INTEGRATION, the VLSI journal 54 (2016) 109–117

Contents lists available at ScienceDirect

INTEGRATION, the VLSI journal

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

Invited paper

# Digital implementation of a virtual insect trained by spike-timing dependent plasticity ☆

P. Mazumder [a,*], D. Hu [a], I. Ebong [a], X. Zhang [b], Z. Xu [b], S. Ferrari [b]

[a] University of Michigan, Ann Arbor, MI 48109, USA
[b] Duke University, Durham, NC 27708, USA

ARTICLE INFO

*Article history:*
Received 25 June 2015
Accepted 18 January 2016
Available online 13 February 2016

*Keywords:*
Spike timing dependent plasticity
Neural network

ABSTRACT

Neural network approach to processing have been shown successful and efficient in numerous real world applications. The most successful of this approach are implemented in software but in order to achieve real-time processing similar to that of biological neural networks, hardware implementations of these networks need to be continually improved. This work presents a spiking neural network (SNN) implemented in digital CMOS. The SNN is constructed based on an indirect training algorithm that utilizes spike-timing dependent plasticity (STDP). The SNN is validated by using its outputs to control the motion of a virtual insect. The indirect training algorithm is used to train the SNN to navigate through a terrain with obstacles. The indirect approach is more appropriate for nanoscale CMOS implementation synaptic training since it is getting more difficult to perfectly control matching in CMOS circuits.



## 1. Introduction

The neural network approach to data processing has undergone continued research and development even with the widespread success of the von Neumann architecture, traditionally sequential in nature. Recent widespread advancement of the von Neumann architecture to utilize multi-core processors [2] is similar to the neural network approach, providing a much needed boost to the area. The difference between the parallelism of multi-core processors and that of neural networks is the latter uses much less complex processing elements, therefore, allowing opportunities for massively parallel structures. Hardware neural networks or neuromorphic circuits have been around for quite some time with proposals that span both digital CMOS and analog CMOS approaches [3–6]. Specific VLSI reviews and methodologies are provided in [7,8]. Although neural hardware have been proposed, implemented, and commercialized, their widespread adoption is still unrealized. Neural software implementations running on digital computers are much more prevalent, leaving hardware adoption behind. Hardware implementations have found niche uses in peripheral devices and various subsystems [3]. Software implementations have the advantage of ease of programming through well-known languages like C and C++, large software engineering support due to a lower barrier of entry for software engineers than those for hardware engineers, high precision calculations if the processing capabilities are present, and more flexibility regarding the implemented algorithm. Even with these advantages, hardware implementations are still sought after because of the speed associated with hardware computing; realization of adequate neural processors or neurocomputers will enable applications that require real-time processing, feedback, and learning. The most promising implementations use the digital components and have granted programmable neurocomputers like CNAPS [9] and SYNAPSE-1 [10]. Although neurocomputers are very powerful, efforts have been made to scale down applications to even less powerful machines, ones that run on battery and do not rely on a large number of processing elements. These efforts have led to the widespread appeal of spiking neural networks [4,11–15].

Spiking neural network (SNN) implementations provide a powerful computation fabric where a smaller number of processing elements can potentially be utilized in order to realize desired functionality. When working with SNN implementations, usually the designer provides different, specified goals during design phase. The goals determine the level of detail needed when designing the neuron and synapse behavior. Hardware neural networks have been used to study different phenomena in biological neural networks. This has brought about different neuron models with their hardware implementations. Hardware neural networks seek to be simple in functionality in order to minimize

☆This work was supported by the National Science Foundation under ECCS Grant 1059177 and CCF Grant 1421467.

* Corresponding author.
*E-mail addresses:* pinakimazum@gmail.com (P. Mazumder), hudi@umich.edu (D. Hu), idong@eecs.umich.edu (I. Ebong), xz70@duke.edu (X. Zhang), dec.ziyer@gmail.com (Z. Xu), sferrari@duke.edu (S. Ferrari).

http://dx.doi.org/10.1016/j.vlsi.2016.01.002
0167-9260/

the area associated with this processing element. This brings about the massive tradeoff between complex models like the Hodgkin–Huxley and the leaky integrate and fire (LIF) [15]. Since SNN solutions require numerical solutions with no closed form representations, their behavior in hardware is much harder to predict. Therefore, learning algorithms tenable to hardware adoption are crucial to pushing widespread SNN adoption. Software SNN implementations are widespread; hardware implementations need to catch up, hence the thrust behind this work. The contributions in this paper are: abstraction and mapping of a complex learning process to a digital spiking neural network fabric. A digital approach is used in order to encourage repeatability when dealing with complex SNNs. A virtual bug example is used to illustrate the strength of this algorithm. Section 2 will provide details on the model of the virtual insect. Section 3 will expand on the specific example by showing top level neural network organization, the training algorithm, and the CMOS circuit adaptation. Section 4 will provide simulation results and discussion, and Section 5 relays some concluding remarks.

## 2. Virtual insect model

The test setup and scenario chosen is a virtual insect (bug) model. The virtual insect model is constructed to demonstrate and evaluate the indirect training algorithm [1] and hardware-level rapid prototyping design. Offline training with limited information is adopted for the chosen application. After training the virtual insect, the virtual insect is used in a homing application where it is used to find a given target on a two-dimensional space with obstacles.

The virtual insect is a moniker based on the given construct in Fig. 1, since the sensors are attached to the body like antennae on a biological insect. The virtual insect is modeled as a rigid object that can move in any direction on a map. Fig. 1 shows the external structure and environment of the virtual insect. The environment of the virtual bug consists of obstacles which are denoted as black objects and a target which is denoted as a bright spot on the map. The virtual bug has four sensors which provide terrain and target information. The bug has an elliptical shape and is symmetric along its major axis. On each symmetric half, a target sensor, a terrain sensor, and a motor is modeled. By convention, the labels for these sensors and motors are either “Left” or “Right,” depending on which half they reside as depicted in Fig. 1.

