A Hardware Instruction Generation Mechanism for Energy-Efficient Computational Memories

Léo de la Fuente, Jean-Frédéric Christmann, Manuel Pezzin, Matthias Remars, Olivier Sentieys

A Hardware Instruction Generation Mechanism for Energy-Efficient Computational Memories

Léo De La Fuente, Jean-Frédéric Christmann, Manuel Pezzin, Matthias Remars
Univ. Grenoble Alpes, CEA, List
F-38000 Grenoble, France
first-name.name@cea.fr

Olivier Sentieys
Univ. Rennes, Inria
Rennes, France
olivier.sentieys@inria.fr

***Abstract*—In the Computing-In-Memory (CIM) approach, computations are directly performed within the data storage unit, which often results in energy reduction. This makes it particularly well fitted for embedded systems, highly constrained in energy efficiency. It is commonly admitted that this energy reduction comes from less data transfers between the CPU and the main memory. Nevertheless, preparing and sending instructions to the computational memory also consumes energy and time, hence limiting overall performance. In this paper, we present a hardware instruction generation mechanism integrated in computational memories and evaluate its benefit for Integer General Matrix Multiplication (IGeMM) operations. The proposed mechanism is implemented in the computational memory controller and translates macro-instructions into corresponding micro-instructions needed to execute the kernel on stored data. We modified an existing near-memory computing architecture and extracted corresponding energy consumption figures using post-layout simulations for the complete SoC. Our proposed architecture, NEar memory computing Macro-Instruction Kernel Accelerator (NeMIKA), provides an $8.2\times$ speed-up and a $4.6\times$ energy consumption reduction compared to a state-of-the-art CIM accelerator based on micro-instructions, while inducing an area overhead of only 0.1%.**

***Index Terms*—near-memory computing, macro-instruction, matrix multiplication, GeMM, embedded systems.**

I. Introduction

In embedded systems, reducing energy consumption is a priority to increase battery lifetime. However, with the growing importance of Internet of Things (IoT) and Artificial Intelligence (AI), increasingly intensive calculations have been transferred to edge devices. Hardware accelerators were introduced to reduce the workload of complex tasks, but they are often specialized for a specific kernel. The Computing-In-Memory (CIM) paradigm consists of moving parts of the computations in or near memory instead of moving data to/from the Central Processing Unit (CPU). This significantly reduces data movement while keeping flexibility and is therefore considered as a promising solution to the memory wall problem inherent in Von-Neumann architectures. Reducing data movement has the benefit of significantly lowering power consumption.

In the context of embedded systems, several CIM architectures have been proposed [1]–[3]. Although reducing data transfers, CIM accelerators still need to transfer instructions from the CPU to the memory to control the computations. In this paper, we propose NEar memory computing Macro-Instruction Kernel Accelerator (NeMIKA), a CIM architecture that also reduces the energy consumption of instructions transfer by managing them close to the memory. An in-depth analysis of the applications reveals repetitive patterns of micro-instructions. Managing this repetitiveness with a dedicated control logic located closer to computational memory would reduce the number of instructions transiting on the interconnect. In NeMIKA, in-memory instructions are constructed by the CPU and sent to the computational memory. We propose to handle the execution of repetitive patterns near to the memory with a hardware instruction generation mechanism. This mechanism is managed by a set of sequencers triggered by dedicated CIM macro-instructions. This mechanism transforms the computational memory into a kernel accelerator which uses the same memory blocks and arithmetic/logic operators to execute a predefined set of elementary micro-instructions and thus achieve maximum performance while reducing the associated energy cost. Our analysis is focused on the Integer General Matrix Multiplication (IGeMM) kernel. We use 22nm FD-SOI process technology from GlobalFoundries for implementation and to extract power consumption using post-layout simulation.

This paper is organised as follows: Section II presents the state of the art on CIM relevant to our work. Section III explains our vision of a second level of macro-instructions. Section IV presents our contribution, which is an instruction generator for computational memories with the example of IGeMM operations and the results we have extracted. Finally, Section V concludes this paper.

