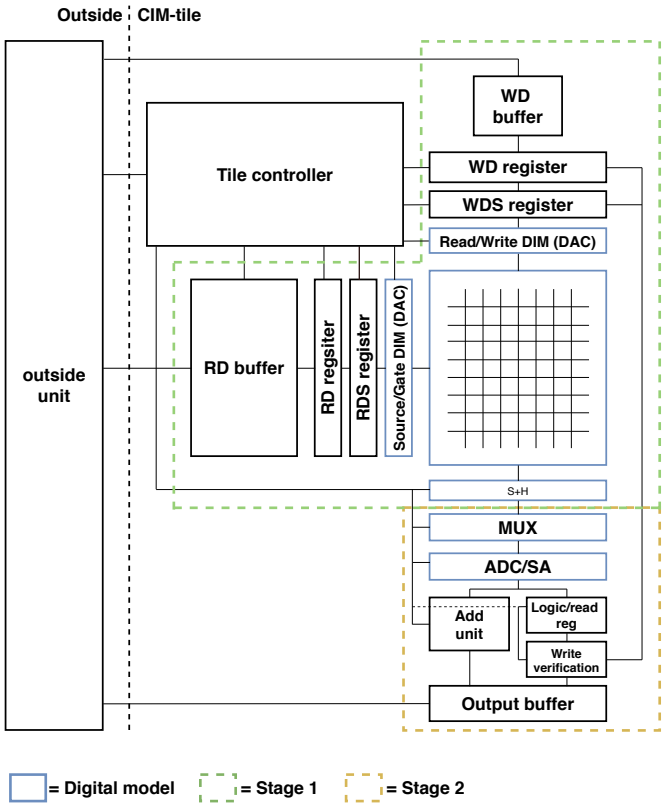


Fig. 3: CIM-tile architecture and pipeline configuration

C. Simulation

To be able to explore different concepts of the CIM-tile architecture, a cycle-accurate CIM-tile simulator has been developed. This simulator is written in systemC to be able to accurately simulate hardware [42], [43]. The simulator provides a basis which can be altered and expanded upon to allow for initial testing and exploring of solutions proposed throughout this paper. The simulator provides numbers for performance and energy consumption and also allows for tracking of the CIM-tile control signals. Equations eqs. (1) to (4) are used to calculate the energy and power consumption of the crossbar for reading/computation and writing. R_{rc} represents the resistance value of the cell located at row 'r' and column 'c' in the crossbar. $P_{DIM_{read/write}}$ correspond to the power consumption for reading and writing, respectively. The parameter $activation_{r/c}$ is a binary value indicating whether a

Component	Parameters	Spec		
		ReRAM	PCM	STT-MRAM
Memristive devices	cell levels	2	2	2
	LRS	5k	20k	5k
	HRS	1M	10M	10K
	read voltage	0.2V	0.2V	0.9V
	write voltage	2V	1V	1.5V
	write current	100 μA	300 μA	200 μA
	read time	10 ns	10 ns	10 ns
	write time	100 ns	100 ns	60 ns
Crossbar	structure	1T1R		
	num. columns	256		
	num. rows	256		
S&H	number	256		
	hold time	9.2 ms		
	latching energy	0.25 pJ		
	latency	0.6 ns		
DIM		read DIM		write DIM
	number	256		256
	power	3.9 μW		3.9 μW
ADC	power	2.6 mW		
	precision	8 bits		
	latency	1.2 GSps		

TABLE II: Value of parameters used for the computations [2]

row or column has been activated or not. Finally, $V_{(read/write)}$ is the crossbar read or write voltage, and $I_{(write)}$ is the crossbar write current. The values for the parameters used in the equations can be seen in Table II. The analog components have been studied as part of the MNEMOSENE project [2], providing an accurate model for the analog components regarding energy and power consumption.

