Construction of a Spike-Based Memory Using Neural-Like Logic Gates Based on Spiking Neural Networks on SpiNNaker

Alvaro Ayuso-Martinez, Member, IEEE, Daniel Casanueva-Morato, Member, IEEE, J. P. Dominguez-Morales, Member, IEEE, Angel Jimenez-Fernandez, Member, IEEE, and Gabriel Jimenez-Moreno, Member, IEEE

Abstract—Neuromorphic engineering concentrates the efforts of a large number of researchers due to its great potential as a field of research, in a search for the exploitation of the advantages of the biological nervous system and the brain as a whole for the design of more efficient and real-time capable applications. For the development of applications as close to biology as possible, Spiking Neural Networks (SNNs) are used, which are considered biologically-plausible and constitute the third generation of Artificial Neural Networks. This work presents a spiking implementation of a memory, which is one of the most important components in computer architecture. In the process of designing this spiking memory, different intermediate components were also implemented and tested. The tests were carried out on the SpiNNaker neuromorphic platform. This work goes into the development of spiking blocks using a logic gate approach based on previous work and includes a comparison between other works in the state of the art related to spiking memories and the one proposed here. All the implemented blocks and developed tests are available in a public repository.

Index Terms—Spiking memory, spiking neural networks, neuromorphic engineering, SpiNNaker, bio-inspired building blocks.

I. INTRODUCTION

NEUROMORPHIC engineering was presented by Carver Mead in the late 1980s [1]. This concept proposed the development of hardware and software applications based on the fundamental principles of biological nervous systems, which are among the most optimal and useful natural mechanisms. Over the years, this concept has given rise to a new and interesting field of research that has greatly evolved in the last two decades [2].

Neuromorphic systems are analog, digital or mixed-signal systems that rely on artificial neurons and spikes to transmit information. In biology, these spikes are large peaks in the membrane potential of neurons that occur when the membrane potential reaches a specific threshold. To mimic this behavior, artificial neurons in neuromorphic systems generate these spikes as asynchronous electric pulses. Thanks to this bio-inspired approach, neuromorphic systems can achieve low power consumption and high real-time capability, which could greatly improve the performance and possibilities of existing systems.

As these neuromorphic systems aim to mimic biological nervous systems, it is not only necessary to use bio-inspired neurons, but also bio-inspired architectures. A specific type of biologically-plausible neural networks called Spiking Neural Networks (SNNs) are commonly used for this purpose. These SNNs have two basic bio-inspired elements: neurons and synapses. They can be seen as graphs according to mathematical graph theory, where neurons would be nodes and synapses would be edges, which have associated weights (as in other types of neural networks) and delays.

There exists two different alternatives to work with SNNs in order to develop neuromorphic applications: software simulators and hardware platforms.

Both software simulators and hardware platforms allow building and testing new applications by implementing simulated SNNs based on artificial neuronal models. While software simulators can run on any platform with minimal requirements, neuromorphic platforms aim to increase the performance of simulations using specific hardware.

Some very popular examples of software simulators are NEST [3] and Brian [4]. In the case of neuromorphic hardware platforms, although most of them are fully digital, such as SpiNNaker [5], Loihi [6] or TrueNorth [7], there are other platforms, including BrainScaleS [8], which use a mixed-signal approach.

As is explained in [9], biological mechanisms in neurons and synapses in the brain use energy-efficient analog techniques that lack the noise immunity of traditional digital systems and are not fully deterministic, two aspects that are inherent to analog circuits. Since mixed-signal neuromorphic hardware platforms contain analog circuits, they are closer than their digital counterpart to this biological processing in the brain. Using these
platforms is a great option to achieve applications that are as bio-inspired as possible and in which great precision in the signal and pure determinism are not needed. For those cases in which those aspects are needed, digital neuromorphic platforms should be considered.

There is a long list of interesting neuromorphic applications that promise great advances in many other different fields. Some of these fields are mentioned in [2], such as image processing [10] and emotion recognition [11]. Other related works focus on speech recognition [12], [13], sensory fusion [14], [15], motor control [16], [17] or bio-inspired locomotion [18], [19]. This evidences the evolution and importance of neuromorphic engineering, whose progress allow thinking of new and exciting possible future applications, which will require a better understanding of SNNs and current applications.

As there are no rules on how to build a SNN in order to achieve specific behaviors or functionalities, the development of applications that require them may not be a simple task for neuromorphic engineers. To deal with this, a novel set of building blocks based on SNNs was proposed in [20] where the functionalities of some of the most basic digital circuits are implemented, including essential logic gates such as OR, AND and NOT. In Boolean algebra any function can be constructed from these gates, which guarantees the possibility of implementing a digital circuit that performs the desired function. By implementing building blocks based on SNNs that provide such basic functions, this possibility is also extended to the spiking domain. Thus, this new set of building blocks establishes a basis for the implementation of more complex blocks with functionalities similar to those of complex digital components. In this way, this new approach uses bio-inspired structures (SNNs) to achieve functions that can be useful for neuromorphic engineers in the development of their research work.

As mentioned in the aforementioned work, one of the functions to be implemented that may be of most interest is memory. Memory is essential in the design of traditional digital circuits, so without it most of today’s computers could not have been implemented as we know them. On the other hand, memory is also of great importance in the evolution and behavior of living beings. The interest in understanding how memory occurs in the biological nervous system has been reflected in the work of neuroscientists and neuromorphic engineers [21], [22], [23], [24]. However, few works have succeeded in implementing systems based on SNNs with the ability to memorize input spikes. An implementation of an associative memory based on Spike-Timing-Dependent Plasticity (STDP) [25] was presented in [26]. It uses Hebb’s learning rule [27] to dynamically build connections between two internal layers to achieve the recall ability of biological memory. Other works also proposed implementations of associative memories following Hebb’s learning rule to make synaptic modifications in their recurrent network models [28], [29], [30]. All of them are far from the logic gate approach.

This work proposed the implementation of a spiking memory block based on the previously mentioned building blocks. Its main contributions include the following:

- The development of intermediate level spiking functional blocks needed to build the memory block.
- The final implementation of this memory block.
- The study of the resources required for such implementations.
- The development of an exhaustive set of experiments to prove that the behavior of all blocks is as expected.
- The release of all the implemented code in a public repository\(^1\) easily accessible to the entire neuromorphic community.

The SpiNNaker hardware platform was used for running large-scale neural network simulations in real time.