II. Related Work

CIM is a relevant solution for handling the memory wall in Von Neumann architectures because it reduces data movement on the memory interface while increasing computation parallelism. Within this paradigm, operations are executed in or near memory and often performed on entire memory lines to mitigate their activation cost. We can classify CIM into two main categories depending on whether the computation occurs: inside the memory macro or outside of it. In the literature, these two categories are often named differently [4], [5]. In this paper, we will refer to In-Memory Computing (IMC) and Near Memory Computing (NMC), and use CIM to describe all these types of computing memories. IMC is based on analog computing into the memory array by adding (or subtracting)

currents on columns when multiple physical memory lines are activated during the reading [6]. NMC relies on digital computational elements placed close to the memory macro to reduce timing and energy consumption of data transfers [7]–[10]. In an NMC architecture, data is read from the memory, computed into this specific unit, and stored back to the memory. A strong advantage of the full NMC architecture is that no modification is required on the memory macro. NMC is thus different from IMC, where the macro needs to be modified for a specific memory technology. NMC is also used together with IMC to implement more complex operations, such as carry propagation [11], Multiply And Accumulate (MAC) operation [12], matrix-vector multiplication [13], or other specific operations [14].

In the context of embedded systems, CIM coprocessors are relevant architectures to meet energy consumption constraints. Energy saved by these approaches will increase system's lifetime and/or help reducing battery size. In [9] and [10], the CPU manages the control flow by generating and sending CIM instructions to be executed by the computing memory. Generating instructions on the CPU has a cost in terms of energy or timing, but it makes the CIM module easier to connect to any CPU.

To increase the Instruction Set Architecture (ISA) without developing a specific design, microprocessors such as the MicroVAX 78032, [15] or the Fairchild CLIPPER [16] implement macro-instructions. They use Read-Only Memory (ROM) to store the sequences of micro-instructions required to execute macro-instructions. In this way, on the MicroVAX 78032, floating-point operations could be accelerated with the 78132 floating point external chip connected to the microprocessor or executed by multiple micro-instructions generated from a macro-instruction without this external chip.

The CIM architecture we modified to evaluate our proposal relies on CIM instruction generation by the processor, which comes at the cost of CPU time and system energy. In this paper, we propose a solution that uses macro-instructions to mitigate this problem.

III. NMC ISA Extended with Hardware Instruction Generation

NeMIKA, presented in Fig. 1, is an NMC architecture for embedded systems based on computational SRAM memories. It includes a Vector Processing Unit (VPU) located near to an SRAM memory macro, which allows computations to be performed on data vectors occupying entire lines of memory. NeMIKA is accessed by the CPU through an interconnect. When acting as a coprocessor, NeMIKA is controlled by CIM instructions generated and sent by the processor. Our contribution consists in having incorporated a hardware instruction generation mechanism to improve energy efficiency of the CIM architecture.

The proposed specific ISA includes two levels of instructions: micro and macro-instructions. Micro-instructions are classical arithmetic and logical operations. They are performed

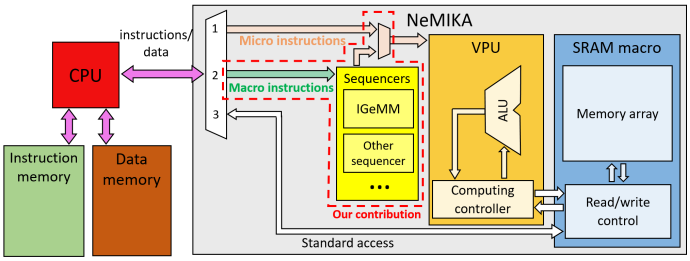


Fig. 1. Overview of NeMIKA architecture connected to a host CPU

on each data of an entire memory line. The ISA of NeMIKA implements micro-instructions on integers of size 8, 16 and 32 bits, which can be specified in the arithmetic operations. The level of parallelization depends on the data size. For example, if the memory width is 128 bits and the data size is 8 bits, NeMIKA can execute 16 operations in parallel. Whereas, for the same memory, if the data size is 16 bits, NeMIKA will execute 8 operations in parallel. Results are stored in a memory line. On the other hand, macro-instructions leverage the repetitiveness of instruction kernels in applications. They implement more complex computations by generating a sequence of micro-instructions directly inside NeMIKA, as shown in Fig. 1. Macro-instructions transform CIM elements of NeMIKA into a kernel accelerator, which allows the processor to work on another task during the processing of the macro-instruction since NeMIKA controls the flow of micro-instructions.

Given that the inclusion of macro-instructions adheres to the NMC guidelines, the SRAM component is therefore not affected. This facilitates its seamless integration into an existing CIM architecture, with the aim of minimizing instruction transfers between CPU and memory.

