BUILDING BRAINS WITH ARM PROCESSORS AND FPGAS

BY

FELIPE GALINDO SANCHEZ

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Science in Advanced Microelectronic Systems Engineering in the Department of Electrical and Electronic Engineering.

September 2016

ABSTRACT

The human cerebral cortex is one of the largest parts of the brain consisting of billions of interconnected neurons allowing to perform actions, movements and reactions. The understanding of such complex systems have resulted in simulations of high computational costs; hence, the development of high-performance solutions has become a research field with great relevance in neuroscience and high-performance computing (HPC) by describing spiking neural networks (SNN) and biologically realistic neuron models in hardware accelerated solutions using Field Programmable Gate Array (FPGAs) and Graphical Processing Units (GPUs).

This ability to understand parts of the human brain, learning processes and being able to perform large-scale simulations has allowed to apply these high-performance techniques for learning processes in the vast amount of data that is often processed nowadays; an endless number of application such as speech recognition, computer vision, natural language processing, drug discovery, pattern recognition, among others, resulting in an infeasible processing with the traditional approaches of engineering, mathematical and statistical. Thereupon, with the implementation of optimized high-performance neural network solutions, these data can be processed using learning algorithms, in what is commonly referred as deep learning.

Therefore, this project is initially focused in a background research on neuron anatomy, neuron models, spiking neural networks (SNN), network topologies, deep learning, neural coding schemes and learning methods. Afterwards, the proposed FPGA-based solution of a fully connected feed-forward network using Izhikevich's model is implemented in a Zynq-7020 SoC using High-Level Synthesis optimizations and AXI4-Stream as the primary data protocol. The current solution is able to simulate more than 25K neurons and 5M synapses with floating and fixed point; resulting from 5 to 7 times faster than with traditional computing solutions and using only 0.2% to 2.3% energy. Finally, some benchmark applications are showcased alluding to deep learning.

***Keywords*:** Spiking Neural Network; FPGA; Deep Learning; High Level Synthesis; Izhikevich; Hardware acceleration; Vivado HLS.

DECLARATION AND DISCLAIMER

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Taught Postgraduate Programmes and that it has not been submitted for any other academic award.

Except where indicated by specific reference in the text, this work is my own work. Work done in collaboration with, or with the assistance of others, is indicated as such. I have identified all material in this dissertation which is not my own work through appropriate referencing and acknowledgement. Where I have quoted from the work of others, I have included the source in the references/bibliography.

Any views expressed in the dissertation are those of the author.

Signed:

Date:

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF EQUATIONS

LIST OF ABBREVIATIONS

AMBA	Advanced Microcontroller Bus Architecture
ANN	Artificial Neural Network
AXI	Advanced eXtensible Interface
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
BRAM	Block Random Access Memory
EDLUT	Event-Driven LookUp Table
FF	Flip-Flop
FLOPS	Floating-Point Operations per Second
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
HLS	High-Level Synthesis
HPC	High-Performance Computing
IDE	Integrated Development Environment
LUT	Lookup Table
LTD	Long-Term Depression
LTP	Long-Term Potentiation
MLP	Multi-layer Perceptron
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
ReSuMe	Remote Supervised Method
SDK	Software Development Kit
SNN	Spiking Neural Network
SRM	Spike Response Model
STDP	Spike-Timing Dependent Plasticity
STP	Short-Term Potentiation
SoC	System on a Chip
TDP	Thermal Design Power
VHDL	Very High Speed Integrated Circuit Hardware Description Language
XPE	Xilinx Power Estimator

$\mathrm{A}_{+}, \mathrm{A}_{-}$	Amplitude of the positive and negative STDP model
α_{n}	Alpha function for channel n depending on the membrane potential v
β_{n}	Beta function for channel n depending on the membrane potential v
$\mathrm{C_m}$	Membrane capacitance
E_L	Leaky reversal potential
E_K	Reversal potential for the potassium channels
E_{Na}	Reversal potential for the sodium channels
E_{syn}	Synaptic reversal potential
$\mathrm{g_K}$	Potassium conductance
$\mathrm{g_{Na}}$	Sodium conductance
g_{syn}	Synaptic conductance
$\mathrm{I_K}$	Potassium current
$\mathrm{I_e}$	Input (electrode) current
$\mathrm{I_L}$	Leaky current
$\mathrm{I_{Na}}$	Sodium current
$\mathrm{I_{syn}}$	Synaptic current
m, n, h	Hodgkin-Huxley's gated channels
n_{total}	Total number of neurons
n_{layer}	Number of neurons per layer
η	Learning rate
$\mathrm{R_m}$	Membrane resistance
s	Delay between pre and post-synaptic firing
$S^a(t)$	Actual output spike train
$S^d(t)$	Desired (target) output spike train
$S^{in}(t)$	Input spike train
t	Time
t^a	Time of actual output spike
t^d	Time of actual desired (target) output spike
t^{in}	Time of input spike
τ_{+}, τ_{-}	Time constants of the learning STDP model
τ_{m}	Membrane time constant
τ_{s}	Synaptic time constant
w_{jk}	Synaptic weight from neuron j to neuron k
w_{bits}	Number of bits per synapse weight
$\Delta \mathrm{w}^{\mathrm{STDP}}$	Synaptic weight update caused by STDP process
v	Membrane voltage/potential
$\mathrm{V_{eq}}$	Equilibrium voltage/potential
$\mathrm{V_T}$	Threshold voltage/potential

1 INTRODUCTION

The aim of this project is to demonstrate that combining the power of a Xilinx FPGA, an ARM Cortex CPU architecture and **high-level synthesis (HLS)** optimizations in a **spiking neural network (SNN)** implementation based on biological models such as **Izhikevich's;** the **performance** and **power consumption** can be improved against implementations using traditional architectures such as high-end Intel processors.

1.1 AIMS AND OBJECTIVES

- **Research** different **neuron models** and **neural network topologies** in the literature, focusing on computational complexity, implementation cost, power consumption, and biological plausibility.
- **Evaluate** the neuron **model** and neural network **topology** selected in C/C++.
- **Analyze** and **profile** the C/C++ **implementation** in order to detect critical sections for the different simulation architectures.
- Apply **high-level synthesis (HLS)** optimizations for proposed algorithm and generate the hardware accelerated block using Vivado tools.
- **Analyze** the overall **performance** and **power consumption** of the solutions after implementing the different simulation architectures.
- **Explore** the research field of **deep learning** and evaluate **learning algorithm**s for the resolution of specific tasks.

1.2 SCOPE

The overall project is scoped by certain limitations due to time, resources and specific constraints defined during different sections in the report. A summary of those limitations is the following:

- **Neuron model** implemented is based on Izhikevich's equations; details of the selection of this model is based on discussions in *Section 1.6.1*
- **Network topology** is being discussed in the *Section 1.7* based on the simulation limitations.
- **Simulation architectures** are limited to four different topologies defined in *Section 2.2.*
- **Hardware acceleration** is limited the resources described in *Section 2* and the implementation based from *Section 3.*

1.3 Related Work

The following table details a comparison of the most relevant related works with the current aimed project.

Neuron Models:
IZH - Izhikevich, ***HH*** - Hodgkin-Huxley, ***LIF*** - Leaky Integrated and Fire, ***2PR*** - Two-compartment P–R

Data Precision:
FLO - Floating Point, ***FIX*** - Fixed Point

Network Type:
FFW - Feed Forward, ***FCN*** - Fully connected, ***2LA*** - Two layers, ***3LA*** - Three layers

Framework/Language:
C - C HLS, ***JAV*** - Java HLS, ***VER*** - Verilog, ***VHD*** - VHDL, ***SYC*** - SystemC, ***OCL*** - OpenCL,
OMP - OpenMP, ***VIV*** - Vivado HLS tools, ***CUD*** - CUDA (Compatible with C, C++, Fortran),
BSV - Bluespec SystemVerilog, ***HWN*** - Hardware description architecture (language no specified),
CAR - Carte programming environment (Compatible with C and Fortran),
EDL - EDLUT application for neural networks simulations (CUDA compatible)

Key	*Ref.*	*Neuron Models*	*Data Precision*	*Network Type*	*Network Size*	*FPGA Family*	*FPGA Freq.*	*LUTs Used*	*GPU Family*	*Framework /Language*
W1	[19]	IZH	FLO	FFW	10 K	Altera	N/A	N/A	NVIDIA	C, OCL
W2	[11]	IZH	FLO	FCN	1K	Virtex 5	133 MHz	27 K	-	C, CUD
W3	[13]	IZH	FIX	2LA	1K	Virtex 4	198 MHz	N/A	-	C
W4	[20]	IZH	FIX	N/A	64K	Virtex 6	100 MHz	205 K	-	JAV
W5	[21]	IZH	FIX	FCN	800	Virtex 5	110 MHz	35 K	-	VHD
W6	[22]	IZH	FIX	N/A	64K	Altera II	200 MHz	N/A	-	BSV
W7	[10]	IZH	FLO	FCN	1K	-	-	-	-	-
W8	[12]	IZH	FLO	FCN	100 K	-	-	-	NVIDIA	SYC, CUD
W9	[15]	IZH	FLO	FCN	117	Virtex 4	85 MHz	1 K	-	HWN
W10	[14]	IZH, HH	FLO	2LA	2 M	Altera II	150 MHz	N/A	-	CAR
W11	[16]	LIF	FLO	3LA	100 K	-	-	-	NVIDIA	OMP, EDL
W12	[8]	IZH	FLO	N/A	1	Virtex 5	100 MHz	6.5 K	-	HWN
W13	[9]	ION/HH	FLO	FCN	96	Virtex 7	100 MHz	251 K	-	C, VIV

W14	[5]	2PR	FLO	FCN	105	Virtex 6	100 MHz	108K	-	VHD

Table 1. Comparative of the key aspects of the related works

Extended details, advantages, disadvantages and key aspects of each related work will be discussed in the following sections; these related works are being referenced by their respective keys in order to facilitate it to the reader accordingly to the table.

1.4 BACKGROUND

The body consist of several sensory organs that react to different input stimuli that exist in the environment, so that they can perform reactions such as movement. Some of the most studies have been based on the **firing rates** of neurons; although there are some cases [1] where it has been found that this property is not sufficient to distinguish some of the brain behaviors.

Therefore, the study of these systems, is nowadays a field of study with great relevance in neuroscience in order to achieve a better understating by modeling living organs and behaviors such as:

- The human **visual cortex** [17] and hence developing **visual** and **pattern recognition** [13] techniques.
- The **cerebellum** and **inferior olive** [9] or **cerebral cortex** and **thalamic neurons** [18] by using a more complex model such as Hodgkin-Huxley's.
- **Leech heartbeats** [2] by using a **two-compartment P–R model**.

1.5 NEURON ANATOMY

The human cerebral cortex is one of the largest sections of our brain; this consists of billions of interconnected neurons that allow the human body to perform different actions such as movements and reactions for instance. These neurons create then neural networks that together help to create the basic human body functions.

1.5.1 Neuron

A neuron is a cell that receives electrical/chemical signals and **transmits information** between the **brain** and other parts of the **body** through interconnected neurons. Some neurons can be connected to sensory organs for instance, or to others cells in order to create **neural networks**.

This cell consists basically of three sections: **dendrites, cell body** or **soma** and the **axon**. The dendrites and the cell body receive an input signal, then it flows through the axon and finally the signal is shared via **electrical** or **chemical synapses.**

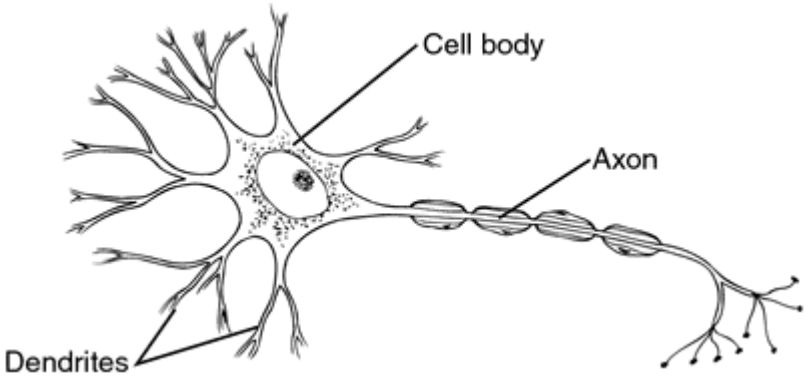


Figure 1. General structure of a neuron with its three basic sections. [23]

1.5.2 Synapses

A **synapse** is a structure that allows the **exchange** of **information** between the axon terminal of the transmitter neuron and dendrites of the neuron receptor. The information to be transferred is contained inside **neurotransmitters**; this is sent as an **electrical activity** to the receiver dendrite through **gated channels** to finally be processed by the neuron receptor.

These structures modify the potential of the receiver neuron and potentially produce an **action potential** (or **spike**). There are two types of synapses: **excitatory synapses** and **inhibitory synapses**. Excitatory synapses are more likely to produce action potentials rather than inhibitory synapses.

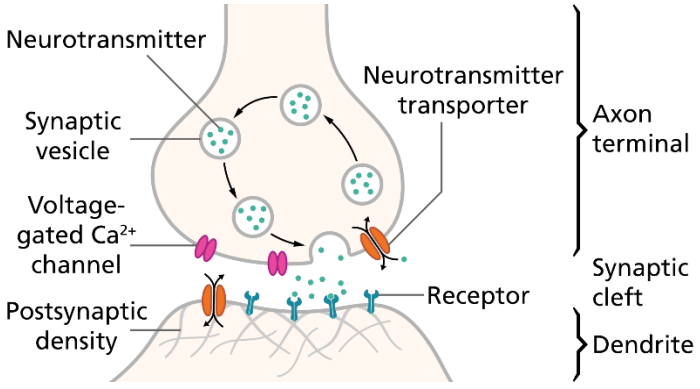


Figure 2. Synapses structure between the transmitter and receptor neuron [24]

1.6 SPIKING NEURAL NETWORKS (SNN)

In this project, the focus of the project is based on **spiking neural networks** (**SNN**) in order to increase the biological realism during simulations in contrast with non-biological neuron models.

1.6.1 Models of Neurons

Biological neuron models intend to increase the realism of simulations performed against the real behavior of a neural system for instance. These **models** described **mathematically**, usually involve several floating-point operations per second (FLOPS), which lead to models with different computational complexities.

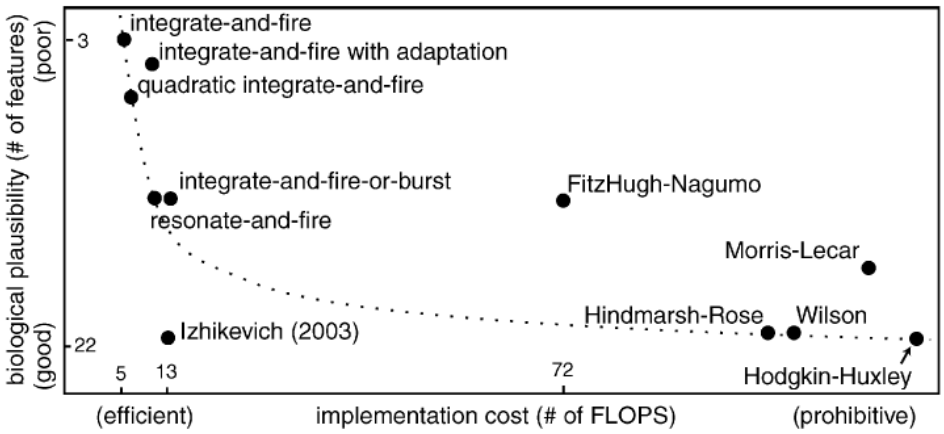


Figure 3. Biological realism against algorithm complexity of different neuron models [4]

A comparison of different neuron models, their biological properties and cost of implementation based on an analysis performed by E. Izhikevich [3] [4] can be summarized in *Figure 3.* The next sections describe the three most relevant models: the simpler model (Integrate and Fire), the most complex model (Hodgkin-Huxley) and an optimal trade-off in terms of complexity and neuron properties described (Izhikevich).

1.6.1.1 Integrate and Fire

Integrate and fire model is one of the simplest biological models that describes the **leaky current** $\mathrm{I_L}$ and the **input current** $\mathrm{I_e}$ flowing through the neuron. The sum of these currents is defined as the membrane capacitance $\mathrm{C_m}$ proportionally to the differential potential $\mathrm{dV/dt}$ as in *Equation 1*.

$$\mathrm{C_m}\frac{\mathrm{dV}}{\mathrm{dt}} = \sum \mathrm{currents} = \mathrm{I_e} + \mathrm{I_L}$$

$$\tau_m \frac{dV}{dt} = R_m I_e + (E_L - V)$$

Equation 1. Currents interacting in the neuron for integrate and fire model

In this model the condition for a neuron to produce an **action potential** (or **spike**), the potential of the neuron needs to exceed a **threshold voltage** V_T (causing the potential to decay again to the initial or **reset voltage** V_R. This spiking property can only be achieved if the equilibrium voltage V_{eq} is satisfied against the threshold voltage V_T.

$$if\ V \geq V_T \rightarrow\ V = V_R$$

$$\text{spiking if } V_{eq} > V_T \rightarrow (R_m I_e + E_L) > V_T$$

Equation 2. Integrate and fire equations

Networks can be constructed on a large scale due to its simple implementation cost, as in **W11**. This is a trade-off that can be selected if the biological realism is not a critical property in the required implementation.

1.6.1.2 Hodgkin and Huxley

The complexity and biological realism of Hodgkin-Huxley's model arises in the simulation of the sodium and potassium **voltage-gated ion channels.** The conductance of these voltage-dependent channels in the membrane together with the probabilities of their gates to be open, is what produces an **action potential.**

These channels commonly referred as n, m, and h are initially described as alpha functions as in *Equation 3*.

$$\text{(channel n)} \quad \alpha_n(v) = \frac{v+55}{10} \Big/ \left(1 - e^{-\frac{v+55}{10}}\right) \qquad \beta_n(v) = 0.125e^{-\frac{v+65}{80}}$$

$$\text{(channel m)} \quad \alpha_m(v) = \frac{v+40}{10} \Big/ \left(1 - e^{-\frac{v+40}{10}}\right) \qquad \beta_m(v) = 4e^{-\frac{v+65}{18}}$$

$$\text{(channel h)} \quad \alpha_h(v) = 0.07e^{-\frac{v+65}{20}} \qquad \beta_h(v) = 1 \Big/ \left(1 + e^{-\frac{v+35}{10}}\right)$$

Equation 3. Hodgkin-Huxley alpha and beta equations for the n, m and h gated channel

Afterwards, their time constants τ_x are modeled as $\tau_x(\alpha_x, \beta_x) = 1/\alpha_x + \beta_x$ and their asymptotic values as $x_\infty(\alpha_x, \beta_x) = \alpha_x/\alpha_x + \beta_x$ for each channel n, m, and h as in *Figure 4*.

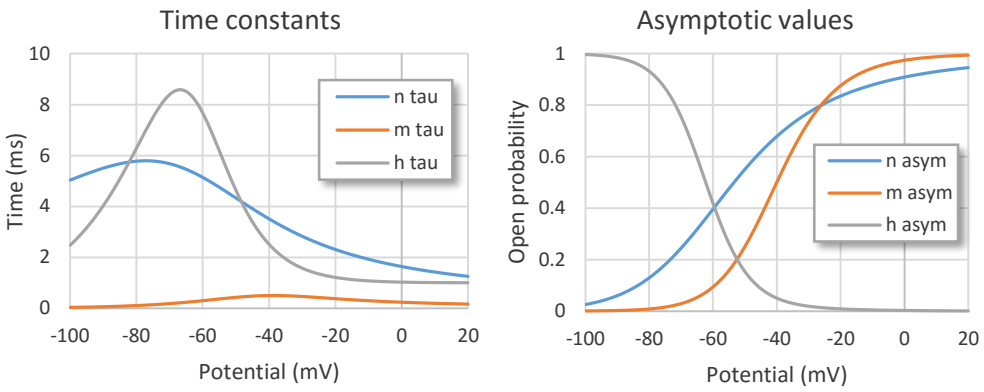


Figure 4. Hodgkin-Huxley time constants and asymptotes gating probabilities.

