POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Event driven fully digital neural net

Supervisors

Prof. Maurizio MARTINA

Ph.D. Paolo MOTTO ROS

Candidate

Alberto VENZO

December 2020

Summary

Machine learning and neural net use is increasing for many tasks, which require high computational parallelism and but also high energy efficiency and low power consumption. These requirements become crucial when neural nets need to be integrated in embedded systems, which require also a significant reduction in area occupation. Another main aspect to be considered is that conventional architectures, such as convolutional neural networks, do not take into account the possibility to implement fault tolerant systems in which the produced results, while still satisfying the specifications, do not have necessarily to be deterministic. Furthermore, nowadays new generations of neural networks are emerging with the purpose to reduce the parallelism of data transfer by using an event driven approach. This thesis work analyzes the recent bio-inspired neuronal models and implements one of these, the Izhikevich neuron. The different architectural solutions used include fixed point arithmetic, one arithmetic operator and stochastic computing. The solutions are then synthesized with the 45 nm Nangate Opencell Library and the UMC 65 nm one and compared with each other, in order to evaluate the metrics change with the same technology.

Acknowledgements

I am immensely grateful to many people, who contributed directly and indirectly to this important achievement of my career.
I would like to thank my supervisors, Professor Maurizio Martina and Ph.D. Paolo Motto Ros, for all the support given during the meetings and mail discussions in order to reach an accomplishment of this thesis work.
Many thanks are reserved also to all my university colleagues, in particular Lorenzo Di Lella, Anthony Di Labbio, Andrea Malacrino, Giulio Masinelli, Daniele Volpintesta and Silvio Zanoli. An important part of this achievement was possible thanks to all the talks and comparisons about academic works and personal visions we discussed together.
I would also like to thank all my friends, in particular Leonardo Boldrini, Chiara Bovicelli, Alice Grandi, Mattia Neri, Dante Pio Pallotta, Matteo Tontodonati, Oana Vatamanu and Alessandro Vitelli, for all the understanding and the time spent together.
Eventually, I would express my sincere gratitude to my family and all its members. Particular thanks are due to my parents, who gave me the opportunity to reach this achievement.

Alberto Venzo

Table of Contents

List of Tables

List of Figures

Chapter 1

Dynamic neuronal models

1.1 Introduction

Neural networks are improving each day, and one trend pursued is to reduce computational operations in order to maintain a good degree of neural network complexity without an exceeding of energy consumption. This issue is relevant mostly with Deep Neural Networks (DNNs), which have the best performance but also the worst energy cost. A first try to reduce the impact of energy consumption in these neural networks is proposed in [1]. It is shown that the most energy consumption happens when data are transferred from the main memory to the processing unit, which can be a CPU or a GPU. As a consequence, the first way adopted to improve energy efficiency is to reduce data movements algorithmically. This approach can be achieved by working on hyperparameters, such as the number of layers, the number of filters in each layer, width and height of the filter.

A far better way to resolve energy consumption issue is to implement all MAC (Multiply and ACcumulate) operations in hardware and design a PCB with the amount of DRAM required. A commercial solution, the Tensor Processing Unit (TPU) is already present and heavily used in datacenters, so this unit is the state-of-art computing system for training DNNs. Although TPUs are more energy efficient than any CPU or GPU and remain an excellent choice for training and inference, they still remain unsuitable for embedded solutions.

The increasing demand of low cost machine learning is leading to reconsider the concept of neural networks itself, which should be see not only as a pure mathematical model, but also as the natural derivation of biological neurons. This approach is leading to new biological plausible, low power and low cost neural network.

1.2 Important biological plausible models

The research of biological plausible is far from recent, but only a decade ago an efficient way for neuron patter simulation is found, when spiking neural networks study becomes more interesting for practical applications.

1.2.1 The Hodgking-Huxley neuronal model

The first biological plausible model seen is the Hodgkin-Huxley one [2]. Experiments carried by these two researchers leads to one of the first electrical network to describe the behavior of the membrane current of a nerve. This network is the one of figure 1.1 and reports all the electrical components involved in the analysis. The characterization of these components is described in the mathematical model proposed by Hodgkin and Huxley, which is discussed later. The most important physical quantity is the current I, which represents the overall movement of electrical ions inside the neuron. Current I is mostly composed by I_{Na} and I_K, which correspond to sodium and potassium ion currents, respectively. Other contributions to current I are due to chloride and less important ions and are treated as one leakage current I_l.

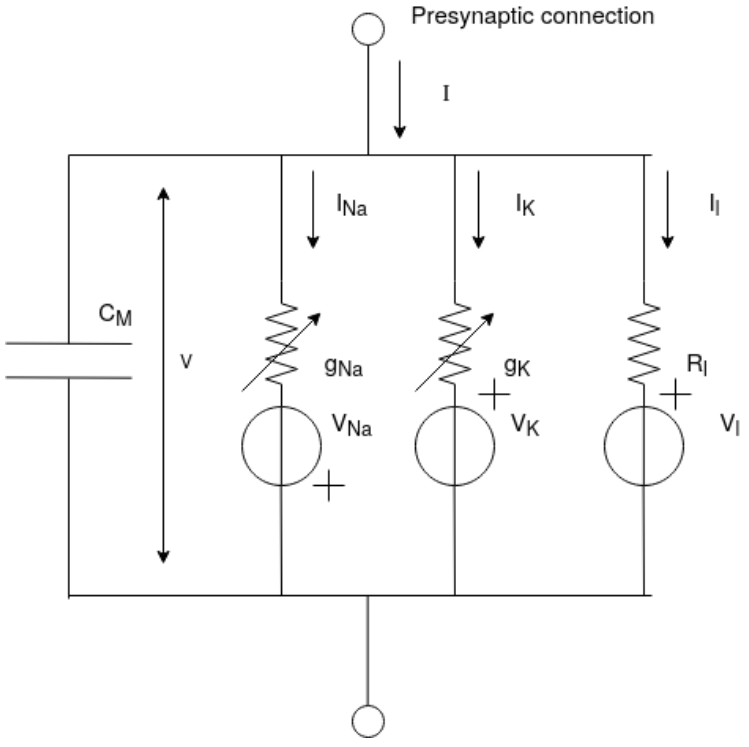

Figure 1.1: Electrical schematic of the Hodgking Huxley neuron.

The topology of the electrical schematic in figure 1.1 allows to formulate simple characteristic relations between the membrane voltage V and the ion currents. The parameters g_{Na} and g_K depend from time and the membrane voltage. The other parameters seem to be characteristic properties of the neuron and are treated as constants. A first qualitative analysis states that a reduce of membrane voltage, which is called depolarization, increases the sodium and potassium conductance. The former increases faster than the latter. This dependence on the membrane voltage shows that the internal electric field, which is generated by the orientation of molecules with an electrical dipole, modifies significantly the flow of the main ion currents. Sodium conductance depends more directly on the number of molecules involved in the change of the total electric dipole. The molecules considered must be related to sodium ions inside the membrane but not to the ones outside the membrane. Since the dependence of sodium conductance from the membrane potential is stronger than the dependence of potassium conductance and, with some approximations, it can be explained on its own, a quantitative analysis can be made. With these considerations, the Bolzmann principle is used:

$$\frac{P_i}{P_o} = e^{\frac{w+ze(V+V_r)}{kT}} \tag{1.1}$$

where P_i is the fraction of molecules inside the membrane and P_o the proportion of molecules outside the membrane. The value w is the work needed to carry one molecule from the inside of the membrane to the outside of it when the membrane potential is in its resting value. The value k is the Bolzmann constant and T is the absolute temperature. Since the proportions must satisfy also the (1.2):

$$P_i + P_o = 1 \tag{1.2}$$

so it is found that:

$$P_i + P_o = P_i\left(1 + \frac{P_o}{P_i}\right) = 1 \Rightarrow P_i = \frac{1}{1 + \frac{P_o}{P_i}} \tag{1.3}$$

Using the (1.2) in the final step of the (1.3) it is found that:

$$P_i = \frac{1}{1 + e^{-\frac{w+ze(V+V_r)}{kT}}} \tag{1.4}$$

If the membrane polarization is sufficiently high, which can be expressed by the condition (1.5):

$$ze(V + V_r) \gg 3kT + \frac{3w}{kT} \tag{1.5}$$

Then the (1.4) becomes:

$$P_i \approx e^{\frac{w+ze(V+V_r)}{kT}} \tag{1.6}$$

As a consequence, it is found that the proportion of sodium molecules inside the membrane is related to the membrane voltage with an exponential function. With other considerations, it can be assumed that the relation between the sodium conductance and the membrane voltage must present the same characteristic.

From figure 1.1 the electrical characteristics can be derived:

$$I = C_M \frac{dV}{dt} + I_i \tag{1.7}$$

where $I_i = I_{Na} + I_K + I_\ell$ is the total ionic current.

$$I_{Na} = g_{Na}(V - V_{Na}) \tag{1.8}$$

$$I_K = g_K(V - V_K) \tag{1.9}$$

$$\frac{I_\ell}{R_l} = V - V_\ell \tag{1.10}$$

where V_{Na},V_K and V_ℓ are the potential voltage of each current branch at the resting potential of V.

Potassium conductance. From the experimental results found for the potassium conductance the following assumption is made: g_k is proportional to the fourth power of a variable, which variable depends on a first order equation. In a more formal way:

$$g_K \propto (1 - e^{-t})^4 \tag{1.11}$$

during the de-polarization phase and:

$$g_K \propto e^{(-4t)} \tag{1.12}$$

during the re-polarization of the membrane. A possible empirical solution can be the (1.13):

$$\begin{cases} g_K = \overline{g_K} n^4 \\ \frac{dn}{dt} = \alpha_n(1 - n) - \beta_n n \end{cases} \tag{1.13}$$

where $\overline{g_K}$ is a constant and α_n and β_n depends from the membrane voltage. These equations are physically meaningful if it is assumed that only potassium ions can cross the membrane when a particular region of the membrane is occupied by four similar particles. The variable n represents the fraction of particles inside the membrane, whereas 1-n represents the proportion of the molecules outside the membrane. α_n and β_n are rate constants and represents, respectively, the rate

transfer from the inside of the membrane to the outside and the transfer rate in the opposite direction.

The second of the (1.13) is a first order differential equation whose solution is an exponential of time. The change of the values of α_n and β_n can be considered instant because it is assumed that these parameters do not depend from time. With respect to this consideration, if the membrane voltage is considered at its resting state, which means that $V = 0$, then the resting value of n can be expressed by the (1.14):

$$n_0 = \frac{\alpha_{n0}}{\alpha_{n0} + \beta_{n0}} \tag{1.14}$$

The values α_{n0} and β_{n0} are evaluated so that the solution found may fit the experimental results found for the potassium conductance [2]. The solution of the differential equation in (1.13) that can satisfy the following boundary condition: (1.15):

$$n(t_0) = n(0) = n_0 \tag{1.15}$$

is then:

$$n(t) = n_\infty - (n_\infty - n_0)e^{-\frac{t}{\tau_n}} \tag{1.16}$$

where:

$$n_\infty = \frac{\alpha_n}{\alpha_n + \beta_n} \tag{1.17}$$

and:

$$\tau_n = \frac{1}{\alpha_n + \beta_n} \tag{1.18}$$

With respect to the solution for the parameter n, the sodium conductance equation in (1.13) can be represented also in a more appropriate form in order to examine the similarities with the (1.16):

$$g_K = \left(\sqrt[4]{g_{K_\infty}} - \left(\sqrt[4]{g_{K_\infty}} - \sqrt[4]{g_{K_0}} \right) \cdot e^{\frac{-t}{\tau_n}} \right)^4 \tag{1.19}$$

where g_{K_∞} is the potassium conductance value at the end of the transition and g_{K_0} is the same variable at the beginning. For each value of the membrane voltage analyzed, the parameter τ_n is derived so that the curved generated could explain the experimental results properly.

In order to derive the rate constants from experimental results, the (1.17) and the (1.18) can be rearranged in the following forms:

$$\alpha_n = \frac{n_\infty}{\tau_n} \tag{1.20}$$

$$\beta_n = \frac{1}{\tau_n}\left(1 - n_\infty\right) \tag{1.21}$$

since the time constant was already chosen to fit the experimental results with the (1.19).

The assumption made in order to use the (1.20) and the (1.21) is that the membrane voltage changes instantly from the resting value to a new steady state. Since in the real dynamics the membrane voltage changes during all the observation period, a relation between the rate constants and membrane voltage must be derived. From experimental data the following formulas can be derived:

$$\alpha_n = \frac{0.01\Big(10 + V\Big)}{e^{\frac{10+V}{10}} - 1} \tag{1.22}$$

$$\beta_n = 0.125e^{\frac{V}{80}} \tag{1.23}$$

The values of the constants depends on the measure unit adopted, which are millivolts for the membrane voltage and the reverse of milliseconds for the rate constants. Although more complex formulas could have been chosen, the (1.22) and the (1.23) are the simplest ones "which gave a reasonable fit" ([2], page 510).

Sodium conductance. The behaviour of sodium conductance can be expressed as a function, which contains two parameters obeying a first order differential equation each:

$$g_{Na} = m^3 h \overline{g}_{Na} \tag{1.24}$$

where $\overline{g}_{Na}$ is a constant,

$$\frac{dm}{dt} = \alpha_m(1 - m) - \beta_m m \tag{1.25}$$

$$\frac{dh}{dt} = \alpha_h(1 - h) - \beta_h h \tag{1.26}$$

These equations can have a physical meaning: m is the proportion of activating molecules inside the nerve membrane, and h is the proportion of inactivating molecules inside it. As a consequence, the values (1-m) and (1-h) represents the proportion of activating and inactivating molecules, respectively, outside the nerve membrane. The parameters α_m, α_h, β_m and β_h represent the transfer rate constants of each proportion. The solution of the (1.24) and (1.25) are the ones of a first order differential equation, while the (1.26) derives directly from these. In a way similar to the derivation of the potassium transfer rate constants, the dependence of sodium conductance parameters α_h, β_h, α_h and β_h with respect to the membrane voltage can be derived from the experimental results:

$$\alpha_m = \frac{0.1(V + 25)}{e^{\frac{V+25}{10}} - 1} \tag{1.27}$$

$$\beta_m = 4e^{\frac{V}{18}} \tag{1.28}$$

$$\alpha_h = 0.07e^{\frac{V}{20}} \tag{1.29}$$

$$\beta_h = \frac{1}{e^{\frac{V+30}{10}} + 1} \tag{1.30}$$

In order to summarize, since the overall current is generally an external input given by other neurons or sources, the (1.7) can be rearranged so that:

$$\frac{dV}{dt} = \frac{1}{C_M}\Big(I - I_i\Big) = f\big(V, I\big) \tag{1.31}$$

which can be integrated with a proper numerical method [3]. The parameters of potassium and sodium conductance, as well as the other time-dependent variables in the latter equations, must be integrated first in order to assure the correct behaviour of the membrane voltage.

The Hodgkin-Huxley model has a total of 10 equations which must be evaluated each integration steps, with an estimation of around 1200 floating point operations [4].

1.2.2 The Fitzhug-Nagumo model

Although the Hodgkin-Huxley model has biological meaningful variables, such as the potassium and sodium conductance, it is very expensive in terms of arithmetic operations. An attempt to reduce this complexity is done by the Fitzhug-Nagumo model [5] [6], whose formulas have been rearranged by Izhikevich in the following form [4]:

$$\begin{cases} v' = a + bv + cv^2 + dv^3 - u \\ u' = \epsilon(ev - u) \end{cases} \tag{1.32}$$

The following model requires less than 100 floating point operations for one integration step, so it is more suitable for low cost implementations. It has only two non linear terms and can exhibit most of the biological neuron spiking patterns [4], although not all.

1.2.3 The Izhikevich model proposal

Besides all other models presented before, the model that is analysed with more accuracy is the Izhikevich one [7]. This model presents several interesting features. First of all, it is biological plausible, so it presents a spiking behaviour at the neuron output. Furthermore, this model can exhibit several spiking patterns with the same differential equations [4]. The last positive aspect is that the Izhikevich model require far less floating point operations with respected to the more complex models described previously.

The Izhikevich neural model description consist of two differential equations:

$$\begin{cases} v' = 0.04v^2 + 5v + 140 - u + I \\ u' = a(bv - u) \end{cases} \tag{1.33}$$

The two differential equations in (1.33) represent the membrane potential v and the recover variable u, which is used in order to give stability to the variable v and refractoriness. The reset conditions are:

$$v \geq 30\ mV \Rightarrow \begin{cases} v \leftarrow c \\ u \leftarrow u + d \end{cases} \tag{1.34}$$

Chapter 2

Stochastic computing compendium

2.1 Introduction

Nowadays, the requirement of less complex computational units, combined with high error tolerance in the design specification, has lead to the rediscovery of stochastic computing [8][9][10][11], which represents values as probabilities instead of a deterministic symbol or number. Stochastic computing gives the designers a new challenge to implements low complex computational units for non conventional neural network implementations.

There are several ways to represent numbers with a finite number of bits. The most commons are the positional representation, which includes integer and fixed-point formats, the floating point one and its less accurate format called posit. (cite references). There are other ways to represent numbers, though. Here a briefing of the positional representation is presented, so that the stochastic representations can be introduced, with their characteristics and differences with respect to conventional number representation techniques.

2.2 The positional representation

In arithmetic units the architecture depends strongly on the number representation. The majority of digital computational units use the positional representation [12], in which every digit is a power of 2. The position of each digit related to the others determines the real value the digit represents. For example, the number 1010_2 represents the value:

$$1010_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 10_{10}$$

Generally, an integer binary number B with n bits represents the value:

$$B_{integer} = \sum_{k=0}^{n-1} b_k \times 2^k \tag{2.1}$$

where $b_k \in \{0,1\}$.

Fixed point numbers use the same form, but they use also negative powers of 2 in order to represent numbers with fractional parts:

$$B_{fixed} = \sum_{k=-m}^{n-1} b_k \times 2^k \tag{2.2}$$

where -m is the number of fractional bits and n is now the number of integer bits. The overall number of bits are now m + n.

2.3 Stochastic representations

A different way to represent a number is using a stochastic approach [9], in which numbers are represented as probabilities. There are several ways to represents probabilities and the most important ones are described further.

2.3.1 Unipolar representation

The simplest way to define a stochastic number is using a binary stream, where the number of 'ones' over the number of bits of the streams represents the probability of having a logic '1'. A stream used in this way can be seen as an instance of a Bernoulli sequence of parameter p [13]. For example, the number 10 can be expressed as a binary stream with 10 bits equal to logic '1'. With a precision 4 bits, a stream of $2^4 = 16$ bits must be generated, so the parameter p of the Bernoulli sequence is equal to:

$$p = \frac{10}{2^4} = \frac{10}{16}$$

Generally, if the initial number representation requires n bits, then the number of stream bits must be at least 2^n. This aspect is very important, because a linear increase in the number of bits in the positional representation corresponds to an exponential increase in the number of stream bits in the stochastic one and, consequently, in the number of clock cycles required. Precision can be easily traded off for less computational time by just reducing the number of stream bits generated

or taken into account. For example, the previous probability can be represented also as:

$$p = \frac{10}{16} = \frac{5}{8} = \frac{5}{2^3}$$

which is in this case the exact value of the previous probability with a precision of 3 bits.

As it can be seen, in order to obtain a probability, the initial value is normalized with respect to the maximum value it can reach. For an integer number x in the range [0,2^N], with N the number of bits of the deterministic number, is chosen, then the probability represents the following normalized part:

$$p_x = P(X = 1) = \frac{x}{2^N} \tag{2.3}$$

It is possible to avoid this "implicit" normalization by considering normalized values in the range of [0,1], so only fractional numbers. The maximum value now is 1 and the (2.3) becomes:

$$p_x = \frac{x}{2^0} = x \tag{2.4}$$

The (2.4) shows clearly that a fractional number corresponds directly to a probability. This property simplifies the implementation of multistage stochastic processing, where multiple elementary operations must be performed, such as sums or products.

2.3.2 Single line bipolar representation

Signed numbers can be normalized in the range [0,1] and represented as probabilities, too. The most common way to do that is to use the single-line bipolar representation [14]. In this case, the minimum probability represents the most negative number used, whereas the maximum probability stands for the maximum positive value mapped. Generally, given a number $x \in [-2^{N-1},2^{N-1}]$, where N is the number of bits used to represent the variable, the transformation given by Gaines is:

$$p_x = p(X = 1) = \frac{x}{2^N} + \frac{1}{2} \tag{2.5}$$

In order to obtain the deterministic value or, in the real case, a good approximation of it, the formula to be used is derived from the (2.5):

$$\frac{x}{2^{N-1}} = 2 \cdot p_x - 1 \tag{2.6}$$

If the deterministic numbers are normalized in the range [−1,1], the (2.6) becomes:

$$x = 2 \cdot p_x - 1 \tag{2.7}$$

The latter representation is the most suitable to make stochastic computations, since it avoids implicit normalization as well as the unipolar case.

The bipolar notation must be used in all the cases where mixed signed and unsigned operations must be performed. In some cases, such as the subtraction $1 - p_x$, the unipolar representation can still be used, but generally the bipolar is preferred since it allows to perform all elementary arithmetic operations directly, or nearly.

2.4 Stochastic computational elements

2.4.1 Digital to probabilistic converter

The digital to probabilistic converters (DPC) is the component which associate a probability to a deterministic number, which is given with a positional representation. The general schematic representation is reported in figure 2.1: the X represents a deterministic input, while Y indicates a uniform stochastic variable. Both X and Y have a bit-width equal to N, while the output Z one bit large.

