EPIC: A Sub-6mW In-Memory Computing-based RISC-V Microcontroller Unit with On-Chip Training Support for TinyML

Chuan-Tung Lin, Seunghyun Moon, Paul Xuanyuanliang Huang, Mingoo Seok
Department of Electrical Engineering, Columbia University, New York, NY, USA
Email: {cl4030, sm5508, xh2373, ms4415}@columbia.edu

***Abstract*—We present EPIC, a sub-6mW digital in-mEmory comPutIng-based RISC-V microController unit. We create the hardware and software systems with the optimized dataflow for both inference and training. EPIC supports the industrial software development framework based on the GCC compiler for RISC-V IM32 and TensorFlowLite-micro. The hardware and software systems can support any near-arbitrary deep neural network model having as large as 340,000 parameters. EPIC is prototyped in 28-nm CMOS, achieving 4-22× improvement in the energy-delay product over the prior best neural accelerator in all the MLPerf-Tiny benchmark suites.**

***Keywords*—Deep learning, tiny machine learning (TinyML), in-memory computing (IMC), microcontroller unit (MCU), RISC-V, accelerator**

I. Introduction

Tiny machine learning (TinyML) aspires to collect data, execute machine learning models, and adapt to users and environment, all in an ultra-low-power device near sensors. TinyML can enable key benefits, such as reduced latency, extended battery life, security, and privacy. Toward this vision, it is paramount to create a microcontroller unit (MCU) that can perform deep neural networks (DNN)-based inference and training at high efficiency and low latency [1]. Several such neural MCUs and accelerators have recently been proposed, demonstrating much-improved efficiency and performance [2-10]. However, to improve them, most of the existing neural hardware employs mostly fixed hardware, thus exhibiting limited programmability. For example, [2] can support only regular convolutions, not depthwise (DW) ones. [3] can support only three types of activation functions and limited filter dimensions like 1×1 and 3×3. Also, most of the prior works support only inference, while it is increasingly crucial to support training for user and environment adaptation [11-12].

We propose EPIC, a sub-6mW digital in-mEmory comPutIng (IMC)-based RISC-V MCU. We also created the matching software development framework based on the GCC compiler for RISC-V IM32 [13] and TensorFlowLite (TFLite)-micro [14]. The resulting hardware and software platform can support a near-arbitrary DNN model having as large as 340,000 parameters. While EPIC enables such full programmability, it still significantly improves efficiency. The 28-nm test chip achieves 4-22× improvement in the energy-delay product (EDP) over the prior best neural accelerator [3] in the MLPerf-Tiny benchmark suite [1].

II. Hardware and Software

A. Hardware Architecture

Fig. 1 shows the organization of EPIC. It consists of an IMC accelerator, a 32b RISC-V host processor, data memory (DMEM), instruction memory (IMEM), direct memory access unit (DMA), I/O (GPIO, UART), 32b ARM AHB bus, and 512b accelerator bus. The IMC accelerator consists of a scratch pad, an IMC cluster having 16 digital IMC macros, one custom IMC macro for DW convolution (DW-IMC), a post-processing unit, an adder tree, etc.

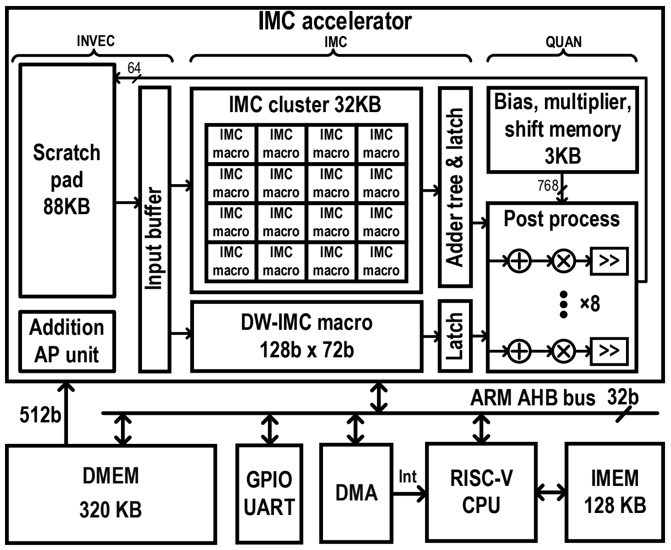
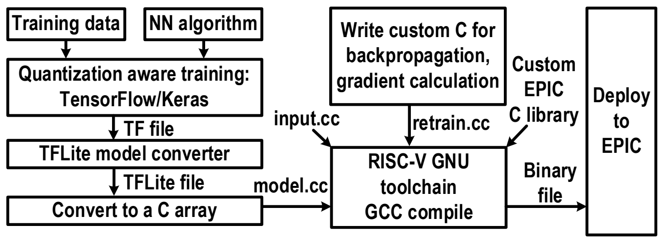


Fig. 1. Proposed EPIC hardware architecture.



