Massively Parallel FPGA Hardware for Spike-By-Spike Networks

David Rotermund and Klaus R. Pawelzik

December 18, 2018

University of Bremen, Institute for Theoretical Physics, Bremen, Germany

Correspondence: davrot@neuro.uni-bremen.de; Tel.: +49-421-218-62003

1 Abstract

The increase in computational power by faster computers as well as application specific integrated circuits (ASICs) lead to re-consideration of neuronal networks for pattern recognition and other applications in artificial intelligence (AI). Fueled by this technological progress, large improvements were made, up to the advent of self-driving cars. Often the underlying deep neuronal networks are derived from the many decades old multi-layer preceptron. While inspired by the brain, technical applications lack many key features of the biological original. For example the exchange of information among neurons by electrical pulses (spikes) is missing. On the other hand, approaches using spikes in neuronal networks typically are the more computationally demanding, the more realistic they get. Among the less demanding models, there is a network type called Spike-By-Spike (SbS) network that realizes a compromise between machine learning approaches and biologically realistic models. In this network type computations occur onlyif an incoming spike is detected by a group of neurons. Even though the underlying equations are simple, such a network – simulated on a normal computer – is several orders of magnitude slower than multi-layer perceptron based deep network with comparable size and task. One of the main reasons lies in large number of memory accesses for intermediate results during the calculations, which can be reduced if specialized computational pipelines are used. The other main reason is that a commercial CPU has only up to several dozens of cores. Thanks to the communication via spikes, already a small deep network based on SbS type neurons allows a parallelization into thousands of fully independent and simple computational cores. While running a deep SbS network on a normal processor takes a lot of time, it would thrive on an ASIC with many optimized computational SbS cores. In particular, larger networks composed of many such ASICs have the potential to out-compete even the largest presently available supercomputers.

In this paper we develop and investigate a framework as well as these computational SbS cores for a network on chip. As long term goal, an implementation with as many as possible SbS cores on an ASIC is envisioned. Here we demonstrate the feasibility of our design on a Xilinx Virtex 6 FPGA while avoiding all types of Xilinx proprietary cores (except block memory) that can not be realized on a custom-designed ASIC. We present memory access optimized circuits for updating the internal variables of the neurons based on incoming spikes as well as for learning the connection's strength in between neurons. The optimized computational circuits as well as the representation of variables fully exploit the non-negative properties of all data in the SbS network and compare the sizes of the arising circuits for floating and fixed point numbers. In addition we show how to keep the number of components required for the computational cores small by reusing their components for different functions. Furthermore we present the necessary supporting modules required for operating the network in an autonomic fashion on a chip.

2 Introduction

Nowadays, deep neuronal networks ([1]) are the basis for successfully applying neuronal networks on problems from artificial intelligence research ([2, 3, 4, 5]). The revival of using neuronal networks was provoked by the increase of computational powers in modern computers and boosted even more through modern 3D graphic cards as well as specialized application specific integrated circuits (ASICs) and field programmable gate arrays (FPGAs) ([6]). The most successful type of networks is based on multilayer perceptrons ([7, 8]) and consists of several so called hidden layers. Typically information is processed and feed forward from one hidden layer to the next, beginning at the input layer and ending at the network's output layer. In theory such a network is able to calculate arbitrary functions. However, for doing so the weights – describing the connection of elements of one layer to the next – must be learned based on the intended task. During learning these weights, the error between the desired outcome and the actual outcome of the network is minimized. This leads to the question which of the weights need to be modified for improving the performance of the network (called credit assignment problem). While there are other learning algorithms ([9]), the most widespread algorithm for finding suitable weights is called Error-Backpropagation (in short backprop; [7]). With backprop, the error on the output layer is propagated back in the direction of the input layer and the weights are changed based on this error.

Another recent discovery in the field of machine learning is compressed sensing ([10]), which allows to reconstruct underlying causes from incomplete measurements if the underlying causes are sparse. This lead to applications in many fields from reducing measurement time for Magnetic Resonance Imaging ([11]) to building swarms of robots for efficiently exploring other planets ([12]). Sparseness and its benefits for information processing is also an important topic in brain research ([13, 14, 15]). For the present context interesting is the finding that using non-negative representations ([16, 17]) in a network can be sufficient to induce some degree of sparseness.

In [18] we presented a new type of neuronal network (called Spike-by-Spike network, SbS) based on the family of generative models ([19, 20, 21, 22]) where its neurons exchange information via stochastic spikes. The SbS network lies in between the normal perceptron based network and typical spike-based networks ([23]). The latter ones use neuron models ([24]) that simulate its neurons in the time domain with fine resolution. In the SbS network time is only progressed with each incoming spike. Thus no computations have to be performed in between spikes, which drastically reduces the computational demand. Furthermore all entities in SbS networks are described by positive numbers which leads to sparse representations. In [**?**] we extended our old approach to a deep network.

In a SbS network, each neuron i owns a latent variable $h(i)$ which can have values in the range of $0 \leq h(i) \leq 1$. Neurons are organized in groups and a competition exists between all neurons within such a group. The competition is mediated via a common normalization ($\sum_i h(i) = 1$) on all the $h(i)$ of one group. Beside the normalization, there is no information exchanged inside a group. However, information is exchanged via spikes to connected other groups. A spike s^t is an index that describes which neuron in the group is responsible for this spike. Algorithmically, this translates in drawing a random number $R \in [0, \ldots, 1]$, comparing R to the cumulative sum $C(i) = \sum_{j=1}^{i} h(j)$ and selecting the i as spike s^t for which the cumulative sum $C(i)$ surpasses R. Spikes are sent to connected groups. Every neuron i of the receiving group has weights $p(s|i)$ (with $p(s|i) \in [0, 1]$ and $\sum_s p(s|i) = 1$) to spike emitting neurons which it is connected to (enumerated by s). [18, **?**] give us equations for updating the latent variables $h(i)$ as well as learning rules to extract weights $p(s|i)$ from a set of training data (in the following these equations will be introduced and analyzed.).

While this type of network is far less computationally demanding than networks with more detailed neuron models, it still requires high computational effort, especially if it is compared to a perceptron-based deep neuronal network. Thus simulating a SbS network, in particular deep ones for large data sets, is a problem when only normal computer CPUs or even computer clusters are available. Furthermore, the operations (e.g. the way how memory needs to be accessed) for the equations are not well suitable for graphic cards. However, every single group in a SbS network is a compact local module that only communicates via spikes with its environment. This allows to parallelize the whole network into arbitrary many parallelly operating modules.

We investigated how the algorithm, described in [18] and extended into multi-layer networks in [**?**], can be

realized as an ASIC. We tested the resulting VHDL implementation with Xilinx Virtex 6 LX550T-1FF1759 FPGA on 4DSP (now abaco systems) FM680 cards. The long term goal is to use this design for an ASIC, thus all FPGA specialized circuits (e.g. hardware DSP cores) or other IP cores were not used. The only exception to this rule is block RAM (BRAM).

3 Materials and methods

3.1 Design goals

There are multiple ways to realize the required circuits. For example, the number of components necessary for the arithmetic units can be minimized by using sequential processing as well as reusing the arithmetic units as much as possible. Another approach would be to design the circuits such that the calculations are done as fast as possible (measured in clock cycles) under the penalty of using much more components in the process. The design goal for the results of this study is leaning strongly in the latter direction by using pipeline designs for the arithmetic units. The reason for this decision lies in the low clock speeds. Compared to commercial CPUs, we can ensure with FPGAs or ASIC combined with the desire to implement a fast calculation.
For implementing a flexible multi-layer Spike-by-Spike network on chip, the following main ingredients are required:
a) SbS groups: Each such group consists of a population of neurons that are in a competition which is realized by normalization. A normalization group receives spikes as indices s of sending groups and uses them to update internal states $h(i)$ of each group member i and weights $p(s|i)$ as well as produce outgoing spikes which reflect its internal latent variable $h(i)$.
b) input groups: A population of neurons that are programmed with an input distribution (e.g. the pattern of an input picture) which is used to produce spikes according to this input distribution. The input distribution is set constant during processing. The input groups don't react to any spikes produced in the network and thus can be considered simplified versions of the normalization group.
c) network communication fabric: Spikes need to be exchanged between groups.
Focusing on the calculations that a SbS group needs to perform, a set of equations have to be realized: The basic equation realizes the update dynamic for the latent variables $h(i)$ based on an incoming spike. In [18] it was shown that only the identity (i.e. the index) s^t of the subsequently active neuron needs to be taken into account:

$$h^{t+1}(i) = \frac{1}{1+\epsilon}\left(h^t(i) + \epsilon \frac{h^t(i)p(s^t|i)}{\sum_j h^t(j)p(s^t|j)}\right). \tag{1}$$

Besides updating the latent variables $h(i)$, it is necessary to optimize suitable weights $p(s|i)$ from training data for allowing the network to perform the desired function (e.g. pattern recognition). Two different approaches were found useful for learning weights. One is based on changing weights based on only single spikes observed during processing the actual pattern. This procedure is called online learning. The other approach utilizes information gathered over many spikes and several patterns, which is called batch learning.
For online learning we focus on a multiplicative learning rule for the weights $p(s|i)$:

$$p^{t+1}(s|i) = \frac{p^t(s|i) + \gamma \frac{h^t(i)p(s^t|i)}{\sum_j h^t(j)p(s^t|j)}\delta_{s,s^t}}{1+\gamma \frac{h^t(i)p(s^t|i)}{\sum_j h^t(j)p(s^t|j)}} \tag{2}$$

where γ is a learning parameter which can change during learning.
For batch learning, a variety of implementations exits, we focus on batch learning rules that base on

$$\Delta p(s|i) = \sum_\mu \frac{h^t_\mu(i)\hat{p}_\mu(s)p(s|i)}{\sum_j h^t_\mu(j)p(s|j)} \tag{3}$$

where μ identifies the training pattern and where the sum may run over the whole or only parts of the complete set of training patterns. $\hat{p}_\mu(s)$ denotes the input probability distribution of the incoming spikes into the SbS group and is approximated by analyzing (counting) the spikes processed by the SbS group. For allowing a more flexible implementation of these type of rules, $\Delta p(s|i)$ is handed over by the FPGA to a CPU, which allows to implement batch learning rules based on

$$p^{new}(s|i) = \mathscr{U}\left[p(s|i), \Delta p(s|i)\right] \tag{4}$$

A simple example for an update rule falling into this category is

$$\begin{aligned} p^{new'}(s|i) &= (1-\alpha)\, p(s|i) + \alpha \sum_\mu \frac{h^t_\mu(i)\hat{p}_\mu(s)p(s|i)}{\sum_j h^t_\mu(j)p(s|j)} \\ p^{new}(s|i) &= \frac{p^{new'}(s|i)}{\sum_r p^{new'}(r|i)}\,. \end{aligned} \tag{5}$$

$\mathscr{U}$ could also be realized by Adam ([25]) or L4 ([26]) using mini-batches. Since $\Delta p(s|i)$ is based on a multitude of patterns, offloading $\Delta p(s|i)$ to a separate CPU and performing the weight update on the CPU occurs at a lower rate than all other operations. Thus the reduction in performance by using a CPU for handling the weight updates is outweighed by the gain of flexibility.
Furthermore, we need the means to generate spikes from probability distributions or the latent variables using random numbers. Also specialized circuits are required for observing the spikes entering a SbS group and calculating the corresponding probability distribution $\hat{p}_\mu(s)$ from these observations.