The target sensor generates a signal $S_{target}$ based on the distance between the sensor and the target. By convention, the further the virtual insect is from the target, the higher the magnitude or intensity of $S_{target}$. The terrain sensor generates a signal $S_{terrain}$ based on the roughness of the map, with a rougher map corresponding to a more intense or higher magnitude of $S_{terrain}$. The two

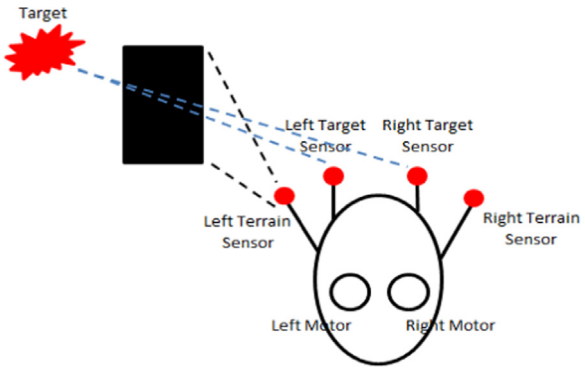


**Fig. 1.** External structure of the virtual insect.

motors effect the direct motion of the insect. Intuitively, if the left motor has a higher revolutions per minute (rpm) compared to the right motor, the insect will turn right. The insect turns left if the right motor rotates faster than the left motor. If the two motors have the same rpm, then the virtual insect will move forward in its direction of orientation. The motion of the virtual insect is restricted in that its motors are allowed to rotate in only one direction. Therefore, the insect is incapable of reversing (moving opposite its oriented direction). If the insect needs to follow the direction opposite its direction of orientation, it will need to turn towards that direction then move forward.

Since virtual insect motion is determined by both the relative angular velocities of the two motors and the current sensor inputs regarding proximity to obstacles and the target, the internal connection between the sensors and the motors is described.

When the insect moves in the prescribed environment, its motion can be described by (1), adapted from the modified unicycle robot locomotion in [16,'1]. By restricting the environment to a 2D Cartesian plane, when the insect moves, its linear velocity can be described as $v$. $v$ can be decomposed into components in both the $x$-direction and $y$-direction, denoted in (1) as $v_x$ and $v_y$, respectively. $v_{left}$ and $v_{right}$ are the speeds of the left and right motors, respectively. $\theta$ is the variable used to represent direction of orientation and is defined as the angle between the major axis of the elliptical insect and the $x$-axis. $L$, $\tau_{motor}$ and $\eta$ are scaling constants; and $t_L^f$ and $t_R^f$ are the firing times of output neurons (more on this later).

$$\begin{cases} v_x = v \times \cos(\theta) \\ v_y = v \times \sin(\theta) \\ v = \frac{v_{left} + v_{right}}{2} \\ \Delta\theta = \frac{v_{right} - v_{left}}{L} \\ \Delta v_{left} = -\frac{v_L}{\tau_{motor}} + \eta * \left(t = = t_L^f\right) \\ \Delta v_{right} = -\frac{v_R}{\tau_{motor}} + \eta * \left(t = = t_R^f\right) \end{cases} \quad (1)$$

According to (1), $\Delta v_{left}$ and $\Delta v_{right}$ are always negative and become positive only when $t = t_L^f$ or when $t = t_R^f$, respectively. This translates to a motor's rpm, $v_{left}$ or $v_{right}$, is always decreasing unless its corresponding output neuron spikes. Therefore, the more frequently an output neuron fires or spikes, the faster the speed of the corresponding motor. The next section will elucidate the connections between the output neurons and how its spiking events are controlled. Essentially, the dependence on the firing frequency of an output neuron on the synaptic weights of the neural network reduces the training of the virtual insect to weight parameter adjustment.

## 3. Spike-based training approach

### 3.1. Top level NN organization

In the previous discussion, the control of the motors was due to a spiking pattern of the outputs of some neural network. This section provides the top level architecture and inherent connectivity of the spiking neural network (SNN) controlling the motors. The SNN architecture (illustrated in Fig. 2) resembles a feedforward neural network with an input layer, a hidden layer and an output layer. The input layer interfaces with the four sensor inputs while the output layer interfaces with the two motors. The two output layers are shown as separate entities to make clear there is no interconnectivity between the two layers. Additionally, the structure of the SNN is flexible (i.e. each layer can have any number of neurons and can be of any shape, and the connections

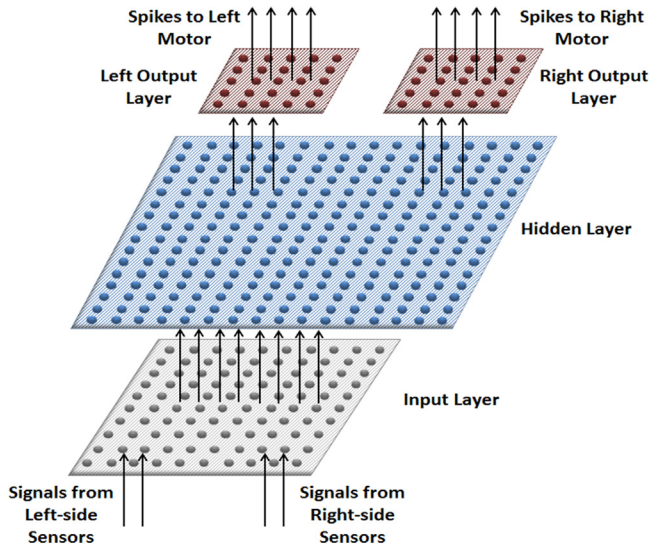


**Fig. 2.** The structure of the SNN showing information flow and layers.

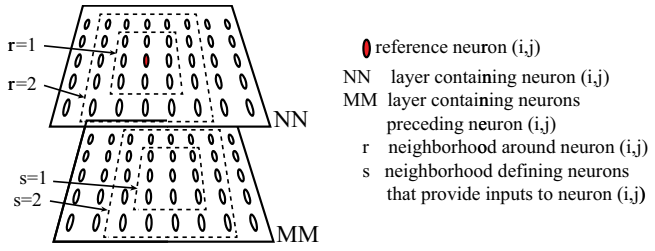


**Fig. 3.** Layer connection relaying the concept of neighborhoods for the large SNN design.

between neurons can be arbitrary) as long as the overall structure is kept. There needs to be an input layer, hidden layer and output layer as demonstrated in Fig. 2.

The SNN resembles a feedforward network but is not by definition feedforward due to the connections within layers, which we will dub the intra-layer connections. The parameters that describe this connectivity are: neighborhood connectivity and special treatment of neurons on the edge. The hardware implementation of this SNN with the prescribed connectivity is highly dependent on the resources available. In the case of a field programmable gate array (FPGA) implementation, the number of logic elements will limit the connection scheme and neighborhood. In the case of CMOS implementation, the area it takes to implement a synapse will limit the connectivity. The chosen connectivity for this work is discussed in the CMOS implementation subsection.