The gated channels n, m, and h can be expressed in time as $x_\infty + (x_0 - x_\infty)e^{-dt/\tau_x}$ based on the time constants τ_x and opening probabilities of the asymptotic values x_∞ previously described. Consequently, these gated channels lead to define the currents of the potassium I_K and sodiumI_{Na} channels as in *Equation 4*.

$$I_K = g_k(E_K - V) \rightarrow g_k = \overline{g_k}n^4$$

$$I_{Na} = g_{Na}(E_K - V) \rightarrow g_{Na} = \overline{g_{Na}}m^3h$$

Equation 4. Hodgkin-Huxley potassium and sodium currents

Thus, the current flowing through the neuron is expressed as the sum of the input I_e and leaky I_L currents, together with the potassium I_K and sodiumI_{Na} currents.

$$C_m\frac{dV}{dt} = \sum \text{currents} = I_e + I_L + I_K + I_{Na}$$

$$C_m\frac{dV}{dt} = I_e + (E_L - V) + g_k(E_K - V) + g_{Na}(E_{Na} - V)$$

Equation 5. Hodgkin-Huxley model differential equation

1.6.1.3 Izhikevich

A simple spiking model combining the biologically plausibility of Hodgkin-Huxley's model and the computational flexibility of the integrate and fire model is described by Izhikevich [3] [4]. This model is able to represent different spiking and bursting neuron behaviors with only two equations involving four model parameters and one non-linear term v^2.

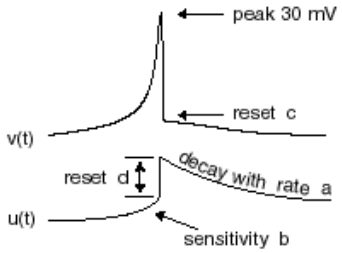


Figure 5. Izhikevich model parameters [3]

$$\mathrm{v}' = 0.04\mathrm{v}^2 + 5\mathrm{v} + 140 - \mathrm{u} + \mathrm{I}$$

$$\mathrm{u}' = \mathrm{a}(\mathrm{bv} - \mathrm{u})$$

$$if\ v \geq 30mV\ \rightarrow \begin{cases} v = c \\ u = u + d \end{cases}$$

Equation 6. Izhikevich model equations [3]

Common excitatory and inhibitory neuron behaviors can be described by Izhikevich [3] using the model parameters ($\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}$) as it is illustrated *Figure 6*.

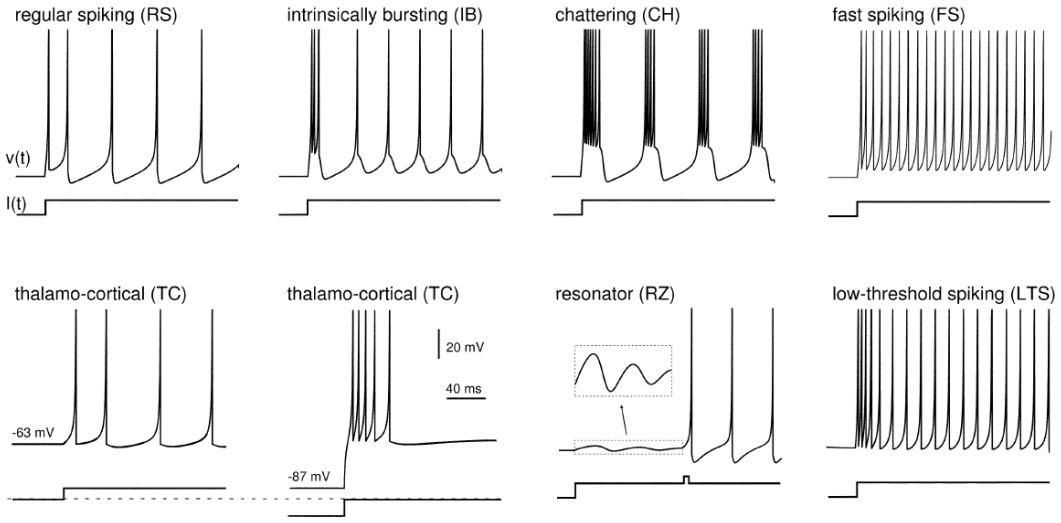


Figure 6. Common excitatory and inhibitory neuron behaviors using Izhikevich model [3]

It can be seen an expected improvement in the quantity of neurons than can be processed in parallel in a single network in the range of thousands of neurons interconnected (***W1*-W6**).

1.6.2 Models of Synapses

The post-synaptic current I_{syn} is modelled as a function of the synaptic conductance g_{syn} and the differential of the membrane potential v and the reversal potential E_{syn}. For excitatory synapses the typical reversal potential is $E_{syn} = -75mV$ whereas for inhibitory synapses it is $E_{syn} = 0$; hence the behavior of increasing or decreasing the membrane potential of the neuron for excitatory and inhibitory synapses respectively can be inferred by the sign of the differential of both potentials $v(t) - E_{syn}$.

$$I_{syn}(t) = g_{syn}(t) \cdot \big(v(t) - E_{syn}\big)$$

Equation 7. Postsynaptic current representation

The synaptic current I_{syn} for a N number of synapses connected to a neuron, is described in the following equation as the sum of the multiple synapses with synapses weights and reversal potentials.

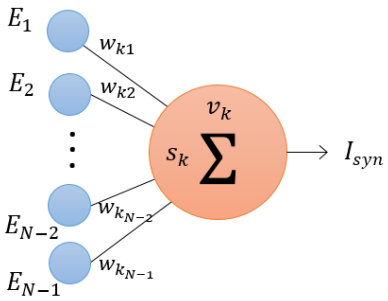


Figure 7. Synaptic input model representation

$$g_{syn}(t) = w \cdot s(t)$$

$$I_{syn}(t) = \sum_j w_{jk} \cdot s_k(t) \cdot \left(v_k(t) - E_j\right)$$

Equation 8. Synaptic current model equation

The **synapse conductance** s is modelled as an alpha function where the **rising** part refers to the **binding** of the **neurotransmitter** of the pre-synaptic neuron to the different gating **channels**; while the **falling** part refers to the **unbinding** from the cleft at a certain rate modeled by the synaptic time constant τ_s.

$$s(t) = te^{-t/\tau_s}$$

Equation 9. Synaptic conductance model equation

Because of the **high-cost** implementation of **exponential** functions in a **computational system**, the **falling** part of the synaptic conductance s is described as a **proportional decay** with a synaptic time constant expressed in *Equation 10,* and the **rising** part as an **increase** wherever a **spike** arrives to the neuron.

$$\tau_s \frac{ds}{dt} = -s(t) \text{ with } s(t) \rightarrow s(t) + 1 \text{ whenever there is a spike}$$

Equation 10. Synaptic conductance model equation (no exponential)

After a $1000\ ms$ simulation of the synaptic conductance with random spikes, it is seen in *Figure 8* the non-significant impact between the response of the exponential and non-exponential model.

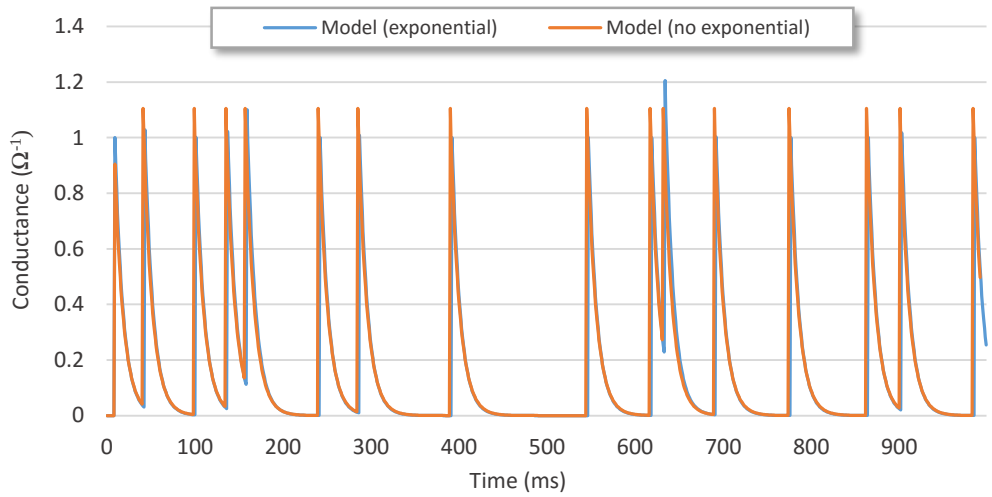


Figure 8. Synaptic conductance response to random pre-synaptic stimulus

1.7 Network Topologies

Feed forward network refers to a topology consisting of multiple layers of neurons, where **neurons** in a certain layer are **connected** to the neurons of the **incoming layer**. The information in this topology flows in **one direction**, from the input layer to the output layer as shown in *Figure 9*. This topology is often considered a **logical layer** due to the fact that this layer does not contain feedback loops, and so, results are not based on previous results.

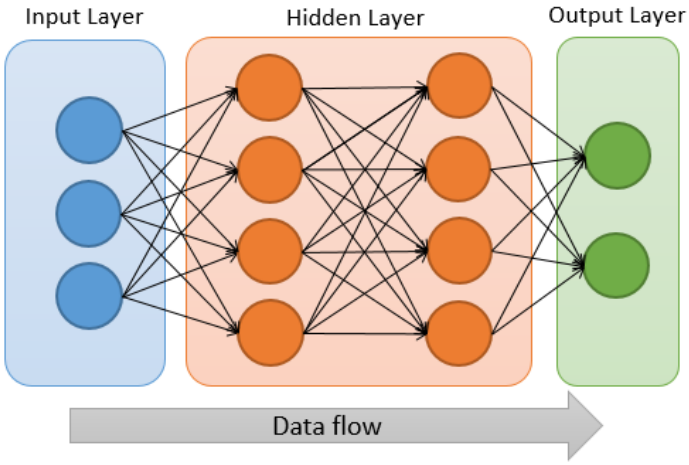


Figure 9. Fully connected feed forward network topology

Recurrent network refers to a topology where neurons **can** create **connections** to **previous layers** by forming **feedback loops**. This permits to retain a certain amount of information as an **internal memory**, so that outputs can **depend** on **previous results** as it is shown in *Figure 10*. Although this topology is able to learn patterns depending on past results, this may lead to slower learnings for logic patterns where there is no need of past results.

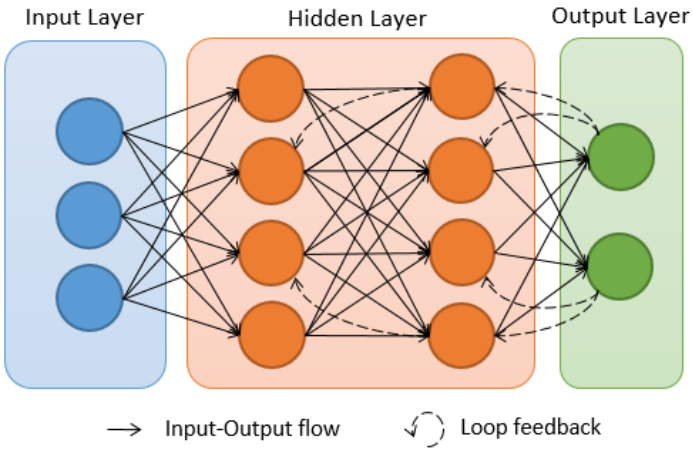


Figure 10. Fully connected recurrent network topology

1.8 Simulation Limitations

The optimal selection of a neural network might be driven by three main implementation properties: **neuron model**, **network size**, **network topology**, **data precision** [7] and **hardware availability**. These properties define the scope of the implementation and its limitations, and hence, the optimal trade-off can be selected for the desired performance without discarding the biological realism as it is illustrated in *Table 2*.

Impact	*Neuron model*	*Network topology*	*Network size*	*Data Precision*	*Hardware availability*
Latency and Performance	Yes	Yes	Yes	Yes	Yes
Variable Dependencies	Yes	Yes	-	-	-
Memory Limitations	-	Yes	Yes	Yes	Yes
Parallelism Feasibility	Yes	Yes	-	-	Yes

Table 2. Simulation impacts against selected implementation properties.

A detailed comparison of relevant works defined in *Table 1* using FPGA acceleration against neuron models selected and their implementation properties is presented in *Table 3*.

Neuron Model	*Simple (Integrate and Fire)*	*Medium (Izhikevich)*		*Complex (Hodgkin-Huxley)*
Network topology	Any	Non-Fully connected	Fully connected	Any
Complexity	Simpler → More Complex			
Network size	+100K ← 100K ← 10K ← 1K ← 100 ← 1			
Related works	W11	W1, W4, W6	W2,W5, W7,W9	W12,W13

Table 3. Network configurations against simulation limitations of FPGA related works

1.9 DEEP LEARNING

A large quantity of data is processed nowadays for **speech recognition**, **image recognition**, **natural language processing**, **genomics**, **drug discovery**, among others. Data size, processing speed and an **endless number of different applications** result in a non-viable possibility of describing all these applications with the traditional engineering, mathematical or statistical approaches. Hence, with the help of **neural networks** able to process large data sets and a set of **learning algorithms**, is what is commonly known as **deep learning** [46]**.**

However, as it is mentioned before, large data sets require **high processing speed** and **computing power**, therefore, deep learning development's trends [45] are being pushed to be assisted by **CPU/GPU** [6] and/or **CPU/FPGA** alternative solutions.

1.9.1 Neural Coding Schemes

Coding information in Artificial Neural Networks (ANN) is a trivial process when units (neurons) are modeled with analog inputs; however, when it comes to SNN, this becomes a relevant research in order to understand all the rich ways for **coding information** in neurons.

Matthew W. Jones [27] describes that **firing rates** are known to be useful for encoding information when **decision-making** is required in order to perform spatial memory tasks. Jones shows through some experiments that the correlation of the same information between a population of neurons can also influence certain decisions.

Neural coding can be classified into three main categories: **rate coding**, **temporal coding** and **population coding** [26] [28].

1.9.1.1 Rate Coding

Rate coding is a common traditional scheme where the information is contained basically in the **mean firing rate** produced by the action potential of a neuron. Other **statistical** analysis may be applied for correlation between the rates over different neurons in order to determine decisions and behaviors for instance.

1.9.1.2 Temporal Coding

Temporal coding is another coding scheme where information is contained in **the spike timings** allowing to express **different features** that may not be express purely by the firing rate. This scheme is used in the electric fish experiment [25] for the recognition of different species that reveal specific signals accordingly to the "listening" of their electric organ's discharges.

1.9.1.3 Population Coding

Population coding is a scheme where information is encoded over a **set of neurons** known as a **population**. Hence, the different input patterns of the combination of spike timings and firing rates determines the data to be expressed.

1.9.2 Learning Methods

Changes in synaptic weights affect the **timing** and **strength** of **spikes** between neurons. If a synapse **increases** its weight, it is known as a **long-term potentiation (LTP)** process, whereas if it **decreases,** it is said to be a **long-term depression (LTD)** process**.** This behavior of updating synaptic weights is called **learning process**; this can be categorized as a **supervised** or **unsupervised** learning [38].

1.9.2.1 Unsupervised Learning

Hebb's theory introduced in 1949 that is often summarized as ***"neurons that fire together wire together***" [55]. This to be said the basis of unsupervised learning, where synaptic connections are reinforced accordingly to the **pre-synaptic** and **post-synaptic spike timings** of interconnected neurons; commonly referred as **Spike-Timing Dependent Plasticity (STDP)** [29] [30].

A phenomenon discovered by Bi and Poo [31] and adapted for STDP learning processes is often modeled as in *Equation 11.*

$$\Delta w^{STDP}(s) = \begin{cases} A_{-} \cdot exp(+s/\tau_{-}), & s \leq 0 \\ A_{+} \cdot exp(-s/\tau_{+}), & s > 0 \end{cases}$$

Equation 11. Spike-Timing Dependent Plasticity (STDP) learning function

In the previous equation, $\mathbf{A}_+$ and $\mathbf{A}_-$ represent the **maximum weight update** allowed in the STDP learning function for the positive and negative delay of the pre and post-synaptic firings respectively, while $\boldsymbol{\tau}_+$ and $\boldsymbol{\tau}_-$ represents the **rate** at which the **weight update approaches** $\mathbf{A}_+/\mathbf{A}_-$ as the delay tends to zero. *Figure 11* illustrates the response to the synaptic behavior described in *Equation 11* using $\mathrm{A}_+ = \mathrm{A}_- = 1$ and a time constant of $2ms$, $5ms$ and $12\ ms$ for the exponential time constant τ_+ and τ_-.

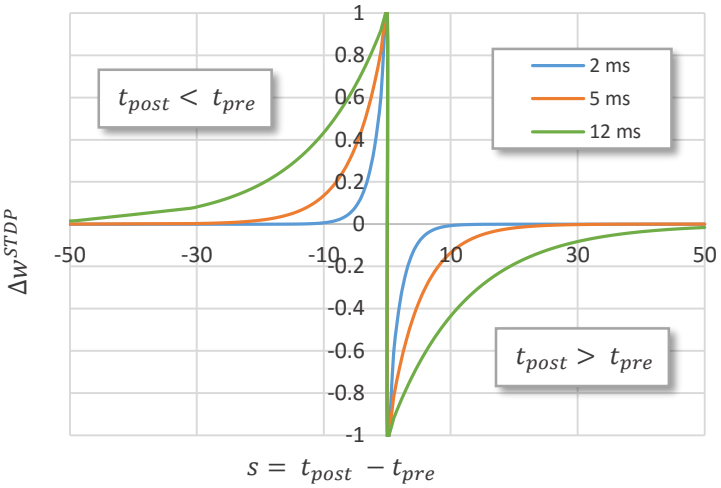


Figure 11. STDP response for learning process with different $\tau_{+,}\tau_-$ and $A_+ = A_- = 1$

1.9.2.2 Supervised Learning

Unlike **unsupervised** learning that requires **competition** between neurons in order to learn, **supervised** learning requires explicit **expected goals** to be learned. Many learning methods [28] [32] currently implemented in spiking neural networks use approaches such as gradient estimation, statistics, linear algebra, Hebbian learning, remote supervision, among others; nonetheless, the main objective of any algorithm is that given an input train of spikes $\sum S^{in}(t)$ and a target output train of spikes $S^d(t)$, a set of $\sum w_{jk}$ need to be found so that the **difference** between the **actual** and **target output** $\left|S^a(t) - S^d(t)\right|$ **tends** to **zero.** A list of some of the most relevant supervised learning methods compared by Kasiński and Ponulak [39] is given in *Table 4.*

Author/Method	*Approach*	*Presented Tasks*	*Coding Scheme*	*Ref.*
S. Bohte - SpikeProp	Gradient estimation	– Classification	Time-to-first-spike	[13]
J. Pfister, D. Barber, W. Gerstner	Statistical approach	– Spike sequence learning – Classification	Precise spike timing	[40] [41]

A. Carnell, D. Richardson	Linear algebra formalisms	– Spike sequence learning	Precise spike timing	[42]
B. Ruf, M. Schmitt R. Legenstein, Ch. Naeger, W. Maass	Supervised Hebbian learning	– Classification – Spike sequence learning – Input-output mapping	Time-to-first-spike and precise spike timing	[43] [44]
F. Ponulak - ReSuMe	Remote supervision	– Spike sequence learning – Input-output mapping – Neuron model independence – Real-life applications	Precise spike timing	[32] [36] [37]

Table 4. List of supervised learning methods compared by Kasiński and Ponulak [39]

1.9.2.2.1.1 SpikeProp

Bohte [33] presented a method based on **gradient estimation** called *SpikeProp* as an analog version of backpropagation algorithm [34] [35] version used in traditional artificial neural networks (ANN).

SpikeProp algorithm takes as a reference the Spike Response Model (SRM) to describe the neuron potential behavior and train the network by **minimizing the quadratic error** of the real and target output errors as in *Equation 12.*

$$E = \frac{1}{2}\sum_{j}\left(t_j^a - t_j^d\right)^2$$

Equation 12. Back-propagation quadratic error function

The back-propagation error can be expressed neuron as the partial derivative of the error E with respect to the weight of the k^{th} synapse with weight w_{ij}^k as in *Equation 13*.

$$\Delta w_{ij}^k = -\eta\frac{\partial E}{\partial w_{ij}^k} = -\eta y_i^k\left(t_j^a\right)\delta_j$$

Equation 13. Back-propagation rule for SpikeProp

Mathematical proofs and a full development of derivative equations for the back-propagation learning rule in the output and hidden layers of feed-forward network can be found in [33].

