Contents lists available at ScienceDirect

Neurocomputing

journal homepage: www.elsevier.com/locate/neucom

# IEEE 754 floating-point addition for neuromorphic architecture

Arun M. George[a], Rahul Sharma[b,*], Shrisha Rao[c]

[a] TCS Research and Innovation, Bangalore, India
[b] Continental Automotive Components India Pvt Ltd, Bangalore, India
[c] International Institute of Information Technology-Bangalore, Bangalore, India

ARTICLE INFO

*Article history:*
Received 6 July 2018
Revised 27 January 2019
Accepted 16 May 2019
Available online 22 July 2019

Communicated by Dr Chenchen Liu

*Keywords:*
Neuromorphic computing
Neuromorphic architecture
Floating-point arithmetic
IEEE 754
Nengo
Neural engineering framework

ABSTRACT

Neuromorphic computing is looked at as one of the promising alternatives to the traditional von Neumann architecture. In this paper, we consider the problem of doing arithmetic on neuromorphic systems and propose an architecture for doing IEEE 754 compliant addition on a neuromorphic system. A novel encoding scheme is also proposed for reducing the inter-neural ensemble error. The complex task of floating point addition is divided into sub-tasks such as exponent alignment, mantissa addition and overflow-underflow handling. We use a cascaded approach to add the two mantissas of the given floating-point numbers and then apply our encoding scheme to reduce the error produced in this approach. Overflow and underflow are handled by approximating on XOR logic. Implementation of sub-components like right shifter and multiplexer are also specified.

## 1. Introduction

Neuromorphic computing has emerged in recent years as a plausible future alternative to the pervasive and long-standing von Neumann architecture. The term “neuromorphic computing” was coined by Mead [1], who referred to Very Large Scale Integration (VLSI) systems with analog components that mimicked biological neural systems. These neuromorphic architectures are known for their high parallelism and low power needs. Neuromorphic architectures have also received increased attention because of the approaching end of Moore’s law [2] and the bandwidth restrictions between CPU and memory known as “the von Neumann bottleneck” [3]. With energy efficiency being crucial for future high performance and embedded computing platforms, neuromorphic computing offers a solution to today’s rising demands in computing [4].

Neuromorphic computing is not yet a practical technology in a mainstream sense (see the tutorial by Volanis et al. [5] and the survey by James et al. [6]), but there are efforts to make it work using classical VLSI technologies [7,8]. One issue that is currently being addressed is energy efficiency [4,9,10]. Newer technologies, particularly memristors [11,12] are also being considered as possible routes for implementing neuromorphic architectures. Neuromorphic computing is also studied by neuroscientists who wish to model or understand the brain [13–15], and by those who wish to mimic the functionality of sense organs [16]. Machine learning also provides another important reason for interest in neuromorphic computing [8,17,18].

Recent years have seen several breakthroughs in neuromorphic hardware which address some important concerns with the uses of the same (such as the lack of accuracy), and also expand the range of applications possible with neuromorphic computing. Interesting new hardware avenues include the neuromorphic processors Loihi [19] and Braindrop [20]; the use of non-volatile memory [21], on-chip communication [22] and reconfigurability through hierarchical addressing [23]; and the DYNAPs architecture [24]. New proposed uses of neuromorphic computing include network applications [25] and matrix-vector multiplication [26].

Floating-point arithmetic is crucial for most scientific computing. Any new computing architecture, therefore, would need a system that can perform floating-point arithmetic. In this paper, we propose a system which can perform the addition of two IEEE 754-compliant floating-point numbers on a neuromorphic architecture. We have designed a system which follows the conventional addition process [27] but works on groups of neurons instead of logic gates. There have been previous attempts to develop systems of computation on neuromorphic architectures [6,28] but not much has been done in the area of numerical computations, especially in the area of floating-point arithmetic. The novelty and significance of our work is to explore this area to understand how

* Corresponding author.
*E-mail addresses:* arun.george@iiitb.org (A.M. George), rahul.sharma112@iiitb.org (R. Sharma), shrao@ieee.org (S. Rao).

https://doi.org/10.1016/j.neucom.2019.05.093
0925-2312/© 2019 Elsevier B.V. All rights reserved.

floating-point arithmetic on a neuromorphic architecture can be applied to problems of computation. Our work also gives an estimate of the total number of neurons that would be required to implement such an architecture in reality. The estimate also indicates the number of neurons required to implement each component (see Table 2).

Our system is modular by design and therefore its components (like the two's complement converter or the mantissa adder) can be used in other applications too. This system is also scalable in the sense that it can work with different formats of floating-point numbers, for instance, single-precision and double-precision formats. It can also be easily extended to form solutions to other floating-point operations like subtraction, with relatively small changes required, given the modular nature of the architecture.

The system design is based on the Neural Engineering Framework (NEF) [29] which provides the basic principles to develop a neuromorphic system. For the implementation, simulation and testing of our design we used Nengo [30,31], a graphical and scripting based software package for simulating large-scale neural systems. To use Nengo, we defined groups of neurons in terms of what they represent, and then formed connections between neural groups in terms of what computation [32] should be performed on those representations.

One of the results of our work was to come up with an appropriate value of *radius* (see Sections 3.1.1 and 4.1), so that the error in representation of the values in a group of neurons could be minimized. We also came up with an encoding scheme (see Section 3.1.2) which reduced the inter-ensemble error, i.e., the error generated when an output value of one ensemble is given as an input value to another. We also devised a way to use a conventional multiplexer circuit to map one of the two input values to the output (see Section 3.4). We further modified the typical multiplexer to develop a modified one which could map two different inputs to two different outputs (see Section 3.5). The stage-wise mantissa adder (see Section 3.7) of our design takes as input two 23-bit mantissas and produces their 23-bit sum along with a carry. The stage-wise computation approach utilizes a lower number of resources as compared to a sequential one. This is turn reduced the total number of neurons substantially.

Our system also indicates if there is an overflow or an underflow during the addition process (see Section 3.8) though a flag, which can then be used to handle it appropriately. The most important and also complex part of the floating-point addition process is the matching of the exponents of the two different inputs. This in turn requires shifting of the mantissa bits. We came up with an approach to accomplish this which is supported by Theorems 3.1 and 3.2 (see Section 3.6). These two theorems are a salient contribution of our work as they address the crux of the problem, i.e., the procedure of shifting mantissa bits in a neuromorphic architecture when the exponents of the two given inputs are different.

We also did a performance analysis of our implementation over three different groups of inputs (see Section 4.2). The first group contained random pairs of inputs with large differences, whereas the second group contained random pair of inputs with small differences. The third group contained random pair of inputs with random differences. We used two performance analysis metrics: Mean Absolute Error (MAE) and Mean Encoded Error (MEE) to estimate the performance of our system. We also observed the effect of varying the number of neurons on the accuracy and bit error (see Sections 4.3 and 4.4. We found that the overall bit error for adding two floating-point numbers was around 0.002%. We earlier mentioned that we came up with an appropriate value of radius to lower the error. We also observed the effects of different values of radius on the accuracy (see Section 4.1).

Our work addresses the issue of performing floating-point calculations on a neuromorphic architecture. We have proposed a system for adding two IEEE 754-compliant floating-point numbers using the conventional adder circuits made up of spiking neurons. This system can also be extended for subtraction, multiplication and division, thereby making possible a completely neuromorphic arithmetic unit.

The rest of the paper is structured as follows. We first give a brief description of the IEEE 754 floating-point addition process in Section 2.1. Then we move on to briefly describe the Neural Engineering Framework (NEF) and its three basic principles: representation, transformation and dynamics, in Section 2.2. After this we explain the overall architecture in Section 3 with the help of Fig. 3. In Section 3.1 we talk about the value of radius in Section 3.1.1 and the encoding scheme in Section 3.1.2. These two concepts are have important roles in our implementation. After this we explain the most important component of our system, viz., the exponent aligner, in Section 3.2. This component is responsible for matching the two exponents of the inputs by shifting the bits of one of their mantissas. Next in Section 3.3 we explain how two's complement is computed. In Sections 3.4 and 3.5 we explain the workings of the two multiplexers that we mentioned earlier. In Section 3.6 we explain the workings of the mantissa shifter which solves the most complex part of our problem, viz., shifting the bits of a mantissa such that the exponents of both the inputs are matched. We also state and prove Theorems 3.1 and 3.2 in this section. Next we explain how the stage-wise addition is performed by the mantissa adder in Section 3.7. Then we describe the last of the remaining components which computes the sign bit of the output and overflow/underflow flag in Section 3.8. In Section 3.9 we explain how our system can be extended to perform floating-point subtraction. We present the observations and results of our work in Section 4. Here we explain the effect of different values of radius on accuracy in Section 4.1; the performance analysis metrics in Section 4.2 deals with the two metrics that we have used to evaluate our system: the Mean Absolute Error (MAE) and Mean Encoded Error (MEE). In Section 4.3 we describe the relationship between the number of neurons and accuracy, and in Section 4.4 we describe the relationship between the number of neurons and bit error. In Section 4.5 we describe how we estimated the optimal number of neurons required for all the ensembles and list them in Table 2. Finally, we present the conclusions in Section 5.

## 2. Background

First we briefly discuss the floating-point addition process as per the IEEE 754 standard [27], then we describe the Neural Engineering Framework (NEF) which we have used to design, simulate and evaluate our system. [29].