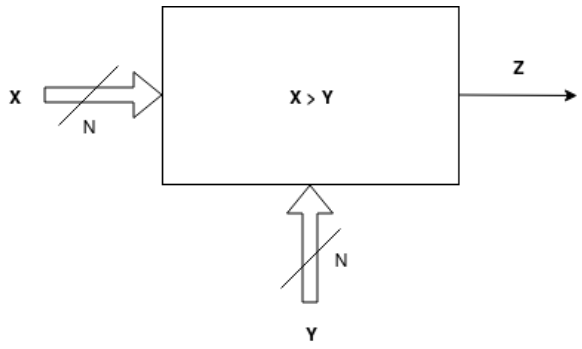

Figure 2.1: General schematic of the digital to probabilistic converter.

The DPC generates a bit-stream, which has the following property: the number of ones in the stream over the all bits generated corresponds to the probability associated to the deterministic number. In fact, for a stochastic unipolar representation, if all possible values are generated by the stochastic variable Y, which are 2^N, the number of ones generated are exactly the unsigned value of X. Formally it is:

$$P(Z = 1) = P(X > Y) = \frac{X}{2^N} \tag{2.8}$$

If Y is purely random, which means that generated numbers can occur more than once in a period of two, the number of ones generated can be not exactly equal to the unsigned values of X due to the generation of the stochastic variable Y. The latter consideration is crucial when an accurate analysis of stochastic-induced error has to be performed. This logic block can be directly implemented in hardware,

although there can be a significant increase in area if the number of bits required are too many. A lower-cost implementation of the DPC, proposed by Brown-Card [8] is shown in figure 2.2.

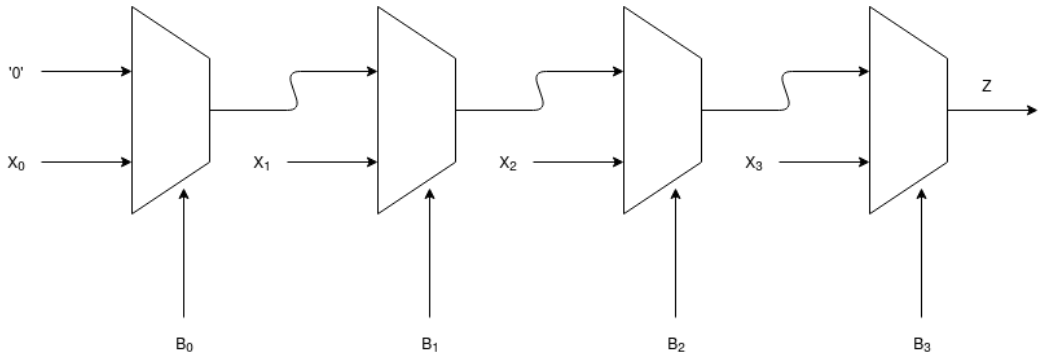

Figure 2.2: Mux implementation for a digital to probalistic converter. In this case, N = 4.

Each bit of the signal X is sent to the second port of a mux, and the output of each mux is the first input of the following one. The output of the most significant mux represents the current value of the stochastic bit-stream. The stochastic signals $B_i, i \in \{0,1,2,\ldots N-1\}$ are Bernoulli variables with parameter $p = \frac{1}{2}$, which means that the values '0' and '1' have the same probability to occur [13]. The (2.8) works for unsigned numbers and the probability associated to it is in the unipolar format. If the deterministic number uses the 2's complement scheme, then the correct representation to use is the single line bipolar. The most simple way to convert the number is by inverting the most significant number, so that all negative values are associated with the smaller probabilities. In this case, the new DPC is the one in figure

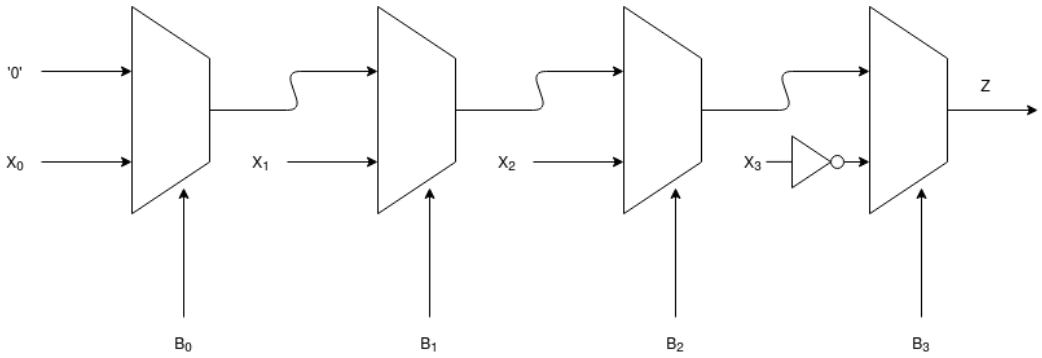

Figure 2.3: Bipolar implementation of a mux DPC.

2.4.2 Linear feedback shift registers

The implementation of a stochastic architecture requires the generation of random numbers or pseudo random numbers. Random numbers can be generated in different ways by using unstable components, such as ring oscillators [15] or diodes [14], although it is not a practical approach if the goal is to design a fully digital system. On the contrary, pseudorandom numbers can be generated by using logical cells and/or arithmetic componets only. A typical example is the generation of pseudorandom numbers using the standard C function [15], which uses the following equation:

$$N_{next} = N_{current} \times 32'h343FD + 32'h269EC3 \tag{2.9}$$

where the constants are 32-bit numbers represented in hexadecimal representation. The 16 most significant bits are considered pseudorandom sequences. The requirements for this implementation is at least one 32-bit adder and a 32-bit multiplier, which is unacceptable for a low cost design.

An alternative way to generate pseudorandom sequences is by using Linear Feedback Shift Registers (LFSR). The general schematic of an LFSR is reported in figure 2.4.

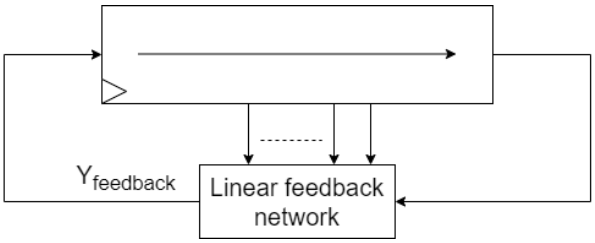

Figure 2.4: General schematic of a linear feedback shift register.

The linear feedback network is composed by two input XOR gates and is called feedback polynomial. Since the exclusive or function represents also a direct sum, the following notation is generally used to represent the feedback polynomial:

$$Y_{feedback} = X_0(t+1) = \sum_{i=0}^{N-1} b_i X_i + 1 \tag{2.10}$$

where b_i can be '0' or '1', depending on the the feedback polynomial chosen. The bits of the shift register can be seen as the state variables of a finite state machine. The number of possible states depends on the feedback polynomial, and each state can be interpreted as a pseudorandom number [14], since the sequence of the state machine is a cyclic graph, where each transition is not correlated with a deterministic

variable coding. The only correlation is the feedback polynomial, which determies the pseudorandom nature of the LFSR. The best feedback polynomial are the ones which generates all the possible states with N bits and are called maximal polynomials. In this particular case, the cyclic graph contains all the N possible states.

Random number generation is crucial in order to have good bit streams that can be used for further evaluations [16][17]. One major issue is that the LFSR used as random number generators can occupy a signficant amount of area, as it can be seen from the synthesis results reported further. This problem can be solved by sharing the same resource for parallel and not related bit streams, such as the ones belonging to different neurons. In order to share the same LFSR for correlated bit streams without a significant increase of the correlation-induced error, the circular shifting technique can be adopted [18]. Another useful way to reduce the number of LFSRs involved is to reuse the same bit stream multiple times. This goal can be achieved by using duplicating techniques [19][20] or proper delay elements, such as D-Flip Flops [21].

2.4.3 Basic stochastic arithmetic blocks

The main advantage of stochastic computing lies in the logic used to perform arithmetic operations. While in deterministic arithmetic logic functions with several gates must be implemented in order to process operands, the complexity in stochastic computing is overwhelmingly small, which can be exploited for very low cost implementations. Another positive aspect is that, in some cases, it is possible to process numbers represented differently with the same arithmetic block, which allows to save even more area to the design. For example, a stochastic number in bipolar notation can be processed and the resulting output can be interpreted as an unipolar stochastic value.

The basic arithmetic blocks are reported in figure 2.5 whereas the basic operation performed by these blocks are shown in table 2.1 [22].

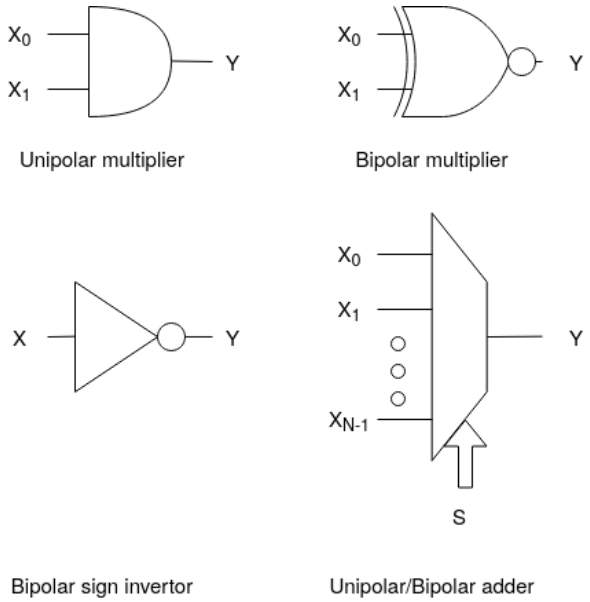

Figure 2.5: Basic stochastic arithmetic elements.

Combinational component	Unipolar Output	Bipolar Output
AND	$Y = X_0 \cdot X_1$	$Y = \frac{X_1X_2+X_1+X_2-1}{2}$
XOR	$Y = 1 - X_1 - X_2 + 2X_1 \cdot X_2$	$Y = X_1 \cdot X_2$
NOT	$Y = 1 - X$	$Y = -X$
MUX	$\sum_{i=0}^{N-1} p(S = i)X_i$	$\sum_{i=0}^{N-1} p(S = i)X_i$

Table 2.1: Stochastic functions implemented by each combinational block. Hybrid computations are not considered.

If the selector signal S has uniform distribution of its value, which means that each selection bit is a Bernoulli variable of parameter $p = \frac{1}{2}$ and the selection bits are uncorrelated from each other, the multiplexer performs a scaled sum of the input probabilities, which can be in bipolar or unipolar format.

Stochastic computing may present very high error and poor accuracy if the bit stream are correlated or several sequential operations are performed [23][24][25]. Error and variance of the stochastic processes can be improved with custom architectural solutions and analysis [26][27][28]. Latency can be reduced by using unary positional computing [29] and its derivations [30][31]. For large fully stochastic

systems, the polysynchronous clocking approach can be used to reduce the clock tree design, with a significant reduce of area, power and energy consumption [32].

2.4.4 State machines for complex functions

Complex functions such as hyperbolic tangent and exponential can be implemented in stochastic computing with the use of simple state machines [8][33]. The working principle is based on the saturated counting: given a general state machine with N states, the possible transitions occur only between the adjacent ones, giving generally a chain graph as the one on figure 2.6 for the case of one stochastic input only.

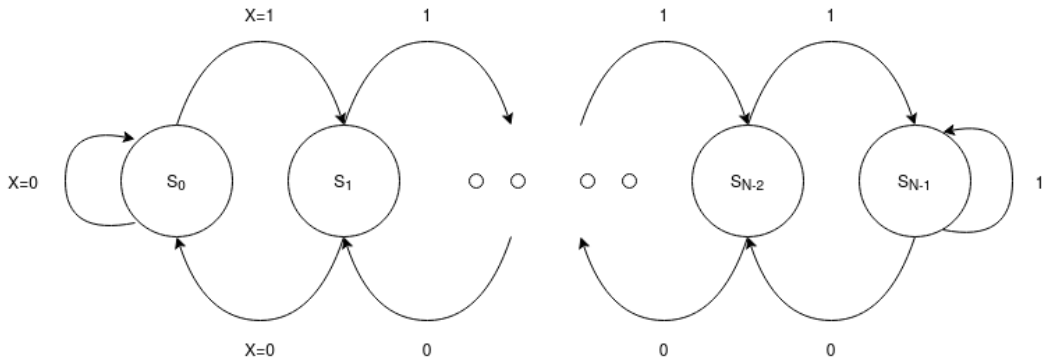

Figure 2.6: General finite state machine graph to generate arbitrary one input stochastic funtion.

A transition between first state and the last one, which would correspond to a closed loop state machine as the one in figure 2.7, is not allowed.

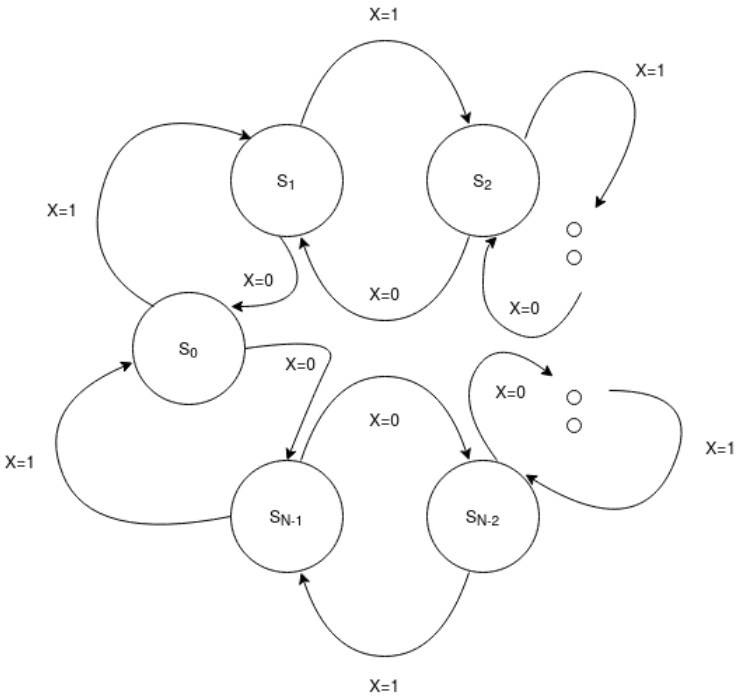

Figure 2.7: Closed loop FSM graph.

Stochastic hyperbolic tangent If the state machine considered is the one in figure 2.6 and the output Y is '1' when the state index is $i \in [N/2, N-1]$ and '0' otherwise, the stochastic output is the stochastic hyperbolic tangent, or $stanh(N, X)$, which is approximately $tanh\left(\frac{N}{2}x\right)$ [8].

Estimation of a stochastic value

The last part of a stochastic computation is the estimation of the output stochastic stream. For the unipolar representation a simple saturating up counter can be used, while a saturating up-down counter is needed for the bipolar representation which is actually a particular use of the ADaptive DIgital Element (ADDIE), described by Gaines [14].

2.5 Applications of stochastic computing

Stochastic computing is successfully used in several fields, in which high parallel and fault tolerant computation are required. The main applications for which stochastic computing can be good alternative are:

- Machine learning and neural network accelerators [34][35][36][37][38][39][40][41][42][43][44][45][46][47][48][49][50][51][52][53][54][55][56][57][58][59][60][61][62][63][64][65][66][67][68][69][70]
- Massive parallel computing, Digital Signal Processing and very low cost hardware for simple operations or custom functions [71][72][73][74][75][57][59][67][76][77]
- Low Density Parity Check (LDPC) decoders [78][79]
- Low power and energy efficient image processing accelerators [37][80][81][82][83]
- Digital filtering [84][85]
- Mixed signal and analog applications [86][87][88][89][90][77]
- Robust architecture for fault tolerant technologies and reliability evaluation [91][92][93]
- Ultra low cost fully custom designs [94][46][53]

2.5.1 Stochastic computing for accelerating neural nets

Nowadays there is an increasing demand of computation tasks, especially the ones related to detection and recognition. The results are remarkable and can compete, or even outperform, several human tasks [95]. Some of these are speech recognition and its convertion into text, object identification in images or videos, even in real time such as in [96]. Complex neural networks can easily run in large server clusters, where high performance computational elements are present, such as CPU and GPU. Unfortunately, they have not reached good results in embedded systems, yet. Many accelerators are present in literature, which demonstrate that stochastic computing can significantly improve several important metrics, such as area occupation and power consumption. Table 2.2 reports some of these ASICs and the improvement each work brings on these metrics.

Neural Net	Tech	Area	Power	Energy
LeNet-5 conf. 11 [95]	45 nm NanGate OpenCell Library	13.98 mm^2 (-98%)	3.1 W (-99%)	7.9 μJ (-75%)
Electro EncephaloGram (EEG) classifier [97]	32 nm	6651 μm^2 (-92%)	203.9 μW (-88%)	N/A
Restricted Boltzmann Machine, larger configuration [98]	45 nm NanGate OpenCell Library	4.76 mm^2 (-98%)	545.67 mW (-98%)	69.86 nJ (-41%)
Multi Layer Perceptron, larger configuration [98]	45 nm NanGate OpenCell Library	10.76 mm^2 (-98%)	1234.7 mW (-98%)	180.26 nJ (-34%)
Custom Hybrid Stochastic/Binary neural network[47]	65 nm TSMC	1.321 mm^2	33.17 mW	543.42 nJ (-19%)
LeNet-5 larger configuration (tanh) [48]	45 nm NanGate OpenCell Library	12.5 mm^2 (-98%)	3.1 W (-99%)	15.8 nJ @1024 bit stream (+699%) 1 uJ @64 bit stream (-50%)
HEIF for LeNet-5 and AlexNet [68]	45 nm Nangate OpenCell Library	24.7 mm^2 (-61%)	1.9 W (-19%)	754 μJ (-86%)

Table 2.2: Neural net accelerators in literature and improvement with respect to the reference binary architecture reported in the design work.

Chapter 3

Design of a hardware Izhikevich neuron

The design of special purpose hardware needs several steps in order to satisfy all the constraints required. First of all, an high descriptive level of the algorithm is requested in order to have a comparison model, which is assumed to have infinite precision in number representation. The algorithm of the Izhikevich neuron is analyzed with Brian2 [99], which is a *Python* library used for spiking neural network simulations. From the results, obtained with a 64-bit double precision, a fixed point model is made in C, in order to view the differences with the floating point one, in terms of spiking timing and spike curve approximation. From these results it is derived the minimum number of bits which preserves the original membrane voltage and recover variable pattern. The last step is to design the Izhikevich neuron with an Hardware Descriptive Language (HDL). The architecture is designed in order to improve area usage, while its reduction is compared with the performace, in terms of clock period and cycles, and power consumption.

3.1 Brian2 simulation

A first integration of the Izhikevich equations is made by using the Brian2 simulator, from which the exact behaviour of the membrane voltage and recover variable is derived. The integration method adopted is the forward Euler, since it requires much less arithmetic operations than other higher order methods, such as Runge-Kutta [3]. The equations in (1.33) become:

$$\begin{cases} v(t+\Delta t) = v(t) + \Delta t(0.04v^2 + 5v + 140 - u + I) \\ u(t+\Delta t) = u(t) + \Delta t \cdot a(bv - u) \end{cases} \tag{3.1}$$

A simulation result to evaluate the membrane voltage pattern is reported in figure 3.1.

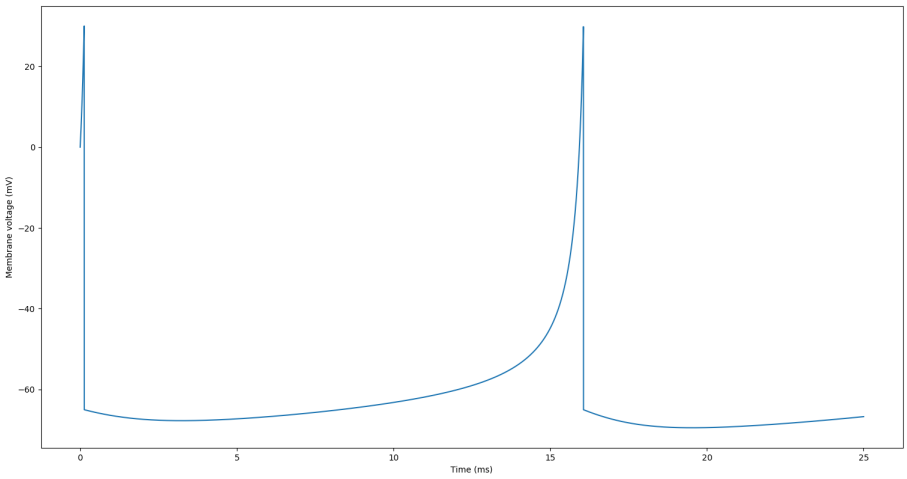

Figure 3.1: Membrane voltage of the Izhikevich neuron in tonic spiking mode. Integration step is 2^{-10}.

For the next design steps, the tonic spike behavior is used as a reference pattern for checking purpose, although other spiking pattern can be used.

3.2 Fixed point model

The Izhikevich equations in Brian2 have variables and constants in floating point representation, which is a problem when a low power and area design is required. The most simple and efficient solution is to implement a fixed point arithmetic, which does not require dedicated hardware for its execution, but only integer arithmetic components, which can be easier to be integrated into a single circuit. One important issue is to estimate the neuronal pattern change when the floating point representation is replaced with a fixed point one. For this purpose, a C program implementing the Izhikevich equations is written. Although this design step is not explicitly mandatory for an hardware design process, it is very useful for the following reasons:

- Result estimations with a C program are definitely faster than designing an entire hardware, testing it and comparing its results with the floating point arguments. As a matter of fact, the error from floating point representations

to fixed point one can be seen only in the end, when all the design steps are made.