1.9.2.2.2 ReSuMe

ReSuMe is a ***Re**mote **Su**pervised **Me**thod* [32] [36] [37] or supervised learning model able to code neural information in the **timings of the spike trains** and providing properties such as

locality, computational simplicity and the **online processing** suitability. Its learning is based on the **error minimization** between the real and target output spikes and the principle of this model stands in two basic rules known as **learning windows** (similar to STDP representations).

A **spike train** $S^k(t)$ is defined as the sequence of impulses at certain firing times as $\sum_k \delta(t - \mathrm{t_k^f})$. Hence, the basic *ReSuMe* learning rule can be described in *Equation 14.*

$$a) \quad \Delta w = \Delta \mathrm{w}^{\mathrm{STDP}}\left(S^{in},\, S^{d}\right) + \Delta \mathrm{w}^{\mathrm{STDP}}\left(S^{in},\, S^{o}\right)$$

$$b) \quad \frac{d}{dt}w(t) = S^{d}(t)\left[a^{d} + \int_0^{\infty} \mathrm{W^d(s^d)} \cdot S^{in}\left(t - s^{d}\right) ds^{d}\right] + S^{o}(t)\left[a^{o} + \int_0^{\infty} \mathrm{W^o(s^o)} \cdot S^{in}(t - s^{o})\, ds^{o}\right]$$

Equation 14. ReSuMe learning rule. (a) General and (b) formal expression

In the previous equation, a^d and a^o describe the amplitude of the Hebbian process; for excitatory synapses $a^d > 0$ and $a^o < 0$, whereas $a^d < 0$ and $a^o > 0$ for inhibitory synapses. **Learning windows** $\mathrm{W^d(s^d)}$ and $\mathrm{W^d(s^d)}$ are proposed by Ponulak [32] in *Equation 15*.

$$\mathrm{W^d(s^d)} = +\mathrm{A_+^d} \cdot \exp\left(-\mathrm{s^d}/\tau_+\right), \quad \text{if } \mathrm{s^d} > 0 \quad \text{otherwsise } 0$$

$$\mathrm{W^o(s^o)} = -\mathrm{A_+^o} \cdot \exp(-\mathrm{s^o}/\tau_+), \quad \text{if } \mathrm{s^o} > 0 \quad \text{otherwsise } 0$$

Equation 15. Learning windows for ReSuMe algorithm

By setting $a^d = -a^o = a$ and $\mathrm{A_+^d} = \mathrm{A_+^o} = \mathrm{A_+}$, the ReSuMe learning rule can be simplified as follows:

$$\frac{d}{dt}w(t) = [S^{d}(t) - S^{o}(t)]\left[a + \int_0^{\infty} \mathrm{W(s)} \cdot S^{in}(t - s)\, ds\right]$$

$$\mathrm{W(s)} = \mathrm{A_+}\exp(-\mathrm{s}/\tau_+) \quad \text{if } \mathrm{s^d} > 0 \quad \text{otherwsise } 0$$

Equation 16. Simplified ReSuMe learning rule.

Mathematical development and complete details of ReSuMe method can be found in [32] [36] [37], and an interesting heuristic discussion with mathematical forms for the output layer and hidden layers in a multilayer spiking network in [38].

2 RESOURCES

2.1 SOFTWARE

The table below list the software utilized for the development of this project.

Vendor	*Tool name*	*Description*
Generic	*C/C++ IDE*	Generic IDE that supports C/C++ language for **building**, **debugging** and **profiling** such as *Eclipse*, *Visual Studio*, *Code Blocks*, etc.
Xilinx	*Vivado High-Level Synthesis (HLS)*	Xilinx tool able to **synthetize** the C/C++ functions to be accelerated, **generate** the **RTL** description and **analyze** the **performance** estimates and **simulate** the HLS block implemented [48] [49].
	Vivado IP Integrator	Xilinx tool able to **integrate** the HLS **block accelerated** with the **SoC** system along with data movers and interconnection blocks required (e.g. DMAs, reset system, specific data protocol IP cores, etc)
	Xilinx Software Development Kit (SDK)	Xilinx tool able to **build** and **generate** the **SoC binaries** of the **application** created to be uploaded along with the drivers required to communicate with the HLS block accelerated.
	SDSoC Development Environment	Xilinx tool for **rapid prototyping** and **development** in **SoC** devices by using **scripting** with existing Vivado tools with a **single development environment** [51]. A detailed comparison of this approach is analyzed further in the section *3.2.2.1*

2.2 HARDWARE

The hardware utilized for this project involves two conventional processor cores such as an **Intel Core i7**-4510U CPU @ 2.00GHz and a Dual **ARM Cortex-A9** MPCore. For the acceleration it is being used an **Artix-7 FPGA** and an **Intel HD Graphics** 4400 along with the **OpenCL** framework for a GPU acceleration benchmark. ARM CPU along with the FPGA are integrated into the **Zynq-700 SoC** [50].

The next table displays the four benchmarks of the architectures implemented and simulated though this project.

#	Processor	Accelerator	Brief description
1	Intel x86	N/A	- **Traditional** fully-software architecture - **High**-**performance** and **high-power** processor - No acceleration
2	ARM CPU	N/A	- **Traditional** fully-software low-power architecture - **Low**-**performance** and **low-power** processor - No acceleration
3	Intel x86	GPU	- **High**-**performance** and **high-power** processor - **GPU** acceleration using the **OpenCL** framework and the **Intel HD Graphics** 4400
4	ARM CPU	FPGA	- **Low**-**performance** and **low-power** processor - **FPGA** acceleration using the **Artix-7 FPGA** from Xilinx family

Table 5. Implemented and simulated benchmarks

3.1 NETWORK TOPOLOGY SELECTED

The network topology implemented is a **feed forward** topology and it is defined as a $\boldsymbol{L\ x\ n_{layer}}$ network configuration, where every layer can have a maximum number of n_{layer} neurons, and every neuron with at most s_{neuron} synapses.

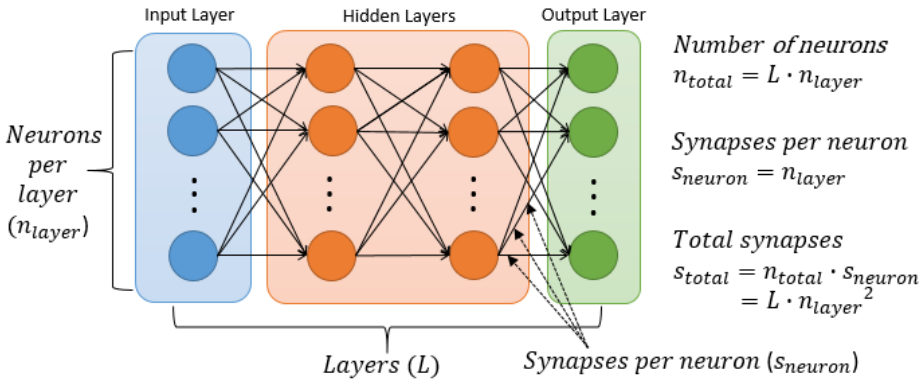


Figure 12. Network topology implemented and parameters

If a non-fully connected network is required with different number of neurons per layer or different number of synapses per neuron, that can be achieved by setting the desired weights to zero and hence those will behave as inactive neurons or synapses as in *Figure 13*.

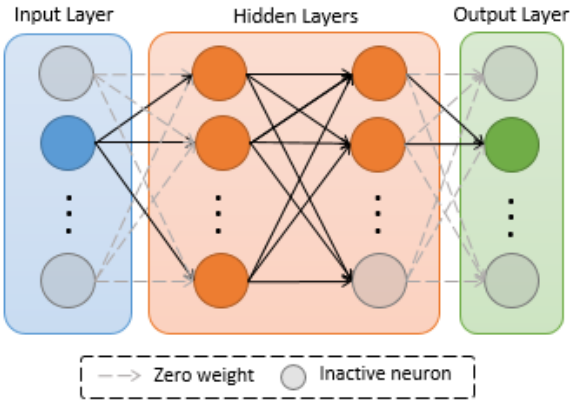


Figure 13. Alternative non fully-connected network topologies

3.1.1 Proposed Algorithm

The algorithm proposed and implemented is based on **Izhikevich's** model described in *Equation 6* and the **non-exponential synaptic conductance** model described in *Equation 10*.

The three main processing blocks that need to be executed every simulation step of $1\ ms$ are described in *Figure 14.* Processing blocks, variables and loops*4.*

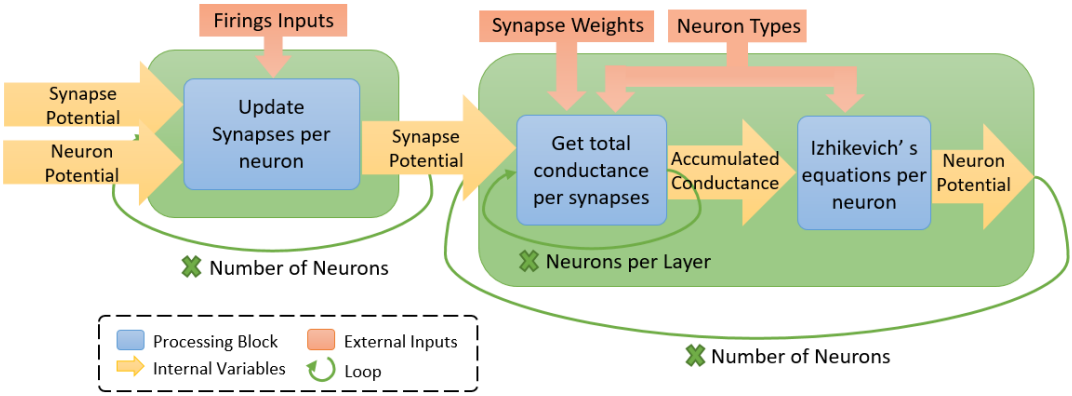


Figure 14. Processing blocks, variables and loops

The following table based on the C implementation on an Intel x86 architecture reflects that the current implementation's bottle neck resides in the second processing block where the number of iterations is n_{layer} larger. Thus, that is the main objective in order to achieve better acceleration results.

Processing block	*Execution time*	*Number of iterations*	*Multiplications per neuron*	*Add/sub per neuron*
Update synapses per neuron	2.15%	n_{total}	1	1
Total conductance per synapse	90.20%	$n_{total} \cdot n_{layer}$	n_{layer}	1
Izhikevich's equations per neuron	7.65%	n_{total}	14	17
Total	100%	$n_{total} \cdot (2 + n_{layer})$	$n_{layer} + 15$	19

Table 6. Algorithm profiling and operations analysis in a 100x100 network

3.2 HLS Acceleration using FPGA

High-level synthesis (HLS) is an **automated designing process** that interprets **C/C++** code into a **register-transfer level (RTL)** hardware description language and implements the same functionally described by the software code. Typically, HLS tools accept C, C++ or SystemC as the input code to be interpreted and the synthetized output (RTL) is expressed in a VHDL or Verilog description.

The purpose of this process is the **acceleration** of certain software functionalities that due to the nature and architecture of conventional computational systems, it results in a non-optimal execution. Hence, by describing a custom hardware that best fits the software code being executed, **better performance** with **lower power** results can be obtained.

3.2.1 Optimization Techniques

High-level synthesis tools tend to optimize the C/C++ code to be implemented by inferring how data flows or how resources are required; so that the tools can **schedule** and **implement** the hardware description with the best results, while **preserving** the **original functionality**.

HLS tools may be assisted by using tools-specific pragmas and/or directives that describe and define details that may not be inferred by automated processes. In this section some HLS techniques are described with specific pragmas from Xilinx Vivado HLS tool [48] [49], although those may be generalized into tools from other vendors.

3.2.1.1 Pipelining Loops

Pipelining is one of most common ways of optimizing, scheduling and accelerating certain functionalities if the executed work is repetitive or loops are involved. Thus, with the adequate hardware description, the executed work can be pipelined so that the interval between the execution of one iteration and the next one is minimized. The following parameters after synthetization are taking into account when analyzing performance results and estimates obtained:

- ***Trip count*** is the number of iterations to perform.
- ***Pipeline depth*** is the number of clock cycles to execute one iteration.
- ***Pipeline interval*** is the number of clock cycles to start the next iteration.
- ***Latency*** is the total number of clock cycles of the pipelined loop

Hence, the latency of a particular loop pipelined may be described as:

$$Latency = (Trip\ count \cdot Pipeline\ interval) + (Pipeline\ depth - \ Pipeline\ interval\ )$$

Equation 17. Latency of a pipelined loop without unrolling

Therefore, ideally without unrolling the latency of the pipeline, the best latency results may be obtained when $Pipeline\ interval \rightarrow 1$ and if the trip count is typically in terms of hundreds of iterations or more ($Trip\ count \gg \ Pipeline\ depth$), the latency may be approximated to $Latency \ \approx Trip\ count$ with $Pipeline\ interval \rightarrow 1$.

Module/Cycle	*1*	*2*	*3*	*4*	*5*	*6*	*…*	*N*	*N_{+1}*	*N_{+2}*	*N_{+3}*
Iteration 1	A	B	C	D							
Iteration 2		A	B	C	D						
Iteration 3			A	B	C	D					
…											
Iteration N								A	B	C	D

Figure 15. Pipelined process with 4 stages and pipeline interval of 1

By default, loops are not pipelined by Vivado HLS; if one loop is desired to be pipelined, the pragma `#pragma HLS PIPELINE` is required or alternatively, or `#pragma HLS PIPELINE II=X` if the targeted pipeline interval is X. Although Vivado HLS tries to achieve that result, the target pipeline interval is not guaranteed if resources, dependencies or scheduling constraints cannot be met.

3.2.1.2 Unrolling Loops

Unrolling a loop is a common technique for improving the performance estimates of a block synthetized. The basics of this is to **multiply** the **processing units** that different stages of an iteration require in order to execute **many iterations** in **parallel**. The overall latency expression for unrolled loops without data and resource dependencies can be described by the *Equation 18*, as well as an illustrative example in *Figure 16* of a four-stages pipelined process, pipeline interval of one and unroll factor of two.

Iteration/Cycle	*1*	*2*	*3*	*4*	*5*	*6*	*…*	*N/2*	*$N/2_{+1}$*	*$N/2_{+2}$*	*$N/2_{+3}$*
Iteration 1	A1	B1	C1	D1							
Iteration 1	A2	B2	C2	D2							
Iteration 3		A1	B1	C1	D1						
Iteration 4		A2	B2	C2	D2						
…											
Iteration N-1								A1	B1	C1	D1
Iteration N								A2	B2	C2	D2

Figure 16. Pipelined process with 4 stages, pipeline interval of 1 and unroll factor of 2

$$Latency = \left(\frac{Trip\ count \cdot Pipeline\ interval}{Unroll\ factor}\right) + (Pipeline\ depth - \ Pipeline\ interval\ )$$

Equation 18. Latency of a pipelined loop and unroll factor greater than one

Unrolling may be enforced in Vivado HLS by setting the pragma `#pragma HLS UNROLL` for a loop to be unrolled completely or `#pragma HLS UNROLL factor=X` where X denotes the number of iterations to be unrolled in parallel if resources and data dependencies allow it.

3.2.1.3 Dependencies

Pipelining and unrolling loops requires that all iterations of the inner loop to be synthetized **does not contain inter-dependencies** in order to be able to executed at least more than one iteration in parallel. Some of those limitations may be removed by swapping a nested loop with an external loop or vice versa, if the functionality does not get affected at all.

This code rewrite typically works for instance when accumulations are performed over arrays, and the way of how arrays are being accessed is not affected. An illustrative example of this is shown in the snipped code of the following figure.

```
// True dependency
for (int i=1; i<50; i++)
    array[i] = array[i-1] + 5;

// True dependency
for (int i=0; i<50; i++)
    for (int j=0; j<20; j++)
        // same location of array is read/wrote
        // 20 (J_SIZE) times sequentially
        array[i] += compute_value(j);

// True dependency
for (int i=1; i<50; i++)
    array[compute_index()] = array[i-1] + 5;
```

```
// False dependency
for (int i=0; i<50; i++)
    for (int j=0; j<20; j++)
        array[i][j] = compute_value(i);

// False dependency
for (int j=0; j<20; j++)
    for (int i=0; i<50; i++)
        // swap loops so that the array location
        // is always different
        array[i] += compute_value(j);
```

Figure 17. False and true inter-dependencies between inner loop iterations

If Vivado HLS is unable to infer the right inter-dependency of a variable, the process may be aided by setting `#pragma HLS DEPENDENCE variable=X inter distance=Y false/true`, where X denotes the name of the variable, Y refers to the optional distance parameter of the maximum number of iterations where the variable can be inferred as a false or true inter-dependency accordingly if it is defined as false or true at the end of the pragma.

3.2.1.4 Perfect Loops

Iterative code and loops are one of the most common code sections in any implementation. If a loop needs to be pipelined in the best way possible, the loop need to be constructed as a **perfect loop**. A perfect loop consists basically on the following properties:

*The **size** and **boundaries** of the loop must be **constant***. In case that the size of the loop is not constant, a solution for this is to set the size of the loop to be a constant value of the maximum number of realistic/allowed iterations that the loop may take, and insert a

conditional execution in the inner loop to skip the inner code if the real condition of the loop size is being met, as it is illustrated in the following snipped code.

```
// Perfect loop = no
for (int i=0; i<N; i++) {
    do_stuff();
}
```

```
// Perfect loop = yes
const int MAX_N = 10    // max value of N
for (int i=0; i<MAX_N; i++) {
    if (i < N) do_stuff();
}
```

Figure 18. Perfect loop transformation with variable size.

*If **nested loops** are implemented, all **code** need to be only inside the **inner loop***. No code statements can be between the upper loops. This can be easily solved by defining in the inner code the conditional execution of the upper loop by using its indexes in order to perform one action whenever one of the upper indexes reaches the desired value.

```
// Perfect loop = no
for (int i=0; i<50; i++) {
    do_stuff_before_j_looop();
    for (int i=0; i<10; i++)
        do_stuff_j();
    do_stuff_after_j_looop();
}
```

```
// Perfect loop = yes
for (int i=0; i<50; i++) {
    for (int j=0; j<10; j++) {
        if (j == 0) do_stuff_before_j_looop();
        do_stuff_j();
        if (j == 9) do_stuff_after_j_looop();
    }
}
```

Figure 19. Perfect loop transformation with code outside the inner loop

3.2.1.5 Memory Access

Data often used like arrays is stored sequentially by default in block RAMs of the FPGA. There are cases when arrays are accessed within loops; if the code is pipelined or unrolled, it will be necessary to fetch multiple parallel data, therefore, due to block RAMs have limited read/write ports (one or two), parallelization is not possible immediately due to **memory limited ports**.

One solution is to split the data between the different block RAMs available in such a way that if it needs to access five data in parallel, at least five different block RAMs are required. Vivado HLS provides some options to solve these issues by using pragmas that allow to reshape a specific array of any dimension into different block RAMs with the following parameters:

- **Type**: block, cyclic, complete. The way how data is organized in the block RAMs
- **Factor**: The number of block RAMs to use for the array required
- **Dimension**: The dimension to reshape

3.2.1.6 Resource Clocking and Latencies

Latencies and clock frequencies of pipelined resources may result in better results when those are configured accordingly to the overall block architecture. It is known that the **maximum clock frequency** is estimated as the **maximum latency** of any **stage in a pipeline task** without considering further implementation and routing timing constraints. In some cases, by increasing the number of stages and the overall latency of a particular resource to produce an output, the stage latency of that particular resource is decreased, and hence, the clock frequency can be increased when working with deep pipelines over hundreds or thousands of iterations (trip count).

An example of this technique is illustrated in *Figure 20*, *Resource A* and *Resource B* latencies are increased along with the number of stages to produce an output, however, because this allows a smaller cycle period and a higher clock frequency, the overall iteration latency for *Case A* is 20 ms and 15 ms for *Case B*, in other words *Case B* is 25% faster.