3.2 Non-negative numbers

Investigating the three main equations 1, 2 and 3 reveals that no subtractions are required and that no negative numbers appear. Furthermore most numbers (especially $h(i)$ and $p(s|i)$) are in the range of $[0, ..., 1]$. Incorporating these facts into the arithmetic units allows simplify to the usual designs ([27]).
Before creating the design, we examined different versions of representation for these non-negative numbers through Matlab simulations, using custom mex C extensions. Reanalyzing the MNIST network from [**?**], we found that 16 bit fixed point numbers for latent variables $h(i)$ and weights $p(s|i)$ can be a viable choice. In this case, we represent non-negative real numbers for the value range $[0, ..., 1]$ via 16 bit integers through $X = \frac{X_{Int}}{2^{16}-1}$. For preventing loss of accuracy, we extended the number of bits for intermediate results. For SbS groups with 32 and 64 neurons this approach worked for the MNIST network. However, the dynamic range is too small for groups with significant larger number of neurons, like e.g. 1024 where learning of the weights failed. In the case of a fully connected layer with 1024 neurons in the MNIST network, we unsuccessfully tried to solve this problem by increasing the number of bits from 16 to 32. Instead of increasing the number of bits further, we also tested 32 bit floating point number (IEEE 754) in Matlab simulations. These tests ran successfully. Overall we didn't notice a difference in classification performance between 32 bit floating point (single precision) and 64 bit floating point (double precision) in the Matlab simulations. Since all information is exchanged via spikes, floating point number and fixed point numbers groups can be mixed within a network.
Taking the memory structure of the Xilinx Virtex 6 FPGA into account, where the block memory is organized into blocks of multiples of 1024 words with 18 bits each, we decided to used a custom variant of 36 bit floating point numbers as well as 18 bit fixed point numbers ($X = \frac{X_{Int}}{2^{18}-1}$).
In more detail, we designed the 36 bit floating point numbers as follows: Since only non-negative numbers are used, the usual sign bit was not necessary anymore. As part of batch learning, calculations on a typical CPU occur. This requires an easy way to convert our floating point numbers into their IEEE 747 counterpart. Thus we keep the number of bits for exponents, removed the sign bit and appended 5 extra bits to the lower significant bits of the mantissa. This allows conversion to be performed just by removing the not required bits of the mantissa or filling the extra bits of the mantissa up with zeros. We also introduced a similar derivative for 64 bit floating numbers (double precision) with 72 bits. Here 9 additional bits were added to the mantissa. These

72 bit floating point numbers are only used for representing $\Delta p(s|i)$ because summing over the contributions from larger amounts of patterns may otherwise lead to a degeneration of precision. For this reason, a similar extension from 18 bits to 36 bits was done for the fixed point numbers.

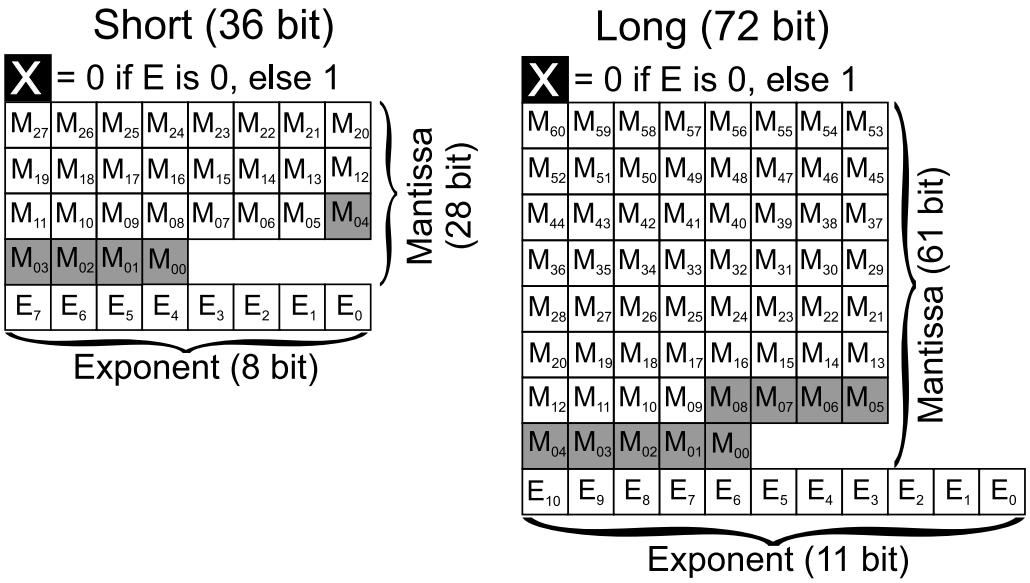

Figure 1: Representation of custom non-negative floating point number with 36 bits and 72 bits. The most significant bit of the mantissa is shown as X in the black box and is not stored in memory. X is set to 0 if the exponent is zero and set to 1 otherwise. The bits shown in gray are the extension to the precision for the mantissa over the IEEE 747 standard for single or double precision.

Figure 1 shows the coding for the two types of floating point numbers used in the design. The bits marked with gray boxes as well as the not available sign bit are different from the IEEE 747 standard. To simplify the floating point arithmetic units even more, only 0 as a sub-normalized number is allowed.

3.3 Analyzing the equation's structure

Analyzing the three equations 1, 2 and 3 reveals that all have common terms. Especially they share the term

$$\Omega(i; s; \zeta) = \zeta \frac{h(i)p(s|i)}{\sum_j h(j)p(s|j)} \,. \tag{6}$$

ζ can be ϵ, γ or $\hat{p}(s)$. All three variants of ζ have in common that they change on a slower time scale than i. Also the s in $p(s|i)$ changes slower than i. s can be selected through an observed spike s^t and only changes after all latent variables of that SbS group has been updated. Furthermore, for batch learning s should be selected as the slower changing index because this reduces the amount of times $\hat{p}(s)$ needs to be calculated.

Since memory is a scarce source on FPGAs and ASICs, it is advantageous to recall all the corresponding $h(i)$ and $p(s|i)$ pairs twice from memory during calculating $\Omega(i; s; \zeta)$ while is s fixed. Since the arithmetic operation Y/X is known to be computational demanding, the following approach was chosen:

0.) All $h(i)$ and $p(s|i)$ pairs are streamed through a multiplication pipeline, thus making $h(i) \cdot p(s|i)$ available at the same speed with which they are recalled from the memory. However $h(i) \cdot p(s|i)$ is only available after a fixed delay.

1.) During the first sweep through the $h(i)$ and $p(s|i)$ pairs, $\sum_j h(j)p(s|j)$ is calculated by observing the output from the multiplication pipeline. In the case of floating point numbers, we found that it is beneficial to implement the summation operation via an addition pipeline, where the output of the addition pipeline is feedback to its input (this procedure will be explained in detail later).

2.) After calculating $\sum_j h(j)p(s|j)$, $\frac{\zeta}{\sum_j h(j)p(s|j)}$ is calculated in a sequential fashion. This keeps the required amount of components for this arithmetic operation lower while a pipeline approach wouldn't lead to a faster calculation anyhow.

3.) Finally, in a second sweep the $h(i)$ and $p(s|i)$ pair are recalled from memory. After the first multiplication pipeline which produces $h(i) \cdot p(s|i)$, a second multiplication pipeline is placed. The latter one multiplies $\frac{\zeta}{\sum_j h(j)p(s|j)}$ with $h(i) \cdot p(s|i)$ which results in $\Omega(i; s; \zeta)$.

For reducing the amount of overall delay, it is beneficial to start step 3.) some time before step 2.) is fully complete. The timing has to selected such that $\frac{\zeta}{\sum_j h(j)p(s|j)}$ is just ready when the first $h(i) \cdot p(s|i)$ result leaves the first multiplication pipeline.

For the implementation of online learning, it is self-evident that the results from the circuits for step 0.) and 1.), calculated during the update of the latent variables, should be reused. Adding an additional sequential division unit and an additional multiplication pipeline allows to calculate step 2.) and 3.) fully in parallel to the ongoing update of the latent variables.

Since the calculation of $\Omega(i; s; \zeta)$ of the batch learning update for the actually processed pattern isn't done at the same time as the update of the latent variable, it is beneficial to use the same circuits for these two tasks.

4 Results

4.1 Computational building blocks

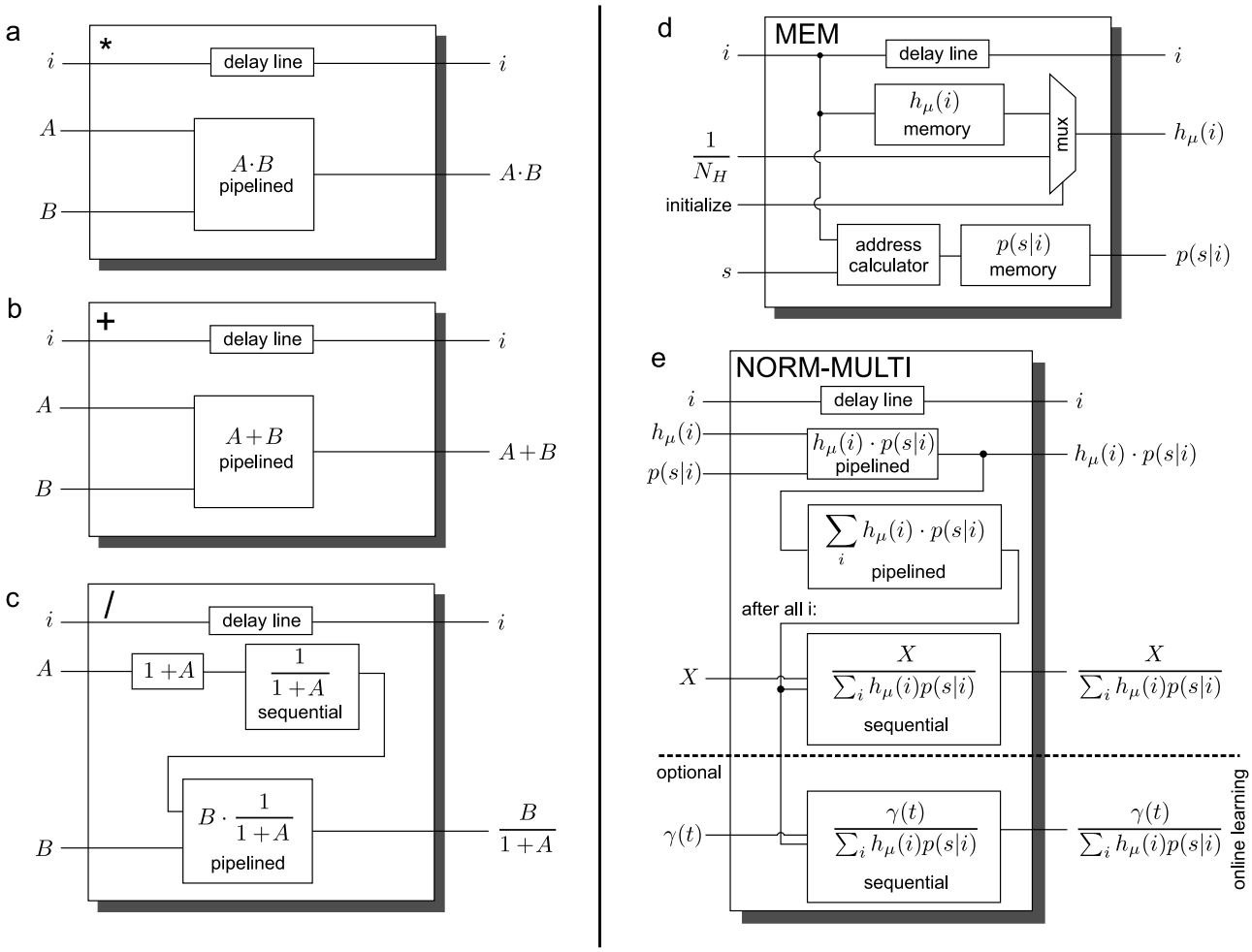

Figure 2: Basic computational building blocks: a.) **Module** $*$, b.) **Module** $+$, c.) **Module** $/$, d.) **Module MEM**, and e.) **Module NORM-MULTI**.