### 2.1. IEEE 754 floating-point addition

Fig. 1 illustrates the overall process of addition and subtraction of two floating-point numbers – $Input_1$ and $Input_2$ represented in binary format. Fig. 2 is an example of how a 32-bit floating-point number is represented according to the IEEE 754 standard [27]. There is a sign bit which is used to represent whether the number is positive or negative. There are 8 bits and 23 bits which are used to represent the exponent and mantissa values, respectively. While designing this system we assumed that both inputs, i.e., the two floating-point numbers are represented according to the IEEE 754 standard in binary representation.

Now in Fig. 1 the first step of the process involves the matching of the exponents and shifting of the mantissas. Once this is done, we check the sign bit to decide whether we need to add the two mantissas or subtract them. If it is an addition then it is

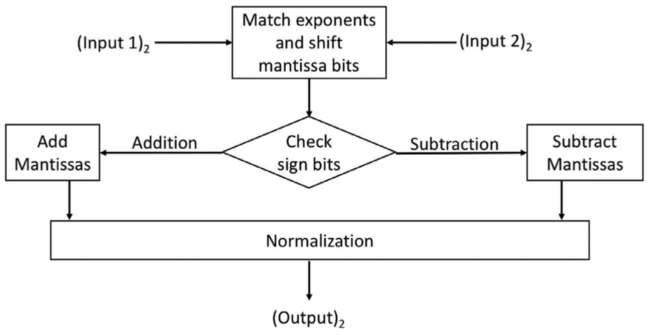

**Fig. 1.** Overall process for addition/subtraction of floating-point numbers.

| | | |
|---|---|---|
| 0 | 1000 0001 | 0101 0000 0000 0000 0000 000 |
| Sign (1) | Exponent (8) | Mantissa (23) |

**Fig. 2.** IEEE 754 32-bit floating-point representation.

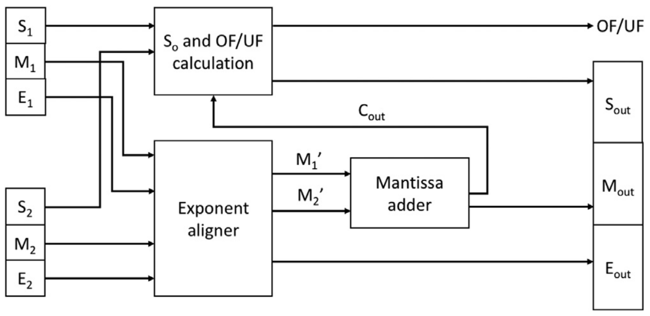

**Fig. 3.** System architecture with two 23-bit inputs and a 23-bit output.

a straightforward process of adding two binary numbers, else we subtract the two numbers which involves representing one of the two inputs in two's complement representation [33].

Here is an example of addition of two floating-point numbers. We take two inputs as shown below and represent them in the normalized scientific notation:

$$Input_1 : 123456.7 = 1.234567 \times 10^5$$

$$Input_2 : 101.7654 = 1.017654 \times 10^2$$

Now, we modify $Input_2$ such that it has the same exponent as $Input_1$:

$$Input_2 : 1.017654 \times 10^2 = 0.001017654 \times 10^5$$

Now both $Input_1$ and $Input_2$ have the same exponent, therefore we can go ahead and add their respective mantissas. Hence:

$$\begin{aligned} 123456.7 + 101.7654 &= (1.234567 + 0.001017654) \times 10^5 \\ &= 1.235584654 \times 10^5 \end{aligned}$$

Note that the same process can be adapted for these two numbers when they are represented in the binary format as shown in Fig. 2.

We now state a few assumptions regarding the format of the inputs which we considered while designing the system:

1. Both inputs should be in binary representation according to the IEEE 754 format [27] with sign, exponent and mantissa bits as shown in Fig. 2.
2. Both inputs should be normalized [34], for example, $Input_A$ below is normalized whereas $Input_B$ is not.

   $$Input_A : 1.234567 \times 10^5$$

   $$Input_B : 101.7654 \times 10^2$$
3. Since our system is designed to only handle positive inputs, therefore a negative input should be converted into a positive one using two's complement conversion [33] and then fed into the system.

### 2.2. The Neural Engineering Framework (NEF)

The Neural Engineering Framework [29,35,36] is a generalized computational framework used for modelling large and complex neural systems. In NEF, we can create ensembles of spiking neuron models, represent values on these neural ensembles and solve for synaptic connection weights for computing functions using them. There are three underlying principles for NEF, on which neural systems of fairly high complexity can be built.

#### 2.2.1. Representation

NEF uses distributed representations for values. It distinguishes the value that is being represented from the neural activity. A scalar or a vector can be represented on a neural ensemble with different values corresponding to different neural activity patterns. This representation involves non-linear encoding and weighted linear decoding. If **x** is the value getting represented on a neuron ensemble and $e_i$ is the encoding vector for the neuron, then activity $a_i$ for each neuron can be obtained as

$$a_i = G(\alpha_i e_i \cdot \mathbf{x} + b_i) \tag{1}$$

where G is the neural non-linearity, $\alpha_i$ is a gain parameter and $b_i$ is the constant background bias current for the neuron. Given an activity, the set of weights to estimate the value can be done by finding a linear decoder $d_i$ as:

$$\hat{\mathbf{x}} = \sum a_i d_i \tag{2}$$

$d_i$ can be calculated as the set of weights that minimize the difference between **x** and $\hat{\mathbf{x}}$.

#### 2.2.2. Transformation

Computation using spiking neural ensembles can be done by transformations which approximate a function of value that is being represented in an ensemble. Transformation is done by another weighted linear decoding. For approximating a function **f(x)**, the decoded weights $d^{\mathbf{f(x)}}$ can be computed as

$$d^{\mathbf{f(x)}} = \Gamma^{-1} \Upsilon^{\mathbf{f(x)}} \tag{3}$$

$$\Gamma_{ij} = \sum_{\mathbf{x}} a_i a_j \tag{4}$$

$$\Upsilon_j^{\mathbf{f(x)}} = \sum_{\mathbf{x}} a_j \mathbf{f(x)} \tag{5}$$

This is similar to a Support Vector Machine (SVM) [37] as we do a random projection on $e_i$ and the function **f(x)** is expressed as the sum of tuning curves of neurons. This method of approximating a function, although not as powerful as approximation using a learning rule, works well for linear functions. Some non-linear and discontinuous functions can also be computed with a trade-off between number of neurons in the ensemble and accuracy.

#### 2.2.3. Dynamics

The dynamics of the neural systems can also be modelled in NEF using control theoretic state variables. NEF can compute dynamic functions of the form $d\mathbf{x}/dt = A(\mathbf{x}) + B(u)$ where **x** is the value getting represented, $u$ is some input, $A$ and $B$ are some arbitrary functions.

## 3. System architecture and implementation

We designed a system that performs floating-point addition and subtraction according to the process illustrated in Fig. 1. Fig. 3 illustrates the system architecture. The two inputs are represented as ($S_1$, $M_1$, $E_1$) and ($S_2$, $M_2$, $E_2$) and the output is represented as ($S_{out}$, $M_{out}$, $E_{out}$). Here $S_i$ represents the sign bit, $M_i$ represents the mantissa bits and $E_i$ represents the exponent bits where $i \in \{1, 2, out\}$ This representation follows the IEEE-754 32-bit floating-point standard [27] as illustrated in Fig. 2. The core of the system is the exponent aligner component which ensures that the exponents of the input numbers match and any shifting of mantissa bits (if required) is performed before the mantissas are added by the mantissa adder component. The $S_{out}$ and OF/UF calculation component computes the sign bit of the output and the overflow/underflow flag which can then be used for rounding the resultant output [27].

We now go on to explain the simulation method followed by the description and workings of all the components of the system as shown in Fig. 3 in detail.

### 3.1. Simulation

To represent information, say one bit from the input, we created neural ensembles (a group of a number of neurons) using Nengo [30,31]. We used the Leaky Integrate-and-Fire (LIF) model [38] in Nengo to create the neural network. Each of these ensembles has two important properties — dimension and radius. Dimension refers to the number of values represented by the ensemble. In case of a scalar quantity, it is 1 whereas for a vector it can be more than 1. Radius, on the other hand, defines the range of values that can be represented by the ensemble. For instance, if we simulate a neural ensemble (of say, 10 neurons) with dimension as 1 and radius as 1, then these neurons can accurately represent the values between −1 and 1. Another way to understand this is to think about it geometrically. If the dimension is 1 and radius (or length) is 1 then from a point we can either go +1 or −1. Similarly, if dimension is 2 and radius is 1 then we have a circle with radius 1.

#### 3.1.1. The value of radius

In our system, since we work with binary numbers, the addition or subtraction of two bit values always lies between 0 and 2, both inclusive. Therefore, we set the radius of all neural ensembles to 2.