Model	Workload	Parameter size	Software size
DSCNN	Keyword spotting [1]	53KB	246KB
MobileNetv1	Visual wake words [1]	325KB	440KB
ResNetv1	Image classification [1]	96KB	302KB
Auto encoder	Anomaly detection [1]	270KB	405KB
DSCNN	Retraining the last layer of DSCNN	Total: 53KB Retraining: 2KB	254KB

Fig. 2. Software development framework based on RISC-V GCC and TFLite-micro (top). The inference and retraining workloads considered for benchmarks (bottom).

B. *Software Development Framework*

Fig. 2 top shows the matching software development framework based on GCC and TFLite-micro. We can write a C++ code or generate one from TFLite-micro. Then, we can compile the code with the custom EPIC library. The compilation produces the instruction and data hexadecimal files. Using the framework, as summarized in Fig. 2 bottom, we developed several software programs that perform the inference and the last-layer retraining of the following models: DSCNN, MobileNetv1, ResNetv1, and Autoencoder.

III. Dataflow Optimizations

A. *Workload Profiling*

We started our hardware design by profiling which layers of DNN models are worth accelerating. We intend to accelerate only those layers to save silicon area. Based on our workload profiling using SPIKE [15], we choose to accelerate the following layers: convolution, DW convolution, average pooling (AP), and addition. As shown in Fig. 3 top, the profiling indicates that if we can accelerate those layers by 500×, the cycle counts reduce by 200-450×. All the other layers (i.e., max pooling, fully connected, softmax) are not worth accelerating due to the limited cycle count reduction.

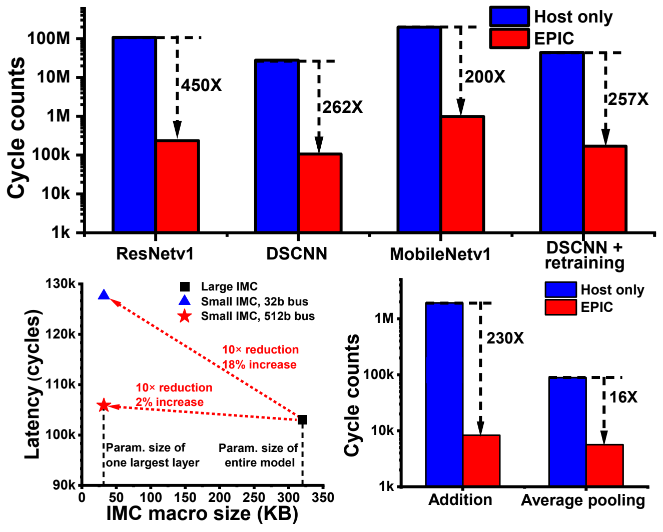


Fig. 3. Acceleration projection (top). The proposed dataflow reduces the IMC size by 10× with only 2% latency increase (bottom left). EPIC improves addition by 230× and AP by 16× (bottom right).

B. *Dataflow for DNN Weight Data*

Driven by the software/workloads, we optimized the datapath hardware and the dataflow. In this process, we strive to add the least amount of new hardware yet still enable a significant acceleration. First of all, in the recent literature, the SRAM-based IMC circuits have demonstrated extremely high efficiency [5-8]. However, many prior works employ many of them to store all of the parameters of a model to maximize efficiency [5-6]. The IMC macros are bulky, so such an approach severely increases area overhead.

Therefore, we adopt a dataflow, where the DMEM that is implemented in the dense foundry 6T bitcells stores all the parameters, but the IMC macros buffer the parameters of only one (or partial) layer right before the accelerator starts to compute on that layer. As shown in Fig. 3 bottom left, this proposed dataflow reduces the IMC macro size by 10× (i.e., 320KB to 32KB). However, moving the parameter to the IMC macros could increase the cycle count by 18%. To avoid this overhead, we deploy a dedicated 512b bus between DMEM and the accelerator, which reduces the cycle count overhead only to 2% of the total cycle count (Fig. 3).

