Computational method and neuromorphic processor design applied to event-based sensors

Thomas Mesquida

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : NANO ELECTRONIQUE ET NANO TECHNOLOGIES

Arrêté ministériel : 25 mai 2016

Présentée par

Thomas MESQUIDA

Thèse dirigée par **Edith BEIGNE**, CEA

préparée au sein du **Laboratoire CEA/LETI**
dans **l'École Doctorale Electronique, Electrotechnique, Automatique, Traitement du Signal (EEATS)**

Méthode de calcul et implémentation d'un processeur neuromorphique appliqué à des capteurs évènementiels

Computational method and neuromorphic processor design applied to event-based sensors

Thèse soutenue publiquement le **20 décembre 2018**,
devant le jury composé de :

Monsieur FREDERIC PETROT
PROFESSEUR, GRENOBLE INP, Président
Monsieur RYAD BENOSMAN
PROFESSEUR, UNIVERSITE PIERRE ET MARIE CURIE, Rapporteur
Monsieur SEBASTIEN LE BEUX
MAITRE DE CONFERENCES HDR, ECOLE CENTRALE DE LYON, Rapporteur
Monsieur LAURENT PILATI
INGENIEUR, NXP - FRANCE, Examinateur

As I was starting my Ph.D. on Computational Method and Neuromorphic Processor Design Applied to Event-based Sensors at CEA Leti I discovered the ever-expanding world of Neuromorphic Engineering. I was coming from a relatively standard engineering course and this new way of computing data was fascinating and radically different from what I knew.

When I joined Alexandre Valentian's team back in 2015, study on sensors and signal processing using Neuromorphic Engineering was merely beginning. During those 3 years, our team grew and was able to demonstrate interesting results. This Ph. D. involved the study of Neural Networks from multiple aspects, led to fruitful collaborations and innovative solutions. I was able to meet several people, some of I can now call friends, and share useful knowledge.

I will not try to exhaustively enumerate people who had an influence over my Ph. D., as I might forget some of them, but they are still to be thanked for their help. Nevertheless, I will emphasize some of them:

- Edith B. and Alexandre V. for their unending support, reviews and lead.
- My family and girlfriend for their support at home.
- The Neuromorphic team for their everyday entertainment and collaboration.
- Ph. D. students for their joy and inputs.
- The LISAN laboratory and Benjamin C. for supervising my work and their technical mentoring.
- Members from others laboratories including Elisabeth D. for their inputs in the biology field.

Abstract

Computational method and neuromorphic processor design applied to event-based sensors

Studying how our nervous system and sensory mechanisms work lead to the creation of event-driven sensors. These sensors follow the same principles as our eyes or ears for example. This Ph.D. focuses on the search for bio-inspired low power methods enabling processing data from this new kind of sensor. Contrary to legacy sensors, our retina and cochlea only react to the perceived activity in the sensory environment. The artificial retina and cochlea implementations we call dynamic sensors provide streams of events comparable to neural spikes. The quantity of data transmitted is closely linked to the presented activity, which decreases the redundancy in the output data. Moreover, not being forced to follow a frame-rate, the created events provide increased timing resolution. This bio-inspired support to convey data lead to the development of algorithms enabling visual tracking or speaker recognition or localization at the auditory level, and neuromorphic computing environment implementation. The work we present rely on these new ideas to create new processing solutions. More precisely, the applications and hardware developed rely on temporal coding of the data in the spike stream provided by the sensors.

Méthode de calcul et implémentation d'un processeur neuromorphique appliqué à des capteurs évènementiels

L'étude du fonctionnement de notre système nerveux et des mécanismes sensoriels a mené à la création de capteurs événementiels. Ces capteurs ont un fonctionnement qui retranscrit les atouts de nos yeux et oreilles par exemple. Cette thèse se base sur la recherche de méthodes bio-inspirés et peu coûteuses en énergie permettant de traiter les données envoyées par ces nouveaux types de capteurs. Contrairement aux capteurs conventionnels, nos rétines et cochlées ne réagissent qu'à l'activité perçue dans l'environnement sensoriel. Les implémentations de type « rétine » ou « cochlée » artificielle, que nous appellerons capteurs dynamiques, fournissent des trains d'évènements comparables à des impulsions neuronales. La quantité d'information transmise est alors étroitement liée à l'activité présentée, ce qui a aussi pour effet de diminuer la redondance des informations de sortie. De plus, n'étant plus contraint à suivre une cadence d'échantillonnage, les événements créés fournissent une résolution temporelle supérieure. Ce mode bio-inspiré de retrait d'information de l'environnement a entraîné la création d'algorithmes permettant de suivre le déplacement d'entité au niveau visuel ou encore reconnaître la personne parlant ou sa localisation au niveau sonore, ainsi que des implémentations d'environnements de calcul neuromorphiques. Les travaux que nous présentons s'appuient sur ces nouvelles idées pour créer de nouvelles solutions de traitement. Plus précisément, les applications et le matériel développés s'appuient sur un codage temporel de l'information dans la suite d'événements fournis par le capteur.

Contents

List of figures

List of Tables

Introduction

Neuromorphic engineering, both at the sensory [1][2][3] and processing[4][5] levels, have gained interest in the research community. Initial studies [6] created a rapidly fading trend and later reference publications [7][8] renewed the interest in the domain. This Ph.D work focuses mainly on biomimetic event-driven time-based sensors we will reference as Dynamic Sensors, such as [1][2], and the development of bio-inspired processing solutions fitting their nature.
Dynamic Sensors such as Dynamic Vision Sensors (biological retina) and Dynamic Audio Sensors (biological cochlea) are sensible to events in the sensory scene. When legacy sensors sample the sensory data every fixed amount of time, dynamic sensors only provide events linked to activity in the sensory scene. By dropping the frame-based scheme, these sensors produce less redundant data to be later processed. Moreover, dropping this constraint also allows for previously sub-frame information to be now retrieved.
The first part of the Ph.D. focused on analyzing the characteristics of the obtained event streams and previously developed processing solutions [9][10] to cope with this sparse asynchronous data. The focus was rapidly made on the methods used to retrieve relevant data from such sensors and how to process this entity. The first explored subject is the implementation of lightweight processing algorithms fitting the sensors' nature to enable low power processing.

In Chapter 1, we focus on the developed Dynamic Sensors, the bio-inspired processing algorithms designed to interface them and their accelerators. We show the various implementations of bio-inspired sensors available and highlight the base idea and factors around which they revolve. Depending on the implementation, relevant information can be retrieved using the spike train rates or relative timings. Keeping in mind the embedded application target, classes of bio-inspired processing solutions are depicted and associated with the corresponding hardware implementations. Neural Networks achieve high efficiency in multiple fields, and their spiking implementations can fit the event stream format, making promising candidates.
During this phase we chose to explore promising works based on the implementation of library of operators based on Spiking Neural Networks. The implemented building blocks can

leverage machine learning efficiency and offer possibility for pre-processing.

In Chapter 2, we focus on two coding strategy used for implementing such library. First, we define the implementation based on rate coding, meaning the rate of the events of a channel define the data conveyed by this channel. The second implementation was optimized from [11] and is based on a temporal approach. The data is conveyed as relative timings in between events with the possibility for operations using few events.
These two approaches are compared with two main metrics, synaptic events usage and result latency, that help us estimate the hardware cost of such approach. Functionality is also compared in order to determine which of the proposed solutions is a better fit for processing and leveraging machine learning by network conversion. The time coded approach was found to better suit specifications of dynamic sensors and was used for the rest of the manuscript.

In Chapter 3, we explore the hardware architecture developed for simulating Spiking Neural Networks and propose an approach fitting dynamic sensors to be used with Precise Timing Networks. A Globally Asynchronous Locally Synchronous architecture is proposed with synchronous clusters of neurons served by an asynchronous network on chip using the Address Event Representation Protocol. The main choices to be made are the approach used for scheduling and processing the activity in each cluster. As we analyze the characteristics of the activity produced by our spiking neural network topologies, we can assert the usage and footprint of the described solutions.
This analysis lead to the development of a SystemC model of the obtained architecture. Such a model is essential to enable fast verification and estimations. This model was also used to size and test the Proof Of Concept.

In Chapter 4, we implement the ideas developed in Chapter 3 in a test chip taped out in FDSOI 28nm. The chip is composed of a fully synchronous cluster relying on the synchronous implementations of the scheduling and processing modules. It exhibits 26.4 pJ per synaptic operation while being able to support the topologies exposed in chapter 2.

In Chapter 5, we highlight the conversion and pre-processing capabilities of our topologies. This was done with two main benchmarks, namely MNIST and N-TIDIGITS18. MNIST, the handwritten digits classification database, is used as a reference benchmark for implementing and tweaking classification algorithms. This first database was converted to spiking inputs and used to perform the conversion of a convolutional neural network. First, the convolutional neural network is trained with a set of rules to constrain its weights and

dynamics. Then, an equivalent spiking topology is defined to receive the trained weights and perform the spiking inference.
N-TIDIGITS18, the spoken digits classification database, was obtained by recording the TIDIGITS database with a Dynamic Audio Sensor [12]. With this second database we explore the conversion of Recurrent Neural Networks using an approach similar to the MNIST one. This database is already composed of spiking samples, which requires to retrieve features to train the non spiking bases. This database lead to the development of the first recurrent networks based on temporal features.

The work presented in this document led to the publications [13][14][15]. The complete list including to be published paper can be found in the Appendix A.

Chapter 1

Asynchronous event-based sensing and processing

Sommaire

1.1 Introduction

The development of Dynamic Sensors (DS) [16][17][2][18][19][20][1][12][21] allows for new computing paradigms to be used. They inspire themselves from biological examples using asynchronous events to convey information. Legacy sensors output complete sensory data every defined amount of time. This leads to important redundancy, bandwidth requirements and generated workload [2]. On the other hand, DS are not constrained to frame-based sensing. Their output event streams are sparse and the resulting bandwidth requirements decreased. The output redundancy and workload are reduced due to their dynamic sensing nature. Moreover, their implementation provide information on sub-frame rate changes that could not be processed with legacy sensors. Targeting the same sensitivity with legacy sensors would lead to greatly increase its frame rate and other drawbacks: data redundancy, communication bandwidth and processing workload.

This chapter presents event-based approach to vision and audio sensing and processing inspired from biological examples. The Dynamic Vision Sensors (DVS) mimic the biological visual system [17][2][18][3][22] while the Dynamic Audio Sensors (DAS) [1][12][21] mimic the biological auditory system. Both of these approaches output sensory data in the form of asynchronous events or spikes that can encode for dynamic change and/or static information depending on the implementation.

As the output spike streams can follow different rules in term of coding and the ways they can be exploited, different processing solutions were developed. Data can be retrieved and processed from the output events using different coding and computing biology-inspired approaches that will be presented in this chapter. Each of the described approaches require different hardware as they use different ideas and hardware notions will also be developed. The main focus will be made on highlighting combinations that fit the idea developed by DS: sparse and low power processing.

1.2 Event Driven Sensors

For the reader to better understand the rest of this manuscript, we first depict the sensory inputs we aim at interfacing with. These DS output streams of events with peculiar characteristics that will be later processed with. The first explored category is the DVS inspiring themselves from biological visual system and the second category is the DAS inspired from the biological auditory system.

1.2.1 Dynamic Vision Sensors

The inspiration for DVS comes from the retina. When legacy sensors are implemented to output complete sensory information at every frame generated at constant timings, the activity generated by the retina is closely linked to visual changes. DVS implement similar dynamic sensitivity in order to output event streams linked to visual events. Different ideas can be retrieved from the retina, and the change sensitivity can be temporal, with events created as the luminance changes [2][23]. The focus can also be made on edges with gradient-based sensors [24]. Event-based ideas can also be adapted to optical-flow sensors [25].

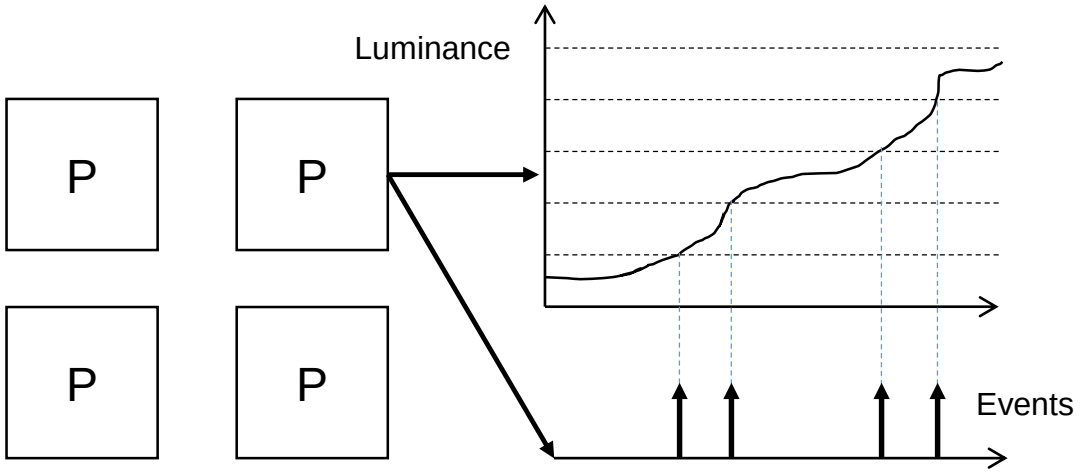


Figure 1.1: Luminance change events produced by a dynamic pixel in DVS.

The output data is encoded in asynchronous events using Address Event Representation (AER) [26] [27]. Each event can be represented by its origin address, the pixel it was emitted from, and emission time-stamp. The data conveyed by those events can only be retrieved by monitoring the timing, order of count of said events. Fig.1.1 shows working principle for the luminance changes events used in [2]. When using those differential events, methods can be applied for tracking of entities in the visual scene [28][29] while having low processing workload and sub-frame timing resolution.

Moreover, some DVS implementations [2][3] also use asynchronous events to encode for the "static" luminance measured by active pixels. Fig.1.2 shows the working principle of such pixels. Both the change and static luminance events can be complementary: entities and their borders can be tracked using the dynamic information and the determined region of interest can be used with static luminance events to retrieve information on the entity itself.

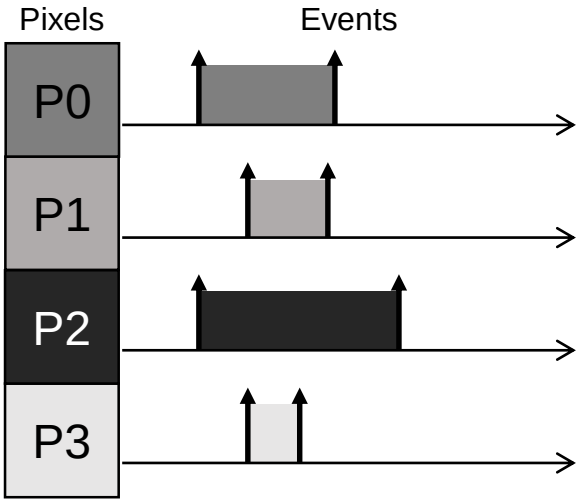


Figure 1.2: Static luminance measured by a DVS using active pixel and sent as a couple of events. The inter-event time interval defines the luminance value.

1.2.2 Dynamic Audio Sensors

The same principles were applied for DAS, inspired from the cochlea. Legacy sensors sample the auditory data at defined rate function of their use. The cochlea is composed of ranges of hair cells that are tuned to certain frequency ranges by their location and mechanical properties. Each set of hair cells corresponding is stimulated when receiving their specific sound frequency, leading to the emission of neural events via underlying chemical mechanisms.

The DAS implementations use this idea of multiple channels each sensitive to a frequency range and emitting asynchronous events when stimulated with enough energy. Their implementations [30][31][32][12][33][34][1] are based on cascaded band filters with configurable quality factors to implement the different frequency ranges for the different output channels and mostly perform in the sub-mW range.

The AER EAR [12] was utilized to convert TIDIGITS made in 1993. Samples of spoken digits and sequences of digits were replayed and recorded [35] using the AER EAR. Using only output neuron per channel for 1 side of the binaural setup, the obtained event rate is lower than 2k events per second decreasing greatly the data rate to be processed. Fig.1.3 shows the chronogram obtained for the sequence "5 8 9 9 2" and the obtained event sparsity result of the dynamic implementation.

Using binaural system composed of two DAS enables event driven sound location using the interaural time difference [12][36]. The main characteristics of the output spike streams have also been exploited for speaker identification [37] or spoken digits recognition [35].

The DS principle leads to an output data rate and redundancy decrease [2][34]. The

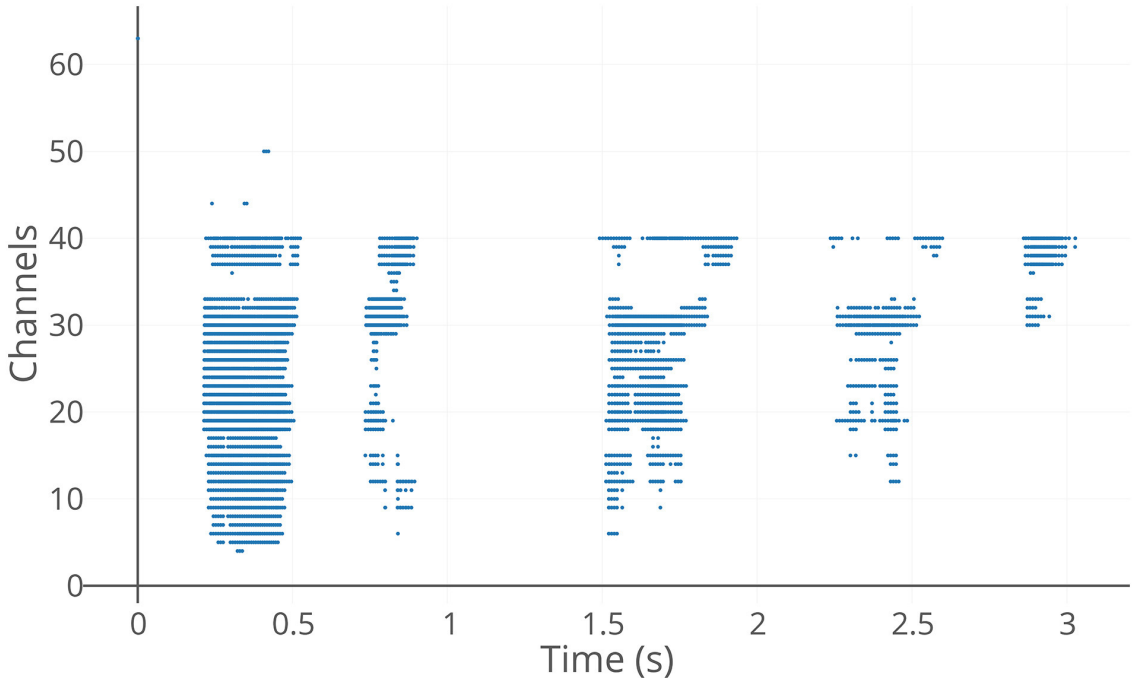


Figure 1.3: Chronogram of the spoken sequence "5 8 9 9 2" from TIDIGITS recorded with the AER EAR [12].

output stream of events is sent asynchronously using the AER protocol, meaning the output data has to be retrieved from the timing at which each origin address emitted spikes. The main problematic when processing such stream of events is choosing appropriate solutions to fit the nature of data.

1.3 Bio-inspired processing

This section presents the methods inspired from our nervous system that have been implemented in order to use the data retrieved from the DS. Conventional signal processing methods can be applied in a event driven fashion and this manuscript does not focus on this branch. This section focuses on the different processing methods inspired from our nervous system and their application to DS.

The base of our nervous system is the neuron, composed of a soma, input synapses and dentrites and an output axon as depicted in Fig.1.4. The link between neurons is made by the synapses having variable stimulating strength. A core potential can be associated to the soma of the neuron handling the emission of spikes. When this potential reaches a threshold, the soma emits an action potential through its axon that fires neurotransmitters and resets the neuron to its rest potential. The pool of synapses connected to the axon each connect to a postsynaptic neuron. Each synapse receives the action potential and transmits a signal to its postsynaptic neuron making it more or less likely to fire its action potential.

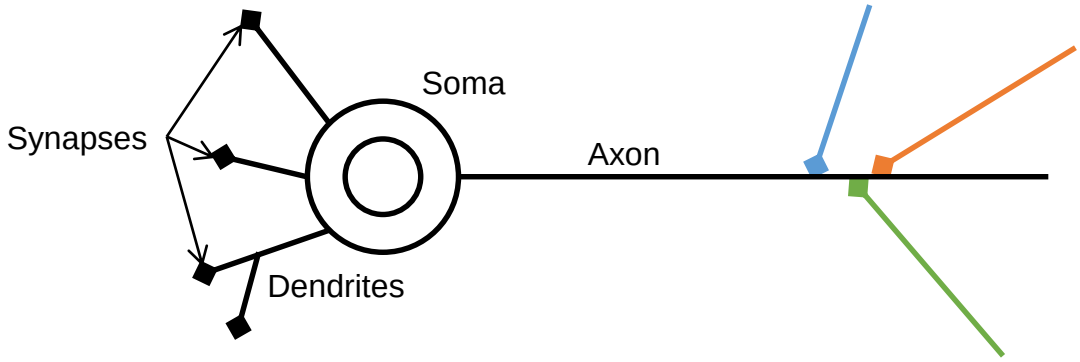


Figure 1.4: Simplified drawing of a biological neuron.

The core potential uses temporal and spatial summation, meaning it integrates signals from each synapse and a leak to its rest potential.

The strength of the stimulation transmitted via synapses when the presynaptic neurons emit its action potential is variable. The Hebb's rule [38] was the first hypothesis of the mechanisms of synaptic plasticity and stated that the synaptic efficacy increases when the presynaptic neuron repeatedly and persistently stimulates the postsynaptic neuron. Later the Spike Timing Dependent Plasticity was demonstrated [39][40][41] and gave explicit synaptic efficacy behavior as a function of the timing difference between the pre and postsynaptic action potentials.

The behavior presented here is highly simplified compared to the actual neuron dynamics leading to a plurality of possible behaviors [42]. These neurons are arranged into networks such as cortical columns that are still under heavy researches [43]. Even though part of the brain and their roles can be mapped, their actual underlying mechanisms are not fully understood. Nevertheless, abstract versions of Neural Networks (NN) can be derived from these mechanisms and have proven to be efficient in various domains.

1.3.1 Artificial Neural Network - fame-like processing

The first class of NN explored for processing the DS is the Artificial Neural Networks (ANN). The beginnings of ANN date back to 1958 with the Perceptron [6]. It is the most abstract model of neuron developed by dropping the timing feature of a spike train. As depicted in Fig.1.5, each channel holds a static value representing the normalized average firing rate of the presynaptic neuron. The synapses are only represented by their weight, and the soma is performing the sum of the weighted firing rates. An activation function is then performed on the result to compute the output value of the neuron. The activation function can be Linear, Heaviside, Sign, Rectified Linear Unit (ReLU), Sigmoid, Tanh among other possibilities. These neurons are arranged in networks having different topologies for different applications. The Multi Layer Perceptron (MLP) depicted in Fig.1.6 is one of the simplest example,

composed here of 1 input layer, 1 hidden layer and 1 output layer.

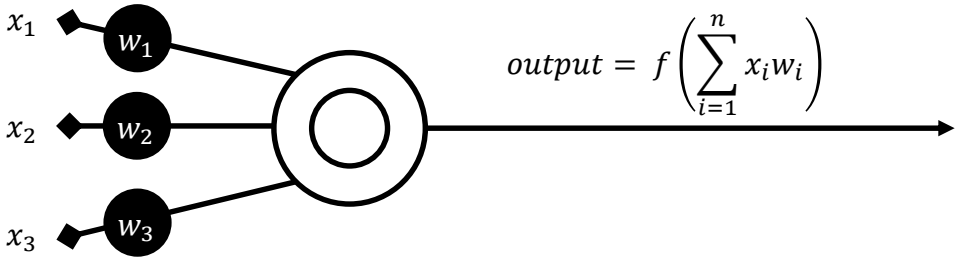


Figure 1.5: Perceptron: neuron model using normalized firing rates as data.

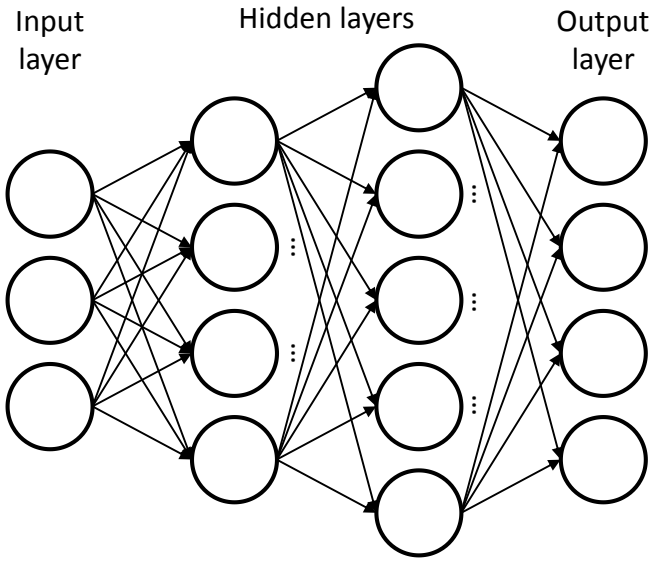


Figure 1.6: Multi Layer Perceptron: ensemble of perceptrons arranged in interconnected layers. Each layer feeds its output as input to the next layer. Layers that are not the output of the input one are characterized as hidden layers. This MLP has 4 layers of respectively 3, 4, 5 and 4 neurons.

This model of neuron drops the spiking aspect of biological NN and thus cannot apply a learning rule such as Spike-Timing-Dependent Plasticity (STDP). The learning rule used and studied for such network is the gradient backpropagation. While the STDP can be applied at a synapse level, the gradient backpropagation requires to perform a forward pass through the whole network before performing a learning step. Its principle relies on performing a backward pass designed to rectify the synaptic weights to force the networks closer to the desired output. The desired output is compared to the output of the forward pass and the resulting error is used to modify weights of the last layer. This process then sweep the layers backward until it reaches the input layer. By repeating this process, inputs can be associated to required outputs with accuracy dependent on the application and network.

ANNs have already proven to be efficient in multiple domains, achieving highest software accuracy for the MNIST database (handwritten digit recognition) [44][45][46][47] and even reaching higher than human accuracy. Their success lead to the development of

a plurality of turnkey solutions for Computer Aided Design of ANNs [48][49][50][51]. Their efficiency in audio [35][36], image [52][44] and video processing [53][54] on various tasks makes them unavoidable candidates. Using derived learning strategies such as Reinforcement Learning, they also achieve great results in more complex and abstract tasks such as Go and Chess [55] or video games [56][57].

Using ANN with DS requires some gymnastic when handling asynchronous events to feed a network needing frame-like inputs. The main task is to define how to convert these events into frames exploitable by the ANN while still maintaining the relevant part of the data via preprocessing. This task is problematic when using DVS based on luminance change for instance as the frame creation strategy (counting events per fixed amount of time or using a fixed number of events per frame for instance) can be countered by the environment (variations of global event rates and event rates linked to studied entities).

When used with DVS, ANNs have already been exploited for tracking in predator-prey scenario [58], MNIST classification with sensor fusion [59]. Using purely spiking audio data, ANNs have already been exploited for spoken digits recognition [35].

Using ANNs with DS brings efficient results when the data is correctly retrieved from the event stream. However, the computation heavy ANN model does not fit the ideas developed with DS: sparse event driven sensory data and reduced workload. The next part of this section focuses on possibilities for networks closer to biology exploiting those properties.

1.3.2 Spiking Neural Networks

This second class of NN is closer to the biological inspiration as it utilizes streams of spikes as data rather than static values. The Spiking Neural Networks (SNN) rely on sets of equations handling the behavior of the neuron of the network and interaction. Fig.1.7 describes a simplified model of spiking neuron that can be used. More complex models were developed in order to mimic the behavior of biological neurons [42] and rely on sets of differential equations for representing the different biological mechanisms. However, it is still unclear how to use networks composed of those biologically plausible neurons for a target application.

The SNN explored in this section are composed of much simpler dynamics while still being powerful enough to answer application needs, mainly the (Leaky) Integrate and Fire neurons. The events are transmitted via the AER protocol, dropping the action potential shape from the biological neuron. The synapses include simple mechanisms such as programmable delay, weight and simple integration mechanisms. As for the biological neuron, the soma integrates the stimuli coming from its different synapses. The "Leaky" version also includes a leak pushing the soma potential to its resting potential. As for the biological neuron, when the soma potential reaches its threshold, the neuron emits a spike and resets to

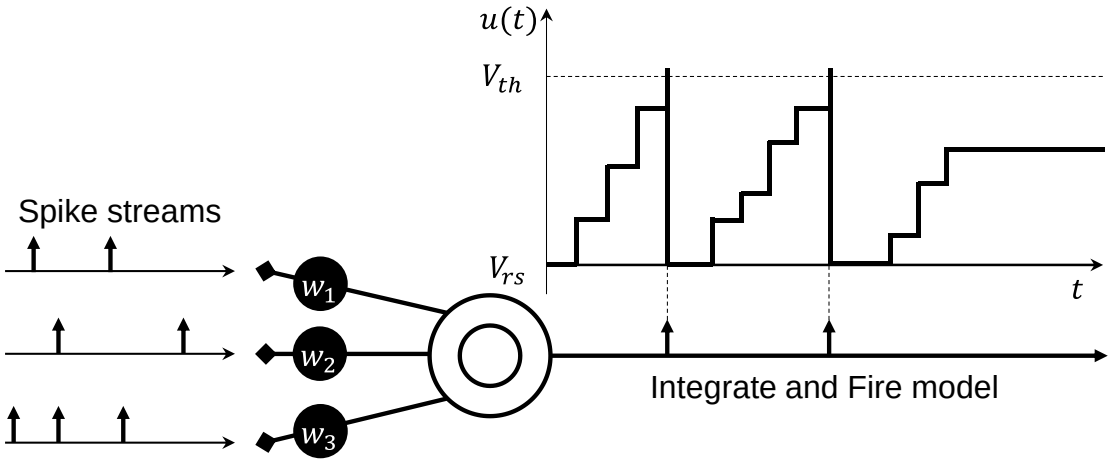


Figure 1.7: Example of spiking neuron model: Integrate and Fire. The synapses used here induce jumps in the membrane potential $u(t)$. Once $u(t)$ reaches V_{th}, the neuron emits a spike and resets to V_{rs}.

its resting potential.

Using those basic mechanisms, multiple implementations are possible interfacing DS. The main idea is to fit the shape of the input spike stream when processing it, avoiding converting back to frame and potentially retrieving more information [60][61][62]. Moreover, the underlying mechanisms are less computation heavy than their ANN counterparts based on Tensor calculation. The first possibility explored here is the use of STDP or other learning methods with SNNs to design networks answering application needs. The following sections will explore SNN design possibilities using results achieved with ANN and conversion methods relying on different coding strategies.

Learning based networks

In order to reach the results obtained with ANN, it is necessary to develop methods that give SNN equivalent error rates as their continuous-valued counterparts. The first explored method relies on bio-inspire learning rules, mainly STDP and Spike Driven Synaptic Plasticity (SDSP), while the second branch of studies relies on adapting supervised learning rules to SNNs.

The most common implementation of the STDP is using window function at each synapse. This window function will increase the synapse weight if the presynaptic neuron fires briefly before the postsynaptic neuron and decrease it otherwise. Several variations of this principle exist, some being more suitable to hardware usage.

The SNN using STDP and forward connections (no recurrence) have proven efficient in pattern recognition using DVS. Tasks such as traffic [63], digit recognition [64] or

movement recognition [62] have been explored recently, showing the possibilities for the use of such SNN with DVS. To the best of our knowledge, no such network was developed for interfacing with DAS as the topologies of networks required for the corresponding applications do not support STDP.

STDP is a local unsupervised learning rule, meaning the learning is done at the synapse level and no global supervision can be made as for the backpropagation. Using lateral inhibition implementing a winner-takes-all, limited forward-only networks can force different neurons of the same layer to specialize into different patterns [63][65][66][67]. As the STDP is local and unsupervised, it makes a great candidate for on-chip and on-line learning as we will see when exploring hardware implementations. However, the STDP still has issues scaling to greater networks [64] and achieving accuracy achieved by ANNs for the same tasks.

Spike Driven Synaptic Plasticity (SDSP) was proposed as another learning rule for SNNs. This rule does not use the timing factor from the STDP, and only monitors the postsynaptic membrane potential and average activity to determine correlation and weight modifications [68]. However, at this point no clear results can be retrieved from such rule [68][69] even in supervised cases.

Supervised learning rules give ANNs part of their efficiency and were adapted to SNNs. Multiple methods were created around this idea[70][71][72][73][74][75][76] including SPAN[77], Chronotron[78] or ReSuMe[79].

Successful approaches include direct training of SNNs using spike-based supervised gradient descent backpropagation [74], the SNN classifier layers using stochastic gradient descent [76]. [75][80] allow STDP usage and backpropagation. [77] and [70] approximate the behavior of spiking neurons to enable backpropagation usage. Finally, [71] produced an algorithm to calculate the network weights.

While the results are promising, these novel methods have yet to mature to the state where training increasingly important spiking architectures becomes possible, and the same state-of-the-art error rate as the equivalent ANN is achieved. While training directly SNN is not yet mature, training ANNs and converting them to SNNs using set of operators or mathematical equivalence might prove key to success.

ANN conversion

Taking advantage of the efficiency developed with ANNs, the conversion approach relies on taking the parameters of a pre-trained ANN and to map them to an SNN using methods guaranteeing the mathematical equivalency of both networks. Early work [81] presented

conversion of Convolutional Neural Networks (CNN) [8] to biologically inspired SNN units to interface DVS.

One of the factors differentiating conversion methods is the data encoding in spike trains. Data can be stored in a spike train in multiple ways [61][60]. The most commonly used strategy is rate coded, meaning the spiking rate of a channel is holding its data. Other strategies use time based approaches, where the information is stored in relative or absolute timings of spikes. Table 1.1 enumerates coding strategies using 10 channels on 10 ms with a time precision of 1 ms. The count strategy refers to counting the spike in each channel, leading to similar to rate coded results for 10 spikes at most. The Binary approach would associate 1b of data to the presence of a spike in a channel. The time based approaches can retrieve the data from the time window at which each spike was emitted. The rank approach monitors the order in which each spike arrived only, not the time window they belong. Finally the rate coded approach is a generalization of the count method for n bits of information.

Coding	Bit/10ms	Bit/Spk
Count	3.46	0.346
Binary	10	1
Time	>33	>3
Rank	>21	>2.1
Rate*	n	$1/2^n$

Table 1.1: Comparing spike coding: using 10 channels allowed to spike at most 1 time in a 10ms time window with 1ms discernible intervals. *The Rate coded approach uses at most 2^n spikes.

In order to convert ANNs, one has to first choose a coding strategy and implement the needed operators to enable the network conversion. The two most used and developed approaches are the rate coded and time coded families we will explore.

1.3.3 Rate coded networks

The first approach explored, and the most widely used, is the rate coded network. [82] presented the link between the transfer function of a spiking neuron (relation between the input current and the output firing rate) and the activation of a ReLU widely used in ANN training. While the implementation is quite limited as it did not include bias and only average pooling layers, the results still showed possibilities for rate based conversion. [83] uses the same principle adding weight normalization to achieve loss-less conversion. The first implementation had accuracy loss due to neurons firing at too high or low rates, leading to errors in output results.

Using LIF neurons and noise injection during training, [84] produces SNNs more robust to approxiation errors that were solved in [83]. [85] uses ANN with binary weights and restricted connectivity to be converted to SNN and used with the TrueNorth chip [4]. As rate coded networks can be spike consuming, [86] used firing threshold balancing while converting in order to reduce the number of spikes required to encode the information.

The previously described approaches achieve high accuracy on MNIST but scaling up to higher difficulty databases such as CIFAR-10 is not possible. As mentioned, the number of available ANN mechanisms that can be converted to SNN is quite limited. For example, accuracy can be gained by using max-pooling layers, changing activation functions or also using batch-normalization. Thus, previously described methods do not obtain SoA ANN results as they cannot mimic their behavior. [87] solves this issue by SNN building blocks performing equivalent operations. They achieved the best results in term of accuracy but are still not able to reach SoA ANNs.

Another approach is based purely on developing SNN operators able to perform mathematical function and build any pre-processing or ANN functions around it. For example, the multiplication of input firing rates can be achieved by using a coincidence detector [88]. This approach was further developed in [9] [89] with associated hardware support [10] and influence [13]. However, this framework was not intended to perform ANN conversion and do not include building blocks needed for it. As mentioned, [87] achieves the best results as they produced the important building blocks needed in an ANN. Equivalent method could be developed with building blocks from [9][89] and should be investigated. Another issue of the resulting rate coded SNN is often the number of spike used which was mitigated in [86]. The time based approaches solve that issue naturally.

1.3.4 Time coded networks

The time coded networks revolve around using spike timing in order to convey information. This approach was used in [90] with Time To First Spike (TTFS). TTFS encodes the information in the timing or arrival of the first spike of each channel, meaning only 1 spike is used to convey the luminance information for instance. This method creates SNN using 7-10x fewer operations than their ANN counterparts at the cost of less than 1% accuracy loss.

Another possibility was created by the work of X. Lagorce [28][11] creating a framework of linear and non linear time coded operators using relative spike timing as data. Similar to [9][89] for the rate coded side, this method allow pre-processing as the implemented building blocks are designed to perform linear and non linear operations. Moreover, the implementation of logic and memorization operators creates new possibilities for ANN conversion. However, this framework was not implemented to perform ANN conversion and no

efficiency can be estimated.

We depicted the main existing bio-inspired possibilities for interfacing DS. We are now going to explore the designed hardware support corresponding to previous solutions for a better understanding of the cost of each solution.

1.4 Neuromorphic architectures

The networks developed in previous sections require different type of hardware to support them. This section will present the main chips and ideas developed to host the presented network shown.

1.4.1 Artificial Neural Network

ANN typically require Floating-Point Multiply ACcumulate (MAC) to compute stimulation sum to be used with the activation function. Those operations can be assembled layer wide to form Matrix or Tensor operations. The most optimized common hardware when it comes to performing this type of operation is the Graphics Processing Unit (GPU). It performs those operations efficiently in parallel, but their power consumption can prohibit their use in embedded applications. A considerable effort is currently dedicated to developing hardware accelerators and algorithmic improvements in order to bring ANNs within reach of embedded devices power budget.

In the race for minimal TOPS/W, the Tensor Processing Unit [91] is one of the most advanced and used product, composed mainly of 8-bit MACs. Other implementations aim at lower budget networks, imposing limits in terms of accuracy to lower the output TOPS/W result. [92] implemented low power ANN accelerator for binary ANN, making essentially every computation a boolean logic operation.

GPUs and their derivative are still heavy optimization subjects in order to fit the low power envelop required for embedded applications. Moreover, the hosted ANN would require feature extraction from the input spike stream leading to power consumption overhead.

1.4.2 Spiking Neural Network

On the SNN side, the most computed operation for the simplest IF model is the addition, adding the input stimulation to the membrane potential. The hardware implementation

differs greatly from the ANN ideas. Machine learning solution designed to be performed with SNN can be resource demanding when executed with frameworks such as [93][49] on conventional hardware. The need for SNN solutions in embedded application lead to the development of SNN accelerators, some of the flagships being explored in this section.

When comparing SNN accelerator implementation, multiples factors have to be taken into account. The main recurring figure in SoA chips is the energy per Synaptic Event, corresponding to the energy needed to integrate an incoming event in the soma. Moreover, the chips capacities for supporting networks have to be taken into account, i.e. its target application, supported mechanisms, number of neurons and synapses and target approach. The main ideas used when implementing SNN accelerators can be retrieved from flagships implementing large-scale hardware SNN supports. Two mixed signal implementations, BrainScales[94] and Neurogrid[5], and three fully digital implementation, TrueNorth[4], SpiNNaker[95] and Loihi[95] are shown in Tab.1.2.

Work	TrueNorth[4]	Loihi[96]	BrainScaleS[94]	Neurogrid[5]	SpiNNaker[95]
Techno	28nm	14nm FinFET	180nm	180nm	130nm
Implem	Digital	Digital	Mixed	Mixed	Digital
Time	Discretized	Discretized	Discretized	Real time	Discretized
Learning	No	Multiple	STDP	No	Multiple
Neurons	10^6	$< 131k$	$< 180k$	$65k$	$16k$
Syn./neur.	256	$16-16k$	$256-16k$	16	$1k$
En/syn. evt.	27pJ	>105pJ	174pJ	180pJ	27nJ

Table 1.2: Flagship neuromorphic chip comparison.