The neuron model adopted is a leaky-integrate and fire model [17] expressed in (2):

$$\tau_m \frac{dV_{ij}(t)}{dt} = -V_{ij}(t) + R_m \sum_{(k,l)\in NN_r(i,j)} a_{ij,kl} f(V_{kl}(t)) + R_m \sum_{(k,l)\in MM_s(i,j)} b_{ij,kl} f(V_{kl}(t)) + I_{stim} \quad (2)$$

where $V_{ij}(t)$ is the membrane potential or the internal state variable of the $ij$ neuron or processing element, and $R_m$ and $\tau_m$ are constants modeling membrane resistance and membrane passive time constant, respectively. $I_{stim}$ is used to model direct stimulation of the neuron; this is useful for training signal inputs. $f(V_{kl}(t))$ provides the firing state of neuron $kl$ and takes on the value of either 0 for a non-firing neuron or 1 for a firing neuron. $a_{ij,kl}$ and $b_{ij,kl}$ represent synaptic conductance for intra-layer pre-neurons

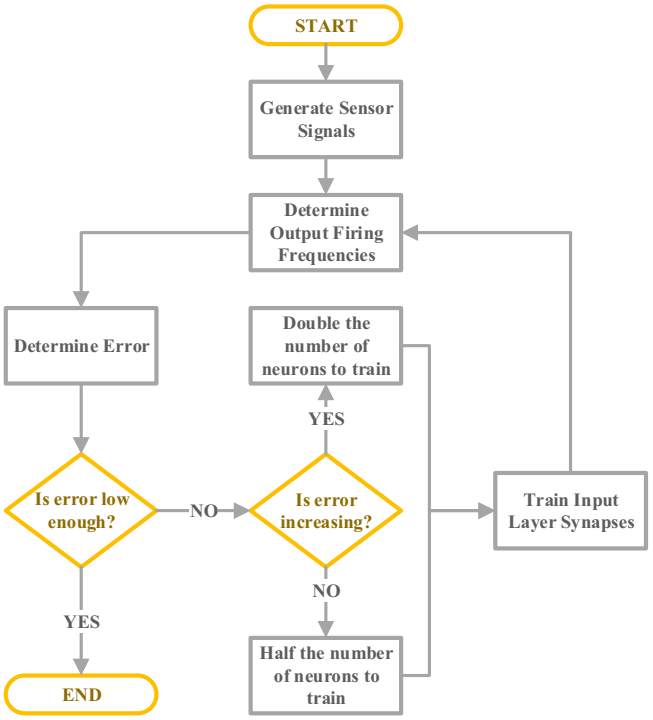


**Fig. 4.** Flowchart showing the different steps involved in the SNN training algorithm.

and inter-layer pre-neurons, respectively. $NN_r(i,j)$ and $MM_s(i,j)$ represent the intra-layer and inter-layer neighborhoods, respectively, of neuron $(i,j)$. $r=1$ or $s=1$ represents a neighborhood of 9 neurons. Fig. 3 provides a graphical depiction of the arrangement in relation to this definition. When $V_{ij}(t)$ reaches a predefined threshold due to its pre-synaptic neuron activity, neuron $(i,j)$ fires (Fig. 4).

Eq. (2) uses prescribed neighborhoods but the exact connections can actually differ randomly. For example, in [1], a probability density function was used to determine the connectivity. This approach is conducive in software implementation and rapid prototyping with FPGA but is not recommended for CMOS implementation. The CMOS implementation in this work utilizes different connection schemes to determine how the effect of connectivity influences the integrity and training effort of the solution. The synaptic conductances, $a_{ij,kl}$ and $b_{ij,kl}$, can either be fixed or varying. In this work, the synapses are modified using asymmetric spike timing dependent plasticity (STDP) [18–20]. The specific details of the implementation are divulged in the training methodology of the paper, presented next.

### 3.2. SNN training algorithm

#### 3.2.1. Flowchart

The training methodology chosen is limited and different from other methods due to the following ratiocination: stimulate the input and or probe the output. With this restriction, direct control of synapses is prohibited and therefore unconducive to setting conductances to specific values. STDP, therefore, is adopted to allow indirect control and training of synapses. The indirect training algorithm can be broken down into five steps. The different steps in the algorithm are further explicated:

Step 1.*Generate sensor signals*: Twelve sensor scenarios are provided in Table 1. For parameter adjustment, the first step is to generate terrain and sensor signals for the 12 possible cases. Essentially, provide values for $S_{terrainL}$, $S_{targetL}$, $S_{terrainR}$, and $S_{targetR}$. These essentially are the left terrain sensor input, the left target

**Table 1**
12 sensor signal cases.

| Case # | Left terrain | Right terrain | Left target | Right target |
|---|---|---|---|---|
| 1 | Plain | Plain | Strong | Weak |
| 2 | Plain | Rough | Strong | Weak |
| 3 | Rough | Plain | Strong | Weak |
| 4 | Rough | Rough | Strong | Weak |
| 5 | Plain | Plain | Equal | Equal |
| 6 | Plain | Rough | Equal | Equal |
| 7 | Rough | Plain | Equal | Equal |
| 8 | Rough | Rough | Equal | Equal |
| 9 | Plain | Plain | Weak | Strong |
| 10 | Plain | Rough | Weak | Strong |
| 11 | Rough | Plain | Weak | Strong |
| 12 | Rough | Rough | Weak | Strong |

sensor input, the right terrain sensor input and the right target sensor input, respectively.

Step 2. *Determine output firing frequencies*: For each case in Step 1, the desired output frequencies are calculated using the linear model in (3). In (3), $f_{desiredL}$ and $f_{desiredR}$ are the desired firing frequencies of both the left and right output neurons while $C_1$ and $C_2$ are constants. $C_1$ is set to be larger than $C_2$ to prioritize the terrain sensors.

$$\begin{pmatrix} f_{desiredL} \\ f_{desiredR} \end{pmatrix} = \begin{pmatrix} S_{terrainL} & S_{targetL} \\ S_{terrainR} & S_{targetR} \end{pmatrix} \begin{pmatrix} C_1 \\ C_2 \end{pmatrix} \tag{3}$$

Step 3. *Determine Error*: Neural network designs hinge on improving a certain performance metric. Only output neuron firing patterns can be probed, therefore, the performance metric is tied to the desired firing and observed firing frequencies. The error metric, provided in (4) as $e_t$, is essentially an average of the deviations from actual firing frequencies for all the 12 cases in Table 1. $L$ and $R$ designate the left and right output neurons, respectively, and $p$ designates the 12 cases.

$$e_t = \sum_{X \in L,R} \sum_{p=1}^{12} \frac{|f^p_{desiredX} - f^p_{actualX}|}{24} \tag{4}$$

Step 4. *Generate Training Signals*: After determining the value of $e_t$, it is compared with both its previous value in the previous training epoch and an upper-bound error value. In the case, we are not under the upper-bound error value, the error value is compared with its previous value to determine if error is decreasing or increasing. If decreasing, then the training signal directions are kept the same. If increasing, then the training signals are reversed.