- An HDL simulator requires more computational resources than an simple C program and it is usually proprietary. It is much better to have a general purpose, open source code, which can be compiled and executed in every platform.
- The C code is well suited for intermediate solutions such as microcontrollers or when an ASIC design is not required or not able to be achieved.
- For the next neural net simulations, topology analysis is faster in software than in hardware, since the latter requires an implementation for each neural net topology.

Although no strict specifications are given, the choice of this design step can be very useful for further testing, allowing to make fixed-point simulations (and also neural net training) without implementing necessarily an hardware architecture. All the variable used for testing are 64-bit integers, so the maximum number of bits available to represent fixed-point variables are 32. More larger variables can be used if available by the C compiler used.

The general functions implemented with the C program covers the following steps:

- Conversion from floating-point to fixed-point representation of the occurring values.
- Setup of the neuron parameter and initial values.
- Making of an integration step for a neuron.
- Testing of the results and comparing them with the ones obtained with the Brian2 simulation.

3.2.1 Float to Fixed conversion

The first step for fixed-point simulation is the conversion of the Izhikevich parameters from the the floating point to the fixed notation. The functions used are reported in figure 3.2.

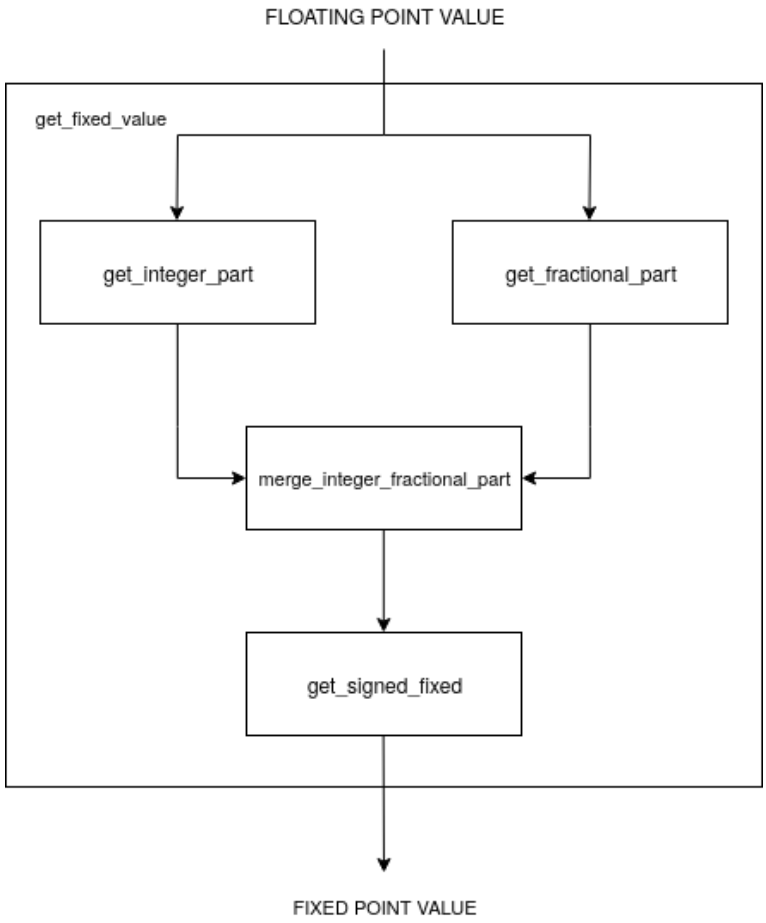

Figure 3.2: Float to fixed-point flowchart

The get_integer_part function takes the absolute value of the integer part of the floating point number and returns the integer part of the final fixed point number. The number of integer bits used must be specified, and the function warns the user if the number of bits used are not sufficient. The procedure adopted is the same as the one explained by Brown-Vranesic [12].

The get_fractional_part function takes the absolute value of the fractional part of the floating point number and returns the fractional part of the final fixed point

number. The number of fractional bits used must be specified, but no warning is sent to the user if the number of fractional bits were not sufficient. This choice is due to the fact that, generally, the conversion usually brings a conversion error, even with limited and non periodic fractions. This error decreases as the number of fractional bits provided increases. The procedure is quite similar to the one of the get_integer_part function and is reported in the flowchart of figure 3.3.

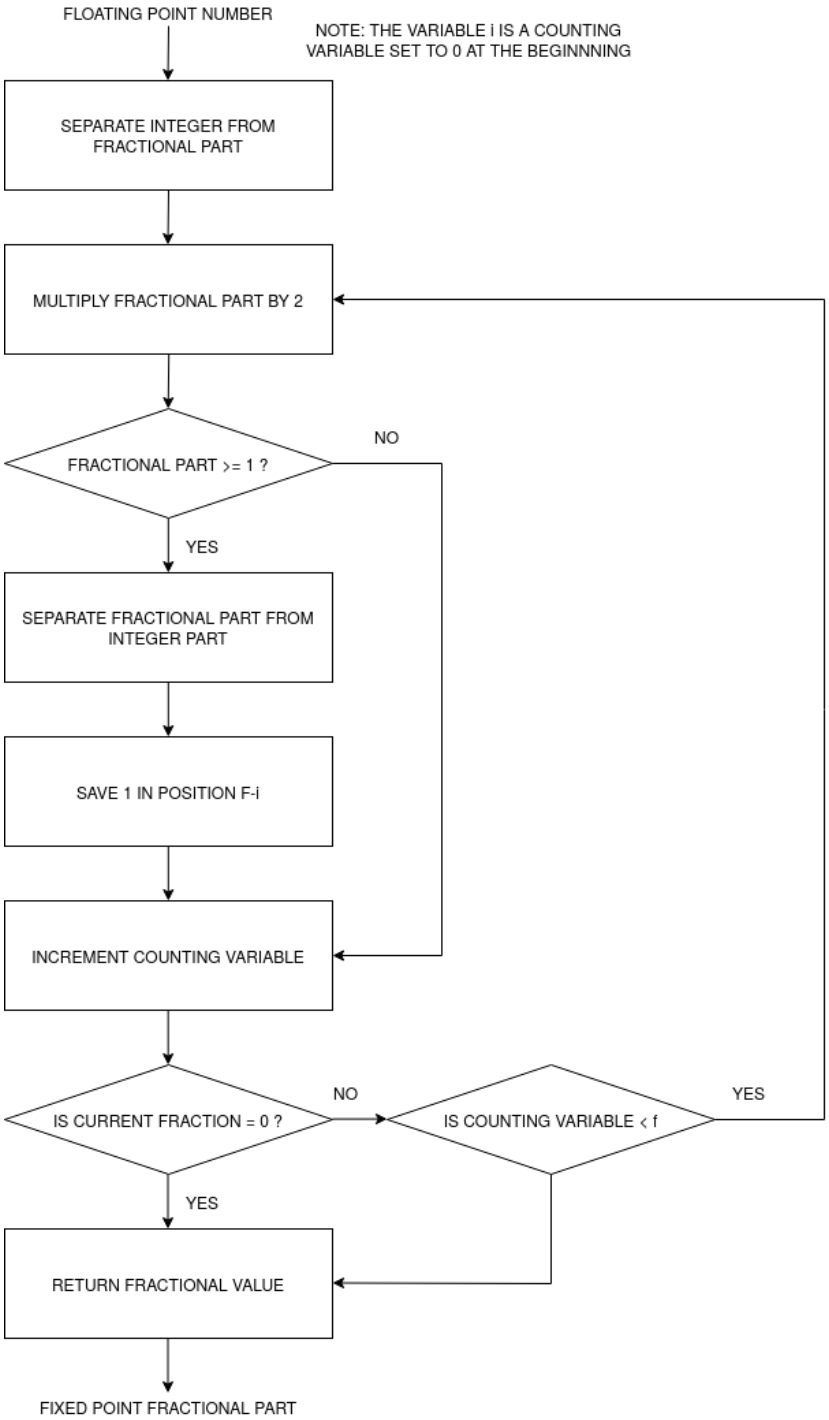

Figure 3.3: Flowchart to get a fixed-point fractional number from a floating point one.

Since only fractional part is involved in this function, then it can be written:

$$V_{frac} = \sum_{i=1}^{f} b_{-i} \cdot 2^{-i} \tag{3.2}$$

Multiplication by 2 shifts the digits left so that the (3.2) becomes:

$$V_{frac} \cdot 2 = b_{-1} + \sum_{i=2}^{f} b_{-i} \cdot 2^{-i} \tag{3.3}$$

The value b_{-1} can be 0 or 1 and it stored into the new integer variable as the most significant bit. Then the integer part is removed and the cycle continues, until the number of fractional bits available are 0 or the fractional part is already represented correctly with current bit available.

The merge_integer_and_fractional_part takes the two results obtained previously and merges them into a single fixed point number. As a matter of fact, the results obtained with the previous functions occupy the same position into the integer variable. If m is the number of integer bits and f the number of fractional bits, the get_integer_part function returns a number:

$$V_{int} = \sum_{i=0}^{m-1} I_i \tag{3.4}$$

All the other bits past the bit $m-1$ are zero. The get_fractional_part function also returns a similar result:

$$V_{frac} = \sum_{i=0}^{f-1} Q_i \tag{3.5}$$

Also here the bits past the bit $f-1$ are zero. In order to have both integer part and fractional part on the same integer variable, the merge function shifts logically left the integer bits by the number of fractional bits f and stores the fractional bits into these least significant bits. The function also checks whether $m+f > N$, where N is the number of bits available with the integer variable chosen and returns a warning message in case of overflow. The final result obtained is then:

$$V_{fixed-point} = (V_{int} << m) + V_{frac} = \sum_{i=0}^{f-1} Q_i \cdot 2^i + \sum_{i=f}^{f+m-1} I_{i-f} \cdot 2^i \tag{3.6}$$

The description of the result obtained in (3.6) is important when the fixed-point number has to be printed, saved or compared with a floating point number. The conversion requires a division by 2^f and saving the result into a floating point variable. This action brings of course to another error, which is smaller when the conversion takes a 32 bits fixed-point number to a 64-bit double variable and it is not taken into account.

The get_signed_fixed function returns the signed value of the unsigned fixed-point number if the initial floating point value is negative. The algorithm used is the simple complement and increment described in the radix complement schemes in Brown-Vranesic [12].

The overall procedure is summarized by the use of function get_fixed_value, which uses the four function previously described and returns a signed fixed point number stored into an integer variable.

3.2.2 Setup functions

Following the float to fixed-point functions a set of new ones are implemented to setup all the neuron parameters and variables.

The setup_spike_neuron_parameters setups the neuron fixed parameters in the Izhikevich equations a,b,c and d. This value are given as floating point numbers, so no previous conversions are required.

The setup_spike_neuron_current setups only the neuron input stimuli I. This parameter is taken separately since it can can change over the simulation time, while the other parameters are treated as intrisic neuron properties and are mantained constant.

The setup_spike_neuron_membrane_voltage function setups the membrane voltage into the Izhikevich equation. This function is used for simulation setup and reset only.

The setup_spike_neuron_recover_variable function setups the recover variable into the Izhikevich equation. This function is used for simulation setup and reset only.

3.2.3 Step functions

The next functions are used to proceed over the integration steps and derive the new variable values. These functions also change the current variable values, so also the current time changes by an intgration step.

The make_step function make an integration step to the membrane voltage and the recover variable by using the discrete Izhikevich equations (3.1) with integer arithmetic.

The fire_and_reset function checks the reset condition for a single Izhikevich neuron and resets the neuron variable with the (1.34) if the reset condition occurs.

3.2.4 Test function

A separate test function is implemented in order to compare the Brian2 results with the fixed-point model. In order to estimate the error introduced by changing from floating-point to fixed-point variable, this test function makes the following steps for each integration step:

- It reads a reads the results obtained previously from the Brian2 simulation.
- It makes an integration step for a testing neuron, which is set up to the same initial conditions. All the parameters for this neuron are in the fixed-point notation described in the previous section.
- It evaluate the current step absolute error AE with this error formula:

$$AE_i = \frac{V^i_{Brian2} - V^i_{fixed\_value}}{V^i_{Brian2}} \times 100 \tag{3.7}$$

  where V^i_{Brian2} is the value of the membrane voltage in the step i obtained with Brian2, while $V^i_{fixed\_value}$ is the value of the membrane voltage obtained with the fixed-point model.

For better error evaluation, figure reports the comparison between the floating-point value and a particular of 32 bits fixed-point implementation. As it can be seen from the two waves' behaviour, the error is mostly focused nearby the spike: the maximum absolute error found there is very large due to the fact that the error formula in (3.7) increases very quickly with small differences of almost-zero values. The error found here is more than 170%, but fortunately for only one integration step. Furthermore, the firing and reset condition happens in the same integration step for both models: this is a crucial aspect, since it is the spike-timing activity that actually carries information. For all other 95% cases, the absolute error is always below 0.1%, making the fixed point very precise despite of the number of bit reduction.

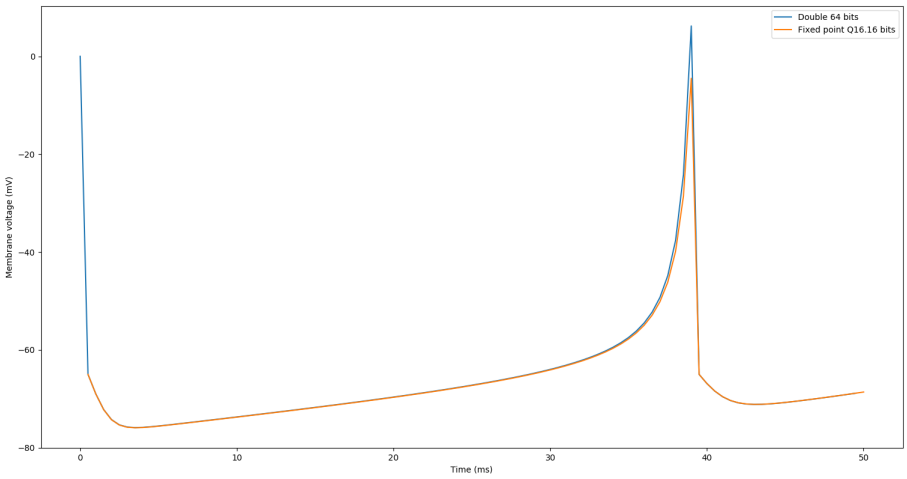

Figure 3.4: Comparison between floating-point results and the fixed-point implementation.

3.3 Hardware implementation

The fixed-point values found with the previous design step becomes a requirement for the following hardware architecture: since the first implementation in hardware uses deterministic arithmetic in order to make further comparisons with a stochastic one, the result obtained with the former must be same of the fixed-point algorithm.

The basic blocks of all architectures are reported in figure 3.5. The DATAPATH block contains the equation variables and constants of a single neuron and the arithmetic units to process them. The uControl Unit (uCU) contains all the logic to control the dataflow of the DATAPATH. The commands from the uCU to the DATAPATH are divided into fields:

- The REG_DECODER_ENABLE is the signal which enables the datapath variable registers for write procedures.
- The REG_DECODER_SELECT is the signal used to select the register to be written and, when necessary, also the data to be written into it.
- The OPERAND_SELECTOR selects the register variable or variables which are to be directed to the arithmetic operators. This signal is used when there is resource sharing and the arithmetic operations must be scheduled.

- RCS stands for "Reset Commands Strobe". It is a signal used for equations with reset condition, which is present for most of the spiking neuron equations. When this signal is set to '1', the datapath must set the SPIKE signal to '1' if the reset condition occurs. The implementation for reset condition evaluation must respect this constraint, although other aspects, such as the change of the internal variables, can be implemented without particular restrictions.

The write procedures for internal variable is general and can be implemented in different ways with no particular constraints, so a particular implementation is shown when the variable storing design is explained. The I_STIMULUS signal can be used to load external values to the datapath internal registers, although this approach is not recommended. This signal is thought to load the input stimuli current only, since it is the only variable that must be loaded to the datapath for each integration step and for most spiking neuron equations.

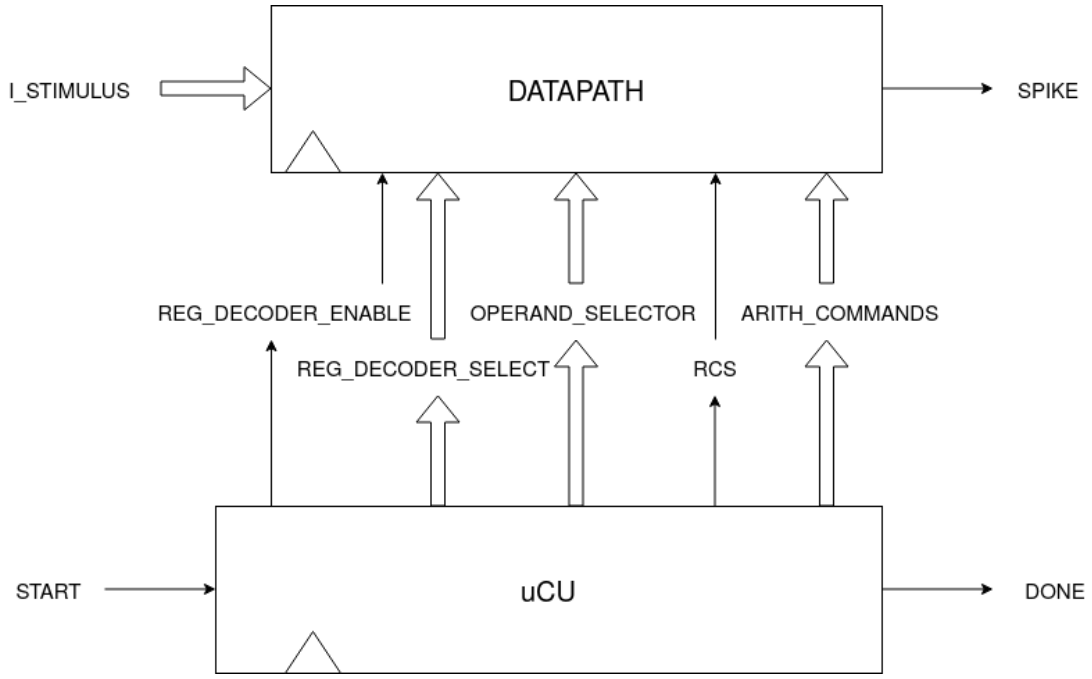

Figure 3.5: Basic blocks for hardware implementation of the neuron equations.

A preliminary analysis is made by drawing the Data Flow Graphs (DFG) in order to find the main operators involved. Then it is presented the general schematic of the architecture, which is composed by a single Contronl Unit (CU) per layer and by one DataPath (DP) for each neuron. Then the architecture designs are compared with the direct implementation in hardware of the DFGs.

3.3.1 Data Flow Graph

With respect to the discrete Izhikevich equations of 1.33, the Data Flow Graph are derived and reported in figure 3.6. Furthermore, a direct implementation is made in order to derive the initial reference metrics. As it can be seen, the DFGs includes several multiplications, which should not be directly implemented with multipliers in order to save area. In addition to this, the adders can be also area consuming when there is the need to instantiate several of these units.

In order to derive low area architectures, the following choices are taken as specifics:

- The solutions are completely multiplierless
- All arithmetic operation, including also squaring and multiplications, are implemented with sums and shifts algorithms only.
- The resource sharing policy is adopted: an instantiated operator is reused for several operations in the DFGs.

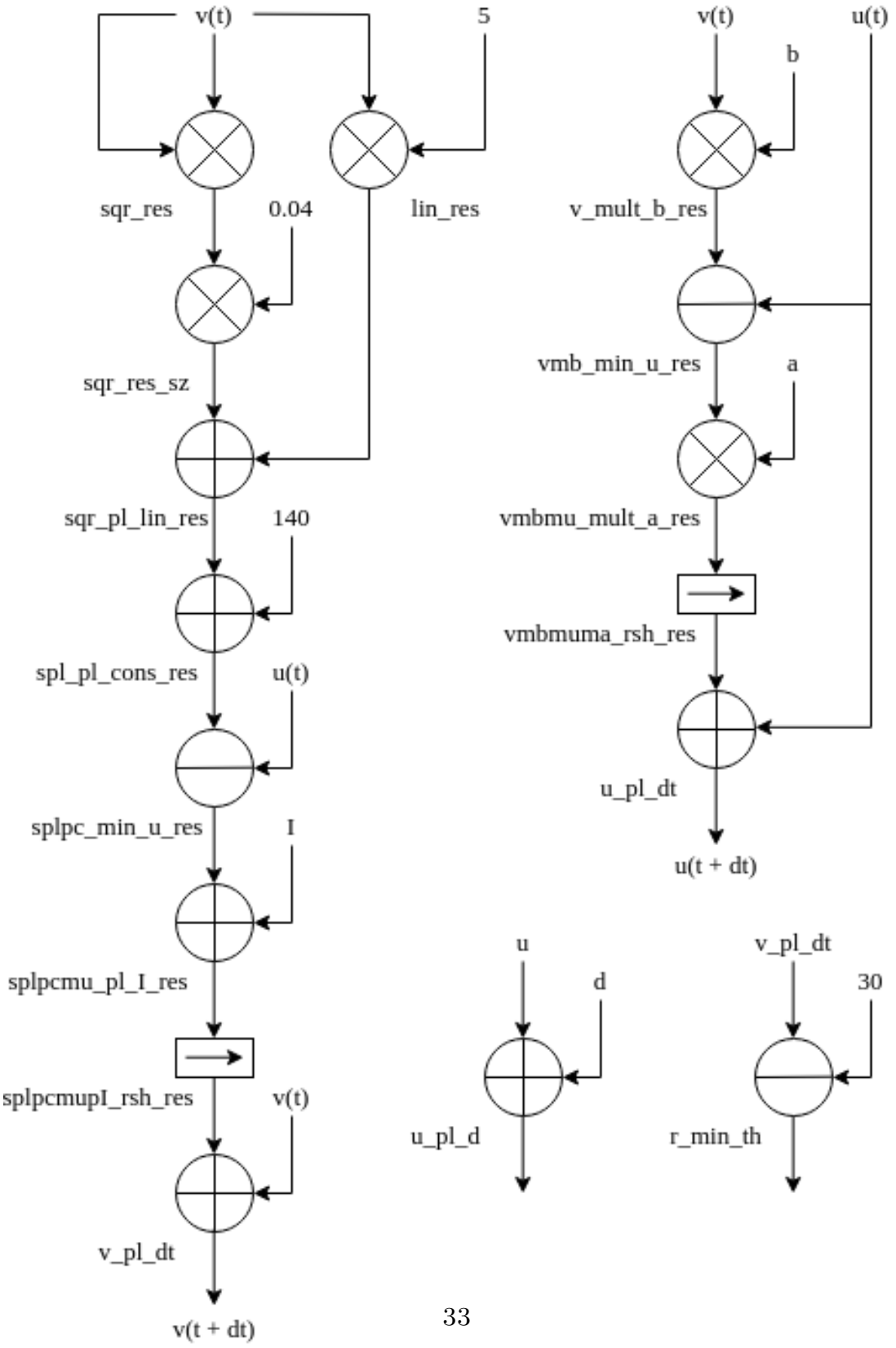

Figure 3.6: Data Flow Graphs of the discrete Izhikevich equations. The labels are the names of each internal signal.