The rest of the paper is structured as follows: Section II introduces memory design and provides the theoretical basis used to build the spiking memory block; in Section III, the software and hardware materials used, as well as some other concepts, are detailed in depth; Section IV introduces some of the blocks presented in [20] that were necessary for the development of the spiking memory block; in Section V, the process of developing the spiking memory block is explained, starting from building blocks to construct other intermediate blocks, and all the designs are shown; in Section VI, a list of the tests that were performed is presented with a discussion of their associated results; in Section VII, a comparison is made between the related works and this work, breaking down the most important differences between the paradigms used. The importance of the proposed blocks in the development of future applications or new components is also discussed in depth; finally, in Section VIII, the conclusions of the work are presented.

### II. First Steps in Memory Architecture Design

Computers need to store large amounts of information, which is divided into instructions and data; programs are made of instructions that use data to perform operations. Thus, memory is a key component within their architecture. In the digital domain, the basic memory unit is the bistable circuit, which is an electric circuit with two stable states, zero and one, with the ability of storing one bit for an indefinite period of time.

There are two main types of bistable circuits: asynchronous and synchronous circuits, which are commonly called latches and flip-flops, respectively [31]. Moreover, there are different types of latches and flip-flops, whose names usually depend on their inputs and outputs.

Registers are arrays of these bistable circuits. To have the capability of addressing these registers, a decoder is needed. A decoder is a component with \(n\) inputs and \(2^n\) outputs, in which the inputs can be seen as the binary representation of a number and the outputs can be seen as a representation of that number in one-hot encoding, which means that each output channel represents a unique number. Decoders are built from basic logic gates, usually NOT and AND gates. Its digital circuit is shown in Fig. 1.

Fig. 2 shows the usual structure of a memory based on bistable blocks and the decoder mentioned above. Note that this is the

\(^1\)https://github.com/alvayus/sPyBlocks

Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.
Figure 1. Diagram of a decoder circuit based on NOT and AND gates, extracted from [32]. This decoder has 3 inputs and 8 outputs. Note that the AND gates in this figure have a fan-in of 2.

Figure 2. Diagram of the usual memory structure. This example shows a memory of 8 registers, one of each has a width of one byte.

III. MATERIALS AND METHODS

A. Spiking Neural Networks

There are currently three different generations of Artificial Neural Networks (ANNs). While the second generation is associated with Deep Learning, SNNs form the basis of the third generation [33]. All ANNs share a common structure: they are always built from neurons (nodes) and synapses (connections). This structure is based on the biological nervous system, although it has a different level of abstraction in the third generation: while the first two generations only preserve the structure (neurons and synapses without deeply considering its biological aspects), SNNs are characterized for the use of bio-inspired neuron and synapse models. Thus, the first two generations are based on highly simplified brain dynamics [34] and SNNs are the closest current approach of neural networks to biological functioning [35].

Another important aspect in SNNs is the way in which the information is transmitted. In the biological nervous system, information is transmitted across synapses in the form of spikes, which are large peaks in the membrane potential of neurons that occur when the membrane potential reaches a certain threshold potential. Spikes are generated in neurons and then propagated to other neurons across synapses. In the artificial approach, which is similar to its biological counterpart, these spikes are represented by asynchronous electric pulses.

Although SNNs are more complex than ANNs, information coded in spikes makes them more energy-efficient, as they deal with precise timing, having a low computational cost. This precise timing is also related to sparse coding, which means that the spike rate is usually low. This justifies the low power consumption of SNNs [35]. Some improvements in the hardware implementation, such as avoiding multiplications, processing spikes using shifts and sums, and only transmitting single bits of information instead of real numbers, allow achieving real-time execution [36].

B. Neuromorphic Hardware Platform

As was mentioned in Section I, the SpiNNaker platform was used to design new functional blocks and test their correct operation. SpiNNaker is a massively-parallel multi-core computing system that was designed to allow modelling very large SNNs in real time and whose interconnected architecture is inspired by the connectivity characteristics of the mammalian brain [5].

Both SpiNN-3 and SpiNN-5 machines were used in this work. The main difference between them is the number of chips: while SpiNN-3 has 4 chips, SpiNN-5 has 48 chips. Each of the chips is made up of 18 ARM968E-S cores operating at 200 MHz. More details regarding these machines can be found in [37].

Since our designs aim to use an optimal amount of resources, all of them can be simulated in both platforms. On the other hand, to perform simulations of blocks that require more inputs, more outputs or a greater number of internal blocks, as for example in the case of the memory, whose capacity depends on the number of internal latches, a SpiNN-5 machine is used.

C. Software Packages

PyNN [38] is a Python package for the simulator-independent specification of neuronal network models. Currently, PyNN supports NEURON [39], NEST [3] and Brian [4] as neural network software simulators, as well as the SpiNNaker [5] and BrainScaleS neuromorphic hardware systems. Thanks to this Python package, the whole code can be executed in all supported
simulators and hardware platforms. In this work, PyNN 0.9.6 was used.

Another important software package is sPyNNaker [40], which is required to work with PyNN and the SpiNNaker hardware platform. In this work, sPyNNaker 6.0.0 was used.

Other software packages that were used to test the implemented designs are Matplotlib 3.5.1, XlsxWriter 3.0.2, and Numpy 1.22.1.

D. Neuron Model and Parameters

All functional blocks presented in this work use the Leaky Integrate-And-Fire (LIF) neuron model and were made to be as independent as possible from the neuron parameters used. Thus, their functionality is based on the behavior of the network as a whole, rather than on individual neurons. In this way, static synapses with different weights and delays are used to achieve specific behaviors, which ensures that the designed functional blocks will work as intended in any case, disregarding the neuron parameters. Notice that, since static synapses are the basis for making these designs, there is no learning involved.

However, although the proposed designs were made to be as independent as possible from the parameters of the neurons, there are two fundamental details that must be taken into account when selecting these parameters:

First, a set of input spikes arriving to a neuron at the same timestep should make it fire once. This is very important for the designed blocks to work as expected. On the other hand, since only one spike is needed to know the result of the operation performed by the block, if the output response of the neurons contains more than one spike, performance would be diminished, since output responses would be longer and the sets of inputs of consecutive operations should be separated in time to avoid the overlap of output responses, which could lead to unexpected behaviors of the functional blocks. Thus, it must be ensured that the expected set of inputs produces exactly one output spike, which will prevent the decrease in the efficiency of the designed blocks, i.e., the decrease in the operating frequency.

Fig. 3 shows the optimal output response at the left of the dotted line (a), in which a neuron fires one output spike per input spike. Note that there is a latency of 1 ms and there is no overlap between the spikes in the output response. At the right of the dotted line (b), Fig. 3 shows the output response of a neuron that fires two output spikes per input spike. Since firing every millisecond would cause an overlap in the output spikes, input spikes were separated in time. Thus, these are separated by as many milliseconds as output spikes are contained in the output response for a single operation, in this case, 2 ms.