Today, NeMIKA embeds only one sequencer (IGeMM), but the architecture allows the implementation of several more sequencers adapted to the targeted application.

IV. IGeMM Operations in NeMIKA

In this section, we first describe how NeMIKA is able to execute IGeMM with only micro-instructions, and then with the help of macro-instructions. We finally provide some results of the IGeMM sequencer.

A. IGeMM Implemented with Micro-Instructions

NeMIKA does not implement scalar to vector multiplication. Matrix multiplication therefore needs a specific organization of the operands to take advantage of its vectorization support. For matrices $A \epsilon \mathbb{Z}^{m \times n}$, $B \epsilon \mathbb{Z}^{n \times p}$, $C \epsilon \mathbb{Z}^{m \times p}$ and the computation of $A \times B = C$, the storage of matrices is illustrated in Fig. 2 and performed as follows:

- Each element of A is repeated on a whole memory line.
- Matrix B is placed row by row in the memory. If the size of a matrix row is larger than the width of the memory, we use several memory lines. The alignment of each matrix row is kept by using a new memory line for each row.
- Once calculated, C is stored in the same format as B.

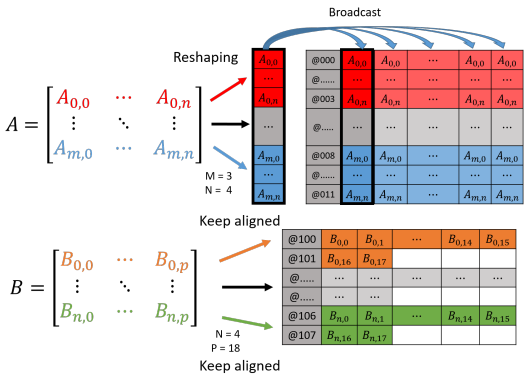


Fig. 2. Specific matrix storage in the memory of NeMIKA

Algorithm 1 micro-instructions flow of IGeMM

```
for (i = 1; i < m; i++) do
    for (l = 0; l < q ; l++) do
        MUL (reg, @A[0,0] + i · n, @B[0,0] + l)
        for (j = 0; j < (n − 1); j++) do
            MAC (reg, @A[0,0] + i · n+j, @B[0,0] + j · q + l)
        end for
        MAC (@C[0,0]+i·q+l, @A[0,0]+i·n+j, @B[0,0]+j·q+l)
    end for
end for
```

Due to the possibility of adding a Direct Memory Access (DMA) module to move data and reshape them from a memory to the computational memory, the movement and reorganization of data is not considered in this work. In the following, we assume that data are already well positioned in memory.

The computation of C mainly relies on MAC instructions. The sequence of instructions generated by the processor to be executed by NeMIKA is shown in Algorithm 1, with m, n, p representing the matrix sizes, D the number of data in a memory line, and $q = \lceil p/D \rceil$ the number of memory lines needed to store a row of the B or C matrices. The accumulation is done in an internal register (reg) present in the VPU. The first MAC is replaced by a multiplication and stored in the internal register to avoid useless write and read preliminary data into the SRAM, while preventing from resetting the register before. The last MAC is stored in the memory. With only micro-instructions, NeMIKA requires $m \times n \times q$ NMC instructions generated by the CPU and sent through the interconnect to execute IGeMM, which is further optimized in the next section using macro-instructions.

B. IGeMM Implemented with Macro-Instructions

The main advantage of NeMIKA relies the definition of an IGeMM macro-instruction, which executes the matrix multiplication in computing memories by receiving a fix number of macro-instructions, independent of the matrix sizes. These instructions allow to set some internal registers with the size of m, n, p and the addresses of the memory line containing the first element of each matrix A, B, and C. A hardware sequencer generates all the NMC micro-instructions

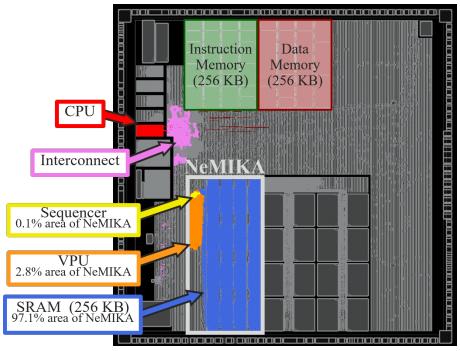


Fig. 3. Physical view of NeMIKA in a 22nm development platform

needed to execute the IGeMM. It generates the same flow of micro-instructions as the one described in Algorithm 1. The sequencer contains three registers: i, j and l to iterate in ranges from 0 to m, n and q respectively.