Five basic building block can be identified for implementing the three equations. All five modules have in common that input entering the module is accompanied by an organizational index i. Delay lines in the modules ensure that this index labels its corresponding output. Figure 2 shows an overview of the computations performed in these modules. Table 1 lists the components required to map the circuits onto Xilinx Virtex 6 FPGA hardware. Resources listed are Slice Registers (687360 available on the used LX550T version), Look-Up-Tables (LUT, 343680 available), LUT Flip Flop pairs used (85920 available), and block Random-Access Memory (BRAM, 1264 blocks with 18k bits each or organized as 632 blocks with 36k bits). Every type of these resources represent highly complex circuitry (see Xilinx user guide UG364 'Virtex-6 FPGA Configurable Logic Block Table' for more details). Every pipelined operation is designed such that one new operation can be started per clock cycle. However, it typically takes more than one clock cycle (also called latency) before the result of an operation leaves its pipeline. Furthermore some modules contain sequential components that need a given number of clock cycles before they are initialized for their task and can perform their actual computation. Table 2 gives details concerning the latencies or how long the operations take.

Module $*$ (figure 2a) represents a simple multiplication pipeline. Two inputs A and B enter the module and the result $A \cdot B$ leaves the module. In the case of fixed point numbers, the output has twice the number of bits compared to the input.

Module $+$ (figure 2b) represents a simple adding pipeline. Two inputs A and B enter the module and $A + B$ leaves the module. In the case of fixed point numbers, the output has one bit more than the input.

		Floating point	Fixed point
Module $*$	LUT	1425	442
	LUT-FF pairs	1674	614
	Slice registers	1844	676
Module $+$	LUT	433	20
	LUT-FF pairs	358	20
	Slice registers	429	0
Module $/$	LUT	2407	1333
	LUT-FF pairs	2594	1409
	Slice registers	2992	1750
Operation $1/X$ seq.	LUT	301	197
	LUT-FF pairs	304	311
	Slice registers	213	180
Module MEM	LUT	383	345
	LUT-FF pairs	728	562
	Slice registers	816	661
	BRAM $h(i)$	1x 36k	1x 18k
	BRAM $p(s\|i)$	1x 36k	1x 18k
Module NORM-MULTI	LUT	3306	1345
	LUT-FF pairs	3916	1501
	Slice registers	4349	1503
Operation $\sum$	LUT	452	48
	LUT-FF pairs	496	60
	Slice registers	569	41
Operation Y/X seq.	LUT	426	322
	LUT-FF pairs	457	324
	Slice registers	340	227

Table 1: Basic computational building blocks: Components used for the building block. All addresses are represented by 10 bit integer values. The BRAM counts for **module MEM** are given for up to 1024 $h(i)$ values and up to 1024 $p(s|i)$ values. For the fixed point modules, 18 bit integers are used as input.

Module $/$ (figure 2c) computes $B \cdot \frac{1}{1+A}$, where A is assumed to be changing on a slower time scale than B. For a newly given A, first $1 + A$ is calculated and then $\frac{1}{1+A}$ is determined in a sequential fashion. This result is used as one factor for a multiplication pipeline, while the fast changing B is the other factor. For fixed point numbers, the result of $1 + A$ increases by one significant bit over A and the reciprocal operation doubles the number of bits again. The multiplication pipeline doubles the number of bits a second time, before the results exiting the module is cut down to the original number of bits for A.

Module MEM (figure 2d) stores the latent variables $h(i)$ and the corresponding weights $p(s|i)$. The main components of this module are two **block RAM modules** (which can have different sizes), which provide a vector of memory values each. For each of the **block RAM modules**, only one reading operation and one writing operation can be done in parallel during one clock cycle. While the access of the $h(i)$ memory could be done directly, for $p(s|i)$ the memory location needs to be calculated via an address calculator from i and s first. Thus the module contains two address calculators, one for writing and one for reading operations. Since the calculation of the memory address takes some clock cycles (one multiplication and one addition) and the output of the module are pairs of $h(i)$ and $p(s|i)$ based on the same i, read requests to $h(i)$ memory are appropriately delayed. Figure 2d shows in a simplified fashion the reading part of the module, for which a multiplexer is placed at the output of the $h(i)$ memory. This multiplexer allows to usurp the $h(i)$ memory output by an initialization value without writing it into $h(i)$ memory first. For floating point numbers, at the input and output of the module, the first bit of the mantissa is removed or re-added (see figure 1). This bit is necessary for calculations but is directly defined by the exponent and hence doesn't need to be stored in memory.

Module NORM-MULTI (figure 2e) is a modification of **Module** $*$. In addition to the multiplication functionality, the cumulative sum over the output of the multiplication pipeline is calculated. In case of fixed point numbers, adding two numbers can be done in one clock cycle. However, for floating point numbers it takes several clock cycles for adding two numbers. In combination with receiving one new output from the

		Floating point	Fixed point
Module $*$	clock cycles	13	8 (42bit)
Module $+$	clock cycles	8 (36bit), 13 (72bit)	1
Module $/$			
(calculate $\frac{1}{1+A}$)	clock cycles	47	46 (23bit)
(multiply with B)	clock cycles	14	10 (23bit)
Operation $1/X$ seq.	clock cycles	34	39 (24bit)
Module MEM (and $W(S\|i)$)	clock cycles	14	14
Module NORM-MULTI			
(ϵ/norm, after last value)	clock cycles	97	61 (ϵ 22bit)
(multiplication pipeline)	clock cycles	15	11
Operation $\sum$ (after last value)	clock cycles	39	4 (22bit)
Operation Y/X seq.	clock cycles	35	45 (22bit)

Table 2: Basic computational building blocks: Number of clock cycles between the input into the module and when the output of the computation is available. For the **module** $\sum$ and where it is used in **module NORM-MULTI**, the number of clock cycles after when the last input values was presented are shown. Since the numbers are measured in circuit, deviations from the 18bit fixed point input into the modules are noted in brackets.

multiplication pipeline in every clock cycle, this poses a problem. As a solution, we used an adder pipeline feedback on itself. The cumulative sum goes through three stages for floating point numbers: For stage 1 and 2, input A into the adder pipeline is defined by the output of the multiplier pipeline. In stage 1, which lasts as many clock cycles as required to pass through the adder pipeline, B are set to 0. Then stage 2 is entered where B is set to the output of the adder pipeline and thus creates a kind of circular buffer. After the last input value from the multiplication pipeline's output was received (defined by an external constant), stage 3 is entered. In stage 3 A and B is set to 0 by default. The output of the adder pipeline is collected until two valid outputs are available. These two values are send as an A-B-pair through the adder pipeline. This is done several times and thus combines more and more remaining pairs until one value is left, which is the desired output of the cumulative sum. Finally, an external value (in figure 2e X is a placeholder for ϵ or $\hat{p}(s)$) is divided by the actual cumulative sum in a sequential way. In the case of online learning the weights it is beneficial to perform a second division with the learning parameter γ in parallel. A note concerning the number of bits for the fixed point case: the cumulative sum can not exceed 1 since $h(i)$ is normalized to 1 and the values of the weights obey $0 \leq p(s|i) \leq 1$. Since ϵ values which are larger than 1 can be interesting for some applications, the design allows ϵ values with up to 22bits (which allows for values up to 16). Thus $\frac{\epsilon}{\sum_i h(i)p(s|i)}$ has 22+18 bits.

Besides the basic computation building blocks for calculating the three main equations, additional modules are required for realizing the network. In particular a module that allows to generate spikes from a probability distribution and a random number, a module that analyzes spikes and calculates a rate $\hat{p}(s)$ out of them, and a module that offsets the weights in a normalized fashion during learning. The figures 3, 4 and 5 show simplified schematics for these modules. Table 3 shows the amount of components necessary to build them as well as the number of clock cycles they require.

Module Spike Generator (figure 3a) converts a probability distribution and one random number into one spike. A spike is an index describing a position in the probability distribution that elicited the spike.

The values of the probability distribution (e.g. $h(i)$ or normalized input pattern) are presented to the module sequentially.

The module sequentially calculates the cumulative sums over the observed part of the probability distribution and compares it to the random number. Is the actual value of the partial cumulative sum equals or is larger than the random number, the index of the probability distribution value that just contributed to the sum is the desired index. If every value of the probability distribution was shown but the last value and no index was found yet, then the wanted index is the last position in the probability distribution. In addition, for this design the values of the probability distribution are converted into a fixed point representation, since adding two fixed point numbers can be done in one clock cycle and the available random numbers are in a fixed point representation