#### 3.1.2. Encoding scheme

Next we observed that when we give the output of one neural ensemble as an input to another, the error propagates. For example, if a neural ensemble represents the value 1 and when we probe it, we get the value 1.1. Then this value is given as an input to the next ensemble which further adds its own representational error. To avoid this error, we use the following encoding scheme. If the output of a neural ensemble is $\hat{\mathbf{x}}$, the $i^{th}$ component of the encoded output value of an ensemble is given as:

$$E(\hat{\mathbf{x}}_i) = \begin{cases} 0, & \hat{\mathbf{x}}_i \leq 0.5 \\ 1, & otherwise \end{cases} \tag{6}$$

By using this encoding scheme on outputs of a neural ensemble, we reduce the error to one side. All the values less than or equal to 0.5 are considered 0, as such the error towards the negative direction is reduced to zero. Similarly all values greater than 0.5 are considered as 1.0, reducing the positive direction error to zero. On top of this, the encoding scheme provides a margin of 0.5 in the other direction as well. Our encoding scheme acts similar to a rounding scheme on neuron signals and can be easily implemented in hardware. We now explain the workings of each component of the system.

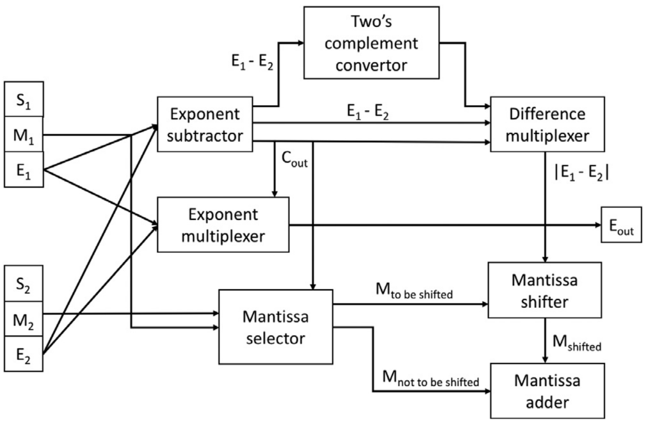

**Fig. 4.** Exponent aligner architecture.

### 3.2. Exponent aligner

The exponent aligner is the core of our system. It takes the two exponents from the inputs and finds their difference and then shifts the mantissa of the smaller exponent by the magnitude of the difference. Fig. 4 illustrates the workings of the exponent aligner. The exponent subtractor takes in the two exponents and computes their difference. It produces a carry which signifies whether the difference is positive or negative. If the difference is in two's complement form [33] then it is converted back to sign magnitude form. The difference multiplexer then uses the carry produced by the exponent subtractor to generate the magnitude of the difference between the two exponents.

The exponent multiplexer uses the carry produced by the exponent subtractor to choose the larger of the exponents to be the output exponent $E_{out}$. The mantissa selector also uses the carry produced by the exponent subtractor to differentiate between the input mantissas to figure out the one which has to be shifted. Once the mantissa shifter shifts one of the mantissas by the magnitude of the difference between the two exponents then, the two mantissas are added by the mantissa adder.

We now explain the workings of each of these components in detail.

### 3.3. Exponent subtractor and two's complement converter

Fig. 5 illustrates the working of the two's complement converter. It takes in each of the 8-bits of the exponent and represents each of them using a neural ensemble. Next, at each of these ensemble outputs, we perform a transformation such that each bit is flipped, i.e., it changes 0's to 1's and 1's to 0's. Finally, we take in all the flipped bits into an 8-bit adder and add one to it, with input carry as 0 to perform the standard two's complement transformation.

The exponent subtractor in Fig. 4 is essentially performing two's complement binary subtraction. It works as follows:

- Take in both exponents $E_1$ and $E_2$ as input and convert $E_2$ into its two's complement form $E_2'$ using the two's complement converter as described above.
- Add $E_1$ and $E_2'$ using the 8-bit adder and let the carry produced be $C_{out}$.
- If $C_{out}$ is 1, then it means that the result is positive and it is given directly as an input to the difference multiplexer in Fig. 4.

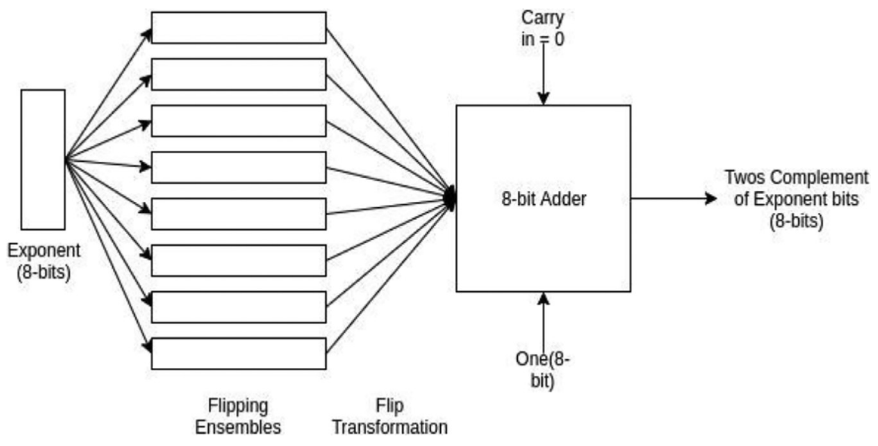

**Fig. 5.** Two's complement converter architecture.

- If $C_{out}$ is 0, then it means that the result is negative and therefore we perform two's complement of the result and then give it as an input to the difference multiplexer.

### 3.4. Exponent/difference multiplexer

This component's working is based on that of a multiplexer. It selectively outputs one of the input values based on the value of a selection input. In the exponent multiplexer and the difference multiplexer, the carry generated by the exponent subtractor, i.e., $C_{out}$ acts as the selection bit.

The difference multiplexer is given three inputs — $C_{out}$ as the selection bit, the difference of the two exponents $E_1$ and $E_2$ when $C_{out}$ is 1, i.e., the result is positive and when $C_{out}$ is 0, i.e., the result is negative. The difference multiplexer now works as follows –

- If the selection bit, i.e., $C_{out}$ is 1, then it means that the result is positive and it is generated as the output of the difference multiplexer as the magnitude of the difference of the two exponents $E_1$ and $E_2$.
- If the selection bit, i.e., $C_{out}$ is 0, then it means that the result is negative, therefore, its two's complement converted form (as explain in the workings of the exponent subtractor) is generated as the output of the difference multiplexer.

The output of the difference multiplexer (the number of shifts to be performed on the mantissa) is then given as an input to the mantissa shifter.

We now look at the workings of the exponent multiplexer. It uses the selection bit $C_{out}$ to figure out the larger exponent from $E_1$ and $E_2$ and outputs it as $E_{out}$. It works as follows:

- If the selection bit, i.e., $C_{out}$ is 1, then it means that $E_1$–$E_2$ is positive, therefore $E_1$ is larger and it is output as $E_{out}$.
- If the selection bit, i.e., $C_{out}$ is 0, then it means that $E_1$–$E_2$ is negative, therefore $E_2$ is larger and it is output as $E_{out}$.

$E_{out}$ is needed in the final representation of the output, i.e., the sum of two floating-point numbers.

### 3.5. Mantissa selector

The mantissa selector takes in the two mantissas $M_1$ and $M_2$ as inputs and decides which one of them has to be shifted based on the carry generated by the exponent subtractor. It is similar to the exponent/difference multiplexer but here we have two outputs — the mantissa to be shifted and the mantissa not to be shifted. These are chosen based on the selection bit $C_{out}$.

The mantissa selector works as follows.

- If the selection bit, i.e., $C_{out}$ is 1, then it means that $E_1$–$E_2$ is positive, therefore $E_1$ is larger and therefore $M_1$ is the mantissa which is not to be shifted whereas $M_2$ is the mantissa which is to be shifted.
- If the selection bit, i.e., $C_{out}$ is 0, then it means that $E_1$–$E_2$ is negative, therefore $E_2$ is larger and therefore $M_2$ is the mantissa which is not to be shifted whereas $M_1$ is the mantissa which is to be shifted.

The mantissa to be shifted is given as an input to the mantissa shifter which performs the shift operation and outputs the modified mantissa to be used as an input by the mantissa adder along with the mantissa which is not to be shifted.

### 3.6. Mantissa shifter

The mantissa shifter is a logical right shifter for our 23-bit mantissa on an offset value of 8-bit difference bits from our exponent subtraction circuit. Unlike synchronous clock-driven digital shift circuits, neuromorphic computing is asynchronous in nature. So normal shifting method using a shift register and a counter will not work for neuromorphic chips. We propose a different approach for achieving logical right shift. If the 8-bit difference from exponent subtraction circuit is negative, then the difference is in two's complement form and needs to be converted to the magnitude form with the help of a two's complement converter. The appropriate offset is then selected from the difference bits of the subtraction circuit and those from two's complement conversion circuit using an 8-bit difference multiplexer circuit. This ensures that the offset to right shifter is always in magnitude form and the right shift is logical in nature.

Although there exist approaches for making shift registers asynchronous, we found them to be complex, considering that the circuit needs to be implemented with spiking neurons and that we need only right shift which can be carried out using a cascaded approach. The task of logical right shift of the 23-mantissa bits can be divided as right shift of the mantissa bits by the positional value of each of the individual offset bits. To this end we state and prove the following theorems.