Step 5. *Train input layer intra-layer synapses*: This step is optional and only performed when the error value obtained is not enough to reach the stop condition. If stop condition is not met, then the training signals generated in Step 4 are used to train random input neuron pairs. The training that occurs in each training epoch utilizes the modified model in (5) to determine how many neuron pairs ($N^r$) to train. When the error metric $e_t$ decreases, the number of pairs trained will double compared to the previous training epoch. On the other hand, if $e_t$ increases, $N^r$ will be halved. There are limits to the growth of $N^r$ placed on the hardware design by endorsing maximum and minimum values, thereby discouraging training that does not converge.

$$N^{r+1} = \begin{cases} 2 * N^r, & \Delta e(t_l) < 0 \\ 0.5 * N^r, & \Delta e(t_l) > 0 \end{cases} \tag{5}$$

### 3.2.2. Spike timing dependent plasticity

Steps 1 through 5 list the broader activities that occur in a training epoch. This subsection will provide more detail pertaining to steps 4 and 5. Only one neuron pair receives training inputs during a training epoch. The objective of the neuron pair stimulation is to modify the synapse between them, and this modification is achieved through STDP. STDP relates synapse conductance change to the spike time between the neurons sharing the synaptic connection. Defining the spike time difference between two neurons as $D_{ij,kl}$, the value for the value for the next $D_{ij,kl}$ can be obtained for each training epoch.

In Step 4, given a certain $D_{ij,kl}$, training signals are generated. In Step 5, the training signals are applied to an input neuron pair. The $D_{ij,kl}$ for input layer neurons can be controlled quite readily by modulating the time difference for the application of the training signals. Also within Step 4, the $D_{ij,kl}$ for the next training epoch is dependent on the change in the error value. This signifies, $D_{ij,kl}$ is changed from positive to negative and vice versa if the error value is moving in a non-intended direction. The synaptic weight difference corresponding to a certain $D_{ij,kl}$ is defined as $\Delta w_{ij,kl}$ in (6):

$$\Delta w_{ij,kl} = \begin{cases} A_+ \cdot e^{-\frac{D_{ij,kl}}{\tau_+}}, & D_{ij,kl} > 0 \\ -A_- \cdot e^{\frac{D_{ij,kl}}{\tau_-}}, & D_{ij,kl} < 0 \end{cases} \tag{6}$$

where $w_{ij,kl}$ is the synaptic weight between neuron $(i,j)$ and neuron $(k,l)$ neurons $A_+$ and $A_-$ are constants determining the maximum increase or maximum decrease in weight for each pair of pre- and post-synaptic spikes and $\tau_+$ and $\tau_-$ are time constants.

## 3.3. CMOS circuit adaptation

With the virtual insect application explained and the training method for the SNN within the insect model understood, the idea of rapid prototyping and generation of CMOS circuits in hardware is discussed. The indirect training algorithm lends itself to evaluation in three formats: a small FPGA implementation, a small digital CMOS SNN, and a large digital CMOS SNN. Both CMOS implementations utilized 130-nm process technology. The small designs consisted of 7 neurons while the large design housed 406 neurons. In addition to SNN size difference, the small and large designs had varying connectivity and different synapse constraints.

Fig. 5 shows the intra-layer and inter-layer connection for the large SNN design. Since the small design has fewer neurons, the number of synapses is drastically reduced and hence there is flexibility to use complex STDP synapses for all synapses. The large design, on the other hand, contains many neurons with the following specifications for the modification of (2): the decay factor $-V_{ij}(t)$ is set to 0, $r=1$, and $s=1$. Additionally, within the different layers, the nature of $a_{ij,kl}$ differs – $a_{ij,kl}$ follows STDP modification within the input layer. Within the hidden and output layers on the other hand, $a_{ij,kl}$ has fixed weights. This modification was made in order to: (a) deal with the drawback of large area STDP synapses and (b) reduce the training time since simulation takes too long with all synapses changing values with respect to spikes. The small designs, especially the FPGA implementation, are used as verification engines for the training methodology. Unfortunately, we were not in the possession of a larger FPGA to display the prowess of the training algorithm but the small design suffices.

In addition to the simplification of synaptic connections, some approximations had to be made for hardware implementation. The following list provides the key approximations made:

1. The current implementation was a focus on the SNN design, therefore, interface circuits for the mechanical components of