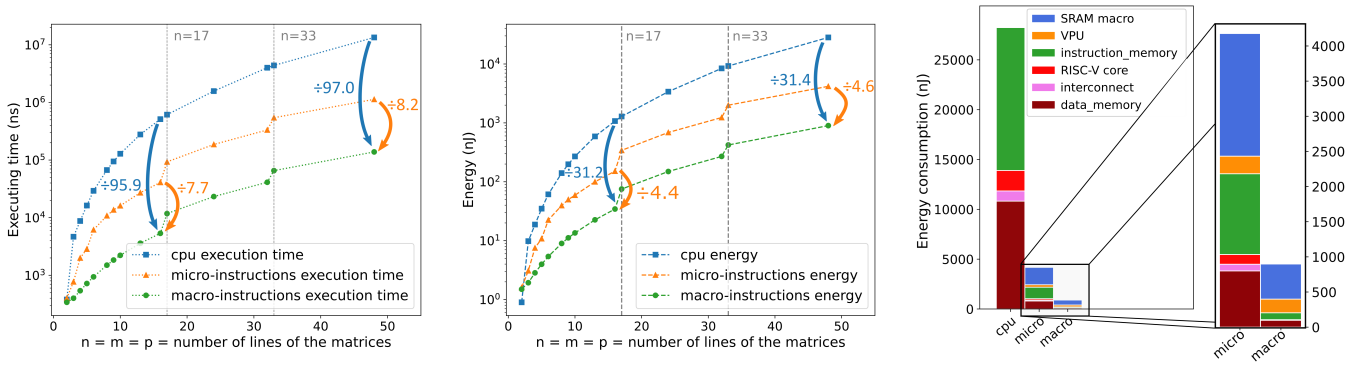
The IGeMM macro-instruction reduces the number of instructions sent by the CPU to NeMIKA from $m \times n \times q$ to only 2 instructions. The generation of one micro-instruction by the sequencer is done in one cycle and allows the CPU to feed our computing memory to his maximum rate and reach the best performance.

C. Experimental Results

To study the proposed architecture and evaluate the energy consumption reduction, we have integrated NeMIKA with a *cv32e40p* RISC-V core [17]. During the matrix multiplication with our instruction generation mechanism, the host CPU is idled and waits for the end of the computation. Thus, no instructions are fetched to the CPU which excludes the energy that is not linked to the IGeMM computation.

Fig. 3 represents the physical view of the SoC designed in GF 22-FDX (22nm FD-SOI). We ran post-layout energy analysis to evaluate the consumption of the architecture on real executed algorithms. Our contribution increases the size of the VPU by $610\mu m^2$, which corresponds to an increase of 3.7%, and represents only 0.1% of the total size of the entire computational memory (256 KB).

We evaluate IGeMM in NeMIKA with and without macro-instructions. In this study, the RISC-V core sends the necessary instructions to compute all the matrix multiplication. Matrices are already stored in the memory of NeMIKA following the organization described in Section IV-A. Data are 8-bit integers and have been generated randomly. To compare with a Harvard architecture we also implemented IGeMM on the RISC-V core. This experiment is composed of three simple nested loops, and matrix elements are stored contiguously in the data SRAM. There are therefore three different software implementations which run on the same hardware, but they use different functionalities of the design. In the following, these implementations will be called *cpu*, *micro* and *macro*. To simplify the analysis, we focus only on square matrices of size $n{=}m{=}p$ from 2 to 48. Energy consumption is extracted using Synopsys PrimePower tool. Clock frequency is 100Mhz. The analysis does not include the clock network energy.



