In-Memory Computing (IMC) architectures based on Static Random Access Memory (SRAM) can improve system performance and energy-efficiency dramatically. However, most of the existing SRAM-based implementations are designed for specific purposes like accelerating neural networks, which limits the application scenarios of IMC. In this paper, we propose DM-IMCA, a novel IMC architecture with two work modes for general purpose processing. It utilizes our proposed 9T bitcell based computational SRAM as the location to perform IMC operations. Besides, a new IMC Instruction Set Architecture (ISA) as well as an automated vector computing mechanism are also proposed to facilitate DM-IMCA’s programming and accelerate in-memory computing, respectively. The simulation results show that DM-IMCA can bring a performance increase by up to 257x, and SRAM energy saving by up to 3x, compared to a baseline system.

key words: Processing-In-Memory (PIM), In-Memory Computing (IMC), Logic-in-Memory (LiM), Computational SRAM, Data-intensive Computing, Vector Computing
Classification: Integrated circuits

1. Introduction

Nearly all modern computing machines are designed based on the famous von Neumann architecture, where memories and computing components are physically separated. Therefore, it’s inevitable to move data back and forth between memories and computing components before and after an ALU operation, incurring massive energy and execution time waste. Besides, there exists a growing speed gap between logic circuits and memory since the manufacturing process separation, known as the memory wall[1]. As a result, it brings about the von Neumann bottleneck[2].

One of those feasible solutions to memory wall is In-Memory Computing. The key idea is to move memory and computation as close to each other as possible[3]. Research on In-Memory Computing became flourished in 1990s due to the increasingly prominent von Neumann bottleneck. Most of the IMC designs[5, 6, 7, 8, 9, 10, 11] then concentrated on integrating computing abilities with DRAM, by either embedding logic on DRAM process or DRAM on logic process. However, this kind of approach incurred lots of problems mainly due to the difference between logic process and DRAM process[4]. Research on IMC[12, 13, 14, 15, 16, 17, 18, 19, 20, 21] resurrects these years due to the emergence of 3D stacking packaging and Non-Volatile Memories (NVM) like ReRAM. Nevertheless, In-Memory Computing based on these emerging technologies is not widely applied to real scenarios yet due to many unresolved problems. Recently, several IMC designs [22, 23, 24, 25] implement multiply-accumulate operations inside SRAM arrays in an analogue fashion to accelerate Artificial Intelligence (AI) algorithms. While this kind of approach can improve execution and energy efficiencies remarkably, it also has some shortages. For example, it incurs extra overhead due to the utilization of ADCs and DACs. In particular, it can’t be used for other scenarios where accurate computation is needed because it actually performs approximate computation along SRAM bitlines, which is tolerable for AI algorithms, but not for many other applications.

In this work, we pay our effort to explore and improve the SRAM-based IMC for general purpose processing in all aspects of circuit, architecture and software. To be specific, our contributions are as follows:

• We improve a bit-line computing technology[26, 27, 28] by employing a customized 9T SRAM bitcell to eliminate the read-write-disturb problem. We also implement several in-memory arithmetic and memory operations by introducing a few extra gates in SRAM. We then propose a novel computational SRAM macro called IMC-SRAM based on these circuitries.

• We then propose the novel IMC architecture DM-IMCA, which can run both in-memory computational and conventional instructions due to the utilization of dual work mode.

• We also propose a dedicated ISA to facilitate in-memory computing and DM-IMCA’s programming.

1School of Electronic Science and Applied Physics, Hefei University of Technology, Hefei 230009, China
2School of Electronics and Information Technology, Sun Yat-sen University, Guangzhou 510006, China
a) zhangzhang@hfut.edu.cn
b) yuzhiyi@mail.sysu.edu.cn

DOI: 10.1587/elex.17.20200005
Received January 1, 2020
Accepted January 14, 2020
Publicized January 23, 2020
• An automated vector computing mechanism is introduced to DM-IMCA, under which large scale parallel vector computation can be performed in SRAM automatically. Consequently, memory access, energy cost, as well as instruction memory footprint can be saved significantly. Meanwhile, the CPU can still go on execute conventional instructions.