Among the digital implementations, SpiNNaker [95] is using a set of ARM cores and memory linked by a network. While being highly re-configurable and flexible SNN wise, its energy efficiency is the worst among the presented chips. BrainScaleS [94] and Neurogrid [5] were both implemented in 180 nm technology with mixed signal implementations. Both revolve around analog implementation of bio-realistic neural models and digital communication using the AER protocol. Their aim is to better model and understand neural and network mechanisms.

The two last implementations, also the most efficient energy wise, use the most advanced technologies among the studied chips. Truenorth [4] is the most efficient fully digital implementation. It uses time multiplexing for the neural update during the step-by-step network simulation. Each simulation time step lasts 1ms enabling the 256 neurons of each core to be updated and communicates their output spikes. Finally, the Loihi processor [96] produced by Intel is designed to be a flexible support for large scale SNN learning and inference. It integrates learning rules and possibilities for multiple neural models. The implemented mechanisms allow multiple approaches and coding strategies to be used. Its creation is also

aimed at pushing research on SNNs and their possibilities.

1.4.3 Discussion and conclusion

The DS inspire themselves from our nervous system providing event driven data of the sensory scene. Both the visual and auditory branches allow for dynamic information to be retrieved, lowering the resulting data rate and processing workload to be used if handled correctly.

Bio-inspired solutions were also developed to process mentioned visual and auditory data among other possibilities. Their most abstract model, the ANN, has the most advanced results in every field due to their efficient learning rules. However, its working principle does not fit advantages retrieved from the dynamic sensors. ANN are computation heavy and need frame-based data thus requiring conversions from the input spike stream. GPUs and theirs ANN optimization while very efficient do not match the embedded application requirements.

SNN on the other hand have the possibility to reduce considerably the energy consumption of Neural Network inference while interfacing Dynamic Sensors. Their nature makes them fit perfectly the asynchronous data sent from the sensors and their accelerators exhibit promising energy efficiency for specialized hardware. However, designing SNNs is not as straight forward. While STDP based approaches enable on-chip and online learning, the resulting efficiency is still far from SoA ANNs. SNNs following supervised learning rules inspired from ANN ideas are making great progress.

The most promising approach might reside in converting SoA ANN using SNN operators. Building blocks for rate coded operators and time-coded operators could enable high coverage of the needed building blocks for ANN conversion. Moreover, these building blocks also allow for data pre-processing while remaining in the spiking domain. The rest of this manuscript will focus on the investigation of such methods at the theoretical, hardware and application levels.

1.5 Références

[1] Yang et al. A 0.5v 55µw 64x2-channel binaural silicon cochlea for event-driven stereo-audio sensing. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 388–389, Jan 2016. 1, 6, 8

[2] C. Posch et al. A qvga 143 db dynamic range frame-free pwm image sensor with lossless pixel-level video compression and time-domain cds. *JSSC*, 2011. 1, 6, 7, 8

[3] C. Brandli, R. Berner, M. Yang, S. Liu, and T. Delbruck. A 240 × 180 130 db 3 µs latency global shutter spatiotemporal vision sensor. *IEEE Journal of Solid-State Circuits*, 49(10):2333–2341, Oct 2014. 1, 6, 7

[4] F. Akopyan et al. Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(10):1537–1557, Oct 2015. 1, 16, 18

[5] B. V. Benjamin, P. Gao, E. McQuinn, S. Choudhary, A. R. Chandrasekaran, J. Bussat, R. Alvarez-Icaza, J. V. Arthur, P. A. Merolla, and K. Boahen. Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. *Proceedings of the IEEE*, 102(5):699–716, May 2014. 1, 18

[6] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958. 1, 10

[7] C. Mead. *Analog VLSI and Neural Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989. 1

[8] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998. 1, 15

[9] Belhadj Bilel et al. Continuous real-world inputs can open up alternative accelerator designs. *SIGARCH Comput. Archit. News*, 41(3):1–12, June 2013. 1, 16

[10] A. Joubert et al. A robust and compact 65 nm lif analog neuron for computational purposes. In *2011 IEEE 9th International New Circuits and systems conference*, pages 9–12, June 2011. 1, 16

[11] X. Lagorce and R. Benosman. STICK: spike time interval computational kernel, A framework for general purpose computation using neurons, precise timing, delays, and synchrony. *CoRR*, abs/1507.06222, 2015. 2, 16

[12] V. Chan, S. C. Liu, and A. van Schaik. Aer ear: A matched silicon cochlea pair with address event representation interface. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 54(1):48–59, Jan 2007. vii, 3, 6, 8, 9

[13] T. Mesquida et al. Impact of the aer-induced timing distortion on spiking neural networks implementing dsp. In *2016 PRIME*, pages 1–4, June 2016. 3, 16

[14] T. Mesquida, A. Valentian, D. Bol, and E. Beigne. Architecture exploration of a fixed point computation unit using precise timing spiking neurons. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–8, Sept 2017. 3

[15] T. Mesquida, A. Valentian, D. Bol, and E. Beigne. Architecture exploration of a fixed point computation unit using precise timing spiking neurons. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–8, Sept 2017. 3

[16] Shih-Chii Liu and Tobi Delbruck. Neuromorphic sensory systems. *Current opinion in neurobiology*, 20 3:288–95, 2010. 6

[17] P. Lichtsteiner, C. Posch, and T. Delbruck. A 128×128 120 db 15µs latency asynchronous temporal contrast vision sensor. *IEEE Journal of Solid-State Circuits*, 43(2):566–576, Feb 2008. 6

[18] Massimo Barbaro et al. A 100 x 100 pixel silicon retina for gradient extraction with steering filter capabilities and temporal output coding. *IEEE Journal of Solid-State Circuits*, 37:160–172, 2002. 6

[19] B. Son et al. 4.1 a 640x480 dynamic vision sensor with a 9 µm pixel and 300meps address-event representation. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 66–67, Feb 2017. 6

[20] H. Guo et al. Dynamic resolution event-based temporal contrast vision sensor. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1422–1425, May 2016. 6

[21] S. C. Liu, A. van Schaik, B. A. Minch, and T. Delbruck. Asynchronous binaural spatial audition sensor with 2, $times$,64, $times$,4 channel output. *IEEE Transactions on Biomedical Circuits and Systems*, 8(4):453–464, Aug 2014. 6

[22] D. P. Moeys, F. Corradi, C. Li, S. A. Bamford, L. Longinotti, F. F. Voigt, S. Berry, G. Taverni, F. Helmchen, and T. Delbruck. A sensitive dynamic and active pixel vision sensor for color or neural imaging applications. *IEEE Transactions on Biomedical Circuits and Systems*, 12(1):123–136, Feb 2018. 6

[23] J. A. Lenero-Bardallo, T. Serrano-Gotarredona, and B. Linares-Barranco. A 3.6µs latency asynchronous frame-free event-driven dynamic-vision-sensor. *IEEE Journal of Solid-State Circuits*, 46(6):1443–1455, June 2011. 7

[24] Kwabena A Boahen and Andreas G. Andreou. A contrast sensitive silicon retina with reciprocal synapses. In J. E. Moody, S. J. Hanson, and R. P. Lippmann, editors, *Advances in Neural Information Processing Systems 4*, pages 764–772. Morgan-Kaufmann, 1992. 7

[25] Alan A. Stocker. Analog integrated 2-d optical flow sensor. *Analog Integrated Circuits and Signal Processing*, 46(2):121–138, Feb 2006. 7

[26] K.A. Boahen. Point-to-point connectivity between neuromorphic chips using address events. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, 47(5):416–434, May 2000. 7

[27] Zamarreño-Ramos et al. Multicasting mesh aer: A scalable assembly approach for reconfigurable neuromorphic structured aer systems. application to convnets. *IEEE Trans. Biomed. Circuits and Systems*, 7(1):82–102, 2013. 7

[28] Xavier Lagorce. *Computational methods for event-based signals and applications*. Theses, Université Pierre et Marie Curie - Paris VI, September 2015. 7, 16

[29] G. Taverni, D. P. Moeys, F. F. Voigt, C. Li, C. Cavaco, V. Motsnyi, S. Berry, P. Sipilä, D. S. S. Bello, F. Helmchen, and T. Delbruck. In-vivo imaging of neural activity with dynamic vision sensors. In *2017 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, pages 1–4, Oct 2017. 7

[30] H. Abdalla and T. K. Horiuchi. An ultrasonic filterbank with spiking neurons. In *2005 IEEE International Symposium on Circuits and Systems*, pages 4201–4204 Vol. 5, May 2005. 8

[31] E. Fragniere. A 100-channel analog cmos auditory filter bank for speech recognition. In *ISSCC. 2005 IEEE International Digest of Technical Papers. Solid-State Circuits Conference, 2005.*, pages 140–589 Vol. 1, Feb 2005. 8

[32] R. Sarpeshkar, M. W. Baker, C. D. Salthouse, J. . Sit, L. Turicchia, and S. M. Zhak. An analog bionic ear processor with zero-crossing detection. In *ISSCC. 2005 IEEE International Digest of Technical Papers. Solid-State Circuits Conference, 2005.*, pages 78–79 Vol. 1, Feb 2005. 8

[33] B. Wen and K. Boahen. A silicon cochlea with active coupling. *IEEE Transactions on Biomedical Circuits and Systems*, 3(6):444–455, Dec 2009. 8

[34] S. Liu, A. van Schaik, B. A. Mincti, and T. Delbruck. Event-based 64-channel binaural silicon cochlea with q enhancement mechanisms. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pages 2027–2030, May 2010. 8

[35] J. Anumula, D. Neil, T. Delbruck, and S.-C. Liu. Feature representations for neuromorphic audio spike streams. *Frontiers in Neuroscience*, 12:23, 2018. 8, 12

[36] J. Anumula, E. Ceolini, Z. He, A. Huber, and S. Liu. An event-driven probabilistic model of sound source localization using cochlea spikes. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, May 2018. 8, 12

[37] S. Liu, N. Mesgarani, J. Harris, and H. Hermansky. The use of spike-based representations for hardware audition systems. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pages 505–508, May 2010. 8

[38] Donald O. Hebb. *The organization of behavior: A neuropsychological theory*. Wiley, New York, June 1949. 10

[39] Henry Markram, Joachim Lübke, Michael Frotscher, and Bert Sakmann. Regulation of synaptic efficacy by coincidence of postsynaptic aps and epsps. *Science*, 275(5297):213–215, 1997. 10

[40] Guo-qiang Bi and Mu-ming Poo. Synaptic modifications in cultured hippocampal neurons: Dependence on spike timing, synaptic strength, and postsynaptic cell type. *Journal of Neuroscience*, 18(24):10464–10472, 1998. 10

[41] C. Daniel Meliza and Yang Dan. Receptive-field modification in rat visual cortex induced by paired visual stimulation and single-cell spiking. *Neuron*, 49(2):183 – 189, 2006. 10

[42] E. M. Izhikevich. Which model to use for cortical spiking neurons? *IEEE Transactions on Neural Networks*, 15(5):1063–1070, Sept 2004. 10, 12

[43] Simona Lodato and Paola Arlotta. Generating neuronal diversity in the mammalian cerebral cortex. *Annual Review of Cell and Developmental Biology*, 31(1):699–720, 2015. PMID: 26359774. 10

[44] R. Benenson. Best performances on multiple benchmarks, http://rodrigob.github.io/are_we_there_yet/build/#classification-dataset-type, 2015. 11, 12

[45] Li Wan et al. Regularization of neural networks using dropconnect. In *Proceedings of the 30thInternational Conference on Machine Learning*, page 1058–1066, 2013. 11

[46] Jost Tobias Springenberg et al. Striving for simplicity: The all convolutional net. *CoRR*, abs/1412.6806, 2014. 11

[47] Jurgen Schmidhuber. Multi-column deep neural networks for image classification. In *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, CVPR '12, pages 3642–3649, Washington, DC, USA, 2012. IEEE Computer Society. 11

[48] Jia et al. Caffe: Convolutional Architecture for Fast Feature Embedding. *ArXiv e-prints*, June 2014. 12

[49] CEA-LIST. N2d2, https://github.com/cea-list/n2d2, 2017. 12, 18

[50] Collobert et al. Torch: A modular machine learning software library, 2002. 12

[51] Martín and others. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. 12

[52] Benjamin Graham. Fractional max-pooling. *CoRR*, abs/1412.6071, 2014. 12

[53] C. Wojek, S. Walk, and B. Schiele. Multi-cue onboard pedestrian detection. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 794–801, June 2009. 12

[54] P. Dollar, C. Wojek, B. Schiele, and P. Perona. Pedestrian detection: An evaluation of the state of the art. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(4):743–761, April 2012. 12

[55] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017. 12

[56] Kevin Frans, Jonathan Ho, Xi Chen, Pieter Abbeel, and John Schulman. Meta learning shared hierarchies. *CoRR*, abs/1710.09767, 2017. 12

[57] Trapit Bansal, Jakub Pachocki, Szymon Sidor, Ilya Sutskever, and Igor Mordatch. Emergent complexity via multi-agent competition. *CoRR*, abs/1710.03748, 2017. 12

[58] Diederik Paul Moeys, Daniel Neil, Federico Corradi, Emmett Kerr, Philip J. Vance, Gautham P. Das, Sonya A. Coleman, T. Martin McGinnity, Dermot Kerr, and Tobi Delbrück. Pred18: Dataset and further experiments with davis event camera in predator-prey robot chasing. *CoRR*, abs/1807.03128, 2018. 12

[59] D. Neil and S. C. Liu. Effective sensor fusion with event-based sensors and deep network architectures. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2282–2285, May 2016. 12

[60] Simon J. Thorpe et al. Speed of processing in the human visual system. *Nature*, 381:520–522, 1996. 13, 15

[61] S. J. Thorpe et al. Spike-based strategies for rapid processing. *NEURAL NETWORKS*, 14:715–725, 2001. 13, 15

[62] Simon J. Thorpe. Spike-based image processing: Can we reproduce biological vision in hardware? In Andrea Fusiello, Vittorio Murino, and Rita Cucchiara, editors, *Computer Vision – ECCV 2012. Workshops and Demonstrations*, pages 516–521, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. 13, 14

[63] Olivier Bichler, Damien Querlioz, Simon J. Thorpe, Jean-Philippe Bourgoin, and Christian Gamrat. Extraction of temporally correlated features from dynamic vision sensors with spike-timing-dependent plasticity. *Neural Networks*, 32:339 – 348, 2012. Selected Papers from IJCNN 2011. 13, 14

[64] Johannes C. Thiele, Olivier Bichler, and Antoine Dupret. Event-based, timescale invariant unsupervised online deep learning with stdp. *Frontiers in Computational Neuroscience*, 12:46, 2018. 13, 14

[65] Saeed Reza Kheradpisheh, Mohammad Ganjtabesh, Simon J. Thorpe, and Timothée Masquelier. Stdp-based spiking deep convolutional neural networks for object recognition. *Neural Networks*, 99:56 – 67, 2018. 14

[66] Priyadarshini Panda, Gopalakrishnan Srinivasan, and Kaushik Roy. Convolutional spike timing dependent plasticity based feature learning in spiking neural networks. *CoRR*, abs/1703.03854, 2017. 14

[67] Amirhossein Tavanaei and Anthony S. Maida. Bio-inspired spiking convolutional neural network using layer-wise sparse coding and STDP learning. *CoRR*, abs/1611.03000, 2016. 14

[68] Stefano Fusi, Mario Annunziato, Davide Badoni, Andrea Salamon, and Daniel J. Amit. Spike-driven synaptic plasticity: theory, simulation, vlsi implementation. *NEURAL COMPUTATION*, 12:2227–2258, 1999. 14

[69] Charlotte Frenkel, Jean-Didier Legat, and David Bol. A 0.086-mm^2 9.8-pj/sop 64k-synapse 256-neuron online-learning digital spiking neuromorphic processor in 28nm CMOS. *CoRR*, abs/1804.07858, 2018. 14

[70] Qiang Yu, Huajin Tang, Kay Chen Tan, and Haizhou Li. Precise-spike-driven synaptic plasticity: Learning hetero-association of spatiotemporal spike patterns. *PLOS ONE*, 8(11):1–16, 11 2013. 14

[71] Jonathan Tapson, Gregory Cohen, Saeed Afshar, Klaus M. Stiefel, Y. Buskila, Runchun Wang, Tara Julia Hamilton, and André van Schaik. Synthesis of neural networks for spatio-temporal spike pattern recognition and processing. In *Front. Neurosci.*, 2013. 14

[72] Sander M. Bohte, Joost N. Kok, and Han La Poutré. Error-backpropagation in temporally encoded networks of spiking neurons. *Neurocomputing*, 48(1):17 – 37, 2002. 14

[73] H. Mostafa. Supervised learning based on temporal coding in spiking neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 29(7):3227–3235, July 2018. 14

[74] Junhaeng Lee, Tobi Delbrück, and Michael Pfeiffer. Training deep spiking neural networks using backpropagation. *CoRR*, abs/1608.08782, 2016. 14

[75] Chankyu Lee, Priyadarshini Panda, Gopalakrishnan Srinivasan, and Kaushik Roy. Training deep spiking convolutional neural networks with stdp-based unsupervised pre-training followed by supervised fine-tuning. *Frontiers in Neuroscience*, 12:435, 2018. 14

[76] Evangelos Stromatias, Miguel Soto, Teresa Serrano-Gotarredona, and Bernabé Linares-Barranco. An event-driven classifier for spiking neural networks fed with synthetic or dynamic vision sensor data. *Frontiers in Neuroscience*, 11:350, 2017. 14

[77] AMMAR MOHEMMED, STEFAN SCHLIEBS, SATOSHI MATSUDA, and NIKOLA KASABOV. Span: Spike pattern association neuron for learning spatio-temporal spike patterns. *International Journal of Neural Systems*, 22(04):1250012, 2012. PMID: 22830962. 14

[78] Răzvan V. Florian. The chronotron: A neuron that learns to fire temporally precise spike patterns. *PLOS ONE*, 7(8):1–27, 08 2012. 14

[79] Abouzar Taherkhani, Ammar Belatreche, Yuhua Li, and Liam P. Maguire. Dl-resume: A delay learning-based remote supervised method for spiking neurons. *IEEE Transactions on Neural Networks and Learning Systems*, 26:3137–3149, 2015. 14

[80] Razvan V. Florian. Reinforcement learning through modulation of spike-timing-dependent synaptic plasticity. *Neural Computation*, 19:1468–1502, 2007. 14

[81] J. A. Pérez-Carrasco, B. Zhao, C. Serrano, B. Acha, T. Serrano-Gotarredona, S. Chen, and B. Linares-Barranco. Mapping from frame-driven to frame-free event-driven vision systems by low-rate rate coding and coincidence processing–application to feedforward convnets. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(11):2706–2719, Nov 2013. 14

[82] Yongqiang Cao, Yang Chen, and Deepak Khosla. Spiking deep convolutional neural networks for energy-efficient object recognition. *International Journal of Computer Vision*, 113(1):54–66, May 2015. 15

[83] P. U. Diehl, D. Neil, J. Binas, M. Cook, S. Liu, and M. Pfeiffer. Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, July 2015. 15, 16

[84] Eric Hunsberger and Chris Eliasmith. Training spiking deep networks for neuromorphic hardware. *CoRR*, abs/1611.05141, 2016. 16

[85] Steven K. Esser, Paul A. Merolla, John V. Arthur, Andrew S. Cassidy, Rathinakumar Appuswamy, Alexander Andreopoulos, David J. Berg, Jeffrey L. McKinstry, Timothy Melano, Davis R. Barch, Carmelo di Nolfo, Pallab Datta, Arnon Amir, Brian Taba, Myron D. Flickner, and Dharmendra S. Modha. Convolutional networks for fast, energy-efficient neuromorphic computing. *Proceedings of the National Academy of Sciences*, 113(41):11441–11446, 2016. 16

[86] Davide Zambrano and Sander M. Bohte. Fast and efficient asynchronous neural computation with adapting spiking neural networks. *CoRR*, abs/1609.02053, 2016. 16

[87] B. Rueckauer, I.-A. Lungu, Y. Hu, M. Pfeiffer, and S.-C. Liu. Conversion of continuous-valued deep networks to efficient event-driven networks for image classification. *Frontiers in Neuroscience*, 11:682, 2017. 16

[88] Mandyam V. et al. Srinivasan. A proposed mechanism for multiplication of neural signals. *Biological Cybernetics*, 21(4):227–236, 1976. 16

[89] Catherine Gasnier. Implémentation d'une transformée de Fourier sur une architecture Neuromorphique. Master's thesis, Polytechnique, France, 2011. 16

[90] B. Rueckauer and S. C. Liu. Conversion of analog to spiking neural networks using sparse temporal coding. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, May 2018. 16

[91] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, Richard C. Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *CoRR*, abs/1704.04760, 2017. 17

[92] D. Rossi et al. Pulp: A parallel ultra low power platform for next generation iot applications. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–39, Aug 2015. 17

[93] Goodman et al. The brian simulator. *Frontiers in Neuroscience*, 3:26, 2009. 18

[94] J. Schemmel et al. A wafer-scale neuromorphic hardware system for large-scale neural modeling. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pages 1947–1950, May 2010. 18

[95] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana. The spinnaker project. *Proceedings of the IEEE*, 102(5):652–665, May 2014. 18

[96] M. Davies, N. Srinivasa, T. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse,

G. Venkataramanan, Y. Weng, A. Wild, Y. Yang, and H. Wang. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, January 2018. 18

Chapter 2

Implementing basic signal processing with spiking neural networks

Sommaire

2.1 Introduction

This chapter will focus on different implementations of logic and arithmetic functions using spiking neural network as building blocks. The ideas explored here were cited in chapter 1[1][2], using simple neuron model to build operator library using rate coding or temporal coding approaches. The main goal is to assert possibilities for both implementations to process the input data and have a high coverage in terms of functionality needed for converting ANNs. SNNs and neuron models defined here do not include learning rules as STDP or SDSP, and revolve around simple neuron models, mainly Leaky Integrate and Fire (LIF) and Integrate and Fire (IF) neurons.

Using simple mechanisms, we implement networks designed to compute any signal processing function on the input spike streams. Every network described in this section was implemented using XNet, part of N2D2 [3] developed in CEA-List. XNet is an event-based simulator for designing SNNs and was used to verify the functionalities and implement all studied networks with configurable neural parameters. Using this tool, we explore two main coding scheme for retrieving the data contained in spike trains: the rate coding and the Precise Timing (PT) coding. Rate coding means the input data is retrieved from the input spiking frequency, while precise timing uses the intervals in between spikes to retrieve data. The designed networks compute mathematical operations on those values and output spike streams conveying the results.

Implementing efficient signal processing and primitives for ANN conversion relies mainly on the Multiply ACcumulate implementation. In order to compare both approaches, we will first set the notations used in this chapter and expose the basic operator implementation leading to the MAC function in both strategies. Then we will use target functions to be implemented using both approaches and compare the obtained topologies and their characteristics. Using a set of metrics estimating energy, latency and functionality, we defined the most suitable approach to implement low power signal processing using SNNs.

2.2 Conventions

In this section, we are going to set the notations and conventions that we will be using for the rest of the paper at the data encoding and neural topologies levels. We will then define coding functions, conventions for spiking neural network definitions and the operators they implement.

2.2.1 Encoding and decoding functions

The first notion we need to define is the coding function. This function will be the one defining the relation between the data we want to compute on, and the way they are encoded with spikes. Let $s \in \mathrm{S} = [s_{min}, s_{max}] \subset \mathbb{R}$ be the data we are monitoring. s is contained within some known boundaries which are needed to define the boundaries of the spiking side. Using a rate coding for example, we can define a function c converting data the following way:

$$\begin{array}{rccc} c: & \mathbb{R} & \longrightarrow & \mathrm{F} \\ & s & \longmapsto & f_s.s + f_0 \end{array} \tag{2.1}$$

where F is the rate space, f_0 would be the spike train rate coding for a null signal and f_s the slope defining the relation between data and rate.

This is one of many examples that can be used in order to encode data in rate coded spike trains. Multiple inputs can be coded using multiple coding functions. Converting $s \in \mathbb{R}^n$, one can use an ensemble of n coding functions $(c_1, \cdots, c_n)$ that can be set according to our needs:

$$\begin{array}{rccc} c: & \mathbb{R}^n & \longrightarrow & \mathrm{F}^n \\ & (s_1, \ldots, s_n) & \longmapsto & (c_1(s_1), \ldots, c_n(s_n)) \end{array} \tag{2.2}$$

Once the data is translated into spikes, it is sent to the SNN performing its operation defined by the function f. The output of the used network is thus defined by:

$$\begin{array}{rccc} f: & \mathrm{F}^n & \longrightarrow & \mathrm{F}^m \\ & (f_1, \ldots, f_n) & \longmapsto & (f_{o,1}, \ldots, f_{o,m}) \end{array} \tag{2.3}$$

The set of output values $(f_{o,1}, \ldots, f_{o,m})$ defines the result of the implemented operator. They still need to be decoded, reversing the coding process:

$$\begin{array}{rccc} d: & \mathrm{F}^m & \longrightarrow & \mathbb{R}^m \\ & (f_1, \ldots, f_m) & \longmapsto & (s_1, \ldots, s_m) \end{array} \tag{2.4}$$

Finally, the operation performed by the whole network can be defined by:

$$\begin{array}{rccc} \mathrm{F}: & \mathbb{R}^n & \longrightarrow & \mathbb{R}^m \\ & (s_1, \ldots, s_n) & \longmapsto & (s_1, \ldots, s_m) \end{array} \tag{2.5}$$

with

$$\mathrm{F} = d \circ f \circ c \tag{2.6}$$

2.2.2 Spike train shape

Encoding and decoding functions have to be complemented by the definition of the spike train distribution. While keeping the same target rate, multiple “shapes” of spike trains can

be explored. The main spike train shape we will be using in this section are the regular spike train and Poisson process. For a regular spike train of frequency f, the definition of the inter-spike interval T is straightforward :

$$\mathrm{T} = 1/f \tag{2.7}$$

However, modeling inputs as regular spike trains is not relevant when interfacing with spiking sensors. We will also use Poissonian spike trains for which the probability distribution P handling intervals in between spikes is defined by:

$$\mathrm{P}(\mathrm{T} > \tau) = e^{-f\tau} \tag{2.8}$$

This distribution is memoryless, meaning the time to next event does not depend on the time elapsed since last event. At any time, the probability for a spike to arrive at a channel defined by the frequency f in the next τ interval is defined by:

$$p = \mathrm{P}(\delta t < \tau) = 1 - \mathrm{P}(\delta t > \tau) = 1 - e^{-f\tau} \tag{2.9}$$

This type of spike train is often cited to model biologically realist spike trains. The notations and distributions we will use in this section are now defined. We will now develop basis for describing our networks.

2.2.3 Neural models

The incoming spikes conveying data are sent to the synapses corresponding to their target neurons. As previously mentioned, networks defined here mainly use LIF or IF neurons. The main function defining the behavior of such neurons is linked to their membrane potential.

We here use a neuron with N inputs, each associated to a synapse of weight $w_i i \in [1, \mathrm{N}]$. We define the input activity function A, such that $\mathrm{A}_i(t)$ is worth 1 if the input channel i receives a spike at time t, otherwise 0. We then define I the input "current" at the membrane potential and V the membrane potential.

$$\mathrm{I}(t) = \sum_{n=1}^{\mathrm{N}} w_n \mathrm{A}_n(t) \tag{2.10}$$

$$\dot{\mathrm{V}} = \mathrm{I} - \mathrm{V}/\tau \tag{2.11}$$

where τ is defining the time constant of the leak associated to our LIF neuron. By removing the $-\mathrm{V}/\tau$ term, we obtain the equation of an IF neuron as no leak is involved. Then, a threshold V_{th} is defined such as:

$$V > V_{th} \Rightarrow \begin{cases} V = 0 \\ \text{The neuron emits a spike.} \end{cases} \tag{2.12}$$

Some neurons will also be allowed to have a negative output channel implementing a negative threshold:

$$V < -V_{th} \Rightarrow \begin{cases} V = 0 \\ \text{The neuron emits a negative spike.} \end{cases} \tag{2.13}$$

Another type of synapse will be used in later sections when implementing precise timing computation: the Linear integration synapses. Let P be the ensemble of regular or Punctual synapses and L be the ensemble of linear synapses. Previously defined I is modified as follow:

$$I(t) = \sum_{i \in P} w_i A_i(t) + \sum_{i \in L} w_i L_i(t) \tag{2.14}$$

where L_i defines the activity of the linear synapse i such as:

$$\begin{aligned} &L_i(0) = 0 \\ &\text{Linear synapse } i \text{ receives a spike} \Rightarrow L_i = L_i + 1 \\ &\text{Target neuron spikes } \Rightarrow L_i = 0 \end{aligned} \tag{2.15}$$

This set of equations will handle the behavior of every neuron from our networks for the following sections and chapters.

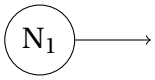
2.2.4 Neural networks notations

The base element of our network is the neuron, or node, which will be defined using the Fig.2.1.

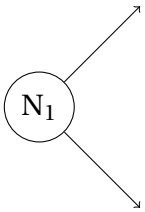
Figure 2.1: Simple node with name N_1

Output links from neurons will be defined one of the 3 ways defined in Fig.2.2. The links in-between neurons are always described by the direction and the synapse characteristics. For Punctual synapses, only the weight will be used describing links as shown in Fig.2.3.

When implementing the precise timing networks, we will add to the weight notation the type T (Punctual or Linear) and delay δt, time after which the input stimulus has to be itegrated, as depicted in Fig.2.4.



(a) Simple output link for “positive” spikes.



(b) Two simple output link for “positive” and “negative” spikes.

Figure 2.2: Output links notations.

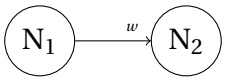


Figure 2.3: Link of weight w from N_1 to N_2

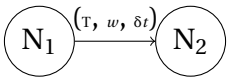
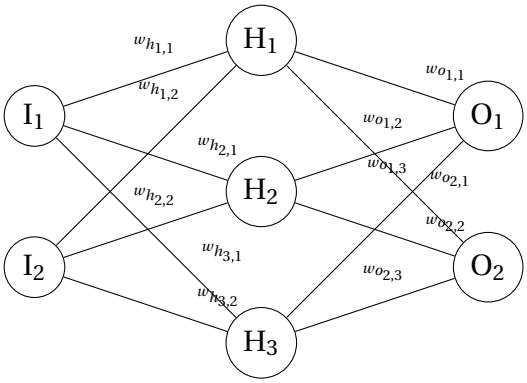
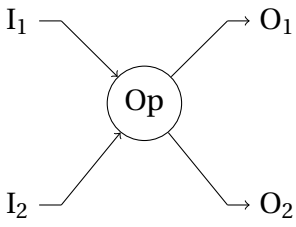


Figure 2.4: Link of type T, weight w and delay δt

Every networks defined in the following sections will be using those norms. As operators get larger, entities will be created in order to build hierarchically. Synthesized operators will keep their inputs and outputs as shown in Fig.2.5.



(a) Elementary network with 2 inputs, 3 hidden neurons and 2 outputs



(b) Resulting operator

Figure 2.5: Synthesizing Network

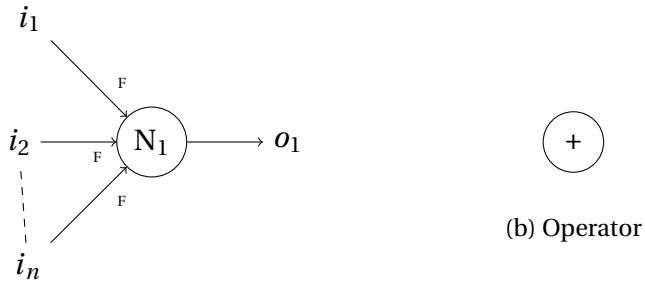
The network used in Fig.2.5a is summarized as an operator using the same inputs I_1, I_2 and outputs O_1, O_2 in Fig.2.5b. Now that the notations used for the rest of this section are defined, we will explore rate coding based operators.

2.3 Rate coding computation basics

This section will depict how we can encode and process input information by the use of spike train frequency conveying data [4][1][5][6]. As mentioned in Chap.1, using the frequency of spike trains to convey data has been widely used and ANN characteristics can be abstracted from it[7]. The coding c and decoding d functions are as described in section 2.2. The threshold potential V_{th} will be given the value 1 and the synaptic weight corresponding to a full neuron charge (0 to V_{th}) will also be given the value 1.

2.3.1 Addition

The most basic operation that can be done with input frequencies is the addition. In order to add frequencies, one just has to use an IF neuron and set every input synapse to stimulate the target neuron to its threshold potential as described in Fig.2.6.



(a) n input network

(b) Operator

Figure 2.6: Adding n input frequencies from i_1 to i_n channels. F stands for full threshold potential.

The function constructed with this operator is defined by f:

$$\begin{array}{rccc} f: & \mathrm{F}^n & \longrightarrow & \mathrm{F} \\ & (f_1,\cdots,f_n) & \longmapsto & \sum_{i=0}^{n} f_i \end{array} \tag{2.16}$$

where (f_n) are respectively the rate of the input channels i_1 to i_n.

Operations performed by those adders hold as long as the operation ranges are respected. Rate addition is respected as long as the result does not reach the maximum admitted spiking rate of the neuron. The order of incoming spikes do not influence the output results, and any spike train shape can be used. Moreover, if both inputs are Poisson processes then the output will also be a poisson process. However, regular spike trains as inputs do not create a regular spike train at the output.

2.3.2 Subtraction

Subtracting frequencies is not as straightforward as adding them. First, if negative membrane potentials are not allowed, implementations as shown in Fig.2.7 will only output f_1 as the negative weighted input will have no effect. Allowing negative membrane potentials and negative spiking using the same setup will only lead to f_1 being generated at the positive output and f_2 being generated at the negative input.

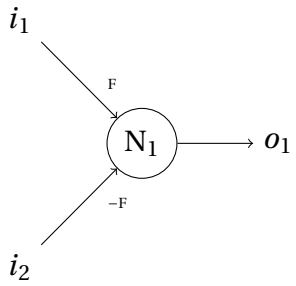
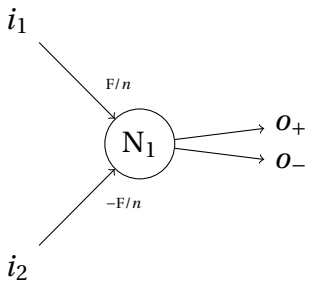


Figure 2.7: Intuitive subtraction.

In order to correctly compute the subtraction, one has to use intermediate states. Let n be the number of positive states we are going to use before spiking. Then, instead of using full weights, the new subtraction uses $1/n$ weights as shown in Fig.2.8.



(a) Subtraction of f_1 and f_2.

(b) Operator

Figure 2.8: Subtraction implementation with parameter n.

While using regular spike train as inputs, the output rate will either be distributed on the positive of negative channel and be worth:

$$f_o = \frac{f_1 - f_2}{n} \tag{2.17}$$

This result holds for Poisson process. Intermediate states can be analyzed as a Markov chain of length $2n-1$ as depicted in Fig.2.9. This Markov chain represents a variant of the "Birth and Death" problem with "borders" looping to the reset state. Assuming both inputs are Poisson process of respective rates λ and μ, the probabilities p_+ and p_- are defined the following way:

$$p_+ = \frac{\lambda}{\lambda+\mu} \qquad p_- = \frac{\mu}{\lambda+\mu} \tag{2.18}$$

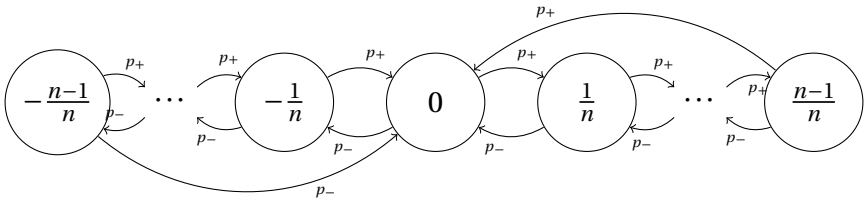


Figure 2.9: Markov process handling subtraction neuron's states.

Solving the Markov chain for given input rates λ and μ provides the equilibrium distributions and shows the influence of the parameter n with this subtraction using Poisson process. We define the matrix P composed of $\mathrm{P}_{i,j}$ defining the probability for a state i to state j transition:

$$\mathrm{P} = \begin{pmatrix} 0 & p_+ & 0 & \cdots & p_- & & & & \\ p_- & \ddots & \ddots & \ddots & & & & & \\ 0 & \ddots & & & & & & & \\ & \ddots & & & & & & & \\ & & 0 & p_- & 0 & p_+ & 0 & & \\ & & & & & & & \ddots & \\ & & & & & & & \ddots & 0 \\ & & & & & \ddots & \ddots & \ddots & p_+ \\ & & & & p_+ & \cdots & 0 & p_- & 0 \end{pmatrix} \tag{2.19}$$

We define $p(k)$ the probability vector at step k. $\forall k \in \mathbb{N}$, the probabilities follow those equations:

$$p(k+1) = p(k)\mathrm{P} \tag{2.20}$$

$$\forall i \in \{0, \cdots, 2(n-1)\}, \sum_{j=0}^{2(n-1)} \mathrm{P}_{i,j} = 1 \tag{2.21}$$

with the equilibrium distribution p being defined by the relation:

$$p = p\mathrm{P} \tag{2.22}$$

Solving equation 2.22 with equation 2.21 gives us the following p_i equilibrium values corresponding to the probability to be in the state i/n at any time:

$$\begin{aligned} \forall i \in \{-n+1, 0\} p_i &= \frac{\mathrm{X}^{-i} \sum_{j=-n+1}^{i} \mathrm{X}^{j-n+1}}{(n+1)\sum_{j=0}^{n-1} \mathrm{X}^j} \\ \forall i \in \{0, n-1\} p_i &= \frac{\sum_{j=0}^{n-1-i} \mathrm{X}^j}{(n+1)\sum_{j=0}^{n} \mathrm{X}^j} \end{aligned} \tag{2.23}$$

We can now compute the probability of state p_{n-1}:

$$p_{n-1} = \frac{1}{(n+1)\sum_{j=0}^{n-1} \mathrm{X}^j} \tag{2.24}$$

$$\text{or } p_{n-1} = \frac{1-\mathrm{X}}{(n+1)(1-\mathrm{X}^{n+1})} \text{ for } \mathrm{X} \neq 1 \tag{2.25}$$

(2.26)

And

$$p_0 = \frac{1}{(n+1)} \tag{2.27}$$

The frequency of the positive output is given by λp_{n-1}:

$$\lambda p_{n-1} = \frac{\lambda-\mu}{(n+1)(1-\mathrm{X}^{n+1})} \text{ for } \mathrm{X} \neq 1 \tag{2.28}$$

leading to the following cases for n great enough:

$$\begin{aligned} f_{o+} &= \tfrac{\lambda-\mu}{(n+1)} \text{ for } \mathrm{X} \ll 1 \\ f_{o+} &= 0 \text{ for } \mathrm{X} \gg 1 \end{aligned} \tag{2.29}$$

Providing λ and μ are different enough and n is great enough, the output frequency will be the difference of the two input frequencies. Three main issues arise from this implementation. Firstly, for input rate leading to cases close to X = 1, we have the following distributions:

$$\lambda p_{n-1} = \mu p_{-n+1} = \frac{\mu}{(n+1)^2} \text{ for } \lambda = \mu \tag{2.30}$$

Meaning for low n implementations and close to equal frequencies spikes can be emitted from both the positive and negative output channels.

Secondly, the result accuracy is closely linked to the number of intermediate states n to spike. Greater accuracy means greater n but also means lower output frequency as it scales with $1/(n+1)$. This scaling is necessary for correct output result but costly in terms of inputs needed to get the desired precision.

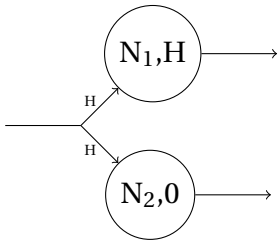
Finally, using this implementation with spike trains coming from the output of other operators which do not preserve the spike train shape can lead to undetermined behavior. As all of the operators do not maintain spike train shapes, measures have to been taken when implementing larger operators.

2.3.3 Multiplication per $\alpha \in \mathbb{R}$

Multiplication is crucial for implementing any mathematical operator. An efficient multiplication operator is mandatory. In order to multiply an input frequency by a factor α we are going to use a part of its binary representation:

$$\alpha = 2^m \sum_{k=1}^{n} \alpha_k 2^{-k} \text{ with } m \text{ such as } \alpha_1 = 1 \tag{2.31}$$

Then, we are going to split input channels using pre-charged neurons as shown in Fig. 2.10a. The upper neuron is half charged, spiking with the odd input spikes. The lower neuron is not pre-charged, meaning it will spike only with even spikes.



