FiCA: A Fixed-Point Custom Architecture FastICA for Real-Time and Latency-Sensitive Applications

Seyed Mohammad Reza Shahshahani and Hamid Reza Mahdiani

***Abstract*— Independent Component Analysis (ICA) is a common method exploited in different biomedical signal processing applications, especially in noise removal of electroencephalography (EEG) signals. Among different existing ICA algorithms, FastICA is a popular method with less complexity, which makes it more suitable for practical implementation. However, and due to its inherent computationally intensive nature, development of a custom FastICA hardware is the best way to utilize it in high-performance real-time applications. On the other hand, development of a custom hardware in a fixed-point manner is also a complex and challenging task due to the algorithm's iterative nature. Moreover, the algorithm intrinsically suffers from some convergence problems which prevents to be practically exploited in latency-sensitive applications. In this paper, a fixed-point fully customized, scalable, and high-performance FastICA processor architecture has been presented. The proposed architecture is developed in an algorithm-aware manner to mitigate the inherent FastICA algorithmic failures. The synthesis results in a 90 nm technology show that the design proposes a computational time of 0.32 ms to perform an 8-channel ICA with a frequency of 555 MHz. The performance-related measurements prove that its normalized throughput is 10 times more, compared to the closest rival.**

***Index Terms*— Artefact detection, brain-computer interface, EEG, FastICA, fixed-point arithmetic, independent component analysis, VLSI, word-length optimization.**

I. Introduction

INDEPENDENT Component Analysis (ICA) is a common mathematical transformation used for solving the Blind Source Separation (BSS) problem in many applications including speech, image, and biomedical signal processing [1], [2], [3], [4], [5], [6], [7], [8], [9], [10]. Mathematically, ICA tries to map the signals to some statistically independent components. Extracting these independent components is specifically useful in many biomedical applications such as separation of maternal-fetal electrocardiograms (ECG) [4], as well as removing task-irrelevant activities such as artefacts from functional magnetic resonance images (fMRI) [5], and electroencephalography (EEG) signals [6], [7], [8], [9], [10]. In the case of artefact removal, the artefacts are extracted as independent components which can be detected and eliminated.

There are several approaches to perform the ICA, among which two are the most common. The first approach named as Infomax [11] is based on information maximization which is derived from higher order statistics and has high computational complexity. The other one is the FastICA algorithm [12] which works based on negentropy maximization. Although it has lower computation complexity and is therefore relatively faster [12], real-time necessities when using this algorithm are still of concern and targeted by several real-time and high-performance hardware implementations [13], [14], [15], [16]. Moreover, considering the practical usage of FastICA, its other significant disadvantage which has not been previously addressed and prevents it to be utilized in latency-sensitive and real-time applications is that it inherently involves some iterative steps initiated by random values, which sometimes fail to converge based on the input data as well as the random number values.

Although the biomedical signals are normally acquired with low frequencies, the significant challenge in their denoising using FastICA is that it requires too much computations which should be executed very fast to meet the requirements of a real-time application. There are some hardware realizations of FastICA to address this issue. Van et al. [13] proposed an energy-efficient architecture using a level of parallelism in the weight update process for EEG signal processing. However, they adopted floating-point arithmetic. Yang et al. [14] designed a low-power FastICA architecture for seizure detection from Electrocorticography (ECoG) signals. They used an approximate systolic array eigenvalue decomposition (EVD) engine to preserve higher performance. Van et al. [15] proposed a cost-effective and variable-channel, yet floating-point arithmetic, hardware FastICA architecture by using Gram-Schmidt orthogonalization instead of EVD for signal whitening. Bhardwaj et al. [16] proposed a coordinate rotation digital computer (CORDIC)-based FastICA processor targeting the intrinsic redundancies of the original algorithm. A significant drawback of the designs proposed in [13] and [15] is that

Manuscript received 19 March 2022; revised 16 July 2022 and 10 September 2022; accepted 28 September 2022. Date of publication 12 October 2022; date of current version 20 October 2022. *(Corresponding author: Hamid Reza Mahdiani.)*

Seyed Mohammad Reza Shahshahani is with the Department of Electrical Engineering, Shahid Beheshti University, Tehran 19839-69411, Iran (e-mail: smr.shahshahani@gmail.com).

Hamid Reza Mahdiani is with the Department of Computer Science and Engineering, Shahid Beheshti University, Tehran 19839-69411, Iran (e-mail: mahdiani@sbu.ac.ir).

Digital Object Identifier 10.1109/TNSRE.2022.3213010

they utilized the floating-point arithmetic to overcome the high complexity of the algorithm hardware implementation, while it also drastically degrades the hardware cost and performance. On the other hand, although [14] and [16] have used fixed-point arithmetic, they have simply used a constant Word-Length (WL) of 32 and 14, respectively, throughout many of the system sub-blocks. Moreover, even for the selected single WL, they have not provided any WL optimization or noise analysis details which is a major requirement for achieving the best cost-performance trade-off in any fixed-point custom hardware. Also, to preserve the higher performance, [14] has applied some significant simplifications in the EVD calculation process (as one of the most computation intensive steps of FastICA) which reduces the overall output precision. Finally, the common significant and most computation intensive drawback of all the mentioned designs is that none of them have considered or even notified about the inherent failure-prone nature of the FastICA algorithm, which arbitrarily results in delayed calculation of the independent components which might violate fundamental requirements of some real-time and latency-sensitive applications. To address the above significant shortcomings, FiCA (a Fixed-point Custom Architecture FastICA) processor is proposed in this paper, with the following novelties:

- *Fully Customized Fixed-point Implementation.* An all-inclusive numerical analysis has been performed throughout all blocks and sub-blocks of the proposed architecture. The result is a fully customized fixed-point architecture with 6 different WLs for specific regions of the architecture, targeting to achieve the best performance while paying the least possible costs.
- *Scalability.* The proposed architecture is augmented with substantial levels of scalability in its different sections to accommodate with different applications with various number of channels, sampling frequencies, and so on, which demand for different performances. FiCA can be configured at design time to meet the desired cost-performance trade-off based on real-time and latency necessities of different applications.
- *Higher Performance.* The inherent parallelism of the algorithm inside and between its different blocks is extracted to maximize the overall system throughput while retaining balance between different blocks. To support the activated parallelism of the blocks, a new Parallel Matrix Manipulation Memory (PM^3) module is also introduced as the core communication mechanism of the architecture to simplify and accelerate the data movement between different parallelized blocks. By means of the explained precautions, the resulted hardware achieved orders of magnitude better performance than the best existing rivals.
- *Suitability for latency-sensitive applications.* A part of the scalability embedded in the design, is mainly devised to overcome the inherent failure-prone nature of the algorithm. This resulted in an algorithm-aware architecture which can be exploited for latency-sensitive applications.

The next sections of the paper are arranged as follows. Section II briefly reviews the theoretical background of the original FastICA algorithm. Section III introduces the FiCA details, its latency, throughput, and numerical precision analysis. Section IV includes the experimental results and a detailed comparative study to demonstrate the superiority of FiCA to the existing state of the art designs. Section V concludes the paper.

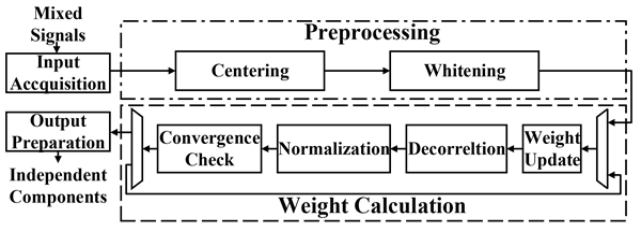

Fig. 1. FastICA algorithm block diagram.