2. IMC-SRAM: An Dual-mode Computational SRAM

2.1 9T SRAM Bitcell and In-Memory Logic Operations

A fabric in [26] proves that if two or more bitcells on the same bitlines of an SRAM column are read out simultaneously, the output of the related Sense Amplifier (SA) will inherently be an AND logic function of the stored bits of those activated cells, i.e., SRAM can work as a multi-input AND gate by allowing more than one bit cells to read at the same time. We further substitute the single-output SA with a double input-double output SA. Then four logic results, namely, AND, NAND, OR and NOR, can be obtained simultaneously, as depicted in Fig. 1(a). The XOR function can also be implemented by introducing an additional NOR-gate after the AND and NOR results.

However, conventional 6T SRAM suffers from the read-write-disturb problem. As shown in Fig. 1(b), Sharing the same gates (M5 and M6) and bitlines for read and write operations brings the read-write-disturb problem, which can become even worse when multiple bitcells on the same bitlines are read simultaneously. We eliminate this problem by employing 9T bitcell, which has separate sets of word-line and bit-lines for read and write, as shown in Fig. 1(c). The physical isolation of read and write path make it possible and more easy to optimize both read and write performance, as well as stability.

![Fig. 1. (a) Bitline logic operations. (b)-(c) Read/write comparison between 6T and 9T SRAM bitcells.](image)

2.2 Area-Efficient In-Memory Arithmetic Operations

A full Adder with reduced area can be implemented behind SAs by taking the advantage of the bit-line logic operations mentioned above. Let’s denote the two addends of a one-bit full adder as A and B, and the carry-in, carry-out and summation as \( C_i \), \( C_o \), and \( S \). A full adder can be expressed as follows:

\[
\begin{align*}
C_o(G, P) &= G + P \cdot C_i \\
S(G, P) &= P \oplus C_i
\end{align*}
\]

where \( G = A \cdot B \) and \( P = A \oplus B \). Both of \( G \) and \( P \) are available behind SAs, which brings about a 40% reduction of logic gates to realize a full adder, thus facilitating integrating multi-bit full adders into an SRAM macro.

Besides, operations of self-increment by 1 can also be easily implemented after the realization of addition. Denote \( M = \{M_31 \ldots M_2 M_1 M_0\} \) a 32-bit number, then

\[
\begin{align*}
P_0 &= M_0 \oplus 1 = \overline{M_0} \\
G_0 &= M_0 \cdot 1 = M_0 \\
P_i &= M_i \oplus 0 = M_i, \quad i \in [1, 31] \\
G_i &= M_i \cdot 0 = 0, \quad i \in [1, 31]
\end{align*}
\]

which equals to

\[
\begin{align*}
P &= \{M_{31}, \overline{M_{30}}, \ldots, M_2, M_1, \overline{M}_0\} \\
G &= \{31\overline{b}0, M_0\}
\end{align*}
\]

where \( \overline{M}_0 \) could be available at one of the SA outputs. Similarly, the self-decrement by 1 operation can be expressed as

\[
\begin{align*}
P &= \{\overline{M}_{31}, \ldots, \overline{M}_2, M_1, \overline{M}_0\} \\
G &= \{M_{31}, \overline{M}_{30}, \ldots, M_2, M_1, M_0\}
\end{align*}
\]

These operations can be realized by only adding some multiplexers behind SAs, bringing an area reduction compared to conventional implementations.

In addition, the subtraction operation can be realized in memory in two cycles, one for striving the opposite number of subtrahend, and the other for addition operation. The former composes of inverting the subtrahend bits and self-increasing by one, both are available in our system.

2.3 Architecture of IMC-SRAM

We propose a novel 32Kb (128 lines by 256 rows) 9T-SRAM-based macro called IMC-SRAM based on the above-mentioned circuits and technologies. Fig. 2 illustrates the microarchitecture of IMC-SRAM. IMC-SRAM utilizes three sets of row address decoders, two of which are for bitcell readout and the other for write-back, in order to perform logic operations and write the corresponding result back to array in a single cycle. In order to perform an IMC operation, source and destination operands should be stored in line vertically (A, B and C in Fig. 2 for example), which means only one common column address decoder is needed. Besides, the column address decoder can be configured to
select multiple words simultaneously for the purpose of allowing parallel computing. In addition, to obtain as many logic results in one operation, double input-double output SA pairs are used in IMC-SRAM. At the bottom of the diagram is a lite ALU called IM-ALU, which is composed of some logic gates to accomplish the rest part of logic and arithmetic operations behind SAs, and some multiplexers to chose the result to be written back into bitcell array according to the input opcode. In addition to functions mentioned in previous paragraph, a 1-bit bi-directional shifter is also implemented by introducing 3 gates per SRAM column.