(a) Splitting odd and even spikes from input channel.

(b) Splitting operator

Figure 2.10: Operator used to split an input spike train in 2 trains containing respectively odd and even numbered events. H stands for Half the threshold potential.

Using the previously defined splitting network, we can use the binary coding of α to send a fraction of the input spikes corresponding to this factor as depicted in Fig. 2.11. The first splitting will send $1/2$ spike to the adder with with coefficient α_1, the second sends $1/4$ spike with coefficient α_2 and the same goes for every other splitting until the last one having only 1 output sending $1/2^n$ spikes from the input train to the adder with coefficient α_n. Thus, the output frequency can be defined the following way:

$$\lambda f_o = \sum_{i=1}^{n} \frac{\alpha_i}{2^i} f = f \sum_{i=1}^{n} \frac{\alpha_i}{2^i} = \frac{f\alpha}{2^m} \tag{2.32}$$

As for the subtraction operator, the multiplication operator used here will have different scale on the output and input sides depending on the used α coefficient. It is proven by construction to output the fraction of the input spike train corresponding to the binary representation of the weight. However, the multiplication implementation used here does not maintain the spike train shapes.

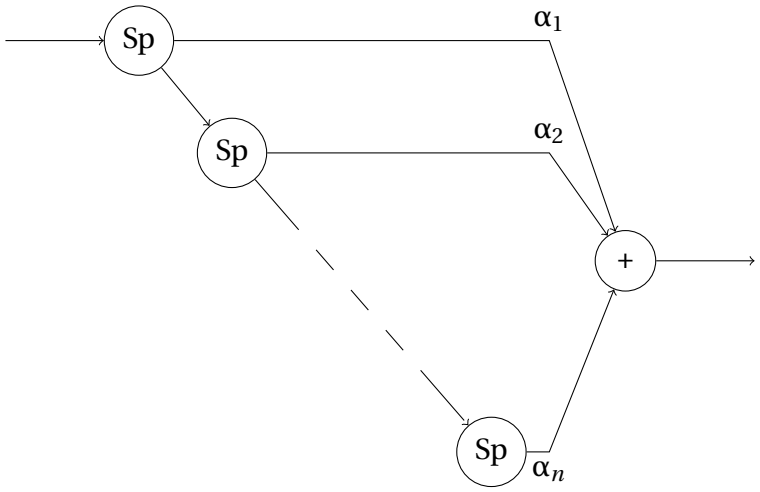


Figure 2.11: Split and add to implement frequency multiplication per $\alpha 2^{-m}$.

2.4 Precise Timing computation basics

In this section, the information conveyed by a spike train will be contained in the intervals in between spikes and their respective timings. The base for our operators was taken from [2] and some ideas will remain in our implementation.

The data encoded by 2 spikes coming from the same or different channels can be decoded the following way:

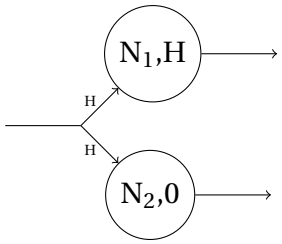
$$\begin{array}{rccc} d: & \mathrm{T} & \longrightarrow & \mathbb{R} \\ & \delta t & \longmapsto & a\delta t + b \end{array} \tag{2.33}$$

T being the interval space. This interval space is discretized using time windows, meaning two spikes arriving in the same timing interval will be labeled with the same time-stamp. Thus, the interval space T can be seen as $\mathbb{N}$ using the number of time windows in between two events.

The base mechanism used in this section are defined in 2.2.3. We will be using non leaky Integrate and Fire neurons with Punctual and Linear synapses. Only positive thresholds are considered in this implementation.

2.4.1 Input separation

The first elementary operator we use is splitting the input spike trains in two spike trains. This operator resembles the Splitting operator used for rate coding. Its aim is to create two distinct channels, the first one holding the first spike for every interval used for PT computation and the second one holding the second spike.



(a) Splitting input spike train into 2 spike trains holding interval information.

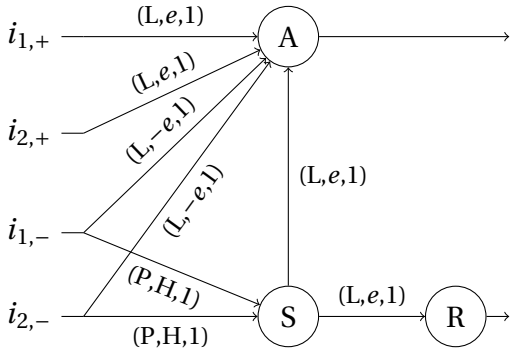
(b) Splitting operator

Figure 2.12: Same splitting operator as used in the rate-coded approach.

In later defined operators, the input channel denoted as “firsts” or + will hold spikes considered to be first and the “lasts” or - channel will hold spikes considered to be the second when coding intervals. Using separate channels for both first and second spikes can also provide room for “negative” intervals values.

2.4.2 Addition

Adding timing intervals is not as straight forward as adding frequencies. The operation can be done with Linear synapses charging a target neuron during the input δt. The result is then stored into the membrane potential of the target neuron and can be recalled with the use of another Linear Synapse. Fig. 2.13a shows the basic idea for the PT adder. P stands for Punctual synapse and L stands for Linear synapse. The weight e represents the elementary weight step used in the membrane potential.



(a) Splitting input spike train into 2 spike trains holding interval information. e stands for elementary weight. + inputs holds the first spike and − the second one for each interval.

(b) Adder

Figure 2.13: Operator used to add input intervals from two channels.

Fig.2.14 shows the operator computing basic interval addition. Two events are considered for both input channels. The first events are inducing a linear charge of elementary weight in the neuron (A) (Accumulation). The last events are stopping this linear charge at

neuron (A). When both last events have arrived, the neuron S for Synchronization will have received half its threshold potential twice and thus emits a spike. The output spike from (S) is recalling the value stored in (A) and starts charging (R) (Recall). The output value of the operator is encoded in between the first event coming from (A) and last event coming from (R).

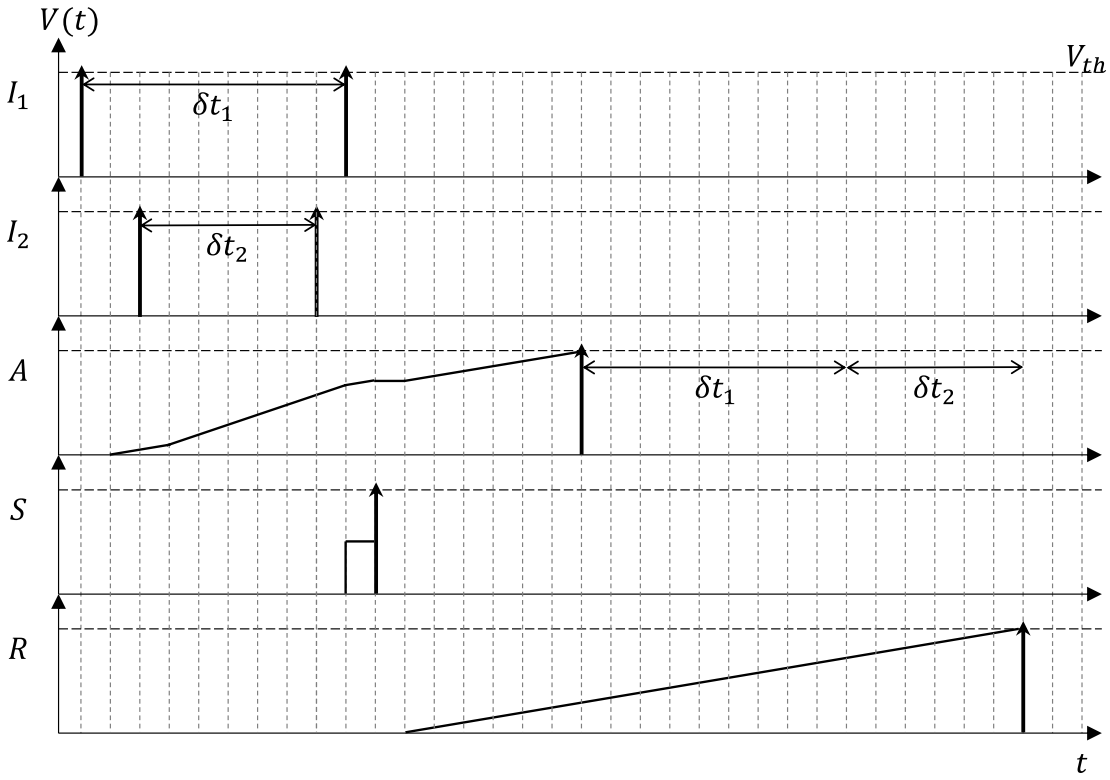


Figure 2.14: Adder chronogram.

This adder can be used with n inputs with $n \leq \mathrm{V}_{th}(\mathrm{S})/e$. The number of states the synchronization neuron (S) can hold is the limit for the number of channels it can synchronize. The Accumulation neuron threshold potential has to be adjusted in function of the input data range in order to avoid overflowing the operator as seen in Fig. 2.15.

2.4.3 Subtraction

Implementing the subtraction of input intervals can be easily done using the previously explained Addition. One only has to reverse the elementary weight connections in order to have negative integration on one of the intervals.

Once again, the output interval is coded in between events from (A) and (R) in Fig. 2.16a. One noticeable difference from the implementation in [2] is that we do not constraint ourselves to use the same channel for both the first and last spikes coding the interval inside our operators. This leads to dramatic reduction of the logic needed when output spikes are

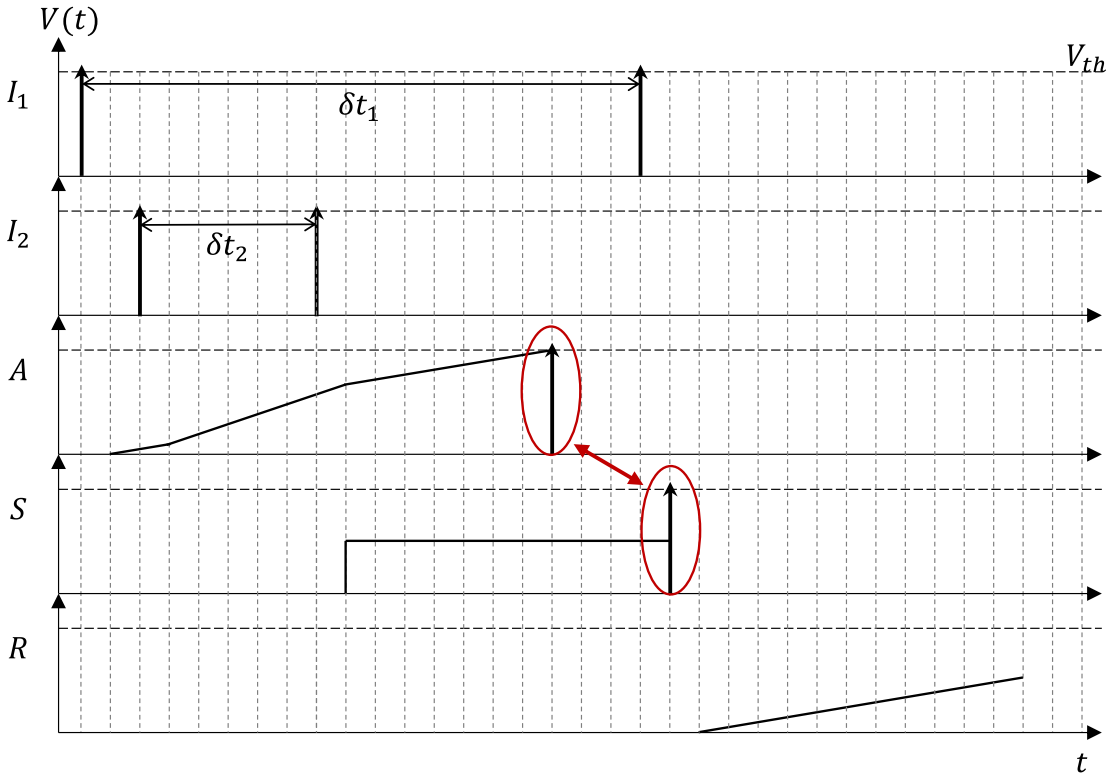
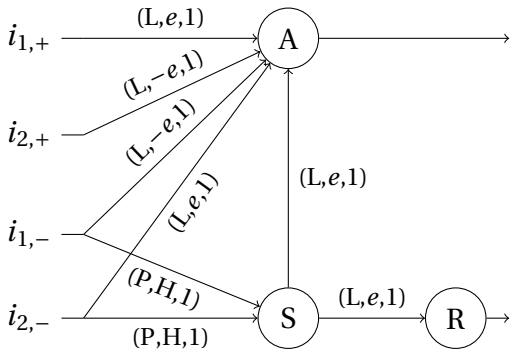


Figure 2.15: Adder chronogram including fixed-point addition overflow denoted by the accumulating neuron spiking before being recalled by the synchronization neuron.



(a) Subtraction generated by negative linear integration from the first spike of input 2.

(b) Subtraction operator

Figure 2.16: Operator used to subtract input time intervals from 2 channels.

supposed to encode for the data 0 for which both output spikes are needed to output during the same simulation interval.

In our implementation, negative values are coded with “negative” intervals, meaning if the output spike from (R) is sent before the output spike from (A) then the coded output value is negative as shown in Fig. 2.17.

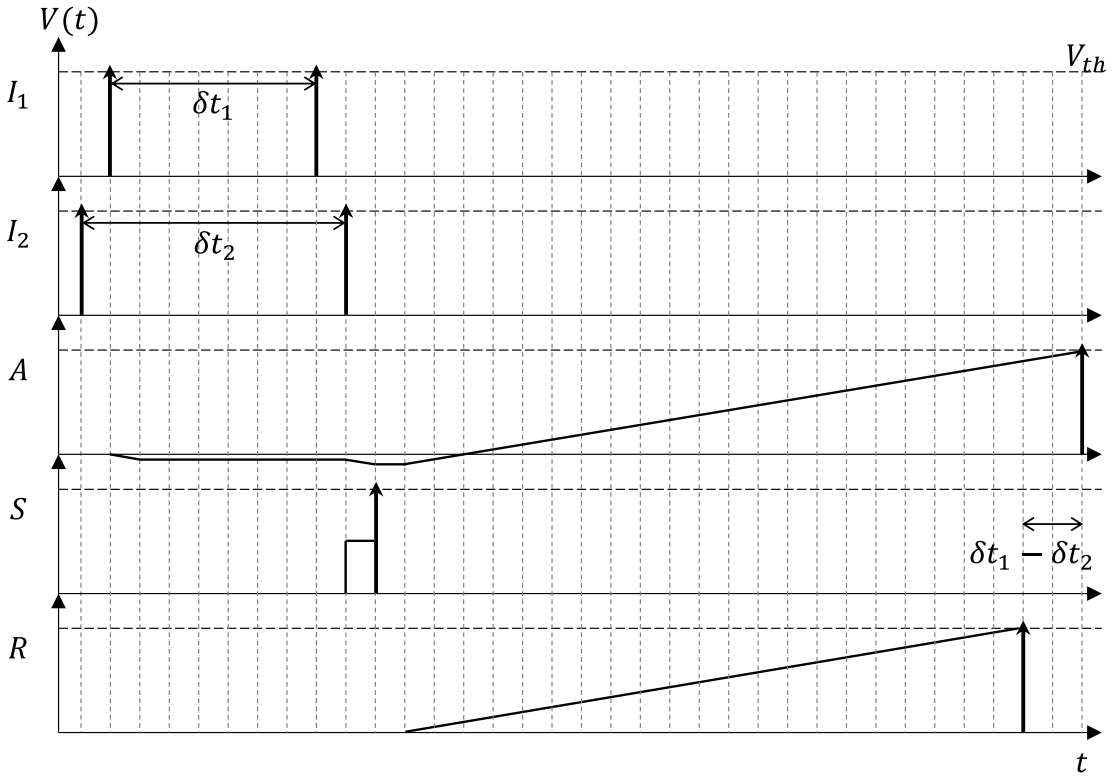
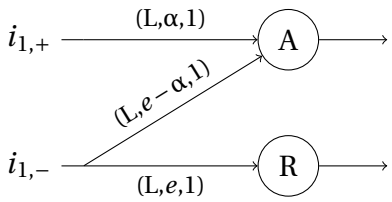


Figure 2.17: Subtraction chronogram.

2.4.4 Multiplication per $\alpha \in \mathbb{R}$



(a) Multiplication per α using linear synapse and 2 neurons.

(b) Multiplication operator

Figure 2.18: Operator used to multiply input interval per α.

Input intervals multiplication per α also uses the linear integration synapse in order to store the result value in the membrane potential of the accumulating neuron (A) as shown in Fig. 2.18a. This operator only has one input and thus does not need a synchronization neuron. The output value can directly be retrieved after the couple of spikes has been received from the input channels. Fig.2.19 shows the implemented operator function. Constraints to be respected on the threshold potential are also dependent on the input interval range and input weight.

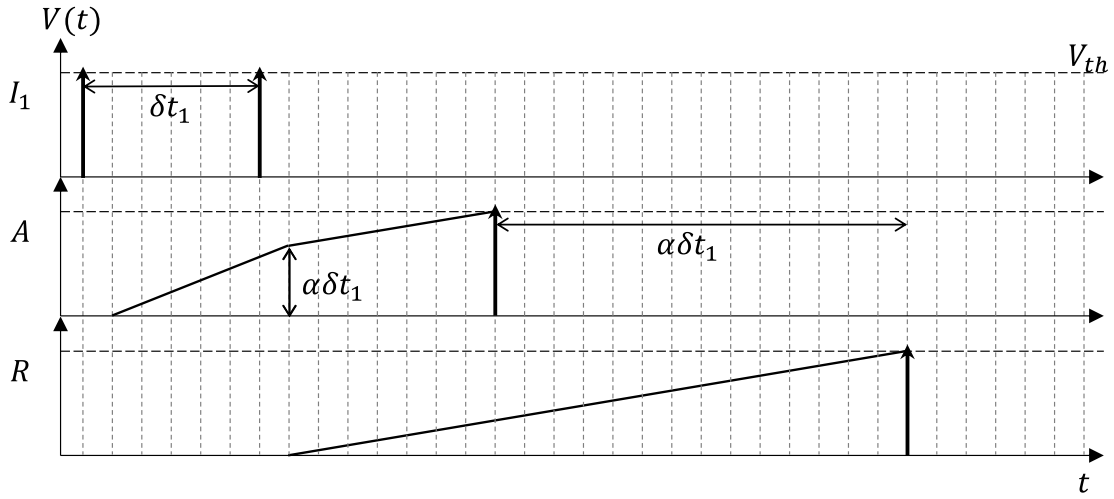
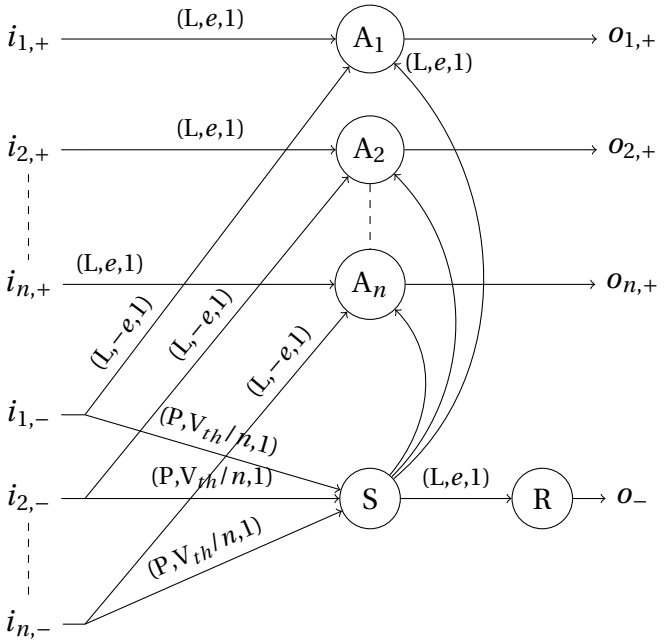


Figure 2.19: Multiplication chronogram.

2.4.5 Logic, memory and synchronization

The base framework [2] also offers possibility for signal synchronization, memorization and logic. Memory and synchronization examples were already used in the previously explained addition, subtraction and multiplication operators. In order to store and synchronize n input intervals, the topology shown in Fig.2.20a can be used. This simplified version of the synchronization originally proposed gives a common last spike to the n input channel without changing their value as shown in Fig.2.21



(a) Synchronization operator: store each value in a membrane potential and recall once channel transmitted its data. The output events have a shared second spike comming from o_-.

(b) Synchronization operator

Figure 2.20: Operator used to synchronize events in-between n channels.

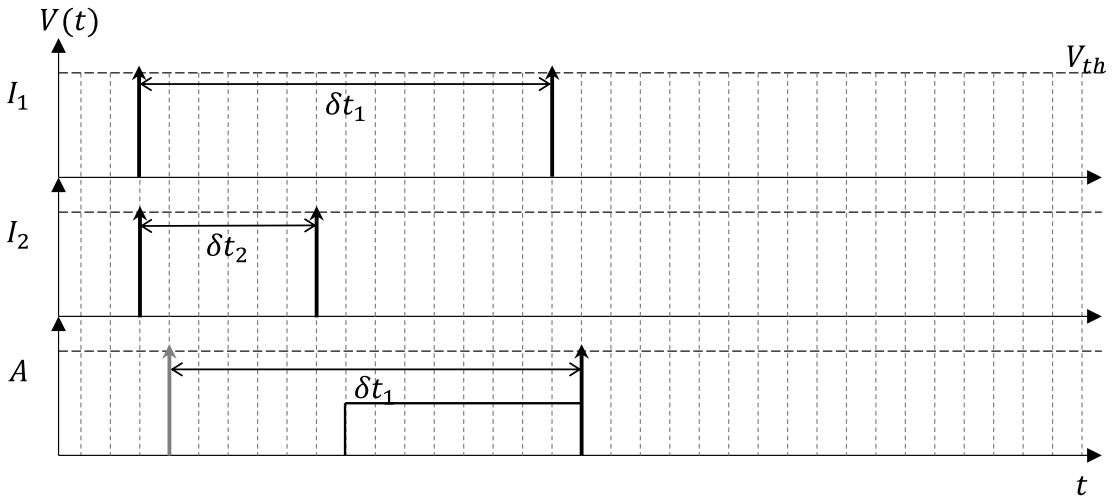
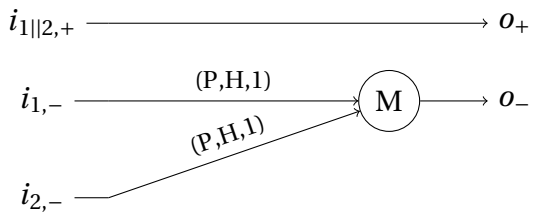


Figure 2.21: Synchronization chronogram.

PT also gives the possibility to compare input data and act accordingly which is not possible using a rate coded strategy. Fig.2.22a shows the topology used to compute the maximum of two input intervals. The first spike of each interval has to be synchronized in order to output the maximum of both intervals. This can be done with modifying the previous synchronization topology in order to output intervals synchronized on their first events.



(a) Maximum operator. The M neuron will spike once it receives 2 input spikes.

(b) 2-inputs maximum operator

Figure 2.22: Operator used to output the maximum of two intervals synchronized on the first spike.

Fig.2.23 shows the computation performed by the topology in Fig. 2.22a. As expected, the output interval is the maximum of both input intervals.

Interestingly, linking the neuron performing the Max operation to both first channels and synchronizing input intervals on the last event will result in performing the minimum operation.

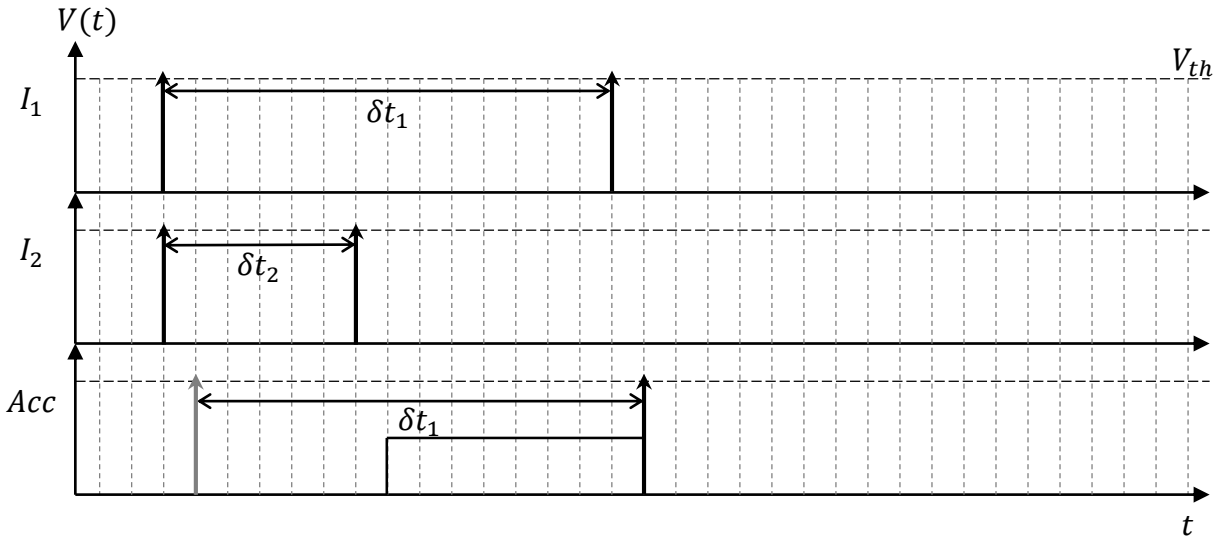


Figure 2.23: Maximum chronogram.

2.5 MAC and functionality comparison

The purpose of the topologies described in the previous part is to be able to perform digital signal processing and mimic ANN behavior interfacing with dynamic sensors. Those topologies naturally exploit the data contained in the spike trains sent by these sensors. Sensory data can be retrieved from the input events and conventional signal processing can be mimicked. Digital signal processing and ANN applications are essentially composed of Multiply ACcumulate (MAC) when used with Digital Signal Processor (DSP) or GPU. Typical Digital Signal Processing applications essentially use filtering, Fourier transform to extract valuable information. Filters 2.34 and transforms such as DFT 2.35 are essentially composed of Linear Combination performing the convolution of the input sensory data by filter weights:

$$\sum_{m=0}^{\mathrm{M}} a_m y_{n-m} = \sum_{k=0}^{\mathrm{N}} b_k x_{n-k} \tag{2.34}$$

$$\mathrm{X}_k = \sum_{n=0}^{\mathrm{N}-1} x_n . e^{-\frac{2\pi i}{\mathrm{N}} kn} \tag{2.35}$$

Various spiking topologies (rate or timing coded) let us perform basic arithmetic including Linear Combination, which is essentially a set of MACs. In order to determine the best approach to be used when performing event based signal processing we will compare the implementation of this basic element in both approaches: rate and timing.

2.5.1 Linear combination implementation and analysis

Basic examples of signal processing 2.34 2.35 can be performed assembling topologies defined in sections 2.3 and 2.4. In order to compare those implementations we need to define the metrics we are going to use.

Comparison metrics

As explained in section 1.4.2, the main metric used for hardware spiking neural network is the energy used per synaptic event. The energy consumption being dominated by synaptic weight storage and usage, building a metric linked to the future hardware usage of synaptic storage is natural. Then, for the same required output precision for our operators we define as first metric the average number of synaptic events used to provide the result.

The second metric is linked to the time resolution and delay of the implemented operators. For a set type of input and precision, we will compare the delay linked to the use of rate coding and precise timings. This metric can be expressed as a function of the time needed to encode desired data range and precision at the input.

Frequency coded implementation

Implementing the Linear Combination of input frequency can be done using the previously defined Multiplication, Addition and Subtraction operators as shown in Fig.2.24. It shows the topology corresponding to the following equation:

$$f_o = \sum_{k=0}^{m} \alpha_k f_k - \sum_{k=m+1}^{n} \alpha_k f_k \text{ with } \forall k \in \{1, n\}, \alpha_k > 0 \tag{2.36}$$

mimicking for instance a filter with m positive weights and $n-m$ negative weights.

In order to calculate both metrics, we will set a few conditions representing realistic operating conditions. First, for simplification purposes, we assume the multiplications do not use scaling factors (α coefficients are in between 0 and 1). We also assume that the positive and negative intermediate results stay within range of the maximal frequency, meaning the partial sum do not exceed the maximum spiking frequency allowed. The scaling included in the subtraction however has to be taken into account as it is mandatory to get correct results. We are able to calculate the scaling coefficient in between the input maximal frequency and the output maximal frequency.

Beginning with the multiplication operator, the first splitting unit uses $2f_i$ events per second, the second one $2f_i/2$, the last one $2f_i/2n^{n-1}$. The associated adder uses $\alpha_i f_i$

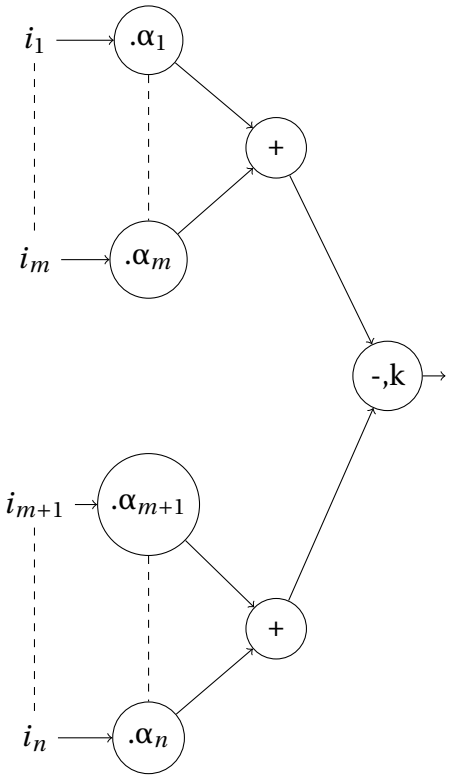


Figure 2.24: Full Linear Combination implementation using frequency coded operators.

events per second as it was designed to. The subtraction uses the sum of all events coming from the adders, meaning the sum of all $\alpha_i f_i$.

As shown in Fig.2.25, operators can be removed when their operation can be mixed with the next one. Here for instance, we use the subtraction operator in order to perform the previous addition thus saving $\alpha_i f_i$ for all channels. The number of events needed per second by this linear combination for N bits precision can be computed as follows:

$$f = \sum_{i=0}^{n} f_i \left(\left(\sum_{j=0}^{\mathrm{N}-1} \frac{2}{2^j} \right) + \alpha_i \right) \tag{2.37}$$

Evaluating this equation for a given output precision also depends on the parameter k used for the subtraction. A N bits precision at the output of the operation requires taking into account the $1/k$ subtraction scaling.

We define the input scale as 1. The scale entering the subtraction is equal to the sum of the filter weights, which is lower than $max(n-m,m)$. The scale at the output of the subtraction is equal to the sum of filter weights divided by k. Then, in order for our operator to maintain N bit precision at the output, the integration has to be held during a time proportional to the input scale. The relation between input and output scales can be defined as follow:

$$\mathrm{S}cale(out) \leq \frac{max(m, n-m)}{k} \mathrm{S}cale(in) \tag{2.38}$$

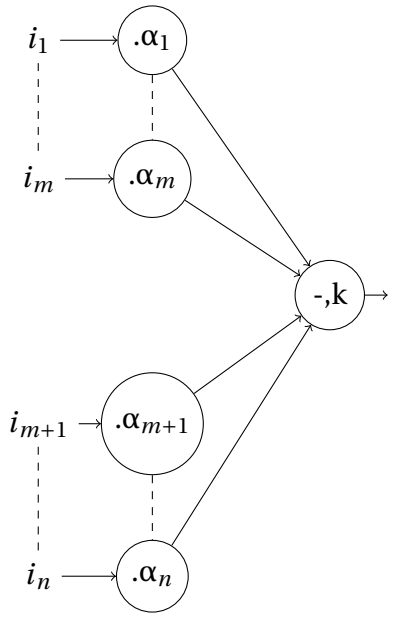


Figure 2.25: Simplified Linear Combination implementation using frequency coded operators.

Requiring N bits precision at the output will mean requiring at least a $2^N k/max(m, n-m)$ spike range possibility at the input. The number of synaptic events used can then be computed averaging input weights. Simplifying, we can round the following result :

$$5 \geq \left(\left(\sum_{j=0}^{N-1} \frac{2}{2^j} \right) + \alpha_i \right) \geq 3 \text{ for } N \geq 2 \tag{2.39}$$

as the sum of powers of $1/2$ will tend to 2 and (α_n) are contained in $[0,1]$.

Then, the number of spikes used for N bits output is greater than 3 times the scaled number of input spikes used:

$$SPK \geq 3 \frac{k.2^N}{max(m, n-m)} \tag{2.40}$$

Setting k close to the number of positive or negative weighted channels, we have a number of spikes used at least equal to 3.2^N.

In terms of time to result, setting k according to the number of inputs will lead to similar frequency range at the input and output, leading to an integration time proportional to the time needed on the input side to encode information.

Precise Timing implementation

We will now perform the same operation using Precise Timing operators. The same operator can be used adding splitting operators as pre-processing layer. Fig.2.26 shows this first naive implementation using all the operators previously defined.

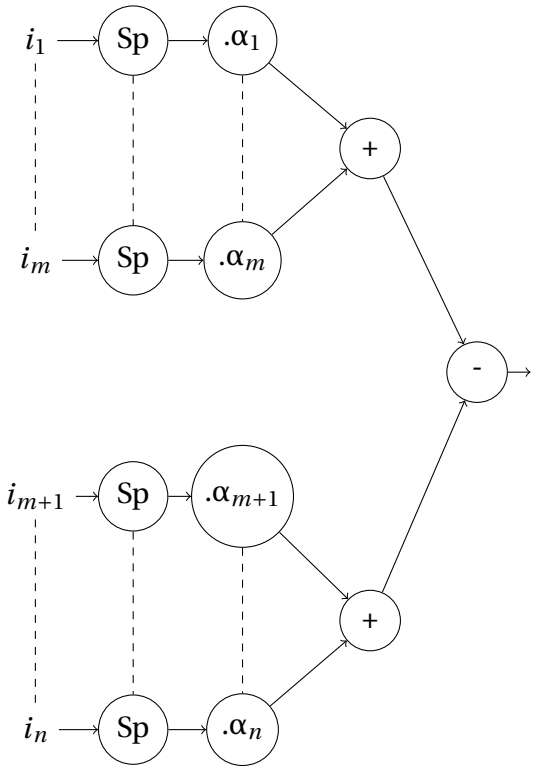
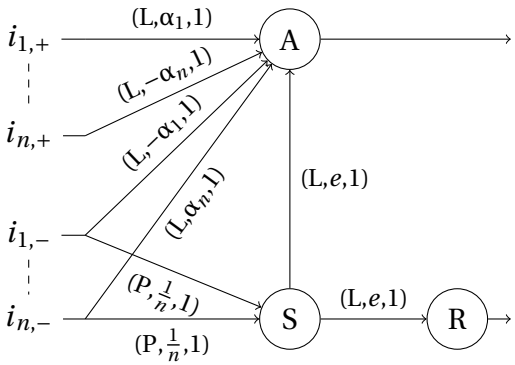
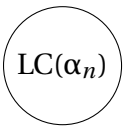


Figure 2.26: Naive Linear Combination implementation using time coded operators.

Fig.2.27a shows a more compact operator. As for the frequency coded side, successive operations can be concatenated into fewer operators when possible. All the multiplication, addition and subtraction can here be done using a single Linear Combination operator. The splitting cost will be counted in the metrics, but can be reduced for multiple layer operators as the first and last spikes will not be held again by the same channel using our topologies.



(a) All operations from linear combination can be done using only few neurons.



(b) Linear Combination operator with weights (α_n)

Figure 2.27: Simplified linear combination using time-coded operators.

The number of spike used per operation is easier to compute. As our data are encoded in spike intervals, input data is always held by couple of spikes. The splitting layer

consumes $4n$ synaptic events. Then, neuron A consumes $2n+1$ synaptic events, neuron S n events and neuron R 1 event. Thus, the total number of synaptic events used is $7n+2$ no matter the required output precision. The result delay is directly linked to the input interval range and output interval range. This means if we use N bits data at the input and require N bits fixed-point result, the output result is given after 2 maximum input intervals for storing and recalling the result.

2.5.2 Other metrics - Functionality

The two first metrics used give us estimates about the power consumption and output latency for the same operators. However, those metrics only do not provide information about the other advantages offered by each strategy: resiliency, logic, memorization or synchronization.

Sparsity advantage

Computing using the precise timing of the input spikes can be seen as using approximates of the instantaneous frequency. Dynamic sensors can produce typically low input event rates as in [8] with event rates lower than 2 kevts/s. Performing computation using instantaneous frequency estimates can utilize these sparse data situations in which rate coded networks would not always be able to retain information.

Moreover, synapses used in the time coded approach allow for data to be kept in membrane potential and logic operations to be performed on these. This can bee useful when implementing operations dependent of previous results. Possibilities to retain rates in rate coded network are quite limited. Using synaptic delays proportional to the time to code for information [9], output rates can be reused in later operations. Yet, those delays are fixed when implementing the network and cannot depend on logical operations for instance.

Room for improvements

However, time coded networks implemented for signal processing also have their drawbacks. First, in order to retrieve the result of any operation, a delay proportional to the output precision is needed.

Another advantage of the rate coded network is its resiliency to spike loss. The loss of input or internal spikes in rate coded networks is equivalent to change in the Least Significant Bit (LSB) of the data and the generated error is quite low. Loosing spikes in time coded

networks is destructive as it is equivalent to doubling the input data for locally regular spike trains for example.

2.5.3 Summary

Table 2.1 summarizes the metrics, advantages and drawbacks of both strategies when implementing base filter examples. Using the time needed at the input side to encode information as reference, both strategies have equivalent time to result ratios with delays linked to the time needed to represent input data. However, the number of synaptic events is linked to completely different factors in function of the coding used. The rate-coded approach uses a number of events linked to the target precision, while the precise timing approach is only linked to the number of inputs. Fig.2.28 shows the influence of the precision needed over the number of synaptic events used for a 4-input linear combination. The number of events used in the precise timing scheme is constant as precision increases while it increases exponentially in the rate coded scheme. SNN implementation based on rate coding requires care when sizing the neurons parameters not to consume as many synaptic events [10]. Using equivalent delay to result and fewer events per operation at fixed precision, using the data contained in the input intervals rather than their frequency will lead to energy savings.

Coding (N bits linear combination n inputs)	Rate	Time
Range in spikes/data	$\{0, 2^N\}$	2
Synaptic events	$3x2^N$	$7n+2$
Output delay	$\propto$ output range	$\propto$ output range
Output retention	Limited[9]	++
Sparse events	-	++
Short features	-	++
Logic	-	++
Spike train dependent	++	-
Spike loss/variation resilient	+	-
Safe operation (over/underflow)	Inherent	-

Table 2.1: Comparison of both strategies when implementing arithmetic and signal processing.