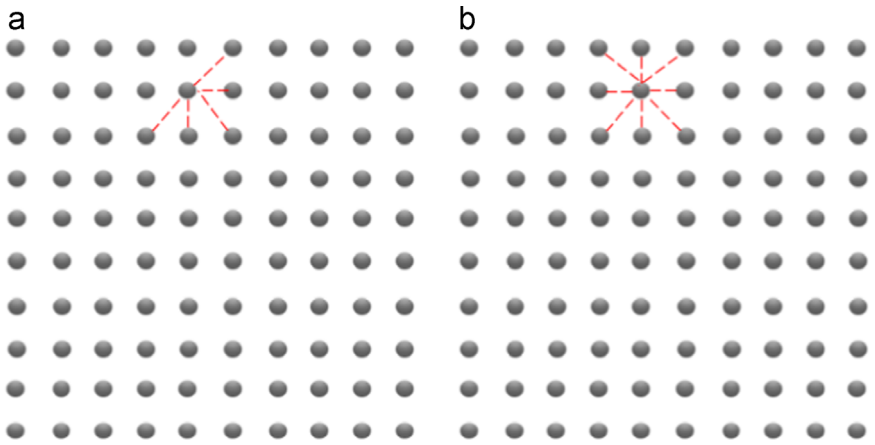


**Fig. 5.** Large SNN design (left) intra-layer connection (right) inter-layer connection.

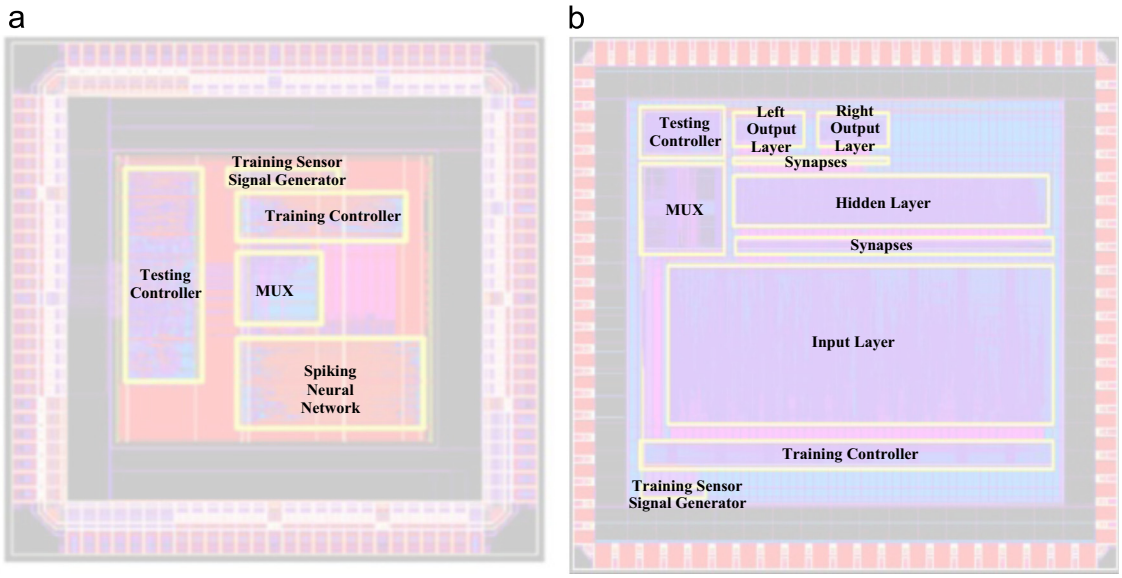


**Fig. 6.** Floorplan of both designs showing high level modules and connections between these modules (a) small design (b) large design.

the insect was not designed. In order to properly translate the requirements of (1) to obtain positional information, the dynamics of (1) was implemented in hardware. Lookup tables (LUTs) were used to implement the trigonometric functions, therefore, tying the performance of the virtual insect to the resolution of this LUT.

2. In order to implement STDP, multiplication to the exponential function is necessary. Base *e* was replaced with base 2 for hardware simplicity, making multiplication translate directly to shifting.
3. The terrain signal adopts a binary representation, either zero or one, in order to reduce the complexity of calculation. Zero indicates the absence of an obstacle while one signals the presence of an obstacle. The idea of terrain roughness is therefore moot and is left for future work.
4. Due to the digital CMOS implementation, a synchronous SNN was more conducive to training. Therefore, the neuron spike signals occurred at the positive edge of the clock signal. This inadvertently quantized the time axis of the STDP curves into factors of the clock period. The positive to this adoption is that the time difference between two spikes is precisely calculated, therefore, minimizing the effects of quantization.
5. Pseudo-random number generator for selecting which neurons to train is implemented with a 32-bit linear feedback shift register to ensure no repeated training pattern occurs.

The decisions and simplifications outlined would be very different for a mechanical insect – the interface design for motors and sensors would have to be part of the design in order for the insect to observe and react to environmental stimulus. By emulating the functions of these devices in circuit, it impacts the functionality and space used on-chip for the SNN design. For example, the testing circuitry is implemented alongside the SNN, separate from the training circuitry. The testing circuitry utilizes the virtual insect's positional information to generate sensor signal inputs to the SNN. The generated sensor signals then cause output spike sequences which are then used to calculate the positional information for the next period based on the relationship in (1). There is room to improve on resource and design effort consumption of the testing controller when moving from a virtual to a mechanical insect, hence, the resource impact of this block should not be a focal point. With the approximations made and the functionality of the testing and training circuits discussed, Fig. 6 shows the floorplan of both the small and large designs. They both

consist of a training controller, a testing controller, a mux to select between training signals and testing signals, a training signal generator and the SNN.

Fig. 6 also shows the dataflow of each design. The biggest difference between Fig. 6a and b is peripheral circuitry needed to handle the differing network sizes. Fig. 6a has a block for the SNN which contains 7 neurons and 8 synapses while Fig. 6b needed to be broken apart in order to handle the wire connectivity. The floorplans correspond directly to the layouts so those are not included in this article. In layout pictures, each major block is circled and corresponds to a block shown in the corresponding floor plan. Table 2 breaks down the layout area of each major component defined Fig. 6. The full chip area without pads for each design is provided as well. With pads included, the small design has an area of 2.8 mm by 2.8 mm while the larger design has an area of 6.5 mm by 4.8 mm.

**Table 2**
Layout area of both CMOS designs.

| Block | SMALL Area | Block | LARGE Area |
|---|---|---|---|
| SNN | 800 μm × 400 μm | Input layer<br>Hidden layer<br>Output layer | 4600 μm × 1600 μm<br>3750 μm × 500 μm<br>800 μm × 200 μm × 2 |
| Training signal generator | 450 μm × 20 μm | Training signal generator | 710 μm × 36 μm |
| Testing controller | 300 μm × 980 μm | Testing controller | 900 μm × 400 μm |
| Training controller | 720 μm × 160 μm | Training controller | 4900 μm × 200 μm |
| Control signal mux | 300 μm × 300 μm | Control signal mux | 950 μm × 800 μm |
| Whole chip without pads | 1500 μm × 1200 μm | Whole chip without pads | 5200 μm × 4000 μm |

**Table 3**
Layout area of both CMOS designs

| | Total power/mW | Leakage/μW | SNN's power/mW | Other power/mW | # of neurons |
|---|---|---|---|---|---|
| Large design | 13.24 | 1.06 | 9.485 | 3.755 | 406 |
| Small design | 1.468 | 0.0679 | 0.889 | 0.579 | 7 |
| Large/small ratio | 9.019 | 15.61 | 10.669 | 6.485 | 58 |

Table 3 shows the power consumption of both designs. Although, in term of the number of neurons, the large design is 58 times of the small design, its area is only 4 times of the small design and its power consumption is only 9 times of the small design. This implies that the area and power consumption of the design do not increase linearly with respect to the size of the design and thus allows us to increase the design's size with lower cost. The next section presents the results of both chips and provides a cross comparison between all platforms.