$$P_{(read,compute)} = \sum_{r=1}^{\#rows} ([\sum_{c=1}^{\#columns} (\frac{V_{(read)}^2}{R_{rc}}) + P_{DIM_{read}}] * activation_r) \quad (1)$$

$$P_{(write)} = \sum_{c=1}^{\#columns} ([V_{(write)} * I_{(write)} + P_{DIM_{write}}] * activation_c) \quad (2)$$

$$E_{(read,compute)} = P_{(read,compute)} * T_{Xbar(read,compute)} \quad (3)$$

$$E_{(write)} = P_{(write)} * T_{Xbar(write)} \quad (4)$$

The main benefit of this new simulator with respect to the previously mentioned simulators is the ability to execute the applications thanks to the detailed execution model and in-memory instructions. Due to this benefit and considering the aforementioned equations, more accurate execution time as well as data-dependent energy number are computed. However, the MNEMOSENE simulator is not yet as advanced as the neuromorphic chip simulator with regards to modeling dynamic effects such as write variability and random noise. A comparison of the different simulators is given in Table III.

IV. Evaluation and discussion

In this section, the proposed architecture will be evaluated. First, the implementation platform will be discussed. Then, the characteristics obtained from synthesis tools is presented. Finally, the characteristics of the digital circuits are compared to that of the analog components.

	Simulation speed	power/energy accuracy	Defect modeling
MNSIM	+	0	0
Neuromorphic	0	+	+
MNEMOSENE	0	++	0

TABLE III: Comparison of the various in-memory computing simulators.

A. HDL platform and test Setup

The compiler and simulator allow for swiftly changing many design parameters such as the crossbar size, number of available ADCs and many more. This simulator does, however, not provide insight in the energy- and power consumption of the digital circuitry present in the architecture. To obtain results regarding these parameters, an HDL implementation of the architecture is built. Having a direct implementation without being able to change configuration parameters like in simluation would, however, not allow for comparison between different configurations in terms of area, power consumption and latency. To this end, The HDL has been written in a parametrized way similar to the simulator. The parameters that can be directly changed are:

1) Crossbar size.
2) Maximum supported datatype size.
3) Number of available ADCs.
4) Block-size for the block-wise masking registers filling.
5) Bandwidth for WD data communication from the outside unit.

Parameters 1, 2 and 3 have direct impact on the sizing of some of the major components of the architecture. To elaborate, the crossbar size has direct impact on all register and buffer sizes, the maximum supported datatype size affects the RD- and output buffer as well as the addition unit and the number of available ADCs affects the addition unit as well. These parameters will thus be the main parameters to change during the results exploration, while parameter 4 and 5 are mainly introduced to allow for easy connectivity once the outside unit and instruction memory are designed in future work. The HDL platform capabilities allow for exploring different configurations. Synthesis- and analysis tool Vivado is used to generate accurate timing and area reports for the design. In addition, the vectorless power analysis is used to provides an estimate for power consumption.

Functional testing of the architecture is performed on FPGA. To enable this functional testing, a digital model of the crossbar array and its analog components is used. The crossbar model should be able to (i) store data in an array fashion and (ii) perform computation of the dot-product and logical AND, OR and XOR operations. A single FF per crossbar position will suffice for modeling a memristive cell capable of representing two resistive states. For computation, a Parallel-In-Serial-Out (PISO) shift register per column is deployed. This register allows for sequential addition of the selected crossbar cell values, which directly corresponds to the dot-product operation. Results for the logical AND, OR and XOR operations can be deduced from the addition results as well.

Any computation performed using this crossbar model will thus require n cycles, where n is the crossbar column size. As this corresponds to a very high cycle latency for larger crossbar sizes, the execution time of large programs rises significantly. In addition, the induced overhead leads to unrealistic scenarios during testing which may affect testing results, especially for testing an architecture containing multiple CIM-tiles. To combat these issues, bit-reduction schemes may be deployed to significantly reduce the cycle latency of the model. As the output of this crossbar model is digital, the S+H, MUX and ADC units can be simply replaced for registers and digital multiplexing.