3.3.2 Direct implementation

Figure 3.7 shows the direct implementation of the DFG. This one is used as a reference for the further implementations, in order to make an architectural comparison in term of speed, area and power consumption. All the arithmetic operations are inside the three main blocks, which are the "Membrane Voltage DFG", the "Recover Variable DFG" and the "Comparator" blocks. Each operation is done by a different arithmetic unit and the internal parallelism remains the same as the initial one. In order to obtain the same results of the fixed point program, the result of each multiplication is right shifted arithmetically by the number of fixed point number, then the MSBs of the integer part and the LSBs of the fractional part are truncated.

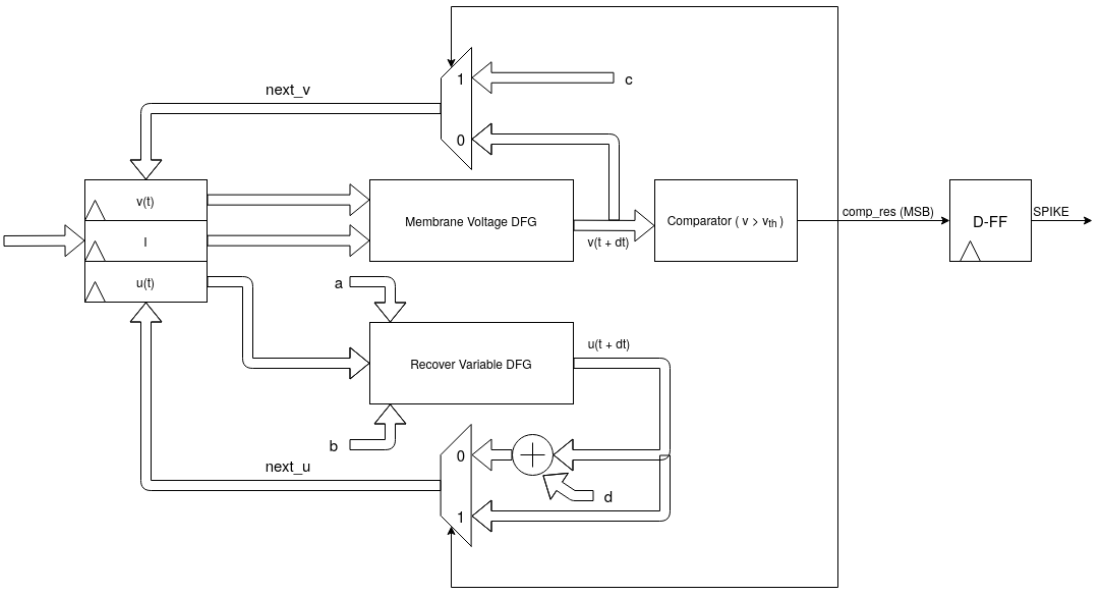

Figure 3.7: Direct implementation of the DFG datapath.

3.3.3 Control Data Flow Graph and number of operators

The first solution which is thought to be sufficient to be comparable to a stochastic arithmetic is a solution with a single operator, which means that all arithmetic operation, including sums, multiplications and subtractions, must be implemented with a single arithmetic unit. With this consideration, it is derived the Control Data Flow Graphs (CDFG) from the DFG. Figure 3.8 shows the steps involved for the generation of one integration step. Each step involves only one operator at a time. The solution adopted requires then an adder with some additional gates and registers in order to implement subtraction, multiplication and shifts with the same adder. Each step of the CDFG requires different clock cycles to be completed. The exact number is shown along with the control unit implementation.

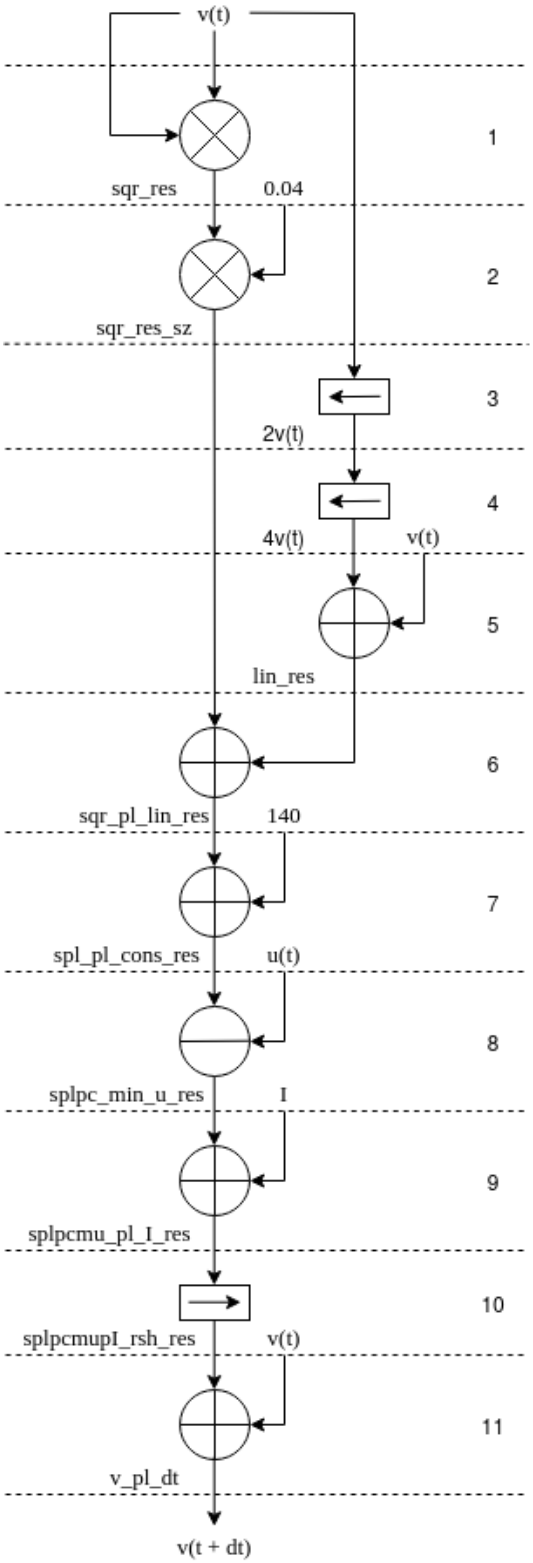

Figure 3.8: Control data flow graph of the discrete membrane voltage equation.

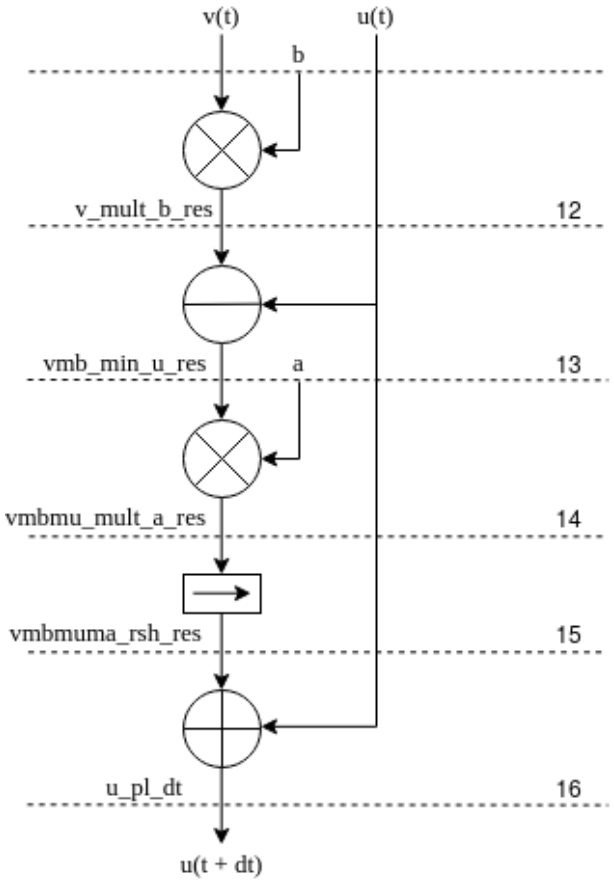

Figure 3.9: Control Data Flow Graph of the discrete recover variable equation.

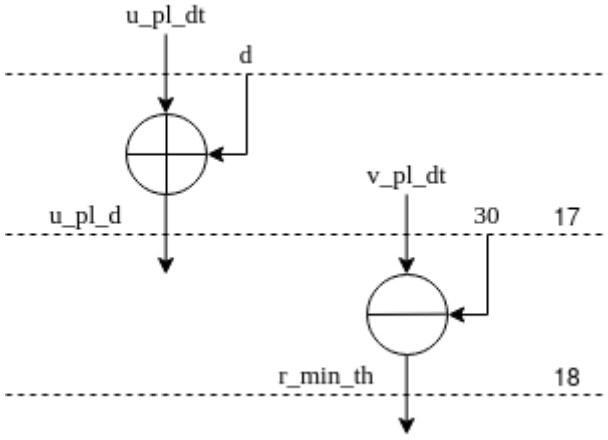

Figure 3.10: Control Data Flow Graph of the reset conditions

3.3.4 Variable lifetime analysis

The CDFG permits also to make an accurate variable lifetime analysis, whose result reveals the effective registers needed to store all variable without exceeding in the number of memory components. Table 3.1 reports the variable lifetime of all results involved into the CDFG previously implemented.

Variable name	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
v(t)	X	X	X	X	X	X	X	X	X	X	X	X						
u(t)	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	
sqr_res	X																	
sqr_res_sz		X	X	X	X													
2v(t)			X															
4v(t)				X														
lin_res					X													
sqr_pl_lin_res						X												
spl_pl_cons_res							X											
splpc_min_u_res								X										
splpcmu_pl_I_res									X									
splpcmupI_rsh_res										X								
v_pl_dt											X	X	X	X	X	X	X	X
v_mult_b_res												X						
vmb_min_u_res													X					
vmbmu_mult_a_res														X				
vmbmuma_rsh_res															X			
u_pl_dt																X	X	X
u_pl_d																	X	X
v_min_th																		X
Variables in step	3	3	4	4	4	3	3	3	3	3	3	4	3	3	3	3	3	4

Table 3.1: Variable lifetime analysis

From this analysis it is derived that the minimum number of registers to be included into the register file are 4. In particular, in the end of step 18, 2 registers called "variable registers" are needed to save $v(t + \Delta t)$ and $u(t + \Delta t)$, while the others to save $v(t + \Delta t) - v_{th}$ (where v_{th} is the threshold voltage) and $u(t + \Delta t) + d$. After step 18 there is another step in order to check if the reset condition occurs and, in case of positive response, it fires a spike and reset the "variable registers" with the corresponding reset values. The register file also saves the input current given by the pre-synaptic block (this block is described in the further chapter). The register file contains also the neuron parameters needed to execute the discrete Izhikevich algorithm.

3.3.5 One adder arithmetic block

Figure 3.11 shows the first implementation of the arithmetic block, which is used to implement all the arithmetic and logic operation necessary.

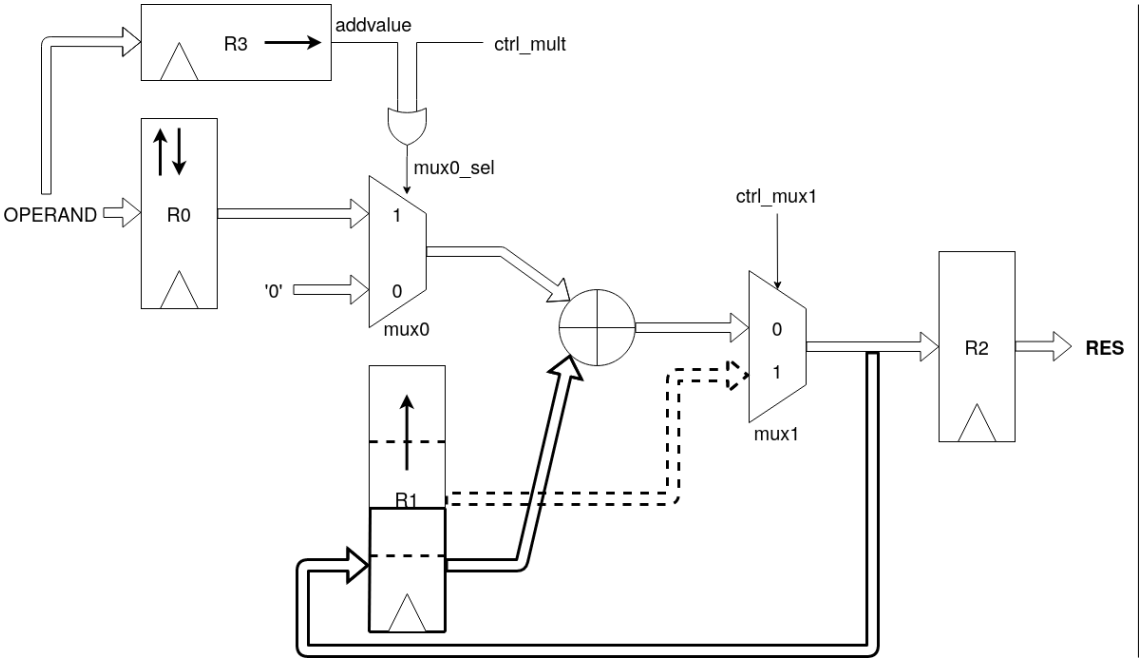

Figure 3.11: Architecture of the arithmetic block with a single adder.

The arithmetic block contains four special register, two n-bits selectors (or n 2-way muxes) and a single arithmetic operator, which is considered the core unit of the design. Each register has its own control signals, since each of these has different functionalities. For example, R0 needs to perform both logical right shift and arithmetic left shift, while R3 needs only the arithmetic left shift property. All registers have an enable signal, in order manage the dataflow of all operations. Register R1 is bigger than the others, since it serves as accumulator for storing the partial products of multiplications. Register R1 has two outputs:

- the register1_accumulator (or R1_acc) output, which is marked with a bold arrow, is the output of the n most significant bits and it is directed to the arithmetic operator.
- the mult_result output, which is marked with a dashed bold arrow, is the signal which contains the final result of a multiplication.

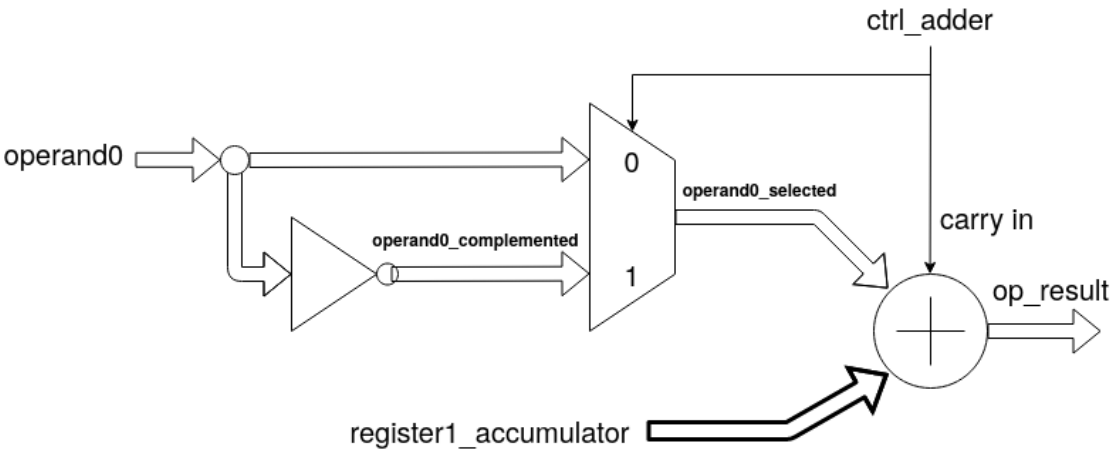

Figure 3.12: Implementation of the arithmetic core.

The arithmetic operator performs both addition and subtraction, so the real core is represented in figure 3.12. As it can be seen, this unit presents an additional n-bit selector and n inverters.

The arithmetic unit has two inputs, which are the OPERAND signal, coming directly from the register file, and the CMD_VECTOR signal, containing all the command signals. The only output is the RES signal, which carries the arithmetic results back to the register file. In order to control the dataflow of the arithmetic unit in a simpler way, the arithmetic command vector is divided into fields, which are described in table 3.2. Each field controls a specific component of the arithmetic block.

CMD_VECTOR [12:0]		
Name	Position	Description
ctrl_r0	[12:10]	controls the register R0
ctrl_r1	[9:7]	controls the register R1
ctrl_r2	[6]	controls the register R2
ctrl_r3	[5:3]	controls the register R3
ctrl_adder	[2]	with 0 the operator performs a sum, with 1 a subtraction
ctrl_mult	[1]	when 0, the arithmetic unit is in multiplication mode and in addition/subtraction mode with 1
ctrl_mux1	[0]	selects multiplication result with 1 and the output of the arithmetic operator with 0

Table 3.2: Command fields for the arithmetic block. The CMD_VECTOR includes all the fields described into this table.

In order to make the control signals of the arithmetic registers more readable, a mnemonic name for each command is used. These mnemonics are summarized and described in table 3.3. When no operation has to be performed by a register, it is sufficient to set its control signals to 0. The description of these mnemonic are useful when there is the need to create or modify the uInstruction set, which is described within the control unit part.

Name	Value	Description
ctrl_r0		
enable_r0	100	enables register
lshift_r0	110 (111)	performs logical shift left. The value in brackets is for the single fulladder architecture implementation
rshift_r0	101	performs arithmetic shift right
rotate_r0	110	performs rotate right shift. This mnemonic is for the single fulladder architecture only
ctrl_r1		
reset_r1	001	resets register
enable_r1	100	enables register
enable_and_shift_r1	110	enables register and performs an arithmetic shift right on the new data
ctrl_r2		
enable_r2	1 (10)	enables register. Value in brackets for the single fulladder architecture implementation
rshift_r2	01	loads the fulladder result to the MSB of the register after a right shift if performed
ctrl_r3		
reset_r3	001	resets register
enable_r3	100	enables register
rshift_r3	110	performs an arithmetic right shift

Table 3.3: Fields mnemonic description for the arithmetic registers and corresponding binary values.

3.3.6 The register file

Internal neuron variables and parameters, including the equation constants, are contained in a custom register file, located into the datapath unit of each neuron instantiation. The register file contains both variables evaluated during the algorithm execution and constants. Constants can be avoided by replacing them with shifts and accumulations performed by the arithmetic block, although this could significantly increase the uROM size, containing all the uInstructions. The only output signal is REG_OUTPUT and its output is described by figure 3.13 The OPERAND_SELECTOR signal has its own mnemonics in order to make code more readable. These mnemonics are reported in table 3.4 and describe the current selection of the multiplexer.

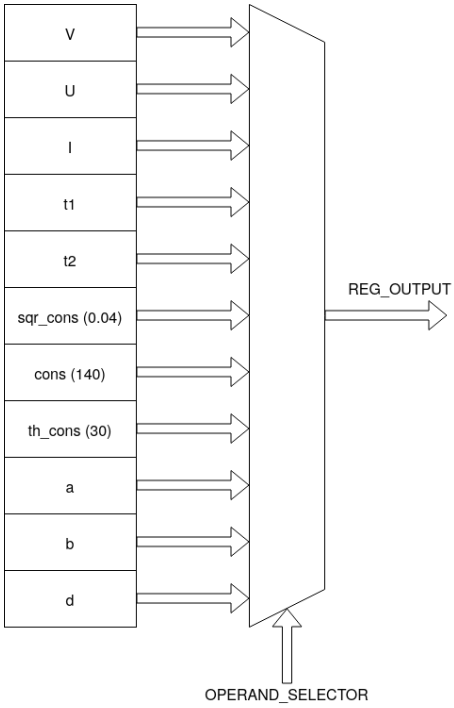

Figure 3.13: Register file selection schematic.