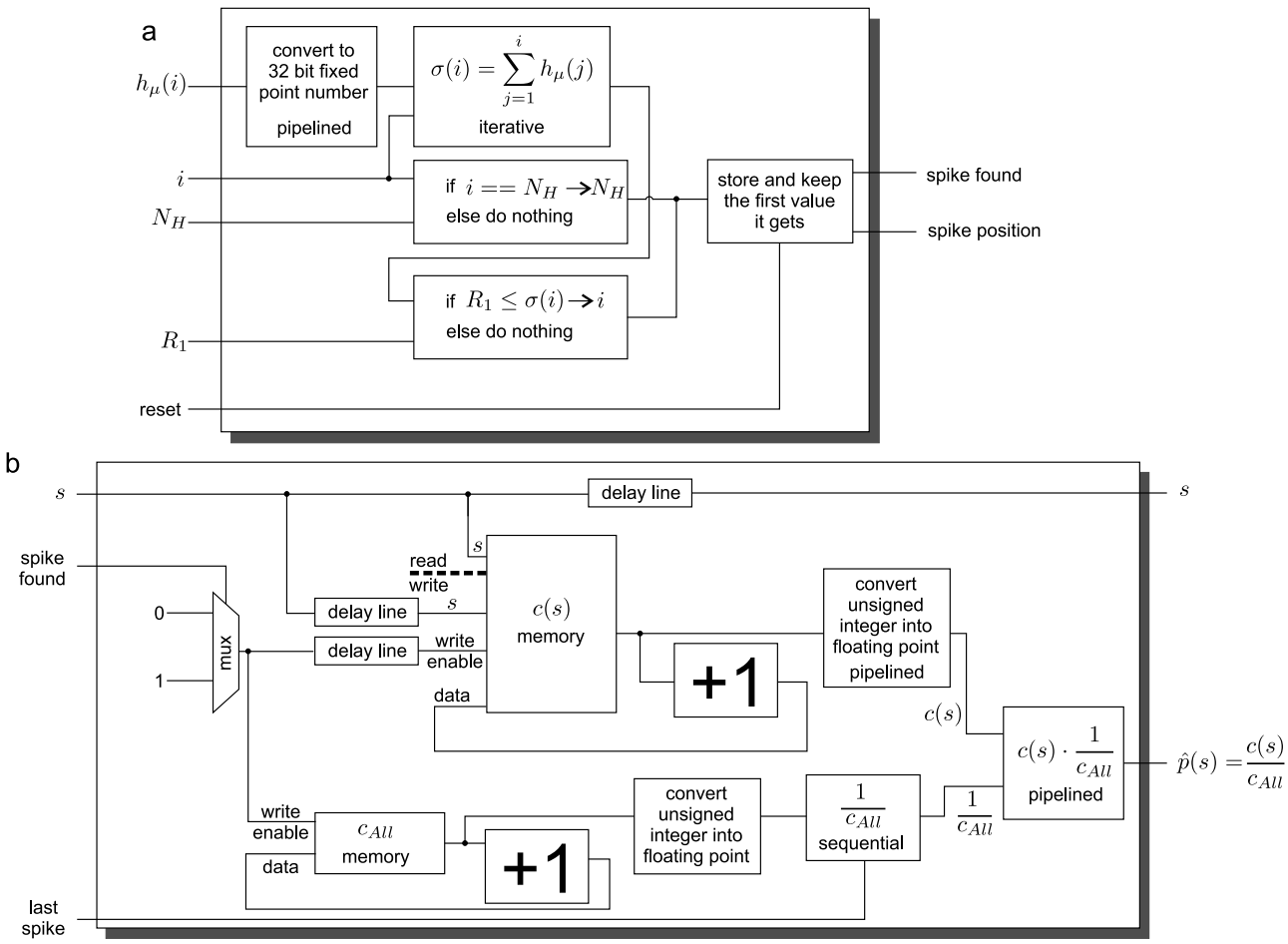

Figure 3: a.) Module for generating spike from a random number and probability distribution. b.) Module for calculating a rate from observed spikes. The conversion from unsigned integer into floating point number is only used in the floating point version.

anyway.

Module Spike Generator with offset (figure 5a): Sometimes, especially during annealing while learning, it is helpful to add an offset α to a probability distribution and to fade out this offset over time. One way to generate spikes with an offset is to add the offset to the probability distribution, re-normalize it and then draw spikes from it. In the context of these circuit designs, a more efficient way is to rely on a double stochastic process using two random numbers. One random number is used to draw one spike from the original probability distribution without offset and one spike from a uniform probability distribution. The second random number is compared to α. The outcome of this comparison decides which one of the two spikes is used.

Random number generator (figure 4): For generating spikes, we need random numbers. For generating those, we use a Mersenne twister (MT 19937) which produces 32 bit random numbers (with a period of 2^{19937}). A requirement for the MT 19937 implementation was that it needs to produce one random number in every clock cycle. Several different designs for such MT circuits are available. We decided to combine the designs from [28] and [29]. Our design uses two block RAM (624x 32bit words each) and 624bit logic RAM as memory. While it uses one more block RAM than such an implementation ultimately needs, it allows us to produce exactly the random number series in the order produced by Matlab or C++. Our **MT modules** also contains a computational unit that calculates the initial table after reset given a seed value.

Module Rate Calculator (figure 3b): For batch learning the weights, the input rate $\hat{p}(s)$ is required. For calculating the rate $\hat{p}(s)$, which is the number of spikes observed in channel s normalized over all channels. Thus the number of spikes for each channel $c(s)$ has to be counted independently as well as the overall number of spikes observed c_{All}. After an externally defined number of spikes has been counted, the rate is calculated and recalled by an external module. This is realized by calculating $\frac{1}{c_{All}}$ first and then multiplying it by $c(s)$.

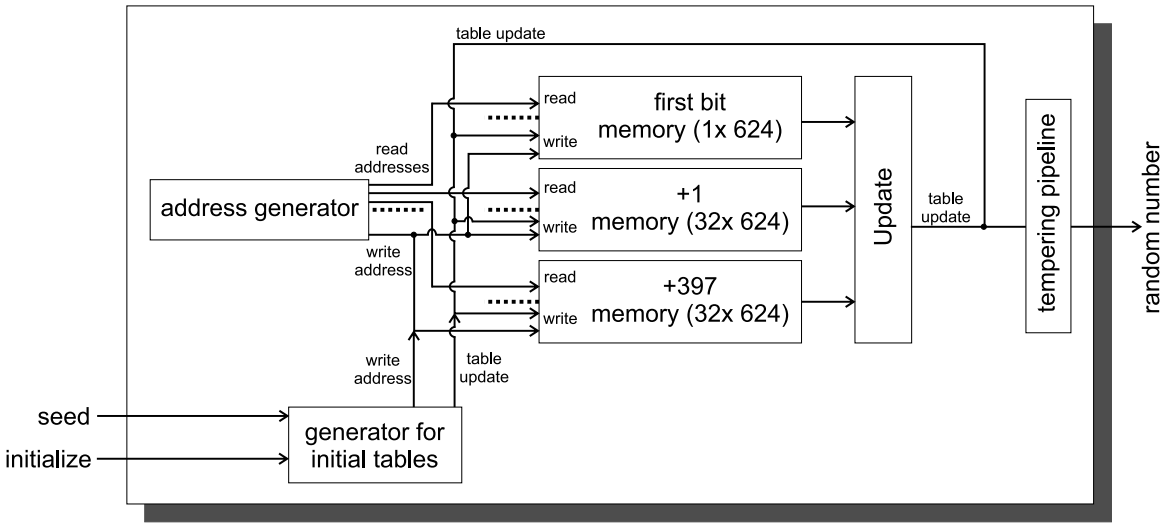

Figure 4: Overview for the Mersenne twister (MT 19937) module.

Depending on the required representation of the rates, conversion into other number formats may be necessary. **Module Normalized Weight Offset** (figure 3b): During learning it may be necessary to keep the weights $p(s|i)$ away from 0 (otherwise the multiplicative learning algorithm may get stuck). Hence this module adds an offset $\frac{\varphi}{N_S}$ to each weight while they pass through this module. However, the module must also ensure that the normalization of the weights $\sum_s p(s|i) = 1$ is still kept intact. Thus the module also divides the intermediate weight by $\frac{1}{1+\varphi}$ before it exits this module. In addition, the corresponding $h(i)$ values are delayed to keep them in sync with their $p(s|i)$ counterparts. The **module Normalized Weight Offset** is directly placed in series after the **MEM module**. To keep the figures 6, 7, and 8 clear, we omit showing the **Normalized Weight Offset module** and show only the **MEM module** and not the combination of both modules.

		Floating point	Fixed point
Module Spike Generator (fig.3a)	LUT	234	82
	LUT-FF pairs	274	162
	Slice registers	257	116
Module Spike Generator with offset (fig. 5a)	LUT	647	231
	LUT-FF pairs	1137	614
	Slice registers	1158	569
(after threshold)	clock cycles	13	9
Mersenne twister (MT) 19937 (fig. 4)	LUT	1326	1326
(produces 32 bit random number)	LUT-FF pairs	1474	1474
	Slice registers	1137	1137
	BRAM	2x 36k	2x 36k
Module Rate Calculator (fig.3b)	LUT	2776	854
	LUT-FF pairs	3114	1000
	Slice registers	3527	957
	BRAM	1x 36k	1x 36k
(calculate $\frac{1}{C_{All}}$)	clock cycles	51	48
(recall $\hat{p}(s)$)	clock cycles	26	14
Module Normalized Weight Offset (fig. 5b)	LUT	3289	1859
	LUT-FF pairs	3771	2067
	Slice registers	4165	2358
(calculate $\frac{1}{1+\varphi}$, $\frac{\varphi}{N_S}$)	clock cycles	49	58
(operate)	clock cycles	25	15

Table 3: Number of components as well as number of clock cycles for the modules concerned with generating spikes, producing random numbers (via Mersenne twister 19937), calculating rates out of observed spikes, and adding a normalized offset to the weights during learning. The number of clock cycles listed for the spike generation module denotes how long it takes after the input value responsible for producing the spike in the end (i.e. exceeding the threshold by the cumulative sum) is observed.

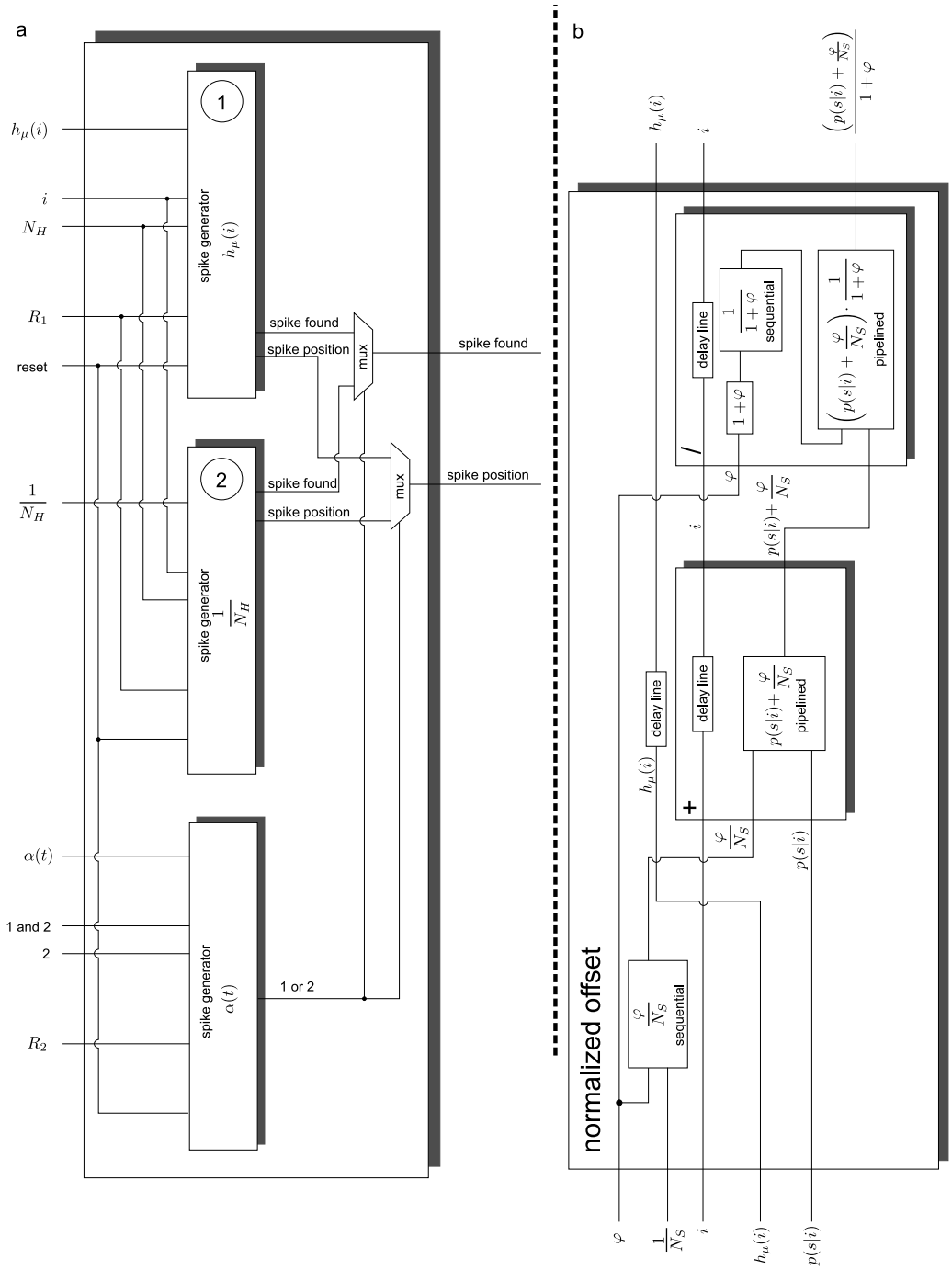

Figure 5: a.) Module for generating spikes with an uniform offset. b.) Module for adding a normalized offset to weights.