(a) Execution time depending on matrix size n

(b) Total energy consumption depending on matrix size n

(c) Energy consumption for $n = 48$ split by system component

Fig. 4. Simulation results and comparison of the three IGeMM implementations proposed in this paper: *cpu*, *micro* and *macro*

Fig. 4a and Fig. 4b report execution time and energy consumption, respectively, for the three different software implementations, for varying square matrix sizes. These figures show that both the *macro* and the *micro* implementations are relevant starting from n=3. Indeed, from n=16, the *micro* implementation already accelerates more than a 7.7× factor with respect to the *cpu* implementation, while the *macro* reaches more than 95.9×. From n=16, our contribution reaches 0.765×10^9 8-bit MAC operations per second (GMAC8S), which is really close to the maximum throughput achievable by our computational memory (0.8 GMAC8S). The energy consumption reported in Fig. 4b shows that from n=13 the reduction factor of energy consumption of *macro* implementation is around 30× with respect to the *cpu* and around 4.5× with respect to the *micro*. These factors are independent of the size n of the matrices. The reduction of the number of generated instructions from $m \times n \times q$ to only 2 decreases the energy consumption associated to the generation and the transfer of CIM instructions (energy consumed by the CPU, by the instruction memory and by the interconnect to fetch the RISC-V instructions to the core and to transfer the CIM instructions to NeMIKA). Steps at n=17 and n=33 (dashed vertical lines) in the curves of Fig. 4 are due to the necessity to use a new line of memory to store a matrix row.

Fig. 4c shows the energy contribution of each system component in the different implementations when n=48. Both CIM architectures show a high energy reduction when compared to the *cpu* implementation. Energy of core and data/instruction memories are particularly reduced, mainly due to much fewer CPU instructions and the shift of matrix data to the CIM. When comparing the two CIM architectures (see the zoom on Fig. 4c), our approach based on macro-instructions leads to a reduction in all architecture components. As already mentioned, energy of core and data/instruction memories is reduced thanks to less instructions generated by the CPU and transferred to the CIM. Only the VPU energy remains the same, as it executes the same micro-instructions. More generally, the overall energy to perform IGeMM operations is also reduced since the total execution time is significantly lower. This also explains why the SRAM macro of NeMIKA consumes less energy, since it remains less time in an idle mode. Finally, the interconnect energy is reduced by a factor of 17×, from 95nJ in the *micro* implementation to 5.5nJ for the *macro*. Although the interconnect energy consumption is only 2.3% of the overall SoC energy for the *micro* case, its reduction in NeMIKA macro could become significant for other systems. In particular, for systems without on-chip computing memories (e.g., chiplets or boards) where interconnect could dominate the total energy, our hardware instruction generation mechanism represents an exciting opportunity.

V. Conclusion

In this paper, we present NeMIKA, a computational memory architecture dedicated to embedded systems that achieves a trade-off between efficiency and versatility, based on a two-level ISA: micro and macro-instructions. NeMIKA uses a hardware mechanism to generate micro-instructions for computational memories, which reduces drastically the number of instructions generated and sent from the host CPU to the CIM accelerator. This macro/micro-instruction approach can be easily adapted to fine-tune a computational memory to application needs. We provide results on an 8-bit IGeMM to show the benefits of our proposal. Without our IGeMM macro-instructions, the number of instructions transferred depends on the matrix sizes, whereas it is constant and much lower with our approach. We implemented NeMIKA in an SoC with a RISC-V core and executed post-layout simulations to extract energy consumption. We show that our contribution significantly reduces the energy consumption and the execution time for square matrix multiplication, even for very small matrices (3x3). The reduction in execution time and energy consumption reaches 97× and 31×, respectively, when compared to a CPU-only solution, with n=48. When compared to a state-of-the-art CIM accelerator based on micro-instructions, NeMIKA still provides significant gains with 8.2× and 4.6× reduction in time and energy, respectively. These gains are almost constant and independent of n from n=13. The area overhead associated represents only 3.7% of the VPU, or 0.1% of the entire computing memory.

REFERENCES

[1] H. Amrouch, M. Imani, X. Jiao, Y. Aloimonos, C. Fermuller, D. Yuan, D. Ma, H. E. Barkam, P. R. Genssler, and P. Sutor, "Brain-Inspired Hyperdimensional Computing for Ultra-Efficient Edge AI," in 2022 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), Oct. 2022, pp. 25–34, iSSN: 2832-6474.

[2] B. C. M. Choong, T. Luo, C. Liu, B. He, W. Zhang, and J. T. Zhou, "Hardware-software co-exploration with racetrack memory based in-memory computing for CNN inference in embedded systems," Journal of Systems Architecture, vol. 128, p. 102507, 2022.