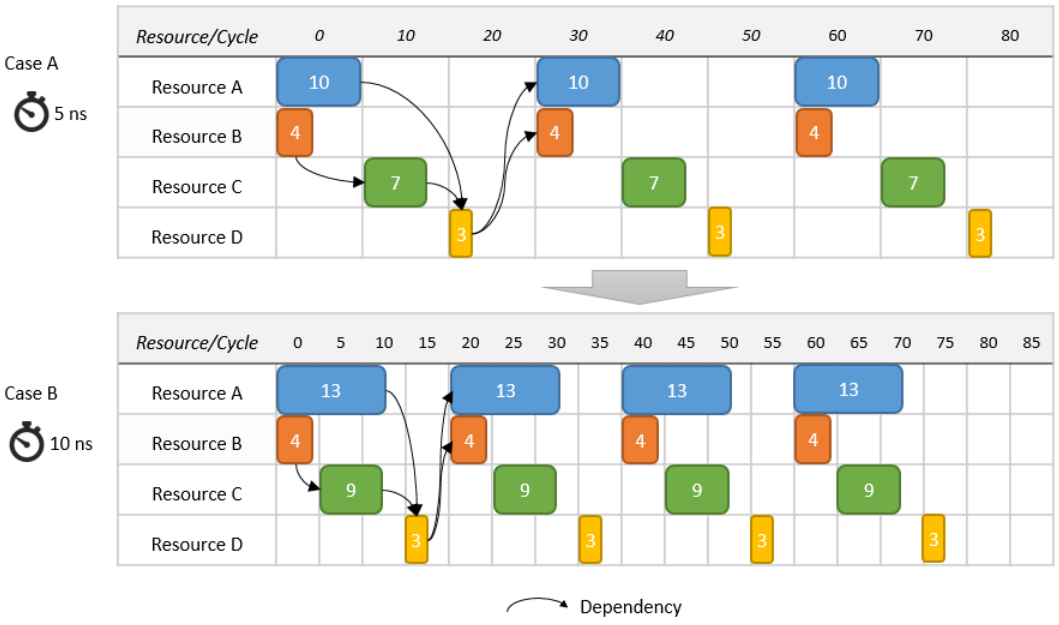


Figure 20. Resource latencies, clocking and pipelining

This technique is applied to different $FAddSub\_fulldsp$ core units in *Appendix 7.1.2* containing the final source code of the processing blocks implemented.

3.2.2 Development Workflow

The development workflow for the HLS acceleration of the spiking neural network implemented involves three Xilinx tools that allow to simulate the final implementation in the SoC device containing the ARM processor and FPGA. *Figure 21* details the different steps required to achieve this final results through the tools and the approximated development

times taken; this time may vary mainly in the synthesis and implementation steps depending on the proportion of FPGA components to be used.

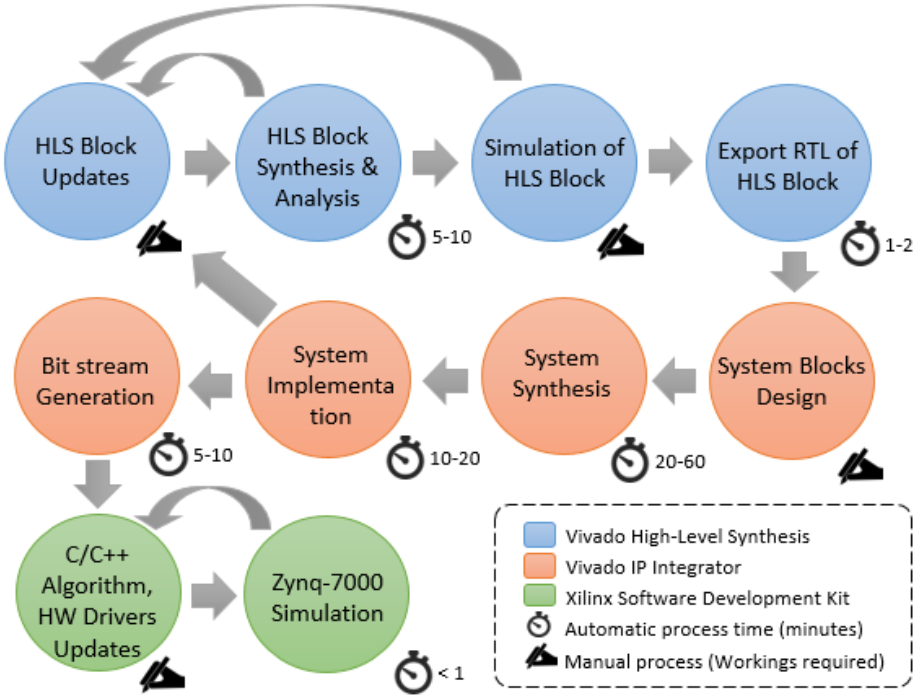


Figure 21. Development workflow diagram

3.2.2.1 SDSoC and the 3-tools Development Process

Xilinx offers the *SDSoC* development environment (Eclipse IDE-based), that allows to performed all the development workflow steps described below in a **single tool** with a **single click** [51]. Starting with the C/C++ acceleration, system design, synthesis, implementation, hardware driver's generation and the final application ready to be uploaded into the SoC system. This is an attractive solution for **rapid prototyping** where specific implementation details are not of high relevance and no further background knowledge is required on hardware acceleration techniques, data transport and IP core details; although it may not lead to the most optimal solution.

Table 7. Comparison between manual and automated workflow compares the most relevant features of the two development workflows; the ***automated*** process that involves the *SDSoC* tool only, and the ***manual*** process involving the *Vivado High-Level Synthesis, Vivado IP Integrator* and *Xilinx SDK* tools and it is reflected in the *Figure 21.* Development workflow diagram.

Features	*Automated*	*Manual*

Tools or steps involved	One (SDSoC)	Three (HLS, IP, SDK)
Tools licenses required	$995 - $1395 [53]	Free (Device limited)
Build and synthesis time	40-60 min	30-40 min
System block design	Automated	Manual (Assisted)
Data protocols knowledge	Not required	Required
IP blocks customization	Low	High
Dual Read/Write IP DMA	Not available	Available
Data size of AXI transfers	32-bit	32 or 64-bit
Tuning of AXI transfers/protocol	Not allowed	Allowed
HLS drivers auto-generated	Completely	Partially
Fixed HLS inputs/outputs	Yes	No
AXI Performance Monitor	Included	Optional

Table 7. Comparison between manual and automated workflow

3.2.3 Algorithm Analysis

The first step consisted to apply the HLS optimization techniques described previously in *Section 3.2.1* were applied to the proposed algorithm described in *Section 3.1.1*. Afterwards, the implementation was driven by the memory and performance estimates in order to achieve an improved performance with the FPGA benchmark as it is described below.

3.2.3.1 Memory Limitation (Block RAM)

Block RAM memory is one of the key features when it is referring to FPGA models, variants, etc. The size of all arrays involved in the proposed algorithm is proportional to the number of neurons n_{total} with the exception of the *synapse weights* proportional to $n_{total} \cdot n_{layer}$. Hence the next figure illustrates the memory required to allocate the *synapses weights* of the proposed algorithm inside the programmable logic (FPGA) as the most critical memory requirement for different *weights* bit sizes.

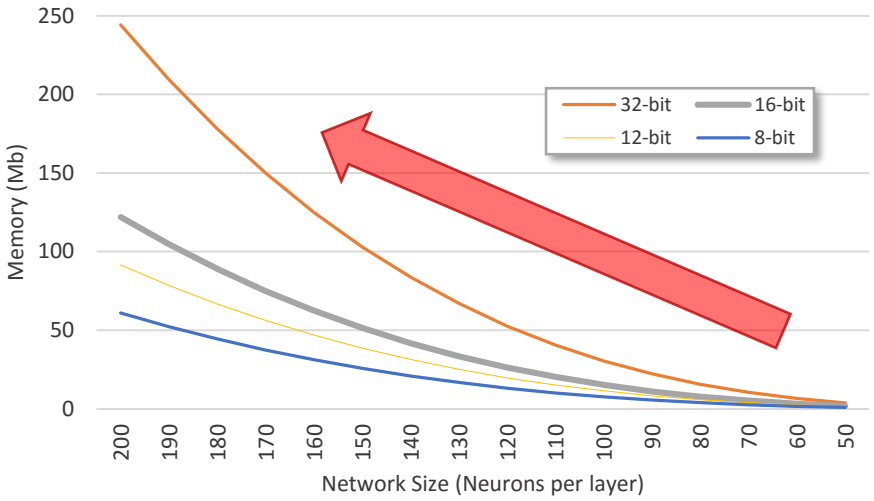


Figure 22. Synapse weights memory required estimates

The different devices of the Xilinx Zynq-7000 family contain between 2 and 27 MB of block RAM, therefore realistically, the largest possible network able to be implemented is in the range of 50 to 100 (with a lower data size precision) neurons per layer. This requirement creates a hard constraint when the physical memory is not available in the available devices, hence, alternative solutions are explored and the data precision becomes an important analysis that is discussed further in this document.

3.2.3.2 Floating/Fixed Point Precision Types

In the domain of realistic values that a *weight* can take, it is observed that it is more critical the precision of the weights rather than a wide range over the real domain. Therefore, in the fixed-point implementation, *weights* are being shifted by *three bits* as a trade-off of achieving a larger precision (smaller Δ weight step) against not being able to take values greater than $2^{-3} = 0.125$. This limitation does not represent any problem with the development of the network as realistic weights are usually smaller than ~ 0.1. Thanks to this approach the precision is being increased by decreasing the Δ weight step from $2^{-w_{bits}}$ to $2^{-(3\,+\,w_{bits})}$. The next table highlights the different digit's precision and the Δ weight step for the floating point and fixed point precisions with different bit sizes.

Precision Type	*Bits*	*Binary Digits*	*Δ Weight Step*
Floating	32	24	$\min = 1.2\ x\ 10^{-38}$
Fixed	32	32	$2.9\ x\ 10^{-11}$
	28	28	$4.6\ x\ 10^{-10}$
	24	24	$7.4\ x\ 10^{-9}$

	20	20	$1.2\ x\ 10^{-7}$
	16	16	$1.9\ x\ 10^{-6}$
	12	12	$3.0\ x\ 10^{-5}$
	8	8	$4.8\ x\ 10^{-4}$

Table 8. Minimum weight's step for different data precision types

Weights are packed into 32 or 64-bit streams of data, hence the fixed-point precision that would make sense withouth compromising complex decoding logic or wasted bits during packaging are floating point and fixed-point of 32, 16 and 8-bits.

One analysis that demostrates the consecuences of the different precision is the analysis of the firing rates and the cross correlation of the fixed-point implementations against the floating point (as target). In the following experiment, $1{,}500\ ms$ simulations were peformed over the four data precision types for the weights data types. This graph plots the average, maximum and minimum firing rate of the 30 neurons in the output layer of a 30x30 fully connected feed-forward network of excitatory neurons with random weights between $2.5\ x\ 10^{-2}$ and $5\ x\ 10^{-4}$ along with the firing rate cross correlation against the implementation with the highest precision (floating point 32-bit).

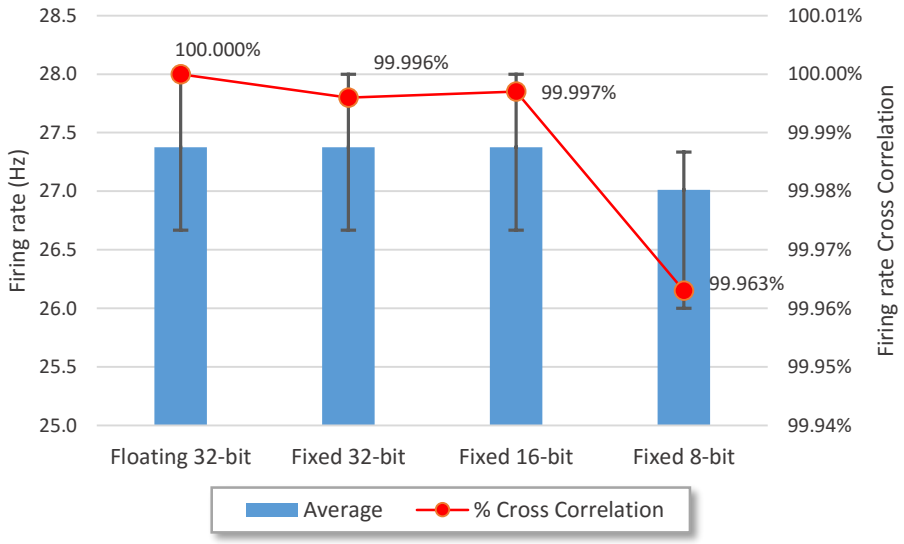


Figure 23. Firing rate statistics over floating and fixed point precision weight's data types

It is observed how the fixed-point implementations of 32 and 16-bit firing rate correlation error is almost null (less than 0.004%), whilst the error in the 8-bit implementation is ten times larger although still less than one perceptual point (0.0437%).

Now the timings of the spikes are analyzed, in a cross correlation of the spikes timings over the fixed point precisions against the floating point implementation. It is observed that 93%

and 92% of the spikes are correlated in the 32 and 16-bits implementation respectively with the floating point implementation with a zero lag delay. On the other hand, the 8-bit implementation is slightly different, with the 87% of the spike shifted between 0 and $3\ ms$ and centered in a $1.5\ ms$ shift average.

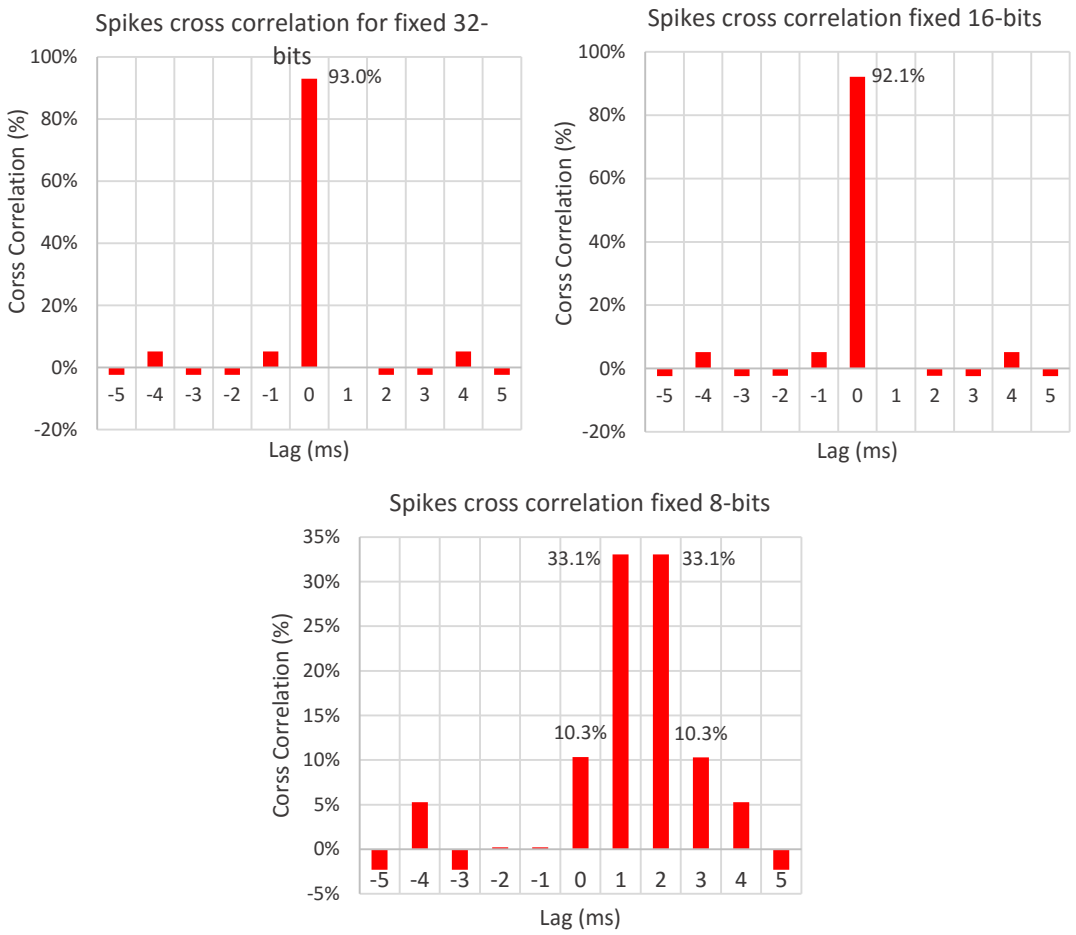


Figure 24. Spikes cross correlation over floating and fixed point precision weight's data types

3.2.3.3 Input Block Throughput

After illustrating the block RAM memory constraint for commercial devices when implementing large networks is desired, the solution to this problem becomes in optimizing the transmission of the ***synapse weights*** **on-demand** without storing them in the FPGA memory. The HLS block processing the algorithm detailed in *Appendix 7.1.3* is implemented to receive **four FIFO streams** in parallel from external memory, matching the four high performance AXI ports available in the Zynq-700 device. Hence **256 bits** are available to be **processed in parallel per cycle** as they are required. Some bytes may be unused, although this approach results in better performance and resource results, as additional logic is avoided for unaligned weights; and full parallelism would not be feasible. An illustrative

example of the different representations in a 100x100 network with 8-bit synapse weights can be expressed in *Figure 25*.

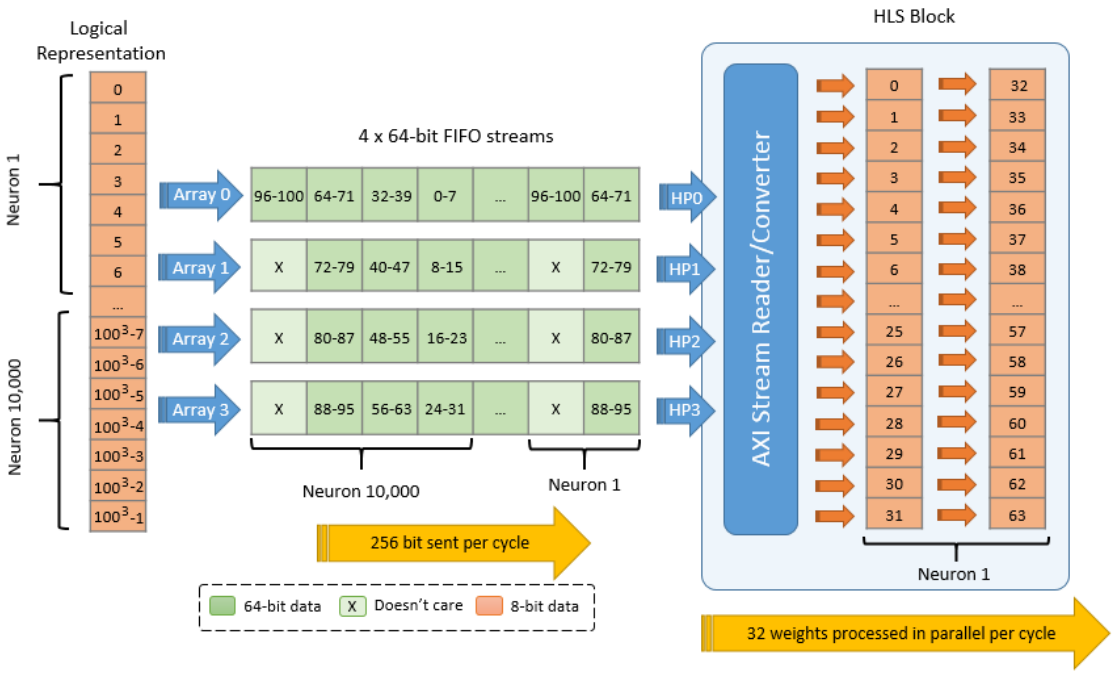


Figure 25. Synapses weight's input block throughput

3.2.3.4 Synapses Updater Block

This synapses updater block presented in *Figure 14* is not a critical path in terms of performance latency, as in a traditional software-only simulation, it lasts around 2.15% and it involves only one addition and one multiplication accordingly to *Table 6;* hence, the latency for this initial block is expressed in *Equation 19.*

$$Pipeline\ interval\ per\ neuron = 1$$

$$Block\ latency = n_{total} \cdot\ Pipeline\ interval\ per\ neuron +\ Iteration\ latency$$

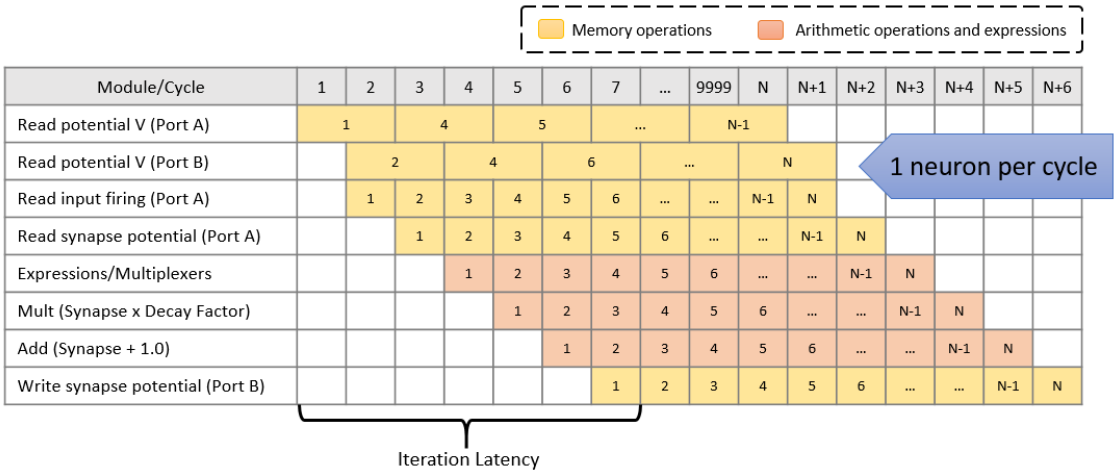
Equation 19. Synapses updater block latency

Due to the fact that $Iteration\ Latency$ is in the range of $5 - 10$ and the number of neurons n_{total} in the range of thousands, then $Iteration\ Latency$ can be skipped, and hence the total latency can be approximated as it follows:

$$Block\ latency \approx n_{total}$$

Equation 20. Synapses updater block latency (approximation)

An example of the pipelining for this block is being described in *Figure 26* for a 100x100 network where it is seen the one-cycle processing interval between each neuron.