Second, it is necessary to ensure that, after firing, these neurons will not receive any more input spikes until they return to the resting potential, which is necessary for the next operation to be performed correctly.

Table I shows the set of neuron parameters that have been used for all neurons in this work. This set fulfills the two requirements mentioned above and has been explicitly chosen for SpiNNaker. It may change for other neuromorphic platforms or simulators as the models of the neurons or the meaning of their parameters may vary.

IV. Spiking Building Blocks

This section gives a brief introduction to some of the blocks presented in [20] thanks to which the different components of the memory block to be developed in this work can be built. These blocks are OR, NOT, AND (classic and fast) gates and SR Latch, as well as Constant Spike Source, which is needed for the development of NOT and fast AND gates. Their designs are shown in Fig. 4.

In addition to the assumptions made in Section III, the aforementioned work explains that, to make an analogy with Boolean logic, 1 will be represented by the existence of a spike at a given time. Thus, the absence of spike at that time is associated with 0.

In both the previous work and this one, weights and delays of 1 are used for standard synapses, which are those whose weights or delays are not explicitly specified, and a timestep of 1 ms is used for the simulation.

A. OR

This is the simplest block, since it consists of a single neuron. Its main function is to fire an output spike when at least one spike is received through the input synapses.

B. Classic AND

Classic AND uses an OR gate to generate an inhibition of weight $n - 1$ in the output neuron, where $n$ is the number of inputs of the block, when at least one input spike is received. This inhibition coincides with the delayed input spikes in the
output neuron. When the inhibition weight is less than the sum of the weights of the synapses through which an input spike is received, the output neuron fires, i.e., when \( n \) input spikes are received.

C. SR Latch

This is the basic element of spiking memory. It consists of a neuron and a self-excitative synapse thanks to which the output spike can be retained, which means that an output spike makes the neuron to fire in a loop. There are two input synapses: set and reset synapses. The first allows the entry of a new input spike, while the second allows the “release” of the retained spike by compensating for its effect on the membrane potential by generating an inhibition.

D. Constant Spike Source

This block makes use of an SR Latch for the continuous generation of output spikes. This idea, as explained in the reference work, may be counterintuitive because one of the advantages of SNNs is precisely their low computational cost. However, this mechanism is particularly necessary in the case of the NOT gate due to the complicated challenge of representing the absence of information by means of spikes. Once this block is available, it can be used to improve the blocks already defined. This is why it is also used for fast AND gate implementation.

Note that only one output spike will be fired at each timestep of the simulation.

E. NOT

When there are no input spikes, NOT gate is expected to generate a continuous stream of spikes. For this reason, NOT gate must use a Constant Spike Source. This Constant Spike Source is connected to NOT gate via an excitatory synapse. In addition, this block has an inhibitory input connection through which the input data is propagated. When there is no input data, the output of the block is a continuous stream of spikes produced in response to the continuous stream of input spikes from the Constant Spike Source, but when there is input data an inhibition is generated that punctually compensates for the excitation produced by it.

F. Fast AND

This variation of the AND gate makes it possible to eliminate the delay of the input synapses of the output neuron by replacing the OR gate with a continuous stream of spikes produced by the Constant Spike Source. Thus, at each timestep of the simulation, an inhibition of weight \( n - 1 \) is generated in the output neuron. Notice that eliminating the delay of the input synapses decreases the block latency, maximizing the number of operations that can be performed.

V. Designs

Logic gates contained in a digital component can be replaced by their associated spiking building blocks in order to achieve a higher-level spiking component that performs the same function as the original digital circuit. Both spiking building blocks
and higher-level spiking components are also named spiking functional blocks.

As was explained in the previous sections, the main objective of this work was to provide a spiking implementation of a memory block, a process that started with the development of a spiking decoder.

Spiking SR latches have been presented as the starting point for storing spiking information. Since biological nervous systems do not seem to use any kind of clock signal, the use of asynchronous circuits, such as these latches, seems the most convenient. In this way, digital bistable blocks in Fig. 2 should be replaced by new spiking blocks with a similar function, which would be spiking SR latches enhanced with the capability to store a bit value at a specific time. These new blocks are called spiking D latches.

Both spiking decoder and spiking D latches were implemented in order to achieve the spiking implementation of the memory block. In order to carry out experiments to prove the correct functioning of spiking decoder, a spiking implementation of encoder, a circuit which fulfills its inverse function, is also developed.

The designs of these new spiking functional blocks are shown in Fig. 5, while the design of the spiking memory block is presented in Fig. 6. Both figures include a legend indicating the meaning of each color and symbol used.

The development process of the spiking implementation of decoder and encoder is based on the truth table of their associated digital circuits. To calculate the total resources used in each implemented block, the resources used by each block contained in them together with the number of blocks of each type were taken into account. The calculations have statements in the form number of repetitions * number of neurons/synapses.

A. Decoder

As was explained in Section II, decoder takes its inputs as the binary representation of a number and outputs its one-hot representation, assigning an output channel to each of the possible values. In Fig. 1, the circuit of a basic digital decoder built from NOT gates and AND gates with a fan-in of 2 is shown. Replacing the digital NOT and AND gates with spiking NOT and classic or fast AND gates, a spiking design of decoder can be achieved. This design is shown in Fig. 5(a).

Table II is the truth table of a decoder with 2 inputs and 4 outputs. For each row, values in the input columns (S0 and S1) represent the connections with a single AND output gate. While “1 s” imply connections directly from the input, “0 s” imply connections from their negated values, i.e., from their associated NOT gates. These connections target the AND gate whose column is “1”. Thus, e.g., the second row (S1 and S0) connects S0 and the NOT gate associated with S1 to AND gate 1.

When calculating the total number of neurons and synapses used in the development of this spiking implementation, as there are n inputs and 2^n outputs in decoder, there are n NOT and 2^n AND gates. One way to count connections from inputs and NOT gates to AND gates is knowing that, since a truth table is used and there are 2^n possible combinations with the values of the inputs that arise from alternating the binary values in said inputs, there will be 2^n−1 connections from each single input and 2^n−1 connections from their associated NOT gate. Note that 2^n−1 = 2^n/2, and that the number of total connections by

---

Fig. 5. Diagram showing the design of each implemented block. The legend on the right shows the meaning for each color and symbol. (a) Decoder. (b) Encoder. (c) D Latch.

| S1 | S0 | AND 3 | AND 2 | AND 1 | AND 0 |
|----|----|-------|-------|-------|-------|
| 0  | 0  | 0     | 0     | 0     | 1     |
| 0  | 1  | 0     | 0     | 1     | 0     |
| 1  | 1  | 0     | 0     | 0     | 0     |

S1 and S0 are input columns, while 3, 2, 1 and 0 are output columns.
Fig. 6. Example design of a spiking memory block with 3 words (rows) of 3 bits (columns) using classic AND gates.