## 4. Results and discussion

The indirect training algorithm was implemented in MATLAB, FPGA and CMOS to train the SNN presented in Section 2 to allow the virtual insect to perform the terrain navigation task. The small design has also been tested on an Altera Cyclone II EPC20F484C7 FPGA board to ensure that the implemented design works in real hardware. To implement the small design 21,172 logic elements and 1104 dedicated logic register were used.

The performance of the trained insect was first evaluated by MATLAB simulations, as demonstrated in Fig. 7, in which the green line and the blue line depict trails of the untrained state and the trained state of the virtual insect, respectively. The results show that after the SNN was fully trained, the virtual bug was capable of

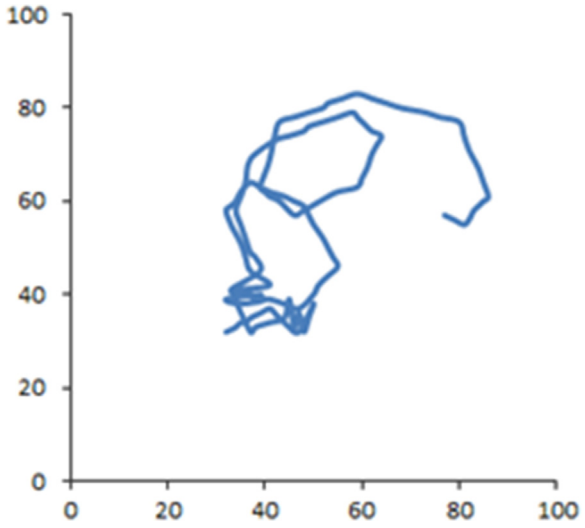


**Fig. 8.** Untrained hardware virtual insect trajectory on a map of size 1000 × 1000.

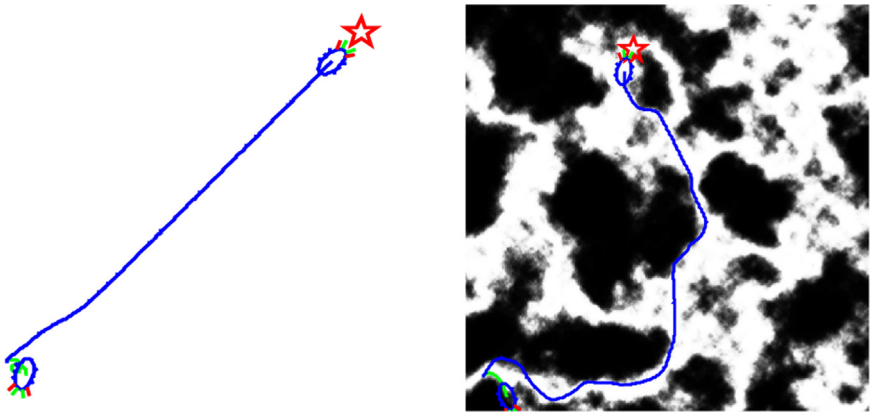

**Fig. 7.** Trails of the virtual insect on a uniform, obstacle-free terrain and on a terrain with different roughness, populated by obstacles [1]. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

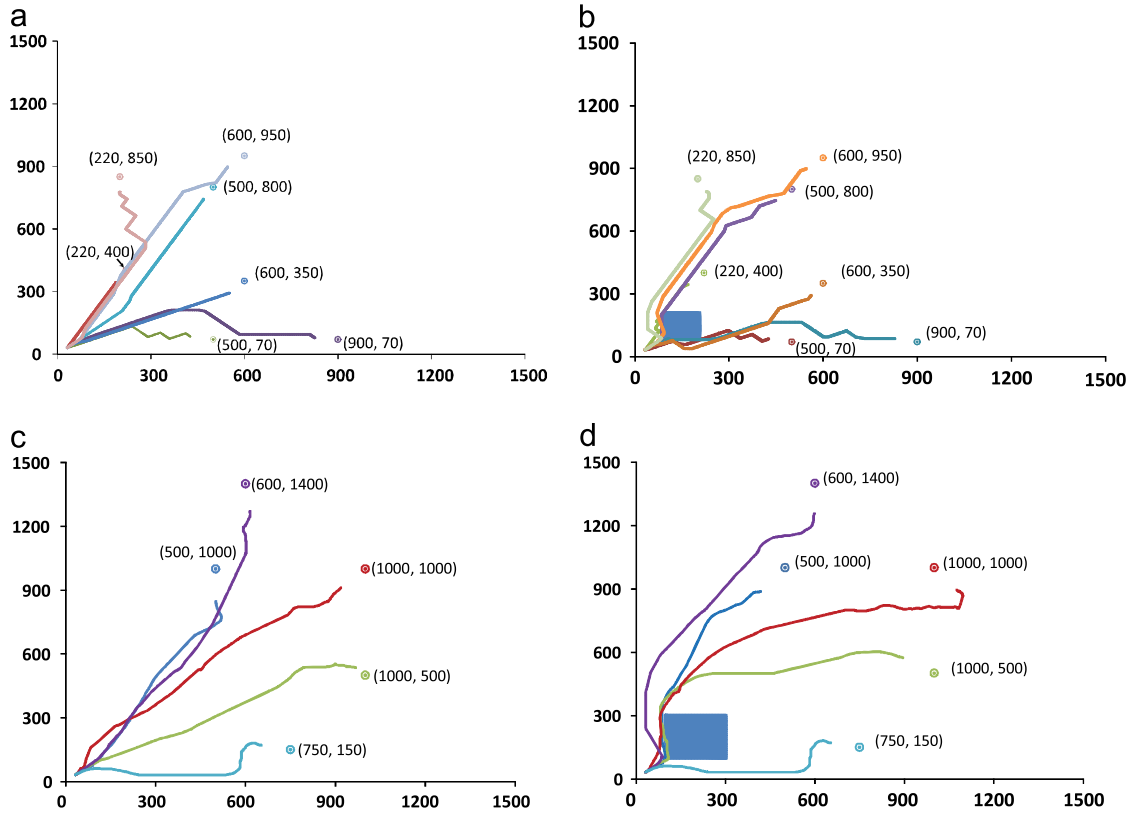


**Fig. 9.** (a) Trails of the small design on a plain map (b) Trails of the small design on a map with a square obstacle (c) Trails of the large design on a plain map (d) Trails of the large design on a map with a square obstacle.