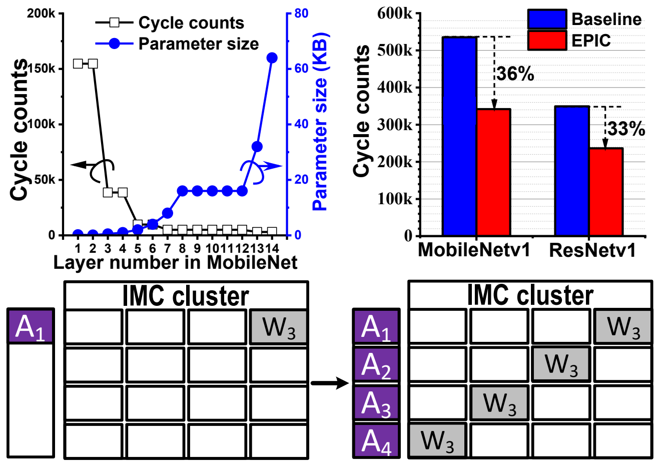


Fig. 4. The first seven layers contain a small number of parameters but dominate the latency due to the low utilization of IMC (top left). The proposed technique reduces the cycle counts by 36% for MobileNetv1 and 33% for ResNetv1 (top right). We load 2-4 copies of a parameter matrix in the IMC macros for a small layer (bottom).

C. *Dataflow for Small Convolution Layers*

We also create a dataflow for a small convolution layer, which otherwise can severely underutilize the IMC hardware. For example, the first seven layers of MobileNetv1 contribute only 1.36% of the total parameter count but are responsible for 87% of the latency without the proposed optimization (Fig. 4 top left). To address this bottleneck, we buffer up to four copies of the same parameter matrix in the IMC macros (Fig. 4 bottom) and perform four convolutions in parallel. It improves the cycle count by 36% for MobileNetv1 and 33% for ResNetv1 (Fig. 4 top right).

D. *Dataflow for Average Pooling and Shortcut Addition*

In addition, we create a dataflow for an AP layer. Here, we decided to accelerate only the addition part of the AP since the division part accounts for only 6% of the total cycles, making it difficult to justify adding a bulky hardware divider. Instead, we have the host to handle the division operation. We add the hardware adders to the accelerator. Indeed, we can recycle these added adders to support the shortcut addition layer in ResNetv1. The simulation shows this update speeds up the AP layer by 16× and the shortcut addition layer by 230× (Fig. 3 bottom right).

E. *Dataflow to Support On-Chip Training*

We also create a dataflow to support on-chip training but without adding any additional hardware. As shown in Fig. 5, the gradient descent-based training mainly has four subroutines: forward propagation, backpropagation, gradient calculation, and parameter update. The forward propagation is the same as the inference operation. Thus, we can recycle the same dataflow and hardware that we added for inference. Here, the IMC macros store weights and receive input activation sequentially (i.e., weight stationary). The backpropagation is similar, except it requires two operations, namely transposing a weight matrix and calculating errors. We have the host to deal with them as they incur the cycle count penalty of only 12%. We could modify the IMC macros to support transpose operation (as done in [16]), but it could increase the macro area by 50%. Finally, the gradient calculation is mostly about vector-matrix multiplication (VMM) between a gradient matrix and input activations, which the IMC macros can effectively calculate. We choose the gradient stationary scheme as it shows the least cycle count.

Algorithm 1: Retraining the last layer

```
Input: NN input ($A_0$), parameter ($W$), label ($Y$)
Output: NN output ($OUT$), retrained parameter ($W'$)
/* Step 1:  forward propagation                    */
for $i := 1$ to 9 do
    $A_i \leftarrow Convolution\_IMC(A_{i-1}, W_i)$
end
$A_{10} \leftarrow AP\_host(A_9)$
$A_{11} \leftarrow FC\_host(A_{10}, W_{FC})$
$OUT \leftarrow Softmax\_host(A_{11})$
/* Step 2:  backpropagation                        */
$\frac{\partial L}{\partial FC} \cdot \frac{\partial FC}{\partial AP} \cdot \frac{\partial AP}{\partial CONV9} \leftarrow host((OUT-Y) \cdot k(W_{FC}-W_{offset}) \cdot \frac{1}{64})$
$W_9^T \leftarrow host(W_9)$
$\frac{\partial L}{\partial CONV8} \leftarrow Convolution\_IMC(\frac{\partial L}{\partial CONV9}, W_9^T)$
/* Step 3:  gradient calculation                   */
$\partial W_8 \leftarrow Convolution\_IMC(\frac{\partial L}{\partial CONV8}, A_7)$
/* Step 4:  parameter update                       */
$W_8' \leftarrow host(W8 + \partial W_8)$
```