B. Implementation characteristics

Exploration of different configurations will be performed by changing the crossbar size and number of available ADCs for designs allowing for different maximum datatype sizes. The different maximum datatype sizes that will be explored are 8-bit, 16-bit and 32-bit. First, the effect of scaling the crossbar while maintaining one ADC for each set of 32 columns can be studied. In theory, all mayor components of the CIM-tile architecture scale linearly with the crossbar size as long as a constant ratio between the crossbar size and the number of ADCs is maintained. Figures 4c and 4a show the area and power consumption figures retrieved from reports generated by Vivado during synthesis and implementation. The linear increase in area usage and power consumption for each of the three different datatype configurations can be seen. In addition, the effect of supporting different datatypes can be seen. Clearly, supporting a larger datatype requires a larger RD buffer as well as a larger addition unit, resulting in more area usage and larger power consumption.

Figure 4b shows the maximum clock frequency the configurations can be run at without any timing errors. It can be seen that up to a crossbar size of 256x256, the maximum achievable clock frequency for each of the configurations is around 95MHz. For a crossbar size of 512x512, the RD buffer starts bottlenecking the design. It can be seen that the larger RD buffer required for supporting the larger datatypes leads to a larger impact on clock frequency.

To be able to pinpoint which parts of the design contribute most to the area and power consumption, the hierarchical reporting from Vivado is used. Figure 5 shows the typical contribution of each of the mayor consumers of the design for the three different maximum supported datatype sizes. It can be seen that, when supporting larger datatypes, the contribution of the RD-buffer becomes increasingly dominant. This shows us a potential lead for further improving the architecture in future works.

C. Ditigal vs Analog

Using the results obtained from the HDL platform and the simulator, the energy and power consumption of the digital components of the CIM-tile can now be directly compared to the energy and power consumption of the analog components.