On the other hand, NOT and fast AND gates need a Constant Spike Source (CSS) block that provides them with constant spikes. Thus, some more resources were added:

- Nodes (CSS): 2
- Connections (CSS)
  - Internal CSS: 2
  - CSS to NOT: 2
  - CSS to AND (fast): 2

Therefore, the number of total neurons depends on the type of AND gate used, and, if the CSS resources are included, the result of the calculation is the following:

- Total resources (Decoder with classic AND + CSS)
  - Neurons: $2^n + n + 2$
  - Synapses: $2^n * (n + 1) + 3n + 2$

where $n$ is the number of inputs of the decoder block. Notice that the use of fast AND gates reduces the number of neurons and synapses required for this design.

### B. Encoder

Encoder performs the inverse function of decoder, that is, converting from one-hot representation to binary representation. A spiking implementation of this requires only OR gates, as is shown in Fig. 5(b). Notice that, in this figure, there are 4 inputs and 2 outputs, which match the number of outputs and inputs of the decoder block, respectively.

In this case, the number of neurons needed can be calculated directly. Since spiking OR gates are a single neuron, the number of neurons is the number of outputs. Given that $n$ is the number of inputs and $2^n$ is the number of outputs in the decoder block, the number of inputs and outputs in the encoder block is $2^n$ and $n$, respectively. As can be seen, the numbers are reverted.

Let $n$ be the number of inputs in the encoder block, the number of outputs in this block would be $\lceil \log_2(n) \rceil$, matching with the number of neurons. The ceiling function has to be used to correctly output the binary representation of the set of inputs, which represents a value in one-hot encoding.
Table III, which shows the truth table of a encoder with 4 inputs and 2 outputs, is used to calculate the number of synapses the encoder block contains. Note that the last two columns in the table show “1 s” when the input is connected to the OR block specified at the top of the column, and, thus, the number of synapses can be calculated as the number of “1 s” in these columns. This procedure is very similar to the one already explained for decoder; however, in this case, there is no need to count the zeros, since there are no NOT gates. Furthermore, the truth table of encoder is the truth table of decoder, although reversed, which makes sense since this block performs its inverse function.

In a similar way to what was explained in Section V-A, there are $2^n$ possible binary values represented by the output columns of Table III. However, the number of inputs cannot be assumed to be a power of two, and thus the number of 1 s is not necessarily $2^n/2$ in each of those columns. The formulae to calculate the total number of resources in this block are as follows:

- **Total resources (Encoder)**
  - Neurons: $\lceil \log_2 n \rceil$
  - Synapses: $\sum_{i=2}^{n} \text{ones}(\text{bin}(i-1))$

where $n$ is the number of inputs of the block and $\text{ones}(\text{bin}(i-1))$ is the number of 1 s of the binary representation of $i-1$, where $i$ represents the rows of the table. Note that this equation ignores the first two rows, as they do not contribute anything to the output. In Table III, the second row has one input but no outputs, since it is associated with the non-operation output channel of the decoder block. In the encoder design shown in Fig. 5(b), D0 represents this first non-operation input, and is not connected to achieve the desired behavior.

### C. D Latch

As was explained in this section, D Latch is required for the design of a memory block and can be built from SR Latch. While SR latches need to receive an input spike through their set connections to set the latch and through their reset connections to reset it, D latches need to receive an input spike through their store connection to store the value (1 if there is a spike, 0 if not) of their data connection.

This new latch is required to build the memory block since, as explained at the beginning of this section, it is necessary to address its different registers. This implies the use of the decoder output as a control signal. In this case, that signal will be used to directly perform write operations on the D latches thanks to the store connection, and not as a pure enable signal.

The spiking design of a D Latch is shown in Fig. 5(C). Note that there are not only store and data connections, but also a connection called data. This connection is expected to receive the negated value of the input data, using an external NOT gate. In the lower part of this design there is a rectangle delimiting a neuron and its associated connections, which corresponds to the design of an SR Latch.

The number of neurons and synapses needed are then calculated:

- **Neurons**
  - AND (classic): $2 \times 2 = 4$
  - AND (fast): $2 \times 1 = 2$
  - SR Latch: $1 \times 1 = 1$

- **Synapses**
  - Store to AND (classic): $2 \times 2 = 4$
  - Data to AND (classic): $1 \times 2 = 2$
  - Data to AND (fast): $1 \times 1 = 1$
  - Internal AND (classic): $2 \times 1 = 2$
  - Store to AND (fast): $2 \times 1 = 2$
  - Data to AND (fast): $1 \times 1 = 1$
  - AND to SR Latch (set): $1 \times 1 = 1$
  - AND to SR Latch (reset): $1 \times 1 = 1$
  - Internal SR Latch: $1 \times 1 = 1$

A CSS block should also be introduced to allow the correct operation of fast AND gates. To facilitate the calculation of resources used for the memory design, CSS neurons and synapses are not included in the total resources of a single D Latch, which are the following:

- **Total resources (D Latch with classic AND)**
  - Neurons: 5
  - Synapses: 13

- **Total resources (D Latch with fast AND. CSS block not included)**
  - Neurons: 3
  - Synapses: 7

The amount of total resources needed to build a D Latch is higher than an SR Latch, since it adds AND gates to achieve the desired behavior.

### D. Memory

The memory block was introduced in Section II. While Fig. 2 shows the basic digital circuit for a memory block, replacing its digital components with the spiking ones presented in this section is enough to reach the associated spiking design, which is shown in Fig. 6.

This memory block is mainly made up of one decoder and a matrix of D latches. The first output channel of the decoder corresponds to the non-operation channel, thus it will be constantly firing spikes when no input spikes are received. Using these output spikes would mean that the associated register (an array of D latches, a row in the matrix of D latches) would be constantly performing writing operations. Therefore, this channel is not used and there is no register associated with it.

Notice that, in Fig. 6, there are rectangles delimiting both the decoder and each D Latch. Moreover, there are additional NOT gates...
gates which have the function of negating each data bit at the top of the design. The negate output is used by all D latches in the same column of the matrix of D latches.