II. FastICA Theoretical Background

In BSS, it is assumed that there are n independent sources as

$$X = AS \tag{1}$$

where X and S are $n \times m$ matrices representing m samples of n observed channels of the input signal as well as its extracted statistically independent components, and A is an $n \times n$ mixing matrix. Assuming there is at most 1 gaussian source, the algorithm tries to find S from X by estimating the $n \times n$ demixing matrix W (equals to A^{-1}). So, the source signals can be calculated by the unmixing model of ICA as below

$$S = W^T X \tag{2}$$

where W, the weight matrix, is an $n \times n$ matrix.

FastICA calculates the weight matrix by maximizing non-Gaussianity estimated by negentropy. As indicated in Fig. 1, the algorithm mainly consists of two steps: 1) preprocessing, and 2) weight calculation. The details of each step are as follows.

A. Preprocessing

To reduce the complexity and the number of iterations in the iterative "weight calculation" steps, the signals are first preprocessed by "centering" and then "whitening" of the input signals as explained in more details in the following:

1) Centering: In this step, the mean of each channel of signal is calculated and subtracted from it. So, for the i^{th} channel of signal the centering process is as

$$\bar{x}_i = x_i - E\{x_i\} \tag{3}$$

where $\hat{x}_i$ is the centered (zero-mean) version of x_i signal and Ex_i is its mean.

2) Whitening: This step contains three sub-steps. First, the covariance matrix of the signals is calculated and then decomposed using EVD

$$C_X = E\left\{\bar{X}^T \bar{X}\right\} = EDE^T \tag{4}$$

where $\bar{X}$ is the matrix containing n channels of zero-mean signals, E is the matrix of eigenvectors, and D is the diagonal

matrix of eigenvalues. Finally, using E and D, the whitening matrix is calculated and applied to the centered signals as

$$Z = D^{-1/2} E^T \bar{X} \tag{5}$$

This transformation transforms the centered signals $\bar{X}$ to the signals Z of which the covariance matrix is the identity matrix.

B. Weight Calculation

As indicated in Fig. 1, the weight calculation step accepts the whitened signals to find the demixing weight vectors in an iterative training process. As mentioned before, the algorithm uses the non-Gaussianity criterion measured by negentropy to find the weight vectors. It tries to find a weight vector w which projects the signals into a space where non-Gaussianity is maximized. The negentropy J is approximated by

$$J(w^T Z) \propto [E\{G(y)\} - E\{G(v)\}]^2 \tag{6}$$

where G is a non-quadratic function defined as

$$G(u) = \frac{1}{a} log\ cosh(au) \tag{7}$$

where a is a constant. The iterative process starts with choosing an initial (e.g. random) unit vector w. Then, w is updated as

$$w^+ = E\{Z\left[g\left(w^T Z\right)\right]^T\} - E\{g'(w^T Z)\}w \tag{8}$$

where g is the derivative of G. Also, to prevent vectors of different components from converging to the same maxima, they need to be decorrelated. The deflationary method decorrelates vectors sequentially based on Gram-Schmidt orthogonalization [12]. This means that before each iteration the projection of the vector being calculated on the previously calculated vectors is subtracted from it and then normalized as

$$w_{k+1}^+ = w_{k+1} - \sum_{j=1}^{k} \left(w_{k+1}^T w_j\right) w_j \tag{9}$$

$$w = \frac{w^+}{\left\|w^+\right\|} \tag{10}$$

where w_0 is initialized with small random values.

This update process continues until the old and new weight vectors, (w and w^+), are in the same direction, i.e., their dot product converges to 1. However, if this convergence criterion is not satisfied in a predetermined number of iterations, the weight calculation should be stopped, rescheduled, and then restarted with another initial random vector. Having calculated all n weight vectors, the demixing matrix W is constructed. The output preparation step, then computes the n ICA components by applying the matrix W to the whitened data based on (2).

III. The Proposed FiCA Processor for Fixed-Point VLSI Implementation

To realize the dataflow in Fig. 1, FiCA, a high-performance and algorithm-aware hardware realization of the FastICA algorithm, is proposed in this paper. The details of the architecture building blocks are presented and discussed first, and the whole FiCA is then explained and its throughput and numerical precision analysis are discussed in details.

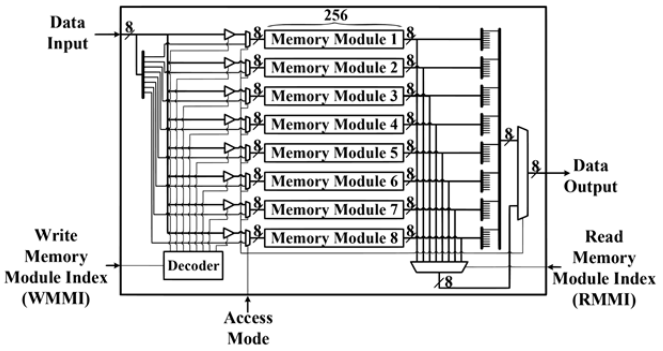

Fig. 2. Parallel matrix manipulation memory (PM^3).

To demonstrate the range and precision of the values throughout the incoming sections, the fixed-point notations of [17] are used. Thus, $S(a, b)$ represents a signed value with a WL of $a+b+1$ where a and b are the WLs of the range and precision parts of that value, respectively. Similarly, $U(a, b)$ represents an unsigned value with a WL of $a+b$.

A. Building Blocks of FiCA

In the following sub-sections, detailed structures and explanations of the processor modules are provided with a one-to-one correspondence with the data flow steps discussed in section II.

1) PM^3 Main Memory: The structure of the FiCA main memory is illustrated in Fig. 2 for an 8-channel processor (i.e. $n = 8$) without loss of generality. It is named as Parallel Matrix Manipulation Memory (PM^3) and consists of 8 separate 8-port memory modules, each containing 256 (i.e., $m = 256$) registers of the format $S(0, WL1-1)$, to store the normalized values of the 8 channels of signals. Each memory module has also two address ports regarding the read and write accesses. A custom-designed circuitry is also provided which makes it possible to access the stored values in two different modes: "8-parallel column access" and "8-parallel row access", where the "Access Mode" is 1 or 0, respectively. The first mode enables parallel access to 8 elements of one memory module at a single clock cycle, whereas the latter provides parallel access to a similar element from all 8 memory modules. Moreover, indicated in Fig. 2, the PM^3 module supports simultaneous read and write operations whose locations are defined using the RMMI and WMMI signals, respectively.