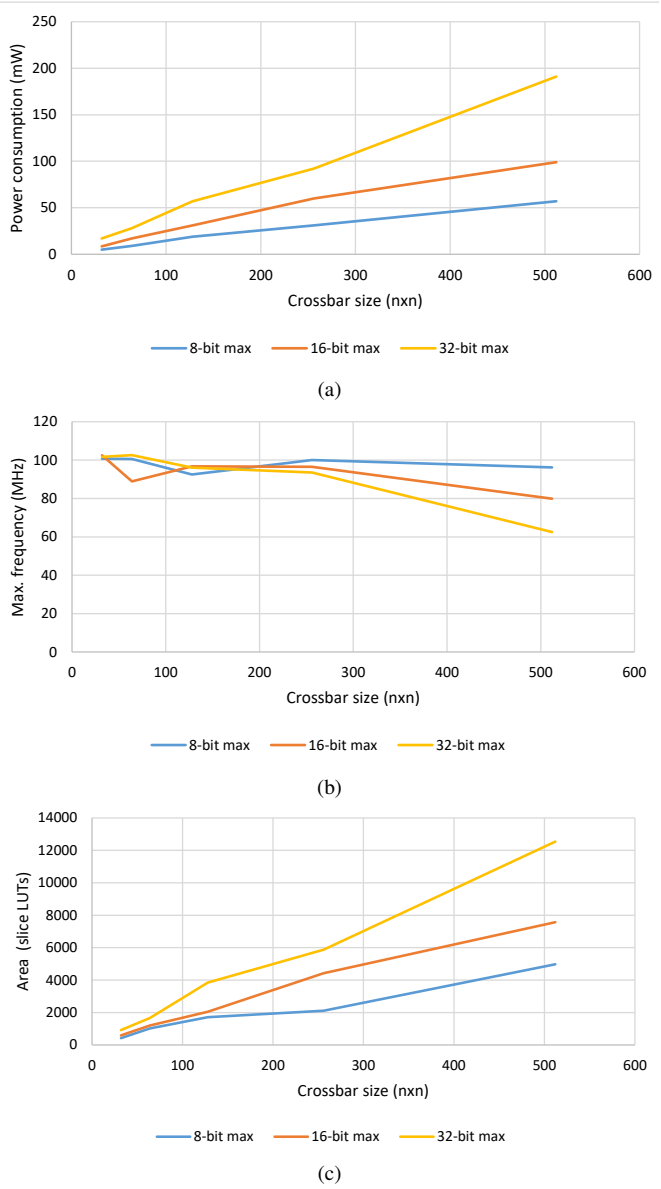


Fig. 4: Power consumption (a), max. operating frequency (b) and Area requirement (c) for different configurations and crossbar sizes.

The characteristics of the digital and analog components are compared for a 256x256 crossbar and 8 available ADCs.

Power consumption: To compare the power consumption of the analog components and the digital circuitry and retrieve a general power requirement for the CIM-tile, the worst case power consumption of the crossbar array is considered for different technologies. This worst case scenario assumes all rows and columns are activated, and all cells are in their low-resistance state. Figure 6 shows a comparison of the power consumption for the analog and digital components of the architecture. The power consumption of an STT-MRAM crossbar array for the read/computation operation is around 20 times larger than the ReRAM crossbar array due to the high read voltage. The STT-MRAM array has been omitted from Figure 6 for better readability of the remaining components. In addition to this worst-case scenario, a more typical case for

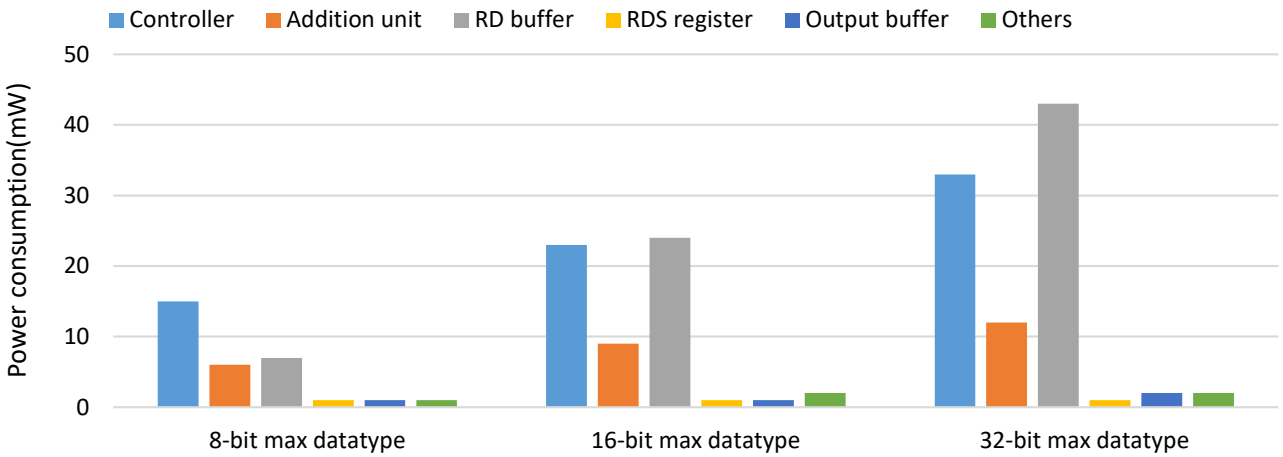


Fig. 5: Contributions of the different digital components to the overall power consumption.

the crossbar and DIM power consumption can be obtained by assuming random data present in the crossbar and RD buffer. In this case, effectively only half of the rows is activated during a computational operation, leading to halving the power consumption. In addition, for ReRAM and PCM technology, 50% of the cells have a negligible contribution to the power consumption due to the high resistance of their HRS leading to again halving the overall power consumption of the array. For STT-MRAM, the power consumption contribution of cells in HRS states is only halved, leading to only a 25% reduction due to the HRS cells.