**Theorem 3.1.** *Logical right shift of a number M represented in binary with length m by an arbitrary binary positional value of* $p \cdot 2^q$, $p \in \{0, 1\}, q \in \{0, 1, \ldots, k\}$ *to another number* $M'$ *has*

$M'_i = \Phi_i$

*where* $\Phi$ *is a fixed shift function defined as,*

$$\Phi_i = \begin{cases} M_i & \text{if } p = 0 \\ M_{i+2^q} & \text{if } p \neq 0 \ \wedge \ (i + 2^q \leq m) \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

**Proof.** We give a proof by cases. Let $M$ be represented as $M_mM_{m-1}M_{m-2}\dots M_2M_1$ and $M'$ as $M'_mM'_{m-1}M'_{m-2}\dots M'_2M'_1$. We prove that $M'_i = \Phi_i$ by considering the values of $p$

*Case 1* : When $p = 0$

$$\begin{aligned} p = 0 &\Rightarrow p \cdot 2^q = 0 \\ &\Rightarrow \text{we want to shift } M \text{ by } 0 \\ &\Rightarrow \text{resultant } M' = M \\ &\Rightarrow M'_i = M_i \text{ or } M'_i = \Phi_i \end{aligned}$$

*Case 2* : When $p = 1$ and $(i + 2^q) \leq m$

$$\begin{aligned} p = 1 &\Rightarrow p \cdot 2^q = 2^q \\ &\Rightarrow \text{we want to right shift } M \text{ by } 2^q \text{ bit positions} \\ &\Rightarrow M_i \text{ becomes } M'_{i-2^q} \\ &\Rightarrow i^{th} \text{ bit of } M',\ M'_i \text{ is shifted from } M_{i+2^q} \\ &\Rightarrow \text{ since } (i + 2^q) <= m,\ \exists \text{ a valid bit position} \\ &\qquad (i + 2^q) \text{ which can be shifted to } M'_i \\ &\Rightarrow M'_i = M_{i+2^q} \end{aligned}$$

*Case 3* : When $p = 1$ and $(i + 2^q) > m$

$$\begin{aligned} p = 1 &\Rightarrow p \cdot 2^q = 2^q \\ &\Rightarrow \text{we want to right shift } M \text{ by } 2^q \text{ bit positions} \\ &\Rightarrow M_i \text{ becomes } M'_{i-2^q} \\ &\Rightarrow i^{th} \text{ bit of } M',\ M'_i \text{ is shifted from } M_{i+2^q} \\ &\Rightarrow \text{ since } (i + 2^q) > m,\ \nexists \text{ a valid bit position} \\ &\qquad (i + 2^q) \text{ which can be shifted to } M'_i \\ &\Rightarrow \text{zeroes are added since it is logical right shift} \\ &\Rightarrow M'_i = 0 \end{aligned}$$ □

**Theorem 3.2.** *Logical right shift of a number M represented in binary with length m by another number K in binary with length k is equivalent to a sequence of right shifts by values $S_i$ where*

$$S_i = K_i \cdot 2^{i-1}, \quad i \in \{1, 2, \dots, k\}$$

**Proof.** We need to shift0 '*M*' by '*K*'. Let $M$ be represented as $M_mM_{m-1}M_{m-2}\dots M_2M_1$ and $K$ as $K_kK_{k-1}K_{k-2}\dots K_2K_1$. We use $\longmapsto$ to denote the logical right shift operation.

$$\begin{aligned} M \longmapsto K &\equiv M_mM_{m-1}\dots M_2M_1 \longmapsto K_kK_{k-1}\dots K_2K_1 \\ &\equiv M_mM_{m-1}\dots M_2M_1 \longmapsto (K_k0_{k-1}\dots 0_20_1 + \\ &\qquad 0_kK_{k-1}\dots 0_20_1 + \dots + 0_k0_{k-1}\dots 0_2K_1) \\ &\equiv M_mM_{m-1}\dots M_2M_1 \longmapsto K_k \cdot 2^{k-1} + \\ &\qquad K_{k-1} \cdot 2^{k-2} + \dots + K_2 \cdot 2^1 + K_1 \cdot 2^0 \end{aligned}$$

Instead of shifting by the entire $K$, we can shift by sum of the positional values of the bits in $K$. Since we are doing a logical right shift by a sum of values which are positive, it is equivalent to do continuous shifting by those values. That is

$$\begin{aligned} M \longmapsto K &\equiv ((\dots((M \longmapsto K_k \cdot 2^{k-1}) \longmapsto K_{k-1} \cdot 2^{k-2})\dots) \\ &\qquad \longmapsto K_2 \cdot 2^1) \longmapsto K_1 \cdot 2^0 \\ &\equiv ((\dots((M \longmapsto S_k) \longmapsto S_{k-1})\dots) \longmapsto S_2) \longmapsto S_1 \end{aligned}$$

Hence the theorem. □

By Theorem 3.2, we split the logical right shift of our 23-bit mantissa as a sequence of cascaded right shifts with positional values of the 8-bit difference offset. By Theorem 3.1, each individual right shift by a positional value can be easily handled by the fixed shift function, Φ. To implement the circuit, we approximated this fixed shift function with NEF on Nengo. There are 24 neural ensembles for each bit, say $j$th bit in the difference offset with $ij$th ensemble connected to $M_i$, the $\Phi_i$ value and $K_j$. This way the ensembles need only represent three dimensions and the function to be approximated becomes a simple decision of which input value $M_i$ of $\Phi_i$ to output based on the $K_j$ value being 0 or 1. Output of $j$th level of ensembles becomes the mantissa input of $(j+1)$th level for shifting the mantissa bits sequentially according to Theorem 3.2. There is a single zero node which constantly outputs value zero for the case in which $\Phi_i$ being 0. Using this approach we perform the logical right shift of 24 mantissa bits (including the hidden bit) by 8 difference offset bits with a matrix of relatively small ensembles which approximates a simple decision function, thereby considerably lowering the error which occurs on direct approximation of logical right shift function with a large neural ensemble representing values of high dimensions.

Fig. 6 illustrates the overall working of the mantissa shifter. On the left hand side we have the 23 mantissa bits and the additional hidden bit. At the bottom we have the 8 offset bits by which the mantissa has to be shifted. The Zero node at the top generates a value of 0 which is used by an array of ensembles $E_{ij}$ where $i \in [1, 24]$ and $j \in [1, 8]$.

### 3.7. Mantissa adder

As shown in Fig. 3, the exponent aligner component matches the two exponents and provides the modified mantissas $M'_1$ and $M'_2$ and the output exponent $E_{out}$. The mantissa adder then takes these two modified mantissas, i.e., $M'_1$ and $M'_2$, where one of them is shifted by the mantissa shifter so that exponents of both the inputs can be matched and then adds them bit by bit as shown in Fig. 7. The corresponding bits of the two mantissas, i.e., $a_i$ and $b_i$ where $i \in [0, 22]$ are added along with the carry from the previous stage. For the first stage, the carry $c_0$ is set to 0. Consider the following example of addition of two 4-bit mantissas A and B:

$$\begin{aligned} c_{in} &= 0 \\ A &= 1010 \\ B &= 1100 \\ A + B &= 0110 \\ c_{out} &= 1 \end{aligned}$$

Similarly, the 23-stage adder adds up the corresponding bits of the modified mantissas and each stage outputs one sum bit each to make up the final output mantissa.

To implement this stage-wise addition process, we constructed a network that takes two inputs (the corresponding bits of the two mantissas, i.e., $a_i$ and $b_i$ where $0 \leq i \leq 22$) and represents them using two different ensembles of neurons, say $A$ and $B$. These two ensembles were then connected to another ensemble, say $C$, through synaptic connections. Since the incoming currents from different synaptic connections interact linearly, ensemble $C$ now represents the sum of values represented by ensembles $A$ and $B$. We then used this arrangement to iteratively build up the mantissa adder as shown in Fig. 7. The mantissa adder also outputs the $C_{out}$ bit which is used in the calculation of the output sign bit $S_{out}$ and the overflow/underflow (OF/UF) flag which we describe next.

### 3.8. $S_{out}$ and OF/UF calculation

This component computes the $S_{out}$ bit of the output along with the OF/UF (overflow/underflow) flag which can then be used for rounding [27] separately. Fig. 8 illustrates the overall working of this component. The Sign XOR block takes in the input sign bits $S_1$ and $S_2$ and performs a XOR operation on them to find if they are same or different. This XOR operation is an approximation since we are doing it on a neuromorphic architecture. The output sign bit $S_{out}$ is then computed using the output from the XOR operation, the $C_{out}$ produced by the mantissa adder and one of the sign bits.