Table 2.1 also summarizes the main characteristics that will influence the capacities of the implemented topologies. The PT approach is able to compute features encoded by few spikes and has memorization and logic operators. The rate coded approach does not implement logic operators and can retain data for fixed period of time [9] dependent on the synaptic delay used. The rate coded implementation also revolves around precisely defined spike train shapes which it cannot always maintain through layered operators leading to un-

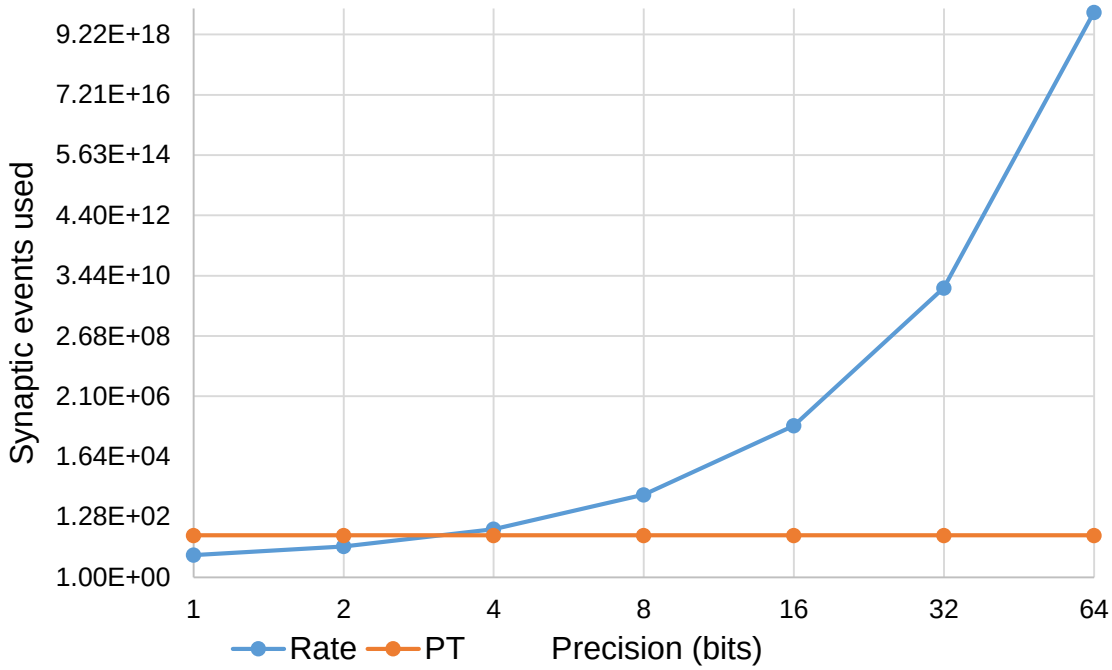


Figure 2.28: Number of synaptic events needed versus required precision for a 4-input linear combination.

determined behaviors. Thus, the PT approach offers more building blocks for implementing signal ANN essential functions such as max pooling, further validating our choice.

On the other hand, the rate coded approach is inherently safe to spike loss or input delay variations and also naturally performs secure operations as the output frequency is limited by the neuron itself. The PT approach uses couple of spikes to encode the information, meaning any variation of the inter-spike interval will influence the output result, and does not naturally implement secure Fixed Point operations.

2.5.4 Time Coding: issues and solutions

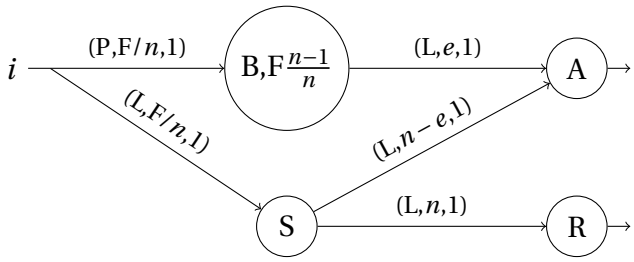
Table 2.1 exhibits the drawbacks associated with using timing to convey and compute data. This section is dedicated to solving the issues presented, mainly being sensitivity to spike variation and loss, output delay reductions and secure operations.

Input spike variations

Our topologies perform fixed-point operations on one-shot sensory inputs coded as intervals in between events. Thus, they exhibit important sensitivity to input variations and input spike loss or delay variations can greatly affect computed results.

Input intervals can be averaged on multiple consecutive inputs. Fig.2.29a shows

averaging operator based on an a modified adder. The neuron (B) used to start the operation need to be pre-charged so the first incoming spike will make trigger it. We use here $n+1$ consecutive events to load the synchronization neuron (S) which recalls the value stored in the accumulating neuron (A) with a linear synapse of weight $n-e$. It is modular and can be modified to use any number of inputs. Proofs can be found in Appendix B. This operator can be modified to average n input intervals coded with $2n$ input events to match data from active pixels in DVS for instance.



(a) The neurons B and S acts as a splitting operator filtering $1^s t$ and $n^t h$ spikes. These are used to start the accumulation and start recalling the stored value.

(b) n intervals average operator.

Figure 2.29: Operator used to average n consecutive intervals.

Using this topology, input delay variations can be mitigated at the cost of lower input sensitivity and higher delay to retrieve the result. Compromises can be made varying the value of n. Variation and loss sensibility will be mitigated by a factor n at the expanse of an increased delay to retrieve data.

Safe operations

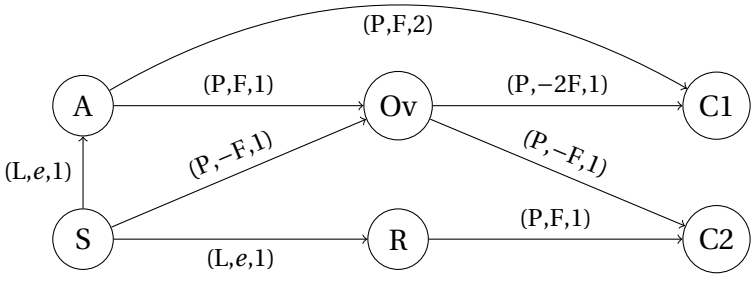


Figure 2.30: Overflow detection and filtering setting up a race between A and S. F weights denotes the full threshold potential. S spiking first inhibits the Overflow neuron Ov. A spiking first makes Ov spike to signal overflow.

Sizing neurons' and operators' parameters according to input data is not always sufficient to take care of overflow issues. The fixed point arithmetic introduces the possibility to report overflow spikes as shown in Fig.2.15. As we are also able to implement logic, another

way to implement secure operations is to flag an overflow. To do so, we setup a race in-between the Synchronization (S) neuron which is present in arithmetic operations and the linked Accumulation (A) neuron(s). The resulting topology is shown in Fig.2.30 and complete proof given in Appendix B. Any output spike sent before the synchronization neuron spikes will not be transmitted to the following operator as the Overflow (Ov) neuron will inhibit the Output (O) neuron. The Accumulation neuron has to wait for the Synchronization neuron to spike, stimulating the intermediate Output neuron and inhibiting the Overflow neuron. In those conditions only the Spike emitted from the output neuron will be transmitted to the next layer.

Result delay

One of the main drawbacks of our topologies is the link between result precision and the delay to recall result. As the output precision grows, the delay needed to retrieve result grows. This can be an issue on both the input and output side: multi-layer computation will be further delayed as latency stacks and new events can be sent to the topology when it did not finish yet previous input computation.

One solution is to split time intervals into coarse grain intervals and fine grain data. Perform parallel low precision operators and use logic to rearrange result. We use an input interval δt coded on 2N bits and a weight α on 2N bits. Then, we can split data and weights to form N bits packets:

$$\begin{aligned} \delta t_1 &= \delta t \text{ div } 2^N \text{ and } \delta t_2 = \delta t \bmod 2^N \\ \alpha_1 &= \alpha \text{ div } 2^N \text{ and } \alpha_2 = \alpha \bmod 2^N \end{aligned} \tag{2.41}$$

We then use 3 Linear Combination operators operating on N bits to perform the following operations corresponding to 3 N bits results.

$$\begin{aligned} o_1 &= \alpha_1 \delta t_1 \\ o_2 &= \alpha_2 \delta t_1 + \alpha_1 \delta t_2 \\ o_3 &= \alpha_2 \delta t_2 \end{aligned} \tag{2.42}$$

The 3 linear combinations need to be linked in order to transfer overflow from lower operators corresponding to the carry. Fig.2.31 simplifies the link between accumulators (operators computing LSB overflow spikes towards operators dedicated to Most Significant Bit (MSB)). The complete topology description for this operator is available in Appendix B. In this implementation, the accumulating neurons need to keep their linear synapse active even when spiking to continue the operation. Therefore, spiking equations are modified:

$$V > V_{th} \Rightarrow \begin{cases} V = V - V_{th} \\ \text{The neuron emits a spike.} \\ \text{Linear synapses maintain their values.} \end{cases} \tag{2.43}$$

The local carries are filtered with topologies equivalent to the overflow filter seen in Fig.2.30. Thus, carries are only transmitted to lateral operators and the output result is provided only when the synchronization neuron spikes.

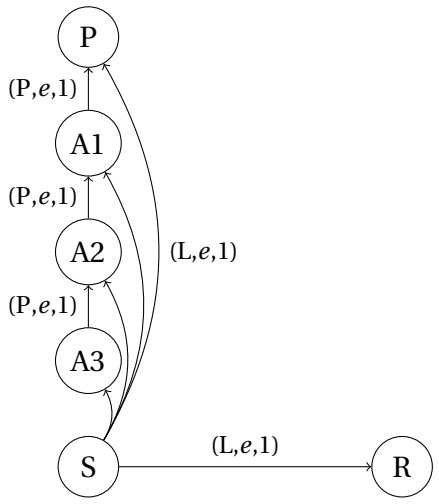


Figure 2.31: Carry in between linear combination operator 1,2 and 3. Carry from 1 goes to neuron P for 4N bits result. S and R are common in between the 3 linear combinations.

The principle shown here is enough for 1-input operations or unsigned fixed point operations. Multiple input signed implementations of the same operator needs two sets of operators computing the positive and negative components of the result. Then subtraction can be used at the output of each pair of positive and negative operation. The result sign is determined by the highest pair having non null result and time intervals are communicated accordingly.

As we split the data in two parts, the time needed to encode data and retrieve result is decreased by a 2^N ratio in term of number of elementary simulation time. However, this is at the expanse of higher number of synaptic events spent per operation. Input data are encoded in 2 couples of spikes sent to 3 operators with carry transferred in between them and output logic for result reconstruction. The number of synaptic events used is then highly dependent on the data and weights, being dominated by carry between operators. Using such topologies can be effective when requiring fast high precision results but should generally be avoided when aiming at low power applications.

2.5.5 Discussion

Precise Timing networks offer multiple advantages when compared to rate coded networks for low power applications generating sparse asynchronous events. They adapt to sparse event environments in which their rate coded counterparts cannot always extract the relevant information. Main drawbacks inherent to the precise timing topologies were mitigated

implementing specialized topologies. Depending on the application specification, the same neural mechanisms can implement different operations in order to adapt themselves to various operating conditions, leading to a flexible framework.

Table 2.2 updates the original results with new implemented features. The fixed-point operators can adapt to various environments using different neuron parameters and topologies. We show here that the Precise Timing approach makes a good candidate as it leads to lower number of synaptic operation required and covers higher number of essential points for converting ANNs. This base precise timing implementation will be further analyzed in the next chapter in order to implement low power hardware.

Coding (N bits linear combination n inputs)	Rate	Time
Range in spikes/data	$\{0, 2^N\}$	2
Synaptic events	$3x2^N$	$7n+2$
Output delay	$\propto$ output range	$\propto$ output range Decrease possibility
Output retention	Limited[9]	++
Sparse events	-	++
Short features	-	++
Logic	-	++
Spike train dependent	++	-
Spike loss/variation resilient	+	Implemented
Safe operation (over/underflow)	Inherent	Implemented

Table 2.2: Updated comparison with enhanced precise timing possibilities.

2.6 Références

[1] Catherine Gasnier. Implémentation d'une transformée de Fourier sur une architecture Neuromorphique. Master's thesis, Polytechnique, France, 2011. 31, 36

[2] X. Lagorce and R. Benosman. STICK: spike time interval computational kernel, A framework for general purpose computation using neurons, precise timing, delays, and synchrony. *CoRR*, abs/1507.06222, 2015. 31, 41, 43, 46

[3] CEA-LIST. N2d2, https://github.com/cea-list/n2d2, 2017. 31

[4] Mandyam V. et al. Srinivasan. A proposed mechanism for multiplication of neural signals. *Biological Cybernetics*, 21(4):227–236, 1976. 36

[5] Belhadj Bilel et al. Continuous real-world inputs can open up alternative accelerator designs. *SIGARCH Comput. Archit. News*, 41(3):1–12, June 2013. 36

[6] A. Joubert et al. A robust and compact 65 nm lif analog neuron for computational purposes. In *2011 IEEE 9th International New Circuits and systems conference*, pages 9–12, June 2011. 36

[7] Yongqiang Cao, Yang Chen, and Deepak Khosla. Spiking deep convolutional neural networks for energy-efficient object recognition. *International Journal of Computer Vision*, 113(1):54–66, May 2015. 36

[8] J. Anumula, D. Neil, T. Delbruck, and S.-C. Liu. Feature representations for neuromorphic audio spike streams. *Frontiers in Neuroscience*, 12:23, 2018. 53

[9] Peter U. Diehl, Guido Zarrella, Andrew S. Cassidy, Bruno U. Pedroni, and Emre Neftci. Conversion of artificial recurrent neural networks to spiking neural networks for low-power neuromorphic hardware. *CoRR*, abs/1601.04187, 2016. 53, 54, 59

[10] P. U. Diehl, D. Neil, J. Binas, M. Cook, S. Liu, and M. Pfeiffer. Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, July 2015. 54

Chapter 3

Precise Timing Architecture implementation

Sommaire

3.1 Introduction

Computing data contained in the intervals in between asynchronous events was proven efficient in term of synaptic events used in the previous and functionality in the previous chapter. A modified set of topologies was implemented using XNet in order to cope with the main drawbacks inherent to computing with precise timing information.

We now aim at developing hardware architectures taking advantage of the developed topologies and their characteristics. In order to define our generic support architecture, we give a number of possibilities and measure their footprints. Aiming at low power architectures, we will then analyze each of the possibilities explored using characteristics retrieved from our topologies.

A SystemC model was implemented based on the architecture we optimize for our topologies. The designed model represents the complete Globally Asynchronous Locally Synchronous (GALS) architecture and can be used for fast topology support verification, usage estimations and sizing. The generic architecture and developed model complete our design flow: create SNNs from application specifications or ANN (chapter 5), simulate those SNNs with the systemC model to size the needed chip and retrieve hardware characteristics estimations.

This first part of this chapter will be dedicated to depicting the possible solutions for our GALS architecture. The choices made will be justified using topologies from chapter 2 and retrieving their characteristics for efficient hardware usage. Finally, the SystemC model implementation and some results will be depicted.

3.2 Architecture - Storing topologies and scheduling activity

The topologies obtained aiming at time computing using SNNs generate a special set of characteristics. As it is the case for rate coded SNNs, the input is asynchronous using the AER protocol. The global ideas and modules will be developed here and the choices justified by analyzing the characteristics of our topologies. Considering those differences, we will now explain choices made at the architecture level.

The complete GALS picture is given in Fig.3.1. The main idea is to split the neurons composing our networks into multiple clusters in which the shared computation is time-multiplexed during each simulation. The synchronization of the simulation steps system-wide is done by the rendezvous bus, ensuring correct network simulation. The hardware time allocated to compute a simulation step will be called a Tick. Choices leading us to implement the shown architecture will be explained in the next sections. We now depict our

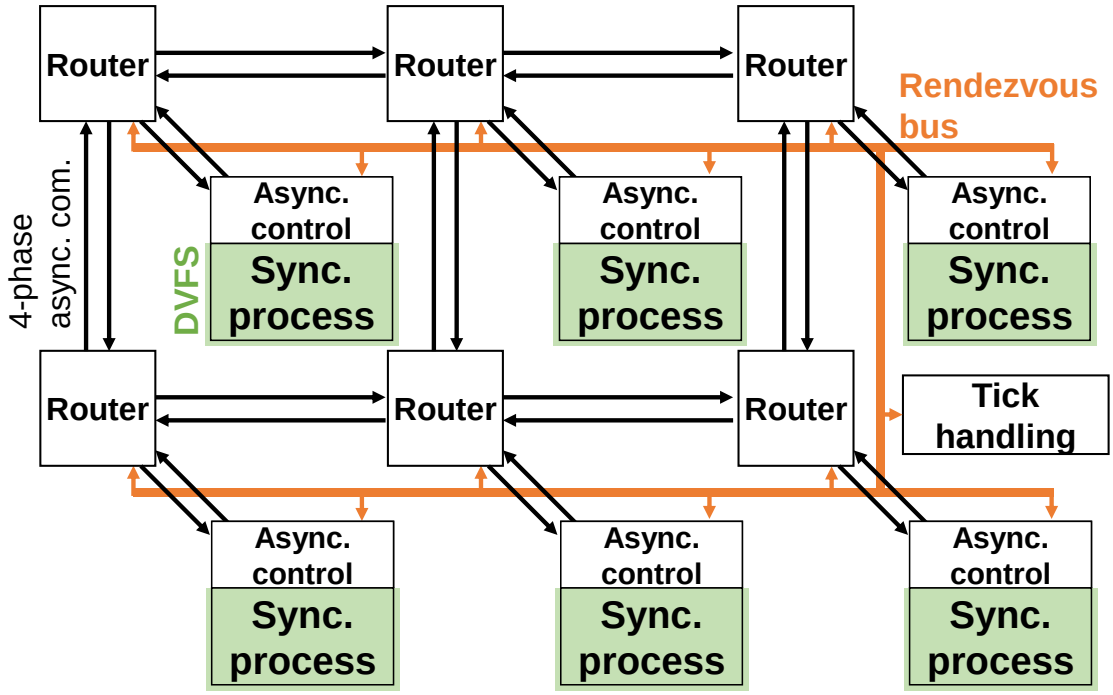


Figure 3.1: GALS architecture.

implementation following an input spike path, thus starting with the Asynchronous Network on Chip (NoC).

3.2.1 Asynchronous Network On Chip

The input spike is defined by its origin address and the timestamp at which it was emitted. The transmitted events are typically asynchronous and sparse [1][2][3]. The most efficient communication scheme for handling these sparse inputs is the asynchronous scheme.

An Asynchronous Network On Chip (ANOC)[4][5] distributed communication architecture fits the GALS paradigm. The served resources are implemented with the standard synchronous design methodologies and each belong to independent frequency and voltage domains. The GALS paradigm thus naturally offers possibility to implement local Dynamic Voltage and Frequency Scaling (DVFS) taking advantage of the frequency and voltage decoupling by construction in between the synchronous clusters.

Regarding power consumption, asynchronous design techniques show interesting dynamic power savings, as they are by definition un-clocked. They take advantage of local handshaking transactions as shown in Fig. 3.2. It offers the possibility for asynchronous circuits to be in a standby state when not solicited. The asynchronous logic scheme offers thus the equivalent of both Register Transfer Level (RTL) clock gating and architectural clock gating, but without the need of any additional software.

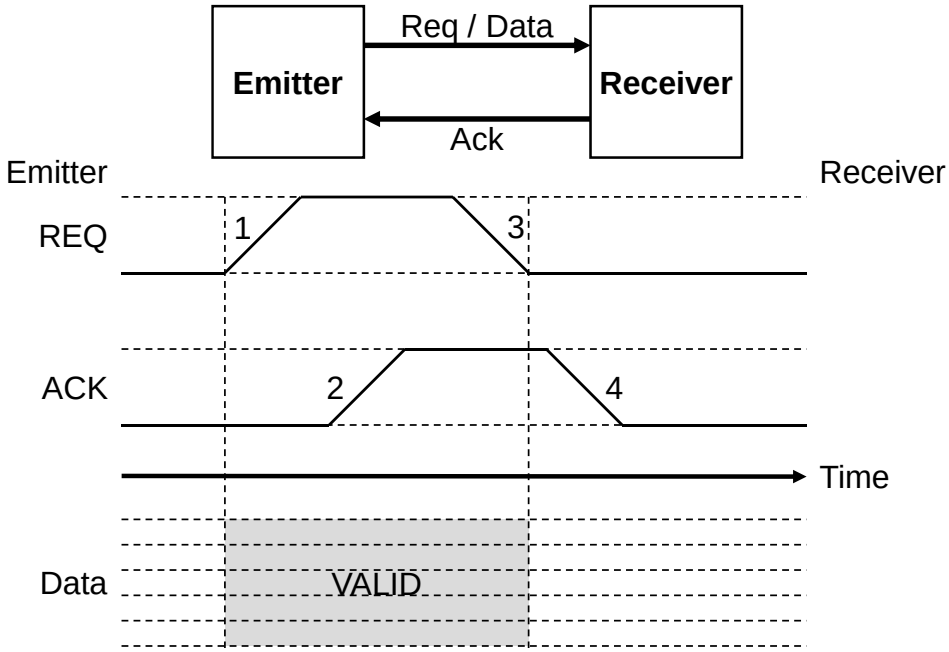


Figure 3.2: 4-phases handshake. Data is guaranteed valid in-between REQ and ACK being pulled up.

The 4-phase handshake is performed as depicted in Fig. 3.2. First, the side initiating the transaction is pulling the Request (REQ) channel up, asking to transfer data. The receiver side retrieves mentioned data and acknowledges reception with the Acknowledge (ACK) channel. Then, the emitter side acknowledges data reception resetting REQ. The last step is the receiver notifying the emitter it is aware of the end of data transfers resetting ACK.

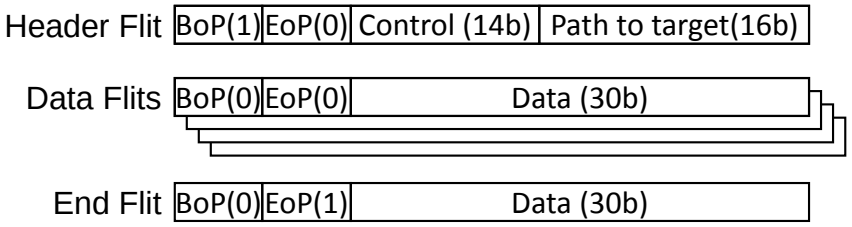


Figure 3.3: Packet composed of 2 flits of 32 bits.

The data communication can be defined on multiple levels, from bit to flit to packet. As shown in Fig. 3.3 a Flit is a defined amount of bits to be sent in between resources and a packet is an ensemble of flits. A packet is usually formed by a header flit (Beginning of Packet = 1), data flits and an end flit (End of Packet = 1). The header flit contains the control sequence (what to do, where in the module) and routing scheme (to what module deliver the data). The routing scheme is shown in Fig.3.4. The 4 directions North, East, South and West are respectively coded by 00, 01, 10 and 11. Local is coded by going in reverse as shown in Fig.3.4 where south encode Local when coming from south. Path To Target (PTT) is shifted 2 bits at every router until arrived at destination, permitting up to 7 jumps with 16 bits.

We use AER data during inference, meaning the data contained in an event are its origin and timing. Therefore, we can simplify the packet format to a single flit that ensures

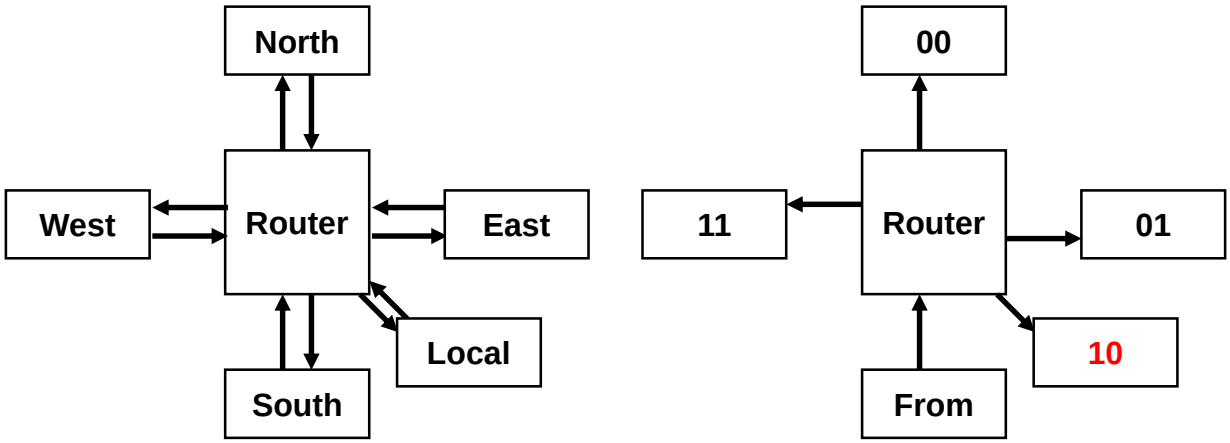


Figure 3.4: PTT explained.

both timing and origin information are maintained. Fig.3.5 shows the reduced flit size fitting AER.

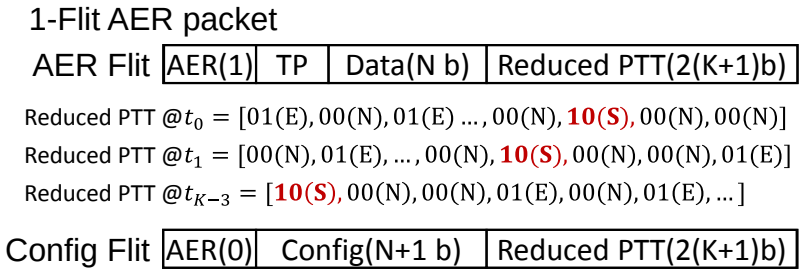


Figure 3.5: Mono-flit packet supporting AER. Timing is respected with the Tick parity bit and origin is determined with the data (neuron ID) and reconstructed PTT (cluster ID).

Using AER, the data are contained in the timing and origin addresses, two information we have to maintain. For the timing information, we have to ensure the flit is sent and received in the same time window defined as simulation tick. This is not an issue for flits being sent early in the simulation ticks but can become one when flits are sent late in the simulation tick, generating the possibility for those flits to be received at the beginning of the next simulation tick and thus conveying wrong timing values. The first measure ensuring a flit is conveying the correct timing value is using a Tick Parity (TP) bit. Each cluster knows the parity of the simulation step count or tick count and receiving a flit not matching the current bit count will induce special mechanisms for this flit to be processed in priority.

On the other side, the origin address is also maintained. The Data part being transferred in the flit is composed of the local ID of the origin node. This origin node can be part of a cluster of the architecture or part of the input channels arriving from the processed sensor. Retrieving the information of the input context can be made using the path-to-target information. As the flit is sent to different routers, the path-to-target part is undergoing circular rotation by 2 bits at every router as shown in Fig.3.5. Each router performs a routing operation and gives the prepared routing information to the next router by making the rotation.

This is used to retrieve the origin information when arrived at target cluster. The resulting path-to-target packet is then converted to origin ID and combined with the ID contained in the data.

We use the defined simplified AER packet to ensure both timing and address information are maintained. The same flit format can be used before inference in order to configure the different clusters. The sizing of the architecture will be developed later in this document.

3.2.2 Storing synaptic stimuli and updates

As data input a cluster, the corresponding origin ID is extracted and the path-to-target converted back to the origin cluster ID. For an incoming spike, we need to retrieve the group of target synapses and corresponding delays in order to know during which later tick we have to integrate our synaptic potentials. The main objective here is double: first, from a given input ID, retrieve and schedule the different synaptic stimuli to be sent to different target neurons. Then, after the corresponding synaptic delay, integrate the stimuli at the target neurons in target ID order. Retrieving and integrating all the stimuli targeting the same neuron at once allows us to retrieve its variables and compute its update only once per simulation step.

The main constraints we have to cope with here is the activity and connectivity heterogeneity specified in section 3.3, associated with multiple synaptic delays and possibility for multiple connections with different synaptic delay, type and weight in between 2 neurons. We thus implemented custom solutions that will be compared to reference designs sized for our needs. The main solutions explored for our needs are the synapse matrix, Content-Addressable Memory (CAM) based approaches and First In, First Out (FIFO) based approaches.

Synapse matrix

Synapse matrix has been used in multiple SNNs hardware [6][7][8][9][10] as it is generic enough to support multiple kinds of networks, the most demanding connection being the Fully Connected (FC) with usage close to 100%. Fig.3.6 summarizes the principle used here.

This principle can be used with n physical neurons or 1 physical computing unit time multiplexed to compute the update of the n neurons. The matrix is composed of $n+m$ columns, the n first columns being used for the recurrent connections and the m last columns for the links from nodes outside this cluster. Input spikes arrive at the decoder and stimulate the column of synapses corresponding to its ID. At each simulation steps, neurons at the end of each row integrates the synaptic stimulation scheduled for this simulation step.

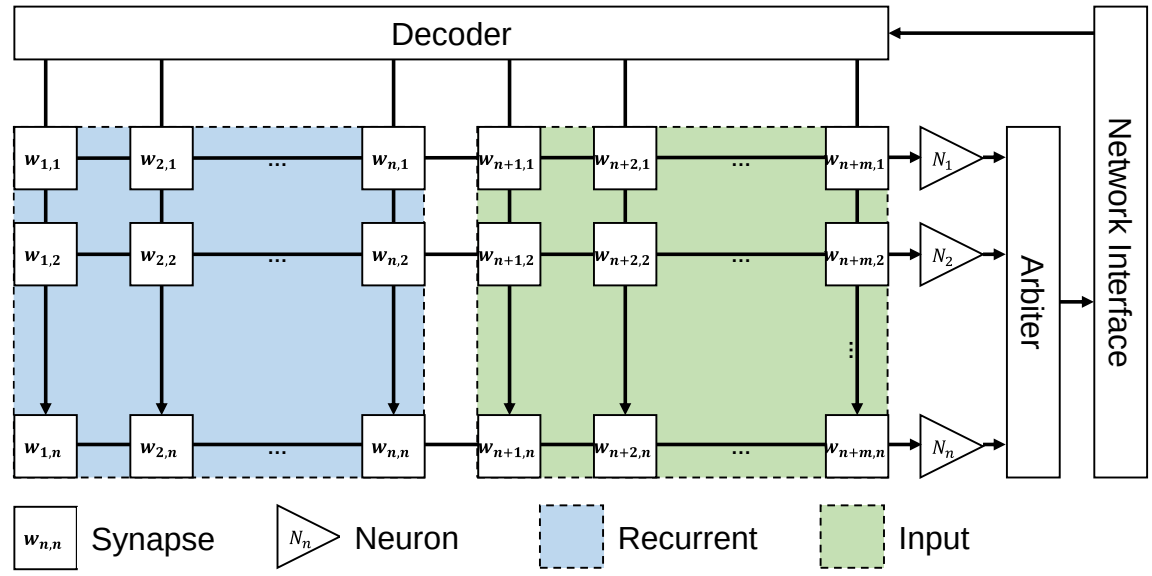


Figure 3.6: Synapse matrix enables high connectivity.

This enables fully connectivity from the $n+m$ origin neurons to the n neurons composing this cluster. This base implementation uses $n(n+m)$ synapses of 2^W bits. At each synapse, $d+1$ supplementary bits have to be used for the corresponding synaptic delays.

CAM-based approaches

An approach giving more flexible storing capacities is using Content Addressable Memory (CAM) [11]. We use here the simplified origin ID as an address to access the synapse memory and retrieve synaptic stimuli to be used at later simulation times. Fig. 3.7 shows the principle applied.

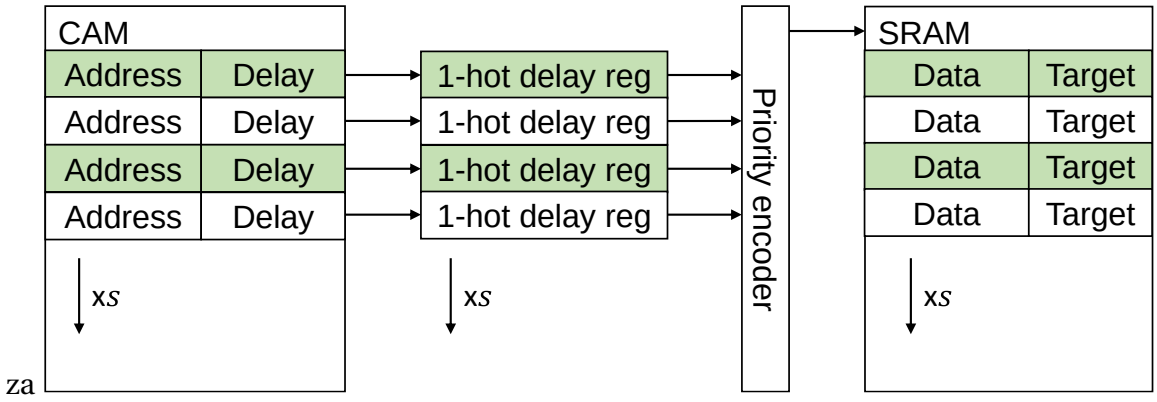


Figure 3.7: CAM oriented synapses handling enables flexible connectivity.

The input ID is used to retrieve target stimuli. Ternary CAMs can also be used for groups of inputs [11]: use only part of the input ID to match targets, essentially creating hierachical groups of neurons to be stimulated. The memory footprint can be extracted from Fig. 3.7. Each synapse has to include the following data:

- Input ID: In order to have the same input capacities as the synapse matrix implementation in which the number of input supported is $m+n$, we will here use M bits for the input ID storage. Each memory slot has comparison logic for retrieving stored data when the corresponding input ID is fed to the CAM.

- Synapse characteristics: here we store the synaptic stimuli information. The 3 characteristics that have to be stored are the type, weight and delay. When a synapse is stimulated, the delay has to be retrieved in order to schedule update at later simulation step. This leads to each synapse being composed of [Type(1),W$eight$(P),D$elay$(D)] bits to store the synapse value and $[d+1]$ to store its scheduled activation.

- Target ID: Allocating a fixed number of synapses to every neuron leads to highly inefficient storage usage. We thus need to store the target ID in order to decouple the number of synapses used from the number of neurons used. Doing so, we can allocate only 1 synapse to logic neurons for instance, increasing greatly the occupation rates of the synaptic storage memory. The target ID occupies N bits.

The CAM implementation thus uses M + (P + D + 1) + $(d+1)$ + N bits per synapse. Each synapse has its comparison logic and stores scheduling information when stimulated. The Priority Encoderis responsible for ordering the operations that have to be done during a simulation step. It sends input stimuli ordered by ID and keeps track of the updates that have to be performed for neurons with active linear synapses.

FIFO-based approaches

This last set of implementations is based on asynchronous-synchronous FIFOs designed to make the transition in between the asynchronous communication and conversion part and the synchronous process. This method was designed to lower the overhead generated by the input comparison logic and overall memory footprint used for storing synaptic stimuli and scheduling them.

Fig. 3.8 gives the base principle of this implementation. The conversion Static Random Access Memory (SRAM) are used for global and internal IDs to synapse ranges composed of the first slot to retrieve and the number of slots to be stimulated. The second SRAM is storing the synapse characteristics which are here its type, weight, delay and target ID.

The stimuli are then stored in the asynchronous synchronous FIFOs according to the target ID and synaptic delay. The Priority Encoder is used to synchronously retrieve the FIFOs' data. This version uses less inputs than the one used in the CAM based implementation which took as many inputs as there are synapses. Here as stimuli are already ordered by

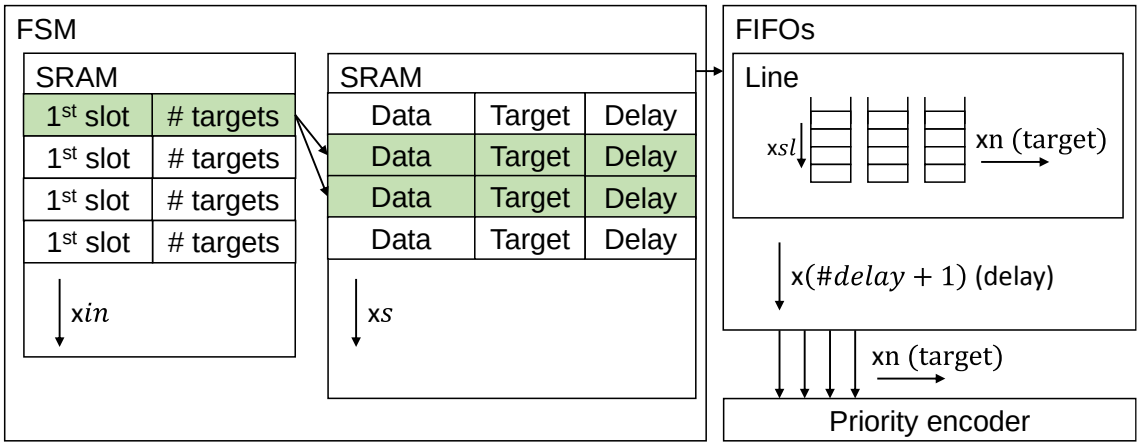


Figure 3.8: SRAMs + FIFOs implementation enables flexible connectivity.

target neuron, the priority encoder only sends the address of the last non empty FIFO and its data to the synchronous process.

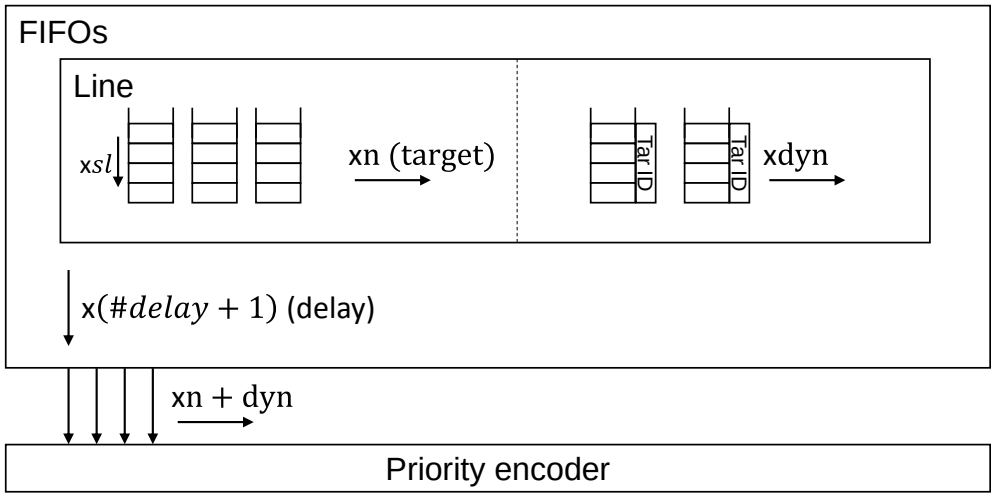


Figure 3.9: Dynamically allocable FIFOs are allocated when the regular FIFO to be written in is full.

The number of FIFO slots available per neuron should be equal to the maximum number of synaptic stimuli a neuron can receive during a simulation step, leading to FIFOs targeting logic neurons to be always near empty and inefficient. A second implementation shown in Fig.3.9 is used with static ID FIFOs (linked to a fixed neuron) of reduced size able to receive average input activity and few dynamically allocable FIFOs to be allocated when a static ID FIFO is full.

There are two main differences in between these implementations and the previous CAM based approach. First, the target synapses are not determined by comparing the input ID to each slot's address but by reading a set of synapses in the second SRAM. This implies retrieving the input stimuli uses as many successive reading operations as there are synapses to stimulate while performing no comparison. Secondly, the output information is retrieved per neuron ID and not monitoring the state of each synapse in the cluster lowering the cost of the output scheduling logic.

The presented approaches can each achieve good performances in specific cases. The correct usage of such architecture depends on the supported topologies and their characteristics. In order to determine which solution could be appropriate for us, we have to keep in mind topology constraints.

3.3 Topology aware scheduling modules

Each of the previously defined scheduling modules could simulate the implemented topologies and applications. However, they are based on different approaches and may not perfectly fit our characteristics. Thus, the first step is withdrawing the main characteristics generated by the implemented topologies. We can model generic DSP/GPU computation using an array of Linear Combinations performing filter operations. Fig. 3.10 shows the topology obtained implementing an array k of linear combination using n input each. The Linear combinations used here have two accumulators mimicking the strategy used in 2.5.4 ensuring full precision while still keeping low delay. This does not represent exactly a filter or an ANN layer but is quite relevant to the average DSP/GPU usage.

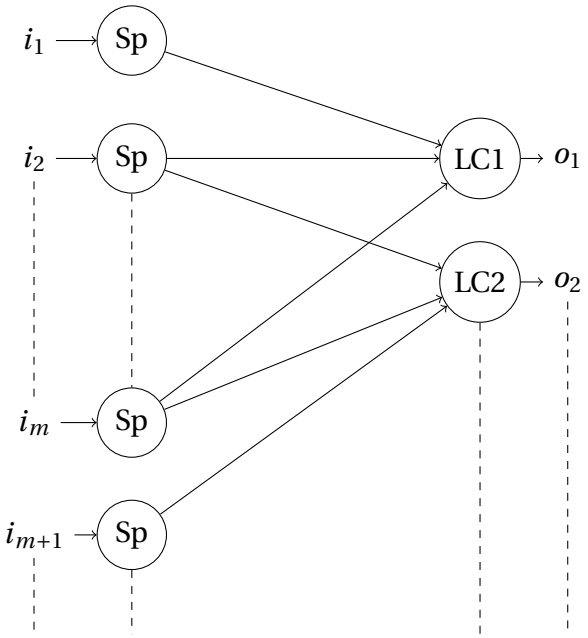


Figure 3.10: Array of Linear Combination used for extracting main characteristics of topologies implementing signal processing.

For simplification purposes, the linear combination coefficients are not given in the graph shown in Fig.3.10. As it is composing the main part of the computation performed by ANNs and DPSs, main characteristics and differences with respect to more conventional neural network will be extracted from this topology.