2) Preprocessing Unit: As mentioned earlier, this block mainly consists of the centering and whitening sub-units which are realized as follows.

a) Centering sub-unit: Fig. 3-a demonstrates the internal structure of this unit. It receives the signals and subtracts the mean of each signal from it as follows

$$\bar{x}(i) = x(i) - E\{x_i\} = x(i) - \left(\frac{\sum_{j=1}^{256} x(j)}{256}\right) \tag{11}$$

This means that for each channel, one accumulator (ACC), one 8-bit shift-to-right (division by 256) and one subtractor are necessary. As indicated in the figure, this unit consists of

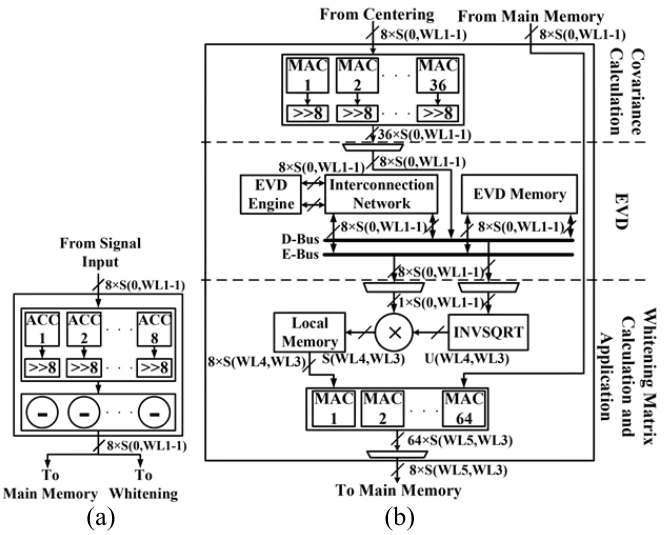

Fig. 3. Preprocessing block details: (a) centering and (b) whitening sub-units.

8 ACCs, 8 arithmetic shifters, and 8 subtractors to increase parallelism and process the 8 input channels concurrently. It both receives and outputs signals with $S(0, WL1-1)$. The output of the unit feeds both the main memory and the whitening unit as indicated in Fig. 3-a.

b) Whitening sub-unit: Fig. 3-b shows the internal structure of this unit which is divided into three parts, i.e., "covariance calculation", "EVD", and "whitening matrix calculation and application" as separated by dashed lines in the figure. This unit is responsible for signal whitening. First, based on (4), the covariance matrix of the signals is calculated. Since C_X is symmetric, only 36 distinct values should be calculated. This can be implemented in a scalable manner. To maximize the performance, here, 36 MACs and 8-bit shifters have been used to compute all matrix entries in parallel (Fig. 3-b, upper part). The calculated values of the covariance matrix are then stored in a memory named as eigenvalue decomposition (EVD) memory. This memory consists of two register banks each one containing 8 register files [18] to store values of the format $S(0, WL1-1)$. One of the register banks dedicated to storing the calculated matrix of eigenvalues D by the EVD is filled with the covariance matrix, while the other one which is for the matrix of eigenvectors E is initialized as the identity matrix. As shown in the figure, the connection to the EVD memory is facilitated by means of two data buses $D-Bus$ and $E-Bus$.

Upon storing the covariance matrix values in the EVD memory, the EVD engine starts its task (Fig. 3-b, middle part). The details of the EVD engine are shown in Fig. 4. Here, we used a simplified single processing element version of the singular value decomposition (SVD) processor proposed in [18]. Based on the Jacobi algorithm [19], the SVD processor receives 2×2 sub-matrices of the 8×8 covariance matrix stored in the register bank dedicated to the matrix of eigenvalues. The calculated rotation parameters, C_ϕ and S_ϕ, are stored in a local memory and after all rotation parameters are calculated for the whole matrix, they are applied both to the matrix itself and the matrix of eigenvectors, a and u, respectively. The results, a' and u', are then stored back in the EVD memory. As shown in Fig. 4, because of symmetry, the adder-subtractor in the sine/cosine calculator unit is replaced with a 1-bit shift-to-right and a subtractor. Also, for the same reason, there is only one set of CORDIC modules calculating the rotation parameters. Moreover, to increase the accuracy, the number of adder/subtractors in the CORDIC modules have been doubled here, increasing to 4 CORDIC iterations per EVD clock cycle. Due to the existing special circuitry which facilitates reading from and writing to the EVD memory, its clock speed is half of that of the I/O operations [18].

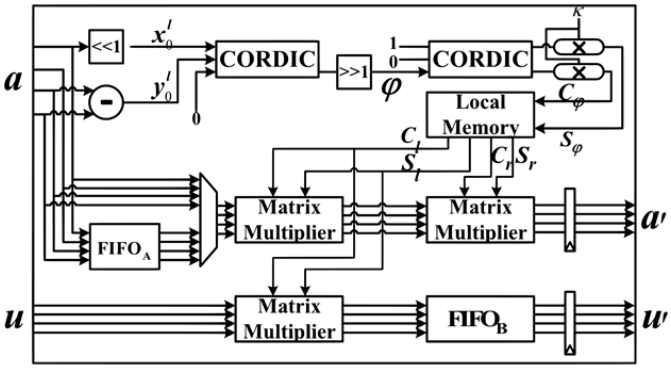

Fig. 4. Modified processing element of the EVD processor of [18].

Upon finishing the EVD process, the eigenvalues and eigenvectors are used as in (5) to whiten the signals (Fig.3-b, lower part). The matrix of eigenvalues D is diagonal. So, it only involves eight inverse square root calculations.

The inverse square root is realized by the Newton-Raphson algorithm [20] as

$$y(k+1) = \frac{y(k)}{2}(3 - x_{in}\, y(k)^2) \tag{12}$$

where x_{in} is the input. Starting from an initial $y(0)$, after some iterations $y(k)$ converges to the inverse square root of x_{in}. Since in this work the eigenvalues are set to be values of less than 1, we have set the initial value $y(0) = 1$. The equation needs 3 multiplications and 1 subtraction. One inverse square root module has been provided to process all eigenvalues sequentially. It should be noted that the inputs to this module are of $U(0, WL1-1)$ but since the square root of values less than 1 are values larger than 1, the output of the module is of $U(WL4, WL3)$ where $WL4$ and $WL3$ represent the range and precision, respectively.

Since the calculated $D^{-\frac{1}{2}}$ is diagonal, its multiplication with E^T reduces to multiplication of each diagonal value of the former with the corresponding row of the latter. To this end, one multiplier is provided to do the job sequentially. Since in each multiplier one input is of $U(WL4, WL3)$ and the other one is of $S(0, WL1-1)$, the output is logically taken as $S(WL4, WL3)$.

Having calculated $D^{-\frac{1}{2}}E^T$, it should be multiplied with the centered data $\bar{X}$. To maximize the performance and exploit the inherent parallelism of matrix multiplication as much as possible, as shown in Fig. 3-b, 64 MACs are used here. The main memory module switches to its 8-parallel access mode at this stage. Since reading from and writing to the main memory are independent, the outputs of the MACs are written in the

memory while the next block of data is read and processed by the MACs, simultaneously.

This module has two input sources, one coming from the main memory which is of $S(0, WL1-1)$ and the other from prior multipliers which is of $S(WL4, WL3)$. However, since the module consists of MACs and shifts and both inputs are signed, the output is taken as $S(WL5, WL3)$.