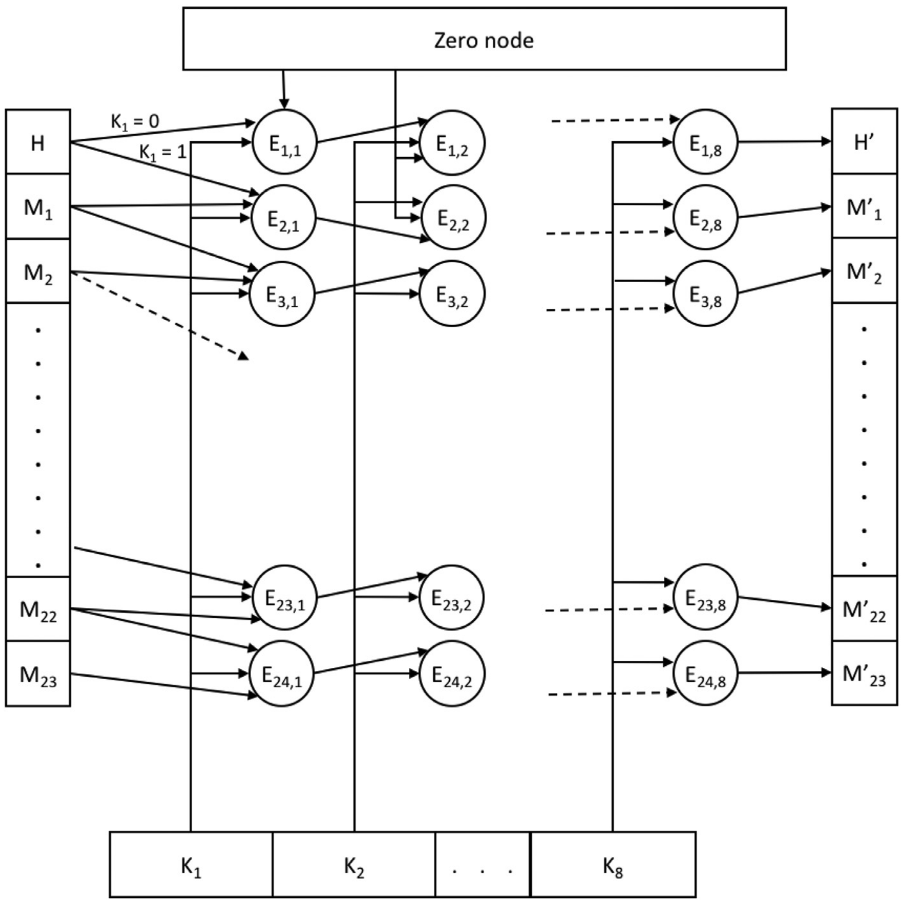

**Fig. 6.** Architecture of the mantissa shifter.

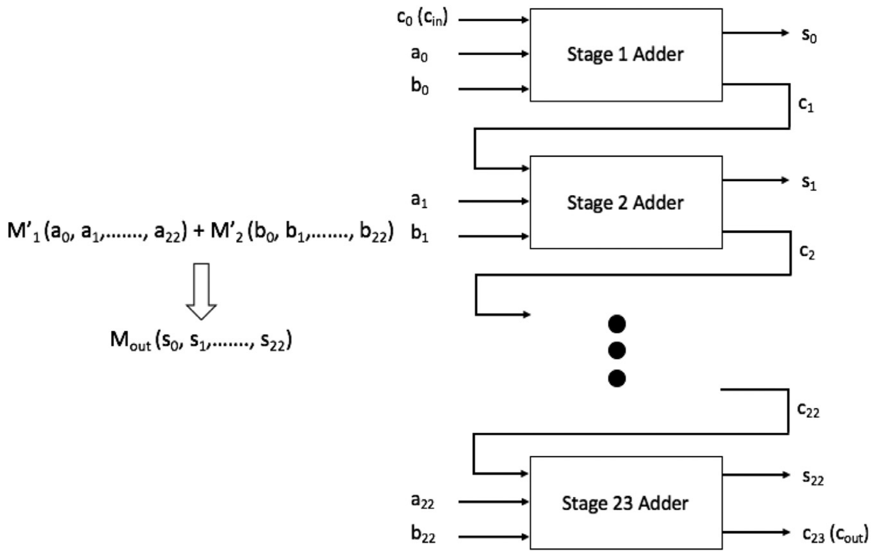

**Fig. 7.** Mantissa adder architecture for two 23-bit inputs.

Table 1 illustrates the computation of $S_{out}$ and OF/UF flags. The $S_{out}$ bit is computed as follows -

1. If $S_{XOR}$ (the output of XOR operation on $S_1$ and $S_2$) is 0, which happens when $S_1$ and $S_2$ are equal, then $S_{out}$ is equal to either of the input sign bits, i.e., $S_1$ and $S_2$ (since they are both equal).
2. If $S_{XOR}$ is 1, which is the case when $S_1$ and $S_2$ are not equal, then $S_{out}$ depends on $C_{out}$ which is the carry generated from the sum of the two mantissas. In this case, since $S_{out}$ does not depend on the input sign bits, therefore they can take any value (0 or 1). According to two's complement addition rules for a positive and negative number [33], if $C_{out}$ is 0 then $S_{out}$ is 1 whereas when $C_{out}$ is 1 then $S_{out}$ is 0.

Now we explain the computation of the OF/UF flag. An overflow or underflow can happen only when the input sign bits ($S_1$ and $S_2$) are equal, in all other cases it is 0. Thus we have the following two cases for OF/UF flag computation -

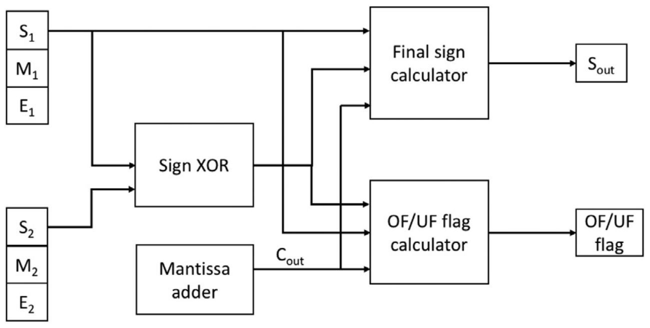

**Fig. 8.** $S_{out}$ and OF/UF component architecture.

**Table 1**
Truth table for $S_{out}$ and OF/UF flag computation.

| $S_1$ | $S_{XOR}$ | $C_{out}$ | $S_{out}$ | OF/UF |
|---|---|---|---|---|
| 0 | 0 | 0/1 | 0 | OF = 1 when $C_{out} = 1$ |
| 1 | 0 | 0/1 | 1 | UF = 1 when $C_{out} = 0$ |
| 0/1 | 1 | 0 | 1 | 0 |
| 0/1 | 1 | 1 | 0 | 0 |

**Table 2**
Number of neurons for each ensemble.

| Ensemble | Number of neurons |
|---|---|
| Linear ensemble | 100 |
| Addition ensemble | 350 |
| Flipping ensemble | 75 |
| Multiplexing ensemble | 700 |
| Selector ensemble | 700 |
| Right shift ensemble | 300 |
| XOR ensemble | 250 |
| Overflow/underflow ensemble | 300 |
| Output sign ensemble | 300 |

1. $S_1$ and $S_2$ are 0 and hence $S_{XOR}$ is 0. Since both numbers are positive, therefore there is an overflow when a carry is generated, i.e., when $C_{out}$ is 1.
2. $S_1$ and $S_2$ are 1 and hence $S_{XOR}$ is 0. Since both numbers are negative, therefore there is an underflow when no carry is generated, i.e., when $C_{out}$ is 0.

Our system does not round off the final result but it does provide the OF/UF flag which is useful when rounding off the result as per the IEEE 754 standard [27].

### 3.9. Floating-point subtraction

So far, we have explained the process of floating-point addition. The same architecture can be adopted for floating-point subtraction as it can be viewed as two's complement addition [33]. The subtraction process is as follows:

- Assume that the first input is bigger and carry out two's complement addition.
- If the carry is 1 then the result is positive and our assumption is correct else it means that we subtracted a bigger value from a smaller one and the resulting difference is in two's complement form.

## 4. Observations and results

### 4.1. Effect of different values of radius on accuracy

To observe the effect of different values of radius on the representation of values, we did the following experiment. We simulated two neural systems with the following configuration:

1. Two input nodes – $N_1$ and $N_2$ (where $N_1 = 2.1$ and $N_2 = 6.9$) connected to two neural ensembles ($E_1$ and $E_2$) each with 100 neurons each. $E_1$ and $E_2$ were then connected to another neural ensemble $S$ (again with 100 neurons) which represented the sum of the values represented by $E_1$ and $E_2$. The radius for $E_1$, $E_2$ and $S$ was set to **10**.
2. This neural system had everything same as the earlier one except the **radius for** $E_1$, $E_2$ and $S$**which was changed to 50.**

We ran the simulation for 5s and probed the values of $N_1$, $N_2$, $E_1$, $E_2$ and $S$ every 10ms and plotted the generated values for both the neural systems. Fig. 9 represents the system with a radius of 10 whereas Fig. 10 represents the one with a radius of 50.

In Fig. 9, the line and the spiking neural activity at the bottom represents $input_1$, the middle segment represents $input_2$ and the one at the top represents the sum of $input_1$ and $input_2$. The two lines with constant values in the bottom and middle are generated by values probed from the nodes — $N_1$ and $N_2$ for $input_1$ and $input_2$, respectively. Whereas the spiking neural activity which runs across these lines are generated by plotting the values probed from the ensembles – $E_1$, $E_2$ and $S$. As mentioned earlier, these values were probed every 10ms for 5s from the nodes as well as from the neural ensembles. Similarly, Fig. 10 is a plot of values generated from the simulation of the second neural system which has a radius of 50.

In Fig. 10 we can observe that because of a much higher value of radius compared to the maximum value to be represented (i.e., the sum with a value of 10), we see a significant amount of error in the value represented by the sum ensemble $S$. It can be inferred that a tight bound on the value of radius limits the neural activity in an ensemble and thus leads to a better representation of a value. This is clearly indicated in Fig. 9. Thus, we can conclude that a neural ensemble represents a value much better, i.e., with less error if it has a radius close to the maximum value that it represents.