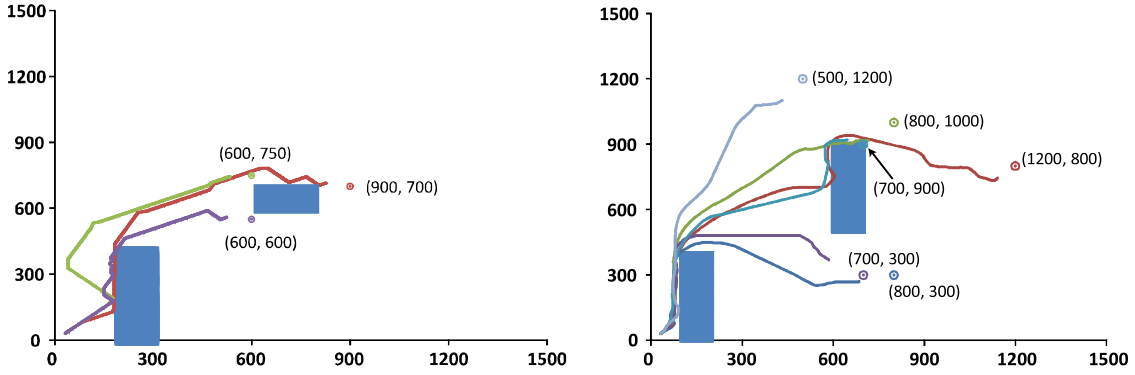


**Fig. 10.** (Left) Trails of the small design on a map with random obstacles (Right) Trails of the large design on a map with random obstacles.

avoiding obstacles and obtaining the target position. The untrained insect really stays in the same vicinity so the trajectory captured by the green line is so small due to the random movement of the insect around the same area.

The confirmation of Fig. 7 in MATLAB is realized with the hardware versions of the virtual insect in Fig. 8. When the insect is untrained its trajectory is unpredictable. In the case of the MATLAB version (green line in Fig. 7), the insect remained in the same surrounding area while that of Fig. 8 shows that the insect area is actually a bit larger than expected. A zoomed in shot is provided in order to show a visual explanation of what is happening. The map size the insect is moving in is actually $1000 \times 1000$ map with a target at (500, 220). The untrained virtual insect wandered randomly within the 100 by 100 area, which is differs a bit from Fig. 7, but provides the flavor of the same phenomenon. The larger space shows that the structure inherent in an untrained hardware is better than a randomized neural network structure in software. The hardware should therefore take less number of training sessions than the software implementation because of this.

Figs. 9 and 10 show the different trajectories under different conditions for both the small and large designs. The testing phase was executed with block obstacles as opposed to the type of map utilized in the MATLAB verification. The reasoning behind the choice stemmed from the fact that the verification hardware is included in each design. Therefore, the complexity of the map had to be reduced in order to achieve reasonable simulation times. Another reasoning behind the choice of maps was the third modification adopted for CMOS circuit simplification. Even if a complex map were generated, the insect would be unable to distinguish a smooth terrain from a rough terrain. It would either see an obstacle present or not present. The cloud map in Fig. 7 from the hardware insect's perspective would just be a black and white map instead of something in grayscale. With the short preamble out of the way we will dive into the results of Fig. 9.

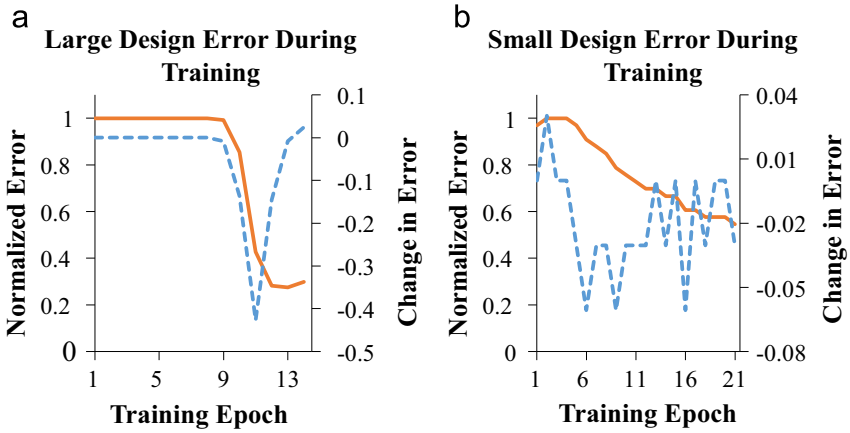


**Fig. 11.** Sample error profile during training. The solid line shows the normalized error between expected and observed firing frequencies and the dotted line shows the change in error from one epoch to the next. The error profile for the large design is shown in (a) and that for the small design in (b). The small design takes more training epochs to minimize error compared to the large design.

Fig. 9a shows the trails of the small design on an obstacle free map. The target locations are at (950, 600), (800, 500), (600, 350), (400, 220), (500, 70), (900, 70) and (850, 200). The target locations were chosen to be in a half-plane as displayed with the dotted diagonal line of slope 1, since a symmetry exists in the structure of the small design. Also visible in Fig. 9a, the targets are represented as dots while the insect originates from the same location each time. For the trained insect, it was able to move towards the target and then stop at a position close to the target. The discrepancy between the location the insect stops and the location of the target comes about because of the way the target sensor signal is generated.

The expression, $\frac{|x_{insect}-x_{target}|^2-|y_{insect}-y_{target}|^2}{2000}$, was used to represent the target sensor signal. As a result, when the numerator had a value is less than 2000, the evaluated expression would take on the value of 0 due to the integer division. This causes the target sensor to generate no signal, thereby causing the insect to stop moving. Therefore, the virtual insect would stop at a location close to the given target but not right next to the target. This is not a drawback but an artifact of the testing circuit. This type of precision error can be overcome by using floating point arithmetic.

Fig. 9b elaborates the trails of the small design on a map with a square obstacle. Targets were kept at the same locations as those in the previous trial (Fig. 9a) in order to exhibit fair comparison. The takeaway here is that the trained virtual insect changed its trajectory due to the presence of the obstacle, even though the target was in the same location. Eventually, the virtual insect was able to stop at a position close to the target. Fig. 9c and d provide similar test results for the large design.