3) Weight Calculation Unit: This unit receives the whitened output signals of the preprocessing block and exploits an iterative process to calculate the weight vectors needed for separation of the signals. It is mainly comprised of 3 different local memory modules (i.e., old weight memory, new weight memory, and weight matrix) and 5 computational sub-units (i.e., weight vector initialization, weight update, orthogonalization, normalization, and convergence check). The overall arrangement and interconnections of the block in FiCA is shown in Fig. 8 and will be discussed later. The details of each sub-unit are as follows.

a) Weight calculation unit memory modules: The old and new weight memories consist of 8 registers for holding the current and updated weight vectors, respectively. Moreover, upon convergence, each weight vector is stored in the weight memory matrix, stacked column-wise next to the previously calculated ones. So, the weight matrix memory module consists of 64 registers, initially filled with zeros. These memory modules are connected to the processing blocks via three separate buses. The data is stored in all the three memory modules with the format of $S(0, WL2-1)$.

b) Weight vector initialization sub-unit: This unit contains a random number generator using Linear-Feedback Shift Register (LFSR) sequence generator [21]. The generated random numbers are stored in the old weight memory. The initial random weights are considered to be of $S(0, WL2-1)$.

c) Weight update sub-unit: As shown in Fig. 5, this unit receives the weight vector stored in the old weight memory. Substituting (7) in (8) results in the weight update function as

$$w^{+} = \frac{1}{256}\left(Z\left[\tanh(w^{T}Z)\right] - \sum_{i=1}^{256} 1 - \tanh^{2} w^{T}Z\right) w \tag{13}$$

To realize this equation, first $w^{T}Z$ should be calculated. w^{T} is 1×8 and Z is 8×256. This means there are 256 independent vector multiplications which can be performed in a scalable manner. Here, to take advantage of the 8-parallel row access mode of the memory, 8 MACs are employed to achieve the highest performance. The results of the MACs are stored in a local memory which is accessed by the Tanh unit which calculates the hyperbolic tangent function for the resulting 256-element vector. Hyperbolic tangent function described as $\tanh x \approx ax + b$ is realized using all piecewise linear approximation (a an b are real-valued coefficients).

The Tanh unit contains a look up table (LUT) storing the coefficients of the format $S(0, WL2-1)$, 10 comparators to check for the ranges, 1 multiplier, and 1 adder. At each clock cycle, it receives and outputs one value. This is used to the benefit of calculating the second half of (13) which needs one input at a time. A multiplier is used to calculate $\tanh^2(w^{T}Z)$. This value is always less than 1. So, its subtraction from 1 can be seen as complementing the value. An accumulator is used to

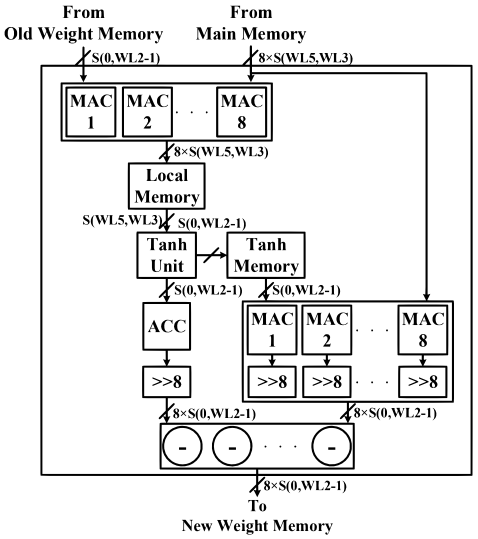

Fig. 5. Weight update sub-unit.

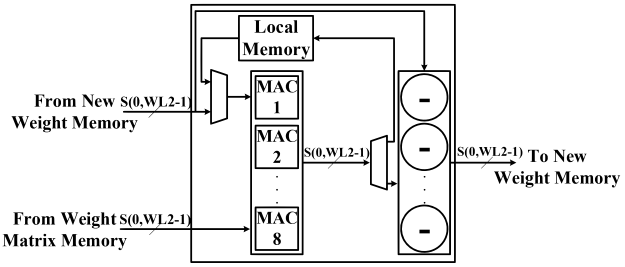

Fig. 6. Orthogonalization sub-unit.

calculate the summation. The result is then passed to an 8-bit shifter to compensate for the division by 256. Calculation of the first half of (13) requires the 8-parallel column access mode of the memory; thus, it starts after all hyperbolic tangent values are calculated. It is realized by means of 8 MACs followed by 8, 8-bit shifters which receive the whitened signals and the hyperbolic tangent results stored in the related local memory. Finally, 8 subtractors are used to do the subtractions in parallel. The updated vector is then stored in the new weight memory.

d) Orthogonalization sub-unit: Fig. 6 shows the details of this sub-unit. It performs (9) and orthogonalizes the current weight vector with respect to the previously calculated weight vectors. It accesses the new weight memory and the weight matrix memory to do the task. To increase the performance while maintaining uniformity, (9) is reshaped as

$$w_{k+1}^{+} = w_{k+1} - B^{T}Bw_{k+1} \tag{14}$$

where B is the 8×8 matrix stored in the weight matrix memory. As shown in Fig. 6, for hardware realization, there are 8 MACs which first calculate $B^{T}w$, and then multiply the results by B. Also, 8 subtractors are used to perform the subtractions in parallel. The result is then written to the new weight memory.

e) 0: Normalization sub-unit This unit accesses both old and new weight memories to perform (10). Vector normalization, as seen in Fig. 7, is realized using a MAC to calculate the sum of squared values of the vector, an inverse square root block similar to the one used in the whitening block, and 8 multipliers to apply the calculated inverse square root to the

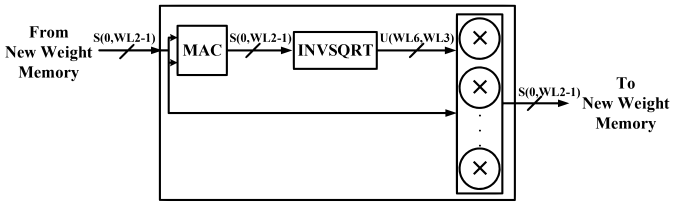

Fig. 7. Vector normalization sub-unit.

vector in parallel. As shown in Fig. 7, the input to the inverse square root module is of $S(0, WL2-1)$ but the output is of $U(WL6, WL3)$.

f) Convergence check sub-unit: This unit performs the dot product of the updated weight vector with its old version (stored in the new weight memory and the old weight memory, respectively) to check for convergence. This is simply implemented using one MAC and a comparator which checks if the result of the MAC is close enough to 1 (i.e., the vectors are in the same direction) based on a preset value. If converged,it sends a control signal to the controller to reset the weight update unit and activate the vector initialization unit to generate the next initial vector. Otherwise, it signals the controller to check if the maximum number of iterations (i.e. 300) is reached. If that is the case, it means the algorithm has failed; so, the controller reschedules the operation, resets the weight update unit and the new and old weight memory modules, and activates the weight initialization unit to generate another initial vector. If neither convergence has reached, nor has the maximum iteration number, the updated weight vector is written into the old weight memory and another iteration starts.

4) Data Separator: Upon calculating all weight vectors, the data separation unit accesses the main memory and the weight matrix memory. This block, which is inherently scalable, realizes (2) by means of 8 MACS outputting one sample of the separated signals every eight cycles.

B. Whole FiCA Architecture

Fig. 8 depicts the whole FiCA architecture. Based on Fig. 1, its four main units, i.e., input preparation, preprocessing, weight calculation, and output preparation units, are highlighted with light gray-shadow boxes. Also, due to multiplicity of the WLs, which are customarily exploited for different system blocks, and to provide a global vision on the computation precision of different blocks, three regions associated with $WL1$, $WL2$, and $WL3$ are indicated in the figure with different dashed lines. $WL4$, $WL5$, and $WL6$ are excluded from the figure because of their less importance (explained later), to avoid any confusion.

Based on the proposed architecture, the FastICA calculation starts by externally writing 256 samples of the 8 input signals into the PM^3 main memory by the input preparation unit. As shown, the access to the main memory is facilitated using a data-bus with a width of $8 \times WL1$, named as $M-Bus$. The 8 channels of signal enter the main memory, and simultaneously, the preprocessing block which first centers, and then, whitens the input samples. When the whitened data is ready in the memory, weight calculation unit starts. As shown in the figure, the architecture supports multiple weight calculation units working in parallel to handle algorithmic failures as described before. The weight matrix memory can be accessed by all the weight calculation units and the unit which converges first, writes the calculated weight vector into that memory. The number of these weight calculation units can be set at design time at user's demand as will be explained later. When all weight vectors are calculated, the data separator in the output preparation unit accesses both the PM^3 and the weight matrix memory to compute the independent components.