3.3.1 Connectivity heterogeneity

Biological neural networks have high connectivity, with an average number of input connections of 10,000. Topologies used in machine learning also use a high number of input connections per neurons. For example, each neuron belonging to a Fully Connected (FC) layer will by connected to every neuron of the previous layer. CNN reduces this number of connection implementing fixed size kernels in order to extract progressively larger features. Depending on the size of the kernels used, the number of input connections per neuron can also be important.

The important characteristic to be extracted here is that these networks all have quite homogeneous figures in term of number of input connections per neuron belonging to the same layer. Emulating neurons' behavior on GPUs, FPGAs or ASICs is facilitated as every node uses comparable number of input weights that can sometimes be shared.

The previous operator defined in Fig.3.10 is also able to perform kernel computation neurons from a convolutional layer would perform. The first main difference here is the number of connections used per neuron. Table 3.1 gives the average number of connection used per neurons for each type of neuron of this implementation.

Operator	Neuron	Input connections	Output connections
Splitters	First	1	k
	Last	1	$2k$
Linear Combination	Acc1	$2k$	2
	Acc2	2	2
	Synchronization	k	5
	Recall	1	$2o$
Output filtering	Carry	2	1
	Transfer	2	o

Table 3.1: Number of input and output connection per neuron. Topology from Fig.3.10. k is the number of input, o the number of outputs.

As shown in Table 3.1, the number of input connection per neuron is heterogeneous. Emulating the implemented topologies using an Application Specific Integrated Circuit (ASIC) requires carefully designing synaptic stimuli storage capacities not to waste resources.

Another point not shown in this table is the possibility to have multiple connections in between the same pair of neurons. Fig. 2.30 from section 2.5.4 illustrates the need for multiple connections in between two neurons, which is mainly used to check the arrival of 2 signals at the same time step at the target neuron. This is another characteristic that

differs from conventional ANNs and that will need special care when designing the hardware support. Section 3.2 will explore solutions to this issue and clearly exhibit the downside of homogeneous storage solutions.

3.3.2 Activity heterogeneity and secure timing operations

In this section we analyze the activity generated by our topologies in order to determine scheduling capacities specifications. We use the topology defined in Fig.3.10 with different target precision and retrieve average activity per simulation step on the different channels.

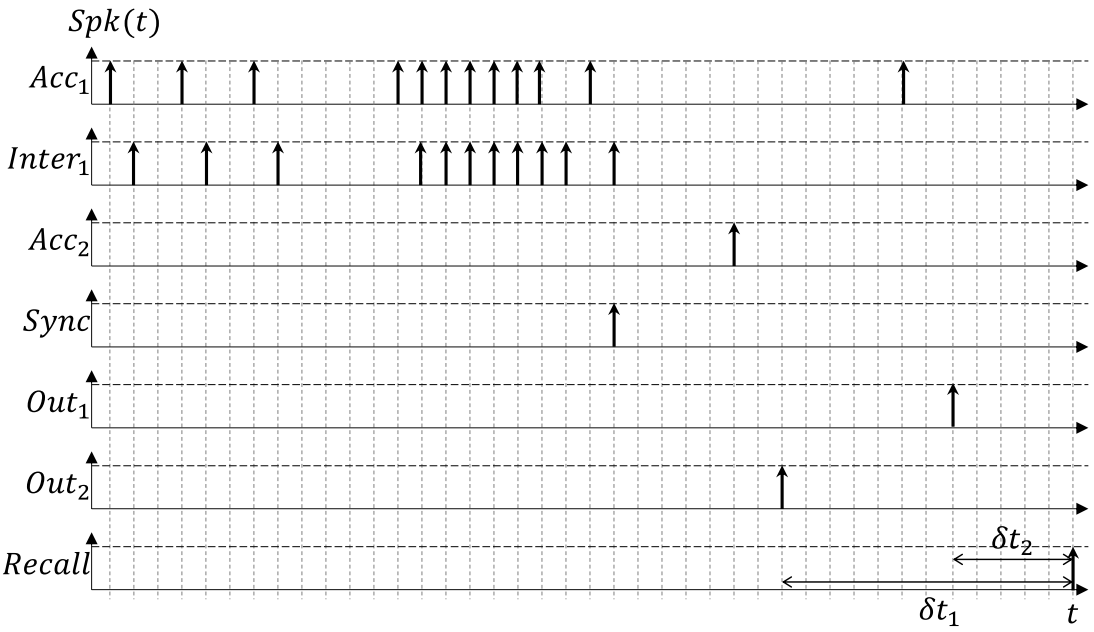


Figure 3.11: Activity for an 8b Linear Combination with 4b inputs.

Fig.3.11 shows the activity generated by a 8b Linear Combination with 4b input precision and 4 inputs simulated with XNet. Each neuron of the operator emits only 1 to 2 spikes while computing the result with the exception of the first accumulator which transmits carry to the second accumulator. This operator has result delay equivalent to the input delay as its output interval is divided into two channels.

The simulation using time steps as done with XNet will be kept at the hardware level. Meaning a fixed amount of hardware time will be dedicated to the update of the whole network for one step which corresponds to the elementary time step used in simulation. The hardware time step length depends on the application and the implemented operators. We can thus explore activity generated per time step with XNet in order to know the required hardware event scheduling capacities.

Table 3.2 shows average activities observed for N bit precision operation. As we use precise timing to encode data, the amount of spikes used per channel per operation is at

Operator	Neuron	Average input activity	Spk per synapse
Splitters	First	2	2
	Last	2	2
Linear Combination	Acc1	$2k$	1
	Acc2	2^{N-2}	1
	Synchronization	k	1
	Recall	1	1
Output filtering	Carry	$1-2^{N-1}$	$1-2^{N-2}$
	Transfer	$1-2^{N-1}$	$1-2^{N-2}$

Table 3.2: Network activity per computation. Topology from Fig.3.10 used with N bit operators.

most 2 for regular operators. Thus, the generated number of synaptic events per synapse per operation is also low, and the number of input spikes per neuron directly dependent on its number of inputs.

Each operator thus contains neurons having various needs in terms of activity. Some being more solicited as the accumulator neuron and other being only stimulated once per operation, mainly the Recall neuron. The hardware scheduling of neuron updates and stimuli integration needs to be able to update only currently active neurons and also retrieve only stimulated synapses.

The last point concerning activity is the synaptic delay and number of available connections in between 2 given neurons. As explained in chapter 2 the implemented operators can have synaptic delays varying from 1 to multiple elementary simulation steps. This aspect is essential for implementing logic operators. The second point here is the need for any given couple of neurons to use two links having the same direction with two distinct synapses. This is important for implementing punctual stimulation followed by inhibition needed for coincidence detection or condition checking as shown in chapter 2.

We thus have networks having heterogeneous connectivity and activity rates. In order to fully take advantage of those aspects, the implemented synaptic stimuli storage and scheduling modules will have to be carefully designed to minimize energy consumption and usage. Heterogeneous connectivity will be proven efficiently exploited by asynchronous heterogeneous synaptic stimuli storage allocating synapses to any neuron per cluster. Heterogeneous activity will affect both synaptic stimuli storage and scheduling units in order to minimize the activity generated per active neuron at each simulation step.

3.3.3 Scheduler analysis

Topology constraints

The synapse matrix based implementation does not fulfill all the needs corresponding to our topologies. We have seen in chapter 2 that some logic operators need couples of neurons (A,B) with 2 (A → B) links with different synapses and delays.

Implementing those double connections can be done multiple different ways, each given with their respective overhead using synapse matrix:

- Double the number of synapse at each slot. Overhead: at most 1 double connection per row is used, leading to under $1/(n+m)$ second slot occupation rate.
- Modify our topologies to include intermediate neurons. Each oriented couple having 2 links will be replaced by 3 neurons. The inserted dummy neuron is used to avoid having double links. Overhead: inserted neurons use a neuron and its $n+m$ potential synapses for only 1 input connection.
- Modify hardware to be able to locally allocate 2 rows to 1 neuron. Overhead: as for the previous solution, one row is almost empty.
- Modify hardware to be able to locally allocate 2 columns to 1 input neuron ID. Overhead: same as previous versions, this column will only be used for 1 of its coefficients.

None of the solutions specified above permit a compact implementation fulfilling our needs. Moreover, not all nodes in our topology need the provided input connectivity. The connectivity heterogeneity demonstrated previously will greatly influence the occupation rates of this type of structure. Assuming 40% logic involved neuron in a typical network, the maximal filling rate of our synapse matrix drops to around 60%. Arithmetic involved neurons are also impacting negatively this occupation rate as the number of synapses used is directly linked to the number of inputs of the implemented operator. The same point can be made for synchronization neurons.

The CAM based implementation solves the occupation rate issues at the cost of higher memory footprint per synapse and computational cost. It is flexible enough to allocate any given number of synapse to a neuron and allow double connection in between a single pair of neurons.

Compared to the CAM-based implementation, the FIFO-based implementation is as flexible. However, the method used to retrieve groups of target synapses to be stimulated uses no matching logic a CAM would use. The expense here would be longer time to retrieve all mentioned stimuli, but the activity sparseness helps us mitigate such delay cost.

Occupation rates, footprint and other metrics

We now characterize and compare the previously defined approaches according to a set of metrics of interest. Those metrics will include occupation rates, memory footprint, the activity generated per an input spike, the activity generated retrieving the output stimuli and updates to be integrated. Those metrics provide hints about the efficiency of the usage and efficiency of our synaptic storage capacities. We define the following variables:

- n the number of neurons per cluster ($2^{\mathrm{N}} = n$).
- nb_i the number of allowed inputs ($2^{\mathrm{NB}_i} = nb_i$).
- s the number of synapses ($2^{\mathrm{S}} = s$).
- p the synaptic weight levels ($2^{\mathrm{P}} = p$).
- T, W and D the synapse characteristics ($2^{\mathrm{D}} = d$).
- tar the number of targets of an input spike ($2^{\mathrm{TAR}} = tar$).
- sti the number of stimuli to retrieve.

Strategy	Matrix	CAM	Static FIFOs	S. + dyn. FIFOs
Memory footprint	$n.nb_i.$(T + W + D + d + 1)	$s.$(NB$_i$ + D + d + 1 + T + W + N)	nb_i(S + TAR) + s(D + T + W + N)	nb_i(S + TAR) + s(D + T + W + N)
FIFOs footprint	-	-	$n.(d+1)$ $.tar.$(T + W)	$n.(d+1)$ $.tar.$(T + W)
Occupation rate	$\ll$40%	Up to 100%	Up to 100%	Up to 100%
Average syn/neur	nb_i	s/n	s/n	s/n
Peak allowed syn/neur	nb_i	s	s	s
Time to convert input	O(1)	O(1)	O(tar)	O(tar)
Activity per input	-	s comp. + tar.R	tar.R/W + 1R	tar.R/W + 1R
Time to retrieve stimuli	O(sti)	O(sti)	O(sti)	O(sti)
Activity per stimulus	1R	1R + PE(s)	1R + PE(n)	1R + PE(n)

Table 3.3: Comparing the efficiency of the synaptic stimuli and update scheduling modules.

Table 3.3 gives the comparison of the implemented synaptic stimuli and update storage and scheduling modules. We can see the influence of the different parameters in the memory footprint here. For CAM and FIFO based implementation, the number of allowed inputs is not constrained by a static synapse-to-neuron partition. The occupation rates shown here suppose proper sizing of the storage module using the topology metrics and arithmetic over logic ratios determined previously.

The more compact CAM and FIFO based implementations also induce overheads. For both of them, priority encoders are needed at the output to schedule the process. The priority encoder used for the CAM based implementation takes the s synapses as inputs when the one used in the FIFO based implementation uses data from n FIFOs.

Using realistic figures, we can compare both compact approaches. We will use here clusters of 512 neurons admitting up to 512 inputs, two synaptic types with 5 bits signed weights and 1 bit of delay. The peak reception capacity per neuron per simulation step is fixed at 32 stimuli. Each cluster includes 8192 synapses, for an average of 16 synapses available per neuron. The peak permitted number of targets per input spike is fixed at 32. Static FIFOs in the first implementation will have 32 slots when static and the 32 dynamic FIFOs in the second implementation will have 4 slots.

Strategy	Matrix	CAM	Static FIFOs	Static + dyn. FIFOs
Memory footprint	5120 kb	232 kb	146 kb	146 kb
FIFOs footprint	-	-	288 kb	38.25 kb
Total	5120 kb	232 kb	434 kb	184.25 kb
Scale	27.78	1.26	2.36	1
Average syn/neur	1024	16	16	16
Peak allowed syn/neur	1024	8192	8192	8192
Time to convert input	-	comp. time	up to 33 R	up to 33R
Activity per input	-	8192 comp.+ <32R	1R + <32R/W	1R + <32R/W
Time to retrieve stimuli	O(sti)	O(sti)	O(sti)	O(sti)
Activity per stimulus	1R	1R + PE(8192)	1R + PE(512)	1R + PE(512)

Table 3.4: Example implementation.

Table 3.4 shows implementation characteristics for the 4 approaches with the given figures. Allowing 1024 input addresses requires having a 512x1024 synapses matrix with the first implementation, which is highly ineffective for our topologies. CAM and FIFO approaches both satisfy the low average number of synapse per neuron and high potential peak synapse per neuron taking advantage of the connectivity heterogeneity. Both the CAM and Dynamically allocable FIFOs cases have comparable total memory footprint

The main overhead of the CAM based approach is the use of input comparison logic, larger output PE and memory footprint. FIFO based approaches on the other hand have to retrieve synapses characteristics one by one and store them in the intermediate FIFOs. The FIFO-based approach including the dynamically allocable FIFOs will be used for the rest of this chapter as Scheduling module.

3.4 Architecture - Synchronous process and scaling conditions

We exhibited the choices made for the AER protocol, event reception and stimuli scheduling. The next step is to proces the scheduled stimuli given by the scheduler.

3.4.1 Pipelined update

The conversion and scheduling modules shown in previous sections are designed to send synaptic stimuli grouped by target ID after their synaptic delay is elapsed. The output of the conversion + scheduler modules is handled by the Priority Encoder and engages a synchronous 4-stage pipeline. Fig. 3.12 shows the output of our FIFO based implementation.

ID	Type	Weight
476	Punctual	5
476	Linear	3
476	Punctual	4
450	Punctual	0
449	Linear	5
422	Linear	4
422	Linear	-4
412	Punctual	16
400	Punctual	-16
322	Linear	7

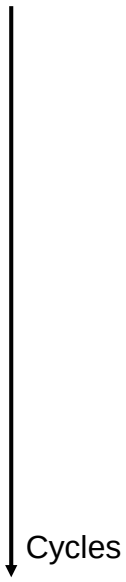


Figure 3.12: Output of the scheduler module during a simulation step.

Data received synchronously from the scheduler begins a 4-stage pipeline to process the network update. The pipeline process is described in Fig.3.13

As synapse stimuli are grouped by target ID, we need at most 1 read and write of the neuron variables per simulation step. The first step of the synchronous process is making the memory requests to retrieve synaptic stimuli when a new target neuron ID is met. This part is done by the manager module, designed to monitor changes of target IDs and send requests to the memory accordingly. When a synaptic stimulus arrives, the synapse's characteristics are kept to be sent to the computation unit a cycle later corresponding to the cycles needed to retrieve the neuron variables if we encounter a new target ID. If the target ID changes, the manager also sends an update order to notify the computation unit it has to write back variables into the DPRAMs.

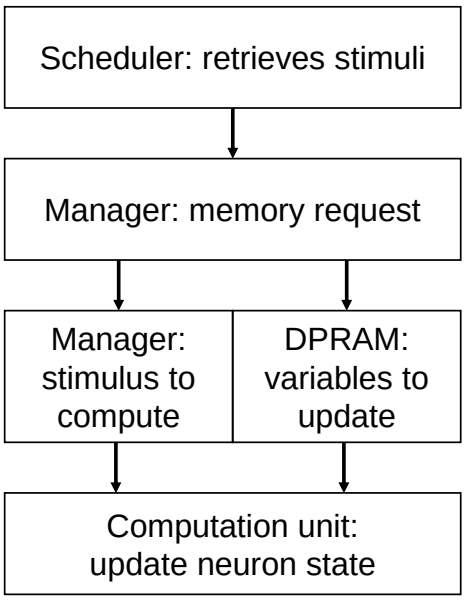


Figure 3.13: Stimulus processing pipeline, from the scheduler to the computation unit.

The shared computation unit is performing the following operations for each active neuron:

- Receive the neuron variables including the neuron membrane potential and the linear synapse level.
- Receive 0 to sti synaptic stimuli and update the corresponding variables. Punctual synapses add their weight to the membrane potential and Linear synapses add their weights to the linear synapse level.
- Receive 1 update order, add the new linear synapse level to the new membrane potential.
- If the linear synapse level is not null, sends this information to the scheduler in order to be recalled to update it at later simulation steps.

The computation unit also checks if the neuron emitted a spike or not. It compares the updated membrane potential to the threshold value and:

- If lower, just writes back updated membrane potential and linear synapse values.
- If higher, checks if the neuron has to be resetted or not when spiking and act accordingly. Resetting a neuron means setting their membrane potential and linear synapse levels to 0. Not resetting a neuron while still spiking means subtracting the threshold potential to its membrane potential and keep the same linear synapse level.

Each active neuron generates at least 1R and 1W to the DPRAMs setting the neuron variables and computes 1 addition on the membrane potential or linear synapse level per synaptic stimuli and update processed. Fig.3.14 shows the output of the computation unit for a simulation step using random activity. The left part represents the input data from both the manager and DPRAMs, the center part represents the operation performed during the current cycle and the right part represents the outputs. Using the Scheduler and manager we achieve 100% usage of the computation unit during the update of the active neurons of the network. This means the workload that has to be processed during a given simulation step can be known in advance monitoring the number of slots taken by the synaptic stimuli and updates in the scheduler.

To NI		**To DPRAM**				**To Scheduler**
ID	Spike	ID	En_w	Core	Linear	Update
476	-	476	0	-	-	-
476	-	476	0	-	-	-
476	1	476	1	4	0	0
450	0	450	1	5	5	1
449	0	449	1			0
422	-	422	0	-	-	-
422	1	422	1	3	7	1
412	0	412	1			0
400	0	400	1			0
322	0	322	1			1

Figure 3.14: Computation and decisions made by the computation unit. It stores intermediate values until it is told to update the neuron state. When updating, it decides if the neuron spikes or not and in which conditions.

The spikes produced by the computation unit naturally code AER events. The origin ID is given by the computation unit to the network interface which sends as much flits as there are target clusters for this given neuron. As seen before the timing information is preserved.

3.4.2 Scaling up

The main constraint using one cluster is to maintain its frequency high enough so that the processing workload can be processed during the intended time of the simulation step. The maximal target frequency that would be needed is defined by the number of allowed synap-

tic stimuli to be processed and the time corresponding to a simulation tick. Using n neurons with s synapses and time steps of duration T, the maximum target frequency for the synchronous processing part has to be over s/T.

As we limit the number of neurons per cluster, we need to use multiple clusters in order to simulate larger operators. Each cluster must respond to the same constraints plus we have to carefully synchronize simulation steps in between clusters and take into account communication overhead. Fig.3.15 shows example of loosely constrained cluster synchronization. Each cluster has a TP information which is copied in the spikes sent to the ANOC. Scenario (1) is intended as the spikes are sent and received during the same simulation step, thus generating no error. Scenario (2) is taken care of using the TP bit in the 1-Filt AER event conveyed by the ANOC. It generates priority processing at the target cluster. However, loosely synchronized clusters could also generate scenario (3). In this scenario, with current implementation, the receiving side will receive a spike with different TP bit and understand it was sent during the previous simulation step leading to a difference of 2 simulation steps in the timing value carried by the spike.

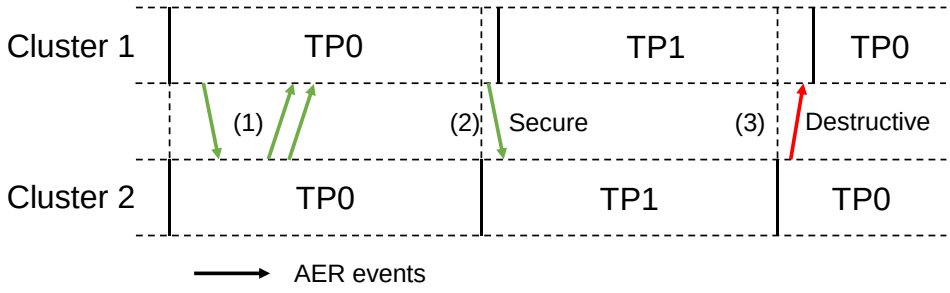


Figure 3.15: Synapse matrix enables high connectivity.

We thus use a synchronization bus, named “rendezvous” which is essentially performing the simulation step synchronization in between each cluster and the input set conditions. The rendezvous module is designed to wait for each cluster to finish processing the current step and the outside condition to be met. Waiting for each cluster to finish processing avoids scenario (3) from Fig.3.15, while waiting for the outside condition to be met let us satisfy simulation step duration coherence with the input spike stream.

Monitoring the reception FIFOs for a given simulation step, we know the exact amount of cycles that will have to be spent in the synchronous process. As the target hardware time per simulation step is fixed and linked to the input spike stream, we can setup local DVFS [12]. The main consequence will be reducing the power consumption of the chips by flattening the activity in each cluster for their respective processing time to be equal to the simulation length. The second consequence is flattening the generated spike communication at system level as they will be generated homogeneously through the simulation step.

We defined the main parts of our system. Sizing and designing an ASIC responding

to specific needs using specific topologies is not straightforward just based on those modules. We thus implement a SystemC model of our architecture in order to retrieve the main usage figures and allow quick application to hardware characterization.

3.5 SystemC model

SystemC [13] is a set of C++ classes and macros, providing to the designer an event-driven simulation environment. It enables the designer to simulate multiple concurrent processes in C++ and offers bases for system development. It is quite flexible and can under certain circumstances be compared to VHDL or Verilog as an hardware description language while still being able to be used at a system-level. It is used for a wide variety of applications, including architectural exploration, performance modeling, functional verification, and high-level synthesis.

The designed SystemC model can be used on multiple parts of the design flow. First, once a defined set of efficient topologies is designed for an application, we are able to simulate those topologies on the generic hardware based on the previously explained implementation. Simulating those topologies for cluster parameters gives optimal sizing information for target application. Later, simulated or measured hardware characteristics can be inserted in the model in order to perform accurate power estimations based on the cluster characteristics.

The different parts of our system do not require the same accuracies in terms of modeling. We will mainly use a Transaction-Level Modeling (TLM) for the asynchronous part and a Cycle-Accurate, Bit-Accurate (CABA) one for the synchronous parts.

3.5.1 TLM ANOC

TLM [14][15] is a high-level approach to modeling digital systems. It abstracts the communication among modules and separates it from the details of the implementation of each units or of the communication architecture. The transaction level focuses more on what is being transferred and in-between which points, than on how it is transferred and its implementation. The usual communication mechanisms such as FIFOs or buses are replaced by channels and presented to resources using the SystemC interface classes.

An Asynchronous NoC was implemented using TLM 2.0 [16]. This model was simplified for our use. Data packets were converted to perform AER event communication which was described in 3.2.1. The only parameters that have to be set in this implementation are the number of neurons used and clusters in order to determine the number of bits

composing the data being transferred. We recall here that the "data" part in an AER event is composed of the origin address and the time at which the event was created. As said in previous sections, timing is kept using a tick parity bit.

We use the previously defined 1-Flit packet which represents the AER event. The origin address is maintained in two parts. The origin neuron ID is the data part of the Flit. The cluster IF is retrieved from the circular shifting of the PTT part.

The list of parameters that have to be defined for the TLM ANOC is the following:

- The number of neurons per cluster, determining the size of the data to be sent.
- The path size limitation. For a PTT of size $2l$, the maximum number of target clusters is defined by the number of clusters the flit can reach in $l-1$ steps. There is a compromise to be found here between the number of possible target neurons and size of the PTT to be conveyed and shifted.
- The router delay, determining the communication latency.

The following metrics can be retrieved for each ANOC router:

- Average and peak number of spikes conveyed per simulation steps.
- Average latency generated.

As the frequency of each cluster will ultimately be controlled by the local DVFS targeting maximum activity rates during operation, the rendezvous bus is also implemented in asynchronous logic. Once a flit is received in a cluster, it is processed and stored consuming a delay proportional to the number of targets to be stimulated in this cluster.

3.5.2 CABA synchronous parts

Modeling using cycle accurate and bit accurate behavior is an essential step between the high level algorithms, defining the way neurons have to be updated and their mechanisms, and the implementation in custom silicon devices, whether it is FPGA or ASIC [17]. It offers hardware aware simulation possibilities that are not always present when performing network simulations using C++ or Matlab.

In order to confidently study the effects of different topologies, the fine grain implementation of the synchronous processing parts using the SystemC environment and accurate behaviors is necessary. This approach allows us to perform functional verification for

the implemented modules and the topologies they support. It also allows us to have a precise idea of the bottleneck of our architecture and the usage of each part of the design.

Each previously described module is modeled at the cycle level with intended behavior. Each cluster updates its synchronous resources with its respective clock in order to enable local DVFS possibilities.

The parameter used for the cluster resources are the following:

- Main clock period, defining the clock period needed for proper function at any input activity.
- Precision needed for the synaptic weights used in the supported topologies.
- Precision needed for the neuron variables.
- The number of available neurons and synapses per cluster.
- The number of allowed input and number of allowed targets per input.
- The number of synaptic stimuli slot per FIFO.
- The number of dynamically allocable FIFOs and their number of slots.
- The maximum number of delays available.

Once parameters, topologies and inputs are provided to the model, the neurons are placed onto the hardware. The neuron placement can be done in multiple ways in order to try and minimize ANOC communication or flatten cluster activity while always respecting the set limitations (number of neurons, synapses and targets per cluster). One can for instance use a link-aware simulated annealing neuron placement algorithm, trying to minimize ANOC communication or use an activity-aware algorithm based on the XNet mean activity in order to minimize activity heterogeneity in between clusters.

3.5.3 Monitored execution

We first monitor the execution for proper execution of the supported SNN topologies. Multiple levels of debug are available at each module for assessing unwanted behaviors.

The second important part to be retrieved from monitoring the usage of each module is the hardware usage. This allows us to design optimal reception capacities for any given application. For a given application, target precision and set of topologies responding to the defined need, we can retrieve here the optimal number of slots per FIFO and number of dynamically allocable FIFOs in order to have minimum energy consumption.

Each resource outputs its metrics at the end of the simulation, composed of the following figures:

- Total time, simulation steps and time per simulation steps spent.
- Average and peak active and idle cycles used per simulation step.
- Average and peak activity percentage.
- Total number of spikes sent and received.
- Total number of flits sent.
- Peak and average use per FIFO in the synaptic stimuli storage module.
- Peak and average usage per Dynamically allocable FIFO.
- Activity per neuron ID.
- Average and peak number of individual neuron's variable read and write.

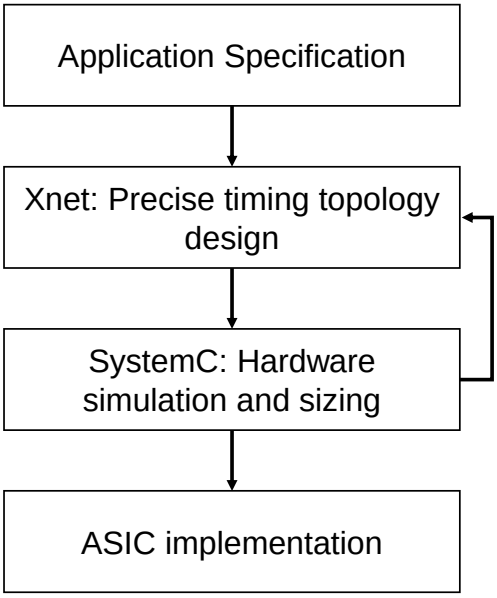


Figure 3.16: Design flow using XNet to design topology answering the application needs and the SystemC model to estimate and size hardware.

Those metrics allow the designer to estimate hardware usage for a given set of parameters and later estimate power consumption. This model integrates itself in the design flow as shown in Fig.3.16. The following section will depict the application of this model to basic Linear Combination networks for sizing the generic architecture.

3.5.4 Example with Linear combinations

We use in this section the topology defined in Fig.3.10 with our hardware model. We use 200 operators of 8 inputs for 8 bit operations. This necessitates 1800 neurons if used with no delay limitation, 2800 if we limit delays to 1 elementary simulation step and use dummy neurons to simulate longer delays. We split those neurons randomly onto 9 clusters each simulating 512 neurons and 4096 synapses, thus allowing an average of 8 input connection per neuron.

First, we can analyze ANOC usage to verify its proper function. We set the time steps to 100µs to represent sparse inputs and monitor the flits sent per router at each simulation time to determine ANOC capacities needed. Fig.3.17 shows average and peak ANOC usages. As we use a $3x3$ mesh, we can determine the average path length and thus the needed average and maximal throughput which would be here respectively 202 Mb/s and 418 Mb/s using 21b (9b ID and 10b PTT) 1-Flit packets. However, this is only an example and placing neurons randomly throughout the architecture is not efficient communication-wise. More effective placement strategies will be explored when needed in the chapter 5.

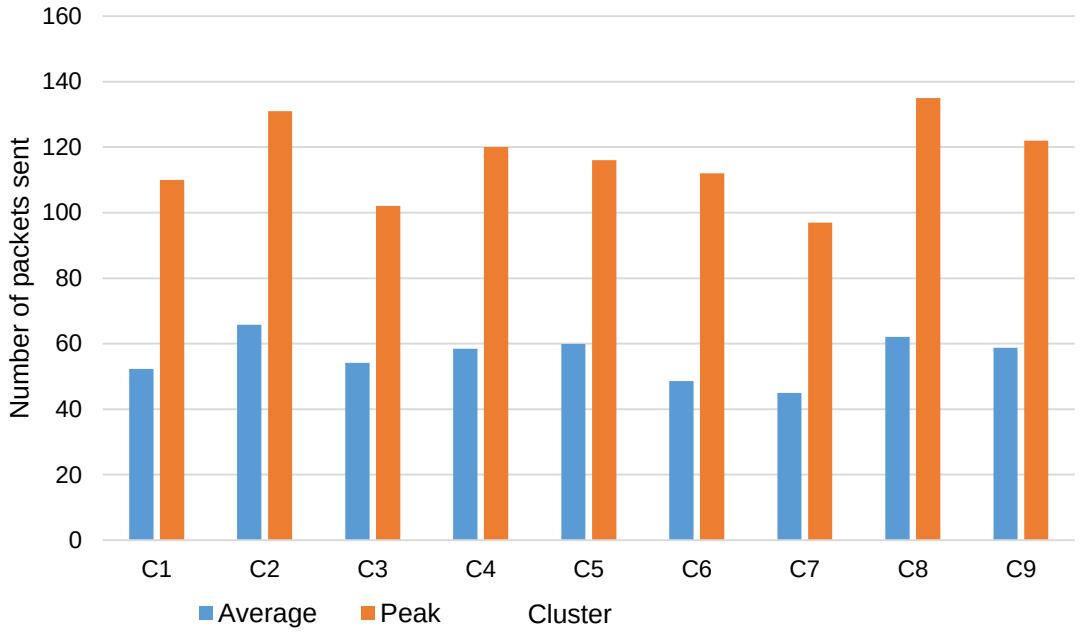


Figure 3.17: ANOC average and peak load, influence per cluster.

The next part of this example will help us size the Hardware Proof of Concept presented in chapter 4. We here explore the sizing of each Cluster for supporting arrays of Linear Combinations. We can explore two main factors, the first one being the number of available synaptic delays for our synapses.

Decreasing the maximum available delay to 1 increases the number of neurons per operator from 9 to 14 but also decreases by 33% the size needed for the synaptic stimuli storage. When comparing the generated activity, the topologies using synaptic delays going

up to 2 simulation steps are using 29% less spikes. The dummy neurons used to reduce hardware requirements are placed on channels conveying more spikes and thus the activity is more important. The use of dummy neurons to simulate longer synaptic delays creates a neuron with only 1 input and 1 output, meaning it lowers the number of synapses needed per neuron at the expense of having more neurons per cluster. The Proof of Concept designed in chapter 4 will be using up to 2 synaptic delays with the possibility to limit to 1 for further comparison.

We use 4096 synapses for 512 neurons in order to allow 8 average synapses per neuron. We now size the FIFOs used in the synaptic stimuli storage module and the number of dynamic FIFOs to be used. Fig.3.18 shows the distribution of synaptic stimuli slots used per active accumulator neuron per simulation step. Neurons belonging to operators in the first layer, interfacing directly with the AER events from the sensor, are not subject to spike synchronization. For 8 inputs, the number of synaptic stimuli to be integrated during a simulation step rarely crosses 4 stimuli. One of the issue here is the synchronization performed per operators for the output spikes. Neurons belonging to operators further in the network will for the same number of inputs receive 8 synchronized inputs and thus need 8 FIFO slots. The peak number of stimuli to be integrated per neuron is defined per the number of synapses linked to it which is linked to the number of inputs used and was defined in table 3.1.

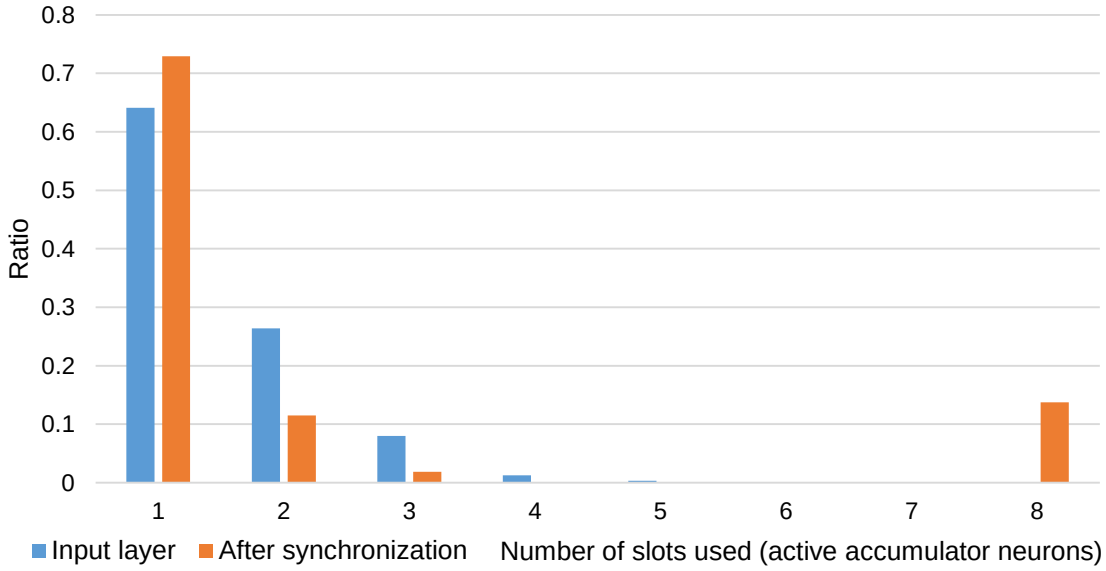


Figure 3.18: FIFO slots used for active accumulator neurons for 8 inputs, at network input vs after synchronization.

In order to reduce the area and consumption of the synaptic stimuli storage module, the number of slots in the regular FIFOs is set to 4 and we allocate 32 dynamically allocable FIFOs of 4 slots in order to allow Accumulator and Synchronization neurons to punctually have more than 4 inputs at a time. The next chapter will focus on the Hardware implementation and testing of a Proof of Concept using sizing results shown here.

3.6 Références

[1] V. Chan, S. C. Liu, and A. van Schaik. Aer ear: A matched silicon cochlea pair with address event representation interface. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 54(1):48–59, Jan 2007. 64

[2] J. Anumula, D. Neil, T. Delbruck, and S.-C. Liu. Feature representations for neuromorphic audio spike streams. *Frontiers in Neuroscience*, 12:23, 2018. 64

[3] C. Posch et al. A qvga 143 db dynamic range frame-free pwm image sensor with lossless pixel-level video compression and time-domain cds. *JSSC*, 2011. 64

[4] Edith Beigne, Fabien Clermidy, Helene Lhermet, Sylvain Miermont, Yvain Thonnart, Xuan-Tu Tran, Alexandre Valentian, Didier Varreau, Pascal Vivet, Xavier Popon, and Hugo Lebreton. An asynchronous power aware and adaptive noc based circuit. 44:1167 – 1177, 05 2009. 64

[5] Pascal Vivet, Yvain Thonnart, Romain Lemaire, Cristiano Santos, Edith Beigne, Christian Bernard, Florian Darve, Didier Lattard, Ivan Miro-Panades, Denis Dutoit, Fabien Clermidy, S Cheramy, Abbas Sheibanyrad, Frédéric Pétrot, Eric Flamand, Jean Michailos, Alexandre Arriordaz, Lee Wang, and Jürgen Schlöffel. A 4 x 4 x 2 homogeneous scalable 3d network-on-chip circuit with 326 mflit/s 0.66 pj/b robust and fault tolerant asynchronous 3d links. 52, 10 2016. 64

[6] P. Merolla, J. Arthur, F. Akopyan, N. Imam, R. Manohar, and D. S. Modha. A digital neurosynaptic core using embedded crossbar memory with 45pj per spike in 45nm. In *2011 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–4, Sept 2011. 67

[7] F. Akopyan et al. Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(10):1537–1557, Oct 2015. 67

[8] J. Schemmel et al. A wafer-scale neuromorphic hardware system for large-scale neural modeling. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pages 1947–1950, May 2010. 67

[9] G. W. Burr, R. M. Shelby, C. di Nolfo, J. W. Jang, R. S. Shenoy, P. Narayanan, K. Virwani, E. U. Giacometti, B. Kurdi, and H. Hwang. Experimental demonstration and tolerancing of a large-scale neural network (165,000 synapses), using phase-change memory as the synaptic weight element. In *2014 IEEE International Electron Devices Meeting*, pages 29.5.1–29.5.4, Dec 2014. 67

[10] Jennifer Hasler and Harry Marr. Finding a roadmap to achieve large neuromorphic hardware systems. *Frontiers in Neuroscience*, 7:118, 2013. 67

[11] Dynap-sel – a mixed-signal multi-core spiking chip for models of cortical computation implemented in advanced st 28nm fdsoi process. http://www.neuram3.eu/achievements/neuram3-chips/dynapsel. Accessed: 2018-10-01. 68

[12] P. Vivet, E. Beigné, H. Lebreton, and N.-E. Zergainoh. On Line Power Optimization of Data Flow Multi-core Architecture Based on Vdd-Hopping for Local DVFS. In *20th IEEE International Workshop d Timing Modeling, Optimization and Simulation (PATMOS'10)*, Grenoble, France, September 2010. ACM IEEE. 81

[13] Preeti Ranjan Panda. Systemc: A modeling platform supporting multiple design abstractions. In *Proceedings of the 14th International Symposium on Systems Synthesis*, ISSS '01, pages 75–80, New York, NY, USA, 2001. ACM. 82

[14] Alain Clouard, Kshitiz Jain, Frank Ghenassia, Laurent Maillet-Contoz, and Jean-Philippe Strassen. *Using Transactional Level Models in a SoC Design Flow*, pages 29–63. Springer US, Boston, MA, 2003. 82

[15] Adam Donlin. Transaction level modeling: Flows and use models. In *Proceedings of the 2Nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '04, pages 75–80, New York, NY, USA, 2004. ACM. 82

[16] C. Koch-Hofer. *Modeling, Validation and Presynthesis of Asynchronous Circuits in SystemC*. Theses, Institut National Polytechnique de Grenoble - INPG, March 2009. ISBN: 978-2-84813-131-3. 82

[17] Andy D. Pimentel and Stamatis Vassiliadis, editors. *Computer Systems: Architectures, Modeling, and Simulation, Third and Fourth International Workshops, SAMOS 2003 and SAMOS 2004, Samos, Greece, July 21-23, 2003 and July 19-21, 2004, Proceedings*, volume 3133 of *Lecture Notes in Computer Science*. Springer, 2004. 83

Chapter 4

Proof of Concept: INSPIRE

Sommaire

4.1 Introduction

We explored in chapter 3 the architectural solutions available for supporting the SNN topologies obtained in chapter 2. Using the hardware model, we implement a Proof Of Concept (POC) in Fully Depleted Silicon On Insulator (FDSOI) 28nm composed of 1 fully synchronous cluster integrating 512 neurons and 4096 synapses. The POC is implemented to support the explored Linear Combination topologies that represent the greatest part of DSP and ANN computations. The first part of this chapter focuses on the POC hardware implementation and its differences compared to the cluster to be integrated in a GALS system. The second part focuses on the characterization of the fabricated chip and comparison to SOA chips for the main figures of merit.