4.2 Circuits for updating $h(i)$, online and batch learning $p(s|i)$

		Floating point	Fixed point
h dyn. plus p online lear. (2. stage)	LUT	6437	5296
	LUT-FF pairs	6908	5670
	Slice registers	8285	7020
(h update, latency)	clock cycles	39	23
(p update, latency)	clock cycles	15	10
p online learning (3. stage)	LUT	2857	1520
	LUT-FF pairs	3293	1787
	Slice registers	3831	2077
Module Memory $U(i)$	LUT	33	9
	LUT-FF pairs	147	155
	Slice registers	143	162
	BRAM	1x 36k	1x 36k
(recall latency)	clock cycles	5	5
p batch learning (2. stage) incl. W	LUT	3329	1374
	LUT-FF pairs	3848	1619
	Slice registers	5000	2100
	BRAM	2x 36k	1x 36k
Module Memory $W(s\|i)$	LUT	459	370
	LUT-FF pairs	644	508
	Slice registers	767	612
	BRAM	2x 36k	1x 36k

Table 4: Number of components and clock cycles for the circuits for updating $h(i)$, online and batch learning $p(s|i)$.

With the presented building blocks it is now possible to implement the three equations 1, 2 and 3 as circuit diagrams. Concerning the update of $h(i)$ based on an observed spike s, figure 6 shows how the building blocks are used. The update is done in two stages, during which the spike s is constant. In the first stage, the index i counts through all allowed values $1, ..., N_H$. This recalls N_H pairs of $h(i)$ and $p(s|i)$ values. After an optional normalized offset is added to the weights $p(s|i)$, these pairs a feed into a **NORM-MULTI module**. As a result $\frac{\epsilon}{\sum_j h(j)p(s|j)}$ is calculated. Now stage two begins, with recalling the $h(i)$ and $p(s|i)$ a second time from the memory. In this stage, the output of the multiplier pipeline $h(i) \cdot p(s|i)$ of the **module NORM-MULTI** is combined with the previous calculated $\frac{\epsilon}{\sum_j h(j)p(s|j)}$ via another multiplication pipeline into $\frac{\epsilon h(i)p(s|i)}{\sum_j h(j)p(s|j)}$. In turn, this intermediate result is added via an addition pipeline to its $h(i)$ which was delayed to be available at the right moment in time. As a last processing step, the output of the addition pipeline is divided by $\frac{1}{1+\epsilon}$ via a **Module** /. The output of the **module** / delivers the new $h(i)$ values. These are then written into the $h(i)$ memory of the **module MEM**. Furthermore a **spike generation module** (we used the 'with offset' variety in our design) observes the new $h(i)$ values in parallel and, if provided with a random number, draws a spike out of the new $h(i)$ distribution. Table 4 shows the number of components required to map this circuit onto the Xilinx Virtex 6 FPGA as well as how many clock cycles of latency the **modules** $*$**,** + and / adds to the processing time.

Measured from the time then the first pair of $h(i)$ and $p(s|i)$ values are requested from **module MEM** until the clock cycle when the last updated $h(i)$ values is written into memory, the $h(i)$ update takes $179 + 2 \cdot N_H$ clock cycles for floating point numbers and $118 + 2 \cdot N_H$ clock cycles for 18bit fixed point numbers.

In parallel to the $h(i)$ update, an online update step for the weights $p(s|i)$ can be performed. Figure 7 shows the three stage process for doing so. Stage one and two of the $h(i)$-update and online $p(s|i)$ - update overlap. While $\frac{\epsilon}{\sum_j h(j)p(s|j)}$ is calculated in state one on the $h(i)$-path, $\frac{\gamma}{\sum_j h(j)p(s|j)}$ is calculated in parallel. In stage two, the output $h(i) \cdot p(s|i)$ of the **module NORM-MULTI** from the $h(i)$-path is multiplied via another multiplication pipeline with $\frac{\gamma}{\sum_j h(j)p(s|j)}$ in parallel. This results in $\frac{\gamma h(i)p(s|i)}{\sum_j h(j)p(s|j)}$ as an intermediate result which is stored in a temporary memory $U(i)$. Furthermore the spike s is stored as s_t for stage three. In stage three the old weight values need to be updated with $U(i)$ and stored as new weight values into their memory in **module MEM**. For

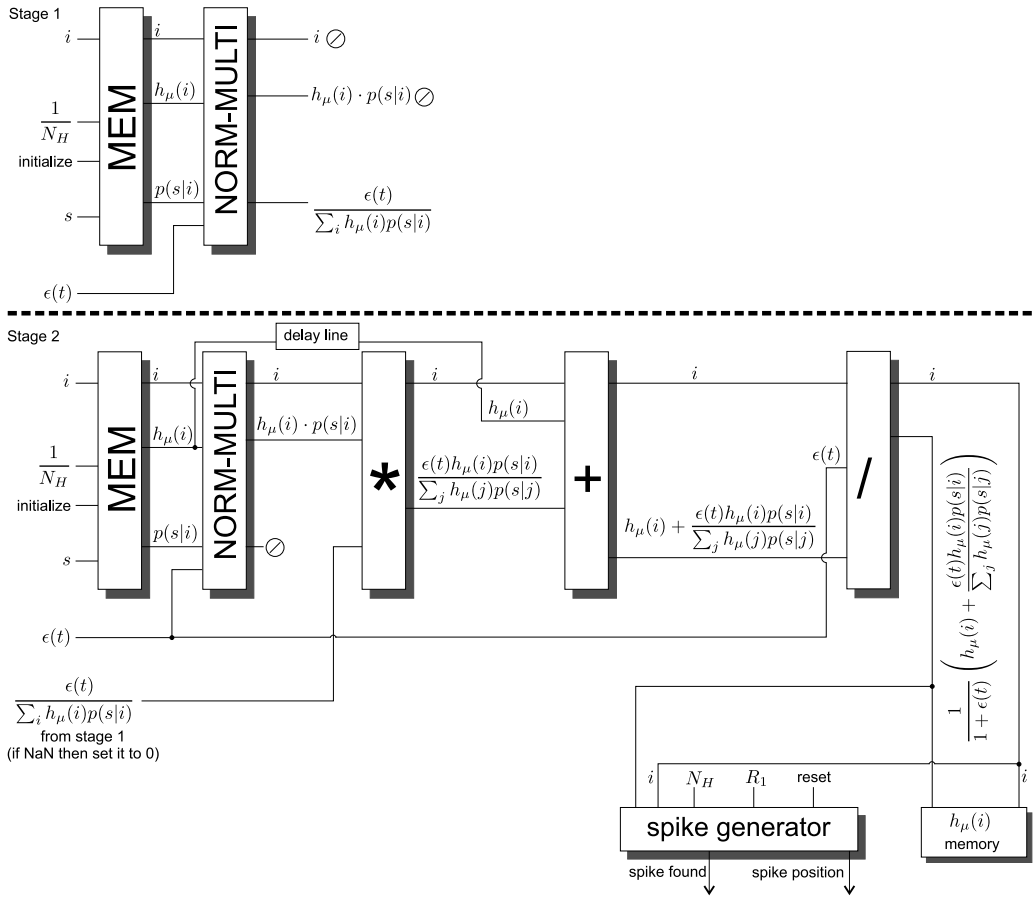

Figure 6: Circuits for updating $h(i)$ based on a spike s. The update is done in two stage.

this process all $p(s|i)$ have to be recalled from **module MEM** (and optionally modified with an normalized offset via the corresponding **offset module**) but with i as the slower changing index. For a given index i the **module** / is prepared with the corresponding $U(i)$. After this preparation is finished, the index s counts from 1 to N_S. In a first step, it is checked if s equals s_t. If this comparison is true, an addition pipeline adds $U(i)$ to $p(s|i)$. Otherwise 0 is added to $p(s|i)$. The output of the addition pipeline is then divided by $1 + U(i)$ via the already prepared **module** /. This result is written as the new $p(s|i)$ values into their memory in **module MEM**. The components required to realize state three and the temporary memory $U(i)$ is listed in table 4.
Since stage one and two of the online $p(s|i)$ update are done in parallel to the $h(i)$-update and with less complex calculations, the number of clock cycles required for stage one and two are defined by the $h(i)$-update. Concerning stage three, we measured $N_H \cdot (124 + N_S)$ clock cycles for floating point numbers and $N_H \cdot (91 + N_S)$ clock cycles for 18bit fixed point numbers. It needs to be noted that the reported clock cycle count for the $h(i)$-path and one $p(s|i)$-path were measured with the **normalized weight offset module** in place. Furthermore, it is important to point out that the listed 124 and 91 clock cycles are mainly a result of the time during which the **module** / needs to be prepared with the actual $U(i)$. Using additional **modules** / and switching between them would reduce these clock cycle counts significantly, however, would cost much more components.
Another approach to learn weights is to perform batch learning. In this learning mode, for a given input pattern $p(s)$, $h(i)$ is updated with many spikes s. Based on the final $h(i)$ and the probability distribution $p(s)$, for generating the observed spikes s in the first place, a contribution to the update of the weights from the actual pattern is calculated. Since $p(s)$ is not available to this circuit, an estimate $\hat{p}(s)$ is calculated via the **module Rate Calculator** from the observed spikes s. The circuit processes several patterns and accumulates these individual contributions into a matrix $W(s|i)$. After the scheduled patterns are processed and their contributions are collected, $W(s|i)$ is sent to an external CPU for updating the weights $p(s|i)$. Using an external CPU for this purpose allows for a high flexibility on the realized update rule. Since the collected contributions can stem from a large number of patterns, $W(s|i)$ is realized with twice the bits as $p(s|i)$ for accommodating much larger