1) FiCA's Latency and Throughput Calculations: In this section, the number of clock cycles each unit takes to finish its task is accurately presented and finally added up to analytically calculate the latency and throughput of the whole FiCA processor. The calculation is done assuming no algorithmic failures. Considering 8 channels of signals, each one with 256 samples, it takes 256 clock cycles for the signals to be completely stored in PM^3. Simultaneously, the input samples also enter the centering unit for mean calculation (as indicated in Fig. 3-a and 3-b). Then, using the calculated mean, the centering unit subtracts the mean values from the signals. Covariance calculation starts one clock cycle after the beginning of mean subtraction resulting in another 256 clock cycles and then, the results take 8 cycles to be written in the EVD memory. For the EVD, as stated in section III.2-b, clock frequency is half of that of the other blocks. So, considering the system clock frequency, it takes 2688 cycles to do the task. The number of cycles regarding the calculation of inverse square root of the 8 eigenvalues is different for each eigenvalue. However, it experimentally takes on average 11 iterations. Since each iteration of the inverse square block takes 6 cycles (due to the pipeline registers in the block), the overall number of clock cycles in this regard is at most $8 \times 11 \times 6 = 528$ cycles. Moreover, applying each inverse square root result to its respective row of the matrix of eigenvectors takes 8 cycles, resulting in 64 cycles overall. On the other hand, applying the results to the signals, takes 256 cycles. Adding up all the mentioned clock cycles, the preprocessing takes 4057 (=256+256+1+8+2688+528+64+ 256) cycles to finish.

Upon finishing the preprocessing, the weight calculation process starts. Weight initialization takes 8 cycles to generate each initial weight vector, being 64 for 8 weight vectors. The weight update and normalization take, on average, 624 clock cycles per iteration. The convergence check unit takes 8 more clock cycles. So, assuming N_{Iter} number of iterations, the whole weight calculation process takes $(N_{Iter} \times 632) + 64$ clock cycles. Finally, the data separator takes 8 clock cycles to calculate each output sample, resulting in $2048(=256 \times 8)$ cycles. Therefore, adding up all calculated clock cycles, the whole number of clock cycles the ICA takes to do one full ICA (i.e., N_{cyc}) and the total calculation time of the processor (i.e., T_c in terms of seconds) can be calculated by

$$N_{cyc} = 4057 + N_{Iter} * 632 + 64 + 2048 \quad (15)$$

$$T_c = \frac{N_{cyc}}{f_{clk}} \quad (16)$$

where f_{clk} is the system clock frequency.

2) FiCA's Numerical Precision and Quantization Noise Analysis: Numerical precision analysis is a very complicated but

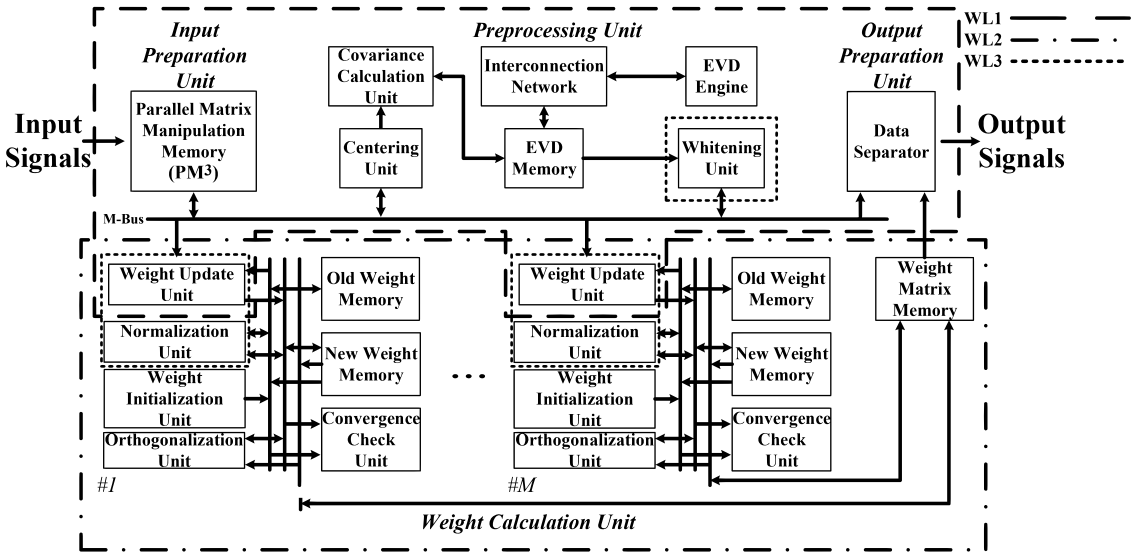

Fig. 8. Architectural details of FiCA.

significant step in custom fixed-point implementation of digital signal processing systems. Although fixed-point implementation highly improves hardware cost and performance, it can highly affect system quantization noise, accuracy, and functionality, especially in computation intensive and iterative algorithms. This section provides a block-level numerical precision analysis of the FiCA processor, in detail.

WL is the main parameter affecting system accuracy and cost in fixed-point custom implementations. As mentioned in section III-a explanations and also in Fig. 8, the FiCA processor utilized six different WLs ($WL1$, $WL2$, $WL3$, $WL4$, $WL5$, and $WL6$) in its different blocks to achieve the maximum performance with the least possible cost. To perform a fine-grained WL optimization and find the optimum values of these WLs, the Mendeley EEG dataset [22] is used which includes 54 different EEG recordings with 19 channels with artificial EOG artefacts. We have used 8 channels covering the brain motor area: F3, F4, C3, C4, P3, P4, Fz, and Pz.

Among the six defined system WLs, $WL4$ and $WL5$ are utilized to demonstrate the range of values in the whitening sub-unit (Fig. 3-b). $WL6$, also, represents the range of values in the vector normalization sub-unit (Fig. 7). These WLs are calculated by means of a worst-case analysis on the range of the values in the floating-point model, in a manner to avoid any probable overflow as the major source of accuracy loss. Based on the performed simulations, the minimum acceptable $WL4$, $WL5$, and $WL6$ are determined as 9, 3, and 10, respectively.