The adder is standard to the extent that it correctly implements the IEEE 754 specification. A more general-purpose adder can be suitably constructed given an appropriate specification.

### 4.2. Performance analysis metrics

In our implementation, we simulated our proposed architecture in Nengo on a 2.5 GHz Intel Xeon E5 Machine with 30 GB of RAM. Two floating point numbers in IEEE 754 single precision binary format was given as input to the system along with an extra carry input for the adder. The inputs were modelled in Nengo as nodes which provide a constant value of 1 or 0. The individual components of the system were simulated as specified in Section 2 and was interconnected to give a fully functional IEEE 754 floating-point adder working on Spiking Neural Networks (SNN) [39].

The output of each component and the final result were probed at a time interval of 10ms. The mean of these output values were then used to compute the error in each component as well as the entire system. Error in the system was calculated with and without the encoding technique (Section 2.2) for performance comparison. We used the following two metrics for assessing the performance of each component –

$$MAE = \frac{\sum |C_v - A_v|}{n_v}$$

Here, MAE is the Mean Absolute Error
$C_v$ is the Computed value
$A_v$ is the Actual value
$n_v$ is the number of values

In our case MAE represents the measure of error which is present in the output of each component due to the error present in approximating a discontinuous function using NEF and the error due to noise and randomness inherent to spiking nature of the

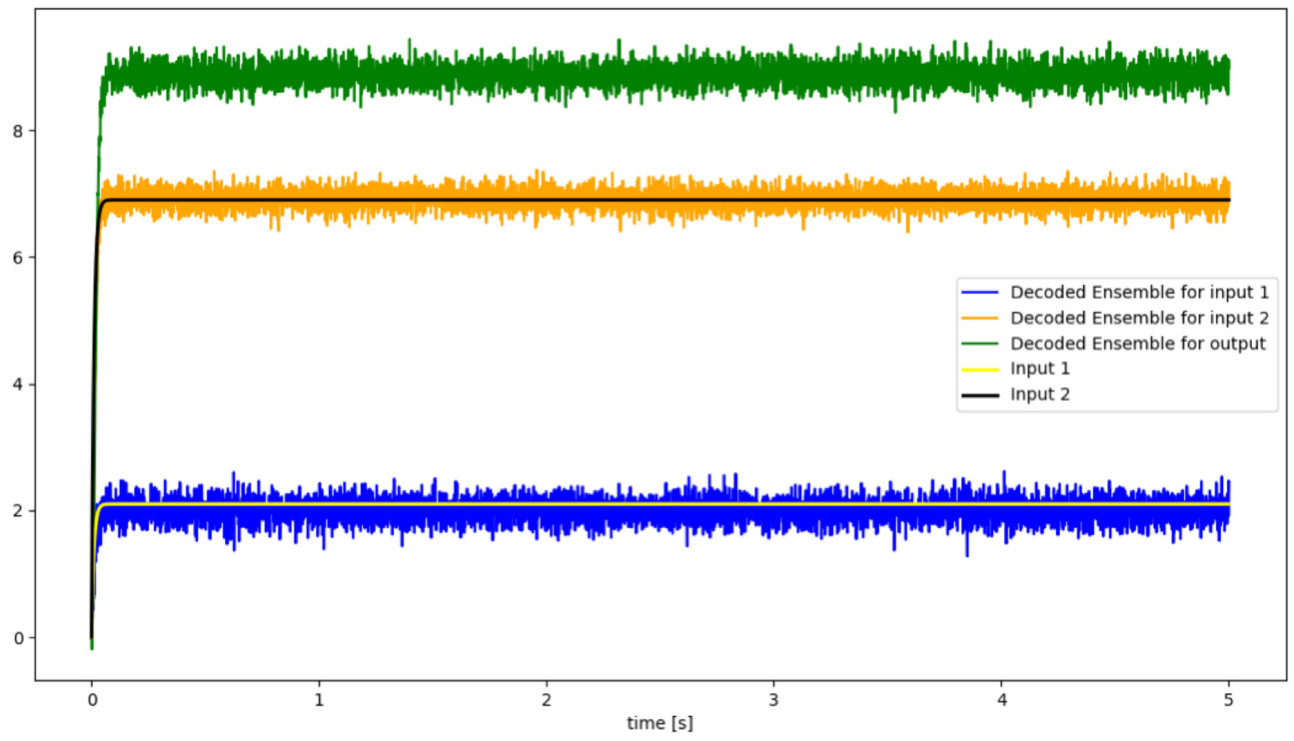

**Fig. 9.** Effect of different values of radius, input 1 = 2.1, input 2 = 6.9, radius = 10.

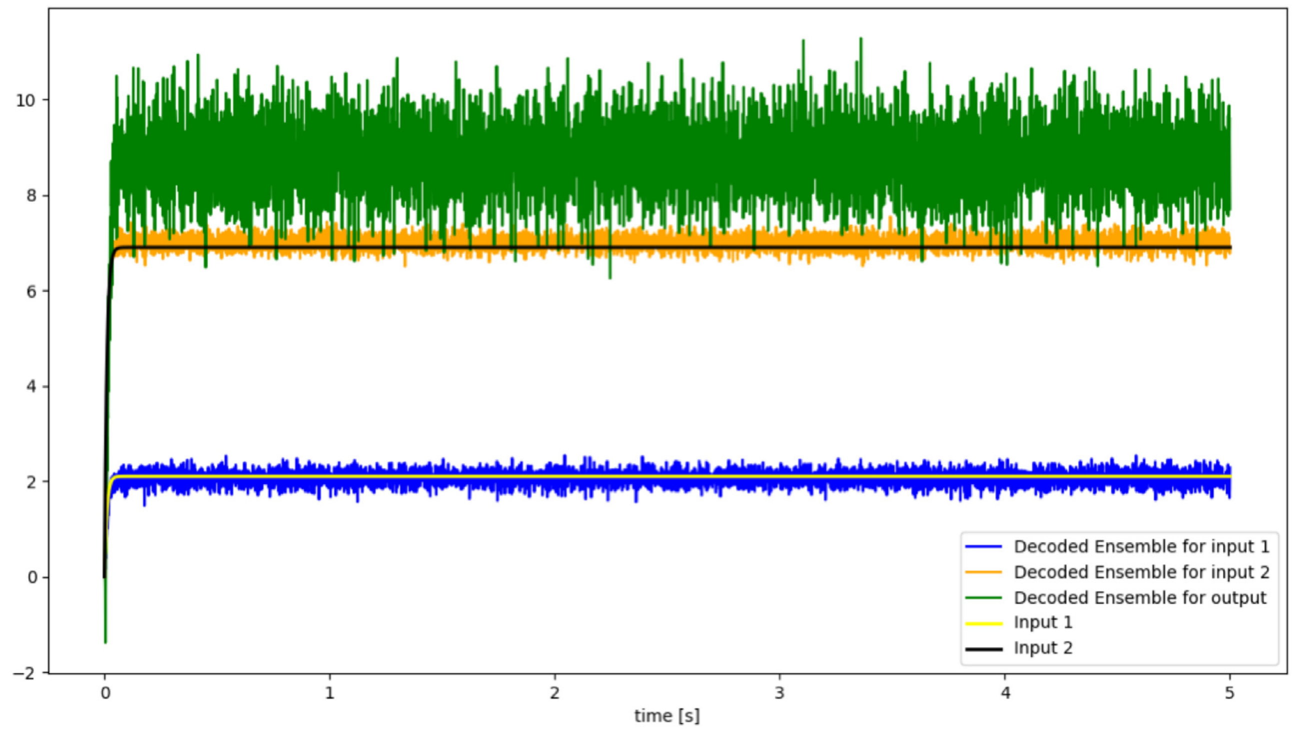

**Fig. 10.** Effect of different values of radius, input 1 = 2.1, input 2 = 6.9, radius = 50.

computation. It is the absolute distance between the actual binary value we expected ($A_b$) and the value computed by our components and the system ($C_v$), normalized by the number of values.

$$MEE = \frac{\sum E_b \oplus A_b}{n_b}$$

Here, MEE is the Mean Encoded Error
$E_b$ is the Encoded bit
$A_b$ is the Actual bit
$n_b$ is the number of bits

Encoding of the output from each component acts as a filter and boosting operation on the output signal which helps in reducing the error present in approximation and noise. MEE is calculated in a similar fashion to that of Hamming distance between the encoded bit values ($E_b$) and the actual bit values ($A_b$) which are expected, normalized by the number of bits in output.