Section V-A shows the number of total resources required for building a spiking decoder together with the needed CSS block. These resources are taken as the starting point to calculate the total resources of this memory block. Let \( r \) be the number of registers (rows) and \( c \) their width, i.e., the number of bits of each register (columns), and the total number of D latches \( r \times c \). As the CSS block is already included, it is only needed to sum the following resources:

- **Added neurons**
  - NOT: \( c \times 1 = c \)
  - D latches (classic AND): \( r \times c \times 5 \)
  - D latches (fast AND): \( r \times c \times 3 \)
- **Added synapses**
  - Data to NOT: \( c \times 1 \)
  - CSS to NOT: \( 2 \times c \)
  - D latches (classic AND): \( r \times c \times 13 \)
  - D latches (fast AND): \( r \times c \times 7 \)
  - CSS to AND (fast AND): \( r \times c \times 2 \times 2 \)

Note that the number of registers is equal to the number of outputs of the decoder minus one, since the first output of the decoder has no register associated with it. Thus, \( r = 2^n - 1 \). As it is more common to work with the number of registers and their width, \( n \) can be isolated from the formula. Then, the following expression defines the number of inputs of the decoder: \( n = \lceil \log_2(r + 1) \rceil \). The use of the ceiling function is required to address all decoder output channels.

Replacing \( n \) in the last two formulae presented in Section V-A, the total amount of resources needed to build the decoder and CSS blocks based on the number of registers, \( r \) (not in the number of inputs), can be calculated. The resulting formulae, simplified, are as follows:

- **Total resources (Decoder with classic AND + CSS)**
  - Neurons: \( 2r + \lceil \log_2(r + 1) \rceil + 4 \)
  - Synapses: \( r + (2r + 5)\lceil \log_2(r + 1) \rceil + 3 \)
- **Total resources (Decoder with fast AND + CSS)**
  - Neurons: \( r + \lceil \log_2(r + 1) \rceil + 3 \)
  - Synapses: \( 2r + (r + 4)\lceil \log_2(r + 1) \rceil + 4 \)

Adding the previously calculated resources to these formulae, the final formulae for calculating the total resources of the memory block can be obtained:

- **Total resources (Memory with classic AND + CSS)**
  - Neurons: \( 2r + c + 5rc + \lceil \log_2(r + 1) \rceil + 4 \)
  - Synapses: \( r + 3c + 13rc + (2r + 5)\lceil \log_2(r + 1) \rceil + 3 \)
- **Total resources (Memory with fast AND + CSS)**
  - Neurons: \( r + c + 3rc + \lceil \log_2(r + 1) \rceil + 3 \)
  - Synapses: \( 2r + 3c + 11rc + (r + 4)\lceil \log_2(r + 1) \rceil + 4 \)

The current memory implementation does not use read signals to allow the internal values of the D latches to be read, which is an aspect that could be interesting to implement in the near future to allow the memory block to be connected with other implemented spiking functional blocks or design higher level spiking functional blocks. For now, the internal value of D latches should be read immediately with a direct connection from their output.

### E. Resources Comparison

Table IV groups all the formulae to calculate the resources of the spiking functional blocks presented in this work based on the number of inputs \( (n) \), while Table V groups the formulae to calculate the resources of said blocks depending on the number of outputs \( (m) \). Both tables are presented to facilitate the calculation of the resources used in any possible case.

In the case of the memory block, the formula depends on \( r \), which is the number of registers (rows of the matrix of D latches). \( m \) and \( r \) are defined by the following expressions:

- \( m = 2^n \) for the decoder
- \( m = \log_2 n \) for the encoder
- \( r = 2^n - 1 \) for the memory

Note that the number of resources of the memory block also depends on \( c \), which is the number of bits (columns of the matrix of D latches).

All formulae in Table IV are explained in this section, except the one associated with the memory block. Likewise, only the formulae associated with this memory block in Table V have been explained. The rest of the formulae are obtained by substituting the variables according to the previous formulae, and they are presented for the sake of simplicity. These formulae assume that all inputs are properly connected, as is shown in the corresponding designs.

Moreover, Table VI shows the latencies for each of the presented blocks, depending on the type of the AND block used. These latency values include all delays from the input connection to the output neuron. Here it is important to emphasize that standard synapses have a delay of 1 ms, as it was explained in Section IV.

### VI. RESULTS

In this section, the implementations of the components whose designs were presented in Section V are tested to prove their expected behavior. To this end, some of the most interesting experiments proposed were the following: a combined component test for decoder and encoder, a multiple D Latch test and a complex memory test. The combined component test served to prove the correct behavior of all the components involved, in this case, decoder and encoder. Additional individual tests are not presented in order to avoid repeating similar results.

All these tests also demonstrated that the number of total resources used match those calculated theoretically using the formulae presented in Section V. The practical calculation of these resources, which was done by counting one by one each of the resources added in the simulation phase, is independent of the theoretical calculation.

#### A. Decoder - Encoder

To test the correct operation of the spiking decoder and encoder designs, a short test was developed in which a decoder with continuously increasing binary values at its input was connected
TABLE IV

| Block         | AND type | Total neurons | Total synapses |
|---------------|----------|---------------|----------------|
| Decoder + CSS | Classic  | $2^n + n + 2$ | $2^n * (2n + 1) + 3n + 2$ |
|               | Fast     | $2^n + n + 2$ | $2^n * (n + 2) + 3n + 2$ |
| Encoder       | -        | $\lceil \log_2 m \rceil$ | $\sum_{i=0}^{\lceil \log_2 m \rceil} \text{ones}(\text{bin}(i-1))$ |
| D Latch       | Classic  | 5             | 13             |
|               | Fast     | 3             | 7              |
| Memory + CSS  | Classic  | $2^n(5c + 2) + n - 4c + 2$ | $2^n * (2n + 13c + 1) + 3n - 10c + 2$ |
|               | Fast     | $2^n(3c + 1) + n - 2c + 2$ | $2^n(2n + 11c + 2) + 3n - 8c + 2$ |

TABLE V

| Block         | AND type | Total neurons | Total synapses |
|---------------|----------|---------------|----------------|
| Decoder + CSS | Classic  | $2m + \lceil \log_2 m \rceil + 2$ | $m + \lceil \log_2 m \rceil + 2$ |
|               | Fast     | $m + \lceil \log_2 m \rceil + 2$ | $m + \lceil \log_2 m \rceil + 2$ |
| Encoder       | -        | m             | -              |
| D Latch       | Classic  | 5             | 13             |
|               | Fast     | 3             | 7              |
| Memory + CSS  | Classic  | $2r + c + 5rc + \lceil \log_2 (r + 1) \rceil + 4$ | $r + 3c + 13rc + 2(2r + 5)\lceil \log_2 (r + 1) \rceil + 3$ |
|               | Fast     | $r + c + 3rc + \lceil \log_2 (r + 1) \rceil + 3$ | $2r + 3c + 11rc + (r + 4)\lceil \log_2 (r + 1) \rceil + 4$ |

TABLE VI

| Block         | AND type | Latency (ms) |
|---------------|----------|--------------|
| Decoder       | Classic  | 3            |
|               | Fast     | 2            |
| Encoder       | -        | 1            |
| D Latch       | Classic  | 3            |
|               | Fast     | 2            |
| Memory        | Classic  | 6            |
|               | Fast     | 4            |

Fig. 8. Trace of the D Latch test.