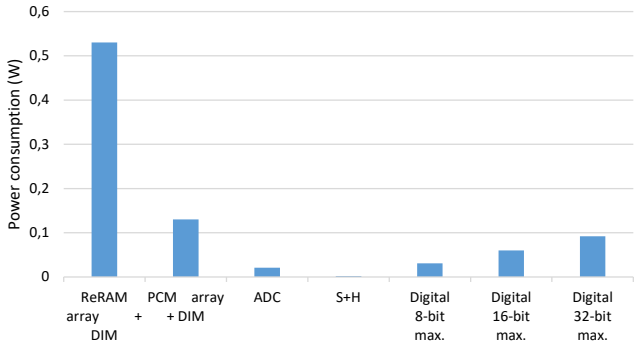


Fig. 6: Power consumption comparison between digital and analog components.

Energy consumption: By embedding Equations eqs. (1) to (4) into the simulator, accurate numbers for energy consumption taking actual data and analog component activation into account can be obtained. Again, the 'GEMM' benchmark is used to obtain these numbers. Figure 7 shows a comparison of the energy consumption of the analog components and the digital circuitry for executing the benchmark at a clock frequency of 90MHz. It can be seen that, while the power consumption of the digital circuitry is a lot lower than that of the crossbar arrays, the energy consumption for running the benchmark is significantly higher. This is due to the fact that the analog components are not activate during the entire time the benchmark program is running. It should be kept in mind that the presented power and energy figures of the digital circuitry are only representative for the actual FPGA implementation of the circuitry. An ASIC implementation of the circuitry may lead to one or even several orders of magnitude lower power consumption [44], [45]. This means that an ASIC implementation of the digital CIM-tile circuitry is expected to be in the same order of magnitude as the analog circuitry regarding these characteristics. Detailed investigation of an ASIC implementation of the design is left for future work.

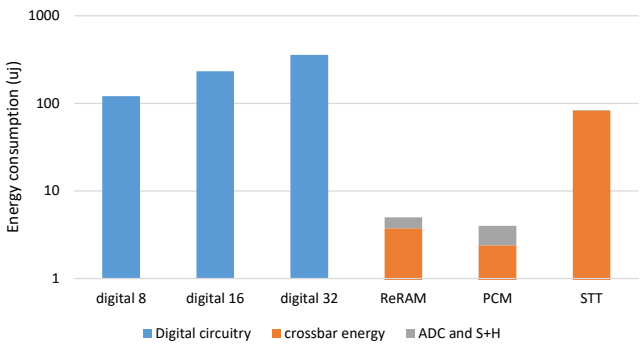


Fig. 7: Energy consumption comparison between digital and analog components.

V. Conclusion

TO BE WRITTEN

References

[1] "Mnemosene - computation-in-memory architecture based on resistive devices," 2020, http://www.mnemosene.eu.

[2] M. Zahedi, M. Lebdeh, S. Wong, and S. Hamdioui, "Mnemosene: Tile organization and simulator for memristor-based computation in-memory," *submitted to MICRO, to be published*, 2020.

[3] S. Yu and P.-Y. Chen, "Emerging memory technologies: Recent trends and prospects," *IEEE Solid-State Circuits Magazine*, vol. 8, no. 2, pp. 43–56, 2016.

[4] Sung Hyun Jo, T. Kumar, S. Narayanan, W. D. Lu, and H. Nazarian, "3d-stackable crossbar resistive memory based on field assisted superlinear threshold (fast) selector," in *2014 IEEE International Electron Devices Meeting*, Dec 2014, pp. 6.7.1–6.7.4.