OPERAND_SELECTOR	
Name	Value
regv	1000
regu	0001
regI	0010
regt1	0011
regt2	0010
squarecons (0.04)	0101
voltagecons (140)	0111
consth (30)	1111
consa (a)	1000
consb (b)	1001
consd (d)	1011

Table 3.4: Description of the mnemonics for OPERAND_SELECTOR signal.

The registers regv, regu, regt1 and regt2 can be loaded with the result signal RES, coming from the arithmetic block. In order to select and enable one of these registers, the REG_DECODER_ENABLE signal must be set to '1' and the

REG_DECODER_SELECT must be set to one of the values reported in table 3.5.

REG_DECODER_SELECT	
select_v	00
select_u	01
select_t1	10
select_t2	11

Table 3.5: Mnemonics for register decoder selection.

The regI register, which contains the input stimuli current, is sampled each clock cycle from the DATAPATH input.

This register file implements also the reset condition for the SPIKE signal. This implementation is The circuitry used is reported in figure 3.14, which also shows the values that must be present in registers t1 and t2.

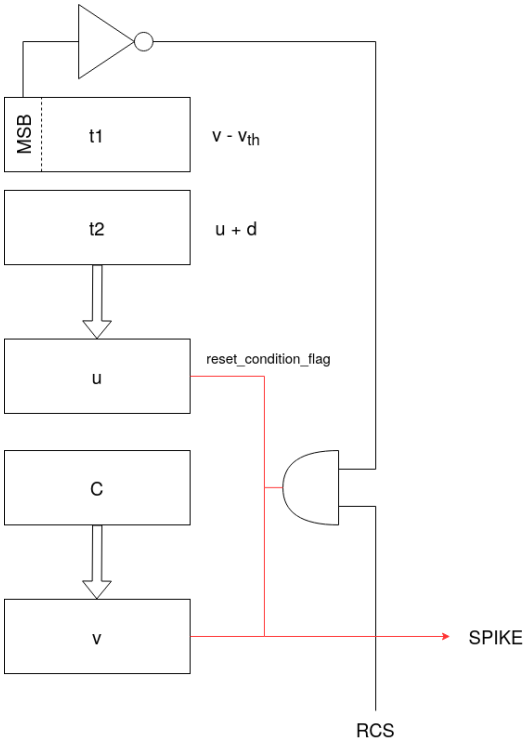

Figure 3.14: Circuitry for SPIKE generation and internal variable reset. Register t1 must contain the value $v - v_{th}$ and t2 must contain the value $u(t+dt)+d$ during the reset condition check period.

As it is requested from the specifications previously defined, the SPIKE signal

is set to '1' in the same period if the RCS signal is asserted and the reset condition occurs. As a matter of fact, the reset condition for the Izhikevich equations can be rewritten as in the (3.8):

$$v \geq v_{th} \Rightarrow v - v_{th} \geq 0 \tag{3.8}$$

If the MSB of the value $v - v_{th}$ is '0' it means that the value is non negative, so only the most significant bit of register t1 can be used for the check circuitry.

This event resets also the values of the membrane voltage and the recover variable, so the reset_condition_flag signal is used as a special reset for registers v and u. The parameter c is considered constant and it is not stored in a register but serves only to define a new reset for register v. The recover variable reset depends on its current value, instead, so t2 is used as a support register. If the reset condition occurs, then value of the recover variable becomes the one contained in register t2. All these register resets described for spike event are synchronous.

3.3.7 uControl Unit

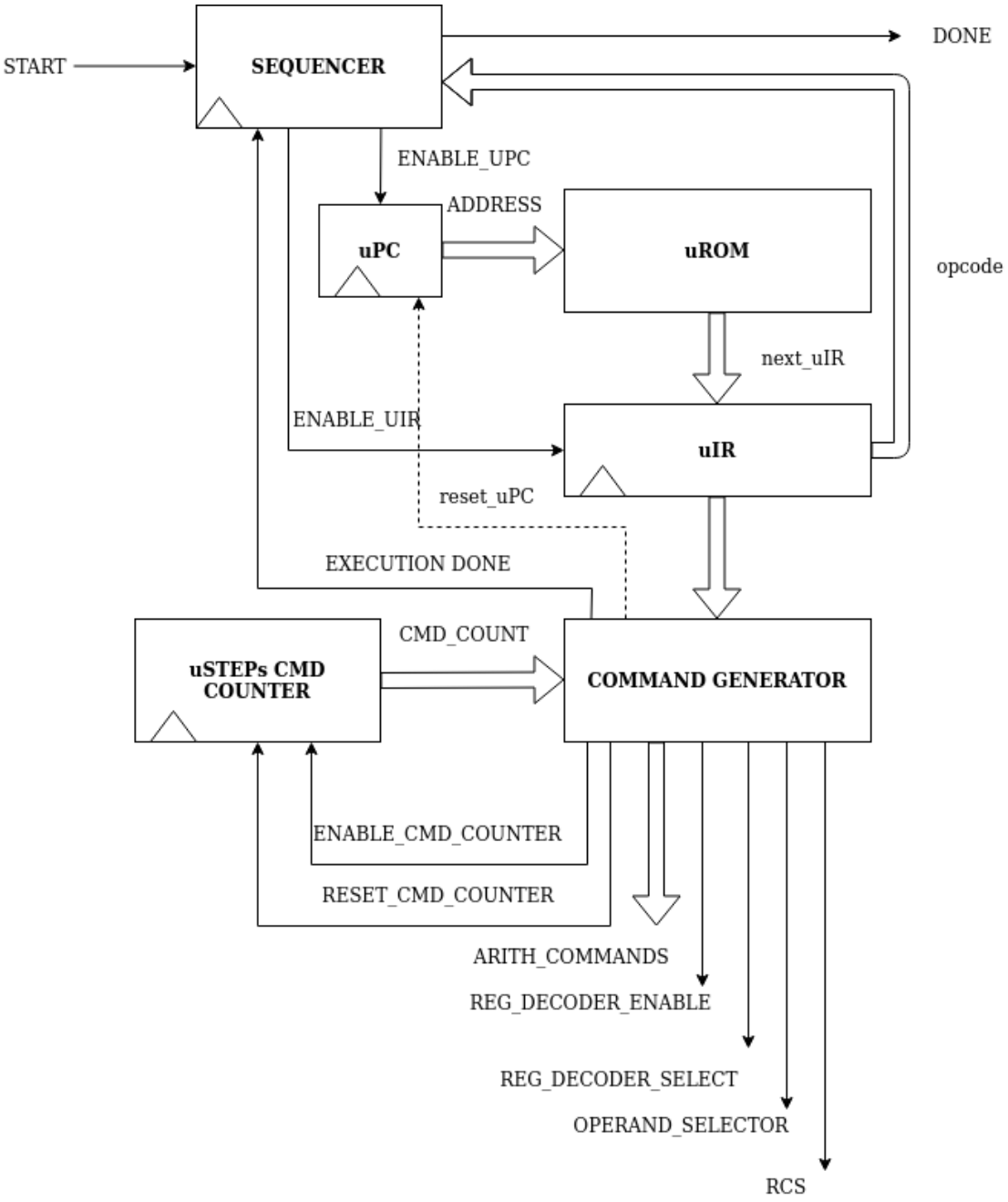

Figure 3.15: Schematic of the uControl Unit. The architectures of the Izhikevich with one adder or one full adder for arithmetic operations use this unit for dataflow control.

In order to control the datapath correctly, a uControl Unit(uCU) is designed. The general schematic of the uCU is reported in figure 3.15.

The uCU is implemented with a uCode structures, which means that the commands through the datapath are generated by starting from uInstructions. These uInstructions are saved in a uROM, which is implemented by using combinational circuitry [100]. The main reason for choosing a uCode architecture is that it is enough flexible for further improvements and in case that the original algorithm is modified because, for example, the equations are arranged in a different way or are changed with total different ones. The Izhikevich equations are indeed only a particular design choice which can be over This type of state machine requires of course more components than a simple Moore state machine, but the little increase of area occupation is balanced by the fact that the uCU can be instantiated only once for each group of neuron with the same equations that are working in parallel.

uPC The uPC is an address generator. It consists of a counter which can increment its value by setting to '1' the ENABLE_UPC signal or reset it by setting reset_uPC signal to '1'.

uIR The uIR is a register which contains the current uInstruction to be executed.

Sequencer The sequencer units is the block that manage the input/output signal for the flow control.

Command Generator The Command Generator is a combinational block which decodes the uInstruction in the uIR and generates the control signals for the datapath. Since each uInstruction may require more than one clock cycle, the Command Generator uses the uSTEPs CMD COUNTER's signal CMD_COUNT to know the current uSTEP of the uInstruction and generate the proper datapath control signals.

3.3.8 uInstruction Set

The uCU can execute the uInstructions reported in table 3.6.

Opcode	First Operand	Second Operand	Result destination	Description
nop	empty	empty	empty	Keep uCU in idle
addOp	reg1	reg2	reg_dest	reg_dest <= reg1 + reg2
subOp	reg1	reg2	reg_dest	reg_dest <= reg1 - reg2
multOp	reg1	reg2	reg_dest	reg_dest <= reg1 × reg2
accOp	reg1	empty	empty	R1_acc <= R1_acc + reg1
accsubOp	reg1	empty	empty	R1_acc <= R1_acc - reg1
shiftlOp	empty	empty	empty	R0 << 1
shiftrOp	empty	empty	empty	R0 >> 1
jumpOp	empty	empty	empty	Reset uPC
movOp	reg1	empty	empty	R0 <= reg1
saveOp	empty	empty	reg_dest	reg_dest <= R0
transOp	reg_src	empty	reg_dest	reg_dest <= reg_src
spikeOp	empty	empty	empty	set RCS to '1' for one clock cycle

Table 3.6: uIstructions available for the single adder architecture. Empty spaces are filled with zeroes.

The possible values for the first operand and the second operand are the ones reported in table 3.4, while the destination register values are in table 3.5.

3.3.9 Single Full Adder Datapath

In order to reduce even more area, a second implementation of the arithmetic block is made. The new structure is reported in figure 3.16.

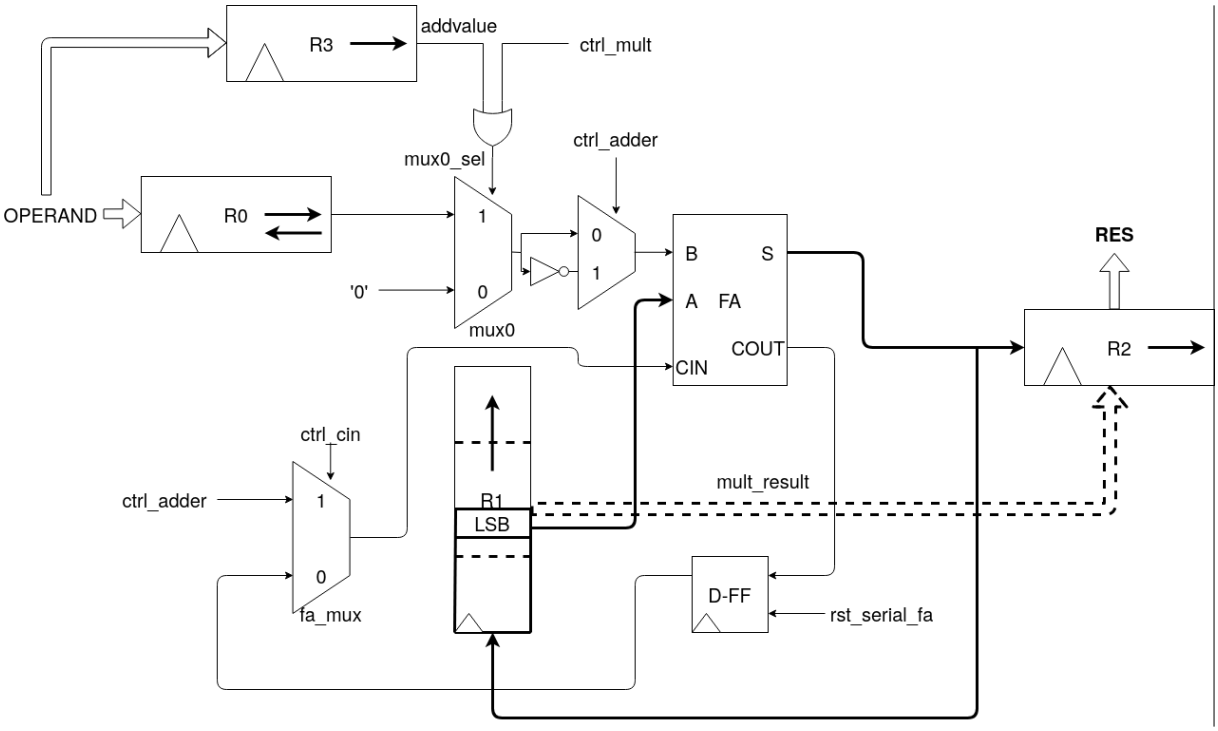

Figure 3.16: Arithmetic block with one fulladder instead of an adder/subtractor unit.

The ctrl_cin signal is controlled by a small finite state machine, whose state graph is reported in figure 3.17, so that the architecture can still perform addition or subtraction with the same fulladder.

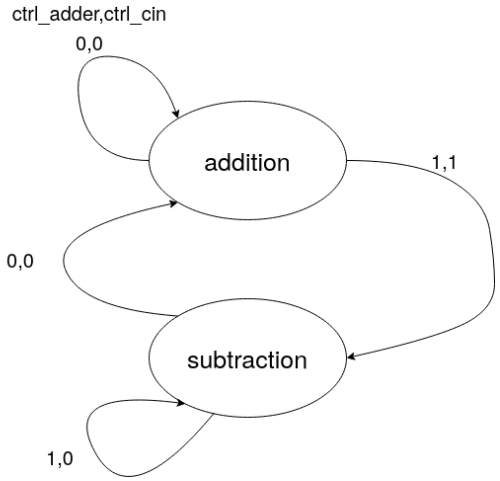

Figure 3.17: Finite state machine for ctrl_cin signal management.

With respect to the single adder implementation, the register R1 receives now one bit at a time from the fulladder and saves its value in the MSB. The register R0 performs arithmetic right shifts to pass the values to the fulladder, and uses rotate right shifts in the case of a multiplication. The registers R2 receives now the arithmetic results serially, except for the final multiplication result, which is called mult_result, which is directly loaded into R2 at the end of the multiplication uSteps.

uCU adaptation for serial arithmetic operations Since now all the fixed point operations are made with serial arithmetic, the uCU has one more counter to manage the serial steps. The uInstructions that can be used are reduced and are the addOp, subOp, multOp,transOp and the spikeOp. The area increase of the uCU due to the new counter is balanced by reducing the uInstructions in the uROM.

3.4 Equation normalization

Stochastic computing with complex functions is much easier to be implemented in hardware when all variables and parameters are normalized. The most common and suitable ranges are [0,1] for the unipolar representation and [−1,1] for the bipolar one. Without normalization most complex functions with several gate stages would require intermediate evaluation of the value, with an unacceptable increase of computation time in terms of clock cycles, since a precision of n bits requires already 2^n clock cycles.

As it is said by Izhikevich itself [7] the equations can be changed in order to obtain a different range values and time scale. In order to do so, the part to be substantially modified is the first one of the membrane voltage differential equation in (1.33):

$$f(v) = 0.04v^2 + 5v + 140 \tag{3.9}$$

The constants in (3.9) can be generalized as parameters, which will be derived by applying the wanted constraints. Thus the equation (3.9) becomes:

$$f(v, k_1, k_2, k_3) = k_1v^2 + k_2v + k_3 \tag{3.10}$$

Where k_1, k_2 and k_3 are the generalized parameters. These parameters must also be normalized, since they are used as probabilities, too.

Although a more accurate and rigorous mathematical analysis can be made, the choice adopted to find the normalized parameters is more simple and leads fast to the result needed for the implementation. The consideration is that, if the parameters a, b, c and d are chosen for most common spiking activity such as tonic spiking or class 2 excitability, then both membrane voltage and recover variable must have a resting value in case of absence of pre-synaptic stimuli, which means $I = 0$. In this quiescent condition the first derivatives must be equal to zero, since no variation occurs for both membrane voltage and recover variable.

Another important property which should be preserved is that, when membrane reaches low polarization values, which means nearly zero or positive, the neuron must fire a spike and then re-polarize the membrane voltage. In order to preserve this aspect, it is clear that the k_3 parameter must be positive

In order to summarize the conditions derived from previous considerations:

- The new equation must contain only normalized parameters, so

$$k_1, k_2, k_3, a, b, c, d \in]-1,1[\tag{3.11}$$

- The membrane voltage and the recover variable must be also normalized;
- A stability point must be chosen in order to impose quiescent condition. The necessary condition for a stability point is that, given the two stability values $v_0, u_0 \in]-1,1[$, the derivatives described into the Izhikevich Equations must be zero:

$$v'(v_0, u_0, I = 0) = 0 \tag{3.12}$$

$$u'(v_0, u_0) = 0 \tag{3.13}$$

- A more strict condition is imposed for the parameter k_3, which must be also positive.

From the previous equation the following new conditions are derived:

$$\begin{cases} v'(v_0, u_0, I = 0) = 0 \Rightarrow k_1 v_0^2 + k_2 v_0 + k_3 - u_0 = 0 \\ u'(v_0, u_0) = 0 \Rightarrow b v_0 - u_0 = 0 \Rightarrow u_0 = b v_0 \end{cases} \tag{3.14}$$

It is interesting to see that the stable recover variable value is related to the stable membrane voltage one by the second equation of (3.14), so only one of these can be chosen. In this case it is more convenient to choose v_0, since b is normalized and the resulting variable is smaller than the membrane voltage, which is also normalized.

If the coefficient k_1 is chosen, the parameter k_3 can be derived as a function of k_2 and vice versa. It is more convenient to choose the first option, since the parameter k_2 is the most critical parameter to determine the working range of the two neuron variables. As a matter of fact, it is found from Brian2 simulations that the k_2 parameter determines significantly the range of membrane voltage and can change significantly the stability point. For some values, the k_2 parameter can lead to a stability point totally different from the expected result.

From the (3.14) the following function is derived:

$$k_3(v_0, k_1, k_2, b) = -k_1 v_0^2 + (b - k_2) v_0 \tag{3.15}$$

It is important to see that, in order to derive the new parameter, the parameter b must be chosen and usually determines the sign of k_3. As a consequence, a good value of b is chosen in order to determine the k_3 parameter, but then the parameters of the (3.10) are treated as constants, as it would be for the original part in (3.9). The b parameter is chosen initially so that the neuron reproduces a tonic spike behaviour or class 2 excitability one, since these patterns are the most common and used in machine learning.

The response of the new normalized Izhikevich neuron is then implemented with both floating point and fixed point and then compared with each other. The result of this comparison is reported in figure 3.18.

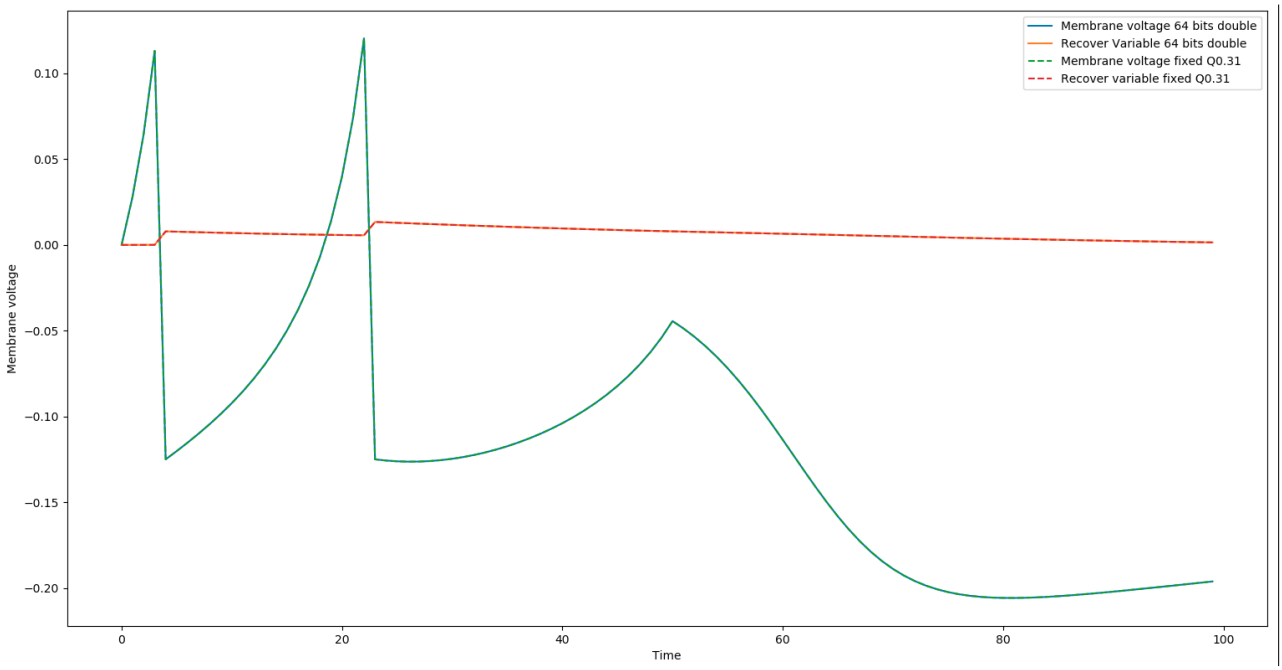

Figure 3.18: Comparison of normalized Izhikevich equations in floating point and fixed point arithmetic.

3.5 Stochastic implementation of an Izhikevich neuron

After the equation being normalized, it is much easier to implement them with the use of stochastic arithmetic.