IMC-SRAM can be configured to work in two modes, Normal mode and IMC mode, performing as a conventional memory and IMC accelerator, respectively. The red dotted arrow lines show the in-memory computing flow path in IMC-SRAM. In IMC mode, two rows will be read out simultaneously to compute, and then SAs will output the results, one of which will be chosen by the IM-ALU according to the input opcode. Finally, the chosen result will be written back to the bitcell array in the same clock cycle. By allowing multiple words in the same row being activated, IMC-SRAM can computes parallelly, like a SIMD CPU does.

3. DM-IMCA: Dual-Mode In-Memory Computing Architecture

3.1 Architecture of DM-IMCA
The overall architecture of DM-IMCA is illustrated in Fig. 3. It consists of a six-stage-pipelined RISC core, an instruction memory (IM), an IMC-Coprocessor (IMC-CP) and data memory (DM).

The data memory comprises of both regular SRAM and IMC-SRAM macros. The IMC-SRAM banks are mapped to a higher address space (≥ 2000H in our case), and the regular ones lower. This allows CPU to access data stored in lower address space when system is in IMC mode, which can obviously promote the CPU’s efficiency. We also optimize the CPU for this.

The RISC core is designed based on a classic five-stage-pipelined MIPS32 architecture from [28]. However, we insert a new stage called Pre-instruction-decoding (PID) into the CPU pipeline before the conventional Instruction-decoding (ID) stage to determine whether an instruction code is IMC or traditional type. If the code is not an IMC instruction, the system continues to run as any other von Neumann machines does, either executing operations in its ALU or retrieving/storing data from/to memory. Otherwise, it will be finally delivered to the IMC-CP, after which, it continues to run.

The IMC-CP is used to decode the IMC instructions from RISC core, and manage IMC operations in IMC-SRAM, including address and mode configuration, as well as IMC operations controlling. It also contains several configurable registers, as shown below:

- **R0**: Register to save operands’ column address.
- **R1-R3**: Register to save operands’ row addresses.
- **Rm**: Register to save system’s mode status.
- **Rn**: Register to save IMC-SRAM bank number.
- **Rv**: Register to save the vector length for vector computing in IMC-SRAM.

DM-IMCA is an hybrid architecture of both von Neumann and non-von Neumann types, because it not only can run as a conventional machine, which usually obeys the paradigm of load-compute-store, but also can perform operations in memory, which is obviously different from the conventional paradigm.

3.2 Instruction Set Architecture
Table I lists the proposed IMC instructions for DM-IMCA. The IMC instructions comprise of two types, configuration instruction and compute instruction. **addrCfg** is for address configuration, and **memCfg** is used to configure the bank number of IMC-SRAM macros to be used for computation by setting the register **Rn**. Memory operation instructions **mcopy** and **mnot** are implemented by setting the two source operands’ addresses to the same when performing **mand** and **mnand** operation, respectively. The rest have already been introduced in previous.

IMC instruction codes are all 32-bit long so that they can be compatible with traditional MIPS32 instructions. Table II lists the formats of different instruction types, with
3.3 DM-IMCA’s Dual-mode and IMC Work Flow

As mentioned above, both traditional and IMC instruction codes are stored together in IM. DM-IMCA is default running in Normal mode, in which situation the IMC-SRAM only works as an ordinary data memory and the whole system has no difference with a conventional von Neumann machine. An instruction will be sent to IMC-CP at PID pipeline-stage if it’s an IMC type. After that IMC-SRAM will be switched to IMC mode by IMC-CP, so is the whole system. The work mode will be switched to Normal mode again once IMC operation is finished.