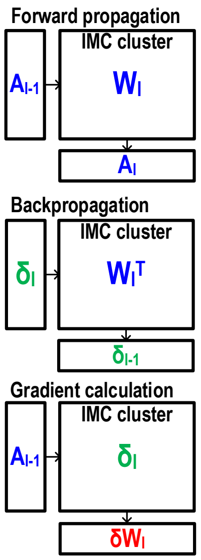


Fig. 5. Retraining model to update the last layer of the DSCNN (left). The proposed IMC mapping methods for forward propagation, backpropagation, and gradient calculation (right).

F. Accelerator

To support all of the proposed dataflows, we create a 3-stage IMC accelerator. Fig. 1 top shows the microarchitecture. The first stage (INVEC) prepares input vectors. The second stage (IMC) performs VMM using the 4×4 IMC macro cluster and one DW-IMC macro. The last stage (QUAN) quantizes the IMC stage's 26b results into 8b. It supports the quantization scheme of TFLite-micro [14], where the quantized value q is defined as $q=2^n \cdot M_0 \cdot (r+Z)$, where n, M_0, Z are the hyper-parameters that TFLite-micro offline computed and r is the IMC stage's result.

G. IMC Macro Circuits

We adopted one of the state-of-the-art digital IMC macros called D6CIM [17]. It achieves a weight density of 126 KB/mm^2, a compute density of 1.25 TOPS/mm^2 at 1V, and an energy efficiency of 40.16 TOPS/W at 0.6V.

However, D6CIM cannot deal with DW convolution effectively as it is optimized for a regular (tall) matrix. The DW convolution has a short matrix. As shown in Fig. 6 top left, in the conventional IMC macro, those short matrices must be stored in different columns, severely reducing hardware utilization. Thus, we developed the DW-IMC macro, which can store multiple short matrices in the same column, vastly improving hardware utilization (Fig. 6 top right).

Fig. 7 shows the schematics of the DW-IMC macro. We add eight accumulators in each column and feed dedicated input vectors to each column. This allows us to compute multiple short vector-vector multiplications simultaneously. In Fig. 6 bottom, the simulation shows that the DW-IMC macro speeds up DW convolution by 8× compared to D6CIM. Having two types of IMC macros may raise concerns about area overhead. However, we found the area overhead is small since a DW layer is small, so we need only one 1.1KB DW-IMC macro.

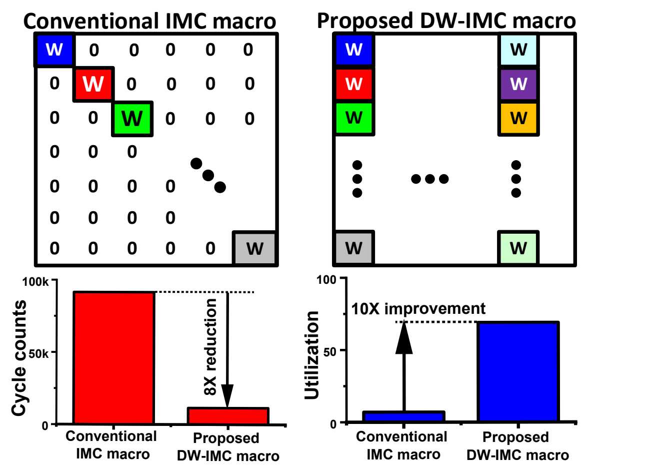


Fig. 6. The conventional IMC macro has been optimized for a tall matrix, suffering from low hardware utilization for DW convolution with short matrixes (top). The DW-IMC macro improves the hardware utilization by 10X and reduces the cycle counts for DW convolution layers by 8X (bottom).