3.5.1 Stochastic arithmetic core

The most important part of all design is the arithmetic core, where all algorithmic operations are performed. This block is composed by two parts, which are reported in figure 3.19 and figure 3.20.

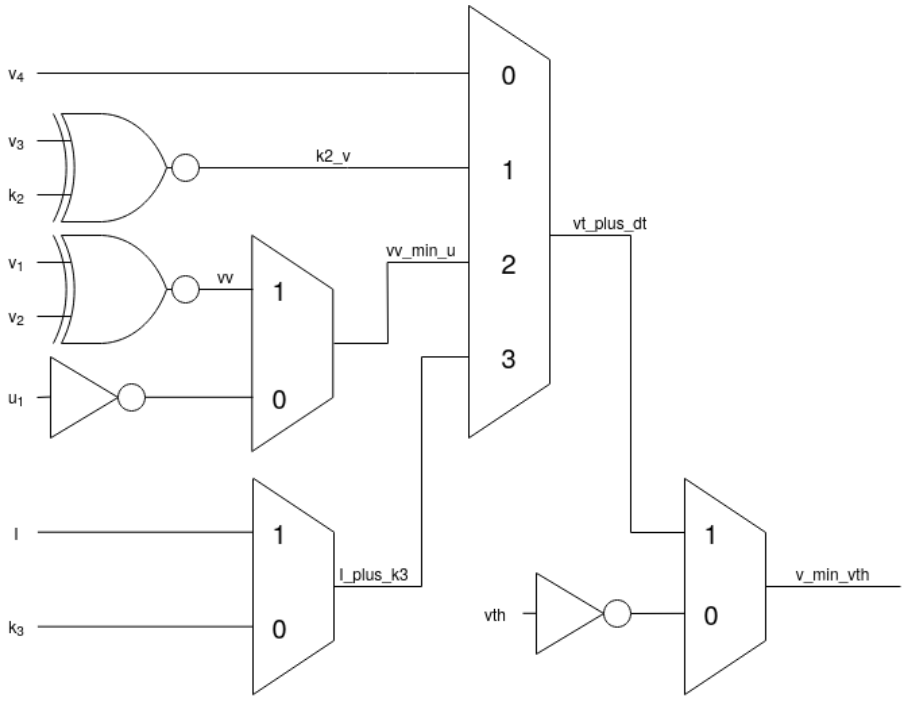

Figure 3.19: Stochastic arithmetic block for membrane voltage and threshold condition estimation.

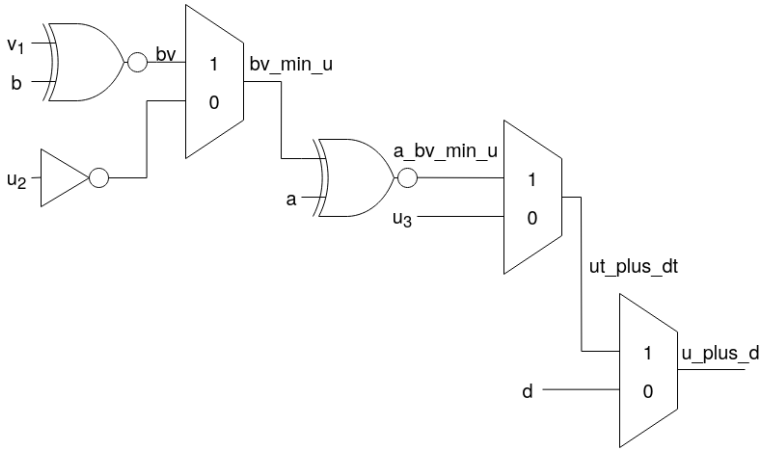

Figure 3.20: Stochastic arithmetic block for recover variable estimation.

The behavior of these arithmetic circuits reflects the Izhikevich equations. As a matter of fact, with respect to the output values explained in table 2.1, the stochastic results of these arithmetic blocks are derived:

$$v(t+\Delta t) = \frac{1}{4}\Big(v_4 + v_3 k_2 + \frac{1}{2}(v_1 v_2 - u_1) + \frac{1}{2}(I + k_3)\Big) \tag{3.16}$$

$$v\_min\_vth = \frac{1}{2}(v(t+\Delta t) - v_{th}) \tag{3.17}$$

$$u(t+\Delta t) = \frac{1}{2}\Big(u_3 + a\Big(\frac{1}{2}(bv_1 - u_2)\Big)\Big) \tag{3.18}$$

$$u\_plus\_d = \frac{1}{2}(u(t+\Delta t) + d) \tag{3.19}$$

In order to obtain the same output of the Izhikevich equations, the variables must be pre-amplified by a proper factor. These coefficients are reported in table 3.7.

Variable/Parameter	Pre-amp coefficient
v_1	2
v_2	4
v_3	2
v_4	4
k_2	2
k_3	8
u_1	8
u_2	4
u_3	2
I	8
a	2
b	2
d	2

Table 3.7: Preamplification coefficients.

A problem that occur with variable pre-shift is that the normalized value of the membrane voltage cannot be larger than 0.25 in magnitude. This condition never happens for positive values, since the positive threshold is set to 0.125. For negative values, it is found from Brian2 simulations that membrane voltage satisfies this condition for the time period analyzed, which the neuron takes to reach a stable value without input currents ($I = 0$).

3.5.2 Stochastic datapath with estimation circuits

In order to exploit the stochastic arithmetic blocks properly, additional support blocks must be included for stochastic numbers generation and next variable values estimation. The schematic of the stochastic datapath is reported in figure 3.21.

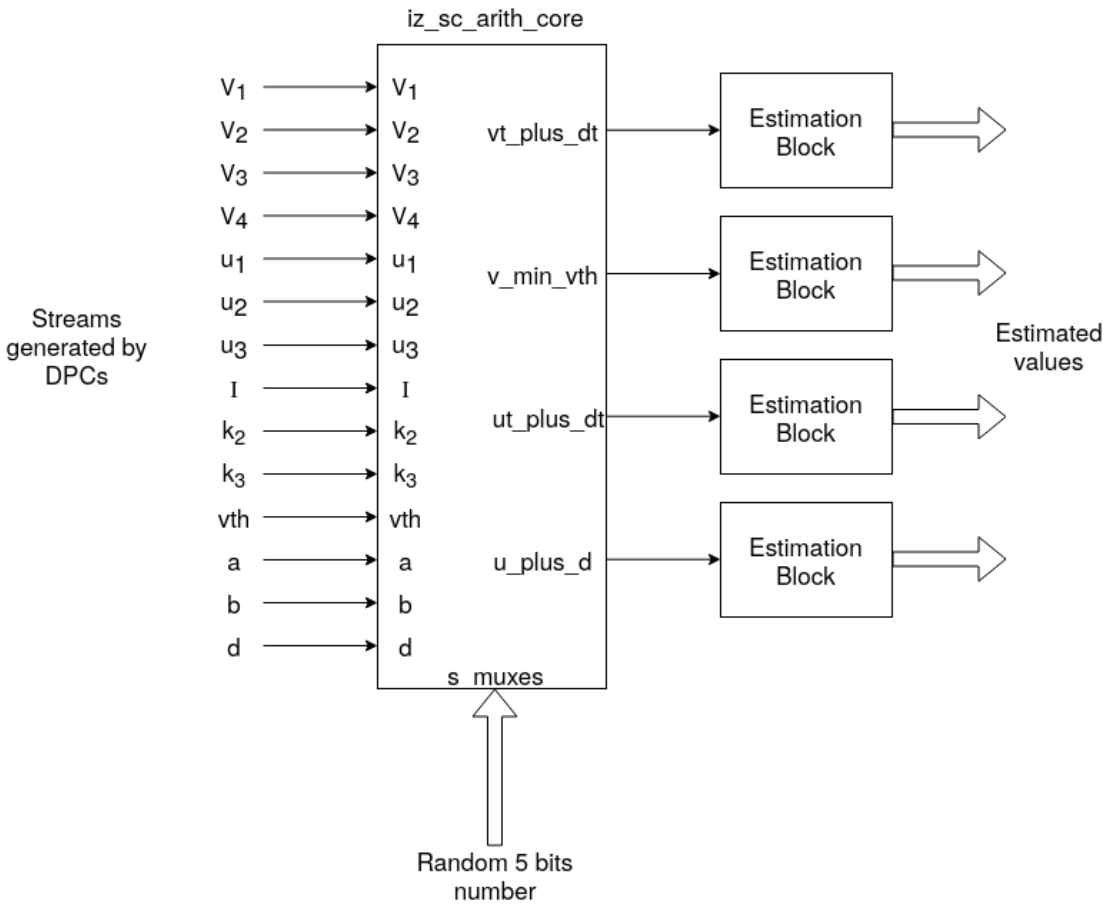

Figure 3.21: Datapath part for stochastic arithmetic computing and variable estimation.

The estimation blocks are saturating up-down counters. The overall number of stochastic variable needed to generate good and uncorrelated stochastic streams are 19, as it can be seen from the schematic.

3.5.3 Control unit for integration step management

The state machine which manages the steps of the stochastic Izhikevich neuron is reported in figure 3.22. The control signals used and their function are reported in

table 3.8.

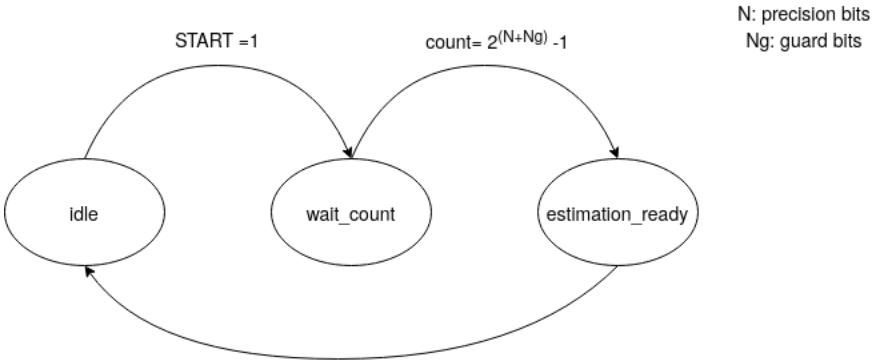

Figure 3.22: Finite state machine for the stochastic Izhikevich neuron control flow.

Control signal	width	Description
ARITH_COMMANDS	1	Resets the up-down counter estimators
REG_DECODER_ENABLE	1	Enables the datapath internal registers to save the new estimated values
RCS	1	Reset Command Strobes. If the reset condition occurs, the datapath set the SPIKE signal to '1'
DONE	1	It is '1' when an integration step is done. The correct spike evaluation is given by the SPIKE signal after a clock period, as it is for the previous implementations

Table 3.8: Stochastic control unit control signals

The values membrane voltage and recover variable with the use of stochastic arithmetic are reported in figure 3.23.

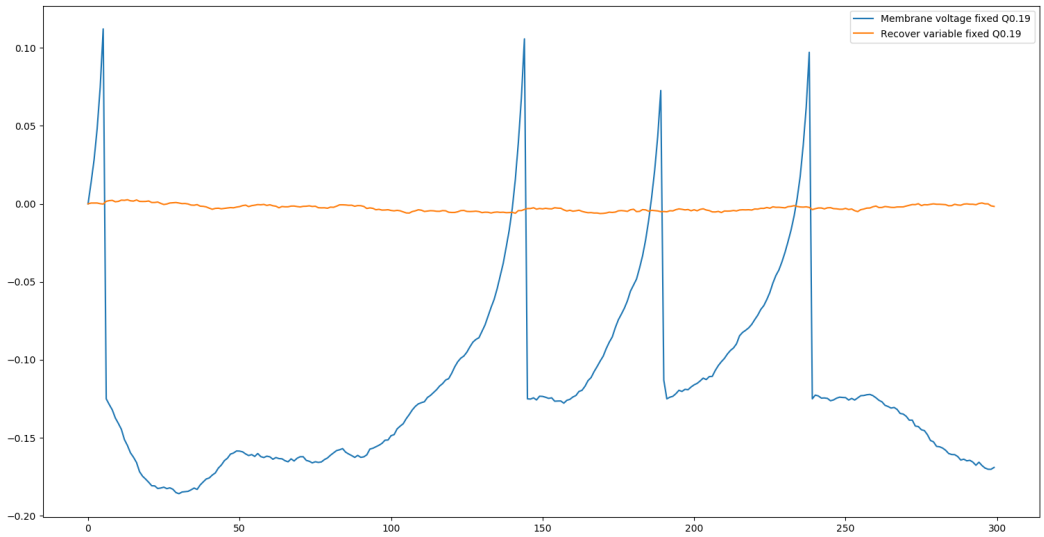

Figure 3.23: Membrane voltage and recover variable values of an Izhikevich neuron with class 2 excitability parameters without input current stimulus.

3.6 XOR Neural Network

In order to test the correct behavior of the neural model proposal, a Spike Neural Network is designed for inference tests. A brief summary about neural nets is presented in order to explain the main reasons behind further choices. The neural network designed recognizes the patterns of the XOR function, which is one of the most simple non-linear separable problems. The correct behavior of the neural net is test with a C program and the results are used for comparisons with the further VLSI implementatios.

Besides the Izhikevich-based neural networks, another purely stochastic implementation of the same topology is made. This architecture uses a classical approach of machine learning, as described in chapter 2 of [101] and the activation function of each neuron is the stanh function.

3.6.1 Neural net topology

Figure 3.24 reports the neural net topology used to implement the XOR function.

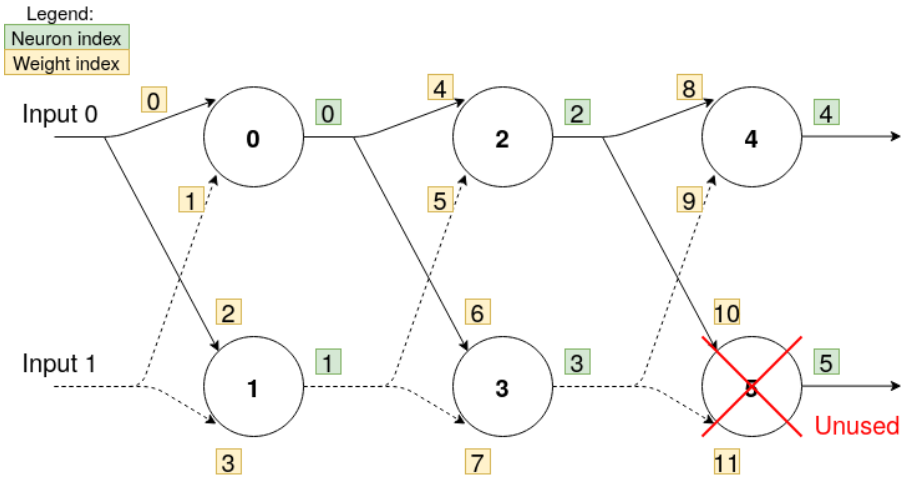

Figure 3.24: Neural net topology used to implement the xor function.

For each weight and neuron an index is assigned in order to reference them when the net is implemented with a high level programming language.

3.6.2 Xor C program comparison variable generation

Similarly to the C program written to compare the fixed point model with the floating point one in the first part of the design, a preliminary procedure is made

to generate all the stimuli. The high level algorithm is described by two flowcharts and can be implemented by procedural approach or in a object oriented manner. The first flowchart, which is shown in figure 3.25, reports the macro steps done to setup all the neural net components and to make it ready to receive the input patterns of the xor functions.

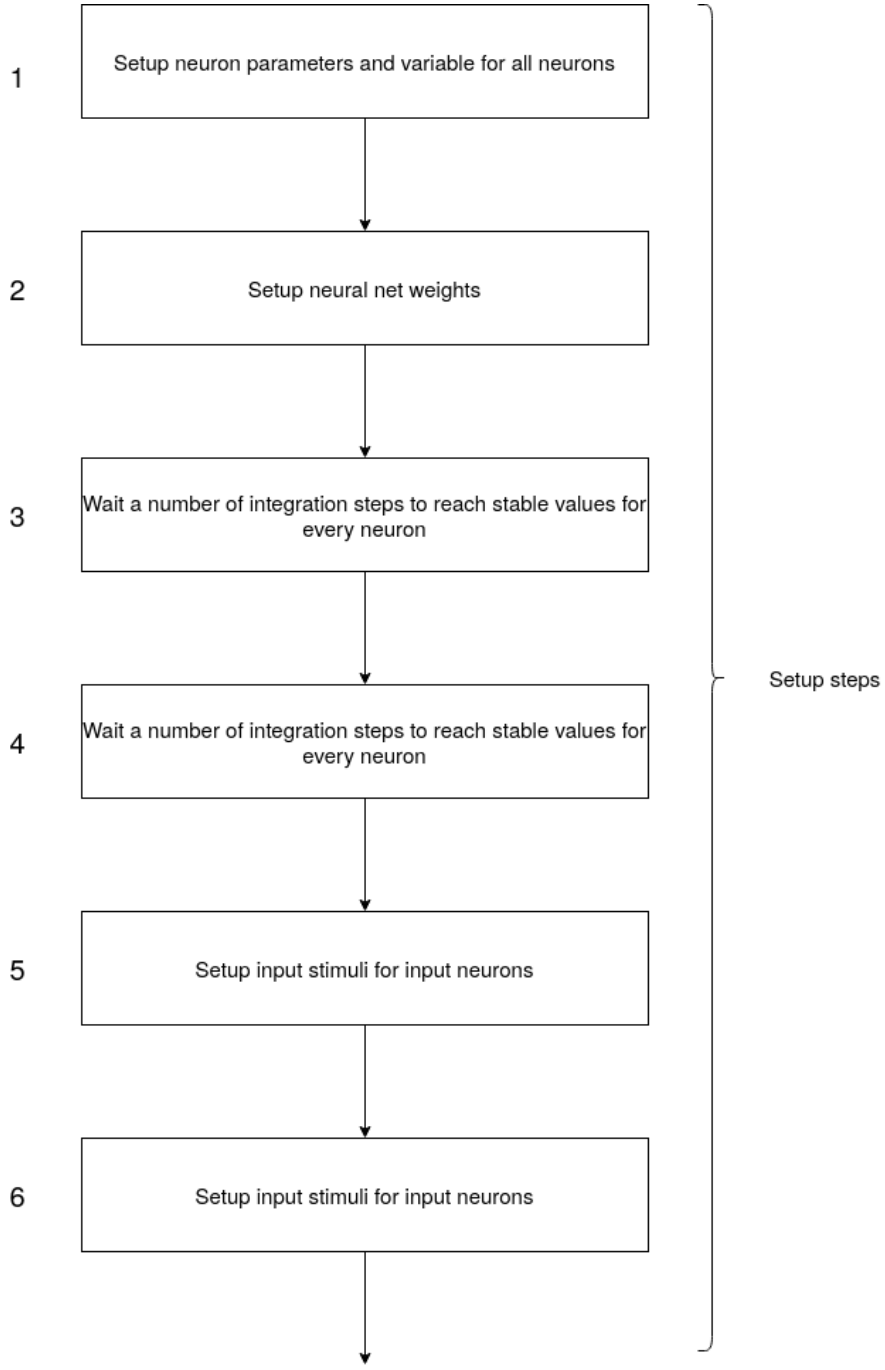

Figure 3.25: Flowchart for proper XOR neural net setup.

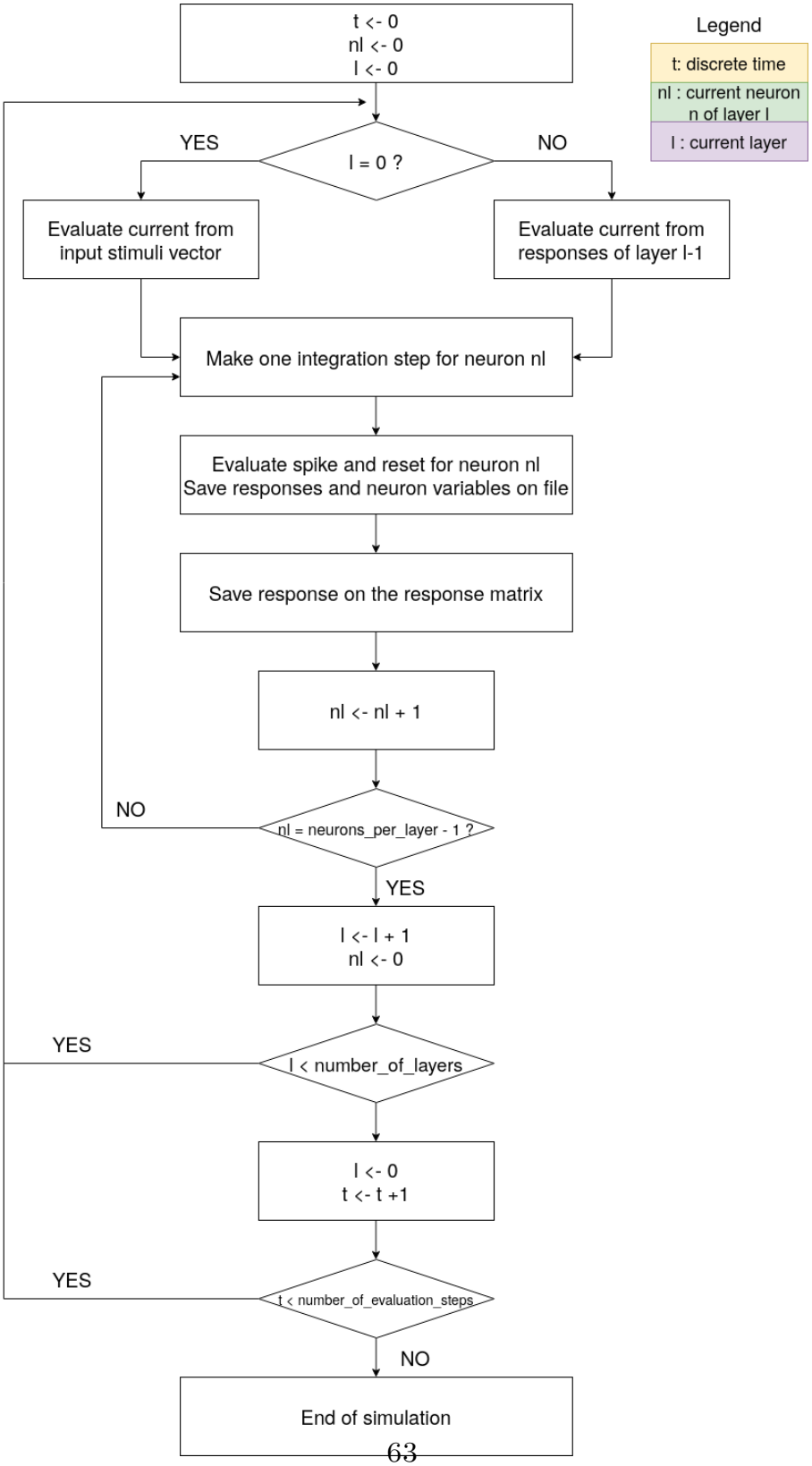

Figure 3.26: Flowchart for xor neural net integration steps evaluation.