Multi-obstacle tests were performed for both large and small designs. The obstacle choices in Fig. 9b and d showed different results pertaining to trajectories around an obstacle therefore further tests were necessary to validate performance. Fig. 10 on the left shows two different obstacle and target configurations for the small design, while Fig. 10 on the right shows two different obstacle and target configurations for the large design. The small design exhibits a bit choppier trajectory showing some wasted space between the path chosen and the obstacle. The larger design seems to show smoother turns and more hugging of the obstacle when going around it. In addition, with more neurons available, the larger design could explore on a larger map than that the small design could explore.

The differences between both designs is actually a bit more pronounced than first realized. The distinction from the trajectory taken and the accuracy pertaining to wasted moves may be linked to the error present for each of the designs. From Fig. 11, the number of training epochs necessary for the large design is smaller than that for the small design. The large design reduces normalized error drastically with each training epoch compared to the small design. This is fully expected since the small design has converging paths that require synaptic weights that need to be in fairly precise proportions, the training is definitely more gradual than that of the large network. The tolerance set for the error may also explain away the discrepancies between the two.

## 5. Conclusion

This work presents a hardware implementation of an SNN with an indirect training algorithm. Two SNN versions were adopted to validate the software algorithm in hardware and also provide comparisons with respect to the efforts of scalability of the approach. Additionally, a virtual insect model was developed as an example to demonstrate how this approach can be used to solve practical problems. Hardware implementation was accomplished at both the FPGA level and the CMOS level. The implemented design was tested on real hardware to show that the proposed SNN structure and training algorithm can be adopted in circuit designs. Future work may include building a complete virtual insect with sensors and motors or expanding the virtual insect model to solve other types of problems, such as pattern recognition and robotic games.

## References

[1] X. Zhang, Z. Xu, C. Henriquez, S. Ferrari, Spike-based indirect training of a spiking neural network-controlled virtual insect, In: Proceedings of IEEE 52nd Annual Conference on Decision and Control (CDC), 2013, pp. 6798–6805.
[2] Y. Liu, X. Zhang, H. Li, D. Qian, Allocating tasks in multi-core processor based parallel system, In: IFIP International Conference on Network and Parallel Computing Workshops, 2007, pp. 748–753.
[3] T. Schoenauer, A. Jahnke, U. Roth, H. Klar, Digital neurohardware: principles and perspectives, In: Proceedings of Neuronal Networks in Applications, 1998, pp. 101–106.
[4] W. Maass, Networks of spiking neurons: the third generation of neural network models, Neural Netw. 10 (9) (1997) 1659–1671.
[5] W. Maass, C.M. Bishop, Pulsed Neural Networks, MIT press, USA, 2001.
[6] C. Mead, M. Ismail, Analog VLSI Implementation of Neural Systems, Springer, Germany, 1989.
[7] P. Treleaven, M. Pacheco, M. Vellasco, VLSI architectures for neural networks, IEEE Micro 9 (6) (1989) 8–27.
[8] M. Glesner, W. Pöchmüller, Neurocomputers: An Overview of Neural Networks in VLSI, CRC Press, USA, 1994.
[9] D. Hammerstrom, A highly parallel digital architecture for neural network emulation, in: Ebong Idong (Ed.), VLSI for artificial intelligence and neural networks, Springer, 1991, pp. 357–366.

[10] U. Ramacher, SYNAPSE—a neurocomputer that synthesizes neural algorithms on a parallel systolic engine, J. Parallel Distrib. Comput. 14 (3) (1992) 306–318.
[11] P. Arena, L. Fortuna, M. Frasca, L. Patane, Learning anticipation via spiking networks: application to navigation control, IEEE Trans. Neural Netw. 20 (2) (2009) 202–216.
[12] H. Burgsteiner, Imitation learning with spiking neural networks and real-world devices, Eng. Appl. Artif. Intell. 19 (7) (2006) 741–752.
[13] S. Ferrari, B. Mehta, G. Di Muro, A.M.J. VanDongen, C. Henriquez, Biologically realizable reward-modulated hebbian training for spiking neural networks, In: Proceedings of IEEE International Joint Conference on Neural Networks, 2008, pp. 1780–1786.
[14] E.M. Izhikevich, Simple model of spiking neurons, IEEE Trans. Neural Netw. 14 (6) (2003) 1569–1572.
[15] E.M. Izhikevich, Which model to use for cortical spiking neurons? IEEE Trans. Neural Netw. 15 (5) (2004) 1063–1070.
[16] S.M. LaValle, Planning Algorithms, Cambridge University Press, UK, 2006.
[17] W. Gerstner, W. Kistler, Spiking Neuron Models: Single Neurons, Populations, Plasticity, Cambridge University Press, Cambridge, UK, 2001.
[18] G.-q Bi, M.-m Poo, Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type, J. Neurosci. 18 (24) (1998) 10464–10472.
[19] G.S. Snider, Spike-timing-dependent learning in memristive nanodevices, IEEE Int. Symp. Nanoscale Arch. (2008) 85–92.
[20] C. Zamarreno-Ramos, L.A. Camunas-Mesa, J.A. Perez-Carrasco, T. Masquelier, T. Serrano-Gotarredona, B. Linares-Barranco, On spike-timing-dependent-plasticity, memristive devices, and building a self-learning visual cortex, Front. Neurosci. 5 (26) (2011) 1–22.

**Pinaki Mazumder** received the Ph.D. degree from the University of Illinois at Urbana-Champaign, Urbana-Champaign, in 1988. Currently, he is a Professor with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor. He was the lead Program Director of the Emerging Models and Technologies Program at the US National Science Foun- dation, and worked for 6 years in industrial R&D centers that included AT&T Bell Laboratories, where in 1985, he started the CONES Project –the first C modeling-based very large scale integration (VLSI) synthesis tool. His research interests include current problems in Nanoscale CMOS VLSI design, CAD tools, and circuit designs for emerging technologies including Quantum MOS and resonant tunneling devices, semiconductor memory systems, and physical synthesis of VLSI chips. Dr. Mazumder is a Fellow of the IEEE (1999) and a Fellow of the AAAS (2007) for his contributions in the field of VLSI.

**Idongesit E. Ebong** received the B.S. and M.S. degrees in 2006 from Carnegie Mellon University, Pittsburgh, PA, both in electrical and computer engineering. In 2012, he finished his Ph.D. degree in electrical engineering at the University of Michigan, Ann Arbor. His research interest includes digital/analog integrated circuit design, focused primarily on new devices and low power applications.