Usually, one memCfg instruction is required to configure register Rm upon system power on, if more than one bank of IMC-SRAM are used. Then It needs to take two successive IMC instructions to perform a computation in memory. The first one should be a addrCfg, which is to be decoded by IMC-CP to set R0-R3 and Rm. Then IMC-CP will switch the system to IMC mode according to Rm’s value. The second one is a compute instruction. Upon receiving the instruction, IMC-CP initiates an IMC operation by sending operands’ addresses, vector length information, and opcode to IMC-SRAM. The IMC operation may last for several or even dozens of cycles, depending on the vector length and IMC-SRAM bank number to be used, IMC-CP will send a notification to CPU when IMC operation is finished and switches the system to Normal mode by resetting Rm to 0.

When the system is in IMC mode, CPU can still function as usual, provided that the next fetched code is a none-IMC type. Nevertheless, the CPU pipeline will have to suspend if the code try to access the computational memory space until the system is switched back to Normal mode.

3.4 Automated Vector Computing Mechanism

An automated vector computing mechanism is introduced to DM-IMCA. Words stored in the same row can be processed in parallel. Thus the more computational macros are utilized, the less cycle it needs to process a vector with a certain length, as Fig.4 indicates. IMC-CP can compute how many cycles it needs to accomplish an IMC instruction, according to the instruction’s word length field VL and register Rn in IMC-CP. With this mechanism, the system can process large scale vector computing automatically with few instructions, thus memory access, energy cost and instruction memory footprint can be saved significantly. Besides, CPU can continue to process other instructions while a large vector is being processed in IMC-SRAM.

| Table I. IMC Instruction Set Table |
|------------------------------------|
| **Configuration Instruction** | **Operation/Function** |
| memCfg | Rn configuration |
| addrCfg | Address registers configuration |

| Table II. IMC Instruction Formats |
|---------------------------------|
| **Type** | **Instruction** |
| 31-27 | memCfg Rn |
| 26-4 | 2-0 |
| op (11001) | Reserved |
| Address Configuration Instruction: addrCfg R3, R2, R1, R0 |
| 31-27 | 26-20 |
| 19-13 | 12-6 |
| 5-0 | op (11000) |
| R3 | R2 |
| R1 | R0 |
| Compute Instruction: opcode VL (VL: Vector Length) |
| 31-27 | 26-23 |
| 22-15 | 14-0 |
| op (11010) | function |
| VL | Reserved |

Fig. 5. Butterfly curves and RSNMs for 6T and 9T Bitcells

4. Evaluation

The IMC-SRAM is customized in a SMIC 55 nm process, and the rest is designed using RTL code. Cycle-accurate mixed-simulation on evaluation system is performed under Cadence AMS environment. Only one IMC-
SRAM macro (4k bytes) is used in evaluation system for a purpose of fair comparison with other SRAM-based IMC architectures. We also model a baseline system only containing a classic MIPS32 core and regular SRAM macros.

We first find out the Read Static Noise Margin (RSNM) to evaluate the stability of our employed 9T bitcell through some simulations where two bitcells on the same bit-lines are activated simultaneously. Fig. 5 plots the butterfly curves and RSNMs (the side of the largest square can be fitted in a butterfly curve) of both 9T and 6T bitcells, from which we can observe a larger RSNM of 9T bitcell, compared to that of 6T bitcell. This also demonstrates the improvement of read-write-disturb problem using 9T bitcell in IMC system.

Then some basic operations like add and nand with different vector length are ran on both DM-IMCA and baseline systems. The performance comparison is depicted in Fig. 6(a), which indicates the potential of DM-IMCA in processing large scale vector computing. For instance, up to 257x efficiency can be achieved for vector computing with a length of 256 length if 8 macros are used.

Besides, several data-intensive algorithms in cryptography and image processing fields are tested on both DM-IMCA and baseline, and the results of execution and power efficiencies are shown in Fig. 6(b) and Fig. 6(c), respectively.

(1) **Rotating Hash Function (RHF)** is one of hash functions used to map data of arbitrary size to fixed size values, and often used in digital signature or file check. RHF outputs a 32-bit number from 256- and 512-bit string in our experimental cases, indicated by RHF256 and RHF512 in Fig. 6(b), respectively.

(2) **One-Time Pad (OTP)** is a crypto algorithm where secret key could be used for only once. We encrypt a 256-bit number with OTP algorithm in experiment.

(3) **Binary Convolution (Bconv)** is the kernel of binarized neural networks (BNNs) where both feature maps and weights are quantized to 1 bit, either +1 or -1. We test Bconv with sizes of 16x16 and 32x32, indicated by Bconv16 and Bconv32 in Fig. 6(b), respectively.