[5] B. Govoreanu, G. S. Kar, Y. Chen, V. Paraschiv, S. Kubicek, A. Fantini, I. P. Radu, L. Goux, S. Clima, R. Degraeve, N. Jossart, O. Richard, T. Vandeweyer, K. Seo, P. Hendrickx, G. Pourtois, H. Bender, L. Altimime, D. J. Wouters, J. A. Kittl, and M. Jurczak, "10×10nm2 hf/hfox crossbar resistive ram with excellent performance, reliability and low-energy operation," in *2011 International Electron Devices Meeting*, Dec 2011, pp. 31.6.1–31.6.4.

[6] Y. Kim, X. Fong, K.-W. Kwon, M.-C. Chen, and K. Roy, "Multilevel spin-orbit torque mrams," *IEEE Transactions on Electron Devices*, vol. 62, no. 2, pp. 561–568, 2014.

[7] H. Lee, P. Chen, T. Wu, Y. Checn, C. Wang, P. Tzeng, C. Lin, F. Checn, C. Lien, and M. Tsai, "Low power and high speed bipolar switching with a thin reactive ti buffer layer in robust hfo2 based rram," *IEEE International Electron Devices Meeting*, December 2008.

[8] S. Hsu and W. Zhuang, "Electrically programmable resistance cross point memory," June 2001.

[9] M. Lee and I. Baek, "Methods of programming non-volatile memory devices including transition metal oxide layer as data storage material layer and devices so operated," October 2004.

[10] K. Campbell, J. Moore, and T. Gilton, "Single-polarity programmable resistance-variable memory element," August 2002.

[11] M. Lanza, "A review on resistive switching in high-k dielectrics: A nanoscale point of view using conductive atomic force microscope," *Materials*, pp. 2155–2182, 2014.

[12] W. HokenMaier, D. Labrecque, R. Jurasek, V. Butler, C. Scoville, A. Willey, S. Loeffler, Y. Li, and S. Sharma, "A 90nm 32-mb phase change memory with flash spi compatibility," *IMW*, May 2014.

[13] R. Bez, P. Cappelletti, G. Servalli, and A. Pirovano, "Phase change memories have taken the field," *IEEE International Memory Workshop*, May 2013.

[14] L. Xu, Y. Xie, and Y. Lin, "High-reliable multi-level phase change memory with bipolar selectors," *IEEE International Conference on ASIC*, pp. 1011–1014, October 2009.

[15] S. Lai, "Current status of the phase change memory and its future," *IEEE International Electron Devices Meeting*, pp. 255–258, December 2003.

[16] S. Bhatti, R. Sbiaa, A. Hirohat, H. Ohno, S. Fukami, and S. Piramanayagam, "Spintronics based random access memory: a review," *Materials today*, vol. 20, no. 9, pp. 530–548, 2017.

[17] X. Guo, E. Ipek, and T. soyata, "Resistive computation: avoiding the power wall with low-leakage, stt-mram based computing," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, 2010.

[18] M. Lebdeh, U. Reinsalu, H. Nguyen, S. Wong, and S. Hamdioui, "Memristive device based circuits for computation-in-memory architectures," *ISCAS*, 2019.

[19] S. Hamdioui, L. Xie, H. Nguyen, M. Taouil, K. Bertels, H. Corporaal, and H. Jiao, "Memristor based computation-in-memory architecture for data-intensive applications," *Design, Automation And Test in Europe*, pp. 1718–1725, 2015.

[20] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. Strachan, M. Hu, R. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analo arithmetic in crossbars," *ACM/IEEE ISCA*, June 2016.

[21] D. Bhattacharjee, R. Devadoss, and A. Chattopadhyay, "Revamp: Reram based vliw architecture for in-memory computing," *Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE*, pp. 782–787, 2017.

[22] P.-E. Gaillardon, L. Amar, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. De Micheli, "The programmable logic-in-memory (plim) computer," *Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE*, p. 427–432, 2016.