Module/Cycle	1	2	3	4	5	6	7	...	9999	N	N+1	N+2	N+3	N+4	N+5	N+6
Read potential V (Port A)	1			4		5		...		N-1						
Read potential V (Port B)		2		4		6		...		N						
Read input firing (Port A)		1	2	3	4	5	6	...	...	N-1	N					
Read synapse potential (Port A)			1	2	3	4	5	6	...	...	N-1	N				
Expressions/Multiplexers				1	2	3	4	5	6	...	...	N-1	N			
Mult (Synapse x Decay Factor)					1	2	3	4	5	6	...	...	N-1	N		
Add (Synapse + 1.0)						1	2	3	4	5	6	...	...	N-1	N	
Write synapse potential (Port B)							1	2	3	4	5	6	...	...	N-1	N

Figure 26. Pipeline analysis of synapses updater l block in a 100x100 network

3.2.3.5 Conductance and Izhikevich's Model Block

Unlike the initial processing block, the conductance and Izhikevich's model block represent **the most critical section** accordingly to profiling in *Table 6* with around 90.20% and 7.65% respectively, as it involves at least $n_{layer} + 14$ multiplications and 18 additions per each processed neuron. The main challenge lies in the **input throughput of** the **weights** that need to be computed in order to get the conductance; therefore, as the **maximum input throughput** achieved by using the four high performance AXI ports available in the Zynq-7000 device is **256 bits per cycle** as stated in *Section 3.2.3.3,* the block latency for this block can be expressed as in *Equation 21.*

$$Pipeline\ interval\ per\ neuron = roundup\left(\frac{n_{layer}\ \cdot w_{bits}}{256}\right)$$

$$Block\ latency = n_{total} \cdot\ Pipeline\ interval\ per\ neuron +\ Iteration\ latency$$

Equation 21. Conductance and neuron potential block latency

The $Iteration\ Latency$ is the latency required to process the **accumulated conductance** along with **Izhikevich's equations** for **each pipelined neuron**. Since the $Iteration\ Latency$ achieved by Vivado HLS is in the range of tens and the number of neurons n_{total} is in the range of thousands, $Iteration\ Latency$ may be omitted, and in other words, the $Block\ latency$ is **driven** by the **accumulated conductance** required to be computed and **not by Izhikevich's equations**. Hence, the total latency can be approximated as follows:

$$Block\ latency \approx n_{total} \cdot\ Pipeline\ interval\ per\ neuron$$

Equation 22. Conductance and neuron potential block latency (approximation)

Consequently, in the following table it can be corroborated the impact of the **data precision** of the **synapse weights** together with the network size to be implemented.

Network Size (n x n)	*8-bit weight*		*16-bit weight*		*32-bit weight*	
	Pipeline Interval	*Block Latency*	*Pipeline Interval*	*Block Latency*	*Pipeline Interval*	*Block Latency*
50	2	5,000	4	10,000	7	17,500
75	3	16,875	5	28,125	10	56,250
100	4	40,000	7	70,000	13	130,000
125	4	62,500	8	125,000	16	250,000
150	5	112,500	10	225,000	19	427,500
175	6	183,750	11	336,875	22	673,750
200	7	280,000	13	520,000	25	1,000,000

Table 9. Performance estimates for conductance and neuron potential block

In *Figure 27*, it can be observed an example of a pipeline execution for a 100x100 network with 8-bit synapse weights and the input throughput of $256\ bits\ per\ cycle$. The implementation of this block can be found in *Appendix 7.1.2.*

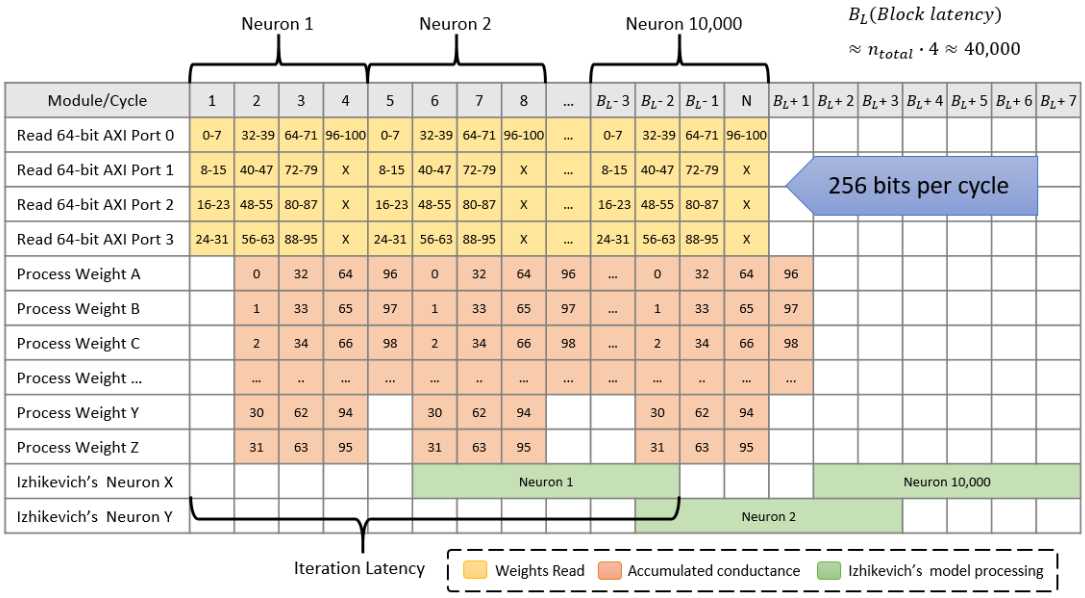


Figure 27. Pipeline analysis of conductance and neuron potential block in a 100x100 network

3.2.3.6 Latency Results Estimates

Finally, the overall latency estimates can be approximated to *Equation 23* and it can be seen the importance that plays the **network size** (n_{total} and n_{layer}), and the data precision for the **synapse weights** (w_{bits}).

$$Overall\ latency \approx Synapses\ updater\ latency + Conductance/potential\ latency$$

$$Overall\ latency \approx n_{total} \cdot \left(1 + roundup\left(\frac{n_{layer} \cdot w_{bits}}{256}\right)\right)$$

Equation 23. Overall HLS block latency estimation

The following figure exemplify the data precision for the **synapse weights** (w_{bits}) for sizes from one to four bytes in either fixed or floating point precision.

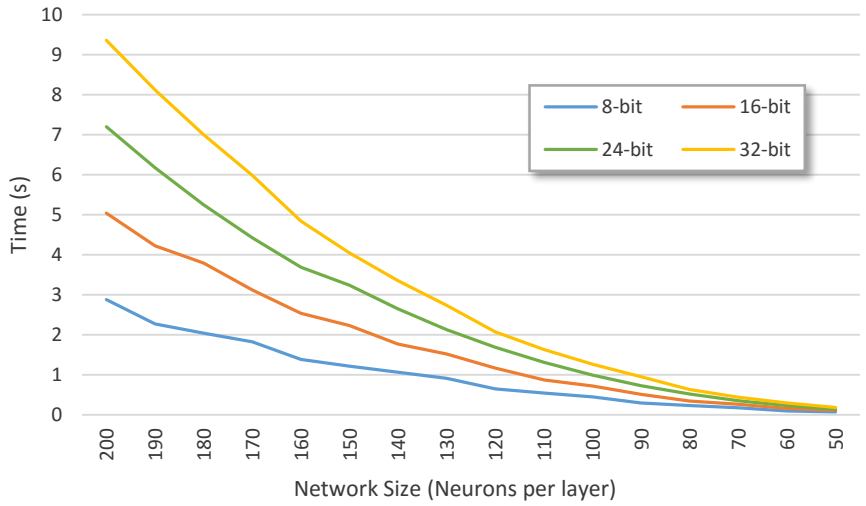


Figure 28. Performance estimates for different weight's data precisions

Four different latency estimates are exposed in *Figure 29* by using three key **implementation's properties***:* the **storage** of the **synapse weight's** inside the programmable logic as BRAM, the **number** of **64-AXI ports** utilized for the input throughput and the different data precision for the **synapse weights** (w_{bits}).

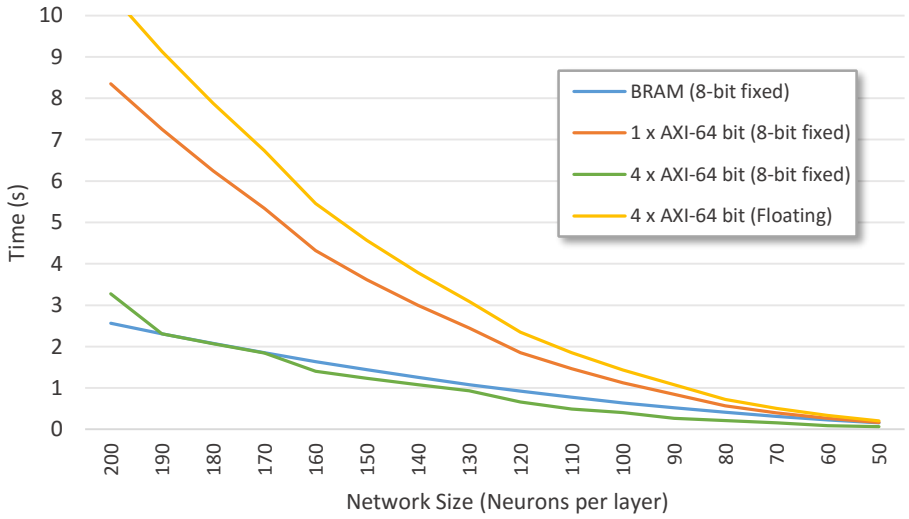


Figure 29. Performance estimates for different implementations

3.2.4 System Architecture and Integration

The overall general architecture of the system implemented involves four main components that are illustrated in the *Figure 30*.

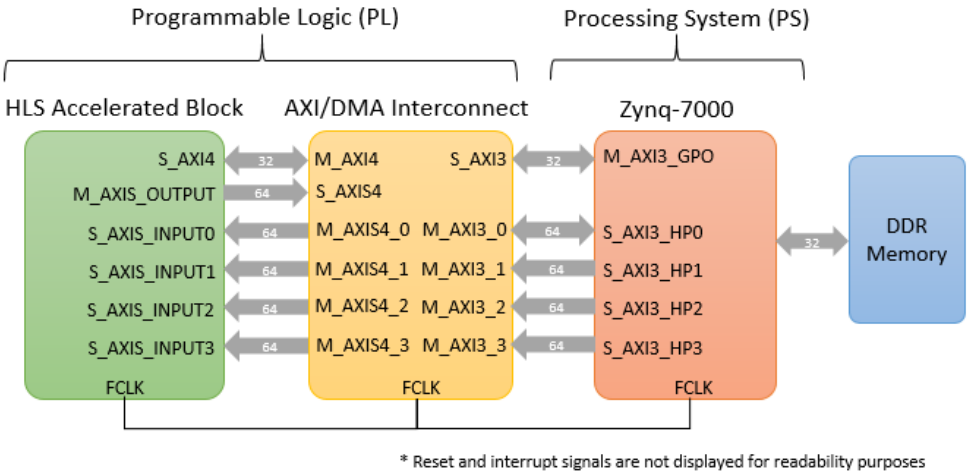


Figure 30. System block diagram implemented

AXI interface is the fundamental data transfer protocol from AMBA family interfaces with its different versions and specifications. Specifications of these protocols and variants can be found in detail in *Chapter 1* of the *AXI Reference Guide* [47].

3.2.4.1 HLS Accelerated Block (Programmable Logic)

This component represents the spiking neural network implementation analyzed and synthetized in the current section. The network within this accelerated block is being stimulated through the following interfaces:

Data	*I/O*	*Port name*	*Protocol*	*Depth size*	*Recurrence*
Neuron types	Input	S_AXIS_INPUT0	AXI4-Stream	n_{total}	$once$
Synapse weights	Input	S_AXIS_INPUT0 - S_AXIS_INPUT3	AXI4-Stream	$n_{total} \cdot n_{layer}$	$every\ 1\ ms$
Input stimulus	Input	S_AXI4	AXI4-Lite	n_{layer}	$every\ 1\ ms$
Neuron firings	Output	S_AXIS_INPUT0	AXI4-Stream	n_{total}	$every\ 1\ ms$
Neuron potentials	Output	S_AXIS_INPUT0	AXI4-Stream	$up\ to\ n_{total}$	$every\ 1\ ms$

Table 10. Data inputs and outputs for the HLS accelerated block

Other common signals such as reset, clock and completion interrupts are implemented in the accelerated block as well. The source code of this HLS Accelerated block can be found in *Appendix 7.1.1*.

3.2.4.2 Zynq-7000 (Processing System)

This component represents the **dual core ARM Cortex-A9 processor** along with the peripherals required to **communicate** with the **programmable logic (PL)** implemented and the DDR memory. *Figure 31* illustrates the components diagram of the processing system such as clock generation, reset, IRQs, High-performance AXI ports, UART, USB and memory interface; components that are being used in the current implementation. Full details can be found in the *Zynq-7000 All Programmable SoC Overview* [50].

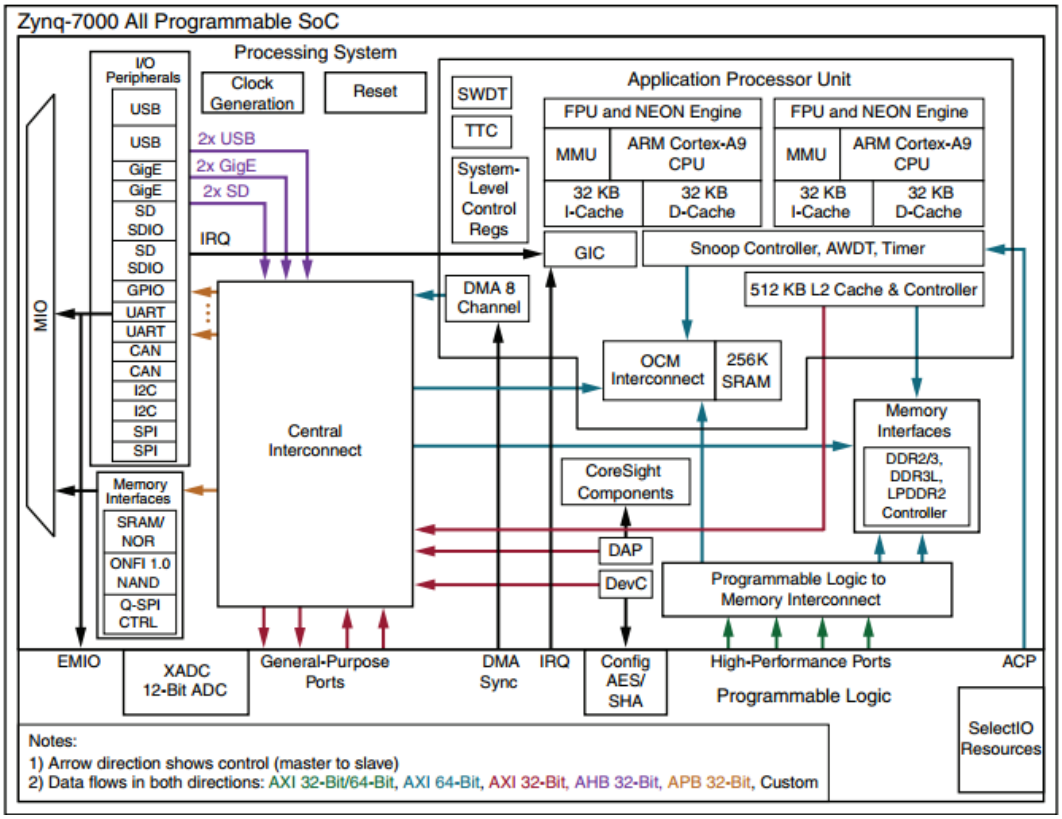


Figure 31. Components diagram of a Zynq-7000 SoC [50]

3.2.4.3 AXI/DMA Interconnect (Programmable Logic)

This component represents the interconnections and translation protocols required to communicate the Zynq-7000 processing system with the HLS accelerated block as it is illustrated in the following figure.

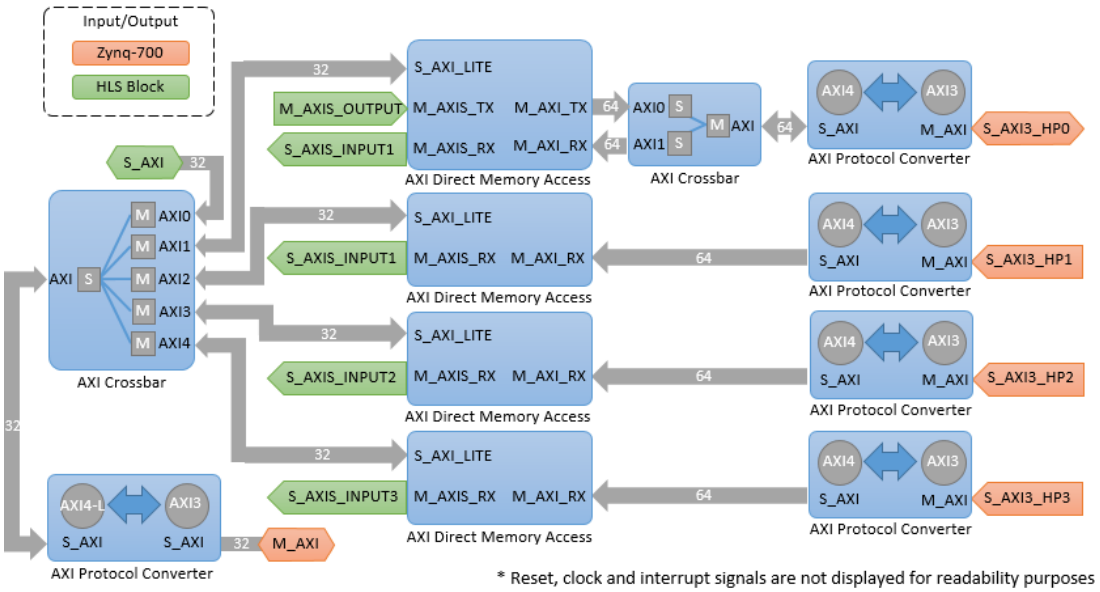


Figure 32. AXI Interconnect block diagram

Full details of AXI Infrastructure IP Cores and the AXI Interconnect can be found in *Chapter 3* of the *AXI Reference Guide* [47].

4 RESULTS

4.1 RESOURCES UTILIZATION

One of the common analysis when benchmarking applications with FPGAs is usually by measuring the utilization of resources when an application is being implemented. *Table 11* displays the utilization of the four main key features (LUR, FF, BRAM and DSP) for different network sizes and both floating and fixed precision types.