This test verified the correct operation of multiple D latches. Fig. 8 shows the results in a trace, in which the information can be better visualized than in a graph. In each cell of each row of the trace a “1” is displayed when a spike was fired from the corresponding block at the time, in milliseconds, indicated by the top row. In this case, classic AND gates and external NOT gates were used.

In this experiment, two data bit signals were used, through which spikes were fired with no specific pattern to prove all possible combinations of binary values. Their associated negated values appear as NOT gate responses in the trace, colored in red and delayed by 1 ms from the spike time. In the same way, there were input spikes received at different timesteps through the store signal. Data signal 1 is associated with the first 3 latches (0, 1 and 2), which were set or reset simultaneously. The same happens with data signal 2, which is associated with latches 3, 4 and 5. Note that there was a delay between the spikes in the data signals and the spikes in the associated latches, which was equal to 4 ms. This value is the result of the sum of the delays of these latches, which in this case was 3 ms since classic AND gates are used.

B. D Latch

The latency for each of the presented blocks is shown in Table VI. The latency for the D Latch is 4 ms, which is the same as the latency for the latches themselves. This is consistent with the results shown in the trace, where the spikes in the D Latch are delayed by 4 ms from the spikes in the data signals.

Fig. 7. Graphs showing the results of the combined decoder/encoder test.

to an encoder. The encoder outputs were expected to represent the binary values used as input to the decoder, since, as it was explained, the encoder performs the inverse function of the decoder.

Fig. 7 shows the results of this test. As it was expected, increasing the binary input values allowed ascending channel selection at the decoder. Moreover, the encoder outputs were almost the same as the decoder inputs, with only one difference: they were delayed in time. This effect also occurred in the decoder, but with a different latency. However, this latency was the same for each of their output channels, for both encoder and decoder. As with digital circuits, these latencies are inherent to the spiking designs, with the total output latency being the sum of the delays of all existing synapses in the output path.
gates were used, plus the latency of NOT gates, 1 ms, since the data connections were delayed to coincide in time with the data connections in the AND gates. This latency depends on the AND type used and was as expected based on Table VI.

The first time that latches 0, 1 and 2 were set to 1 is associated with the first spike of the store signal. This spike at \( t = 1 \) sets latches 0, 1 and 2 at \( t = 5 \). The spike at \( t = 2 \) resets them at \( t = 6 \), since there were no input spikes received through data connections. In \( t = 3 \), the order to set all the latches was given, which occurred at \( t = 7 \). In \( t = 4 \) and \( t = 5 \) no spikes were received through the store signal, thus input spikes in the data connection were ignored. The rest of the cases are very similar to these. Input spikes received through data connections at \( t = 7 \) and \( t = 9 \) were ignored due to the absence of a spike in the store signal at these times. At \( t = 8 \), the value “0” had to be stored, similar to what happened at \( t = 2 \). Consequently, at \( t = 12 \), all the latches were reset.

C. Memory

The experiment conducted to test the memory was carried out on the block presented in Fig. 6, although using fast AND gates, which means that all AND gates were also connected to the CSS block. This test consisted in performing an ascending count through the “activation” or “deactivation” (presence or absence of spikes) of the input signals, and its results are shown in Fig. 9.

The signals colored in green and named “Signal” correspond to the decoder’s control signals, which indicated the register in which the input should be stored at the moment in which said selection occurred.

The two rows labeled as Channel correspond to the calculation of the selected channel. In these rows, empty cells correspond to the value 0, which was ignored to show only the channels of interest. Channel (Expected) shows the selected channel based on the control signal values, while Channel (Decoder) empirically shows the decoder channel that was activated. Note that there was a delay between the expected channel and the channel that was actually selected, which is the decoder latency. Thus, the delay between the empirical activation of the decoder output channel and the writing of the register is the latch latency.

In Fig. 9, each red rectangle on a vertical set of data bits encompasses the binary representation of the count number and is associated with a red rectangle in the register part, at the bottom of the trace, with a delay of 4 ms, which is the latency of this memory block using fast AND gates. Notice that rectangles in the register part represent the count number in hexadecimal format. It is important to remember that each register has 3 bits, coinciding with the number of data bits.

At \( t = 1 \), \( t = 2 \) and \( t = 3 \), the values to be written into the register associated with the output channels 1, 2 and 3 of the decoder are \( 0 \times 01, 0 \times 02 \) and \( 0 \times 03 \), respectively, which were stored at \( t = 5 \), \( t = 6 \) and \( t = 7 \), due to the decoder’s latency. At \( t = 4 \), no control signals were provided, and thus, at \( t = 8 \), no store operation was performed. This is because the output channel 0 of the decoder was selected, which does not have any associated register in the memory. In this experiment, the ascending count reached its maximum value at \( t = 7 \), and was reset to 0 in the next timestep. Thus, the trace is repeated in a loop until the end of the simulation was reached.

VII. DISCUSSION

Associative memories mentioned in Section I are characterized by the high correlation between the sources of information, being able to recover all the information from only a part of it. This is very interesting, since it could reduce the computational cost of certain powerful and complex applications, especially in tasks where it is necessary to find where the information is contained in the memory.

However, there are some points to be discussed. In [26], new connections are dynamically generated by training to define the behavior of the neural network and, in addition, it uses the STDP paradigm to modify their weights also by training. Thus, its computational cost should be higher and its performance, but also the number of resources required, should be lower than in the memory block presented in this work. Similar works mentioned in Section I also use Hebb’s rule to modify their weights, which means that training is necessary and therefore these comments also apply.

This work focuses on the implementation of the spiking memory block. During the development process, the implementation of spiking D latches was proposed, which allow absolute control over memory operations thanks to the asynchronous indication of the time at which the storage operation should be performed. This is very useful in the spiking design of Finite State Machines (FSMs), whose digital circuits are composed of combinational logic for transitioning between states and these latches to store their current state. FSMs are usually used to design any sequence of actions and can be used to build any hardware system with specific functionality. It should be noted that the development of spiking FSMs would lead to a valid alternative for the development of almost any spiking system. Thus, the possibilities of blocks that could be implemented would also be infinite, as in the case of digital circuits.

The implementation of FSMs is especially useful in the field of embedded systems, where applications such as IoT, elevators, ATMs, traffic lights, household appliances, etc. are often discussed. Thus, bringing the advantages of SNNs to these sectors, which are so relevant and present in society, would be another important advance.

Some possible improvements could be made over the proposed blocks. One of them would be the introduction of additional AND gates and signals to perform read operations in...
the memory. Currently, as was explained in Section V-D, the memory block only allows write operations. Output spikes are retained in its D latches, which are not currently expected to be connected to other neurons. In a practical application, it would be necessary to connect these latches to make proper use of the output spikes.