[23] R. Ben Hur and S. Kvatinsky, "Memristive memory processing unit (mpu) controller for in-memory processing," *ICSEE*, pp. 1–5, 2016.

[24] A. Siemon, S. Menzel, R. Waser, and E. Linn, "A complementary resistive switch-based crossbar array adder," *IEEE journal on emerging and selected topics in circuits and systems*, vol. 5, no. 1, pp. 64–74, 2015.

[25] D. Fujiki, S. Mahlke, and R. Das, "In-memory data parallel processor," *In Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems. ACM*, pp. 1–14, 2018.

[26] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," *ACM/EDAC/IEEE DAC*, vol. 53, June 2016.

[27] C. Ping, L. Shuangchen, X. Cong, T. Zhang, Z. Jishen, L. Yongpan, W. Yu, and X. Yuan, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," *ACM/IEEE*, pp. 27–39, 2016.

[28] H. Nguyen, J. Yu, L. Xie, M. Taouil, S. Hamdioui, and D. Fey, "Memristive devices for computing: Beyond cmos and beyond von neumann," *International conference on VLSI-SoC*, October 2017.

[29] L. Yavits, S. Kvatinsky, A. Morad, and R. Ginosar, "Resistive associative processor," *Computer Architecture Letters (CAL)*, 2015.

[30] J. Yu, L. Xie, M. Taouil, and S. Hamdioui, "Memristive devices for computation-in-memory," *in Design, Automation and Test in Europe (DATE)*, 2018.

[31] A. Subramaniyan, J. Wang, E. Balasubramanian, D. Blaauw, D. Sylvester, and R. Das, "Cache automation," *In Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 259–272, 2017.

[32] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," *arXiv preprint*, 2018.

[33] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," *In High Performance Computer Architecture (HPCA)*, pp. 481–492, 2017.

[34] T. Finkbeiner, G. Hush, T. Larsen, P. Lea, J. Leidel, and T. Manning, "In-memory intelligence," *IEEE micro*, vol. 37, no. 4, pp. 30–38, 2017.

[35] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. Kozuch, O. Mutlu, P. Gibbons, and T. Mowry, "Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology," *In Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 273–287, 2017.

[36] S. Li, D. Niu, K. Malladi, H. Zheng, B. Brennan, and Y. Xie, "Drisa: A dram-based reconfigurable in-situ accelerator," *In Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 288–301, 2017.

[37] S. J. E. Wilton and N. P. Jouppi, "Cacti: an enhanced cache access and cycle time model," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 5, pp. 677–688, May 1996.

[38] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 7, pp. 994–1007, July 2012.

[39] M. Poremba and Y. Xie, "Nvmain: An architectural-level main memory simulator for emerging non-volatile memories," in *2012 IEEE Computer Society Annual Symposium on VLSI*, Aug 2012, pp. 392–397.

[40] S. Ambrogio, S. Balatti, A. Cubeta, A. Calderoni, N. Ramaswamy, and D. Ielmini, "Statistical fluctuations in hfox resistive-switching memory: Part i - set/reset variability," *IEEE Transactions on Electron Devices*, vol. 61, no. 8, pp. 2912–2919, Aug 2014.

[41] C. Price, *MIPS IV instruction set*. MIPS Technologies inc., 1995.

[42] K. Einwich, "Introduction to the systemc ams extension standard," *IEEE international symposium on DDECS*, April 2011.

[43] T. Wieman, C. Schröder, B. Bhattacharya, and T. Vanthournout, Band Jeremiassen, "An overview of open systemc initiative standards development," *IEEE Design & Ttest of Computers*, vol. 29, pp. 14–22, April 2012.

[44] I. Kuon and J. Rose, "Measuring the gap between fpgas and asics," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, 2007.

[45] A. Amara, F. Amiel, and T. Ea, "Fpga vs. asic for low power applications," *Microelectronics Journal*, vol. 37, pp. 669–677, 2006.