3.6.3 Hardware implementation

After the performing of all high level testing for the correct behavior of the neural net, several hardware implementations are made in order to compare the evolution of all metrics of the neuron when these are used in a neural net.

The general schematic block of the neural net is report in figure 3.27.

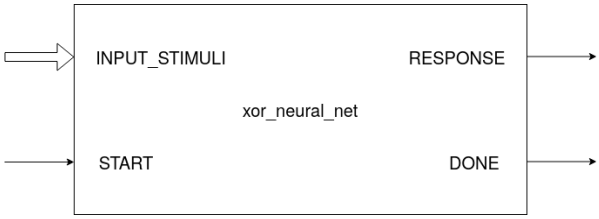

Figure 3.27: General schematic of the XOR neural net.

This scheme is used for all the implementations of the Izhikevich neuron. The behavior of the signals are described in the timing diagram in figure 3.28. Each layer of the neural net follows this timing diagram, too.

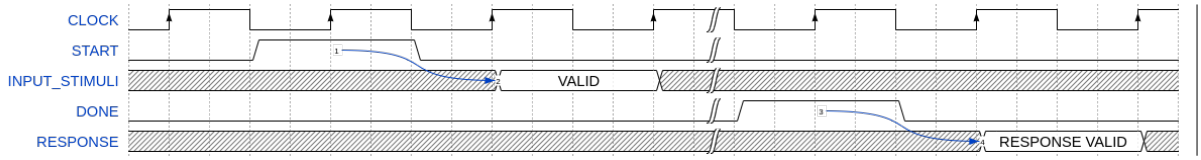

Figure 3.28: XOR neural net timing diagram.

In order to evaluate the stimulus current I for each neuron, the response of the previous layer (or the input) must be evaluated. The formula to evaluate the stimulus current in biological-plausible models is given by the (3.20):

$$I = \delta_0 w_0 + \delta_1 w_1 \tag{3.20}$$

These inputs δ_1 and δ_2 correspond to the presynaptic stimulus, which are the SPIKE signals of the corresponding neuron.

3.6.4 Deterministic presynaptic unit

For all Izhikevich implementations the presynaptic unit in figure 3.29 is used to evaluate the input stimuli current. The outputs consist of a DONE signal for the layer uCU and the current I for the corresponding neuron.

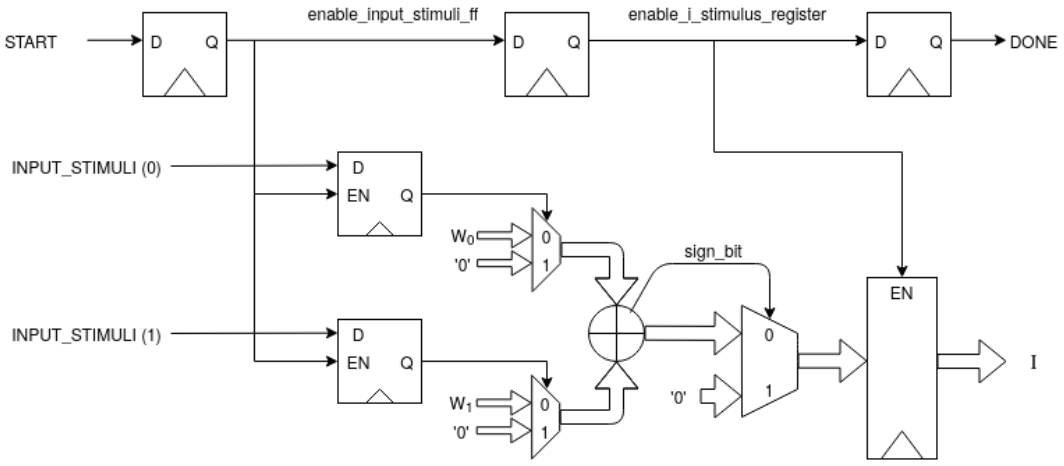

Figure 3.29: Presynaptic block used to evaluate the input stimuli current of each Izhikevich neuron.

3.6.5 The stochastic presynaptic unit for stanh solution

For the stanh neural net, the input of each neuron is given by a presynaptic unit, whose output is given by the (3.21):

$$I_{stanh} = \frac{\frac{x_1 w_1 + x_2 w_2}{2} - \phi}{2} \tag{3.21}$$

where ϕ is the bias. The random sequences to generate these values are 5: 2 random sequences for the weights, 1 for the bias and 2 stochastic bitstream for the scaled addition and subtraction. The scaled addition and subtraction are used in order to perform the operation directly with these probabilities. The schematic implementing the (3.21) is reported in figure 3.30.

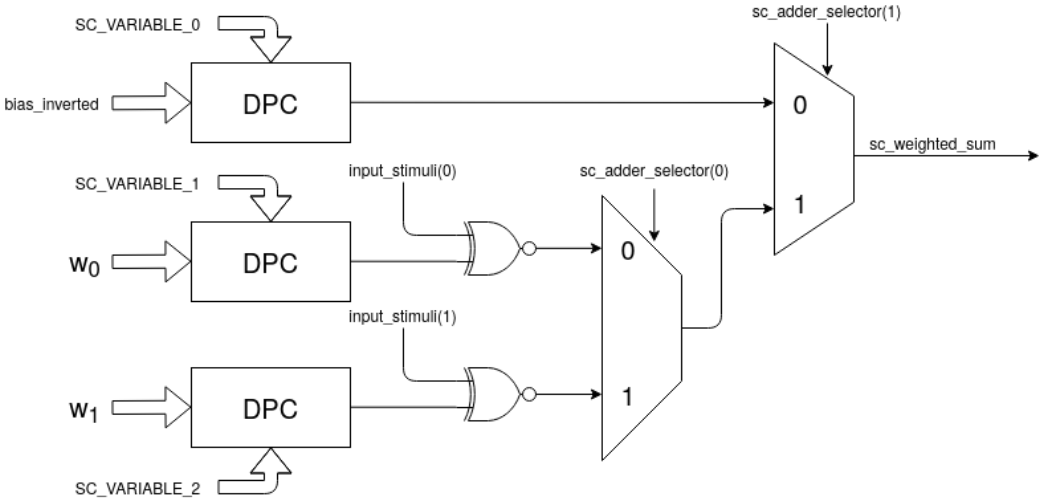

Figure 3.30

Chapter 4

Synthesis of the architectures

After the implementation of different architectures for the same Izhikevich neuron algorithm, a set of analyses are performed to evaluate area occupation and power consumption. In order to estimate these metrics, synthesis and power switching activities simulation are performed with the use of Modelsim [102] and Synopsys Design Compiler [103]. The technology libraries used are the Nangate45 Open Cell and the UMC 65 nm ones.

4.1 Arithmetic core area estimation

An important area comparison is made between the arithmetic cores, without taking into account all the support circuitry not involved directly in the computation tastks such as storing registers, linear feedback shift registers, DPCs. Table 4.1 reports the results found for each implementation. In this preliminary analysis there is no estimation about the area occupation of the interconnections and containers.

	Arithmetic blocks			
Area	DFG(32 bits)	Single adder	Single fulladder	Stochastic
Combinational	11171.20	382.51	265.734	27.398
Non combinational	516.04	430.92	442.89	0
Buffer & inverter	1120.126	43.624	32.452	2.66
Total cell area	11687.24	813.428	708.624	27.398

Table 4.1: Area comparison of the arithmetic blocks with the Nangate Open Cell libraries.

4.2 Xor neural net area estimation

For each implementation of the xor neural net an area estimation is derived and reported in table 4.2.

	Xor neural net type			
Area	DFG(20 bits)	Single adder	Single fulladder	Stochastic
Combinational	21423.64	4895.2	4263.98	7574.62
Non combinational	2218.44	6756.40	6908.02	5000.8
Buffer & inverter	2794	678.57	597.17	1491.728
Total cell area	23642.08	11651.6	11172.00	12575.42

Table 4.2: Area comparison of the xor neural nets.

	Xor neural net type				
Area	DFG(20 bits)	Single adder	Single fulladder	Stochastic (11 bits)	Stanh
Combinational	59930.64	8194.32	6820.56	7630.92	548.64
Non combinational	4041.36	12303.36	12281.76	4559.76	205.2
Buffer & inverter	2851.92	1476	1248	724.68	43.2
Total cell area	63972	20497	19102.32	12190.68	753.84

Table 4.3: Area comparison of the xor neural nets UMC libraries at 65 nm.

Area of a 32-bit LFSR
262.44

Table 4.4: Area of a 32-bit LFSR with UMC 65 nm libraries.

4.3 Switching activities and Power Consumption

After the all synthesis are made, power switching activities simulations are made in order to estimate the power consumption of each architecture.

4.3.1 Arithmetic core power consumption

The first comparison involves the arithmetic cores only. In table 4.5 are reported the results for each arithmetic core.

Consumption type	Architectures			
	DFG(32 bits)	Single adder	Single fulladder	stochastic
Internal (register)	36.3945 uW	28.1429 uW	29.4209 uW	0 uW
Internal (combinational)	504.4610 uW	1.7150 uW	0.6697 uW	1.1608 uW
Switching (register)	10.2681 uW	0.1871 uW	0.4547 uW	0 uW
Switching (combinational)	667.9446 uW	3.6186 uW	0.8390 uW	0.3878 uW
Leakage (register)	8.4576 uW	6.9459 uW	7.1918 uW	0 uW
Leakage (combinational)	266.28 uW	10.802 uW	8.5090 uW	580.1469 nW
Total power	1.4938 mW	51.4115 uW	47.0851 uW	2.1287 uW

Table 4.5: Power consumption comparison from switching activities of the arithmetic cores. Support registers are included in the analysis if needed by the architecture itself.

4.3.2 Xor neural net power consumption

Consumption type	Xor neural net architectures			
	DFG(20 bits)	Single adder	Single fulladder	stochastic
Internal (register)	125.55 uW	362.644 uW	357.9015 uW	474.83 uW
Internal (combinational)	31.44 uW	14.934 uW	6.7146 uW	102.2592 uW
Switching (register)	4.89 uW	6.03 uW	5.4755 uW	6.1491 uW
Switching (combinational)	40.45 uW	5.3701 uW	0.8390 uW	104.37 uW
Leakage (register)	36.64 uW	111.36 uW	113.95 uW	78.827 uW
Leakage (combinational)	523.35 uW	116.73 uW	103.88 uW	153.94 uW
Total power	762.31 uW	632.28 uW	593.2879 uW	924.3733 uW

Table 4.6: Power consumption comparison from switching activities of the xor neural nets.

	Xor neural net type				
Power	DFG(20 bits)	Single adder	Single fulladder	Stochastic (11 bits)	Stanh
Internal (register)	176.9 uW	561 uW	568.6 uW	281.9 uW	15.8 uW
Internal (combinational)	304.2 uW	14 uW	5.632 uW	68 uW	6.097 uW
Switching (register)	4.7583 uW	3.3 uW	2.83 uW	4.466 uW	6.268 nW
Switching (combinational)	243.7 uW	15.3 uW	4.594 uW	44.06 uW	3.25 uW
Leakage (register)	0.301 uW	0.9 uW	0.923 uW	322 nW	13.268 nW
Leakage (combinational)	5.51 uW	0.6 uW	0.457 uW	630 nW	42.836 nW
Total power	735.4 uW	595.2 uW	583.1 uW	399.4 uW	25.218 uW

Table 4.7: Power consumption comparison of the xor neural nets UMC libraries at 65 nm.

4.3.3 LFSR power consumption

The power consumption for a single 32-bit LFSR is reported in table 4.8.

Power consumption LFSR at 32-bits
25.218 uW

Table 4.8: 32-bit LFSR power consumption with UMC 65 nm

4.3.4 Energy consumption comparison between the Izhikevich neurons

Another interesting metric to be evaluated is the energy consumption needed to make an integration step of the Izhikevich equations. Table 4.9 shows the energy required for each neuron to make an integration step.

Izhikevich neuron architecture	Clock cycles	Energy consumption
DFG(20 bits)	1	3.3 pJ
Single Adder	141	349.962 pJ
Single fulladder	1433	3.32 nJ
Stochastic	2049	2.39 nJ

Table 4.9: Energy consumption and clock cycles required for one integration step of each implementation of the Izhikevich neuron with UMC 65 nm.

4.3.5 Energy consumption comparison between the XOR neural nets

The energy consumption for one integration step of each XOR neural network is reported in table 4.10.

XOR neural net architecture	Energy consumption for 1 integration step
DFG(20 bits)	44.124 pJ
Single Adder	5.035 nJ
Single fulladder	50.134 nJ
Stochastic	49.10 nJ
Stanh	1.44 nJ

Table 4.10: Energy consumption for one integration step of each implementation of the XOR neural net with UMC 65 nm.

For each XOR neural net the amount of energy to evaluate the response of the neural net is estimated and reported in table 4.11. The estimation takes into account the number of integration steps needed to evaluate correctly the neural net response to a specific input pattern.

XOR neural net architecture	Energy consumption for 1 complete evaluation
DFG(20 bits)	44.124 nJ
Single Adder	5.035 uJ
Single fulladder	50.134 uJ
Stochastic	49.10 uJ
Stanh	1.44 nJ

Table 4.11: Energy consumption for one evaluation of each implementation of the XOR neural net with UMC 65 nm.

Bibliography

[1] T. Yang, Y. Chen, Emer J., and V. Sze. «A Method to Estimate the Energy Consumption of Deep Neural Network». In: *IEEE* (2017) (cit. on p. 1).

[2] A.L. Hodgkin and A.F. Huxley. «A quatitative description of membrane current and its application to conduction and excitation in nerve». In: *The Journal of physiology* (1952) (cit. on pp. 2, 5, 6).

[3] G. Monegato. *Metodi e algoritmi per il calcolo numerico*. Clut Editrice, 2008 (cit. on pp. 7, 21).

[4] E.M. Izhikevich. «Which Model to Use for Cortical Spiking Neurons?» In: *IEEE transactions on neural networks* (2004) (cit. on pp. 7, 8).

[5] R. Fitzhug. «Impulses and Physiological States in Theoretical models of Nerve Membrane». In: *Biophysical Journal* (1961) (cit. on p. 7).

[6] J. Nagumo. «Impulses and Physiological States in Theoretical models of Nerve Membrane». In: *Biophysical Journal* (1962) (cit. on p. 7).

[7] E.M. Izhikevich. «Simple Model of Spiking Neurons». In: *IEEE transactions on neural networks* (2003) (cit. on pp. 8, 51).

[8] B.D. Brown and H.C. Card. «Stochastic Neural Computation 1: computation Elements». In: *IEEE* (2001) (cit. on pp. 9, 13, 17, 18).

[9] A. Alaghi and J.P. Hayes. «Survey of Stochastic Computing». In: *ACM Transactions on Embedded Computing Systems* (2013) (cit. on pp. 9, 10).

[10] A. Alaghi, W. Qian, and J. P. Hayes. «The Promise and Challenge of Stochastic Computing». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.8 (2018), pp. 1515–1531. DOI: 10.1109/TCAD.2017.2778107 (cit. on p. 9).

[11] M. Alawad and M. Lin. «Survey of Stochastic-Based Computation Paradigms». In: *IEEE Transactions on Emerging Topics in Computing* 7.1 (2019), pp. 98–114. DOI: 10.1109/TETC.2016.2598726 (cit. on p. 9).

[12] S. Brown and Z. Vranesic. *Fundamentals of Digital Logic with VHDL design*. McGraw Hill, 2009 (cit. on pp. 9, 24, 28).

[13] Sheldon M. Ross. *Introduction to probability and statistics for engineers and scientists*. Apogeo, 2014 (cit. on pp. 10, 13).

[14] B.R. Gaines. «Stochastic Computing Systems». In: *Advances in Information Systems Science, J.F.Tou, ed, vol.2, cap.2, pp.37-172, New York, Plenum* (1969) (cit. on pp. 11, 14, 18).

[15] https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/ *Advanced Synthesis Cookbook*. Altera Corporation, 2011 (cit. on p. 14).

[16] F. Neugebauer, I. Polian, and J. P. Hayes. «Building a Better Random Number Generator for Stochastic Computing». In: *2017 Euromicro Conference on Digital System Design (DSD)*. 2017, pp. 1–8. DOI: 10.1109/DSD.2017.29 (cit. on p. 15).

[17] Y. Ding, Y. Wu, and W. Qian. «Generating multiple correlated probabilities for MUX-based stochastic computing architecture». In: *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2014, pp. 519–526. DOI: 10.1109/ICCAD.2014.7001400 (cit. on p. 15).

[18] H. Ichihara, S. Ishii, D. Sunamori, T. Iwagaki, and T. Inoue. «Compact and accurate stochastic circuits with shared random number sources». In: *2014 IEEE 32nd International Conference on Computer Design (ICCD)*. 2014, pp. 361–366. DOI: 10.1109/ICCD.2014.6974706 (cit. on p. 15).

[19] R. Ishikawa, M. Tawada, M. Yanagisawa, and N. Togawa. «An Effective Stochastic Number Duplicator and Its Evaluations Using Composite Arithmetic Circuits». In: *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*. 2018, pp. 53–56. DOI: 10.1109/IOLTS.2018.8474263 (cit. on p. 15).

[20] R. Ishikawa, M. Tawada, M. Yanagisawa, and N. Togawa. «2n RRR: Improved Stochastic Number Duplicator Based on Bit Re-Arrangement». In: *2018 New Generation of CAS (NGCAS)*. 2018, pp. 182–185. DOI: 10.1109/NGCAS.2018.8572289 (cit. on p. 15).

[21] Z. Li, Z. Chen, Y. Zhang, Z. Huang, and W. Qian. «Simultaneous Area and Latency Optimization for Stochastic Circuits by D Flip-Flop Insertion». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.7 (2019), pp. 1251–1264. DOI: 10.1109/TCAD.2018.2846660 (cit. on p. 15).

[22] K.P. Keshab. «Analysis of Stochastic Logic Circuits in Unipolar, Bipolar and Hybrid Formats». In: *IEEE* (2017) (cit. on p. 16).

[23] F. Neugebauer, I. Polian, and J. P. Hayes. «Framework for quantifying and managing accuracy in stochastic circuit design». In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. 2017, pp. 1–6. DOI: 10.23919/DATE.2017.7926949 (cit. on p. 16).

[24] T. Chen and J. P. Hayes. «Analyzing and controlling accuracy in stochastic circuits». In: *2014 IEEE 32nd International Conference on Computer Design (ICCD)*. 2014, pp. 367–373. DOI: 10.1109/ICCD.2014.6974707 (cit. on p. 16).

[25] M. Parhi, M. D. Riedel, and K. K. Parhi. «Effect of bit-level correlation in stochastic computing». In: *2015 IEEE International Conference on Digital Signal Processing (DSP)*. 2015, pp. 463–467. DOI: 10.1109/ICDSP.2015.7251915 (cit. on p. 16).

[26] C. Ma, S. Zhong, and H. Dang. «Understanding variance propagation in stochastic computing systems». In: *2012 IEEE 30th International Conference on Computer Design (ICCD)*. 2012, pp. 213–218. DOI: 10.1109/ICCD.2012.6378643 (cit. on p. 16).

[27] A. Alaghi and J. P. Hayes. «Exploiting correlation in stochastic circuit design». In: *2013 IEEE 31st International Conference on Computer Design (ICCD)*. 2013, pp. 39–46. DOI: 10.1109/ICCD.2013.6657023 (cit. on p. 16).

[28] P. Ting and J. P. Hayes. «Isolation-based decorrelation of stochastic circuits». In: *2016 IEEE 34th International Conference on Computer Design (ICCD)*. 2016, pp. 88–95. DOI: 10.1109/ICCD.2016.7753265 (cit. on p. 16).

[29] M. van der Hagen and M. Riedel. «Unary positional computing». In: *2017 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. 2017, pp. 1335–1339. DOI: 10.1109/GlobalSIP.2017.8309178 (cit. on p. 16).

[30] D. Jenson and M. Riedel. «A deterministic approach to stochastic computation». In: *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2016, pp. 1–8. DOI: 10.1145/2966986.2966988 (cit. on p. 16).