(4) **Image GrayScaling (ISG)**, RGB2YUV and **Histogram Equalization (HE)** are tree algorithms used for image processing. We test these algorithms by applying them to a 28x28 image.

Statistics of power simulation show that it consumes ~3.9pJ energy to perform an 32-bit IMC operation in IMC-SRAM, bringing up to 3x efficiency over the baseline system, as Fig. 6(c) indicates. Evaluation results show that the area of CPU plus IMC-CP is 1.02x that of a classic MIPS32 core (0.05 µm²), and the area of IMC-SRAM macro is about 1.6x that of a 6T SRAM (about 0.034 µm² for a 4k bytes macro) due to the introduction of extra gates in bitcell and periphery.

Table III shows the comparisons of DM-IMCA with other SRAM-based state-of-the-arts. It should be noted that these works including ours didn’t use the same set of benchmarks, making it difficult to directly compare the absolute perform improvement with each other. Though, it still shows that DM-IMCA is competitive in accelerating a set of data-intensive applications. Besides, DM-IMCA is superior to other implementations targeting general purpose processing, i.e., [27] and [29], as far as the range of supported IMC.
operations and energy efficiency.

5. Conclusion

In this paper, we propose the DM-IMCA, in order to exploit and improve SRAM-based In-Memory Computing for general purpose processing from hardware to software levels. By performing operations in DM-IMCA, data transfer between CPU and memory could be reduced significantly, bringing a dramatical reduction on overhead including execution time and energy consumption.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China under Grant 61674049 and 61674173; in part by the Fundamental Research Funds for Central Universities under Grant PA2018GDQT0017.