The fixed point implementation represents a **32-bit fixed point** implementation for all operations and variables except for **synapse weights** with an **8-bit fixed point** precision.

Precision Type	*Network Size (n x n)*	*LUT*	*FF*	*BRAM*	*DSP*
Floating Point	50	32%	26%	28%	26%
	100	47%	32%	54%	24%
	120	54%	36%	54%	25%
	150	62%	38%	88%	25%
	170	72%	41%	96%	25%
Fixed Point	50	24%	10%	11%	66%
	100	29%	15%	20%	70%
	120	31%	23%	49%	73%
	150	34%	26%	89%	88%
	170	37%	28%	90%	97%

Table 11. FPGA resource utilization of entire system implementation in Z-7020 SoC

4.2 PERFORMANCE ANALYSIS

The following *Figure 333* evidences the **final performance results** for different **implementations** of **benchmarks** and **data precision** over different **network sizes** (n_{layer}) with random input stimulus and a $1{,}000\,ms$ simulation time. It can be observed how processor-only solutions result in the worst performance results and the percentage becomes even worst with larger networks whilst accelerated solutions result in better performance results.

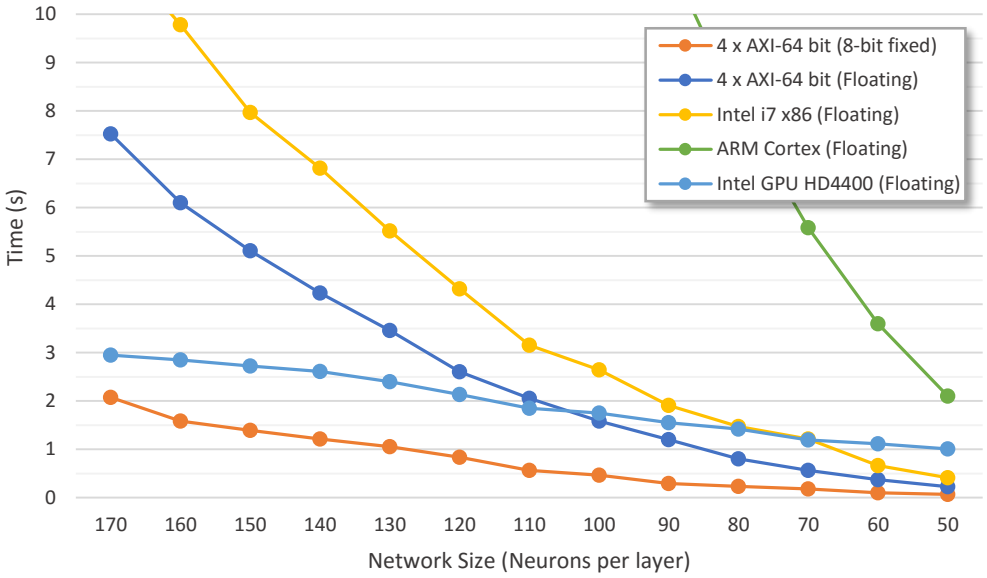


Figure 33. Performance results for benchmarks

4.3 Power and Energy Analysis

Power and energy analysis are relevant metrics in order to evaluate the **actual performance** of a particular solution **against the power** that is being **consumed** by the necessary hardware, and thus achieve such performance results described in *Section 4.2*. Measurement procedures for the different configurations are described below:

FPGA/ARM: The FPGA accelerated solution is measured accordingly to the total on-chip power results provided by Vivado after routing and implementation along with the Xilinx Power Estimator (XPE) tool provided by Xilinx in a Zynq-7020 SoC device. The minimum and maximum energy is computed by using the performance latency of the fixed and floating point implementation respectively. The power consumption range around $1.9\ W$ and $2.3\ W$ for the different network configurations.

ARM CPU: The ARM CPU solution without acceleration is being measured accordingly to the Xilinx Power Estimator (XPE) tool provided by Xilinx in a Zynq-7020 SoC device. The energy computed is being driven by the respectively performance latency of the network configuration over a power consumption of $1.3\ W$ for the minimum boundary and $1.5\ W$ for the maximum boundary.

Intel i7: The energy of this non-accelerated solution is being measured accordingly to specifications in power consumption of Intel i7 mobile processors [52] over the boundaries in standby operation of low-power processors ($\sim 15\ W$), and a higher power consumption of $\sim 100\ W$ such as the TDP specification; still in the range of mobile processors.

Network Size (n x n)	*FPGA/ARM*	*ARM CPU*	*Intel i7 (Mobile)*
50	$9.1 \pm 5.1\ J$	$106.0 \pm 21.2\ J$	$3.61 \pm 3.44\ kJ$
100	$6.8 \pm 3.8\ J$	$71.6 \pm 14.3\ J$	$2.44 \pm 2.32\ kJ$
120	$3.6 \pm 1.8\ J$	$37.6 \pm 7.5\ J$	$1.28 \pm 1.21\ kJ$
150	$2.3 \pm 1.1\ J$	$22.2 \pm 4.4\ J$	$760 \pm 720\ J$
170	$0.3 \pm 0.1\ J$	$3.1 \pm 0.6\ J$	$108 \pm 101\ J$

Table 12. Energy consumption in a simulation of 1000 ms

Table 12 and *Figure 34* illustrates the energy consumption in simulation of $1000\ ms$ accordingly to the procedures described previously possible ranges due to specific hardware under normal conditions of execution. It can be seen that the **FPGA/ARM solution** requires only $\mathbf{0.2}\%$ to $\mathbf{2.3}\%$ of the energy drawn by the **Intel i7 (Mobile)** processor on average.

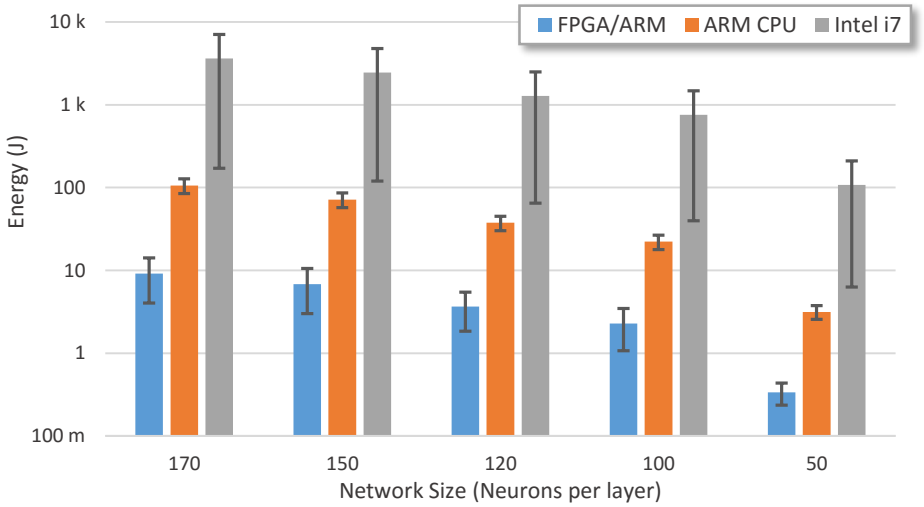


Figure 34. Energy consumption in a simulation of 1000 ms

4.4 SIMULATIONS

The simulations below are based on the proposed algorithm described and implemented in *Section 3*.

4.4.1 Random Network

A simulation of a 50x50 **fully connected** and **feed-forward network** with **random stimulus** over $1000\ ms$, random positive *synapse weights* not bigger than 0.011, random distribution of 10% inhibitory neurons and 90% excitatory neurons is being illustrated in *Figure 35.*

Neuron's spikes over a 50x50 random fully connected network along with the spikes representing the $2{,}500$ neurons and $125{,}000$ synapses over a simulation time of $1\ ms$.

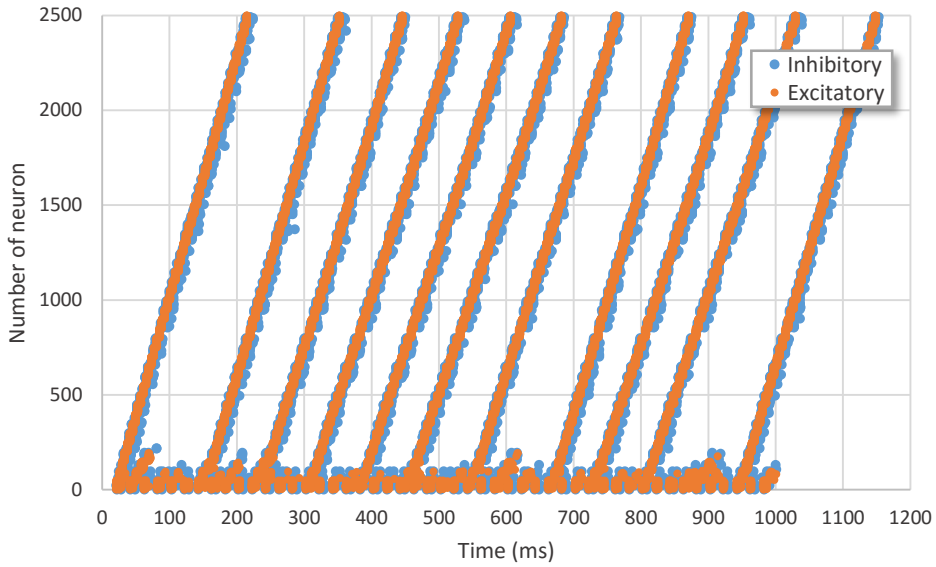


Figure 35. Neuron's spikes over a 50x50 random fully connected network

Due to the feed-forward configuration, the spiking delay from input-to-output layer is non-zero because there is synaptic decay modeled in *Section 1.6.2*, and hence, the more layers a network contains, the longer the spiking delay is. A different representation is illustrated in *Figure 36* with the neuron's spikes from $20\ ms$ to $600\ ms$, where it is seen the uniform delay of neurons belonging to the same layer through time.

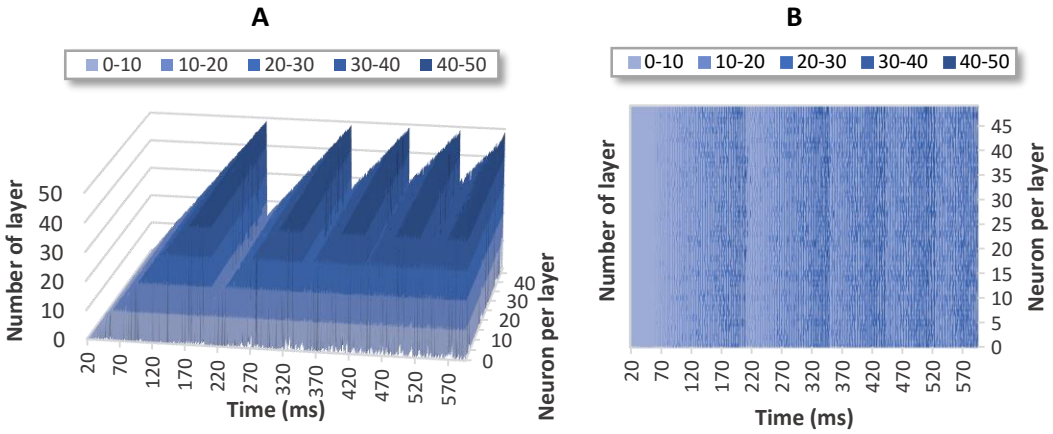


Figure 36. 3D (A) and contour (B) representation of neuron's spikes over a 50x50 random SNN

In *Figure 37,* the model response based on **Izhikevich's equations** described in *Section 1.6.1.3* is being illustrated along with the different spiking delays through the different layers from the previous simulation. Detailed simulation parameters available in *Appendix 7.2.2.*

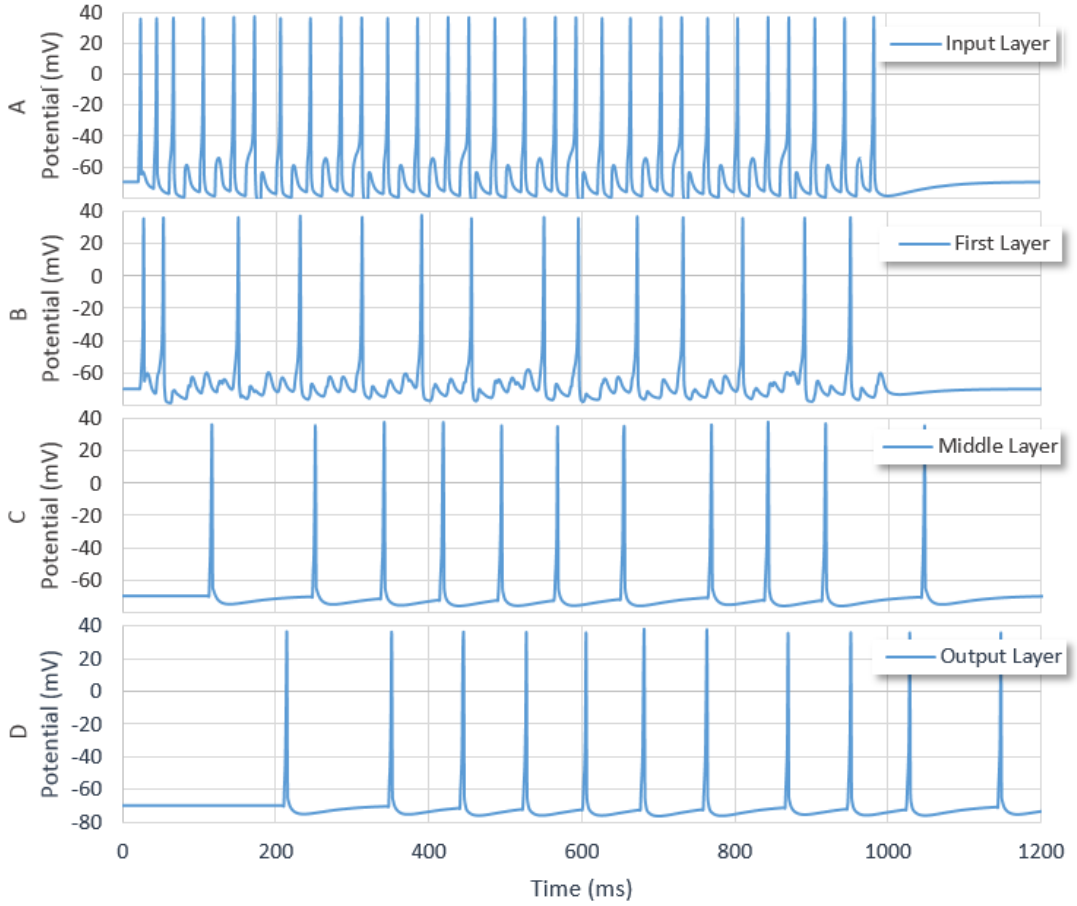


Figure 37. Izhikevich response to neurons in different layers: input, first, middle and output.

4.4.2 Firing Rate Accuracy in STDP Learning Process

The Hebbian learning based on the STDP process described in *Equation 11*, was implemented in a simple 1-4-1 network as shown in *Figure 38* in order to analyze the learning performance behavior to follow a target firing rate response of $30\ \mathrm{Hz}$. A sweep of A_+/A_- from 0.004 to 0.020, along with a τ_+/τ_- swept from 1 up to $18\ \mathrm{ms}$ was performed over the same topology, and the final learning performance in the iteration number 500 was obtained.

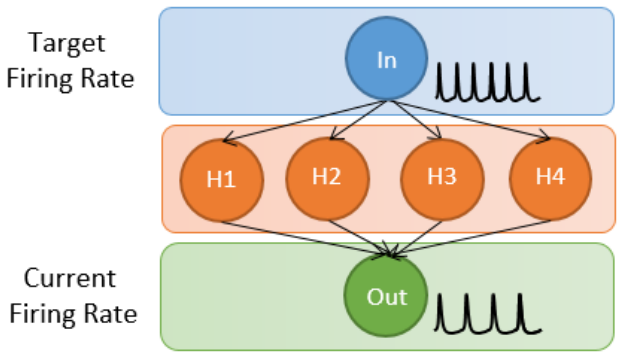


Figure 38. Network topology 1-4-1 for a firing rate input-output follower

Although it might seem that A_+/A_- is directly proportional to the learning rate η, the combination of the exponential time constant τ, together with the STDP amplitude A; they both play a relevant role in order to determine the optimal performance learning, with an expected behavior as it is illustrated in *Figure 39*. Hence, the same topology was simulated over training iterations of 100 ms and 200 ms as in *Figure 39* respectively.

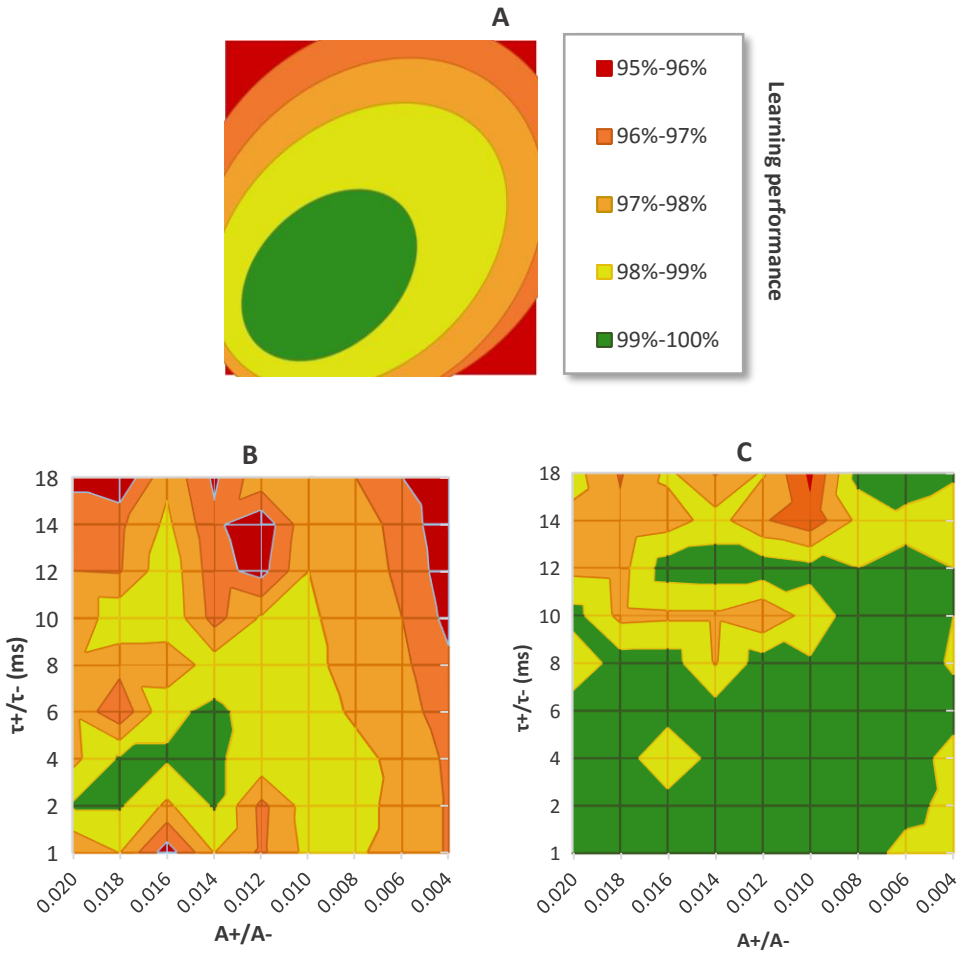


Figure 39. Learning accuracy in a 1-4-1 network (A) expected, (B) 100 ms and (C) 200 ms iterations

Thus, this analysis proves the role being taken by the **Hebbian learning parameters** along with the size of the **training iterations** defined, in order to determine the optimal performance with the **quicker learning response.**

4.4.3 XOR Benchmark

The processing consisting of a **single processing unit** such as perceptron is able to categorize data into two **linear separable** [54] classes. The linear separability for the classical boolean functions can be illustrated in *Figure 40* where the AND/OR functions are able to be linearly

separated into two classes, unlike an **XOR** gate; thus, the relevance of this simple benchmark when it comes to **classification** in neural networks.