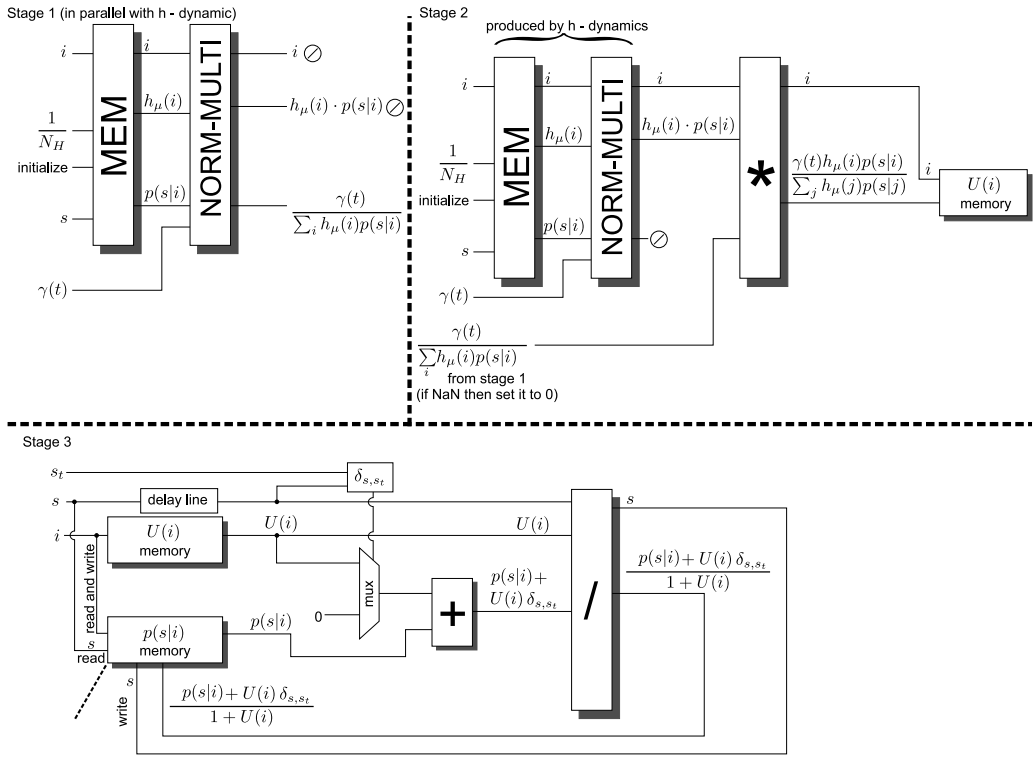

Figure 7: Circuits for updating the weights $p(s|i)$ via online learning. The update of $p(s|i)$ is done in a three stage process.

numbers. The circuit for calculating $W(s|i)$ in a two stage process is shown in figure 8. The required amount of components for stage two are listed in table 4.

For stage one of adding a pattern's information to $W(s|i)$, the circuitry of the $h(i)$-update can be reused, since there are no $h(i)$-update during this time. First, $\hat{p}(s)$ is calculated for one s by **module Rate Calculator** and then used instead of ϵ. As result we get $\frac{\hat{p}(s)}{\sum_j h(j)p(s|j)}$. In stage two, analogous to stage two of the online $p(s|i)$ update path, a multiplexer pipeline is used to create the intermediate results $\frac{\hat{p}(s)h(i)p(s|i)}{\sum_j h(j)p(s|j)}$ for all N_H indices i. The result of this multiplier pipeline is then converted into the corresponding number format with more bits and added to the content of the matrix $W(s|i)$. After all N_H indices i are processed, s is incremented by one and the next $\hat{p}(s)$ is recalled. For this new s the process starts at stage one again. This sequence of fetching $\hat{p}(s)$, stage one and stage two is repeated until it went through all N_S index values of s. Altogether this takes $51 + N_S \cdot (185 + 2 \cdot N_H)$ clock cycles for floating point numbers and $48 + N_S \cdot (124 + 2 \cdot N_H)$ clock cycles for 18bit fixed point numbers.

After implementing these circuits on a FPGA, we investigated how the results differ from a Matlab simulation. Two factors contribute to the differences between the results for these circuits and a Matlab simulation: For all type of divisions, we neglected rounding the least significant bit of the mantissa of the result by just truncating it. Rounding would have required additional clock cycles. And for floating point numbers, the resolution of the mantissa is higher than the 'single' counterpart in Matlab simulations.

Four simple tests have been performed: a.) A non-normalized random $h(i)$-vector with $N_H = 11$ entries was normalized. b.) Given a random input distribution with $N_S = 16$ neurons produced 10 spikes. Using a given random weight matrix $p(s|i)$, added with an offset through the **weight offset module**, the latent variable $h(i)$ with $N_H = 11$ was updated with these 10 spikes. c.) Using the setup of b.) after processing the sixth spike the weights $p(s|i)$ were updated with every spike in an online fashion. d.) Using the setup of b.), after processing the 10th spike an update of $W(s|i)$ was calculated.

For floating point numbers we found a maximum relative (difference divided by the Matlab simulation result) error around $3 \cdot 10^{-8}$ for $h(i)$, $p(s|i)$ and $W(s|i)$. For the 18bit floating point numbers, calculating the maximum absolute difference for $h(i)$, $p(s|i)$ and $W(s|i)$ between the FPGA implementation and the corresponding Matlab

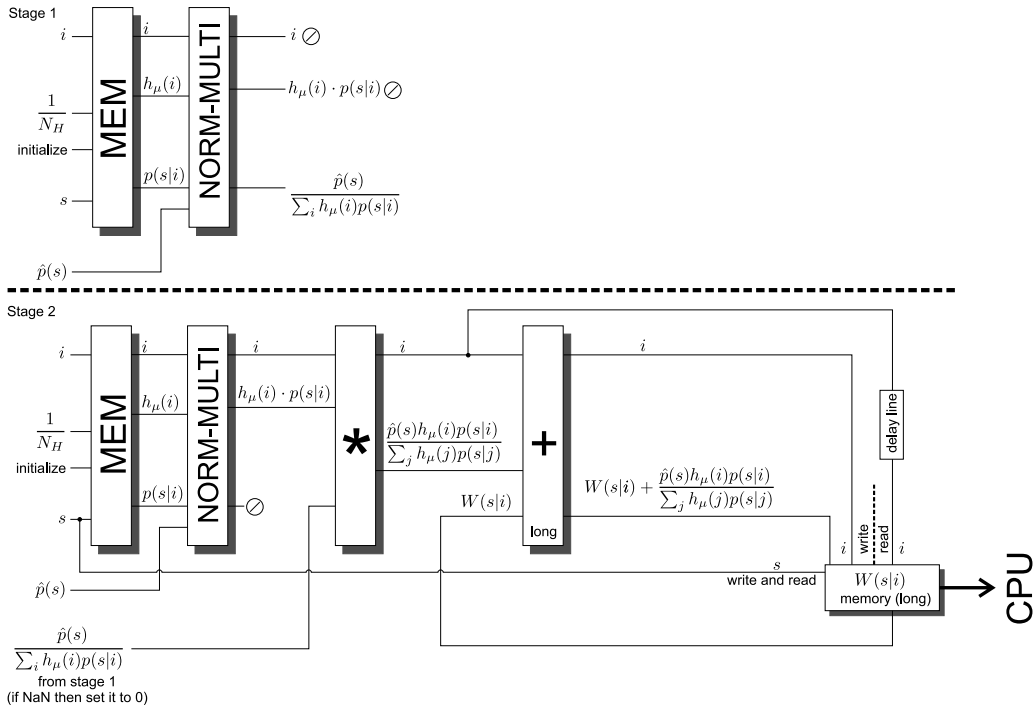

Figure 8: Circuits for batch learning. The matrix $W(s|i)$ is calculated from many spikes and input patterns. The $W(s|i)$ is send to an external CPU and used to calculate the new weights $p(s|i)$. The calculation of $W(s|i)$ is done in two stages.

simulation resulted in single digit differences, typically in the range from 0 to 6 for numbers in the range of $[0, 2^{18} - 1]$. These differences are a result of combining many differences created by neglecting rounding the results for divisions.

4.3 Connecting Spike-By-Spike cores and input groups

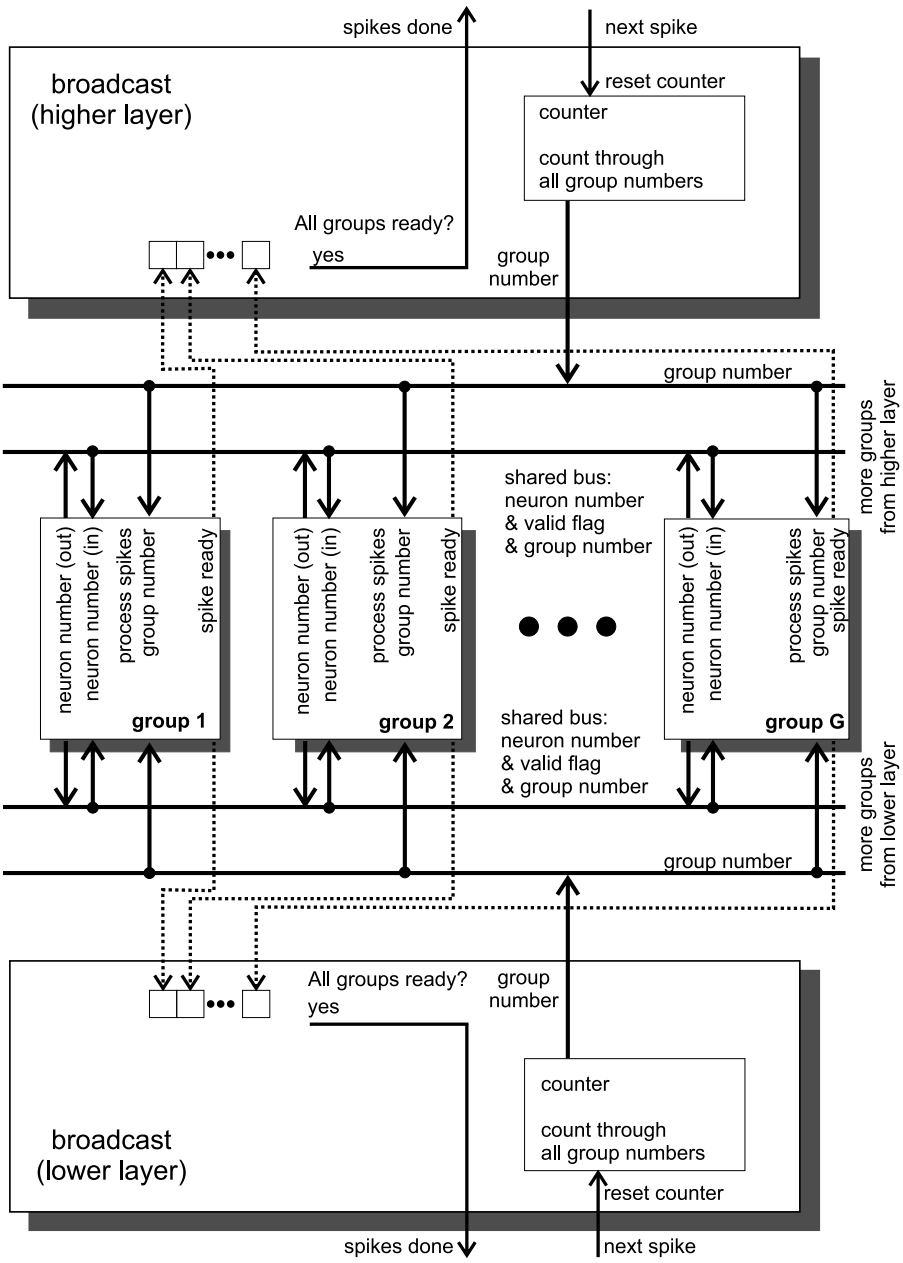

Figure 9: Communication between spike-processing elements in the network via broadcasts.