The other three WLs (i.e., $WL1$, $WL2$, and $WL3$) determine the precision of some system blocks. $WL1$ resembles the precision in input preparation and output preparation as well as the preprocessing blocks (Fig. 3), while $WL2$ represents the precision of weight calculation units (Fig. 5, Fig. 6, and Fig. 7), and $WL3$ stands for the precision of the weight update (Fig. 5) and vector normalization (Fig. 7) units. The values of these parameters significantly affect system cost and quantization noise trade-off and should be accurately optimized in a structured manner. To better demonstrate the significance level of each of these WLs by means of number and importance of its respective sub-block, Fig. 8 depicts their

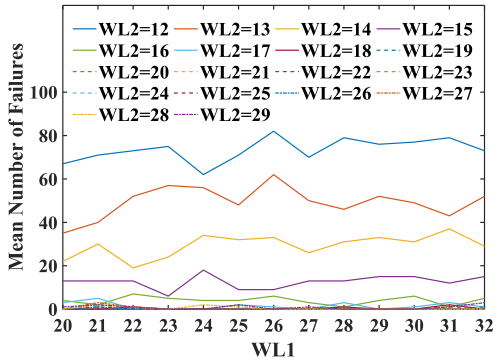

Fig. 9. Mean number of total failures.

region of effect using different legends. It can be inferred from the figure that the order of these WLs based on their impact on system area is $WL1$, $WL2$, and finally $WL3$. To extract the system sensitivity against each one of these WLs from different aspects (i.e., quantization noise and algorithmic failure), the whole system bit-true model is developed and exhaustively simulated while $WL1$, $WL2$, and $WL3$ values are in the ranges of 20 to 32, 12 to 29, and 12 to 21, respectively. To achieve the corresponding results for each WL configuration, the simulation is repeated 100 times and the mean of all runs has been considered for the WL optimization process. The results of this all-inclusive and fully customized simulation are exploited to determine the effects of $WL1$, $WL2$, and $WL3$ on 1) weight calculation failure, and 2) system output accuracy for different WLs as follows.

a) Effects of system WLs on weight calculation failure: The performed exhaustive simulation results clearly demonstrate the very high effect of $WL2$ on the weight calculation process failure, while the other two WLs are ineffective or have much smaller effects. Fig. 9, as an instance, demonstrates a part of the simulation results which shows the mean number of failures at the weight calculation process for different $WL1$ and $WL2$ values while $WL3 = 12$. Based on the

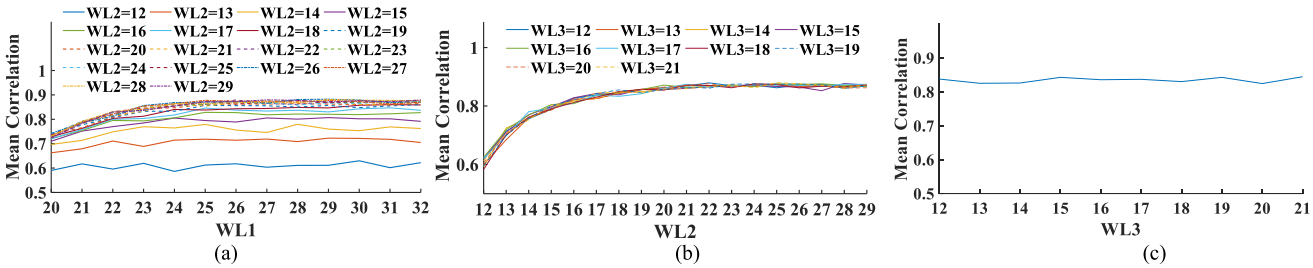

Fig. 10. Mean correlation (best case analysis) (a) different *WL*1s and *WL*2s while *WL*3 = 12, (b) different *WL*2*s* and *WL*3*s* while *WL*1 is fixed to 26, (c) different *WL*3*s* while *WL*1 and *WL*2 are fixed to 26 and 24, respectively.

simulation result, $WL2$ should not be less than 17 to control the number of failures below an acceptable threshold.

b) Effects of system WL on system quantization noise and output accuracy: The simulation results show that $WL1$, then $WL2$, and finally $WL3$ have the largest impacts on the system noise and output accuracy. This is also supported by the previous findings about the impact of WLs on area. This means that $WL1$ which is related to larger area and more system blocks is also expected to more affect system noise and accuracy. Fig. 10 shows the average mean correlation of the bit-true model outputs (for different WLs) with those of the MATLAB double-precision floating-point FastICA for some special values of WLs. Fig. 10-a illustrates a part of simulation results which correspond to the system output correlations for different $WL1$ and $WL2$ values for $WL3 = 12$. Choosing a WL of 26 seems to be suitable. Fig. 10-b also demonstrates the system output precision for different $WL2$ and $WL3$ values while $WL1$ is fixed to 26. Selecting a WL of 24 both reduces the system quantization noise and number of algorithmic failures below an acceptable threshold (based on Fig. 10-b and 9). Finally, Fig. 10-c illustrates system output correlations when $WL1$ and $WL2$ are fixed to 26 and 24, respectively. It shows that a $WL3$ of 14 is a suitable choice. Therefore, and as the result of exhaustive simulation, the optimum WLs are $WL1 = 26$, $WL2 = 24$, and $WL3 = 14$, considering 1 ore more guard bits to support probable data dependency.

IV. Experimental Results

In this section, a significant explanation about the suitable number of weight calculation units in the architecture for any specific application is provided first. Next, some convenient functional validation results of the FiCA processor by means of both synthetic and real EEG signals are presented. The processor's ASIC synthesis results are discussed and compared with some of the state-of-the-art designs at the end.

A. Suitable Number of Weight Calculation Units for a Latency-Sensitive Application

As mentioned before, one FiCA's significant specification is that it supports multiple weight calculation units based on latency requirements of each application. Fig. 11 shows cumulative probability of failures that might happen in the weight calculation process (and, therefore, the whole FastICA) based on the number of exploited weight calculation units in the architecture. The figure shows that augmentation of the processor with only one calculation unit, provides the output within the expected time in 89.5% of the times, while in 10.5% remaining times, the result will be ready after one or more extra weight calculation iterations due to nonconvergence of the algorithm as explained in section II. The probability of weight calculation success increases to 95. 2% and 97% with two and three weight calculation units in the architecture, respectively. The simulation results show that the algorithm needs 50 weight calculation units to theoretically achieve the 100% probability.

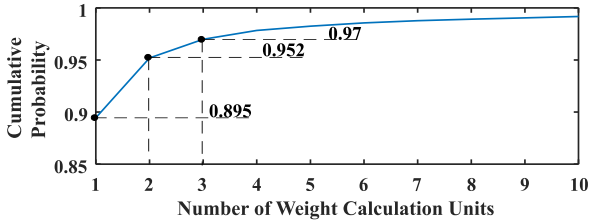

Fig. 11. Cumulative probability of failures in the FastICA weight calculation.

B. Functional Validation

To validate the functionality of the proposed fixed-point ICA processor and measure the effects of the six WLs on system quantization noise and output accuracy, 8 channels of synthetic signals are individually generated and mixed, and the mixture is then fed into the ICA processor, based on the approach proposed in [13]. Fig. 12-a and 12-b show the eight original signals (in blue), as well as their counterparts in the outputs of the double precision MATLAB floating-point FastICA toolbox, and the fixed-point FiCA, respectively (both in red). The value next to each signal demonstrates the correlation between the output signal and its original version as a measure of correct functionality. Fig. 12-a shows that even the FastICA MATLAB floating-point model can recover the original signals with the correlations of about 0.9788 to 0.9997. On the other hand, Fig.12-b shows that, the FiCA provides meaningfully similar correlations (between 0.9554 and 0.9995) with respect to the floating-point ICA, due to the effort put on optimizing and choosing the right WLs.