4.2 Implementation

The functions of the main pipeline of the architecture developed in chapter 3 will be first shortly summarized. As we receive asynchronous spikes during our hardware simulation steps, we want to be able to store the generated stimulus for an arbitrary amount of time depending on the synaptic delay. Once the number of time steps corresponding to the synaptic delay is elapsed, we want to retrieve the stimulus and integrate it in the core level of the neuron according to its type. The retrieving of stimulus is ordered by target ID not to use multiple read and write operations for the same neuron variables during a simulation step.

Once the stimuli arrive at the computation unit, they are integrated and lead to different mechanisms depending on the type of the synapses used. At the end of the stimuli for a neuron, an update is performed deciding if the neuron spikes or not and what to write back in memory. The output spikes are then sent to the ANOC and looped back for targeting the neuron in the same cluster.

As our implementation is fully synchronous here and uses only 1 cluster, the previously explored architecture was modified for simplicity purposes. The Proof of Concept is fully synchronous implying the Network Interface, Controller and Scheduler modules were modified. As it does not provide the support for AER events, those were simulated using input FIFOs loaded at the configuration step and output FIFOs for retrieving output spikes. Fig.4.1 shows the micro-architecture of the Proof of Concept. The main modifications with respect to the previous chapter are highlighted in blue. The implementation and difference of the modules shown here will be discussed on the following sections, starting with the computing unit.

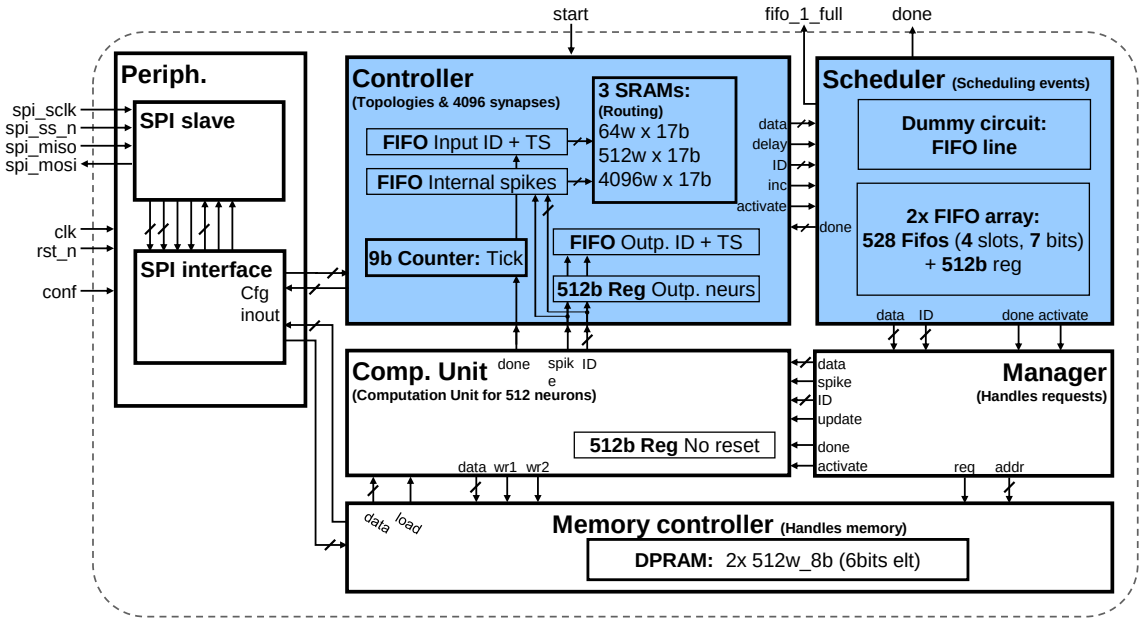


Figure 4.1: Micro architecture of INSPIRE.

4.2.1 Synchronous computation

The modules responsible for the synchronous computation were intended to be used in a synchronous manner. This is composed of 3 main parts: the manager making memory request, the memory itself and the computation unit. The computation unit includes a Finite State Machine (FSM) handling outputs sent to the different neighbor modules. The FSM can be described as follows:

- Receive the stimulus characteristics (type, weight, update order, target).
- Receive the neuron's variables (neuron core potential and linear synapse level).
- Compute intermediate results: adds punctual integration weights to the core potential and linear integration weights to the linear synapse level.

The manager module making the memory requests monitors the target ID and process completion. When the target ID changes or if the synchronous process is complete for this simulation step (no more stimuli to be retrieved), then the manager sends an update request to the computation unit for the current ID inducing the following steps:

- Compute the updated neuron core potential from the previous one and the linear synapse level and compare to the threshold potential.
- If superior, emit a spike and check if the neuron has to reset when spiking or not. If it resets, store rest potential and rest linear synapse level. Otherwise, store the updated

neuron core potential minus the threshold potential and the updated linear synapse level.

- If the stored linear synapse level is not null, then the neuron being processed will have to be updated at later simulation steps in order to perform the linear integration. In this case, the computation unit notifies the Scheduler which activates the "To be updated" register for the current neuron ID.

The output spikes are sent with the current ID to the Controller. The implementation of the Controller slightly differs from the previous chapter and will be depicted in the next section.

4.2.2 Controller

The Controller module earns more roles in this implementation. First, it converts the addresses to synapse groups to be stimulated. It includes a FSM monitoring the ID from the internal events generated by the computation unit and the input events generated at configuration time. Both types of events are stored in FIFOs, input events having time stamps associated to know when to use them. When those events arrive, the origin ID is used with the corresponding SRAM (Input or Internal) depending on where the spike comes from. From this memory is retrieved the address of the first synapse to be stimulated and the number of synapses directly after the first one in memory which will also be used. The FSM then handles memory calls to the 3rd SRAM which outputs the synapses characteristics: type, weight, delay and target ID. Fig. 4.2 summarizes this first role with the different possible scenarios.

Secondly, this module handles the simulation tick changes. The input from the rendezvous bus described in section 3.4.2 corresponds to the "start" input from Fig.4.1. Its role in handling the simulation flow is to wait for the current tick to be finished and for the start signal to be activated. This allows us to have 2 modes of operation: either toggle the start signal to control the simulation tick duration or activate it to create a new tick as soon as the previous one is over.

Finally, we use FIFOs to store input and output events as the chip is not handling AER protocol. These FIFOs can be read by SPI and store the origin ID and tick count of emission. The input events are stored in input time-stamp order at configuration time to be able to retrieve them with the controller when they need to be converted into synaptic stimuli. For this POC, the Controller thus needs a tick counter to know when to introduce input spikes in the SNN. This tick counter is also used for writing time-stamp of emission associated with the ID of spiking output neurons in the output FIFO.

The synaptic stimuli retrieved by the Controller are sent to the Scheduler that will

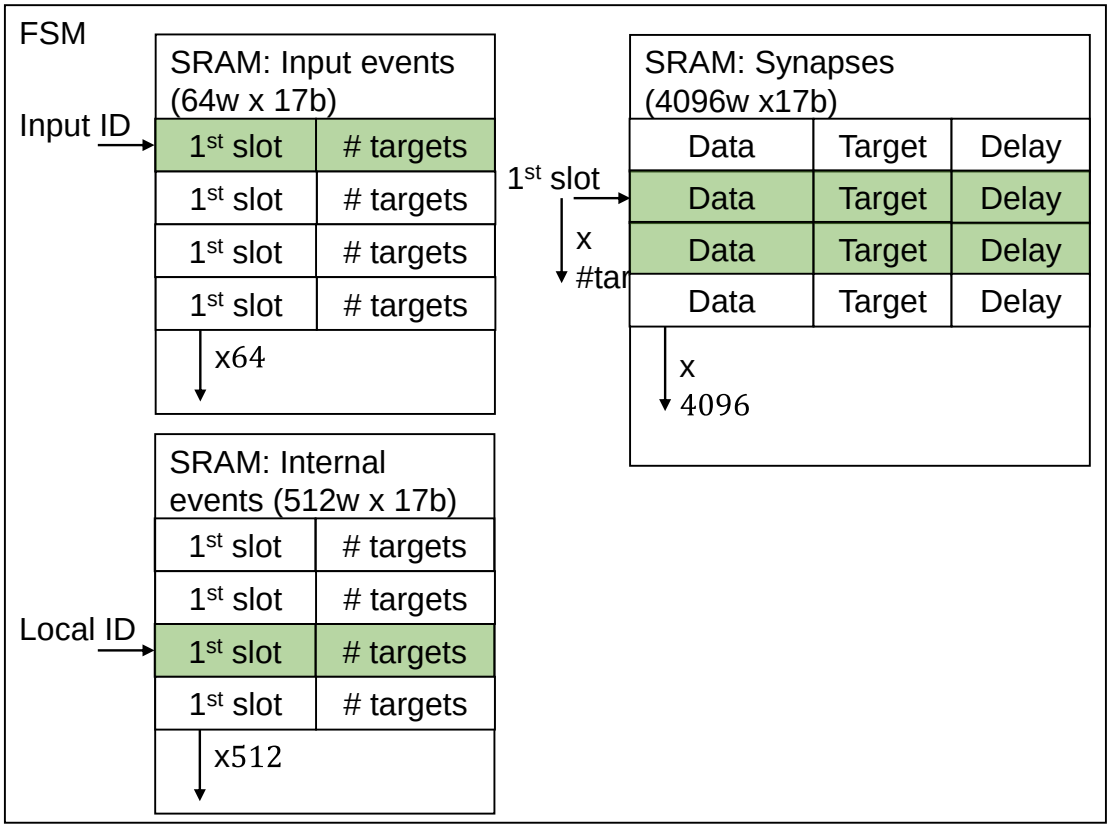


Figure 4.2: FSM handling the 3 SRAMs for ID to synapses conversion. Monitors the Input FIFO and Internal spike FIFO for activity. Retrieves the set of synapses using the corresponding first slot and number of targets to be stimulated.

take care of storing the stimuli until they have to be retrieved ordered by target ID.

4.2.3 Scheduler - Synaptic Stimuli Storage

The Scheduler is designed to synchronously receive synapse characteristics, store the useful information in the correct FIFO and retrieve the stored information when the stimulus has to be processed. As specified at the end of the previous chapter, for the considered topologies, we use in this module 512 FIFOs of 4 slots each plus 32 dynamically allocable FIFOs of 4 slots each per FIFO array. This allows every neurons of the network to receive and integrate its average input activity at each simulation step and also allows peak number of input events for a subset of neurons.

The chip integrates 3 arrays with such arrangement. At each simulation step, data from one array will be used to stimulate the neurons while the 2 other arrays will be filled with events to be processed at time+1 and time+2. When the step changes, the roles of the arrays are shifted in a circular manner. Time+1 becomes the one to be used for stimulation, time+2 becomes time+1 and the previous array used for stimulation becomes time+2.

This allows the synapses to have delays going up to 2 simulation steps. One of the array can also be ignored in order to limit the synaptic delay to 1 simulation step and experiments done with such technique will deduct the power consumption of the dummy array. Fig. 4.3 summarizes the synchronous storing process. Each array stores the events for a simulation step. As the first array is currently being processed, the second one stores event having a synaptic delay of 1 simulation step and the third array stores the events having a synaptic delay of 2 simulation steps. The target address gives the address of the FIFO used to store the synaptic weight and type. If the corresponding FIFO is full, a dynamically allocable FIFO is used in order to extend the reception capacities for the target neuron. The first unused dynamically allocable FIFO gets the ID if the current FIFO is full: the weight and type are stored in it.

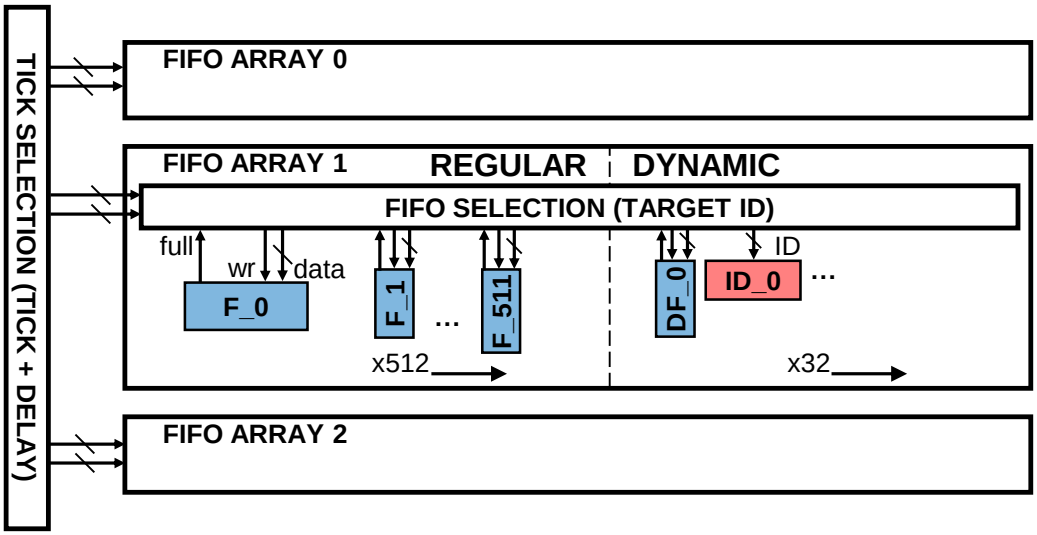


Figure 4.3: Synchronous storing of synaptic stimuli using the synaptic delay to determine in which array and the target ID to determine in which FIFO. F and DF are respectively the regular FIFOs and Dynamically allocable FIFOs (associated with a target ID register).

Fig. 4.4 shows the same configuration a simulation step later. The array that was previously storing events for the next simulation step is now being processed in order. The order is defined by the Priority encoder, going through the following steps:

- Determine the highest ID of the non empty FIFO.
- Determine the highest dynamic ID of the non empty dynamic FIFO.
- Determine the highest ID of the neuron that has to be updated (having active linear synapses).
- Retrieve the stimulus from the highest ID of both regular and dyn. FIFOs.

The synapse weight, type and target ID are then sent to the manager handling the memory calls. If a scheduled update and an input stimuli are found for the same target ID,

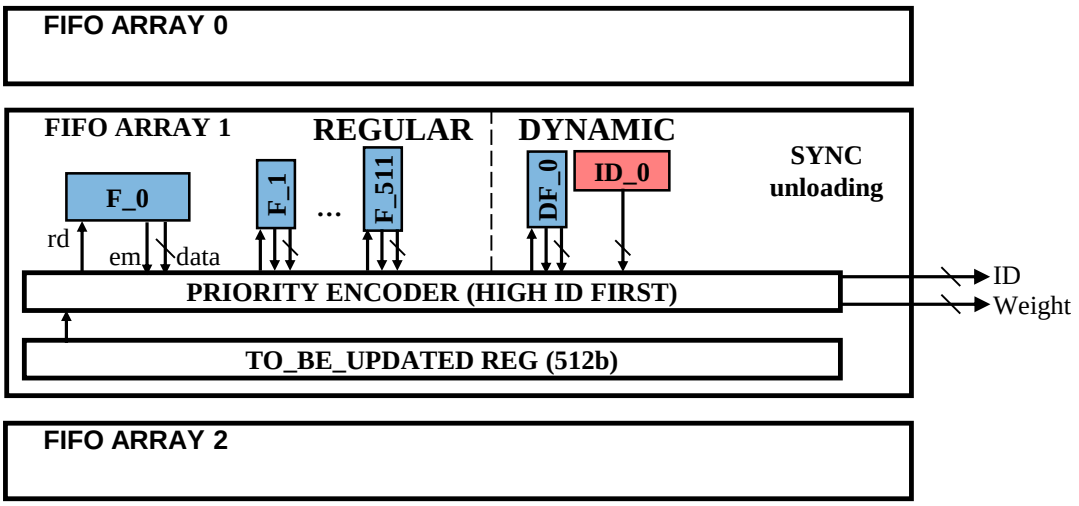


Figure 4.4: Synchronous retrieving of the synaptic stimuli when the simulation step is changed. Starts with the highest ID and sends in order stimuli to the Manager. The "To Be Updated" register represents neurons that have to be updated at this simulation tick: either neurons receiving stimuli or neurons having active linear synapse(s).

both operations are concatenated sending only the stimulus to the manager. In this implementation, the amount of cycles spent processing each simulation step is known when the simulation step starts. However, no strategy was implemented in order to perform local DVFS accordingly. The synchronous processing starts with the priority encoder unloading stimuli from the FIFOs. The synchronous processing ends with the retrieving of the last stimulus from the current array of FIFOs, cycle during which the Scheduler notifies the Controller the stimuli were all sent.

4.2.4 Configuration and inputs

The configuration of the chip is done using Serial Peripheral Interface (SPI), a serial communication protocol. The SPI slave module handling the communication on the chip side is linked to a configuration module responsible for handling read and write operations at the different addresses. The following elements can be accessed via SPI, as illustrated in Fig.4.1:

- Controller module: 2 SRAMs for converting the input and internal addresses to synapses ranges, 1 SRAM for the synapses themselves, a 1-hot register denoting if the corresponding neuron has to send its spikes to the output of the cluster, 2 FIFOs for input and output events and the simulation tick counter.
- Neurons' memory: 2 DPRAMs storing the neurons' core potential and linear synapses values.
- Computation unit: a 1-hot register denoting if the corresponding neuron has to reset

when emitting a spike or not.

The SPI requests for the chip configurations are generated using the SystemC model. We translate the neuron and synapse placement that was performed for the model into SPI requests that will configure the topology simulated in the model onto the chip. This chip is not designed to be interfacing with Dynamic Sensors as it does not support asynchronous communications scheme. The timing and origin address of the input events is loaded at configuration time. These events are stored in a FIFO to be used at simulation time.

4.2.5 Design

In order to implement our Proof of Concept, we use the digital synchronous conception flow. Table 4.1 shows the different tools used at each step of the process. The chip was implemented using the LVT FDSOI 28nm technology provided by ST Microelectronics.

Tool	Step
Questasim 10.5c	Behavioral, RTL, postbackend simulation
DC compiler K-2015.06-SP2	Synthesis
Innovus 15.24	Place and route
Virtuoso	I/O Ring, checks, top level simulation

Table 4.1: EDA tools.

The design is split in between two Power Domain (PD) for power consumption measurements. The first PD corresponds to the modules that were intended to be used synchronously: from the Priority encoder to the output of the computation unit, including the DPRAMs for the neurons' variables. The second PD corresponds to the Controller and Scheduler modules that will ultimately be asynchronous. The chip is composed of 235k cells and 85.56kb of memory for a silicon area of 0.8mm^2. Fig.4.5 shows the layout of the chip and the area used by the different modules in this implementation. The greatest module area-wise is the Scheduler as the FIFO based implementation is less compact than a set of SRAMs for the same memory footprint.

The chip was designed and simulated to reach 75 MHz in nominal conditions @0.9VDD @0VBB @25C. When used at 75 MHz the chip is capable to simulate more than 18k simulation steps per second at its maximal workload. Verification steps were performed with the topologies explored in the previous chapters at each step of the design.

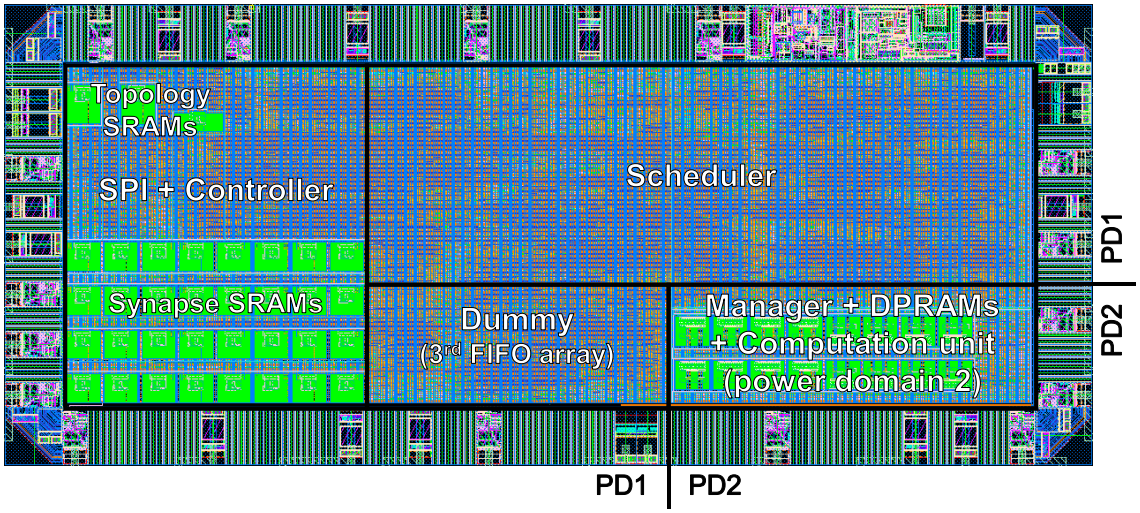
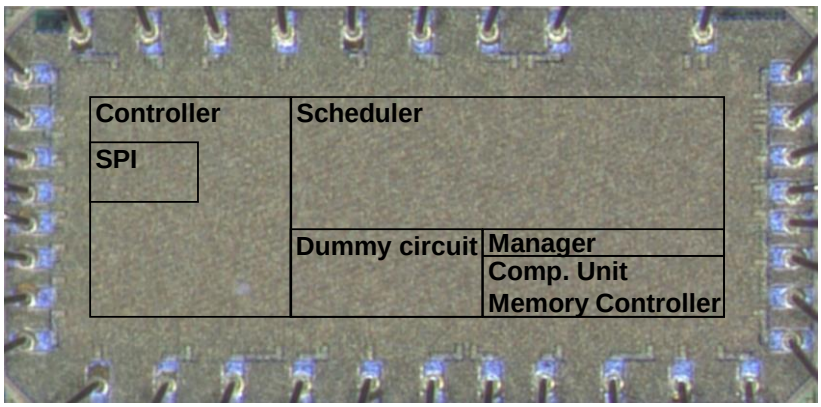


Figure 4.5: Layout of the POC INSPIRE.

4.3 Test

The chip, named INSPIRE, has been fabricated in LVT FDSOI 28nm technology and operates from 0.45 to 1V with a Back Bias from -0.3 to 1.3V (Fig.4.6). It was packaged in QFN 28 integrated onto PCB. Fig.4.7 shows the setup for testing the chip.



Techno	FDSOI 28nm LVT
Neurons	512
Synapses	4096
Area	0,8mm²
VDD range	0,45 – 1,2 V
VBB range	-0,3 – 1,3 V
Cells	235K

Figure 4.6: Chip characteristics.

4.3.1 Topologies

The chip is measured while using n-input 8b signed Linear Combination topologies each corresponding to n MAC operations. These operations were chosen as they represent the majority of the computation performed in conventional applications and ANN inference. Fig. 4.8 recalls the type of networks used for linear combinations. Those operators will also be used with different parameters and inputs while still staying within the possibilities of the chip (6b signed weights, 8b signed core and linear synapse levels) exploring the energy used per operation.

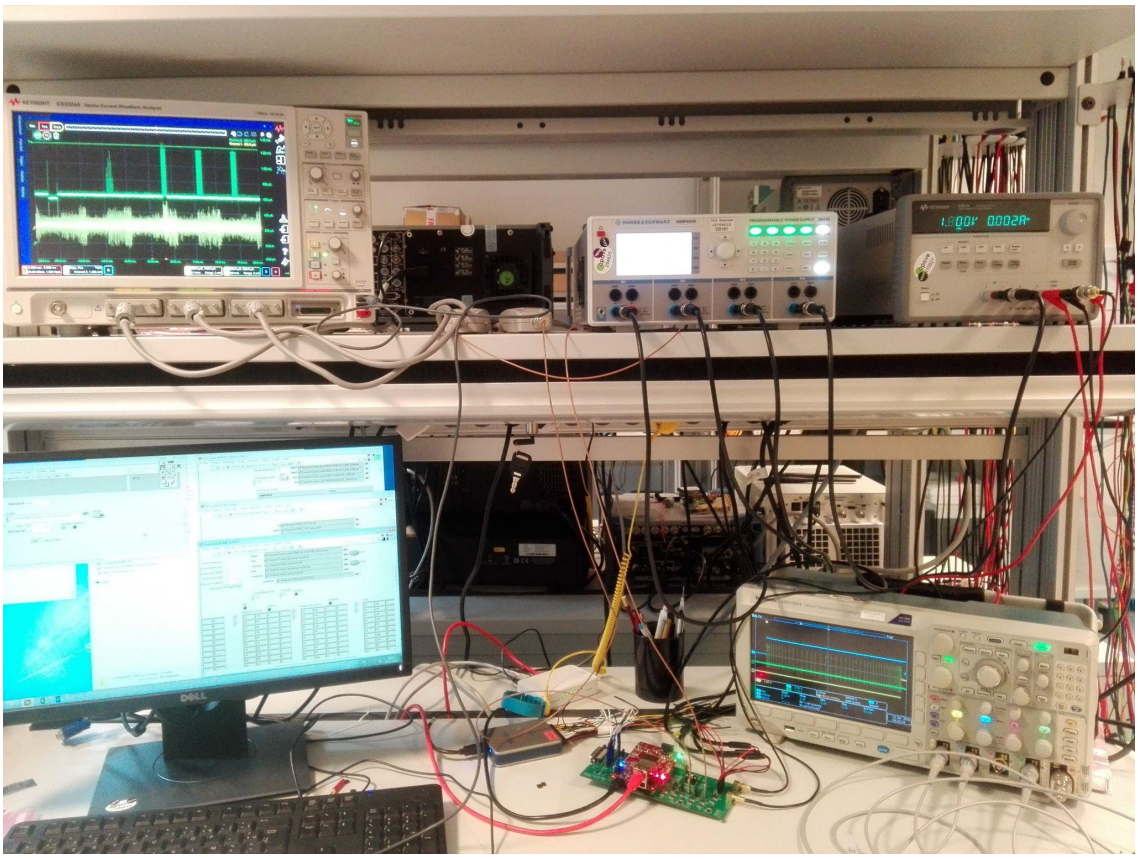


Figure 4.7: Test setup for INSPIRE.

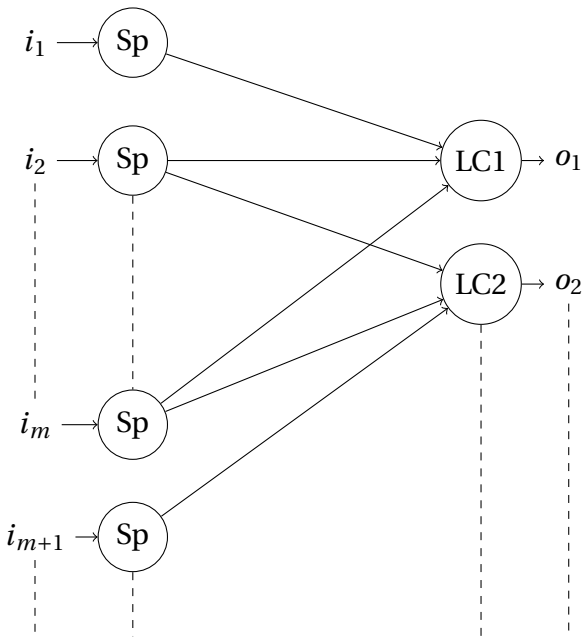


Figure 4.8: Array of Linear combination used for testing the INSPIRE chip.

4.3.2 Power analyzes

During the tests, the tick length is set to 36.8 µs corresponding to an emulated sparse activity of 66 k events/s using the 128 inputs available, within realistic spiking sensor rate range. As mentionned before, the tick length can either be controlled manually or via the Controller ranging up to 4100 clock cycles. As shown in Fig. 4.9, the circuit's maximum frequency is 60MHz@1V and 4MHz@0.45V.

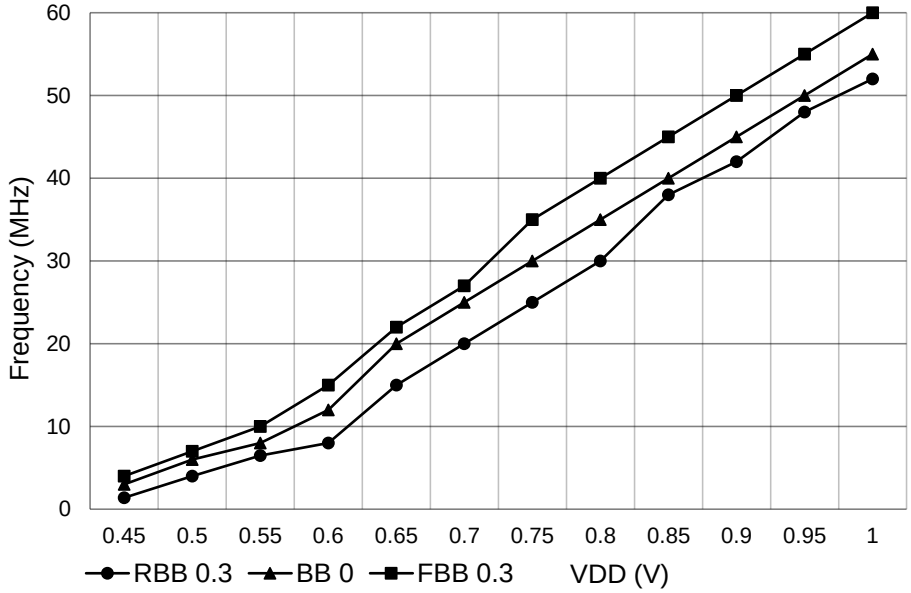


Figure 4.9: Maximum frequency versus VDD @(-0.3;0;0.3) VBB.

The energy per synaptic event is the main metric in spiking neurons architecture and corresponds to the energy needed to integrate an incoming spike and is measured during computation from the Controller to the computation unit in one cycle. As shown in Fig.4.10, the minimum energy is 29pJ/cycle @0.5VDD @-0.3VBB corresponding to 26.4pJ @0.5VDD @-0.3VBB per synaptic event. These results were obtained for topologies having a maximum synaptic delay of 1: only 2 of the 4 arrays of the scheduler was used and thus its power consumption was deduced of the total.

In these conditions, it exhibits 115µW (42.8µW leakage), 92% of the power is consumed by the Controller storing topologies and Scheduler storing synaptic stimuli. Consumption being dominated by Synapses storage and utilization is common among neuromorphic architectures. In order to perform a MAC 8bits, 127 synaptic events are required, corresponding to 3.3 nJ/op in our circuit.

Fig. 4.11 shows the power consumption breakdown per PD in function of the frequency @-0.3VBB. The power consumption of the computation unit and neurons' variable memory represents less than 13% of the total consumption of the chip. The dynamic power generated by the Controller and Scheduler represents the majority of the power consump-

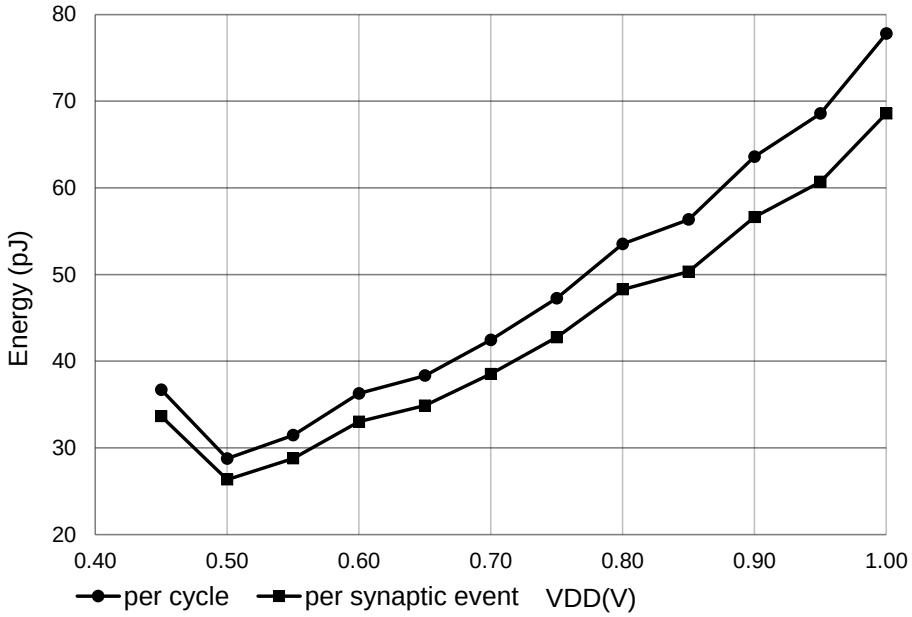


Figure 4.10: Energy per cycle and per synaptic events versus VDD @-0.3VBB.

tion of the chip with 55 to 83% of the total.

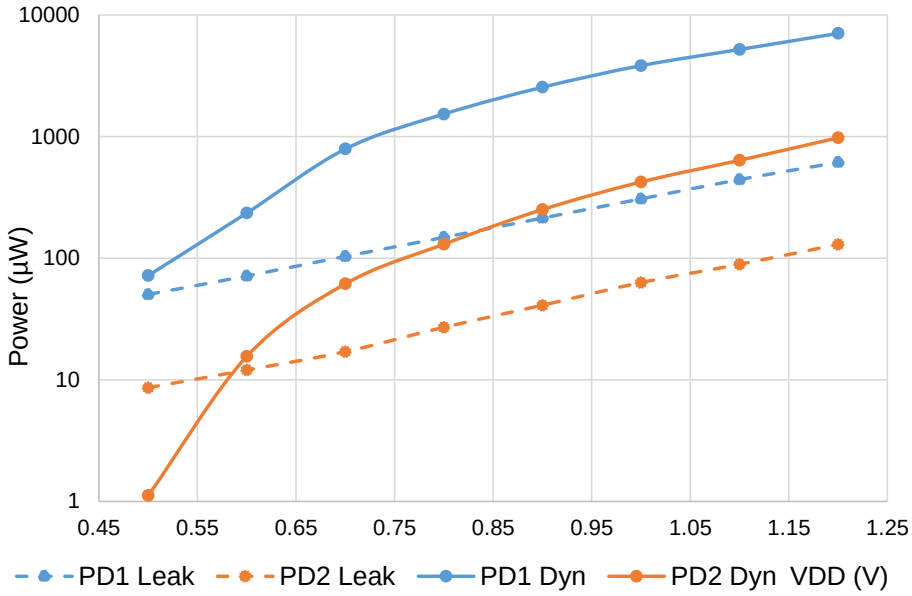


Figure 4.11: Power breakdown @-0.3VBB. The Power Domain 1 (PD1) (SPI + Controller + Scheduler) dominates the power consumption of the chip.

Contrary to usual DSP/GPU cores, our approach is, by nature, approximate as shown in Fig.4.12 where energy is scaled with respect to MAC accuracy. As the need for precision decreases, the tick length can be increased in order to keep operating on the same events but at lower accuracy. Using modular MAC operators, we prove in Fig.4.12 that the energy used by our approach scales with accuracy needs. This approach is thus very suitable to flexible spiking sensors interfaces as it can adapt to different specifications.

Even if no Spiking Neural Networks architectures are comparable to our approach

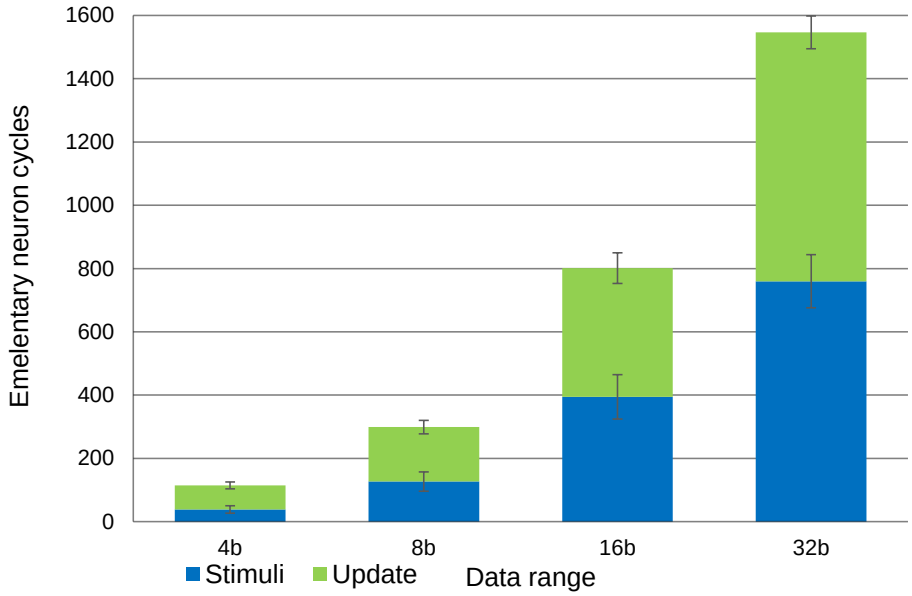


Figure 4.12: Computation unit cycles used and synaptic energy per operation versus accuracy.

(sized for usual Neural Network applications), we can fairly compare energy per synaptic events as shown in Tab. 4.2. Our proposed variable tick length fits sparse activity generated by our topologies, leading to power savings in energy per tick. The main difference remains in precise timing computation (with respect to rate coding in SoA) allowing to compute using less spikes, for example 8x8bits MAC operations require 1.1k spikes with precise timing and 66k spikes with rate coding, saving energy while computing an equivalent DSP computation. This comparison will be extended at the application level in the next chapter and compared to other temporal approaches [1].

Chip	SpiNNaker [2]	TrueNorth[3]	**INSPIRE**
Techno	130nm (ARM)	28nm	28nm FDSOI
Coding	Rate	Rate	**Precise Timing**
Neurons	16k	1M	512
Type	Prog	LIF	IF
Synapses	16M	256M	4096
Power	1W	72mW	**115µW** including **42.8µW leakage**
En./syn. evt.	10nJ	25pJ	**26.4pJ**
Tick length	1ms(var.)	1ms	**36.4µs**(var.)*
En./tick (512 neurs)	-	34.18nJ	**4.23nJ**

Table 4.2: Approach comparison with SOA chips.

*benchmark: 8b MAC operators @0.5VDD @-0.3VBB @4MHz

4.4 Discussion

As emerging Spiking sensors provide new solutions for low power sensing, no matching processing solution has been proposed yet. Their main purpose is to send events only when it receives information, dropping the frame notion. This leads to few events being transmitted and lower processing workload. The implemented Proof of Concept can be seen as a DSP chip able to perform any required mathematical operation directly on data contained in the timing of the input events. Temporal precision is limited by the hardware simulation tick length which can be set according to sensor needs. Our proof of concept, INSPIRE, is fully synchronous but already exhibits promising performances compared to other neuromorphic chips. Our topologies specifications lead to a chip integrating 512 neurons for 4096 synapses and using down to 26.4pJ per synaptic events. Moreover, it also requires less of them compared to its frequency coded counterpart leading to consequent power savings. Considering timing approaches rather than rate coded approaches for future spiking sensors is thus a promising way for breakthrough in low power processing.

As shown in Fig. 4.11, the majority of the power consumption of the chip comes from the dynamic power generated by the synchronous Controller and Scheduler. The POC INSPIRE was designed to prove feasibility and show advantages when processing timing features compared to rate features, but it was not designed to be integrated in a GALS architecture. Designing a cluster to be integrated in a GALS architecture requires implementing the Controller and Storing part of the Scheduler in asynchronous logic. As the input activity is sparse, the generated dynamic power will be mitigated. The power consumed by the synchronous processing will have the same characteristics and the energy consumed will be reduced by the local DVFS strategies [4].

We proved the interest that resides in processing timing data from inputs spike streams with our SNN topologies. We also implemented custom hardware both flexible enough to take advantages of the special characteristics of our topologies and competitive with SOA chips in terms of energy per synaptic events. Moreover, the main source of energy consumption is identified and its impact will be mitigated in the GALS implementation. The next chapter will focus on applying the developed topologies and models to a set of applications and more precisely conversion of Artificial Neural Networks.

4.5 Références

[1] B. Rueckauer and S. C. Liu. Conversion of analog to spiking neural networks using sparse temporal coding. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, May 2018. 103

[2] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana. The spinnaker project. *Proceedings of the IEEE*, 102(5):652–665, May 2014. 103

[3] F. Akopyan et al. Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(10):1537–1557, Oct 2015. 103

[4] Vivet Pascal et al. *On Line Power Optimization of Data Flow Multi-core Architecture Based on Vdd-Hopping for Local DVFS*, pages 94–104. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. 104

Chapter 5

Application of the Framework

Sommaire

5.1 Introduction

We developed in previous chapters the tools for quickly mapping application. The library of Fixed-Point operators enables exploring SNN topologies for a set of constraints and the SystemC model enables fast hardware simulation of the designed topologies. The developed proof of concept already reaches SOA efficiency while being fully synchronous and thus sub-optimal.