Typically, a network consists of more than one Spike-By-Spike core which need to exchange spikes. Or the task requires input patterns that need to be converted into spikes, which can be accomplished by an input group (a simple combination of block RAM for the input probability distribution and a **module Spike Generation** for drawing spikes; see table 5 for a component count for a probability distribution of up to 1024 32bit values). As result a communication fabric between these spike producing and spike received network elements is required. Figure 9 shows the communication fabric that was implemented into the presented design.

The communication fabric functions as follows: The input groups and Spike-By-Spike cores are connected to one or more shared data buses. The data bus conveys a unique identifier for the network element on that particular data bus, a neuron number for describing which neuron in that element produced a spike as well as a valid flag signaling that there is valid information on the bus at that moment. By default a network element is silent (all outputs are low). Every data bus owns a **broadcast module** (see table 5 for a component count) which coordinates the activity on its data bus. When all modules on the bus are ready for exchanging spikes, the **broadcast module** sequentially calls all unique identifiers of the elements connected to this bus. Every element

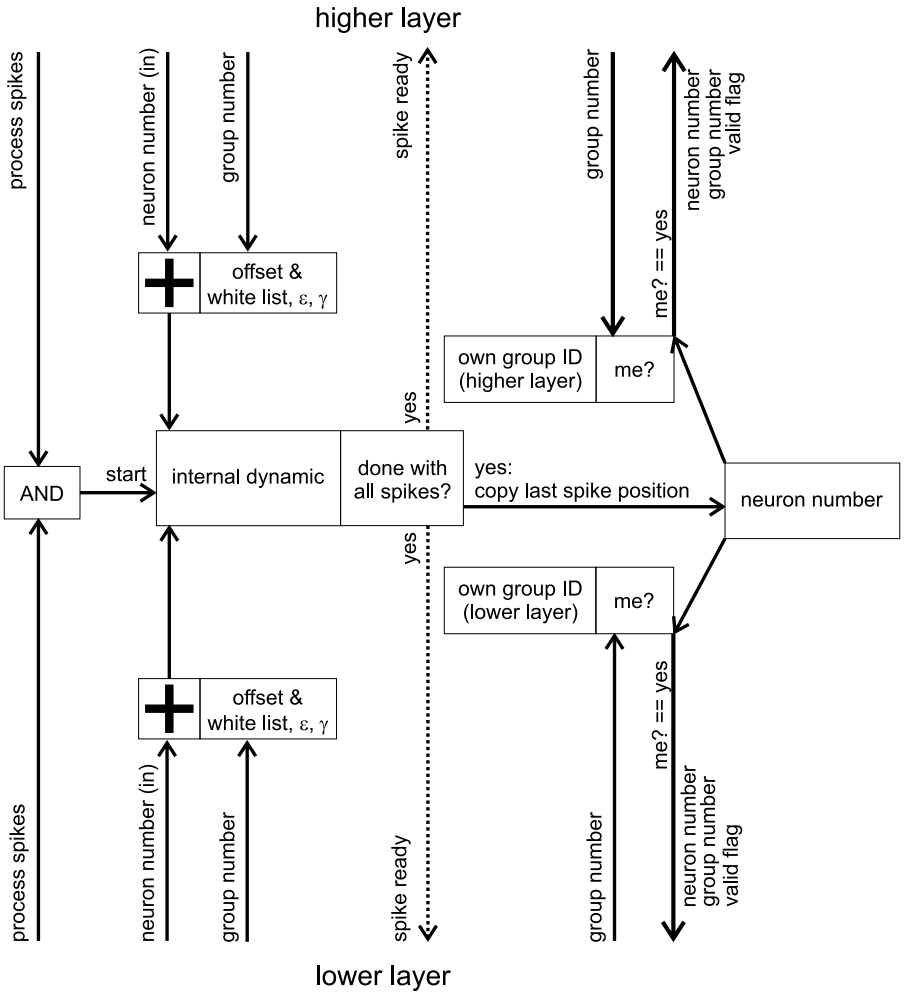

Figure 10: Handling incoming spike that enter a Spike-by-Spike core.

that sees its identifier and has a spike to report, sends out its own identifier and neuron identity that caused the actual spike. After the exchange is performed, the **broadcast module** informs the connected network elements to process the information they received. The **broadcast module** itself is controlled from a controller on a higher level. The shared data bus is realized by XOR elements which ensures that only one network element at the time uses the data bus. For some applications it is helpful to use more than one data bus for selected groups of network elements for keeping the required time to exchange spikes as low as possible. The presented design realizes two data buses.

While an input group doesn't react to spikes on the shared data bus, a Spike-By-Spike core may need to update its internal variables according the observed spikes on the data bus. Figure 10 shows how incoming information on the data buses is handled by a Spike-By-Spike core. Assuming that it sees a valid set of information on the data bus, it compares the network element's unique identifier with a dynamically programmable white list (table 5 details the required component for such a list). In case the unique identifier is on the list, the incoming information is processed further. Otherwise the incoming information is ignored. For white-listed information, the neuron number for the spike is stored and a flag that there was information received from this sending network is set. After the **broadcast module** signals that all spikes have been exchanged and processing can begin (via a 'process spikes' line), the Spike-by-Spike core goes through its white-lists (one per data bus) and processes all flagged entries. It needs to be noted that the white list contains more information: An offset value that allows a dynamically programmed offset on the spike index s for this source of spikes as well as individual ϵ and γ values for every one of the white-listed spike sources.

All the spikes in the white-list are processed which causes an update of the Spike-By-Spike core's $h(i)$. For every processed spike, automatically a new spike is drawn from the updated $h(i)$ probability distribution. However, only the last one of these spikes is stored. After all spikes are processed a 'spike ready' line is raised, informing

		Floating point	Fixed point
SbS core with 2 connection lists	LUT	26915	16794
	LUT-FF pairs	32508	20822
	Slice registers	40239	25317
	BRAM	12x 36k	9x 36k, 2x 18k
SbS arithmetic core	LUT	23779	13914
	LUT-FF pairs	27534	16004
	Slice registers	33965	19443
	BRAM	6x 36k	3x 36k, 2x 18k
Connection list (32 entries)	LUT	1475	same
	LUT-FF pairs	2019	same
	Slice registers	2829	same
	BRAM	3x 36k	same
Broadcast module	LUT	25	same
	LUT-FF pairs	33	same
	Slice registers	29	same
Input group	LUT	407	same
	LUT-FF pairs	548	same
	Slice registers	605	same
	BRAM	1x 36k	same

Table 5: Component counts for the modules shown in figure 9.

the **broadcasts modules** that a new spike is ready for exchange. In the moment when the Spike-By-Spike core sees it's own unique identifier on the shared bus, it puts its own identifier and the generated spike on the bus.

4.4 Coordinating the networks and exchanging data

Type		
Message control center	LUT	2164
	LUT-FF pairs	2703
	Slice registers	2554
	BRAM	8x 36k
MT with message interface	LUT	1536
	LUT-FF pairs	1650
	Slice registers	1475
	BRAM	2x 36k
Arbiter (databus incoming)	LUT	162
	LUT-FF pairs	170
	Slice registers	119
Arbiter (databus outgoing)	LUT	311
	LUT-FF pairs	459
	Slice registers	424

Table 6: Component count for the modules on the 32bit data bus.

While spikes are exchanged on specialized shared buses, also other data are required for operating the network (e.g. input pattern distributions $p(s)$, weights $p(s|i)$, white-lists and random numbers). Some data is generated inside the network, while other data needs to be provided from outside of the network. Furthermore, data needs to be read out of the variables from the network (especially $h(i)$ distributions after processing input pattern as well as weight matrices $p(s|i)$ or $W(s|i)$ after learning) at externally defined moments in time. Thus we implemented two shared 32bit data buses: The first one for exchanging incoming external data as well as internally generated data (mainly random numbers and control sequences). The second one for sending data (e.g. results or information about the status of issued commands) to external receivers. See figure 11 for an overview. On the incoming data bus there are two sources for data: external data and internal data generated by a **message control center**. The **message control center** has several tasks: a.) Based on externally received information, it knows how many spikes the network needs to process as well as after which amount of spikes it needs to start with online learning of the weights. Hence it controls the activity of all the **broadcast modules**. b.) It contains a Mersenne twister random number generator. After a spike, it provides all the network elements with new random numbers. It converts random numbers into messages and puts these messages on the data bus which programs the network elements with these random numbers. c.) It contains a mailbox system that is controlled by the number of already processed spikes. This allows to store every possible – otherwise externally provided – data control sequence in this mailbox together with a number of processed spikes that will release the message at that moment on the data bus. E.g. this message system can be used to change parameters like ϵ during processing the pre-defined number of spikes without waiting of any external just-in-time changes of parameters.

One arbiter joins the two data streams from the incoming data. Another arbiter combines all data sources that are destined for an external data receiver. In table 6 the amount of components required to realize all the participants on these data buses are listed. The network's parts know 17 commands (e.g. set parameters, set $h(i)$, $p(s|i)$, read $h(i)$, $p(s|i)$, $W(s|i)$, reset the **Rate Calculator modules**, normalize $h(i)$, set an input pattern probability distribution, set messages in the mailbox, and start the spike processing) that require up to nine 32bit words.

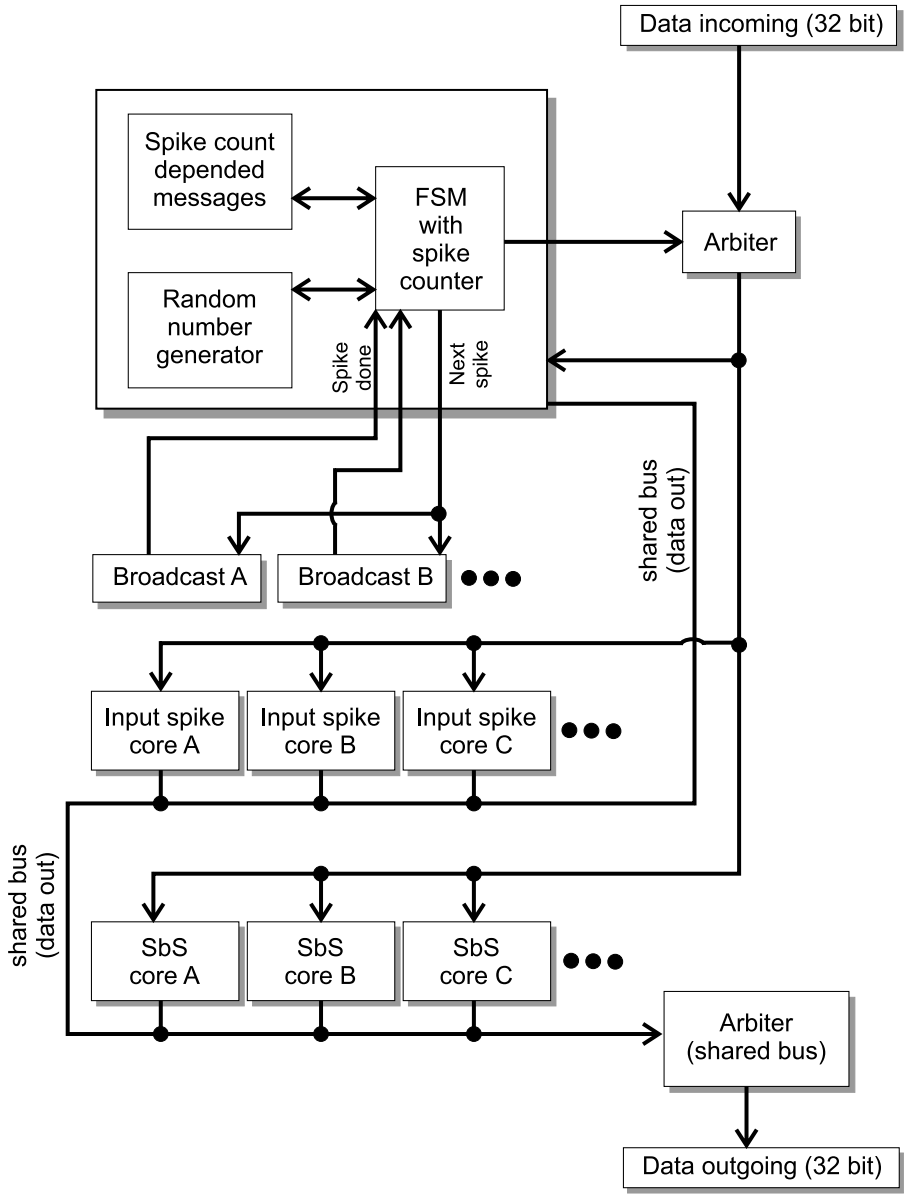

Figure 11: Data bus between the different elements as well as external participants.