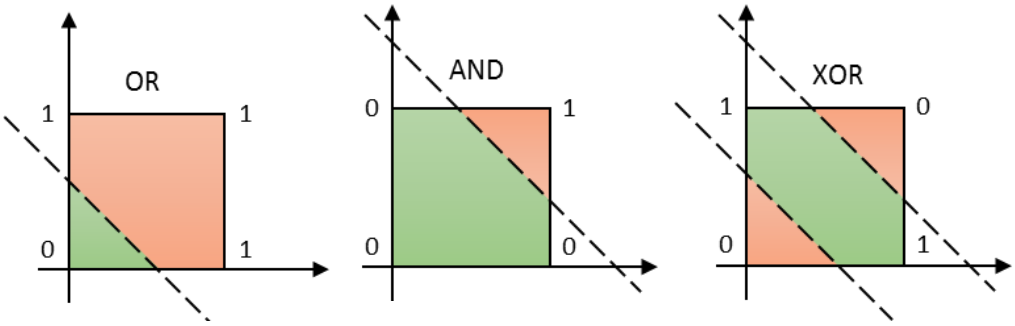


Figure 40. Linear separability of boolean functions

As discussed in *1.9.1,* **temporal coding** is one of the schemes for storing data into the firing times of neurons. For a XOR network, two neurons are required to represent the two inputs, and one for the output. Hence, a **logic '0'** can be represented as a firing in a **given time**, while a **logic '1'** as a **delayed** firing with respect to the logic '0'.

A problem arises when the network tries to differentiate the input patterns with the same values because the pre and post-synaptic delay between them is zero, and so, there is no way to identify if they both are delayed (low '1') or not (logic '0'), against an initial reference point. Hence, the need for an additional input neuron known as **reference,** so that it can indicate the initial time of the pattern to be described in a given **training iteration.**

Type of Neuron	*State*	*Firing Time*
Reference	N/A	0 ms
Input	Low – Logic '0'	0 ms
	High – Logic '1'	5 ms
Output	Low – Logic '0'	7 ms
	High – Logic '1'	13 ms

Table 13. Spike time encoding for reference, input and output neurons

Table 13 describes the timings aimed for the three different type of neurons and their states. The firing timings of two logical patterns are described in *Figure 41* for the three input neurons and the expected output neuron.

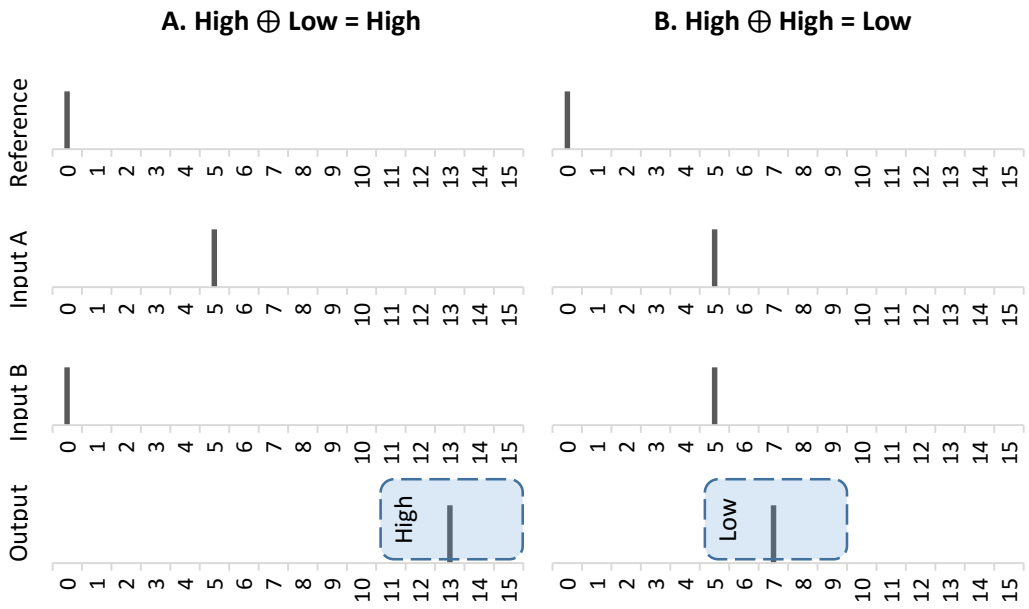


Figure 41. Spike firings for a XOR gate with high output (A) and low (B) output

The learning algorithm ReSuMe discussed in *1.9.2.2.2* was evaluated in a 3-6-1 network topology as represented in *Figure 42* with **random inputs** over **training iterations** of $45\ ms$. The network was generated with a ratio of $20/80$ inhibitory/excitatory neurons and a **decreasing learning rate** as the **current learning increases**. Detailed simulation parameters available in *Appendix 7.2.3.*

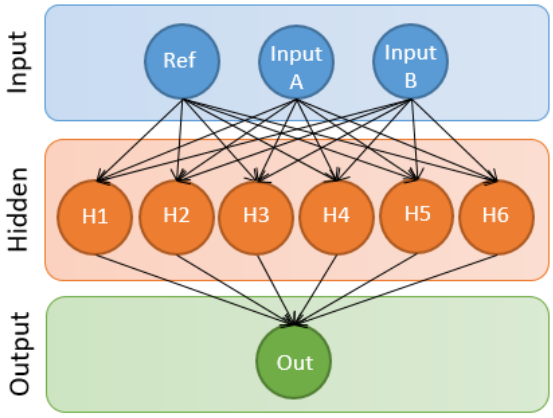


Figure 42. Network topology 3-6-1 for XOR implementation

Thus, *Figure 43.* Learning progress (A) and synapse weight updates (B) of a XOR with a 3-6-1 topology illustrates the learning progress over a simulation of 1500 iterations and the synapse weight updates over five random weights selected from the network evaluated.

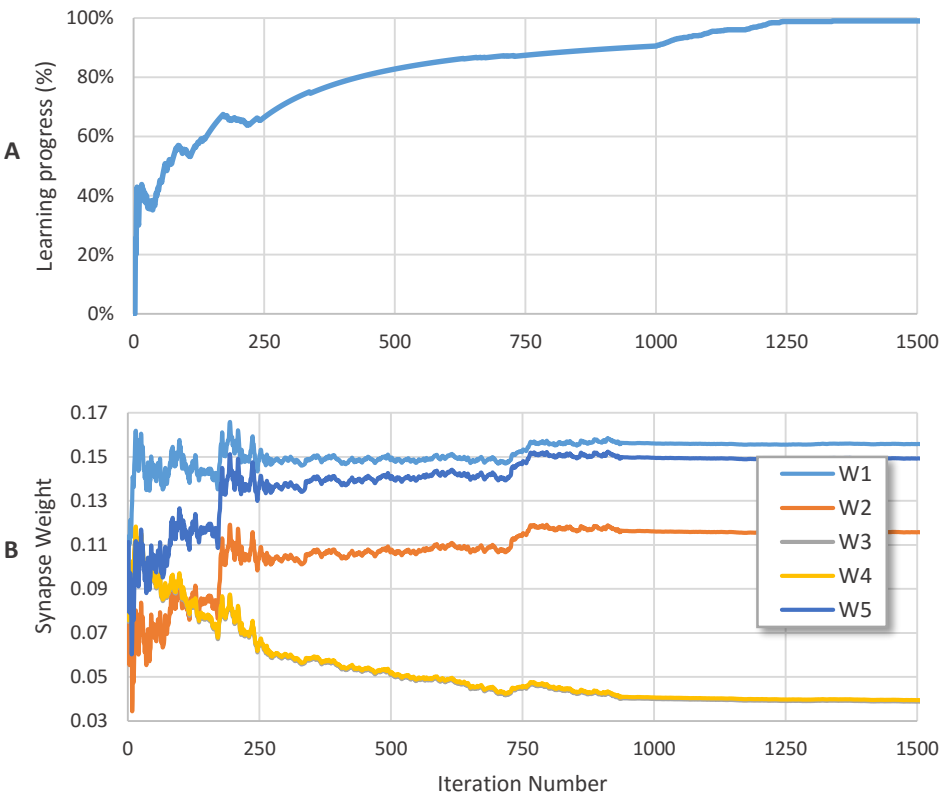


Figure 43. Learning progress (A) and synapse weight updates (B) of a XOR with a 3-6-1 topology

5 CONCLUSIONS AND FUTURE WORK

5.1 CONCLUSIONS

The more complex a neuron model is, the more realistic and more number of biological behaviors can be simulated; consequently, this is a direct impact in the performance of the algorithm required to be implemented for any architecture; thus, a good trade-off of the neuron model selected in terms of complexity and neuron properties described may result in the best results.

High-level synthesis tools nowadays integrate plenty automated techniques for optimization. As a result of this, productivity increases significantly for any simple or complex designs, rather than using native hardware description languages such as VHDL or Verilog. Not before mentioning that although plenty work is done by the tools, better results can be achieved if the source code is modified so that the hardware can take advantage of its structure more easily.

The current FPGA-based implementation is able to simulate more than 25K neurons and 5M synapses with floating and fixed point; resulting from 5 to 7 times faster and using only 0.2% to 2.3% energy instead of using traditional computing solutions. Depending on application requirements, numerical data precision may be varied, and thus, performance results may be increased without any significant impact on the desired results and functionality.

Therefore, hardware-based solutions implemented in SoC devices containing low-power CPUs result in attractive solutions due to the high-performance and low-power consumption that these may achieve, rather than implementing the same solution in a traditional computational using high-end Intel processors.

5.2 FUTURE WORK

The main topic for further exploration proposed **deep learning** as described below.

- **Different network topologies** such as **recurrent networks**, or topologies where the distance between synaptic connections is greater than one may impact the performance latency and feasibility of implementation in a FPGA-based solution. Mainly due to **large data sets**, plenty of **read/write access to memory** are required, thus, this becomes in a key part for an optimal parallelizable solution.

- Impact of different novel algorithms for **spiking neural networks** currently in the literature may result on improved results for **learning purposes.**
- Besides the feasibility of synthesis and optimal implementation results using **different network topologies**, this may lead to improved learning results or being able to solve deep learning applications that with the implemented network configuration is not feasible.
- Feasibility of describing **learning algorithms** together with the neural network in the **FPGA** may lead to improved results when **online learning** is required. This may be scoped by the learning algorithm to use.

6 REFERENCES

[1]. W. Bialek and F. Rieke, “Reliability and information transmission in spiking neurons,” Trends in Neurosciences, vol. 15, no. 11, pp. 428–434, 1992.

[2]. J. C. Moctezuma, J. P. Mcgeehan, and J. L. Nunez-Yanez, “Biologically compatible neural networks with reconfigurable hardware,” Microprocessors and Microsystems, vol. 39, no. 8, pp. 693–703, 2015.

[3]. E. Izhikevich, “Simple model of spiking neurons,” IEEE Trans. Neural Netw. IEEE Transactions on Neural Networks, vol. 14, no. 6, pp. 1569–1572, 2003.

[4]. E. Izhikevich, “Which Model to Use for Cortical Spiking Neurons?,” IEEE Trans. Neural Netw. IEEE Transactions on Neural Networks, vol. 15, no. 5, pp. 1063–1070, 2004.

[5]. J. C. Moctezuma, J. P. Mcgeehan, and J. L. Nunez-Yanez, “Biologically compatible neural networks with reconfigurable hardware,” Microprocessors and Microsystems, vol. 39, no. 8, pp. 693–703, 2015.

[6]. F. Naveros, N. R. Luque, J. A. Garrido, R. R. Carrillo, M. Anguita, and E. Ros, “A Spiking Neural Simulator Integrating Event-Driven and Time-Driven Computation Schemes Using Parallel CPU-GPU Co-Processing: A Case Study,” IEEE Trans. Neural Netw. Learning Syst. IEEE Transactions on Neural Networks and Learning Systems, vol. 26, no. 7, pp. 1567–1574, 2015.

[7]. M. Pietras, “Hardware conversion of neural networks simulation models for neural processing accelerator implemented as FPGA-based SoC,” 2014 24th International Conference on Field Programmable Logic and Applications (FPL), 2014.

[8]. Y. Zhang, J. Nunez-Yanez, J. Mcgeehan, E. Regan, and S. Kelly, “A biophysically accurate floating point somatic neuroprocessor,” 2009 International Conference on Field Programmable Logic and Applications, 2009.

[9]. G. Smaragdos, S. Isaza, M. F. V. Eijk, I. Sourdis, and C. Strydis, “FPGA-based biophysically-meaningful modeling of olivocerebellar neurons,” Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays - FPGA '14, 2014.

[10]. H. Wang and H. Wang, “Improvement of Izhikevich's Neuronal and Neural Network Model,” 2009 International Conference on Information Engineering and Computer Science, 2009.

[11]. D. Thomas and W. Luk, “FPGA Accelerated Simulation of Biologically Plausible Spiking Neural Networks,” 2009 17th IEEE Symposium on Field Programmable Custom Computing Machines, 2009.

[12]. H. Soleimani, A. Ahmadi, M. Bavandpour, A. A. Amirsoleimani, and M. Zwolinski, “A Large Scale Digital Simulation of Spiking Neural Networks (SNN) on Fast SystemC Simulator,” 2012 UKSim 14th International Conference on Computer Modelling and Simulation, 2012.

[13]. K. L. Rice, M. A. Bhuiyan, T. M. Taha, C. N. Vutsinas, and M. C. Smith, “FPGA Implementation of Izhikevich Spiking Neural Networks for Character Recognition,” 2009 International Conference on Reconfigurable Computing and FPGAs, 2009.

[14]. M. A. Bhuiyan, A. Nallamuthu, M. C. Smith, and V. K. Pallipuram, “Optimization and performance study of large-scale biological networks for reconfigurable computing,” 2010 Fourth International Workshop On High-Performance Reconfigurable Computing Technology And Applications (Hprcta), 2010.

[15]. M. Ambroise, T. Levi, Y. Bornat, and S. Saïghi , “Biorealistic spiking neural network on FPGA,” 2013 47th Annual Conference on Information Sciences and Systems (CISS), 2013.

[16]. F. Naveros, N. R. Luque, J. A. Garrido, R. R. Carrillo, M. Anguita, and E. Ros, “A Spiking Neural Simulator Integrating Event-Driven and Time-Driven Computation Schemes Using Parallel CPU-GPU Co-Processing: A Case Study,” IEEE Trans. Neural Netw. Learning Syst. IEEE Transactions on Neural Networks and Learning Systems, vol. 26, no. 7, pp. 1567–1574, 2015.

[17]. J. Rickman, “Roadrunner supercomputer puts research at a newscale,” Jun. 2008, http://www.lanl.gov/news/index.php/fuseaction/home.story/story_id/13602.

[18]. M. Pospischil, M. Toledo-Rodriguez, C. Monier, Z. Piwkowska, T. Bal, Y. Frégnac, H. Markram, and A. Destexhe, “Minimal Hodgkin–Huxley type models for different classes of cortical and thalamic neurons,” Biological Cybernetics Biol Cybern, vol. 99, no. 4-5, pp. 427–441, 2008.

[19]. J, Bateman, “High Performance Computing for Brain Simulation.” April 2016. Final dissertation, University of Bristol, 2016.

[20]. K. Cheung, R. Schultz, W. Luk, “A large scale spiking neural network accelerator for FPGA systems.”

[21]. K. Cheung, R. Schultz, W. Luk, “A parallel spiking neural network simulator.”

[22]. S. Moore, P. Fox, S. Marsh, A. Markettos and A. Mujumdar, “Bluehive — A Field-Programable Custom Computing Machine for Extreme-Scale Real-Time Neural Network Simulation”

[23]. "Neuron structure image", Neuroscientifically Challenged, 2016. [Online]. Available: http://www.neuroscientificallychallenged.com/glossary/dendrite

[24]. “File:SynapseSchematic lines.svg - Wikimedia Commons", Commons.wikimedia.org, 2015. [Online]. Available: https://commons.wikimedia.org/wiki/File:SynapseSchematic_lines.svg#/media/File:SynapseSchematic_en.svg

[25]. C. Hopkins and A. Bass, “Temporal coding of species recognition signals in an electric fish,” Science, vol. 212, no. 4490, pp. 85–87, Mar. 1981.

[26]. B. Meftah, O. Lezoray, and A. Benyettou, “Segmentation and Edge Detection Based on Spiking Neural Network Model,” Neural Process Lett Neural Processing Letters, vol. 32, no. 2, pp. 131–146, 2010.

[27]. W. Jones and A. Wilson (2005) Theha rhythms coordinate hippocampal-prefrontal interactions in a spatial memory task. PLoS Biology, 3, 2187-2199.

[28]. E. Stromatias, “Developing a supervised training algorithm for limited precision feed-forward spiking neural networks” thesis, University of Liverpool, 2011.

[29]. W. Senn and J.-P. Pfister, “Spike-Timing-Dependent Plasticity, Learning Rules,” Encyclopedia of Computational Neuroscience, pp. 1–10, 2014.

[30]. R. Veale and M. Scheutz, “Auditory habituation via spike-timing dependent plasticity in recurrent neural circuits,” 2012 IEEE International Conference on Development and Learning and Epigenetic Robotics (ICDL), 2012.

[31]. B. Guo-qiang and P.Mu-ming, “Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type,” The Journal of Neuroscience, 1998.

[32]. F. Ponulak, “Supervised Learning in Spiking Neural Networks with ReSuMe Method,” dissertation, Poznań University of Technology, 2006.

[33]. S. M. Bohte, J. N. Kok, and H. L. Poutré, “Error-backpropagation in temporally encoded networks of spiking neurons,” Neurocomputing, vol. 48, no. 1-4, pp. 17–37, 2002.

[34]. X. Mu and Y. Zhang, “A New Two-Step Gradient-Based Backpropagation Training Method for Neural Networks,” Advances in Neural Networks - ISNN 2010 Lecture Notes in Computer Science, pp. 95–101, 2010.

[35]. P. Nachtsheim, “A first order adaptive learning rate algorithm for backpropagation networks,” Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN'94).

[36]. F. Ponulak and A. Kasiński, “Supervised Learning in Spiking Neural Networks with ReSuMe: Sequence Learning, Classification, and Spike Shifting,” Neural Computation, vol. 22, no. 2, pp. 467–510, 2010.

[37]. F. Ponulak, “ReSuMe - New Supervised Learning Method for Spiking Neural Networks,” Poznan University of Technology, Technical Report, Poznan, 2005

[38]. Spore and A. Grüning, “Supervised Learning in Multilayer Spiking Neural Networks,” dissertation, University of Surrey, 2012.

[39]. Kasiński and F. Ponulak, “Comparison of Supervised Learning Methods for Spike Time Coding in Spiking Neural Networks,” Int. J. Appl. Math. Comput. Sci., vol. 16, no. 1, pp. 101–113, 2006.

[40]. D. Barber, “Learning in spiking neural assemblies,” Advances in Neural Information Processing Systems 15, MIT Press, Cambridge, MA, pp. 149–156.

[41]. J.P. Pfister, D. Barber and W. Gerstner, “Optimal Hebbian Learning: A Probabilistic Point of View,” ICANN/ICONIP 2003, vol. 2714, pp. 92–98, 2003.

[42]. R. Legenstein, C. Naeger, and W. Maass, “What Can a Neuron Learn with Spike-Timing-Dependent Plasticity?,” Neural Computation, vol. 17, no. 11, pp. 2337–2382, 2005.

[43]. B. Ruf, “Computing and Learning with Spiking Neurons – Theory and Simulations”, thesis, Institute for Theoretical Computer Science, Technische Universitaet Graz, Austria, 1998.

[44]. B. Ruf and M. Schmitt, “Learning temporally encoded patterns in networks of spiking neurons,” Neural Proces. Lett., vol. 5, no. 1, pp. 9–18, 1997.

[45]. S. Areibi G. Lacey and G.W. Taylor, “Deep Learning on FPGAs: Past, Present, and Future,” CoRR, abs/1602.04283, 2016

[46]. LeCun, Y., Bengio, Y., & Hinton, G. (2015, May). Deep learning [Review]. Nature, 521, 436-444.

[47]. “Vivado Design Suite - AXI Reference (UG1037),” Xilinx, Jun-2015. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf. [Accessed: 2016].

[48]. “Vivado Design Suite Tutorial - High-Level Synthesis (UG871),” Xilinx, May-2014. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug871-vivado-high-level-synthesis-tutorial.pdf. [Accessed: 2016].

[49]. “Vivado Design Suite User Guide - High-Level Synthesis,” Xilinx, May-2014. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf. [Accessed: 2016].