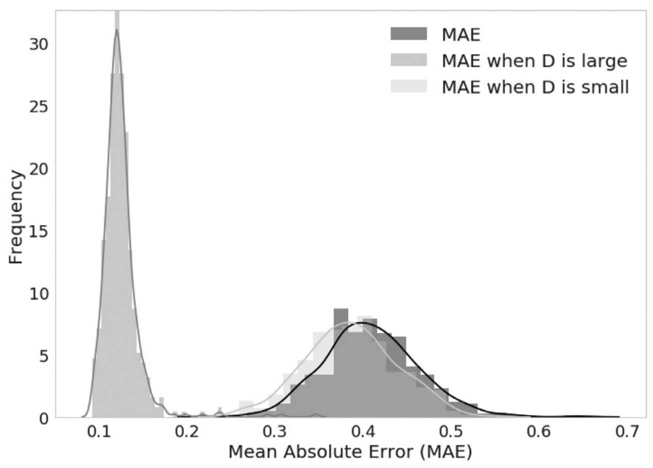

**Fig. 11.** Mean absolute error frequency distribution.

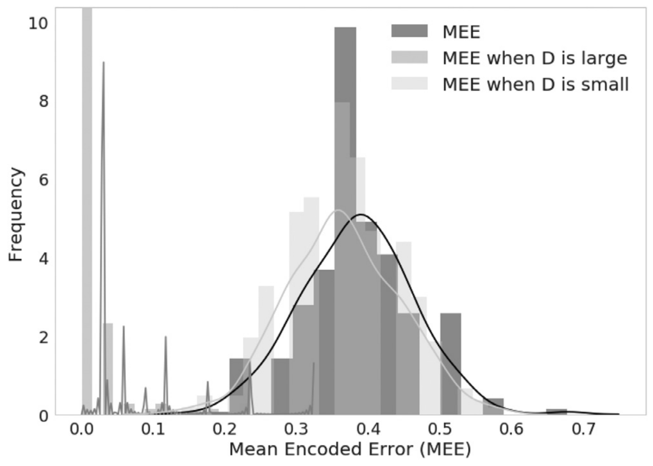

**Fig. 12.** Mean encoded error frequency distribution.

Figs. 11 and 12 illustrate the MAE distribution and the MEE distribution. To generate these graphs, we took three groups of input pairs. Let *D* be the difference between the two inputs. The first group had random input pairs where *D* was large, the second group again had random input pairs but here *D* was small and in the third group the *D* had random values. From Fig. 11 we can infer that there were lots of instances where MAE was low when *D* was large. Whereas there were relatively less instances where MAE was slightly higher when *D* was small or random. Similarly, from Fig. 12 we can infer that there were a significant number of instances where MAE was low when *D* was large. Whereas there were around the same number of instances where MAE was around 0.4 and *D* was either small or random.

Large differences in *D* are more difficult to work with as the values of the exponents are different, which would mean that more shifting of the mantissa bits is needed. A small difference is easier to handle as much shifting is not needed.

### 4.3. Accuracy v/s number of neurons

We define accuracy as the inverse of the MAE. We observed that the accuracy increases with an increase in the number of neurons used to represent one bit. We varied the number of neurons starting from 1 to a maximum of 900 per bit and observed the accuracy across all the components as shown in Fig. 13. From this figure, it is clear that the accuracy increases with an increase in the number of neurons but beyond a certain threshold value (in

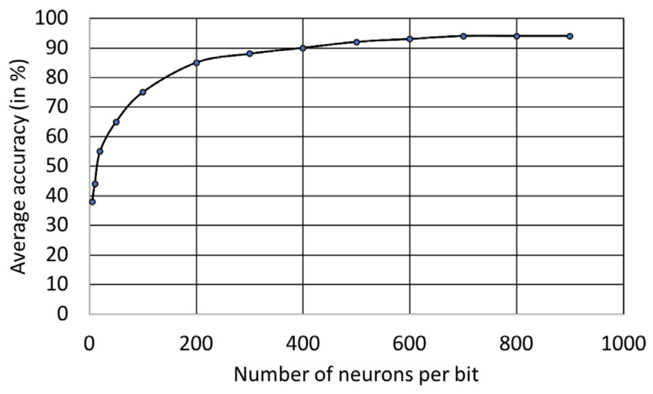

**Fig. 13.** Accuracy v/s number of neurons per bit.

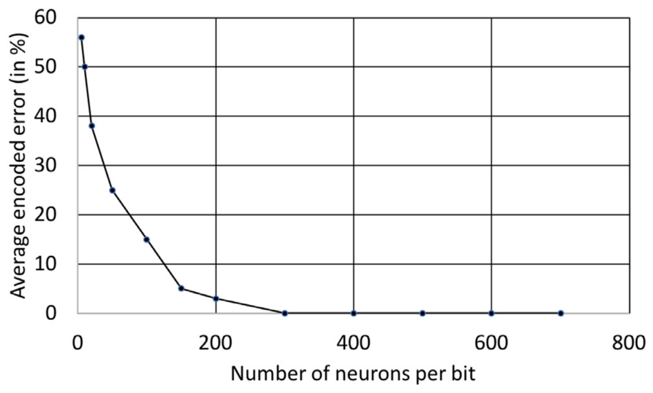

**Fig. 14.** Bit error v/s number of neurons per bit.

this case around 500 neurons) of the number of neurons, the increase in accuracy with increase in number of neurons is not significant. We can also see that the accuracy increases exponentially until the increase in the number of neurons reaches 200 then it begins saturating.

### 4.4. Bit error v/s number of neurons

We define bit error as the number of bits that were generated incorrectly with respect to the actual output. We observed that the bit error was high when number of neurons per bit were very low (close to 1) as shown in Fig. 14. We can see an exponential decrease in the value of the bit error until the number of neurons per bit reaches 300, beyond which the bit error remains 0. Therefore, it is evident that an increase in the number of neurons per bit results in smaller values of error but beyond a certain threshold value, the bit error remains 0. In this case, that value happens to be 300 neurons.

### 4.5. Total number of neurons

According to our observations, the accuracy increases with an increase in the number of neurons as discussed in Section 4.3. With the help of this observation and the threshold (minimum) number of neurons required per bit, we estimated the optimal number of neurons required for all the ensembles in Table 2. The linear ensemble is used to simply output the given input, the addition ensemble adds the two given inputs, the flipping ensemble flips the bit, i.e., changes 0 to 1 and vice versa. The multiplexing ensemble maps two inputs to one output using a selection bit whereas the selector ensemble maps two inputs to two outputs. The right shift ensemble performs the shift operation on the mantissa bits and the overflow/underflow ensemble is used to compute the overflow/underflow flag. The XOR ensemble and output sign ensemble compute the sign bit of the output, i.e., $S_{out}$. We see that the accuracy increases as the number of neurons used increase. We use a threshold of 0.9 for average accuracy (see Fig. 13) to judge how many neurons are needed, and indicate in Table 2 what the numbers are to reach such a threshold.

## 5. Conclusion

In this paper, we look at a relatively unexplored area: numerical computation on neuromorphic computers, and specifically propose an approach to build an IEEE-754 standard complaint floating-point unit using Spiking Neural Networks (SNN) for neuromorphic chips. The architecture that we have presented can either be used in a full-fledged neuromorphic system or as a co-processor in another system. Such usage can not only increase the power efficiency and performance of the system but also help in achieving the dream of true brain-like computing.

Our architecture comprises a complex floating-point unit whose sub-components we have described. This architecture can, because of its modular design, be extended to solve other floating-point arithmetic problems like subtraction, multiplication and division. The complex task of floating-point addition is divided into sub-tasks such as exponent alignment, mantissa addition and overflow/underflow handling. Each component of these sub-systems are then approximated using the Neural Engineering Framework (NEF).

A novel encoding scheme has been used to reduce inter-neural ensemble error. We also estimate the number of neurons required to implement each component, thereby estimating the total number of neurons required for the entire system. With further optimization to the design, this number can be brought down. The value of radius to be used in such a system is another important outcome of our work. The most complex part of the floating-point addition process is the shifting of the mantissa bits and we have accomplished this successfully. Theorems 3.1 and 3.2 are the results that support solving this complex part of the overall problem. This architecture is scalable in the way that it can be used with either floating-point numbers of single precision or double precision. We also indicate the presence of an underflow/overflow (in case it happens) during the addition process, which can then be handled separately. We consider the effect of different values of radius on the accuracy, and also observe the effect of different number of neurons on the accuracy and bit error. Finally, we do a performance analysis of the implementation of our system and draw conclusions from the observations as mentioned in Section 4. The present paper extends the range of computations considered possible using neuromorphic computing, to include classical floating-point addition, thus showing that a neuromorphic chip can hypothetically take over such operations entirely, rather than requiring a traditional CPU as a co-processor. While neuromorphic hardware is yet to reach the point of viability where such a chip can completely replace a CPU, recent developments suggest that this may indeed be possible in the not-too-distant future.