[3] V. Jamshidi, A. Patooghy, and M. Fazeli, "MagCiM: A Flexible and Non-Volatile Computing-in-Memory Processor for Energy-Efficient Logic Computation," IEEE Access, vol. 10, pp. 35 445–35 459, 2022.

[4] A. Gebregiorgis, H. A. Du Nguyen, J. Yu, R. Bishnoi, M. Taouil, F. Catthoor, and S. Hamdioui, "A Survey on Memory-Centric Computer Architectures," J. Emerg. Technol. Comput. Syst., vol. 18, no. 4, Oct. 2022, place: New York, NY, USA Publisher: Association for Computing Machinery.

[5] S. Mittal, G. Verma, B. Kaushik, and F. A. Khanday, "A survey of SRAM-based in-memory computing techniques and applications," Journal of Systems Architecture, vol. 119, p. 102276, 2021.

[6] A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh, and E. Eleftheriou, "Memory devices and applications for in-memory computing," Nature Nanotechnology, vol. 15, no. 7, pp. 529–544, Jul. 2020.

[7] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, and G. Daglikoca, "The Architecture of the DIVA Processing-in-Memory Chip," in Proceedings of the 16th International Conference on Supercomputing, ser. ICS '02. New York, NY, USA: Association for Computing Machinery, 2002, pp. 14–25, event-place: New York, New York, USA. [Online]. Available: https://doi.org/10.1145/514191.514197

[8] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, R. Ausavarungnirun, K. Hsieh, N. Hajinazar, K. T. Malladi, H. Zheng, and O. Mutlu, "CoNDA: Efficient Cache Coherence Support for Near-Data Accelerators," in 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA), 2019, pp. 629–642.

[9] J.-P. Noel, V. Egloff, M. Kooli, R. Gauchi, J.-M. Portal, H.-P. Charles, P. Vivet, and B. Giraud, "Computational SRAM Design Automation using Pushed-Rule Bitcells for Energy-Efficient Vector Processing," in 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE), 2020, pp. 1187–1192.

[10] M. Kooli, A. Heraud, H.-P. Charles, B. Giraud, R. Gauchi, M. Ezzadeen, K. Mambu, V. Egloff, and J.-P. Noel, "Towards a Truly Integrated Vector Processing Unit for Memory-Bound Applications Based on a Cost-Competitive Computational SRAM Design Solution," J. Emerg. Technol. Comput. Syst., vol. 18, no. 2, Apr. 2022, place: New York, NY, USA Publisher: Association for Computing Machinery.

[11] J. Wang, X. Wang, C. Eckert, A. Subramaniyan, R. Das, D. Blaauw, and D. Sylvester, "A 28-nm Compute SRAM With Bit-Serial Logic/Arithmetic Operations for Programmable In-Memory Vector Computing," IEEE Journal of Solid-State Circuits, vol. 55, no. 1, pp. 76–86, 2020.

[12] S. Srinivasa, A. K. Ramanathan, J. Sundaram, D. Kurian, S. Gopal, N. Jain, A. Srinivasan, R. Iyer, V. Narayanan, and T. Karnik, "Trends and Opportunities for SRAM Based In-Memory and Near-Memory Computation," in 2021 22nd International Symposium on Quality Electronic Design (ISQED), 2021, pp. 547–552.

[13] G. A. Eggermann, M. A. Rios, G. Ansaloni, and D. Atienza Alonso, "A 16-bit Floating-Point Near-SRAM Architecture for Low-power Sparse Matrix-Vector Multiplication," 2023, p. 6.

[14] H. Jia, H. Valavi, Y. Tang, J. Zhang, and N. Verma, "A Programmable Heterogeneous Microprocessor Based on Bit-Scalable In-Memory Computing," IEEE Journal of Solid-State Circuits, vol. 55, no. 9, pp. 2609–2621, 2020.

[15] D. W. Dobberpuhl, R. M. Supnik, and R. T. Witek, "The MicroVAX 78032 Chip, A 32-Bit Microprocessor," j-DEC-TECH-J, vol. 1, no. 2, pp. 12–23, Mar. 1986.

[16] W. Hollingsworth, H. Sachs, and A. J. Smith, "The Fairchild CLIPPER: Instruction Set Architecture and Processor," University of California at Berkeley, USA, Tech. Rep., 1987.

[17] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 25, no. 10, pp. 2700–2713, 2017.