[50]. “Zynq-7000 All Programmable SoC Overview (DS190),” Xilinx, November-2015. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf. [Accessed: 2016].

[51]. “SDSoC Development Environment - Xilinx,” 2015. [Online]. Available: http://www.xilinx.com/publications/prod_mktg/sdnet/sdsoc-development-environment-backgrounder.pdf. [Accessed: 2016].

[52]. “ARK | Your Source for Intel® Product Specifications,” ARK Product Launch. [Online]. Available: http://ark.intel.com/. [Accessed: 2016].

[53]. “Software Defined,” SDSoC Development Environment, 2016. [Online]. Available: http://www.xilinx.com/products/design-tools/software-zone/sdsoc.html#buy. [Accessed: 2016].

[54]. D. Elizondo, “The Linear Separability Problem: Some Testing Methods,” IEEE Trans. Neural Netw. IEEE Transactions on Neural Networks, vol. 17, no. 2, pp. 330–344, 2006.

[55]. D. O. Hebb, The Organization of Behavior: A Neuropsychological Theory. New York: Wiley, 1949.

7 APPENDIX

7.1 HLS SOURCE CODE

7.1.1 Top Modules

```
uint1_t hls_snn_izikevich(
            uint1_t state,
            uint32_t p_input[AXI_INPUT_LENGTH],
            uint32_t output_indexes[AXI_POTENTIAL_OUTPUTS],
            hls_stream_64_t& input_stream0,
            hls_stream_64_t& input_stream1,
            hls_stream_64_t& input_stream2,
            hls_stream_64_t& input_stream3,
            hls_stream_64_t& output_stream) {

      #pragma HLS INTERFACE s_axilite port=return bundle=control
      #pragma HLS INTERFACE s_axilite port=state bundle=control
      #pragma HLS INTERFACE s_axilite port=p_input bundle=control
      #pragma HLS INTERFACE s_axilite port=output_indexes
bundle=control
      #pragma HLS INTERFACE axis port=input_stream0
      #pragma HLS INTERFACE axis port=input_stream1
      #pragma HLS INTERFACE axis port=input_stream2
      #pragma HLS INTERFACE axis port=input_stream3
      #pragma HLS INTERFACE axis port=output_stream

      // Process logic
      if (state == STATE_INIT) {
            hls_snn_initialize(input_stream0, output_indexes);
      }
      else {
            hls_snn_process_step(p_input, input_stream0,
input_stream1, input_stream2, input_stream3);
      }

      // Send (v) to output stream
      axis_cp_output_to_stream(output_stream, v_mem,
output_indexes_mem, firings_mem);

      return SUCCESS_OK;
}
```

```
void hls_snn_initialize(
            hls_stream_64_t& input_stream,
            uint32_t output_indexes[AXI_POTENTIAL_OUTPUTS]) {

      // Copy neuron_type and synapse_weights to memory from
stream
      axis_cp_network_to_mem(input_stream, neuron_type_mem);

      // Set output indexes values
      for (int32_t x = 0; x < AXI_POTENTIAL_OUTPUTS; x++) {
            #pragma HLS UNROLL
            output_indexes_mem[x] = output_indexes[x];
```

```
	}

	// Set default values for v's and u's
	for (int32_t x = 0; x < NUMBER_OF_NEURONS; x++) {
		#pragma HLS PIPELINE
		v_mem[x] = -70.0; //resting potential
		u_mem[x] = -14.0; //steady state
	}
	return;
}
```

```
void hls_snn_process_step(
		uint32_t p_input[AXI_INPUT_LENGTH],
		hls_stream_64_t& input_stream0,
		hls_stream_64_t& input_stream1,
		hls_stream_64_t& input_stream2,
		hls_stream_64_t& input_stream3) {

	// Copy inputs (p) from axi-stream to memory
	axis_cp_inputs_to_mem(p_input, p_mem);

	// Process inputs
	snn_process_step(p_mem, neuron_type_mem, synapse_s_mem,
v_mem, u_mem,
				input_stream0, input_stream1,
input_stream2, input_stream3, firings_mem);
	return;
}
```

```
void snn_process_step(
const uint1_t		p[INPUT_SYNAPSES],
const uint1_t neuron_type[NUMBER_OF_LAYERS][NEURONS_PER_LAYER],
	s_dat_t	synapse_s[NUMBER_OF_LAYERS][NEURONS_PER_LAYER],
	vu_dat_t	v[NUMBER_OF_NEURONS],
	vu_dat_t	u[NUMBER_OF_NEURONS],
	hls_stream_64_t& input_stream0,
	hls_stream_64_t& input_stream1,
	hls_stream_64_t& input_stream2,
	hls_stream_64_t& input_stream3,
	ap_uint<AXI_SIZE>firings[AXI_FIRINGS_LENGTH]) {

	// Compute neuron synapses for all neurons
	snn_update_neuron_synapses(
			/* Input  */ v, p,
			/* Output */ synapse_s);

	// Compute matrix of synaptic conductances and apply model
	snn_get_synaptic_conductances_and_izhikevich_model(
			/* Input  */ neuron_type, synapse_s,
			/* Output */ v, u,
			input_stream0, input_stream1, input_stream2,
input_stream3, firings);
	return;
}
```

7.1.2 Processing Blocks

```
void snn_update_neuron_synapses(
const vu_dat_t     v[NUMBER_OF_NEURONS],
const uint1_t      p[INPUT_SYNAPSES],
      s_dat_t      synapse_s[NUMBER_OF_LAYERS][NEURONS_PER_LAYER])
{
      int32_t l, xl, x;

      synapses_layer_updates: for (x = 0, l = 0; l <
NUMBER_OF_LAYERS; l++) for (xl = 0; xl < NEURONS_PER_LAYER; xl++,
x++) {
            #pragma HLS PIPELINE II=1

            s_dat_t synapse_value = synapse_s[l][xl];
            s_dat_t new_synapse = synapse_value *
(s_dat_t)S_DECAY_FACTOR;

            if (l < NUMBER_OF_LAYERS - 1 && v[x] >=
(vu_dat_t)35.0f) // firing input (t-1)
                  synapse_s[l][xl] = new_synapse + (s_dat_t)1.0f;
            else if (l == NUMBER_OF_LAYERS - 1 && p[x-
INPUT_SYNAPSE_OFFSET] == 1) // synaptic input
                  synapse_s[l][xl] = new_synapse + (s_dat_t)1.0f;
            else // only decay
                  synapse_s[l][xl] = new_synapse;
      }
      return;
}
```

```
void snn_get_synaptic_conductances_and_izhikevich_model(
const uint1_t neuron_type[NUMBER_OF_LAYERS][NEURONS_PER_LAYER],
const s_dat_t      synapse_s[NUMBER_OF_LAYERS][NEURONS_PER_LAYER],
      vu_dat_t     v[NUMBER_OF_NEURONS],
      vu_dat_t     u[NUMBER_OF_NEURONS],
      hls_stream_64_t& stream0,
      hls_stream_64_t& stream1,
      hls_stream_64_t& stream2,
      hls_stream_64_t& stream3,
      ap_uint<AXI_SIZE>firings[AXI_FIRINGS_LENGTH]) {

      // Temporal indexes and cache
      uint32_t y, xl, l, x, b;
      uint32_t firings_idx = 0, firings_bit = 0;
      // Cache arrays
      static      s_dat_t     synapse_cache[NEURONS_PER_LAYER];
      static      s_dat_t     synapse_fetch[NEURONS_PER_LAYER];

      PRAGMA_HLS(HLS ARRAY_RESHAPE variable=neuron_type complete
dim=2)
      PRAGMA_HLS(HLS ARRAY_PARTITION variable=synapse_fetch block
factor=SYNAPSE_PARTITION_FACTOR dim=1)
      PRAGMA_HLS(HLS ARRAY_PARTITION variable=synapse_cache block
factor=SYNAPSE_PARTITION_FACTOR dim=1)

      input_synapses_cache: for (y = 0; y < NEURONS_PER_LAYER;
y++) {
            synapse_cache[y] = synapse_s[NUMBER_OF_LAYERS-1][y];
```

```
      }

      synaptic_conductances: for (x = 0, l = 0; l <
NUMBER_OF_LAYERS; l++) for (xl = 0; xl < NEURONS_PER_LAYER; xl++,
x++) {

            PRAGMA_HLS(HLS PIPELINE II=MAX_PIPELINE_THROUGHPUT)

            uint32_t l_pre = (l==0 ? NUMBER_OF_LAYERS-1 : l - 1);
            vu_dat_t sum_g_exh = 0;
            vu_dat_t sum_g_inh = 0;
            s_dat_t current_s = synapse_s[l][xl];
            // Fetch a new 256-bits line
            ap_uint<AXI_WEIGHTS_LINE_BITS> wline =
get_weight_line(stream0, stream1, stream2, stream3);
            uint1_t current_neuron_type = neuron_type[l][xl];

            // Add current neuron synapses to cache pre-fetch
            synapse_fetch[xl] = current_s;

            // Perform sum of synaptic conductances per neuron
            synapses_per_neuron: for (b = 0, y = 0; y <
NEURONS_PER_LAYER; y++, b+= WEIGHT_BITS) {

                  s_dat_t synapse_value = synapse_cache[y];
                  #if PRECISION_TYPE == FIXED_POINT
                  s_dat_t synapse_weight_value = 0;
                  #if WEIGHT_BITS < (SYNAPSE_BITS_FRACTIONAL-
WEIGHT_SCALE_BITS)

      synapse_weight_value.range(SYNAPSE_BITS_FRACTIONAL-
WEIGHT_SCALE_BITS-1, SYNAPSE_BITS_FRACTIONAL-WEIGHT_SCALE_BITS-
WEIGHT_BITS) =
                              wline.range(b + WEIGHT_BITS - 1, b);
                  #else

      synapse_weight_value.range(SYNAPSE_BITS_FRACTIONAL-
WEIGHT_SCALE_BITS-1, 0) =
                              wline.range(b + WEIGHT_BITS - 1, b +
WEIGHT_BITS-(SYNAPSE_BITS_FRACTIONAL-WEIGHT_SCALE_BITS));
                  #endif /* WEIGHT_BITS > 24 */
                  #elif PRECISION_TYPE == FLOATING_POINT
                  ap_uint<WEIGHT_BITS> uweight =
ap_uint<WEIGHT_BITS>(wline.range(b + WEIGHT_BITS - 1, b));
                  s_dat_t synapse_weight_value =
uint32_to_float32(uweight.to_uint());
                  #endif
                  vu_dat_t conductance = synapse_weight_value *
synapse_value;
                  uint1_t neuron_type_value =
neuron_type[l_pre][y];

                  #if PRECISION_TYPE == FLOATING_POINT
                  #pragma HLS RESOURCE variable=sum_g_exh
core=FAddSub_fulldsp latency=8
                  #pragma HLS RESOURCE variable=sum_g_inh
core=FAddSub_fulldsp latency=8
                  #endif
                  if (l == 0 || neuron_type_value ==
EXCITATORY_NEURON)
                        sum_g_exh += conductance;
```

```
                  else
                        sum_g_inh += conductance;
            }

            // Copy synapses pre-fetched from last layer into
cache
            if (xl==NEURONS_PER_LAYER-1) for (y = 0; y <
NEURONS_PER_LAYER; y++) {
                  synapse_cache[y] = synapse_fetch[y];
            }

            // Update neuron states
            snn_update_izikevich_equations_by_neuron(x,
current_neuron_type, sum_g_exh, sum_g_inh, v, u);

            // Save firing
            firings[firings_idx][firings_bit++]= (v[x] >=
(vu_dat_t)35.0f);
            if (firings_bit >= AXI_SIZE) { firings_bit = 0;
firings_idx++; }
      }
      return;
}
```

```
INLINE void snn_update_izhikevich_equations_by_neuron(
      const uint32_t    x,
      const uint1_t     neuron_type,
      const vu_dat_t    g_exh,
      const vu_dat_t    g_inh,
            vu_dat_t    v[NUMBER_OF_NEURONS],
            vu_dat_t    u[NUMBER_OF_NEURONS]) {

      vu_dat_t dv, du;
      vu_dat_t v_t = v[x];
      vu_dat_t u_t = u[x];
      vu_dat_t I = (g_exh * ((vu_dat_t)ES_EXCITATORY - v_t)) +
(g_inh * ((vu_dat_t)ES_INHIBITORY - v_t));
      vu_dat_t IZH_A = (vu_dat_t)IZHIKEVICH_A(neuron_type);
      vu_dat_t IZH_B = (vu_dat_t)IZHIKEVICH_B;
      vu_dat_t IZH_C = (vu_dat_t)IZHIKEVICH_C;
      vu_dat_t IZH_D = (vu_dat_t)IZHIKEVICH_D(neuron_type);

      if (v_t < (vu_dat_t)35.0f) { // Not firing
            // First 0.5 ms
            dv = ((((vu_dat_t)0.04f * v_t) + (vu_dat_t)5.0f) *
v_t) + (vu_dat_t)140.0f - u_t + I;
            du = IZH_A * ((IZH_B * v_t) - u_t);
            v_t = v_t + (dv * ((vu_dat_t)TIMESTEP_MS));
            u_t = u_t + (du * ((vu_dat_t)TIMESTEP_MS));

            // Second 0.5 ms
            dv = ((((vu_dat_t)0.04f * v_t) + (vu_dat_t)5.0f) *
v_t) + (vu_dat_t)140.0f - u_t + I;
            du = IZH_A * ((IZH_B * v_t) - u_t);
            if (v_t < (vu_dat_t)35.0f)
                  v_t = v_t + (dv * ((vu_dat_t)TIMESTEP_MS));
            u_t = u_t + (du * ((vu_dat_t)TIMESTEP_MS));

            // Persist results
            v[x] = v_t > (vu_dat_t)35.0f? (vu_dat_t)35.0f : v_t;
```

```
            u[x] = u_t;
      }
      else { // Firing
            v[x] = IZH_C;
            vu_dat_t new_u = u_t + IZH_D;
            u[x] = new_u;
            #if PRECISION_TYPE == FLOATING_POINT
            #pragma HLS RESOURCE variable=new_u
core=FAddSub_fulldsp latency=8
            #endif
      }
      return;
}
```

7.1.3 AXI Converter Helpers

```
INLINE ap_uint<AXI_WEIGHTS_LINE_BITS> get_weight_line(
            hls_stream_64_t& stream0,
            hls_stream_64_t& stream1,
            hls_stream_64_t& stream2,
            hls_stream_64_t& stream3) {

      ap_uint<AXI_WEIGHTS_LINE_BITS> bits;
      for (uint32_t b = 0; b < AXI_WEIGHTS_LINE_BITS; b+=
AXI_WEIGHTS_THROUGHPUT) {
            // Collect data from input streams
            ap_uint<AXI_WEIGHTS_THROUGHPUT> axi_word;
            axi_word.range( 63,   0) =
ap_uint<AXI_SIZE>(stream0.read().data).range(AXI_SIZE - 1, 0);
            axi_word.range(127,  64) =
ap_uint<AXI_SIZE>(stream1.read().data).range(AXI_SIZE - 1, 0);
            axi_word.range(191, 128) =
ap_uint<AXI_SIZE>(stream2.read().data).range(AXI_SIZE - 1, 0);
            axi_word.range(255, 192) =
ap_uint<AXI_SIZE>(stream3.read().data).range(AXI_SIZE - 1, 0);

            // Store data into cache line(s)
            bits.range(b + (AXI_WEIGHTS_THROUGHPUT) - 1, b) =
axi_word.range(AXI_WEIGHTS_THROUGHPUT - 1, 0);
      }
      return bits;
}
```

```
void axis_cp_network_to_mem(
      hls_stream_64_t& input_stream,
      uint1_t neuron_type[NEURONS_PER_LAYER][NEURONS_PER_LAYER]) {

      int32_t b, y, l, s;
      ap_uint<AXI_SIZE> bits;
      y = 0; l = 0;

      // Store neuron_type from stream
      for (s = 0; s < AXI_NEURON_TYPE_LENGTH; s++) for (b = 0; b <
AXI_SIZE; b++) {
            #pragma HLS PIPELINE II=2
```

```
            // Get 64-bit data from stream
            if (b == 0) {
                  axis64_t data_in = input_stream.read();
                  bits = ap_uint<AXI_SIZE>(data_in.data);
            }

            // Store data into memory
            if (l < NUMBER_OF_LAYERS)
                  neuron_type[l][y] = bits[b];

            // Handle 2-dimensional indices
            if (y < NEURONS_PER_LAYER - 1) { y ++; } else { y = 0;
l++;}
      }
      return;
}
```

```
void axis_cp_inputs_to_mem(
            uint32_t p_input[AXI_INPUT_LENGTH],
            uint1_t p[INPUT_SYNAPSES]) {

      int32_t x, b, stream_id;
      ap_uint<AXIL_SIZE> bits = 0;

      // Get neuron_type stream
      for (stream_id = 0; stream_id < AXI_INPUT_LENGTH;
stream_id++) {
            for (b = 0; b < AXIL_SIZE; b++) {

                  #pragma HLS PIPELINE II=2
                  x = stream_id * AXIL_SIZE + b;

                  // Get new 32-bit data
                  if (b == 0) {
                        bits =
ap_uint<AXIL_SIZE>(p_input[stream_id]);
                  }

                  // Store data into memory
                  if (x < INPUT_SYNAPSES) p[x] = bits[b];
            }
      }
      return;
}
```

```
void axis_cp_output_to_stream(
            hls_stream_64_t& output_stream,
            vu_dat_t    v[NUMBER_OF_NEURONS],
            uint32_t    output_indexes_mem[AXI_POTENTIAL_OUTPUTS],
            ap_uint<AXI_SIZE>firings_mem[AXI_FIRINGS_LENGTH]) {
      // Set output stream
      uint32_t stream_id, x, axi_idx;

      // Potential outputs
      potential_outputs: for (stream_id = 0, x = 0; stream_id <
AXI_POTENTIAL_OUTPUT_LENGTH; stream_id++, x+=2) {
            axis64_t data_out;
            #if PRECISION_TYPE == FIXED_POINT
```

```
            data_out.data =
float32_to_uint64(v[output_indexes_mem[x]].to_float(),
v[output_indexes_mem[x + 1]].to_float());
            #elif PRECISION_TYPE == FLOATING_POINT
            data_out.data =
float32_to_uint64(v[output_indexes_mem[x]], v[output_indexes_mem[x
+ 1]]);
            #endif
            data_out.last = 0;
            output_stream.write(data_out);
      }

      // Set output stream
      firing_outputs: for (stream_id = 0; stream_id <
AXI_FIRINGS_LENGTH; stream_id++) {
            // Store data
            axis64_t data_out;
            data_out.data = firings_mem[stream_id].to_uint64();
            data_out.last = (stream_id >= (AXI_FIRINGS_LENGTH -
1) ? 1 : 0);
            output_stream.write(data_out);
      }
      return;
}
```

7.2 SIMULATION PARAMETERS

7.2.1 Izhikevich Model

Property	*Value*
Simulation time step	1 ms
Neuron model time step	0.5 ms
Excitatory potential E_{exc}	0 mV
Inhibitory potential E_{inh}	−85 mV
Synapse decay constant τ_s	10 ms
Izhikevich parameter a	0.02 (excitatory); 0.10 (inhibitory)
Izhikevich parameter b	0.2
Izhikevich parameter c	−65
Izhikevich parameter d	8 (excitatory); 2 (inhibitory)

7.2.2 Random Network

Property	*Value*
Neurons per layer	50
Number of layers	50

Simulation runtime	1200 ms
Input firing rate	20 Hz
Interconnection probability	100%
Synapse weights	$0 - 0.01$

7.2.3 XOR Benchmark

Property	*Value*
Input neurons	3
Hidden neurons	6
Output neurons	1
STDP amplitude A_+	0.020
STDP amplitude A_-	-0.015
STDP constants τ_+	14 ms
STDP constants τ_-	14 ms
Excitatory neurons	80%
Inhibitory neurons	20%
Training iterations	2000

Learning rate (η) *for a quicker learning performance defined as:*

Progress (%)	*Learning Rate*
$0 - 39\%$	0.14
$40 - 59\%$	0.11
$60 - 69\%$	0.08
$70 - 79\%$	0.05
$80 - 89\%$	0.02
$90 - 99\%$	0.001
$99 - 100\%$	0.000