To also demonstrate FiCA's capabilities in recovering natural signals in a real application, its processing results on a set of eight EEG signals (used for numerical precision analysis in section III) are also included in Fig. 13. Fig. 13 shows the decomposition results of the fixed-point design (in red), overlaid on the MATLAB floating-point FastICA outputs (in blue). The corresponding correlation values have also been shown next to each plot. The results show that FiCA has successfully decomposed the two artefacts as the

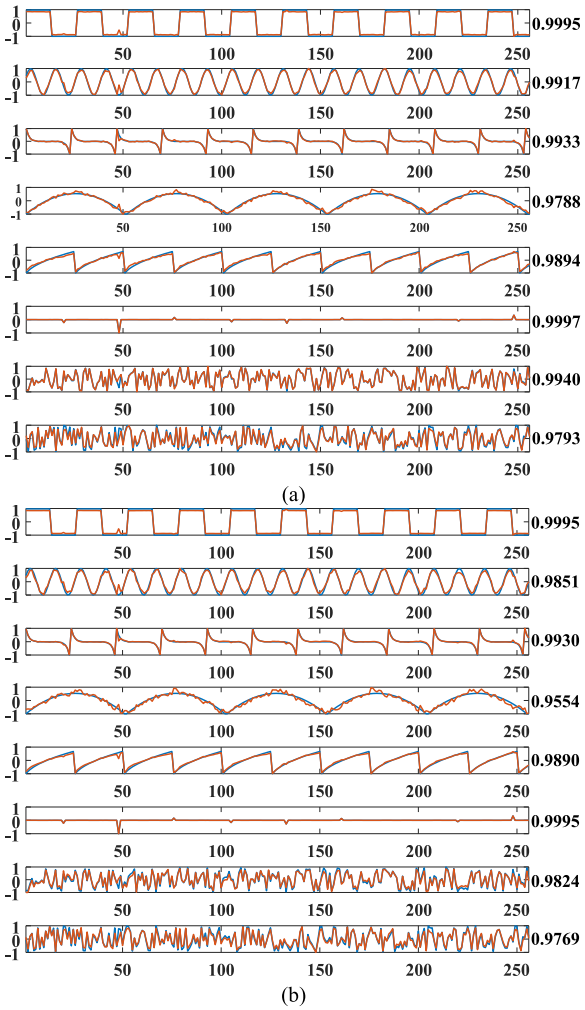

Fig. 12. a) Original 8 channels of synthetic signals (blue) versus the MATLAB floating-point FastICA results (red); b) the original 8 channels of synthetic signals (blue) versus the fixed-point FiCA's results (red).

two top signals in Fig. 13, very close to the estimations by the MATLAB FastICA toolbox with correlation values of 0.9602 and 0.9570. All other six decomposed sources also have high correlations with the MATLAB toolbox results.

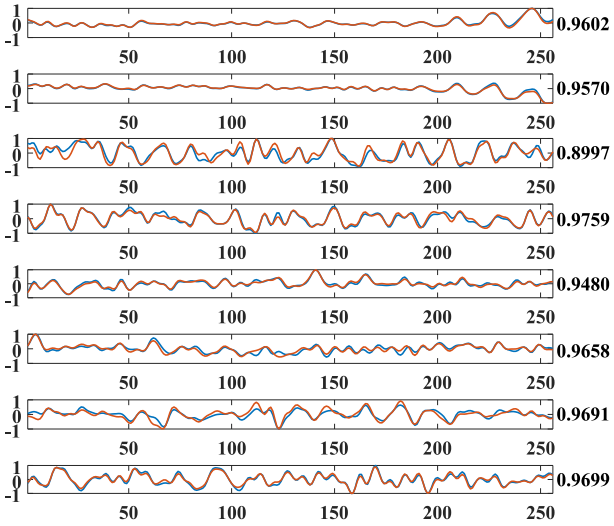

Fig. 13. Decomposition results of the fixed-point FiCA (red) versus the results of MATLAB floating-point FastICA toolbox (blue) for 8 channels of EEG signals containing artefacts.

TABLE I
HIGH-LEVEL SPECIFICATIONS AND SYNTHESIS RESULTS OF THE FICA PROCESSOR IN COMPARISON WITH STATE-OF-THE-ART

	FiCA	[16]	[15]	[14]	[13]
Algorithm	FastICA	FastICA	FastICA	FastICA	FastICA
Application	EEG	EEG	EEG	Ecog	EEG
Number of Channels	8	6	2-16	8	8
Implementation	ASIC 90nm	ASIC 90nm	ASIC 90nm	ASIC 90nm	ASIC 90nm
Latency-awareness	Yes	No	No	No	No
Arithmetic Computations	Customized Fixed-point	Fixed-point	Floating-point	Fixed-point	Floating-point
Scalability	Yes	No	No	Yes	No
Circuit Area (kGE)	1808	N/A	401	69.2	272
Clock Freq. (MHz)	555	240	100	11	100
Power Dissipation (mW)	760	-	19.4	0.0816	16.35
Normalized Power (mW/kHz/kGE)	0.757	-	0.48	0.107	0.601
Max Comp. Time (ms)	0.32	N/A	1840	84.2	290
Throughput (*k*ICA/s)	3.1249	-	0.0005	0.0119	0.0034
Normalized throughput (ICA/s/kGE)	1.7284	-	0.0014	0.0172	0.0127

C. Processor Synthesis Results and Comparison

Table I includes the main high-level specifications as well as the detailed implementation results of the FiCA processor in comparison with its four best rivals, addressed in the literature review. The first four table rows indicate that all the designs: 1) implemented the FastICA algorithm, 2) targeted EEG signals except for [14], 3) mostly developed for 8 channels of signals, and 4) implemented using ASIC 90 nm technology. These simplify the fair comparison process.

The next three table rows demonstrate main architectural features of FiCA. According to the 5th table row, FiCA is the only architecture that can be properly configured to support the significant algorithmic disadvantage of the ICA algorithm and to handle any algorithmic failure based on the latency requirements of the target applications. Also, according to the 6th row of the table, FiCA as well as [14] and [16], used fixed-point arithmetic. However, the latter two instances provided no WL-optimization results or any extra explanations at all, while FiCA is a fully customized, block-level fixed-point implementation. Moreover, based on the 7th row, FiCA and, to some extent [14], are the only designs that can be scaled to match the required cost-performance trade-off.

The last seven table rows provide a thorough comparative study on cost and performance of the synthesized designs. Although, unlike other designs, FiCA can be configured to support different numbers of weight calculation units, for fair comparison only the results of the single weight calculation unit configuration have been listed in this table. The 8th table row presents the circuit area in terms of kilo gates equivalent (kGE), while the next rows show the maximum achieved clock frequency. This area is calculated by dividing the synthesizer's area output by the smallest 2-input NAND gate in the library. Based on these results, although the FiCA area is more than other rivals as expected, according to the tenth row, it has a

higher clock frequency and provides an overall much better performance. The 10th table row shows the power dissipation for this design and other rivals in terms of milliwatt (mW). For fair comparison, the 11th table row presents the same results normalized by frequency and circuit area (in terms of mw/kHz/kGE). According to this row, while the main target in FiCA development has been performance optimization and not power, as can be inferred from row 11, except for [14], other rivals have reported power dissipations of not less than half of that of this work. Since [14] has put their main focus on low-power design using different techniques such as approximation, their normalized power dissipation is meaningfully less than all other designs. The energy efficiency of the FiCA is also evaluated as 243 $\mu J/operation$ based on [23]. The 12th table row shows the maximum ICA computation time of an 8-channel configuration in FiCA and all other designs in terms of milliseconds (ms). However, due to FiCA's much higher frequency and performance, it can be reconfigured to support much greater number of EEG channels, sampling rates, and so on, to accommodate with the necessities of different applications with different performance requirements. The 13th table row presents the absolute performance (in terms of kilo ICA calculations per second) while, for fair comparison, the last row contains the normalized throughput of each design (in terms of number of ICA calculations per second per kGE). According to the last table row results, the FiCA proposes a normalized throughput of about 100 times better with respect to [14] as the closest rival. This becomes of greater importance noticing that the area of [14] is highly decreased by exploiting an approximation of the EVD process (instead of its original version exploited in FiCA) as well as choosing a WL of 14 (in comparison with much higher WLs in FiCA) due to much better signal-to-noise ratio of ECoG signals (in comparison with EEG signals in FiCA).