References

[1] Wm. A. Wulf and S. A. McKee: “Hitting the memory wall: Implications of the obvious,” SIGARCH Comput. Arch. News 23 (1995) 20 (DOI: 10.1145/216585.216588).
[2] J. Backus: “Can programming be liberated from the von Neumann style? A functional style and its algebra of programs,” Communications (1978). (DOI: 10.1145/1283920.1283933)
[3] R. Das: “Blurring the lines between memory and computation,” IEEE Micro 37 (2017) 13 (DOI: 10.1109/MM.2017.4241340).
[4] P. Siegl et al.: “Data-centric computing frontiers: A survey on processing-in-memory,” Proc 2nd Int Symp. Mem. Syst. 2016 (DOI: 10.1109/ICCS.2016.7753257).
[5] D. Patterson et al.: “Intelligent RAM (IRAM): chips that remember and compute,” 1997 IEEE Int Solid-State Circuits Conf Dig Tech Pap (1997) 224 (DOI: 10.1109/ISSCC.1997.585348).
[6] M. Gokhale et al.: “Processing in memory: the Terasys massively parallel PIM array,” Computer 28 (1995) 23 (DOI: 10.1109/2.375174).
[7] M. Hall et al.: “Mapping irregular applications to DIVA, a PIM-based data-intensive architecture,” Proc 1999 ACM/IEEE Conf Supercomput (1999) 57 (DOI: 10.1109/SC.1999.10019).
[8] D. G. Elliott et al.: “Computational RAM: implementing processors in memory,” IEEE Des. Test Comput. 16 (1999) 32 (DOI: 10.1109/54.748803).
[9] G. Kirsch: “Active Memory: Micron’s Yukon,” Parallel Distrib. Process. Symp. 2003 Proc. Int. (2003) 11 (DOI: 10.1109/IPDPS.2003.1213195).
[10] Y. Kang et al.: “FlexRAM: Toward an advanced intelligent memory system,” 2012 IEEE 30th Int. Conf. Comput. Des. ICCD (2012) 5 (DOI: 10.1109/ICCD.1999.808425).
[11] M. Oskin et al.: “Active Pages: A computation model for intelligent memory,” Proc. 25th Annu. Int. Symp. Comput. Archit. (1998) 192 (DOI: 10.1145/279358.279387).
[12] Q. Zhu et al.: “A 3D-stacked logic-in-memory accelerator for application-specific data intensive computing,” 2013 IEEE Int 3D Syst. Integr. Conf. 3DIC (2013) 1 (DOI: 10.1109/3DIC.2013.6702348).
[13] J. Liu et al.: “Processing-in-memory for energy-efficient neural network training: A heterogeneous approach,” 2018 51st Annu. IEEE/ACM Int Symp MicroarchitMICRO (2018) 655 (DOI: 10.1109/MICRO.2018.00059).
[14] P. Chi et al.: “Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory,” Proc 43rd Int Symp Comput Archit. (2016) 27 (DOI: 10.1109/3007787.3001140).
[15] L. Nai et al.: “GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks,” 2017 IEEE Int Symp High-Perform Comput Archit. 2017 457 (DOI: 10.1109/HPCA.2017.54).
[16] D. I. Jeon et al.: “HMC-MAC: processing-in-memory architecture for multiply-accumulate operations with Hybrid Memory Cube,” IEEE Comput. Archit. Lett. PP (2017) 1 (DOI: 10.1109/LCA.2017.2700298).
[17] K. Hsieh et al.: “Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation,” 2016 IEEE 34th Int. Conf. Comput. Des. ICCD (2016) 25 (DOI: 10.1109/ICCD.2016.7753257).
[18] Q. Zhu et al.: “Accelerating sparse matrix-matrix multiplication with 3D-stacked logic-in-memory hardware,” 2013 IEEE High Perform. Extreme Comput. Conf. HPEC (2013) 1 (DOI: 10.1109/HPEC.2013.6670336).
[19] P. E. Gaillardon et al.: “The programmable logic-in-memory (PLIM) computer,” 2016 Des. Autom. Test Eur. Conf. Exhibit. DATE (2016) 427.
[20] Y. Chen et al.: “An efficient ReRAM-based inference accelerator for convolutional neural networks via activation reuse,” IEICE Electron. Express 16 (2019) 1009396 (DOI: 10.1587/elex.16.20190396).
[21] S. Gupta et al.: “RAPID: A ReRAM processing-in-memory architecture for DNA sequence alignment,” 2019 IEEE/ACM Int. Symp. Low Power Electron. Des. ISLPED (2019) 1 (DOI: 10.1109/ISLPED.2019.8824830).
[22] M. Kang et al.: “A multi-functional in-memory inference processor using a standard 6T SRAM array,” IEEE J. Solid-State Circuits (2018) 1 (DOI: 10.1109/JSSC.2017.2782087).
[23] M. Kang et al.: “A 19.4 nJ/desion 364K decisions/s in-memory random forest classifier in 6T SRAM array,” ESSCIRC 2017 - 43rd IEEE Eur. Solid State Circuits Conf. (2017) 263 (DOI: 10.1109/ESSCIRC.2017.8094576).
[24] M. Kang et al.: “A 48lpj/desion 3.4M decisions/s multi-functional deep-in-memory inference processor using standard 6T SRAM array,” ArXiv 161007501 Cs (2016).
[25] J. Zhang et al.: “In-memory computation of a machine-learning classifier in a standard 6T SRAM array,” IEEE J. Solid-State Circuits 52 (2017) 915 (DOI: 10.1109/JSSC.2016.2642198).
[26] Supreet Jeloka: “A 28 nm configurable memory (TCAM/BCAM/SRAM) using push-rule 6T bit cell enabling logic-in-memory,” IEEE J. Solid-State Circuits 51 (2016) 1009 (DOI: 10.1109/JSSC.2016.2515510).
[27] S. Aga et al.: “Compute caches,” 2017 IEEE Int Symp High Perform Comput Archit. (2017) 481 (DOI: 10.1109/HPCA.2017.21).
[28] D. A. Patterson and J. L. Hennessy: in Computer Organization and Design MIPS Edition: The Hardware/Software Interface (Newnes, 2013). 5th ed. 162.
[29] A. Agrawal et al.: “X-SRAM: enabling in-memory boolean computations in CMOS Static Random Access Memories,” IEEE Trans. Circuits Syst. Regul. Pap. 65 (2018) 4219 (DOI: 10.1109/TCSL.2018.2848999).
[30] A. Agrawal et al.: “Xcel-RAM: Accelerating binary neural networks in high-throughput SRAM compute arrays,” IEEE Trans. Circuits Syst. Regul. Pap. 66 (2019) 3064 (DOI: 10.1109/TCSL.2019.2907488).