[31] M. H. Najafi, D. Jenson, D. J. Lilja, and M. D. Riedel. «Performing Stochastic Computation Deterministically». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.12 (2019), pp. 2925–2938. DOI: 10.1109/TVLSI.2019.2929354 (cit. on p. 16).

[32] M. H. Najafi, D. J. Lilja, M. D. Riedel, and K. Bazargan. «Polysynchronous Clocking: Exploiting the Skew Tolerance of Stochastic Circuits». In: *IEEE Transactions on Computers* 66.10 (2017), pp. 1734–1746. DOI: 10.1109/TC.2017.2697881 (cit. on p. 17).

[33] P. Li, D. J. Lilja, W. Qian, M. D. Riedel, and K. Bazargan. «Logical Computation on Stochastic Bit Streams with Linear Finite-State Machines». In: *IEEE Transactions on Computers* 63.6 (2014), pp. 1474–1486. DOI: 10.1109/TC.2012.231 (cit. on p. 17).

[34] H. Sim, D. Nguyen, J. Lee, and K. Choi. «Scalable stochastic-computing accelerator for convolutional neural networks». In: *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2017, pp. 696–701. DOI: 10.1109/ASPDAC.2017.7858405 (cit. on p. 19).

[35] K. Sanni, G. Garreau, J. L. Molin, and A. G. Andreou. «FPGA implementation of a Deep Belief Network architecture for character recognition using stochastic computation». In: *2015 49th Annual Conference on Information Sciences and Systems (CISS)*. 2015, pp. 1–5. DOI: 10.1109/CISS.2015.7086904 (cit. on p. 19).

[36] N. Nedjah and L. de Macedo Mourelle. «Stochastic reconfigurable hardware for neural networks». In: *Euromicro Symposium on Digital System Design, 2003. Proceedings.* 2003, pp. 438–442. DOI: 10.1109/DSD.2003.1231979 (cit. on p. 19).

[37] L. Huang, G. Chen, P. Li, and W. Qian. «Accelerating stochastic computation for binary classification applications». In: *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2016, pp. 6530–6534. DOI: 10.1109/ICASSP.2016.7472935 (cit. on p. 19).

[38] Z. Li, A. Ren, J. Li, Q. Qiu, Y. Wang, and B. Yuan. «DSCNN: Hardware-oriented optimization for Stochastic Computing based Deep Convolutional Neural Networks». In: *2016 IEEE 34th International Conference on Computer Design (ICCD)*. 2016, pp. 678–681. DOI: 10.1109/ICCD.2016.7753357 (cit. on p. 19).

[39] H. Li, T. Wei, A. Ren, Q. Zhu, and Y. Wang. «Deep reinforcement learning: Framework, applications, and embedded implementations: Invited paper». In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2017, pp. 847–854. DOI: 10.1109/ICCAD.2017.8203866 (cit. on p. 19).

[40] A. Ren, Z. Li, Y. Wang, Q. Qiu, and B. Yuan. «Designing reconfigurable large-scale deep learning systems using stochastic computing». In: *2016 IEEE International Conference on Rebooting Computing (ICRC)*. 2016, pp. 1–7. DOI: 10.1109/ICRC.2016.7738685 (cit. on p. 19).

[41] Da Zhang, H. Li, and S. Y. Foo. «A simplified FPGA implementation of neural network algorithms integrated with stochastic theory for power electronics applications». In: *31st Annual Conference of IEEE Industrial Electronics Society, 2005. IECON 2005.* 2005, pp. 1–6. DOI: 10.1109/IECON.2005.1569044 (cit. on p. 19).

[42] J. L. Rosselló, V. Canals, and A. Morro. «Probabilistic-based neural network implementation». In: *The 2012 International Joint Conference on Neural Networks (IJCNN).* 2012, pp. 1–7. DOI: 10.1109/IJCNN.2012.6252807 (cit. on p. 19).

[43] J. Li, Z. Yuan, Z. Li, C. Ding, A. Ren, Q. Qiu, J. Draper, and Y. Wang. «Hardware-driven nonlinear activation for stochastic computing based deep convolutional neural networks». In: *2017 International Joint Conference on Neural Networks (IJCNN).* 2017, pp. 1230–1236. DOI: 10.1109/IJCNN.2017.7965993 (cit. on p. 19).

[44] A. Ardakani, F. Leduc-Primeau, N. Onizawa, T. Hanyu, and W. J. Gross. «VLSI Implementation of Deep Neural Network Using Integral Stochastic Computing». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.10 (2017), pp. 2688–2699. DOI: 10.1109/TVLSI.2017.2654298 (cit. on p. 19).

[45] M. Ranjbar, M. E. Salehi, and M. H. Najafi. «Using stochastic architectures for edge detection algorithms». In: *2015 23rd Iranian Conference on Electrical Engineering.* 2015, pp. 723–728. DOI: 10.1109/IranianCEE.2015.7146308 (cit. on p. 19).

[46] D. Zhang and H. Li. «A low cost digital implementation of feed-forward neural networks applied to a variable-speed wind turbine system». In: *2006 37th IEEE Power Electronics Specialists Conference.* 2006, pp. 1–6. DOI: 10.1109/pesc.2006.1712272 (cit. on p. 19).

[47] V. T. Lee, A. Alaghi, J. P. Hayes, V. Sathe, and L. Ceze. «Energy-efficient hybrid stochastic-binary neural networks for near-sensor computing». In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2017.* 2017, pp. 13–18. DOI: 10.23919/DATE.2017.7926951 (cit. on pp. 19, 20).

[48] Z. Li, A. Ren, J. Li, Q. Qiu, B. Yuan, J. Draper, and Y. Wang. «Structural design optimization for deep convolutional neural networks using stochastic computing». In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2017.* 2017, pp. 250–253. DOI: 10.23919/DATE.2017.7926991 (cit. on pp. 19, 20).

[49] E. M. Petriu, A. Cretu, and P. Payeur. «Neural Network Modeling Techniques for the Real-Time Rendering of the Geometry and Elasticity of 3D Objects». In: *2007 2nd International Workshop on Soft Computing Applications*. 2007, pp. 11–16. DOI: 10.1109/SOFA.2007.4318297 (cit. on p. 19).

[50] S. C. Smithson, K. Boga, A. Ardakani, B. H. Meyer, and W. J. Gross. «Stochastic Computing Can Improve Upon Digital Spiking Neural Networks». In: *2016 IEEE International Workshop on Signal Processing Systems (SiPS)*. 2016, pp. 309–314. DOI: 10.1109/SiPS.2016.61 (cit. on p. 19).

[51] D. Cannisi and B. Yuan. «Design Space Exploration for K-Nearest Neighbors Classification Using Stochastic Computing». In: *2016 IEEE International Workshop on Signal Processing Systems (SiPS)*. 2016, pp. 321–326. DOI: 10.1109/SiPS.2016.63 (cit. on p. 19).

[52] Y. Xie, S. Liao, B. Yuan, Y. Wang, and Z. Wang. «Fully-Parallel Area-Efficient Deep Neural Network Design Using Stochastic Computing». In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 64.12 (2017), pp. 1382–1386. DOI: 10.1109/TCSII.2017.2746749 (cit. on p. 19).

[53] D. Zhang and H. Li. «A Stochastic-Based FPGA Controller for an Induction Motor Drive With Integrated Neural Network Algorithms». In: *IEEE Transactions on Industrial Electronics* 55.2 (2008), pp. 551–561. DOI: 10.1109/TIE.2007.911946 (cit. on p. 19).

[54] S. Sato, K. Nemoto, S. Akimoto, M. Kinjo, and K. Nakajima. «Implementation of a new neurochip using stochastic logic». In: *IEEE Transactions on Neural Networks* 14.5 (2003), pp. 1122–1127. DOI: 10.1109/TNN.2003.816341 (cit. on p. 19).

[55] L. Geretti and A. Abramo. «The Correspondence Between Deterministic and Stochastic Digital Neurons: Analysis and Methodology». In: *IEEE Transactions on Neural Networks* 19.10 (2008), pp. 1739–1752. DOI: 10.1109/TNN.2008.2001775 (cit. on p. 19).

[56] V. Canals, A. Morro, A. Oliver, M. L. Alomar, and J. L. Rosselló. «A New Stochastic Computing Methodology for Efficient Neural Network Implementation». In: *IEEE Transactions on Neural Networks and Learning Systems* 27.3 (2016), pp. 551–564. DOI: 10.1109/TNNLS.2015.2413754 (cit. on p. 19).

[57] H. Li, D. Zhang, and S. Y. Foo. «A Stochastic Digital Implementation of a Neural Network Controller for Small Wind Turbine Systems». In: *IEEE Transactions on Power Electronics* 21.5 (2006), pp. 1502–1507. DOI: 10.1109/TPEL.2006.882420 (cit. on p. 19).

[58] A. Ardakani, F. Leduc-Primeau, N. Onizawa, T. Hanyu, and W. J. Gross. «VLSI Implementation of Deep Neural Network Using Integral Stochastic Computing». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.10 (2017), pp. 2688–2699. DOI: 10.1109/TVLSI.2017.2654298 (cit. on p. 19).

[59] K. Kim and K. Choi. «Synthesis of multi-variate stochastic computing circuits». In: *2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. 2017, pp. 1–6. DOI: 10.1109/VLSI-SoC.2017.8203493 (cit. on p. 19).

[60] R. Xu, B. Yuan, X. You, and C. Zhang. «Efficient fast convolution architecture based on stochastic computing». In: *2017 9th International Conference on Wireless Communications and Signal Processing (WCSP)*. 2017, pp. 1–6. DOI: 10.1109/WCSP.2017.8171031 (cit. on p. 19).

[61] A. G. Andreou et al. «Bio-inspired system architecture for energy efficient, BIGDATA computing with application to wide area motion imagery». In: *2016 IEEE 7th Latin American Symposium on Circuits Systems (LASCAS)*. 2016, pp. 1–6. DOI: 10.1109/LASCAS.2016.7450995 (cit. on p. 19).

[62] V. Canals, M. L. Alomar, A. Morro, A. Oliver, and J. L. Rossello. «Noise-robust hardware implementation of neural networks». In: *2015 International Joint Conference on Neural Networks (IJCNN)*. 2015, pp. 1–8. DOI: 10.1109/IJCNN.2015.7280622 (cit. on p. 19).

[63] Y. Liu, Y. Wang, F. Lombardi, and J. Han. «An energy-efficient stochastic computational deep belief network». In: *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2018, pp. 1175–1178. DOI: 10.23919/DATE.2018.8342191 (cit. on p. 19).

[64] Y. Liu, S. Liu, Y. Wang, F. Lombardi, and J. Han. «A Stochastic Computational Multi-Layer Perceptron with Backward Propagation». In: *IEEE Transactions on Computers* 67.9 (2018), pp. 1273–1286. DOI: 10.1109/TC.2018.2817237 (cit. on p. 19).

[65] Y. Liu, Y. Wang, F. Lombardi, and J. Han. «An Energy-Efficient Online-Learning Stochastic Computational Deep Belief Network». In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 8.3 (2018), pp. 454–465. DOI: 10.1109/JETCAS.2018.2852705 (cit. on p. 19).

[66] S. Liu, H. Jiang, L. Liu, and J. Han. «Gradient Descent Using Stochastic Circuits for Efficient Training of Learning Machines». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.11 (2018), pp. 2530–2541. DOI: 10.1109/TCAD.2018.2858363 (cit. on p. 19).

[67] V. Nguyen, T. Luong, H. Le Duc, and V. Hoang. «An Efficient Hardware Implementation of Activation Functions Using Stochastic Computing for Deep Neural Networks». In: *2018 IEEE 12th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. 2018, pp. 233–236. DOI: 10.1109/MCSoC2018.2018.00045 (cit. on p. 19).

[68] Z. Li et al. «HEIF: Highly Efficient Stochastic Computing-Based Inference Framework for Deep Neural Networks». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.8 (2019), pp. 1543–1556. DOI: 10.1109/TCAD.2018.2852752 (cit. on pp. 19, 20).

[69] Y. Liu, L. Liu, F. Lombardi, and J. Han. «An Energy-Efficient and Noise-Tolerant Recurrent Neural Network Using Stochastic Computing». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.9 (2019), pp. 2213–2221. DOI: 10.1109/TVLSI.2019.2920152 (cit. on p. 19).

[70] Y. Zhang, X. Zhang, J. Song, Y. Wang, R. Huang, and R. Wang. «Parallel Convolutional Neural Network (CNN) Accelerators Based on Stochastic Computing». In: *2019 IEEE International Workshop on Signal Processing Systems (SiPS)*. 2019, pp. 19–24. DOI: 10.1109/SiPS47522.2019.9020615 (cit. on p. 19).

[71] P. Ting and J. P. Hayes. «Stochastic Logic Realization of Matrix Operations». In: *2014 17th Euromicro Conference on Digital System Design*. 2014, pp. 356–364. DOI: 10.1109/DSD.2014.75 (cit. on p. 19).

[72] P. Li, W. Qian, and D. J. Lilja. «A stochastic reconfigurable architecture for fault-tolerant computation with sequential logic». In: *2012 IEEE 30th International Conference on Computer Design (ICCD)*. 2012, pp. 303–308. DOI: 10.1109/ICCD.2012.6378656 (cit. on p. 19).

[73] N. Saraf, K. Bazargan, D. J. Lilja, and M. D. Riedel. «Stochastic functions using sequential logic». In: *2013 IEEE 31st International Conference on Computer Design (ICCD)*. 2013, pp. 507–510. DOI: 10.1109/ICCD.2013.6657094 (cit. on p. 19).

[74] B. Yuan and K. K. Parhi. «Belief propagation decoding of polar codes using stochastic computing». In: *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2016, pp. 157–160. DOI: 10.1109/ISCAS.2016.7527194 (cit. on p. 19).

[75] L. Miao and C. Chakrabarti. «A parallel stochastic computing system with improved accuracy». In: *SiPS 2013 Proceedings*. 2013, pp. 195–200. DOI: 10.1109/SiPS.2013.6674504 (cit. on p. 19).

[76] S. Liu and J. Han. «Dynamic Stochastic Computing for Digital Signal Processing Applications». In: *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2020, pp. 604–609. DOI: 10.23919/DATE48585.2020.9116562 (cit. on p. 19).

[77] S. Liu, W. J. Gross, and J. Han. «Introduction to Dynamic Stochastic Computing». In: *IEEE Circuits and Systems Magazine* 20.3 (2020), pp. 19–33. DOI: 10.1109/MCAS.2020.3005483 (cit. on p. 19).

[78] W. J. Gross, V. C. Gaudet, and A. Milner. «Stochastic Implementation of LDPC Decoders». In: *Conference Record of the Thirty-Ninth Asilomar Conference onSignals, Systems and Computers, 2005.* 2005, pp. 713–717. DOI: 10.1109/ACSSC.2005.1599845 (cit. on p. 19).

[79] S. S. Tehrani, S. Mannor, and W. J. Gross. «Survey of Stochastic Computation on Factor Graphs». In: *37th International Symposium on Multiple-Valued Logic (ISMVL'07)*. 2007, pp. 54–54. DOI: 10.1109/ISMVL.2007.53 (cit. on p. 19).

[80] P. Li and D. J. Lilja. «A low power fault-tolerance architecture for the kernel density estimation based image segmentation algorithm». In: *ASAP 2011 - 22nd IEEE International Conference on Application-specific Systems, Architectures and Processors*. 2011, pp. 161–168. DOI: 10.1109/ASAP.2011.6043264 (cit. on p. 19).

[81] P. Li and D. J. Lilja. «Using stochastic computing to implement digital image processing algorithms». In: *2011 IEEE 29th International Conference on Computer Design (ICCD)*. 2011, pp. 154–161. DOI: 10.1109/ICCD.2011.6081391 (cit. on p. 19).

[82] B. Moons and M. Verhelst. «Energy-Efficiency and Accuracy of Stochastic Computing Circuits in Emerging Technologies». In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 4.4 (2014), pp. 475–486. DOI: 10.1109/JETCAS.2014.2361070 (cit. on p. 19).

[83] K. Boga, N. Onizawa, F. Leduc-Primeau, K. Matsumiya, T. Hanyu, and W. J. Gross. «Stochastic implementation of the disparity energy model for depth perception». In: *2015 IEEE Workshop on Signal Processing Systems (SiPS)*. 2015, pp. 1–6. DOI: 10.1109/SiPS.2015.7344982 (cit. on p. 19).

[84] Y. Liu and K. K. Parhi. «Architectures for stochastic normalized and modified lattice IIR filters». In: *2015 49th Asilomar Conference on Signals, Systems and Computers*. 2015, pp. 1351–1358. DOI: 10.1109/ACSSC.2015.7421363 (cit. on p. 19).

[85] S. Koshita, N. Onizawa, M. Abe, T. Hanyu, and M. Kawamata. «High-Precision Stochastic State-Space Digital Filters Based on Minimum Roundoff Noise Structure». In: *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2018, pp. 1–5. DOI: 10.1109/ISCAS.2018.8351186 (cit. on p. 19).

[86] D. K. McNeill, D. Zhao, C. Shafai, N. Chadha, A. Cuhadar, and H. C. Card. «Processing noisy analog signals from microsensor arrays and VLSI imagers using stochastic binary computations». In: *IEEE CCECE2002. Canadian Conference on Electrical and Computer Engineering. Conference Proceedings (Cat. No.02CH37373)*. Vol. 2. 2002, 986–990 vol.2. DOI: 10.1109/CCECE.2002.1013077 (cit. on p. 19).

[87] M. H. Najafi and D. J. Lilja. «High-speed stochastic circuits using synchronous analog pulses». In: *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2017, pp. 481–487. DOI: 10.1109/ASPDAC.2017.7858369 (cit. on p. 19).

[88] M. H. Najafi, S. Jamali-Zavareh, D. J. Lilja, M. D. Riedel, K. Bazargan, and R. Harjani. «An Overview of Time-Based Computing with Stochastic Constructs». In: *IEEE Micro* 37.6 (2017), pp. 62–71. DOI: 10.1109/MM.2017.4241345 (cit. on p. 19).

[89] L. C. Gouveia, T. J. Koickal, and A. Hamilton. «An Asynchronous Spike Event Coding Scheme for Programmable Analog Arrays». In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 58.4 (2011), pp. 791–799. DOI: 10.1109/TCSI.2010.2089552 (cit. on p. 19).

[90] R. P. Duarte and H. C. Neto. «Stochastic Processors on FPGAs to Compute Sensor Data Towards Fault-Tolerant IoT Systems». In: *2018 IEEE Conference on Dependable and Secure Computing (DSC)*. 2018, pp. 1–8. DOI: 10.1109/DESEC.2018.8625153 (cit. on p. 19).

[91] R. P. Duarte, M. Véstias, and H. Neto. «Enhancing stochastic computations via process variation». In: *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. 2015, pp. 1–7. DOI: 10.1109/FPL.2015.7293962 (cit. on p. 19).

[92] J. Han, H. Chen, J. Liang, P. Zhu, Z. Yang, and F. Lombardi. «A Stochastic Computational Approach for Accurate and Efficient Reliability Evaluation». In: *IEEE Transactions on Computers* 63.6 (2014), pp. 1336–1350. DOI: 10.1109/TC.2012.276 (cit. on p. 19).

[93] P. Knag, W. Lu, and Z. Zhang. «A Native Stochastic Computing Architecture Enabled by Memristors». In: *IEEE Transactions on Nanotechnology* 13.2 (2014), pp. 283–293. DOI: 10.1109/TNANO.2014.2300342 (cit. on p. 19).

[94] R. P. Duarte, H. Neto, and M. Véstias. «XtokaxtikoX: A stochastic computing-based autonomous cyber-physical system». In: *2016 IEEE International Conference on Rebooting Computing (ICRC)*. 2016, pp. 1–7. DOI: 10.1109/ICRC.2016.7738716 (cit. on p. 19).

[95] J. Li, A. Ren, Z. Li, C. Ding, B. Yuan, Q. Qiu, and Y. Wang. «Towards acceleration of deep convolutional neural networks using stochastic computing». In: *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2017, pp. 115–120 (cit. on pp. 19, 20).

[96] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. *You Only Look Once: Unified, Real-Time Object Detection*. 2016. arXiv: 1506.02640 [cs.CV] (cit. on p. 19).

[97] Y. Liu, H. Venkataraman, Z. Zhang, and K. K. Parhi. «Machine learning classifiers using stochastic logic». In: *2016 IEEE 34th International Conference on Computer Design (ICCD)*. 2016, pp. 408–411. DOI: 10.1109/ICCD.2016.7753315 (cit. on p. 20).

[98] B. Li, Y. Qin, B. Yuan, and D. J. Lilja. «Neural Network Classifiers Using Stochastic Computing with a Hardware-Oriented Approximate Activation Function». In: *2017 IEEE International Conference on Computer Design (ICCD)*. 2017, pp. 97–104. DOI: 10.1109/ICCD.2017.23 (cit. on p. 20).

[99] M. Stimberg, R. Brette, and D.F.M. Goodman. «Brian 2, an Intuitive and Efficient Neural Simulator.» In: *eLife* (2019) (cit. on p. 21).

[100] Synopsys. *FPGA COMPILER II / FPGA Express VHDL Reference Manual, Version 1999.05*. Synopsys, 1999 (cit. on p. 47).

[101] D. Floreano and C. Mattiussi. *Manuale sulle Reti Neurali*. Il Mulino, 1996 (cit. on p. 60).

[102] Mentor. *Modelsim*. https://www.mentor.com/. Version: 2007 (cit. on p. 67).

[103] Synopsys. *Synopsys Design Compiler*. https://www.synopsys.com/. Version: Nov 21, 2016 (cit. on p. 67).