V. Conclusion

In this paper, a fully customized fixed-point, scalable, and high-performance FastICA processor architecture is presented. The proposed design has been developed in an algorithm-aware manner to suppress the inherent FastICA algorithmic failures. This was done by augmenting the system with multiple weight calculation units. Moreover, using a thorough WL-optimization approach, the architecture is developed with six different WLs inside and between its different blocks which resulted in a fully customized design. The synthesized design showed a normalized throughput of 10 times more than its closest rival.

References

[1] A. Lombard, Y. Zheng, H. Buchner, and W. Kellermann, "TDOA estimation for multiple sound sources in noisy and reverberant environments using broadband independent component analysis," *IEEE Trans. Audio, Speech, Language Process.*, vol. 19, no. 6, pp. 1490–1503, Aug. 2011, doi: 10.1109/TASL.2010.2092765.

[2] N. Shanmugapriya and E. Chandra, "Evaluation of sound classification using modified classifier and speech enhancement using ICA algorithm for hearing aid application," *ICTACT J. Commun. Technol.*, vol. 6948, pp. 1279–1288, Mar. 2016, doi: 10.21917/ijct.2016.0187.

[3] C.-Y. Yu, Y. Li, B. Fei, and W.-L. Li, "Blind source separation based X-ray image denoising from an image sequence," *Rev. Sci. Instrum.*, vol. 86, no. 9, Sep. 2015, Art. no. 093701, doi: 10.1063/1.4928815.

[4] D. A. Ramli, Y. H. Shiong, and N. Hassan, "Blind source separation (BSS) of mixed maternal and fetal electrocardiogram (ECG) signal: A comparative study," *Proc. Comput. Sci.*, vol. 176, pp. 582–591, Jan. 2020, doi: 10.1016/j.procs.2020.08.060.

[5] G. Salimi-Khorshidi, G. Douaud, C. F. Beckmann, M. F. Glasser, L. Griffanti, and S. M. Smith, "Automatic denoising of functional MRI data: Combining independent component analysis and hierarchical fusion of classifiers," *NeuroImage*, vol. 90, pp. 449–468, Apr. 2014, doi: 10.1016/j.neuroimage.2013.11.046.

[6] R. J. Korhonen, J. C. Hernandez-Pavon, J. Metsomaa, H. Mäki, R. J. Ilmoniemi, and J. Sarvas, "Removal of large muscle artifacts from transcranial magnetic stimulation-evoked EEG by independent component analysis," *Med. Biol. Eng. Comput.*, vol. 49, no. 4, pp. 397–407, Apr. 2011, doi: 10.1007/s11517-011-0748-9.

[7] G. R. Naik, Ed., *Independent Component Analysis for Audio and Biosignal Applications*. Rijeka, Croatia: InTech, 2012.

[8] M. B. Hamaneh, N. Chitravas, K. Kaiboriboon, S. D. Lhatoo, and K. A. Loparo, "Automated removal of EKG artifact from EEG data using independent component analysis and continuous wavelet transformation," *IEEE Trans. Biomed. Eng.*, vol. 61, no. 6, pp. 1634–1641, Jun. 2014, doi: 10.1109/TBME.2013.2295173.

[9] S. Çınar and N. Acır, "A novel system for automatic removal of ocular artefacts in EEG by using outlier detection methods and independent component analysis," *Expert Syst. Appl.*, vol. 68, pp. 36–44, Feb. 2017, doi: 10.1016/j.eswa.2016.10.009.

[10] M. F. Issa and Z. Juhasz, "Improved EOG artifact removal using wavelet enhanced independent component analysis," *Brain Sci.*, vol. 9, no. 12, p. 355, Dec. 2019, doi: 10.3390/brainsci9120355.

[11] A. J. Bell and T. J. Sejnowski, "An information-maximization approach to blind separation and blind deconvolution," *Neural Comput.*, vol. 7, no. 6, pp. 1129–1175, 1995.

[12] A. Hyvärinen, "Fast and robust fixed-point algorithms for independent component analysis," *IEEE Trans. Neural Netw.*, vol. 10, no. 3, pp. 626–634, May 1999. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=761722%5Cnpapers3://publication/doi/10.1109/72.761722

[13] L.-D. Van, D.-Y. Wu, and C.-S. Chen, "Energy-efficient FastICA implementation for biomedical signal separation," *IEEE Trans. Neural Netw.*, vol. 22, no. 11, pp. 1809–1822, Nov. 2011, doi: 10.1109/TNN.2011.2166979.

[14] C. Yang, Y. Shih, and H. Chiueh, "An 81.6 μW FastICA processor for epileptic seizure detection," *IEEE Trans. Biomed. Circuits Syst.*, vol. 9, no. 1, pp. 60–71, Feb. 2015, doi: 10.1109/TBCAS.2014.2318592.

[15] L.-D. Van, P.-Y. Huang, and T.-C. Lu, "Cost-effective and variable-channel FastICA hardware architecture and implementation for EEG signal processing," *J. Signal Process. Syst.*, vol. 82, no. 1, pp. 91–113, Jan. 2016, doi: 10.1007/s11265-015-0988-2.

[16] S. Bhardwaj, S. Raghuraman, and A. Acharyya, "Simplex FastICA: An accelerated and low complex architecture design methodology for nD FastICA," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 5, pp. 1124–1137, May 2019, doi: 10.1109/TVLSI.2018.2886357.

[17] L. D. Pyeatt and W. Ughetta, *ARM 64-Bit Assembly Language*. Amsterdam, The Netherlands: Elsevier, 2019.

[18] S. M. R. Shahshahani and H. R. Mahdiani, "A high-performance scalable shared-memory SVD processor architecture based on Jacobi algorithm and batcher's sorting network," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 67, no. 6, pp. 1912–1924, Jun. 2020, doi: 10.1109/TCSI.2020.2973249.

[19] G. E. Forsythe and P. Henrici, "The cyclic Jacobi method for computing the principal values of a complex matrix," *Trans. Amer. Math. Soc.*, vol. 94, no. 1, pp. 1–23, Jan. 1960, doi: 10.1090/S0002-9947-1960-0109825-2.

[20] M. J. Schulte and K. E. Wires, "High-speed inverse square roots," in *Proc. Symp. Comput. Arithmetic*, 1999, pp. 124–131, doi: 10.1109/arith.1999.762837.

[21] M. George and P. Alfke. (2007). *Linear Feedback Shift Registers in Virtex Devices*. [Online]. Available: https://www.xilinx.com

[22] M. A. Klados and P. D. Bamidis, "A semi-simulated EEG/EOG dataset for the comparison of EOG artifact rejection techniques," *Data Brief*, vol. 8, pp. 1004–1006, Sep. 2016, doi: 10.1016/j.dib.2016.06.032.

[23] P. U. da Costa, G. Paim, L. M. G. Rocha, E. A. C. da Costa, S. J. M. de Almeida, and S. Bampi, "Fixed-point NLMS and IPNLMS VLSI architectures for accurate FECG and FHR processing," *IEEE Trans. Biomed. Circuits Syst.*, vol. 15, no. 5, pp. 898–911, Oct. 2021, doi: 10.1109/TBCAS.2021.3120237.