## Declaration of interests

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## References

[1] C. Mead, Neuromorphic electronic systems, IEEE J. Proc. 78 (10) (1990) 1629–1636.
[2] G.E. Moore, Cramming more components onto integrated circuits, IEEE J. Proc. 86 (1) (1998).
[3] D. Monroe, Neuromorphic computing gets ready for the (really) big time, Commun. ACM 57 (6) (2014) 13–15.
[4] S.K. Esser, P.A. Merolla, J.V. Arthur, A.S. Cassidy, R. Appuswamy, A. Andreopoulos, D.J. Berg, J.L. McKinstry, T. Melano, D.R. Barch, C. di Nolfo, P. Datta, A. Amir, B. Taba, M.D. Flickner, D.S. Modha, Convolutional networks for fast, energy-efficient neuromorphic computing, PNAS 113 (41) (2016) 11441–11446 http://www.pnas.org/content/113/41/11441.
[5] G. Volanis, A. Antonopoulos, Y. Makris, A.A. Hatzopoulos, Toward silicon-Based cognitive neuromorphic ICs—a survey, IEEE Design Test 33 (3) (2016) 91–102.
[6] C.D. James, J.B. Aimone, N.E. Miner, C.M. Vineyard, F.H. Rothganger, K.D. Carlson, S.A. Mulder, T.J. Draelos, A. Faust, M.J. Marinella, J.H. Naegle, S.J. Plimpton, A historical survey of algorithms and hardware architectures for neural-inspired and neuromorphic computing applications, Biol. Inspired Cognit. Architect. 19 (2017) 49–64.
[7] S. Agarwal, R.B.J. Gedrim, A.H. Hsia, D.R. Hughart, E.J. Fuller, A.A. Talin, C.D. James, S.J. Plimpton, M.J. Marinella, Achieving ideal accuracies in analog neuromorphic computing using periodic carry, in: Proceedings of the 2017 Symposium on VLSI Technology, 2017. Kyoto, Japan
[8] C.D. Schuman, T.E. Potok, R.M. Patton, J.D. Birdwell, M.E. Dean, G.S. Rose, J.S. Plank, A survey of neuromorphic computing and neural networks in hardware, arXiv: abs/1705.06963 (2017).
[9] Y. Kim, Y. Zhang, P. Li, Energy efficient approximate arithmetic for error resilient neuromorphic computing, IEEE J. VLSI 23 (11) (2015) 2733–2737.
[10] Q. Wang, Y. Li, B. Shao, S. Dey, P. Li, Energy efficient parallel neuromorphic architectures with approximate arithmetic on FPGA, Neurocomputing 221 (2017) 146–158.
[11] P. Maier, F. Hartmann, M. Emmerling, C. Schneider, M. Kamp, S. Höfling, L. Worschech, Electro-photo-sensitive memristor for neuromorphic and arithmetic computing, Phys. Rev. Appl. 5 (5) (2016).
[12] R. Kozma, E.R. Pino, E.G. Pazienza. (2012). Advances in Neuromorphic Memristor Science and Applications. 10.1007/978-94-007-4491-2.
[13] A. Calimera, E. Macii, M. Poncino, The human brain project and neuromorphic computing, Funct. Neurol. 28 (3) (2013) 191–196.
[14] C. Eliasmith, How to Build a Brain: A Neural Arhitecture for Biological Cognition, Oxford Series on Cognitive, Models and Architectures, Oxford Scholarship, 2013 Online: January 2014, doi:10.1093/acprof:oso/9780199794546.001.0001.
[15] S. Reardon, Artificial neurons compute faster than the human brain, Nature (2018) https://www.nature.com/articles/d41586-018-01290-0.
[16] N. Imam, T.A. Cleland, R. Manohar, P.A. Merolla, J.V. Arthur, F. Akopyan, D.S. Modha, Implementation of olfactory bulb glomerular-layer computations in a digital neurosynaptic core, Front. Neurosci. 6 (2012).
[17] M.D. Tissera, M.D. McDonnell, Deep extreme learning machines: supervised autoencoding architecture for classification, Neurocomputing 174 (2016) 42–49 http://www.sciencedirect.com/science/article/pii/S0925231215011327.
[18] Q. Yu, H. Tang, K.C. Tan, H. Yu, A brain-inspired spiking neural network model with temporal encoding and learning, Neurocomputing 138 (2014) 3–13 http://www.sciencedirect.com/science/article/pii/S0925231214003452.
[19] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S.H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C.-K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, Guruguhanathan, Venkataramanan, Y.-H. Weng, A. Wild, Y. Yang, H. Wang, Loihi: a neuromorphic manycore processor with on-chip learning, IEEE Micro 38 (1) (2018) 82–99, doi:10.1109/MM.2018.112130359.
[20] A. Neckar, S. Fok, B.V. Benjamin, T.C. Stewart, N.N. Oza, A.R. Voelker, C. Eliasmith, R. Manohar, K. Boahen, Braindrop: a mixed-signal neuromorphic architecture with a dynamical systems-based programming model 107 (1) (2019) 144–164, doi:10.1109/JPROC.2018.2881432.
[21] G.W. Burr, R.M. Shelby, A. Sebastian, S. Kim, S. Kim, S. Sidler, K. Virwani, M. Ishii, P. Narayanan, A. Fumarola, L.L. Sanches, I. Boybat, M. Le Gallo, K. Moon, J. Woo, H. Hwang, Y. Leblebici, Neuromorphic computing using non-volatile memory, Adv. Phys. X 2 (1) (2017) 89–124, doi:10.1080/23746149.2016.1259585.
[22] S. Moradi, R. Manohar, The impact of on-chip communication on memory technologies for neuromorphic systems, J. Phys. D Appl. Phys. 52 (1) (2019) 014003 http://stacks.iop.org/0022-3727/52/i=1/a=014003.
[23] J. Park, T. Yu, S. Joshi, C. Maier, G. Cauwenberghs, Hierarchical address event routing for reconfigurable large-scale neuromorphic systems 28 (10) (2017) 2408–2422, doi:10.1109/tnnls.2016.2572164.
[24] S. Moradi, N. Qiao, F. Stefanini, G. Indiveri, A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (DYNAPs) 12 (1) (2018) 106–122, doi:10.1109/TBCAS.2017.2759700.
[25] J.B. Aimone, K.E. Hamilton, S. Mniszewski, L.R.C.D. Schuman, W.M. Severa, Non-neural network applications for spiking neuromorphic hardware, in: Proceedings of the Third International Workshop on Post-Moores Era Supercomputing (PMES 2018), 2018. Dallas, TX
[26] M. Hu, J.P. Strachan, Z. Li, E.M. Grafals, N. Davila, C. Graves, S. Lam, N. Ge, J.J. Yang, R.S. Williams, Dot-product engine for neuromorphic computing: Programming 1t1m crossbar to accelerate matrix-vector multiplication, in: Proceedings of the Fifty-third ACM/EDAC/IEEE Design Automation Conference (DAC 2016), 2016, doi:10.1145/2897937.2898010. Austin, TX, USA
[27] IEEE Standard for Floating-Point Arithmetic, in IEEE Std 754-2008, pp. 1–70, (2008) doi: 10.1109/IEEESTD.2008.4610935.

[28] J. Gosmann, C. Eliasmith, Optimizing semantic pointer representations for symbol-like processing in spiking neural networks, PLoS ONE 11 (2016) https://doi.org/10.1371/journxal.pone.0149928.
[29] T.C. Stewart, A technical overview of the neural engineering framework, AISB Q. 35 (2012) http://compneuro.uwaterloo.ca/files/publications/stewart.2012d.pdf.
[30] Nengo documentation release 1.4.0, (http://ctnsrv.uwaterloo.ca/docs/latex/Nengo.pdf). Accessed February 27, 2018.
[31] T. Bekolay, J. Bergstra, E. Hunsberger, T. DeWolf, T. Stewart, D. Rasmussen, X. Choo, A. Voelker, C. Eliasmith, Nengo: a python tool for building large-scale functional brain models, Front. Neuroinf. 7 (2014).
[32] Addition example nengo core 2.6.0 docs, (https://www.nengo.ai/nengo/examples/addition.html). Accessed February 27, 2018.
[33] D.J. Lilja, S.S. Sapatnekar, Designing Digital Computer Systems with Verilog, Cambridge University Press, 2005.
[34] D. Fleisch, J. Kregenow, A Student's Guide to the Mathematics of Astronomy, Cambridge University Press, 2013.
[35] A.R. Voelker, C. Eliasmith, Methods for applying the neural engineering framework to neuromorphic hardware, arXiv:1708.08133 [q-bio.NC](2017).
[36] A.R. Voelker, B.V. Benjamin, T.C. Stewart, K. Boahen, C. Eliasmith, Extending the neural engineering framework for nonideal silicon synapses, in: Proceedings of the 2017 IEEE International Symposium on Circuits and Systems (ISCAS), 2017. Baltimore, MD
[37] N. Cristianini, J. Shawe-Taylor, An Introduction to Support Vector Machines and Other Kernel-based Learning Methods, Cambridge University Press, 2000.
[38] C. Koch, I. Segev, Methods in Neuronal Modeling – From Ions to Networks, 1999.
[39] W. Maass, Networks of spiking neurons: the third generation of neural network models, Neural Netw. 10 (9) (1997) 1659–1671.

**Arun M. George** is a graduate student enrolled in the Information Technology program, with a specialization in data science, at the International Institute of Information Technology, Bangalore. He is also pursuing research in machine learning and artificial intelligence, as part of his internship at Media iQ Research. He has previously worked on fMRI analysis and deep learning.

**Rahul Sharma** is a graduate student enrolled in the information technology program, with a specialization in data science, at the International Institute of Information Technology, Bangalore. Prior to this, he worked with Cognizant Technology Solutions in Pune, Maharashtra, India, in the area of embedded software development. Currently, he is involved with research in the area of computer vision. His research interests are in the field of autonomous vehicles, computational neuroscience, machine learning, and artificial intelligence.

**Shrisha Rao** received his Ph.D. in computer science from the University of Iowa, and before that his M.S. in logic and computation from Carnegie Mellon University. His primary research interests are in artificial intelligence and other applications of distributed computing, including in bioinformatics and computational biology, as well as algorithms and approaches for resource management in complex systems such as used in cloud computing. He also has interests in energy efficiency, sustainable computing ("Green IT"), renewable energy and microgrids, applied mathematics, and intelligent transportation systems. Dr. Rao is an ACM Distinguished Speaker and a Senior Member of the IEEE. He is also a life member of the American Mathematical Society and the Computer Society of India.