Furthermore, it would be interesting to reduce the amount of resources required to build the blocks presented in this work. Thus, one improvement would be the elimination of the non-operation cases in decoder and encoder. There are two types of non-operation cases: when there are no input spikes and output spikes are generated, and when there are input spikes but they are ignored, which is the opposite case. The truth tables of these components contemplate non-operation cases, which could be eliminated in order to remove their associated neurons and connections.

The case in which output spikes are generated without receiving input spikes is related to the use of NOT gates and, moreover, the use of CSS block. The basic principle of operation of the latter block may be counterintuitive as it goes against the essence of SNNs. However, as explained throughout this work, this block is necessary to generate spikes from nothing, which is the behavior of NOT gate. Although it could present an energy problem in biologically-plausible systems, it should be studied in more depth since there is evidence of recurrent neural networks in biology [41]. A possible solution to this problem, from an entirely biological aspect, could be the implementation of a memory refresh mechanism to compensate for the loss of energy that could occur over time. It would be interesting in this case to study similar mechanisms that might exist in biology. Note that in a digital neuromorphic platform, such as SpiNNaker, this problem does not exist since it allows to implement self-excited neurons without providing an external energy supply.

An aspect that was taken into account when carrying out the implementation of the proposed blocks is that, since SpiNNaker only allows working with integer timesteps and ideal synapses, there should not be any unexpected deviation in the arrival time of input spikes. Thus, the problem of matching input spikes to perform certain operations, which is one of the weak points of asynchronous circuits, is inherently solved. This problem could be especially relevant for spiking AND gates, in which the precise timing of the spikes is necessary to produce the correct output.

Notice that outside the SpiNNaker platform, this spike synchronization problem should be studied in depth, since it is intended for the designs to be as close as possible to the behavior of spikes and neurons in the nervous system. Said synchronization could be achieved in two different ways: the use of spike trains, which would surely imply changes in the presented designs, or the adjustment of the parameters used for the neurons to diminish the effect of small temporary variations in the arrival times of the spikes. This second option seems to be the most reasonable, since it would involve few changes and could help to focus the synchronization problem in a low-level approach.

Finally, the spiking memory implementation presented in this work could have a large number of direct applications, since its digital counterpart is essential in any computer, as was explained in Sections I and II. In fact, it allows thinking about the implementation of a spiking computer, which would be a great advance since it would be one of the highest-level spiking blocks implemented so far and would have the advantages of spiking blocks, which are mainly low power consumption and real-time capability.

VIII. CONCLUSION

This work focuses on the spiking implementation of a memory block, for which other spiking functional blocks such as decoder, encoder and D Latch were developed. The behavior of these blocks was proved by conducting different tests, the results of which are shown to validate their performance.

In addition, the resources required to build each block presented in this paper were studied, showing different mathematical formulae that allow to quickly and easily obtain the number of neurons and synapses used in the implementation process and that can be very useful for neuromorphic engineers.

A thorough comparison between other SNN-based memories in the state of the art and the one proposed here was also carried out, highlighting the main advantages and disadvantages and validating the proposed design.

All spiking functional blocks presented in this work are available in sPyBlocks; this is a public repository that could be very useful for neuromorphic engineers when completing neuromorphic applications, which may require certain complex functionalities and for which there was no SNN-based solution.

On the other hand, some important points that are still open, such as the energy problem of CSS, and the future work to be done using the tools provided, such as different improvements that could be performed over the proposed blocks to reduce their requirements or the future implementation of spiking FSMs and fully spiking computer, were also commented in Section VII. The spiking memory implementation presented in this work paves the road for the development of the last two blocks mentioned above.

REFERENCES

[1] C. Mead, “Neuromorphic electronic systems,” Proc. IEEE, vol. 78, no. 10, pp. 1629–1636, Oct. 1990.
[2] S. Soman et al., “Recent trends in neuromorphic engineering,” Big Data Analytics, vol. 1, no. 1, pp. 1–19, 2016.
[3] M.-O. Gewaltig and M. Diesmann, “Nest (neural simulation tool),” Scholarchipedia, vol. 2, no. 4, 2007, Art. no. 1430.
[4] D. F. Goodman and R. Brette, “Brian: A simulator for spiking neural networks in Python,” Front. Neuroinform., vol. 2, 2008, Art. no. 5.
[5] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana, “The SpiNNaker project,” Proc. IEEE, vol. 102, no. 5, pp. 652–665, May 2014.
[6] M. Davies et al., “Loihi: A neuromorphic manycore processor with on-chip learning,” IEEE Micro, vol. 38, no. 1, pp. 82–99, Jan./Feb. 2018.
[7] F. Akopyan et al., “TrueNorth: Design and tool flow of a 65 mW 1 million neuron programmable neurosynaptic chip,” IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol. 34, no. 10, pp. 1537–1557, Oct. 2015.
[8] C. Peihle et al., “The BrainScaleS-2 accelerated neuromorphic system with hybrid plasticity,” Front. Neurosci., vol. 16, 2022, Art. no. 795876. [Online]. Available: https://www.frontiersin.org/article/10.3389/fnins.2022.795876
[9] S. Furber, “Large-scale neuromorphic computing systems,” J. Neural Eng., vol. 13, no. 5, 2016, Art. no. 051001.
D. R. Valeiras, G. Orchard, S.-H. Leng, and R. B. Benosman, “Neuromorphic event-based 3D pose estimation,” Front. Neurosci., vol. 9, 2016, Art. no. 522.

P. U. Diehl, B. U. Pedroni, A. Cassidy, P. Merolla, E. Nefci, and G. Z. Gu, “SpikeTNT: Neuromorphic emotion recognition on truenorth,” in Proc. IEEE Int. Joint Conf. Neural Netw., 2016, pp. 4278–4285.

J. P. Domínguez-Morales et al., “Deep spiking neural network model for time-variant signals classification: A real-time speech recognition approach,” in Proc. IEEE Int. Joint Conf. Neural Netw., 2018, pp. 1–8.

J. L. Wu, E. Yang, X. Zhang, H. Li, and K. C. Tan, “Deep spiking neural networks for large vocabulary asynchronous speech recognition,” Front. Neurosci., vol. 14, 2020, Art. no. 199.

F. Corradi, G. Adriaans, and S. Stuijk, “Gyro: A digital spiking neural network architecture for multi-sensory data analytics,” in Proc. Drone Syst. Eng. Rapid Simul. Perform. Eval.: Methods Tools, 2021, pp. 9–15.