In this chapter, we use the developed tools in order to design topologies answering specific applications. As mentioned, the library of operators handles Fixed Point operations that can be assembled to process streams of spikes for which the timing of the event is important.

The applications used in this chapter mainly revolve around the use of ANN as designing base. These networks have already proven efficient in multiple fields and we aim here at converting their efficiency to the spiking domain using our operators. ANN to SNN conversions were already performed for different neural coding including rate coded [1][2][3][4] and Time-To-First-Spike [5]. The aim of this chapter is threefold: implement a precise timing-ready training scheme for ANNs; show how the developed topologies enable design of more complex converted ANNs than other coding strategies and prove the benefits of using this conversion. The highlight here is not focused on high classification accuracy, but mainly on the conversion method and its comparison to baseline implementations.

5.2 MNIST

This first section will focus on the MNIST database and the conversion of CNN trained on it. The MNIST database is composed of images of handwritten digits "0" to "9". It is composed of a training set of 60,000 examples and a test set of 10,000 examples. Fig5.1 shows examples of the 28x28 pixel images included in this database.

It is a good database for experiencing with machine learning and implement proof of concepts. It is commonly used as a recognition benchmark for ANN topologies and tweaks in their hyperparameters. State of the Art NNs achieve under 0.3% error rate [6][7][8] when classifying images from the MNIST database. Also this part will not be focused on achieving the highest accuracy, but rather on the method used for converting the trained CNN into a Spiking CNN.

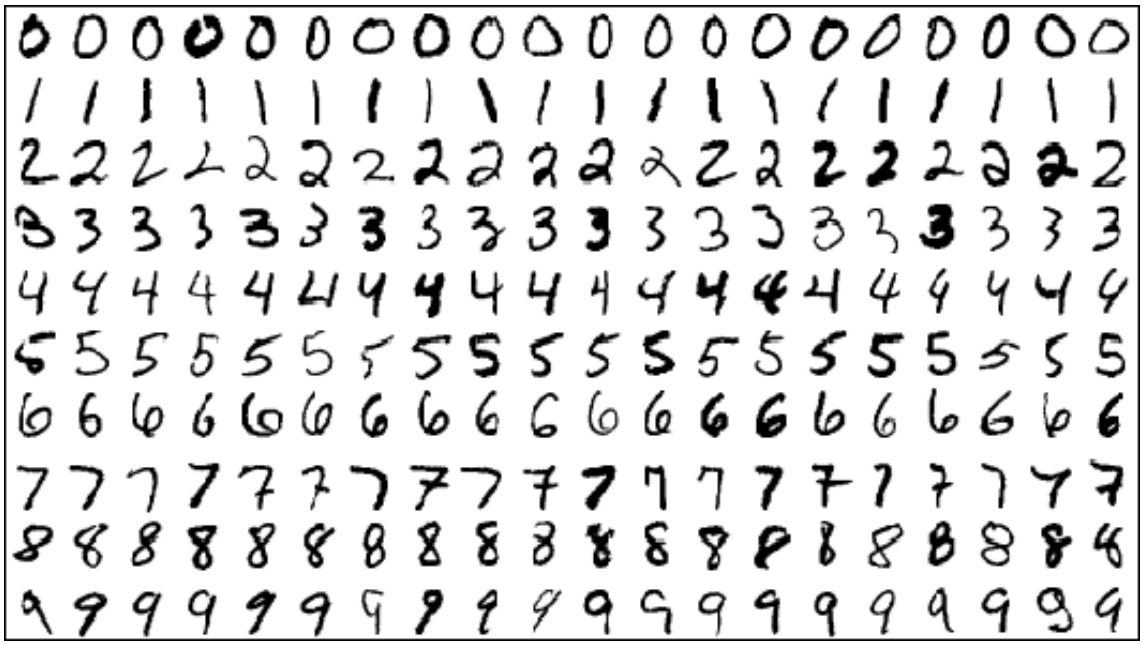


Figure 5.1: Handwritten digits from the MNIST database.

5.2.1 Convolutional Neural Networks

Multi-layers fully connected networks have proven efficient when used with gradient descent for classification tasks. Their ability to learn complex non-linear mapping enables high classification rates. However, they also present weaknesses which are mainly contained in their structure. First, as their name indicates, fully connected networks rely on complete connectivity in-between layers. MNIST images are 28x28 which already represents 784 input pixels that have to be connected to each neuron of the first layer of the network. The trained network are naturally heavy memory-wise. The second weakness is sensitivity to translations and local distortions. As neurons specialize to certain features, they are linked to specific coordinates of the input images. In order to detect the same translated feature, another neuron will specialize with the same weight pattern translated on the input image. Using this mechanism, large fully connected networks could be resilient to translations and local distortion at the cost of weight redundancy.

In order to cope with those issues, CNN are used [9]. They have a layered structure with shared weights and partial connectivity in-between layers. They are composed of 3 main types of layers: the convolution layer, the pooling layer and the fully connected layer. A convolution layer is defined by a set of kernels trained by gradient descent. Each kernel is applied to the whole input image resulting in kernel dedicated channels as shown in Fig.5.2. For a kernel of size k, the computation performed at each neuron (x,y) of the convolution layer is the following:

$$o_{(\mathrm{K},x,y)} = f\Big(\sum_{i=x}^{x+k-1}\sum_{y=j}^{y+k-1} \mathrm{I}_{(i,j)}\mathrm{W}_{\mathrm{K},(i-x,j-y)}\Big) \tag{5.1}$$

where f is the activation function (sigmoid, tanh, ReLU, SeLU...), I is the input activation matrix and W the kernel tensor. The shared kernels each specializes into specific features that are detected through the whole input channel. The pooling layer is designed to reduce the size of the input channels while still maintaining the relevant information. Each neuron of this layer essentially performs a chosen operation (average, max...) on a fixed number of input neurons from the same channel. Performing this operation divides the input dimensions by the pooling size while still maintaining the presence of features within this region. Thus, it also participates to resilience to local distortion and translations.

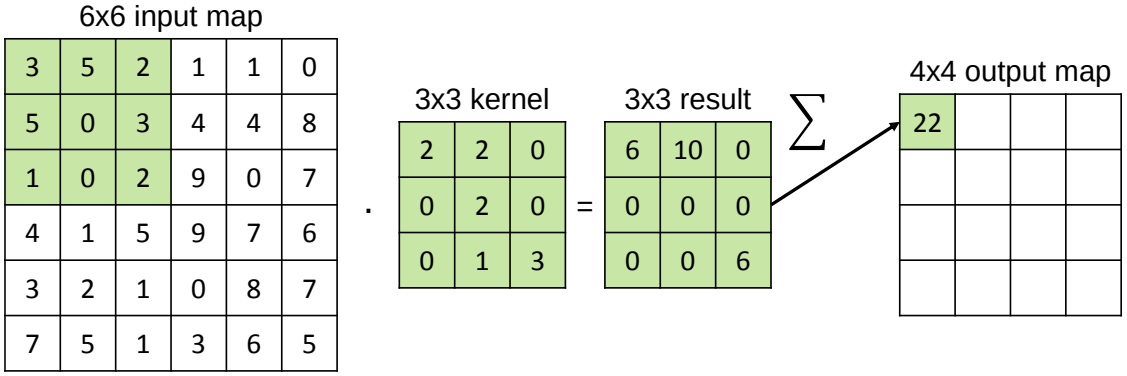


Figure 5.2: Convolution performed with a 3x3 kernel on a 6x6 input map with stride 1 gives an output map of size 4x4. The results follow equation 5.1.

Using multiple pairs of convolutional and pooling layers, the input image dimensions can be reduced to few pixels distributed on multiple channels. Such network when trained with gradient descent can already perform classification tasks as the successive layers hierarchically extract more and more complex features. A fully connected layer is generally added as classifier at the end. It uses the output of every neurons from every kernel-dedicated channel and performs the classification using as many neurons as there are classes to classify. Fig.5.3 shows the global structure of a 5-layer CNN including 2 convolution layers of 5 and 10 kernels, 2 pooling layers and a fully connected layer for the classification. This type of 5-layers CNN is enough to achieve over 95% recognition using the MNIST database and will be used for this section.

5.2.2 Baseline training

For simplification purposes, the size of the MNIST images was reduced to 18x18 pixels and the number of grey levels reduced from 256 to 16. The aim is not to achieve particularly high accuracy but rather demonstrate our method and such degraded input images are sufficient.

The network topology used here is the mentioned 5-layer CNN using the following parameters:

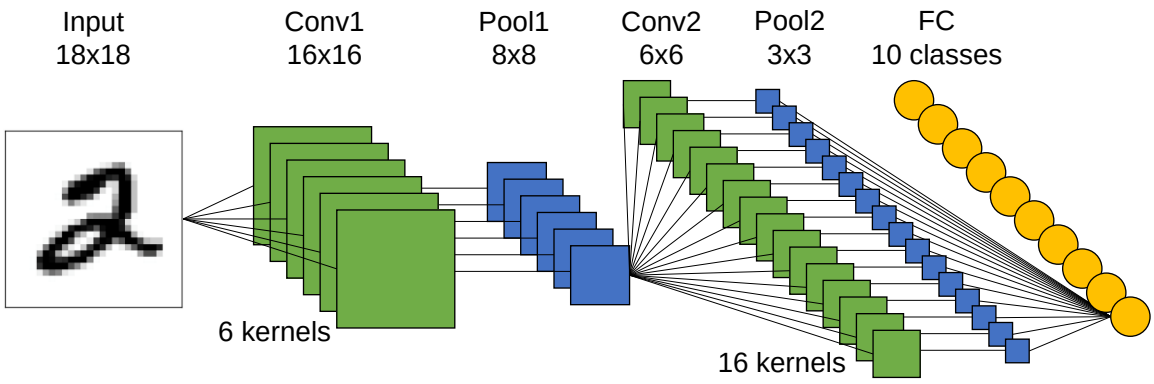


Figure 5.3: 5-layer CNN using 18x18 input images. Conv1 is composed of 6 kernels and conv2 16 kernels. The classifier is Fully Connected layer of 10 neurons.

- 18x18 input images using 16 grey levels
- C1 convolutional layer with 6 (3x3) kernels using the ReLU activation function
- P1 max pooling of size (2x2)
- C2 convolutional layer with 16 (3x3) kernels using the ReLU activation function
- P2 max pooling of size (2x2)
- Fc1 10 fully connected neurons for classification

The ReLU activation function performs the Maximum operation between the input and zero. It is used to accelerate the learning using gradient descent and in this example will be easily converted using our operators. The number of kernels for the convolutional layers was chosen arbitrarily. The pooling function chosen here is the maximum function which fits the examples implemented in Chap.2.

The gradient descent training is done using N2D2, a Neural Network Computer-Aided Design (CAD) tool developed in CEA [10]. It integrates both a framework for designing and training ANNs and XNet for the SNN design and inference. The first training step is done using double weights and 8b grey level 18x18 images. The resulting network achieves 97.06% recognition rate.

Then, in order to prepare the network for exporting weights to our precise timing topology, we need to quantize the grey levels and weights of our network. Thus we perform a second training step while down-scaling the weights to 5b signed and grey level to 4b. The starting point used for this phase is the network obtained during the first training step. This quantization step achieves 96.91% accuracy which will be used for the conversion to precise timing domain.

5.2.3 Building Precise Timing CNNs

We now need to define a network topology that will receive the weights learned in the previous steps. To do so, we use the equations computed by the different layers and convert them using the operators defined in Chap.2. The main function we have to implement is the following:

$$o_{(\mathrm{K},x,y)} = max(\sum_{i=x}^{x+k-1}\sum_{y=j}^{y+k-1} \mathrm{I}_{(i,j)}\mathrm{W}_{\mathrm{K},(i-x,j-y)}, 0) \tag{5.2}$$

We will thus define hierarchically networks computing the previous equations up to the complete CNN topology.

Kernel computation

The first computation to be made is the Kernel convolution. This step can be performed using the Linear Combination defined in Chap.2 with minimal requirements with respect to sign. As only positive results will be sent to the next layer, negative results do not have to be kept. Fig.5.4 shows the topology used for the kernel computation performing the equation:

$$o_{(x,y)} = \sum_{i=x}^{x+k-1}\sum_{y=j}^{y+k-1} \mathrm{I}_{(i,j)}\mathrm{W}_{k,i-x,j-y} \tag{5.3}$$

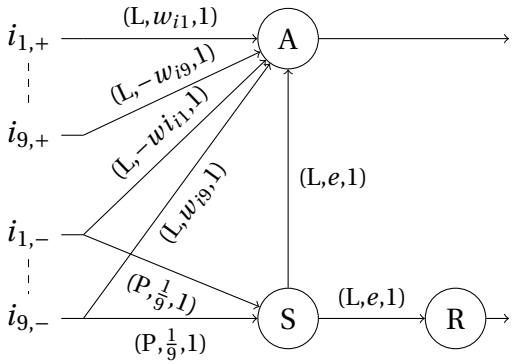


Figure 5.4: Topology used for Kernel computation.

The Maximum computation performed for the ReLU activation function can be performed in the max pooling layer, saving neurons and network activity.

Max pooling computation

As we use 2x2 max pooling, we have to perform here a 5 input maximum following this equation:

$$o_{(x,y)} = max(i_{(x,y)}, i_{(x+1,y)}, i_{(x,y+1)}, i_{(x+1,y+1)}, 0) \tag{5.4}$$

We can derive this operator from the 2-input maximum operator developed in Chap.2. Fig.5.5 shows the obtained topology for this equation.

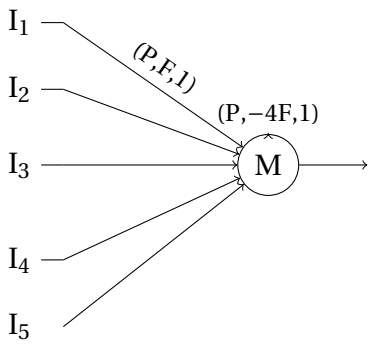


Figure 5.5: 5-input maximum operator used as max pooling unit. 5 neurons will stimulate M for each operation. The first stimulation makes it spike and inhibit itself. The 4 other stimuli reset the neuron to its rest potential.

Operator concatenation

The convolutional units and pooling units can readily be assembled into layers to perform CNN computations. However, it is better to concatenate operators used per pooling unit in order to save neurons and activity. This concatenation uses 1 max pooling operator and 4 kernel computation operators to form a new unit performing the computation of both layers. Fig.5.6 shows the resulting unit. This unit will perform both equations defined previously in a more efficient way.

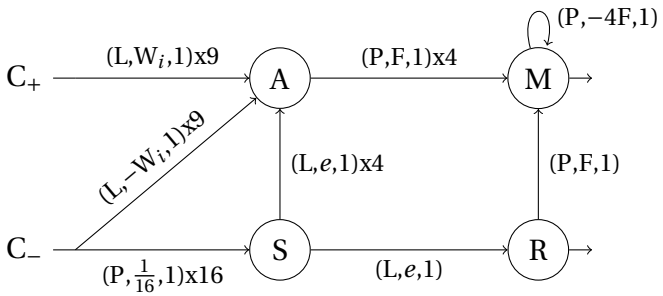


Figure 5.6: Convolution + pooling unit. The synchronization neuron (S) uses the second spike from the 16 input pixel in the max pooling zone. The spikes from the accumulators (A) are directly sent to the neuron (M) performing the maximum operation with the null signal being sent from (R).

The synchronization neurons are merged into 1 serving the 4 kernel operators. This synchronization neuron recalls the 4 accumulators when their stimulation is complete for the current input. The output time of the 4 accumulators are compared to the null interval value generated by the merged recall neuron used for the maximum computation.

Fully Connected

The fully connected layer is essentially composed of 10 linear combination operators performing the classification. As for conventional CNNs, the result class is given by the output operator having the largest interval. Fig.5.7 is a reminder of the principle of the operator with its main links to previous layers.

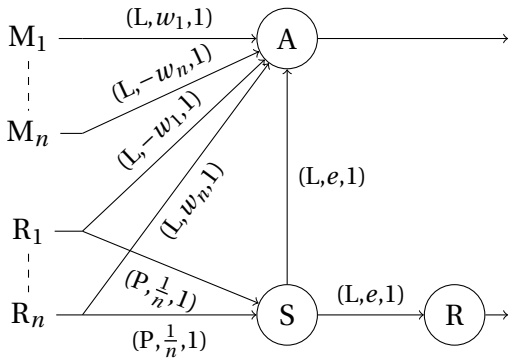


Figure 5.7: Topology used for FC layer computation.

The defined layers once assembled form the complete operator described in section 5.2.2. We can now export the learned weights into our topology and perform the spiking inference.

5.2.4 Precise Timing CNN inference

Once the weights are loaded into our topology, two more points have to be explored. The first one is the conversion from the 18x18 4b grey level to stream of spikes to be integrated by our SNN. The second one is the intermediate results range: the second training step here only performs quantization of the input activation levels and weights, but does not control the range of intermediates results which could not fit our Fixed-Point operators.

MNIST conversion

The conversion from the reduced MNIST database requires associating couples of event per pixel. Each pixel has to emit 2 events defining its grey level, each event having an origin ID associated to a timestamp. This creates the AER events that will be sent to our network for the inference.

First, the pixels are mapped to IDs to be used as input channels for the operator. Then, couple of spikes are created using the following encoding function for the intervals:

$$\delta t(x) = \lfloor \frac{x}{16} \rfloor \tag{5.5}$$

where x represents the pixel intensity. This encodes the pixel intensity in 16 linearly distributed grey levels. A δt of 1 corresponds to a white pixel and a δt of 16 corresponds to a black pixel. Fig.5.8 summarizes the conversion used for the MNIST images.

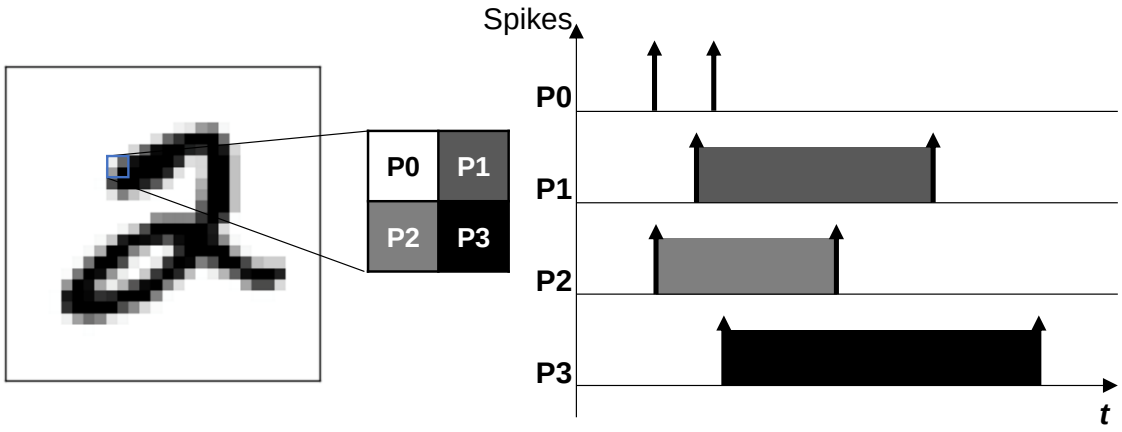


Figure 5.8: Converting MNIST database to inter spike interval values. Brighter pixel have narrower intervals.

Range

The implemented operators compute using Fixed-Point arithmetic. The training done in N2D2 corresponds to Floating Point operations done with quantized weights and input activation but does not constrain the intermediate results in terms of precision. In order to keep the same precision with precise timing SNN, one would have to use different capacities for the neurons of the different layers.

The first convolutional + pooling layer performs the intended operation using intervals coded on 4b and signed weights on 5b. The output of the first couple of layers can thus be contained on 11b timing intervals, which requires the accumulator neurons to have membrane potentials defined on 11b and the further layers to have increasingly important capacities. In order to keep homogeneous capacity requirements neuron-wise, the intermediate results are forced on 4b down-scaling the obtained results. The rounding have to be done after the accumulation phase, when the results are recalled and compared to 0. Fig.5.9 shows the modification needed in order to round the intervals obtained. The linear synapse with elementary weight that was used to recall the values stored in the accumulators and the empty neuron is changed to have a 2^7 weight, meaning the intermediate accumulation result stored on 11b is partly ignored as only the 4 MSB will have an impact on the output interval sent to further layers.

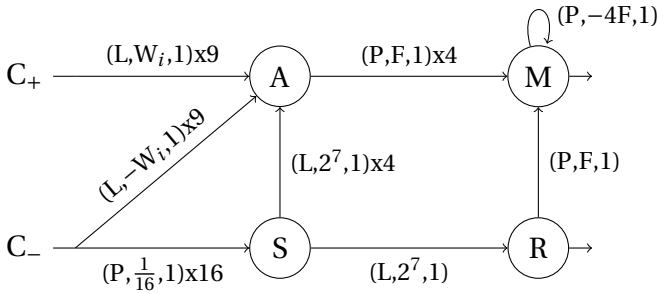


Figure 5.9: Rounding is computed when recalling the values stored in the accumulators and the recall neuron with the updated weight 2^7.

5.2.5 Results

As explained in the previous section, the intermediate results need to be rounded in order to fit uniform neuron capacities. This leads the accuracy achieved by the spiking CNN to drop to 92.8% recognition rate. It uses 11b for neurons variables, 8b for synapses, including synapse type, delay and weight.

When the equivalent rate coded network would use 2,122 neurons with average pooling leveraging ReLU activation functions, we use here 3,706 neurons (C1: 1,536 neurons, P1: 1,152, C2: 576, P2: 432 and Fc:10). The average Fan-in required is 43% higher in the time-coded version (31.9 input connections per neuron) than in the rate coded network (22.3 input per neuron).

However, our temporal approach use at most 4,344 spikes per classification as when the amount of spikes used with rate coding is higher. In order to determine the recognized class in the rate coded version, a specific class has to output N more spikes than any other, N being a parameter. Setting a difference of 20 spikes (N = 20) to keep accuracy over 96% leads to 273% increase in the total number of spikes used compared to the time-coded version (for a total averaging at 11,885 spikes).

5.2.6 Discussion

Precise timing is a promising solution for lightweight spiking CNN implementation as they consume fewer spikes than their rate coded counterpart to achieve equivalent accuracy. Such conversion and comparison were also explored with rate coded SNNs [11], performing 92% accuracy on Field-Programmable Gate Array (FPGA), or TTFS SNNs [5], achieving accuracy equivalent to their ANN counterpart while only using 1 spike per neuron. Moreover, when timing based strategies give us the possibility for short input integration and one-shot operations, rate coding will only compute accurately only if the features are present for a long enough time at the input.

Using Precise Timing networks in order to mimic CNN operations is a promising way to improve its energy efficiency. Compared to a rate coded approach, the same accuracy can be reached using less synaptic events and fewer input spikes. The precise timing approach is closely related to the TTFS coding that also uses timings to encode the information. When TTFS uses a common "Start" signal and 1 "Stop" per channel for the intervals, the precise timing approach computes the intervals from each channels independently. The proposed conversion method provides accurate precise timing CNNs and can be extended to other types of ANNs.

We acknowledge the fact that the example chosen here is not fitting the use of DVS which provides both differential events and grey level event pairs. The MNIST conversion reduces the DVS model to its grey level event pairs using only static images, thus not fully utilizing its capacities. Moreover, some improvements can be made for this particular example. First, using ReLU activation function will inevitably lead to a non negligible proportion of null intermediate results. Those null results have to be sent to the next layer as two events having the same timestamp, generating unnecessary activity. Nevertheless, those events are needed in this operator implementation as they trigger the synchronization neurons designed to recall results when all the input events are arrived. By using external control neurons being able to automatically synchronize the layer computations, the events coding for the null interval can be avoided at each layer.

Secondly, the training of the down-scaled version did not include rounding, leading to a 4.11% accuracy drop when performing the SNN inference. This drop due to information loss in-between layers can be mitigated. The down-scaled training step has to integrate range monitoring in order to optimize the rounding performed in-between layers. The next conversion developed will use this point and a more realistic application.

5.3 N-TIDIGITS18

This section will focus on the conversion of Recurrent Neural Networks (RNN) designed to classify spoken digits. The database used for this application is N-TIDIGITS18 [12][13] that was recently created. It is composed of spoken digits from TIDIGITS [14][15] recorded using the DAS AER EAR [16]. The 25,102 resulting spiking samples represents men, women, boys and girls pronouncing digits or sequence of digits. The mentioned digits are “oh”, “zero” and “1” to “9”.

The database can be extracted from an HDF5 file [17] containing the different sequences as streams of AER events. As specified before, these AER events are coupled of data representing the origin channel ID and emission timestamp. As described in Chap.1 the lower channel ID represents the lower tones and highest IDs the highest tones, following a logarithmic distribution in frequencies. Fig.5.10 gives an example of input chronogram for the input sequence 5 8 9 9 2. For our application, we extract only the digits from the database. This represents a total of 2,464 training samples and 2,486 testing samples.

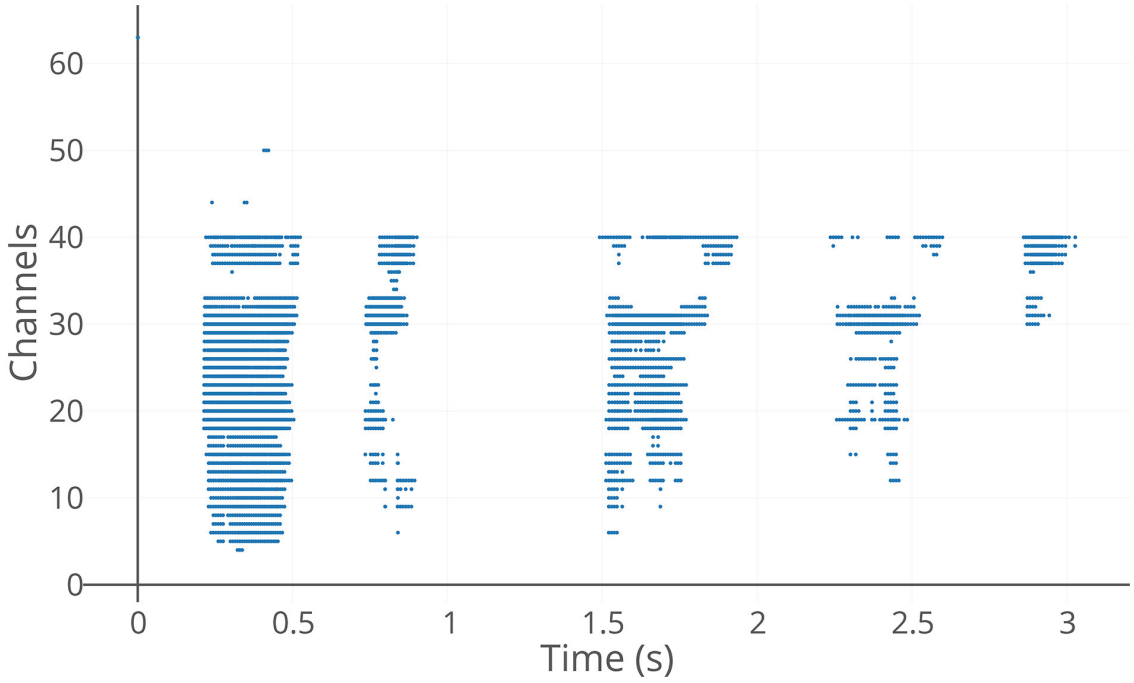


Figure 5.10: Input chronogram for the sequence "5 8 9 9 2".

5.3.1 Long Short Term Memory, Gated Recurrent Unit and Recurrent Neural Networks

In order to classify audio samples, the dynamics of each channel has to be taken into account. One could use the obtained chronogram as an image to be classified with CNNs rec-

ognizing patterns in the channels dynamics. For a network to grasp the input dynamics, it has to keep information about the previous states. For this purpose, Recurrent Neural Networks use previous results in future computations with weights trained by gradient descent. The recurrent connection training is done "unfolding" the network along the time axis, considering the same network at the previous input as a different one linked with the synapses to be trained.

Different classes of recurrent networks will be used to tackle spoken digit recognition in this section. The simpler recurrent network model is the Recurrent Neural Network. For this topology, each recurrent unit is composed of 1 neuron and the output of each unit of the layer is connected to the input of each unit from the same layer. The computed equation at each layer is the following:

$$h_t = f(\mathrm{W}_x x_t + \mathrm{W}_h h_{t-1} + b_h) \tag{5.6}$$

where h is the output, h_{t-1} the previous state, x_t the input activation, b_h the bias, W_x the input weights and W_h the recurrent weights. f denotes the activation function (sigmoid, tanh, ReLU...). For this section, the ReLU activation function will be chosen for the precise timing RNN implementation.

Another recurrent network implementation is the Gated Recurrent Unit (GRU). Its principle is based on having the choice to update or reset the output value in function of internal gates values. The GRU equations [18] used are as follows:

$$z_t = \sigma_z(\mathrm{W}_{x_z} x_t + \mathrm{W}_{h_z} h_{t-1} + b_z) \tag{5.7}$$

$$r_t = \sigma_r(\mathrm{W}_{x_r} x_t + \mathrm{W}_{h_r} h_{t-1} + b_r) \tag{5.8}$$

$$g_t = tanh(\mathrm{W}_{x_g} x_t + \mathrm{W}_{h_g}(r_t \odot h_{t-1}) + b_g) \tag{5.9}$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot g_t \tag{5.10}$$

where z and r are respectively the update and reset gates. h is the activation function, sum of the weighted previous value h_{t-1} and new candidate g_t

The last recurrent network implementation that will be explored in this section is the Long Short Term Memory (LSTM). As for the GRU, it contains internal gates deciding if the LSTM unit forgets its current value, updating it to a new candidate value, or not. The LSTM equations used for our training are those of the LSTM with a forget gate [19]:

$$i_t = \sigma_i(\mathrm{W}_{x_i} x_t + \mathrm{W}_{h_i} h_{t-1} + b_i) \tag{5.11}$$

$$f_t = \sigma_f(\mathrm{W}_{x_f} x_t + \mathrm{W}_{h_f} h_{t-1} + b_f) \tag{5.12}$$

$$o_t = \sigma_o(\mathrm{W}_{x_o} x_t + \mathrm{W}_{h_o} h_{t-1} + b_o) \tag{5.13}$$

$$g_t = tanh(\mathrm{W}_{x_g} x_t + \mathrm{W}_{h_g} h_{t-1} + b_g) \tag{5.14}$$

$$c_t = i_t \odot g_t + f_t \odot c_{t-1} \tag{5.15}$$

$$h_t = tanh(c_t) \odot o_t \tag{5.16}$$

where i, f anf o are respectively the input, forget and output gates. c and h are respectively the cell state and hidden state. The new c_t value is calculated as the centroid of the c_{t-1} value weighted by the forget gate f_t and the candidate g_t value weighted by the input gate i_t. The new h_t is calculated using this new c_t cell value and the output gate o_t.

5.3.2 Baseline training

The training was done using Torch NN with the dataset restricted to digits only. The trained topologies are all composed of 1 to 3 successive recurrent layers of 100 to 200 units. These recurrent layers are connected to a FC layer of size 11 performing the classification. As specified before, the classic RNN-based networks will be using ReLU activation functions for conversion to precise timing topologies purposes.

The training was done using a categorical cross entropy objective on the output of the FC at each "frame". The Adam optimizer was used on 100 epochs with a learning rate starting at 0.001. A validation run was performed using the testing samples every epoch to monitor performances. The topologies performing the best validation scores for each set of parameters were saved.

In [12], the FC layer is solicited only once per digit at the end of the sample to classify. Compared to this approach, the FC layer in our networks outputs a result at every frame rather than once per digit. This changes both the training and inference behavior. During training, the presence of input activity due to a spoken digit is given as information to the network. If the current input is not part of a digit (mainly beginning and ending voids and artifacts), the network is trained to recognize a temporary twelfth class associated to noise and that will be dropped for the inference phase. The network is thus trained to recognize that the current input activity is part of the phonemes of the current label. During the inference process, input activity bulks can be found and for each such block a poll is made at the output of the network. As the input frames loaded with data from the digit phonemes go through the network, a trained network's output will converge to an output class and the polling policy will determine which class was recognized.

Features

As we intend to use ANN with spiking samples, we need to convert back those AER events to have a frame format that can be exploited by ANNs. We use the same 5 ms per frame as in [12] that was determined to be optimal for their features. The spiking samples are thus

split into 5 ms time windows and for each time window an input vector of size 64 is created. The slots of the created vector contain the number of spikes that were received from the corresponding input channel during the current time window.

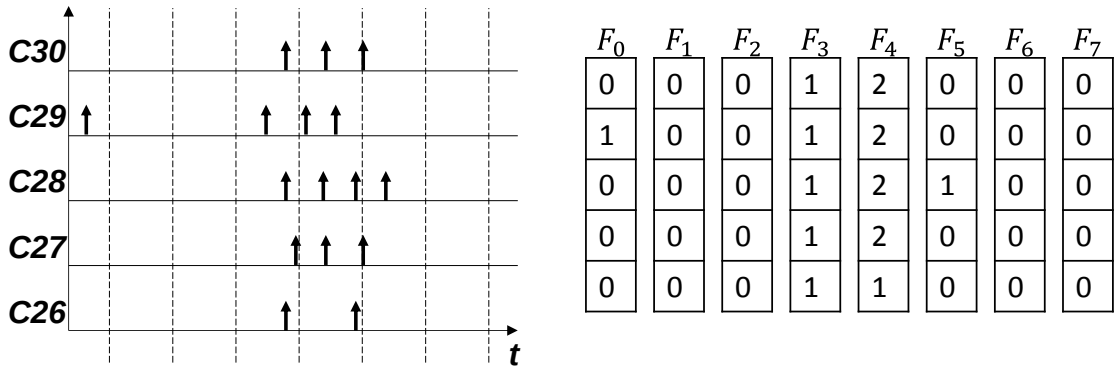


Figure 5.11: Constant time bins features: for each 5 ms window, count the number of event per channel and create an input vector.

The feature created using all of the created frames will be referred as "Constant time bins" (Fig.5.11). Another set of ANN features was created essentially removing the empty frames from the first set of frames. This second set of inputs containing only frames having activity will be referred as the "Event driven constant time bins" (Fig.5.12), meaning that the size of the time window is still constant but the usage of the frame is determined by whether it has activity or not.

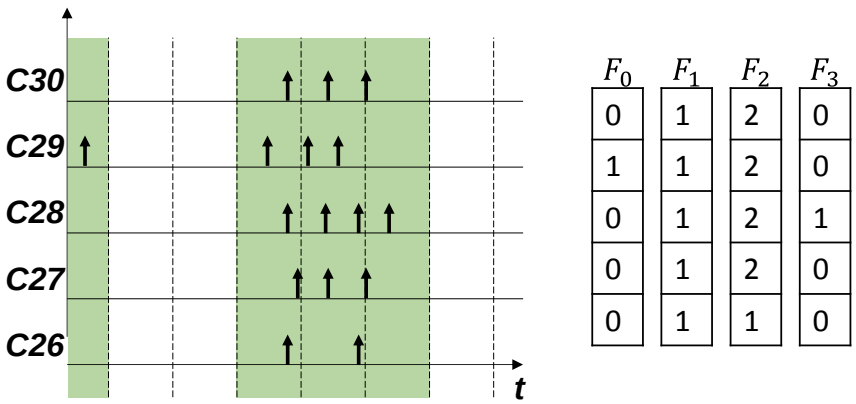


Figure 5.12: Event Driven Constant time bins features: for each 5 ms window, count the number of event per channel and create an input vector if not empty (green).

Quantization

The quantization step performed for the CNNs in sec.5.2 is also performed here for the RNNs. Once the ReLU RNN has followed the training process, the obtained network is exported and its weights are quantized to prepare for conversion. In this section, the intermediate result range is monitored and the output of the recurrent layers rounded during the re-training of the quantized RNN to mitigate accuracy loss.

The quantization method is thus composed of 3 main points here, starting from the 32b float RNN:

- Round input weights, recurrent weights and bias to 5b signed.
- Round the output of RNN units to 8b.
- Monitor the range of the intermediate results.

The intermediate result rounding is done using 8b in this implementation. This does guarantee the precision of the Fixed-Point operations to be respected but does not guarantee overflow protection which is performed monitoring the range of the intermediate results.

The training phase of the quantized RNNs is done as follows:

- Start from the best 32b float RNN with the same dimensions (layers and size) and use quantization.
- Train on 32b precision for the weights while keeping the rounding at the output of RNN units.
- Perform weight and bias quantization before every validation run.

Results

Table 5.1 summarizes the accuracy achieved by the previously defined topologies for the different feature used. As one can expect, at fixed network size, the gated units outperform RNNs and the RNNs perform better than their quantized versions.

The network composed of gated units seems to be able to retrieve further information from the length of the silences contained within digits as they perform better using "Constant time bins". On the other hand, RNNs have a difficult time maintaining relevant information for multiple frames as shown with the 60.56% accuracy achieved by the RNN using Constant time bins. As the information about past phonemes can by erased during silences within digits, they achieve better accuracy using "Event driven constant time bins" with 75.46% for the same topology size.

5.3.3 Building Recurrent SNNs

We will now detail the implementation of the SNN topology used to perform the RNN computation using precise timing. The main points that will have to be explored are the feature

Table 5.1: Summary of investigated recurrent topologies and comparison to state of the art. (layers x size)

	Sensor	Feature type	Network	Accuracy(%)
[20]	AMS1b[16] (N-TIDIGITS18)	Constant time bins	GRU	82.82
			CNN	87.65
[12] [1]	AMS1b (N-TIDIGITS18)	Constant time bins	GRU(2x100)	86.4
		Exponential features	GRU(2x100)	90.9
	AMS1c[21]	Constant time bins	GRU(2x100)	88.6
		Exponential features	GRU(2x100)	91.1
This work [2]	AMS1b (N-TIDIGITS18)	Constant time bins	LSTM(1x100)	85.72
			GRU(1x100)	84.67
			RNN(1x100)	**60.56**
		Event driven constant time bins	LSTM(1x100)	82.78
			GRU(1x100)	81.77
			RNN(1x100)	**75.86**
			Quant. RNN(1x100)	**75.46**

[1] GRU(2x100) here is followed by a Dense layer of size 100 and a Fully Connected layer of size 11. Classification is performed once per sample

[2] Our topologies are followed by a Fully Connected Layer of size 11. Classification is performed every frame of the sample and polling decides the result

conversion, the recurrence implementation and the different update scheduling possibilities.

Feature conversion

First, the recurrent networks were trained using features based on counting the number of events per channel in 5 ms windows. Using such time windows with low event rates, the information contained in the number of event per frame per channel is close to the average inter spike interval that could be used in our topologies.

Nevertheless, for the conversion to be fitting exactly the trained networks, we need to convert the spike count to timing data that can be used in our operators. Fig.5.13 shows the operator principle: for each channel, a pair of neurons (A,R) is used to count the incoming spikes. When the count stored in the pair is recalled, the count is naturally converted to an interval to be processed in the recurrent layers. The (C) neuron controls the presence of input for this particular channel. If no spike was received during the period, the (C) neuron will not emit a spike when stimulated at the “Full - Elementary” potential. Thus no interval will be sent to further layers, reducing the number of synaptic events used.

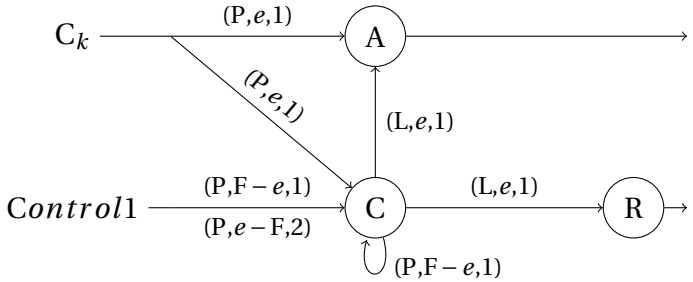


Figure 5.13: Counting input events and sending the result to the recurrent layers when recalled by the Control1 channel.

Recurrence

The second principle used in the spiking RNN is the recurrence. The main issue we have here is that accumulating neurons cannot both compute and output a result at the same time. Two distinct phases have to be used for both the accumulating phase for the computation and the recalling phase for the computation. Thus, when the result for frame t is being recalled, the interval being outputted cannot be used directly in the same operator for the recurrence. This leads to the implementation of copy neurons. Fig.5.14 shows the implementation of a copy neuron. The copy implementation requires 3 input synapses, 2 for the incoming interval to be copied and 1 for recalling the interval stored.