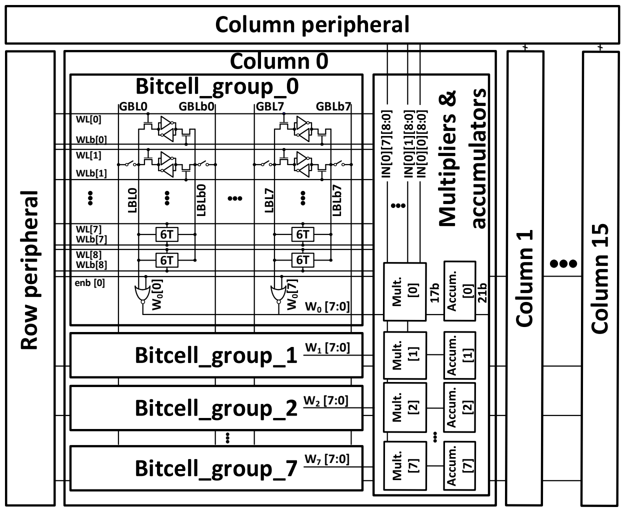


Fig. 7. The proposed DW-IMC microarchitecture.

IV. Measurement Results

We prototyped the test chip in 28nm CMOS. Fig. 8 shows the die photo. As shown in Fig. 9 top, EPIC achieves 407MHz (69MHz) and consumes 251pJ/cycle (78pJ/cycle) at 1.1V (0.6V). Fig. 9 bottom shows the energy and area breakdown. The accelerator takes the largest portion in both. We have

EPIC perform the software for interference and retraining. Fig. 10 shows the latency and energy consumption of each program. Table I shows the comparison to the prior works. Compared to the previous best neural accelerator [3], EPIC achieves 4-22× EDP improvement while requiring only half the SRAM used in [3]. Also, EPIC is the only neural MCU that can perform on-chip training. Retraining the last layer of DSCNN with a batch size of one costs 0.59ms and 27 µJ per epoch, comparable to performing one inference operation using the same model.

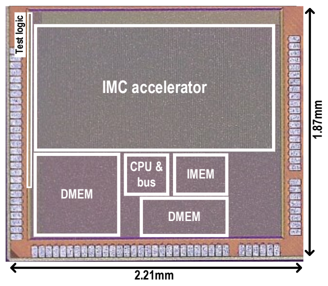


Fig. 8. Die photo.

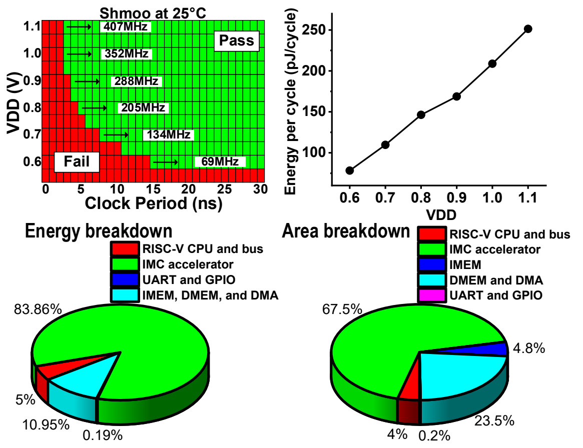


Fig. 9. Measurement results. Shmoo plot (top left). Energy per cycle for ResNetv1 across VDDs (top right). Energy breakdown for ResNetv1 (bottom left). Area breakdown (bottom right).

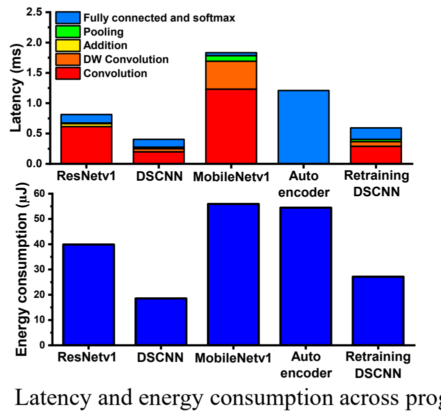


Fig. 10. Latency and energy consumption across programs.

V. Conclusion

We propose EPIC, a sub-6mW digital IMC-based RISC-V MCU. We also created the software development framework based on the GCC compiler for RISC-V and TFLite-micro to support DNN inference and on-chip training. The 28-nm chip achieves 4-22× improvement in the EDP over the prior art in the MLPerf-Tiny benchmark suite.