T. Schoeppe, D. Gutierrez-Galan, J. P. Domínguez-Morales, A. Jimenez-Fernandez, A. Linares-Barranco, and E. Chicca, “Live demonstration: Neuromorphic sensory integration for combining source sound localization and collision avoidance,” in Proc. IEEE Int. Symp. Circuits Syst., 2020, pp. 1–1.

F. Perez-Peña et al., “Neuro-inspired spike-based motion: From dynamic vision sensor to robot motor open-loop control through spike-vote,” Sensors, vol. 13, no. 11, pp. 15805–15812, 2013.

A. Jimenez-Fernandez, G. Jimenez-Moreno, A. Linares-Barranco, M. J. Domínguez-Morales, R. Paz-Vicente, and A. Civit-Balecites, “A neuro-inspired spike-based PID motor controller for multi-motor robots with low cost FPGAs,” Sensors, vol. 12, no. 4, pp. 3851–3856, 2012.

D. Gutierrez-Galan, J. P. Domínguez-Morales, F. Perez-Peña, A. Jimenez-Fernandez, and A. Linares-Barranco, “NeuroPod: A real-time neuromorphic spiking CPG applied to robotics,” Neurocomputing, vol. 381, pp. 10–19, 2020.

P. Batres-Mendoza, E. I. Guerra-Hernandez, A. Espinal, E. Pérez-Careta, and H. Rostro-Gonzalez, “Biologically-inspired legged robot locomotion controlled with a BCI by means of cognitive monitoring,” IEEE Access, vol. 9, pp. 35765–35777, 2021.

A. Ayuso-Martínez, D. Casanueva-Morato, J. P. Domínguez-Morales, A. Jimenez-Fernandez, and G. Jimenez-Moreno, “Spiking-based building blocks for performing logic operations using spiking neural networks on SpiNNaker,” in Proc. IEEE Int. Joint Conf. Neural Netw., 2022, pp. 1–9.

G. Neves, S. F. Cooke, and T. V. Bliss, “Synaptic plasticity, memory and the hippocampus: A neural network approach to causality,” Nature Rev. Neurosci., vol. 9, no. 1, pp. 65–75, 2008.

J. Lisman, G. Buzsáki, H. Eichenbaum, L. Nadel, C. Ranganath, and A. Yariv, “The hippocampus: A neural network approach to causality,” Nature Rev. Neurosci., vol. 6, no. 6, pp. 115–142, 2015.

N. Caporale and Y. Dan, “Spike timing–dependent plasticity: A Hebbian learning rule,” Annu. Rev. Neurosci., vol. 31, pp. 25–46, 2008.

H. He et al., “Constructing an associative memory system using spiking neural network,” Front. Neurosci., vol. 13, 2019, Art. no. 650.

D. O. Hebb, The Organization of Behavior: A Neuropsychological Theory. London, U.K.: Psychology Press, 2005.

C. H. Tan et al., “Associative memory model of hippocampus CA3 using spike response neurons,” in Proc. Int. Conf. Neural Inf. Process., Springer, 2011, pp. 493–500.

C. H. Tan, H. Tang, K. C. Tan, and M. Yuan, “A hippocampus CA3 spiking neural network model for storage and retrieval of sequential memory,” in Proc. IEEE Conf. Cybern. Intell. Syst., 2013, pp. 134–139.

J. P. Domínguez-Morales, A. Ayuso-Martínez, J. P. Domínguez-Morales, A. Jimenez-Fernandez, and G. Jimenez-Moreno, “Spike-based computational models of bio-inspired memories in the hippocampal CA3 region on SpiNNaker,” 2022, arXiv:2205.04702.

R. Prasad, Analog and Digital Electronic Circuits: Fundamentals, Analysis, and Applications. Berlin, Germany: Springer, 2021.

A. K. Mishra, D. P. Acharya, and P. K. Patra, “Novel design technique of address decoder for SRAM,” in Proc. IEEE Int. Conf. Adv. Commun. Control Comput. Technol., 2014, pp. 1032–1035.

S. Nandakumar, S. R. Kulkarni, A. V. Vichare, and B. Rajendran, “Building brain-inspired computing systems: Examining the role of nanoscale devices,” IEEE Nanotechnol. Mag., vol. 12, no. 3, pp. 19–35, Sep. 2018.

S. Ghosh-Dastidar and H. Adeli, “Spiking neural networks,” Int. J. Neural Syst., vol. 19, no. 04, pp. 295–308, 2009.

S. Davidson and S. B. Furber, “Comparison of artificial and spiking neural networks on digital hardware,” Front. Neurosci., vol. 15, 2021, Art. no. 345.

J. L. Lobo, J. Del Ser, A. Bifet, and N. Kozak, “Spiking neural networks and online learning: An overview and perspectives,” Neural Netw., vol. 121, pp. 88–100, 2020.

A. G. Rowley et al., “SpiNNTools: The execution engine for the SpiNNaker platform,” Front. Neurosci., vol. 13, 2019, Art. no. 231.

A. P. Davison et al., “PyNN: A common interface for neuronal network simulators,” Front. Neuroinformat., vol. 2, 2009, Art. no. 011.

M. L. Hines and N. T. Carnevale, “The neuron simulation environment,” Neural Computation, vol. 9, no. 6, pp. 1179–1209, 1997.

O. Rhodes et al., “sPyNNaker: A software package for running PyNN simulations on SpiNNaker,” Front. Neurosci., vol. 12, 2018, Art. no. 816.

R. J. Douglas and K. A. Martin, “Recurrent neuronal circuits in the neocortex,” Curr. Biol., vol. 17, no. 13, pp. R496–R500, 2007.
Angel Jimenez-Fernandez (Member, IEEE) received the BS degree in computer engineering, the MS degree in industrial computer science, and the PhD degree in neuromorphic engineering from the University of Seville, Seville, Spain, in 2005, 2007, and 2010. In October 2007, he became an assistant professor with the Department of Computer Architecture and Technology, University of Seville. His research interests include neuromorphic engineering applied to robotics, real-time spikes signal processing, neuromorphic sensors, FPGA design, embedded intelligent systems development, and biomedical robotics.

Gabriel Jimenez-Moreno (Member, IEEE) received the MS degree in physics (electronics) and the PhD degree from the University of Seville, Seville, Spain, in 1987 and 1992, respectively. He was granted a fellowship from the Spanish Science and Technology Commission (CICYT). Currently, he is a full professor with the University of Seville. From 1996 until 1998, he was vice-dean of the E.T.S. Ingenieria Informatica, University of Seville, where he participated in the creation of the Department of Computer Architecture. He is the author of several papers on robotics, neuromorphic engineering, and computer architecture, and has also directed many research projects on neuromorphic systems. His research interests include neural networks, vision processing systems, embedded systems, computer interfaces, and computer architectures.