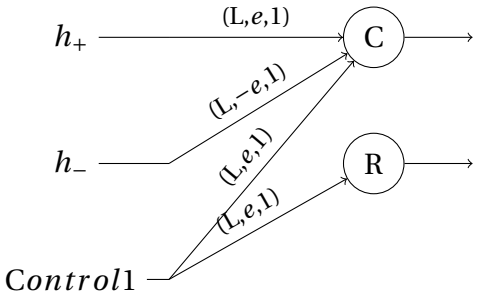


Figure 5.14: Copying timing intervals for reuse purposes. This operator receives an interval from the result of the previous computation and sends the copied result to the whole layer.

We can now reuse the Linear Combination operator and add a max operator and the copy principle for implementing the RNN unit. Fig5.15 shows the complete RNN unit. The Control channels are common to every unit and are responsible for the different steps scheduling. As shown in Fig.5.16, each neuron has specified accumulating and recalling phases. In contrast to the CNN implementation which used local Synchronization neurons to perform local scheduling, this implementation uses global control scheme updating all the units in phases.

The recurrent layer is composed of 100 to 200 units shown in fig5.15. The output of the copy neuron in each unit is also used as input in every unit from the same layer, forming

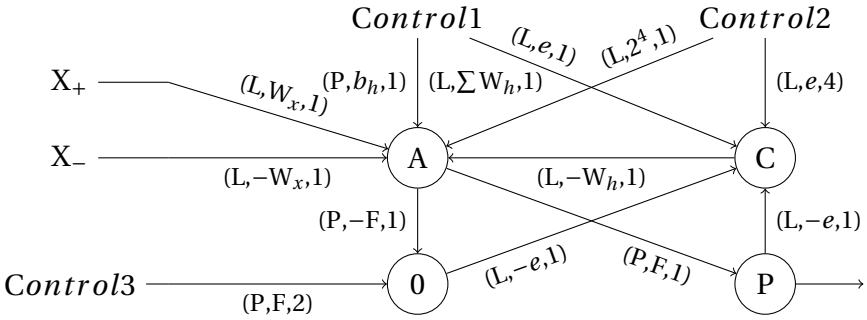


Figure 5.15: Unit used for the spiking RNN topology. The accumulation is performed by the neuron (A), the maximum operation by the neurons (0) and (P) and the copy by the neuron (C). The control channels are responsible for calling the stored values. The value copied in the (C) neuron is sent to every unit from the same layer as input data.

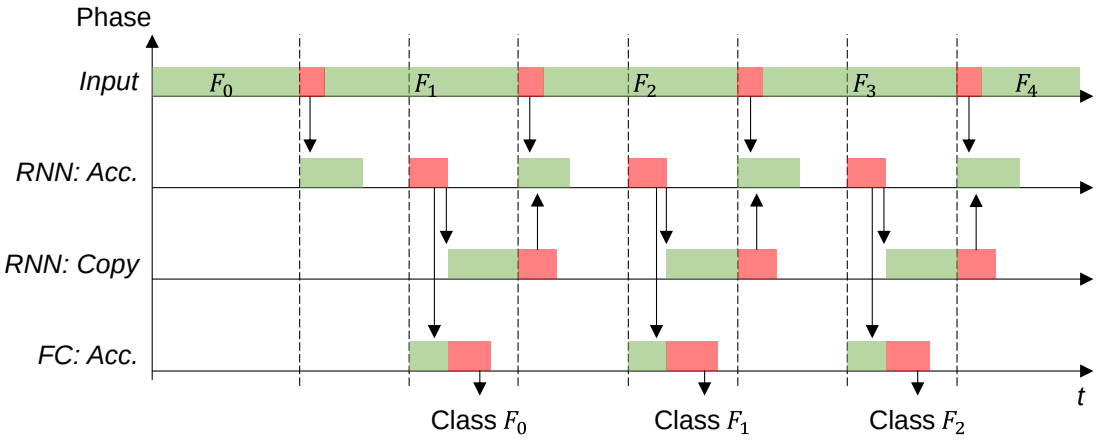


Figure 5.16: Scheduling RNN update. The topology used for this example is 1 RNN layer followed by the FC layer. The green phases represents accumulating the inputs and the red phases represent recalling the stored values. Scheduling the different phases is done using the Control channels.

the recurrence.

Control channels

The RNN update scheduling has to fit the used features during training. For the "Constant time bins", one has to setup a loop using 3 neurons and linear synapses as defined in Fig.5.17. Using a simulation step length of 5ms/512 and neurons having 10b internal variables here, the length of the loop in terms of simulation ticks is 1024. This means two distinct phases of 512 simulation ticks are formed by the neurons C1 and C2 controlling the channels Control 1 and Control 2.

For the event driven constant time bins features, this loop has to start only if input activity was found in the 5 ms window. This can be done setting a condition on the output of the signals of the loop. Using the topology described in Fig.5.18, the control signals from the

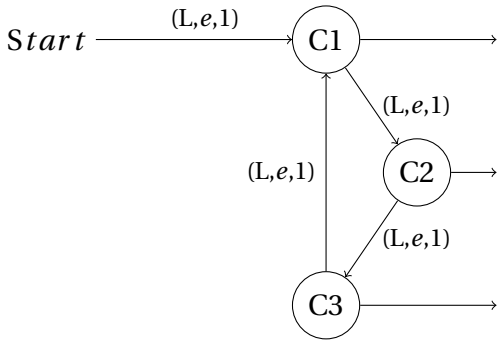


Figure 5.17: Control loop initiated via the "Start" event. This loop will send spikes to the recurrent and FC layers every defined amount of simulation steps.

loop are sent if and only if the frame is not empty.

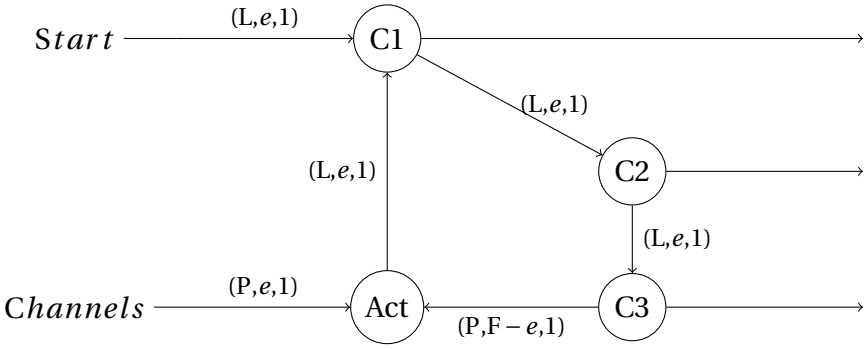


Figure 5.18: A condition is added to the Control Loop. The loop will continue only if it received at least one input spike. If it didnt receive any it will stop and wait for activity.

Fully Connected

The FC layer is essentially the same as the one used for the CNN using 11 classifying neurons. However, for each spiking sample from N-TIDIGITS18 the result of the FC layer is computed and stored. When the sample is over, we use polling over the bulk of the frames susceptible to contain the digit in order to decide which class was recognized by the network.

5.3.4 Recurrent Precise Timing SNN inference

Table 5.2 summarizes the accuracy achieved by the quantized RNNs and their spiking counterparts. In this case there is no accuracy drop as the training phase done before exporting the weights to the spiking topology integrated every constraints for Fixed Point arithmetic. Only the event driven approach was kept for the conversion step as it was proven more efficient for the RNNs here and generates less computation as empty frames are removed.

Table 5.2: Summary of investigated spiking Recurrent Networks. (layers x size)

	Feature type	(Update mode) Network		Accuracy(%)
[12]	**Raw Spiking data**	Phased LSTM		87.75
This work	Event driven constant time bins	LSTM(1x100)		82.78
		GRU(1x100)		81.77
		Quantized RNN(1x100)		75.46
		Quantized RNN(1x150)		76.31
		Quantized RNN(2x100)		77.96
		Quantized RNN(2x150)		78.60
	Raw Spiking data	Event driven update	**Converted** RNN(1x100)	**75.46**
			Converted RNN(1x150)	**76.31**
			Converted RNN(2x100)	**77.96**
			Converted RNN(2x150)	**78.60**

In order to compare the operation per second used per our spiking topologies and baseline GRU and LSTM for the same size, we use the synaptic event as operation for the SNN. On the other hand, GRUs and LSTMs use a majority of 32b MAC that are counted as 2 operations here. Fig5.19 shows the comparison in terms of achieved accuracy versus the number of events per second used.

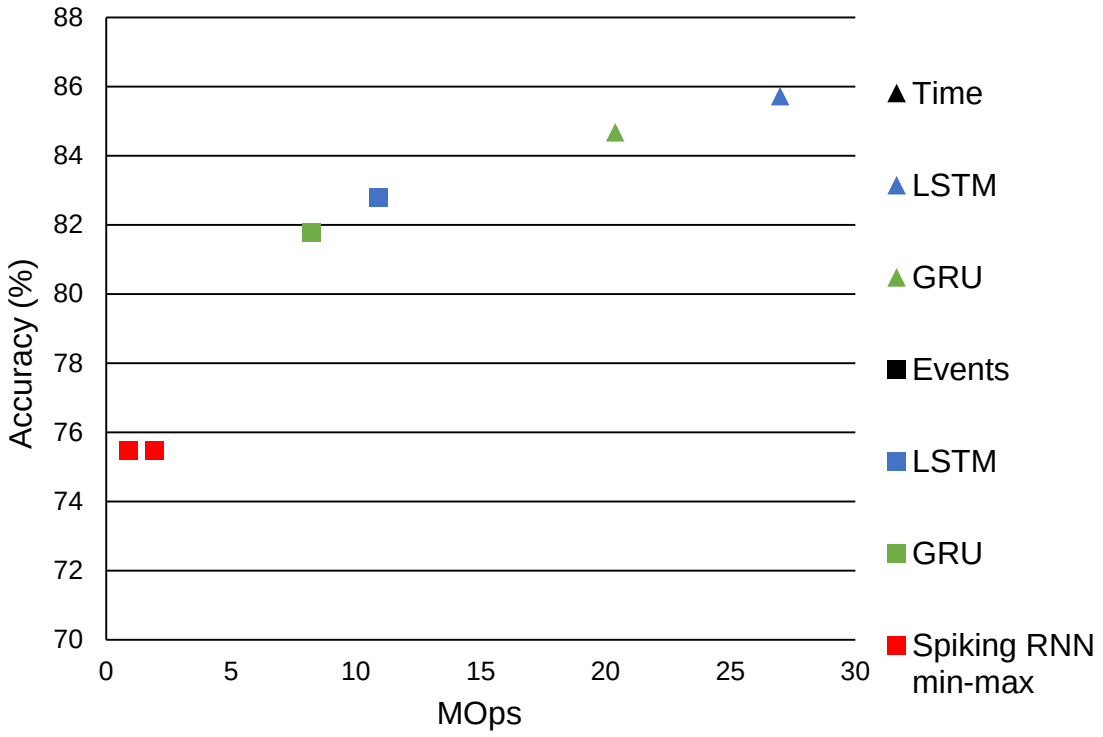


Figure 5.19: Accurary versus MOps required for real time processing @1x100 recurrent layer size. 32b MAC for GRU and LSTM are counted as 2 operations. Synaptic events are counted as 1 operation for the Spiking RNN.

The baseline GRU and LSTM implementations using only non-empty frames use 320 to 1091% more Ops than the Spiking RNN while achieving 6.31 to 7.31% more accuracy. When retrieving information from all the frames from the samples, GRU and LSTM use 947 to 2855% more Ops than the event driven spiking RNN and achieve 9.21 to 10.26% higher accuracy. The accuracy of the RNN can be enhanced using more layers or more units per layers as shown in Fig.5.20. For every topology size, the two data points represent minimum and maximum activity. The minimum activity characterizes frames with low number of events while the maximum activity represents frames with at least one event per channel.

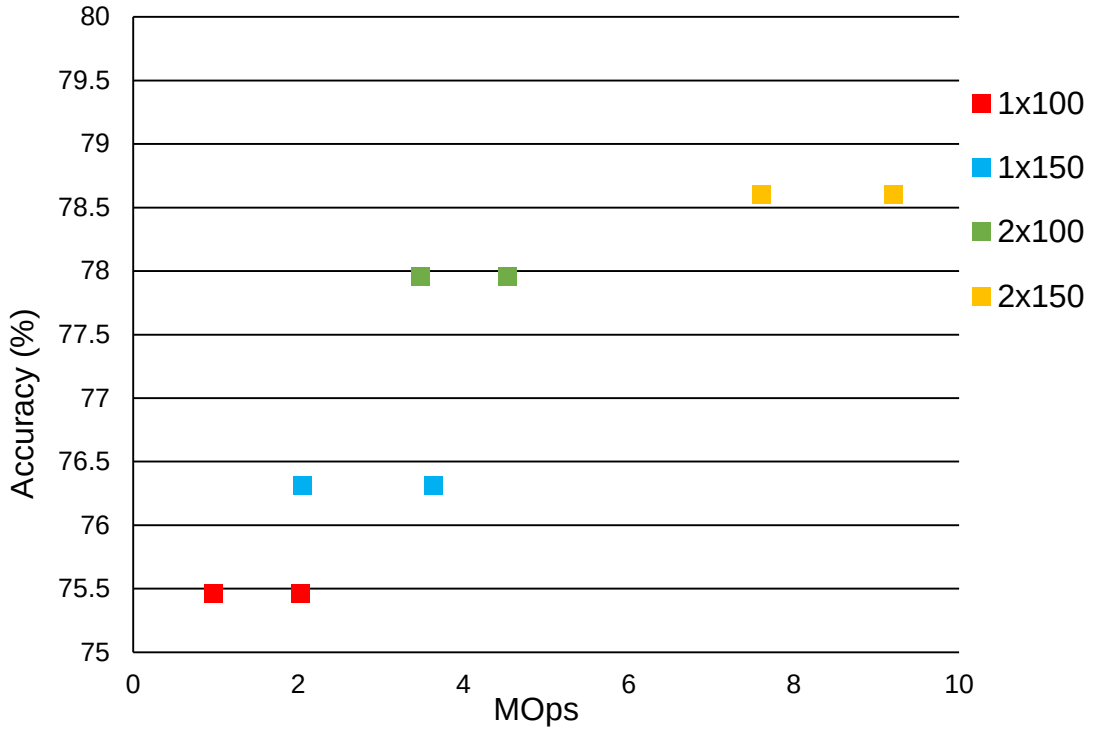


Figure 5.20: Accurary versus MOps required for real time processing. Comparison of Spiking RNNs of different sizes.

5.3.5 Discussion

The spiking RNNs have proven to maintain the same accuracy as their quantized counterparts while generating low activity. The main advantages compared to the gated units is their simplicity which enables low cost spiking implementation. Compared to their ANN implementation, the spiking RNN interface itself naturally with raw spiking data and manages to consume less operations. Using precise timing is key for this implementation, as the developed mechanisms including memorization and synchronization are essential for the recurrence contained in the RNN.

Rate coded spiking RNNs have been implemented [22] by linking the neuron to itself with a sufficient delay. This delay represents the time needed to encode the input data

Table 5.3: Summary of investigated spiking Recurrent Networks @80%train 20%test for N-TIDIGITS18.

	Feature type	(Update mode) Network		Accuracy(%)
This work	Constant time bins	**Quantized** RNN(1x100)		80.73
		Quantized RNN(1x150)		86.27
	Raw Spiking data	**Constant time update**	**Converted** RNN(1x100)	**80.73**
			Converted RNN(1x150)	**86.27**
	Event driven constant time bins	**Quantized** RNN(1x100)		85.22
		Quantized RNN(1x150)		91.11
	Raw Spiking data	**Event driven update**	**Converted** RNN(1x100)	**85.22**
			Converted RNN(1x150)	**91.11**

in this implementation, guarantying proper function of the recurrent link. However, this implementation has its limitations and cannot reproduce the principles achieved by precise timing. For instance, the ability to maintain data for an arbitrary amount of time is mandatory for the "Event driven constant time bins" features and is not within reach of rate coded networks.

Accuracy-wise, the RNNs are naturally outperformed by the gated units. They lack the ability to maintain relevant data for long period of times and the ability to generalize from small training sets. Tab.5.3 explores the accuracy achieved using the same topologies as defined before but modifying the train/test ratio. For this example, 80% of the original database was used as training samples and 20% as test samples. The RNNs trained this way still use the same MOps defined before to process the input spike stream, but can achieve accuracy higher than 90%, figures that were only possible with gated units on the origin database distribution. The RNN, although quite simple in mechanism, has the ability to achieve high accuracy but cannot generalize it from low sample size.

5.4 Références

[1] J. A. Pérez-Carrasco, B. Zhao, C. Serrano, B. Acha, T. Serrano-Gotarredona, S. Chen, and B. Linares-Barranco. Mapping from frame-driven to frame-free event-driven vision systems by low-rate rate coding and coincidence processing–application to feedforward convnets. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(11):2706–2719, Nov 2013. 109

[2] Yongqiang Cao, Yang Chen, and Deepak Khosla. Spiking deep convolutional neural networks for energy-efficient object recognition. *International Journal of Computer Vision*, 113(1):54–66, May 2015. 109

[3] P. U. Diehl, D. Neil, J. Binas, M. Cook, S. Liu, and M. Pfeiffer. Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, July 2015. 109

[4] Eric Hunsberger and Chris Eliasmith. Training spiking deep networks for neuromorphic hardware. *CoRR*, abs/1611.05141, 2016. 109

[5] B. Rueckauer and S. C. Liu. Conversion of analog to spiking neural networks using sparse temporal coding. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, May 2018. 109, 117

[6] R. Benenson. Best performances on multiple benchmarks, http://rodrigob.github.io/are_we_there_yet/build/#classification-dataset-type, 2015. 109

[7] Li Wan et al. Regularization of neural networks using dropconnect. In *Proceedings of the 30thInternational Conference on Machine Learning*, page 1058–1066, 2013. 109

[8] Jost Tobias Springenberg et al. Striving for simplicity: The all convolutional net. *CoRR*, abs/1412.6806, 2014. 109

[9] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998. 110

[10] CEA-LIST. N2d2, https://github.com/cea-list/n2d2, 2017. 112

[11] D. Neil and S. C. Liu. Minitaur, an event-driven fpga-based spiking network accelerator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(12):2621–2628, Dec 2014. 117

[12] J. Anumula, D. Neil, T. Delbruck, and S.-C. Liu. Feature representations for neuromorphic audio spike streams. *Frontiers in Neuroscience*, 12:23, 2018. 119, 121, 124, 128

[13] Datasets from the sensors group, institute of neuroinformatics, zurich. http://sensors.ini.uzh.ch/databases.html. Accessed: 2018-10-01. 119

[14] Tidigits, speech recognition. https://catalog.ldc.upenn.edu/LDC93S10. Accessed: 2018-10-01. 119

[15] A. Graves, D. Eck, N. Beringer, and J. Schmidhuber. Isolated digit recognition with lstm recurrent networks. In *First International Workshop on Biologically Inspired Approaches to Advanced Information Technology*, Lausanne, 2004. 119

[16] V. Chan, S. C. Liu, and A. van Schaik. Aer ear: A matched silicon cochlea pair with address event representation interface. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 54(1):48–59, Jan 2007. 119, 124

[17] N-tidigits18 dataset and instructions. https://docs.google.com/document/d/1Uxe7GsKKXcy6SlDUX4hoJVAC0-UkH-8kr5UXp0Ndi1M/edit. Accessed: 2018-10-01. 119

[18] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014. 120

[19] Felix A. Gers, Jürgen A. Schmidhuber, and Fred A. Cummins. Learning to forget: Continual prediction with lstm. *Neural Comput.*, 12(10):2451–2471, October 2000. 120

[20] D. Neil and S. C. Liu. Effective sensor fusion with event-based sensors and deep network architectures. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2282–2285, May 2016. 124

[21] S. C. Liu, A. van Schaik, B. A. Minch, and T. Delbruck. Asynchronous binaural spatial audition sensor with 2, $times$,64, $times$,4 channel output. *IEEE Transactions on Biomedical Circuits and Systems*, 8(4):453–464, Aug 2014. 124

[22] Peter U. Diehl, Guido Zarrella, Andrew S. Cassidy, Bruno U. Pedroni, and Emre Neftci. Conversion of artificial recurrent neural networks to spiking neural networks for low-power neuromorphic hardware. *CoRR*, abs/1601.04187, 2016. 129

Discussion and Perspectives

The work presented in this document highlights the possibilities created by Spiking Neural Network exploiting timing information. The obtained topologies, conversion methods and hardware were implemented targeting interfacing Dynamic Sensors for embedded applications. Even if the retrieved data from those sensors can be converted back to frame-based content and processed conventionally, we showed how to fully exploit its nature to perform low power processing.

Legacy sensors fall short in comparison to Dynamic sensors due to several limitations. They follow a sampling rate to capture the whole sensory data. This affects the data retrieved in 2 ways: events in the sensory scene that would happen at a sub-frame level cannot be captured and the absence of events or slow events lead to highly redundant data. Any global change of the sampling rate will affect both aspects in an opposite manner. Another consequence of the complete sensory data being outputted at every sampling is the quantity of data to be processed. Using Dynamic sensors, the output events are directly linked to input events: pixel in Dynamic Vision Sensors will emit multiple events as they perceive luminance changing rapidly. Using this mechanism, sub-frame information can be retrieved and the absence of input events that would cause redundancy with Legacy sensors will lead to no event to be produced.

We showed during this Ph.D. that, using the right coding strategy, we can process the relevant information from such a stream of events, using a low number of operations per second. Moreover, we exhibited specialized hardware support for such an approach for which the complexity overhead does not create energy consumption overhead compared to SoA chips.

However, there are still points to explore. Further gains can be retrieved from designing a chips containing the full Globally Asynchronous Locally Synchronous architecture depicted in chapter 3 as the power consumption of the Proof of Concept shown in Chapter 4 is dominated by the dynamic power of the scheduling and routing modules. Moreover, such system would be a good candidate for local Dynamic Voltage and Frequency Scaling as the activity of each cluster is not correlated and known in advance. Another source of gain available hardware-wise is the conversion of the generic cluster into a mixed signal im-

plementation with analog neurons and synapses, lowering even more the energy required to simulate the network. However, the timing and diversity constraints generated by the topologies would not make this implementation an easy task.

By developing methods for conversion from Artificial to Spiking Neural Network, we retrieve some of the efficiency achieved by Machine Learning and make it possible for Spiking solutions perfectly fitting Dynamic Sensors to reach such efficiency. It led to the first implementation of a Spiking Recurrent Neural Network using temporal coding and can lead to other implementations not studied in this paper. The exploration around MNIST initially proves the conversion capabilities for Convolutional Neural Networks, but does not make for a concrete application, as it would make no sense to use Dynamic Sensors on completely static inputs. The robustness of the conversion method and pre-processing was further proved using data from N-TIDIGITS18 and Recurrent Neural Networks. However, this database is small and despite containing noise artefacts does not represent real applications yet.

The challenges for next years will be adapting to plurality of new Dynamic Sensor implementations and use the developed techniques in more practical environments. As the sensors' specification change, the output time resolution, data rate and nature of the event can change. To cope with those new challenges, we produced multiple tools enabling fast SNN and hardware design. The operators and conversion methods use Xnet, part of N2D2 in C++, and the hardware model for generic cluster sizing and evaluation is implemented in SystemC, C++ too. A designer can use both tools to implement topologies suiting application need and use the obtained network and sensor activity to simulate hardware usage before implementing it.

Appendix A

List of Publications

A.1 Conference papers

1. **Impact of the AER-induced timing distortion on Spiking Neural Networks implementing DSP.** Thomas Mesquida, Alexandre Valentian, David Bol and Edith Beigne. In *2016 12th Conference on Ph.D. Research in Microelectronics and Electronics (PRIME)*.

2. **Architecture exploration of a fixed point computation unit using precise timing spiking neurons.** Thomas Mesquida, Alexandre Valentian, David Bol and Edith Beigne. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. Mesquida Thomas, Causo Matteo, Valentian Alexandre, Sicard Gilles, Morche Dominique and Beigne Edith

3. **Artificial to Spiking Neural Network Conversion Using Precise Timing Framework and Event Driven Architecture.** Thomas Mesquida, Matteo Causo, Alexandre Valentian, Gilles Sicard, Dominique Morche and Edith Beigne. In *2018 4th International Conference on Event-Based Control, Communication and Signal Processing (EBCCSP)*.

A.2 Journal papers

1. **Spiking Neural Networks Hardware Implementations and Challenges: a Survey.** Maxence Bouvier, Alexandre Valentian, Thomas Mesquida, Francois Rummens, Marina Reyboz, Elisa Vianello, Edith Beigne. In *ACM Journal on Emerging Technologies in Computing Systems (ACM JETC), Special Issue on Hardware and Algorithms for Energy-Constrained On-chip Machine Learning.*

Appendix B

Proofs

In this section, we exhibit functionality and proofs for the topologies shown at the end of Chapter 2. Neuron B is used to start the operation, neuron S recalls the stored value with a linear weight corresponding to the number of consecutive intervals used.

B.1 Variation mitigation

We here show chronogram and proof for the averaging operator exhibited in Fig.B.1.

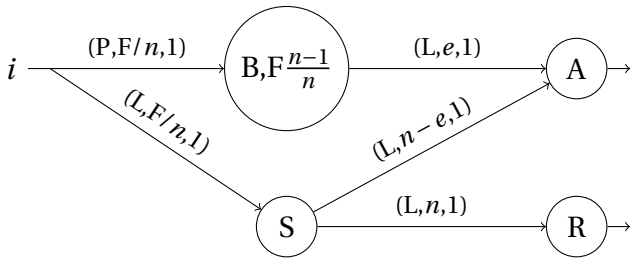


Figure B.1: Operator used to average n consecutive intervals.

Fig.B.2 shows the chronogram for the described operator with 4 inputs. As stated in chapter 2, the neuron (B) needs to be pre-charged to trigger as the first spike enters the operator. Let M be the value that was stored in the membrane potential of A when the neuron (S) starts recalling the result. The created output interval is defined by the respective times at which (A) and (R) emit a spike. This difference is created by the difference of membrane potential and the weight used to recall the stored value.

The potential difference is converted to a time difference using the weight n. Essentially, the neuron (A) will spike $(F-M)/n$ time steps after being recalled and neuron (R) will spike F/n time steps after being recalled. The output interval is thus defined by

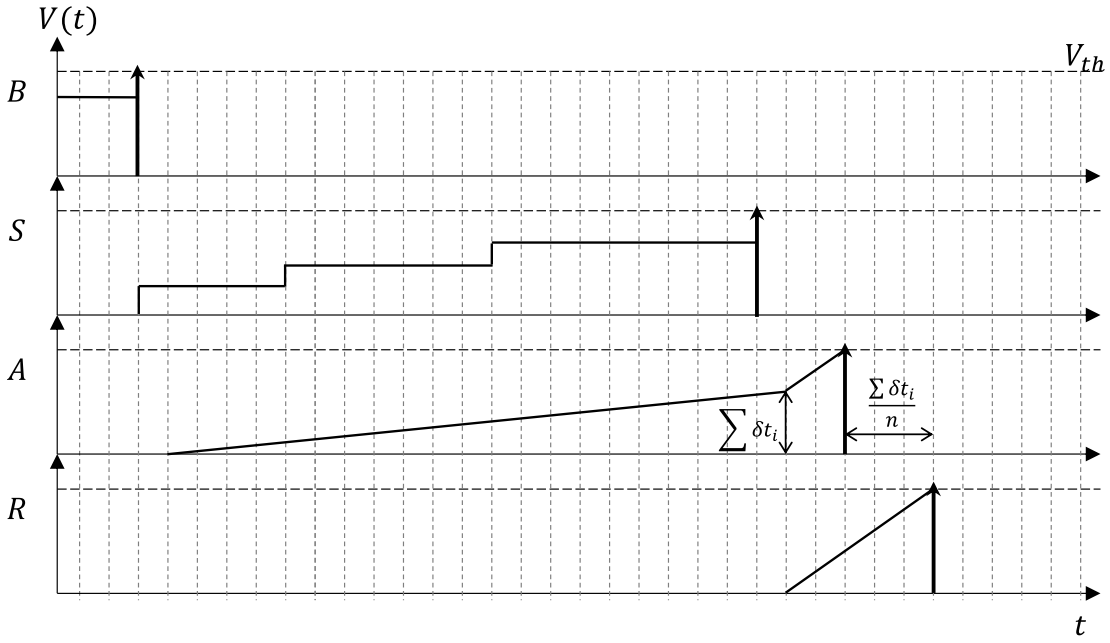


Figure B.2: Chronogram for averaging 4 intervals.

F/n – (F – M)/n which approximates M/n.

B.2 Overflow flag

For flagging overflow while computing addition, multiplication or any other operation requiring accumulation of intervals, we need to set a condition for the output events to be transmitted. Fig.B.3 shows the topology used here for flagging on overflow.

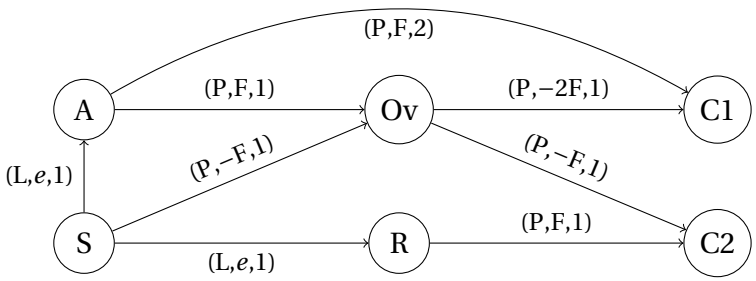


Figure B.3: Overflow detection topology.

We exhibit two chronograms focused on these neurons and the neuron (S) and (A) creating the condition. Fig.B.4 shows (A) spiking before (S) and the flag being transmitted. The overflow flag inhibits the output neurons (C1) and (C2), preventing false results to be transmitted. Fig.B.5 shows normal conditions with (S) spiking before (A) and the output signal being transmitted. This operator is secure and will transmit overflow signal if reached.

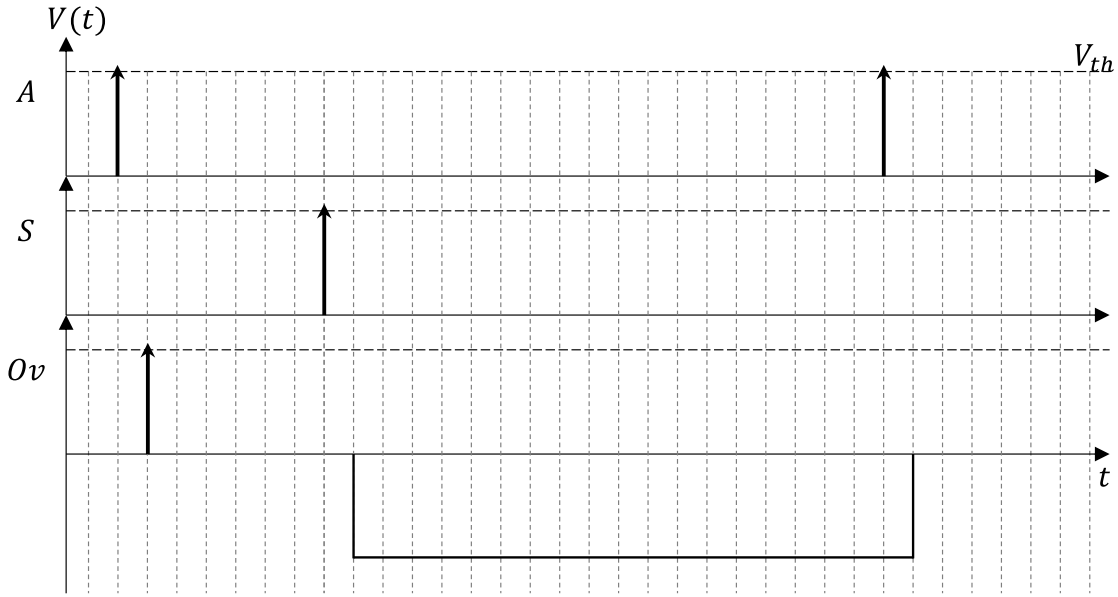


Figure B.4: Overflow chronogram: flagging on overflow.

This topology does not treat instances in which (A) and (S) would spike at the same simulation step. This case should not be able to happen and would still be considered as overflow. Fig.B.6 shows a topology in which the neuron (C) spikes if (A) and (S) emit a spike during the same simulation step. The neuron (C) essentially acts as a coincidence detector. Both (A) and (S) will half charge the neuron (C) for 1 simulation step only and if both half

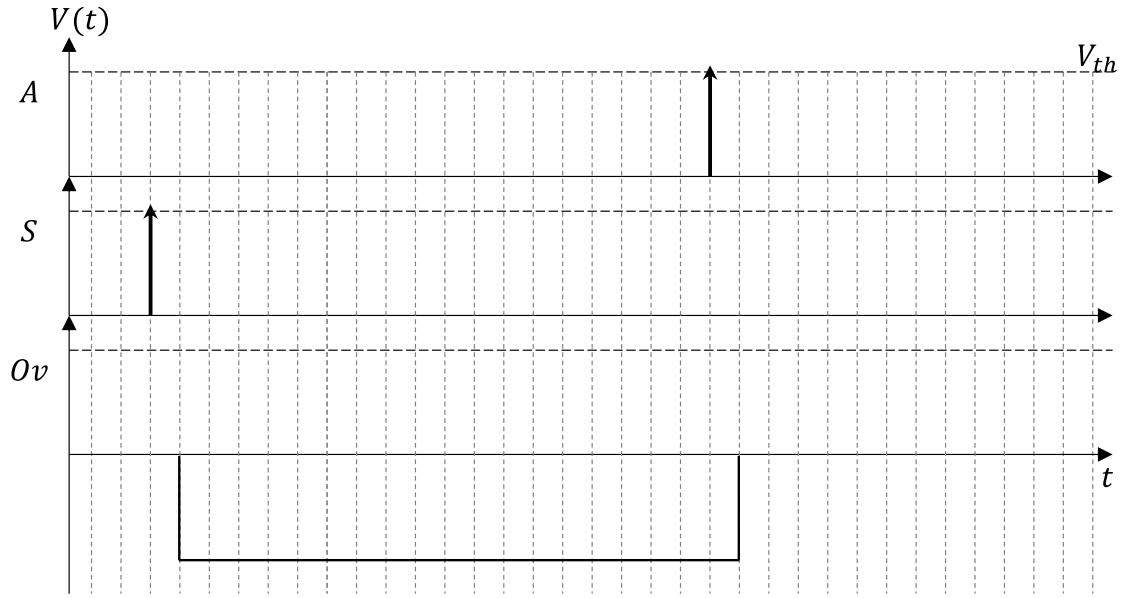


Figure B.5: Overflow chronogram: normal operation mode.

charges happen at the same time then (C) is triggered. The link from (C) to itself is used when (C) is triggered to compensate for the inhibitions to come from A and S (+Full -Half -Half).

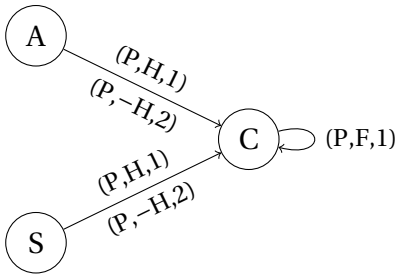


Figure B.6: Coincidence detection topology.

B.3 Delay mitigation

The delay mitigation implementation is based on the fact that the result delay is defined by the precision of the operation: when the result is recalled, the output neurons often have a linear synapse with elementary weight activated leading to delays proportional to the output precision required.

This delay can be mitigated for any precision by splitting the input intervals and using parallel operators for the different result ranges. The accumulation neurons for each operator use similar logic to the one in B.3 to filter local carries to upper operators as simplified in Fig.B.7.

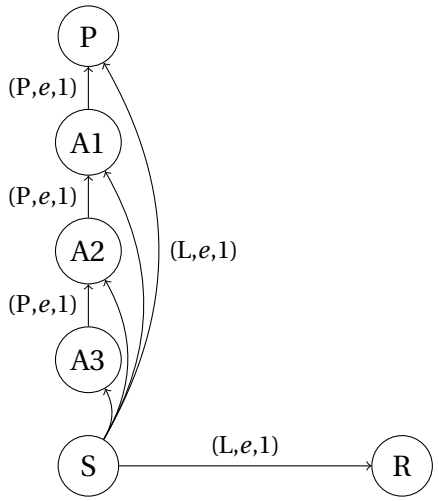


Figure B.7: Carry in between linear combination operator 1,2 and 3. Carry from 1 goes to neuron P for 4N bits result. S and R are common in between the 3 linear combinations.

The logic used to filter carry from actual result is shown in Fig.B.8. The neuron (Ca) transmits the carry as long as the synchronization neuron (S) does not spike. When (S) spikes, (Ca) is inhibited and (T) (Transmit) can be triggered to retrieve the stored result.

For these operators, the accumulation neurons are set to not reset the value of their synapses when spiking. Thus, each accumulation neuron will spike as many times as its current result requires. The output filtering logic is the same as the one defined for the overflow flagging: carries will be transferred as long as the Synchronization neuron did not spike. When the synchronization neuron emits a spike, values in each accumulator is recalled. In order to retrieve coherent result in signed operations using this method, the sign of the result have to be checked and sub-results changed accordingly.

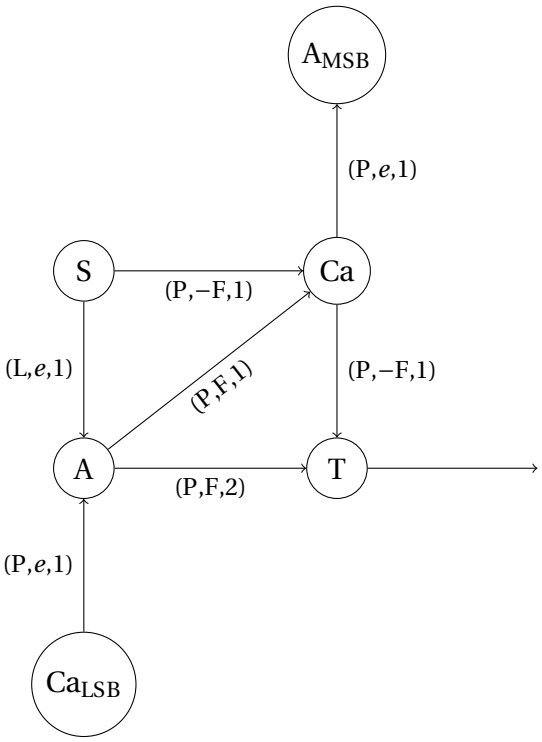


Figure B.8: Logic used to transmit carries as long as the synchronization neuron S does not spike.

Appendix C

List of acronyms

AER Address Event Representation. 7

ANN Artificial Neural Networks. 10

ANOC Asynchronous Network On Chip. 64

ASIC Application Specific Integrated Circuit. 72

CABA Cycle-Accurate, Bit-Accurate. 82

CAD Computer-Aided Design. 112

CAM Content-Addressable Memory. 67

CNN Convolutional Neural Networks. 15, 72, 109, 110

DAS Dynamic Audio Sensors. 6

DS Dynamic Sensors. 6

DSP Digital Signal Processor. 48

DVFS Dynamic Voltage and Frequency Scaling. 64

DVS Dynamic Vision Sensors. 6

FC Fully Connected. 72

FDSOI Fully Depleted Silicon On Insulator. 92

FIFO First In, First Out. 67

FPGA Field-Programmable Gate Array. 117

FSM Finite State Machine. 93

GALS Globally Asynchronous Locally Synchronous. 63

GPU Graphics Processing Unit. 17

GRU Gated Recurrent Unit. 120

IF Integrate and Fire. 31

LIF Leaky Integrate and Fire. 31

LSB Least Significant Bit. 53

LSTM Long Short Term Memory. 120

MAC Multiply ACcumulate. 17

MLP Multi Layer Perceptron. 10

MSB Most Significant Bit. 57

NN Neural Networks. 10

PD Power Domain. 98

POC Proof Of Concept. 92

PT Precise Timing. 31

PTT Path To Target. 65

ReLU Rectified Linear Unit. 10

RNN Recurrent Neural Networks. 119

RTL Register Transfer Level. 64

SDSP Spike Driven Synaptic Plasticity. 14

SNN Spiking Neural Networks. 12

SPI Serial Peripheral Interface. 97

SRAM Static Random Access Memory. 69

STDP Spike-Timing-Dependent Plasticity. 11

TLM Transaction-Level Modeling. 82

TP Tick Parity. 66, 81

TTFS Time To First Spike. 16, 117