TABLE I. Comparison Table

		This work	ISSCC23 Syntiant [3]	xG24DK2601B Silicon Labs [1]	NUCLEO-H7A3ZI STMicro [1]	JSSC24 iMCU [2]
Technology [nm]		28	40	n/a	n/a	28
Host processor		RISC-V	HiFi3 + M0	Cortex-M33	Cortex-M7	RISC-V
Accelerator		IMC accelerator	Syntiant Core 2	Digital accelerator	n/a	IMC accelerator
Supply voltage [V]		0.6-1.1	1.1	1.8	0.74-1.3	0.6-1
Total SRAM size [KB]		572	1024	256	1434	467
Operating frequency [MHz]		69-407	98.7	40-78	280	29-310
MLPerf-Tiny: ResNetv1 on CIFAR10	**Latency [ms]**	**0.82**	5.12	120.93	54.34	60.9
	Energy consumption [µJ]	**39.9**	139.37	1234.65	8707.28	102.18
MLPerf-Tiny: DSCNN on Google Speech Commands	**Latency [ms]**	**0.41**	1.48	36.28	16.76	n/a
	Energy consumption [µJ]	**18.56**	43.8	401.86	2721.78	n/a
MLPerf-Tiny: MobileNetv1 on VWW dataset	**Latency [ms]**	**1.84**	4.1	111.61	50.7	n/a
	Energy consumption [µJ]	**55.9**	97.16	1139.22	7978.47	n/a
On chip training: Last layer of DSCNN	**Latency [ms]**	**0.59**	n/a	n/a	n/a	n/a
	Energy consumption [µJ]	**27.2**	n/a	n/a	n/a	n/a

Acknowledgment

This work was supported by NSF IIP-1919147 and by COGNISENSE, an SRC program sponsored by DARPA.

References

[1] C. Banbury et al., "Mlperf Tiny Benchmark," NeurIPS, 2021.
[2] C. Lin et al., "iMCU: A 28-nm Digital In-Memory Computing-Based Microcontroller Unit for TinyML," JSSC, 2024.
[3] D. Garrett et al., "A 1mW always-on computer vision deep learning neural decision processor," ISSCC, 2023.
[4] V. Jain et al., "Tinyvers: A Tiny Versatile System-on-Chip With State-Retentive eMRAM for ML Inference …," JSSC, 2023.
[5] H. Jia et al., "A Programmable Heterogeneous Microprocessor Based on Bit-Scalable InMemory Computing," JSSC, 2020.
[6] J. Wang et al., "A 28-nm Compute SRAM With Bit-Serial Logic/Arithmetic Operations for Programmable...," JSSC, 2020.
[7] J. Seo et al., "Digital Versus Analog Artificial Intelligence Accelerators: Advances, trends, and emerging designs," IEEE Solid-State Circuits Magazine, 2022.
[8] M. Sinangil et al., "A 7-nm Compute-in-Memory SRAM Macro Supporting Multi-Bit Input, Weight and Output and Achieving 351 TOPS/W and 372.4 GOPS," JSSC, 2021.
[9] T. Wu et al., "A 3D integrated Prototype System-on-Chip for Augmented Reality Applications Using Face-to-Face Wafer Bonded 7nm Logic at <2µm Pitch with up to 40% Energy Reduction at Iso-Area Footprint," ISSCC 2024.
[10] P. Wiese et al., "Toward Attention-based TinyML: A Heterogeneous Accelerated Architecture and Automated Deployment Flow," IEEE Design & Test 2025.
[11] S. Q. Zhang et al., "CAMEL: Co-Designing AI Models and eDRAMs for Efficient On-Device Learning," HPCA 2024.
[12] K. Sunaga et al., "Addressing Gap between Training Data and Deployed Environment by On Device…," IEEE Micro, 2023.
[13] GNU toolchain for RISC-V, https://github.com/riscv-collab/riscv-gnu-toolchain
[14] R. David et al., "TensorFlow Lite Micro: Embedded Machine Learning for…," Proc. of Machine Learning and Systems, 2021.
[15] Spike RISC-V ISA Simulator, https://github.com/riscv-software-src/riscv-isa-sim
[16] J. Su et al., "Two-Way Transpose Multibit 6T SRAM Computing-in-Memory Macro for Inference-…," JSSC 2021.
[17] J. Oh et al., "D6CIM: 60.4-TOPS/W, 1.46-TOPS/mm^2, 1005-Kb/mm^2 Digital 6T-SRAM-Based Compute-in-Memory Macro Supporting 1-to-8b Fixed-Point Arithmetic…," ESSCIRC 2023.