4.5 Test with a Xilinx Virtex 6 FPGA

		Floating point	Fixed point
Network (25 input groups, 1 SbS core)	LUT	38021	27953
	LUT-FF pairs	45984	35821
	Slice registers	58802	43853
	BRAM	45x 36k	42x 36k, 2x 18k

Table 7: Amount of components required for a network with one Spike-By-Spike core and 25 input groups.

We designed circuits for the network (see table 7 for details) such that it can be mapped, placed and routed without any special FPGA components (e.g. DSP cores), except for **block RAM modules**, on a FPGA or ASIC. Our design, especially the number of buffering and pipeline stages, is optimized to generate a firmware that can be operated with 200+MHz clock speed on a Xilinx Virtex 6 LX550T-1FF1759 (speed grade: commercial 2). The Xilinx ISE 14.3 software is capable of producing such a firmware when the circuit is alone and the IO pins can be selected by the software.
However, we weren't able to achieve these high clock rate when the network was embedded into the third party firmware which is required for communicating with the Virtex 6 FPGA on the 4DSP FM680 PCIe card. We used the 'adder' example from the training materials and simply replaced the 'adder' core with our network. We allowed for a separate clock domain for the network by insulating the data-flow by dual clock FIFOs on the input and output side. Nevertheless, Xilinx ISE wasn't able to provide us with a working firmware if we set the clock speed for the network to 200MHz. Thus we were forced to use the overall 125MHz clock provided by the 4DSPs example design also for our network.
With the 125MHz clock speed were were able to run the network on 4DSP FM680 cards under Linux (Centos 7.5 64bit with driver version 04.05.2018). Results were identically to the Xilinx ISim simulations. However, the overall design of the 4DSP FM680 doesn't fit for our application very well. This type of card is designed for high data bandwidth applications where 64bit words in blocks of 1024bits are continuously exchanged. On one side we were forced to incooperate our 32bits into block of 1024bits. On the other side we had problems receiving data from the card because our network doesn't produce data continuously. Thus the Linux driver and our software had to recover from trying to read from the card when it had no data for us. All this combined, slows down the communication with our network on the FPGA significantly.

4.6 Comparing CPU with FPGA

For a comparison of the performance of a modern CPU with the FPGA simulation, we used simulations under Matlab in single (32bit) precision floating point numbers and measured the time with Matlab's tic toc command. For keeping the measurement errors low, we always measured the time over one million spikes. For the FPGA, we checked the performance for the 36bit floating point numbers. Since the problems with the Linux driver and data communication with the FPGA card made it impossible to reliably measure the time required for such a simulation, we calculated the required time from the measured number of clock cycles for an operation and the 125 MHz clock which are found to work with the FPGA firmware. Since the FPGA has a precise timing compared to a normal CPU these calculation lead to very good estimates for the required processing time.
The simulations were run on a system with an Intel Core i5-6600k with 3.8GHz in single core Turbo Boost mode and with 3.6GHz when all four cores are under load. Also we used a system with dual Intel Xeon E5-2640v3 (2x 8 cores) with 3.4GHz in single core and and 2.6GHz with multi cores. All RAM DIMM slots were filled with identical modules. Thus the double memory channel for the Intel i5 and the quad memory memory mode on the Intel Xeon were active.
In figure 12a the time for drawing one million spikes from a random probability distribution on a single CPU core is shown. In comparison, the FPGA requires roughly 10 clock cycles (80ms for one million spikes) to produce and provide the required random number to the input group for producing a spike. As long as $N_S + 10 - (179 + 2 \cdot N_H)$ is smaller or equal zero, no additional time is required and the spike is produced in parallel to updating the latent

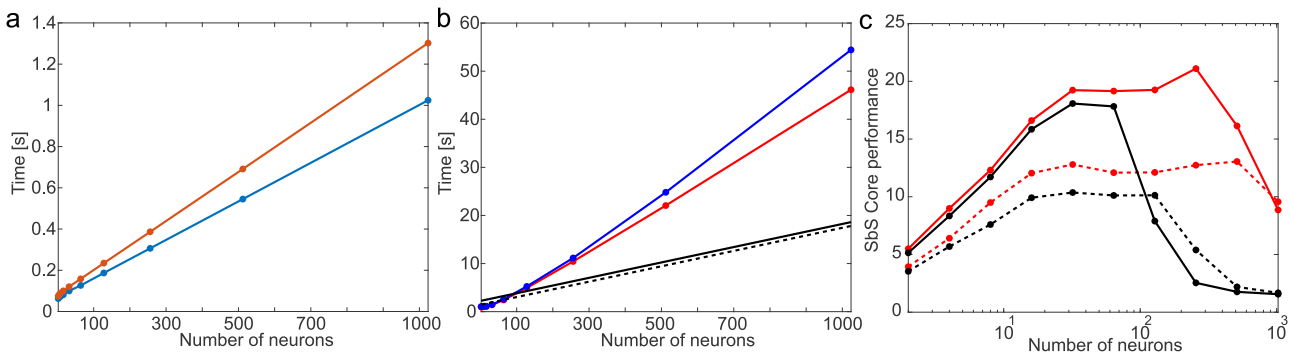

Figure 12: a.) Time for drawing one million spikes from a probability distribution with the number of neurons shown on the x-axis. Simulation was run on a single core on a dual CPU Intel Xeon E5-2640v3 (red) and an Intel Core i5-6600k (blue). If N_H is large enough, the FPGA does not depend on N_S and doesn't require additional clock cycles for this operation. The shown durations are averaged over 100 trials. b.) Duration for updating the latent variables with one million spikes. The input population was kept at $N_S = 1024$. N_H, the number of neurons in one SbS group, was varied according to the x-axis. The dashed line represents the estimated time which the FPGA needs. The black solid line shows how the duration would change if 100 clock cycles of additional overhead per spike would occur. The red line represents the Intel Core i5-6600K running as single core in Turbo Boost mode with 3.8GHz (averaged over 100 trials), while the blue line shows the measured time when all four cores are working in parallel (averaged over 400 trials). c.) Durations for the simulation of b.) with active online learning and adding a normalized offset to the weights. Shown is the time required with the FPGA divided by the time which a CPU needs. The red lines show the performance factor for single core simulations and the black lines for multi-core simulation. The solid lines were measured with an Intel Core i5-6600k (4 cores) and a dual Intel Xeon E5-2640v3 (2x 8 cores) system. For the plots b.) and c.) the time for producing spikes was subtracted.

variable. Otherwise only $N_S + 10 - (179 + 2 \cdot N_H)$ additional clock cycles are required. For the CPUs the time required for producing these spikes scales also linearly with N_S. Without a parallel update of the latent variable which absorbs clock cycles, the FPGA would be ≈8.3x slower in drawing spikes that the Intel Core i5. For figure 12b and c, we subtracted the time required for drawing spikes and focus only on the raw processing power.

Figure 12b shows the duration it takes to update the latent variables one million times on the Intel Core i5 and the FPGA. All results are shown for one SbS core. For SbS groups with small numbers of neurons (below 128 neurons), the Intel CPU is faster. For larger number of neurons the FPGA is faster. In the case of $N_H = 1024$, the FPGA is ≈2.6x (Intel i5 in single core mode) and ≈3x faster (Intel i5 in quad core mode) per SbS core. While the Intel CPU looses some performance when four SbS cores are simulated in parallel, the network of SbS cores on the FPGA requires only one clock cycle more for every additional SbS core on the spike data bus. This translates into 8ms for one million update operations. Using several data buses for exchanging the spikes would even reduce this small amount of time. Furthermore, the FPGA also performs an optional normalized offset to the weights in the reported duration, which we didn't test in the Matlab simulations.

Concerning online learning, the FPGA can calculate part of the learning rule in parallel to the update of the latent variable. However, the re-normalization of the updates weights takes a substantial amount of time ($N_H \cdot (124 + N_S)$ clock cycles or 1175552 clock cycles / 9.4ms for one re-normalization with $N_H = 1024$ and $N_S = 1024$). Due to the high clock rates and the simple calculations, Intel CPUs are able to perform this calculation very fast as long as the memory bandwidth is not hit. Figure 12c shows that for single core operation with intermediate number of neurons per SbS group, the Intel CPUs are roughly 20-21x faster than the FPGA. For larger number of neurons, the memory bandwidth starts to get a bottleneck, where a reduction to ≈10x can be seen. For smaller number of neurons, the performance is only 5x faster than the FPGA because the CPU can't simply stream the data from the memory to it's floating point units but is interrupted by other operations. Using more than a single core shows the full size of the problem with the memory bandwidth limit where the performance for larger number of neurons breaks down to a speed increase of only ≈1.7x over the FPGA. This effect doesn't occur on the FPGA since every SbS core has it's independent memory. Nevertheless, in the case with $N_H = 1024$, 27 SbS cores on a FPGA with a 125MHz clock would be required to reach performance equality

with the 16 core dual Intel Xeon system.
In the case of batch learning, the required time is almost exclusively defined by the update of the latent variables. The calculation of $W(s|i)$ from the $h(i)$, $p(s|i)$ and $\hat{p}(s)$ is performed only once per pattern after many spikes have been processed.

5 Discussion and conclusion

The paper presents a design and an investigation of an optimized circuitry for implementing neuronal networks based on Spike-By-Spike (SbS) neurons ([18]). In this type of network, neurons are organized in groups where the neurons inside a group compete with each other. Every neuron in a group has a latent variable $h(i)$. The value of $h(i)$ is a positive number and the competition is expressed by a normalization over all latent variables of a group ($\sum h(i) = 1$). In between groups, neurons communicate exclusively via spikes (i.e. an index s_t describing a single neuron's identity in a group at a time t). The connections between neurons from different groups are described by weights $p(s|i)$ (with s as the emitting neuron and i as the receiving neuron). Also the values of the weights are positive numbers and limited in size by a second normalization condition ($\sum_s p(s|i) = 1$). Thus everything describing the state of the network is given by positive numbers in the range of 0 to 1. In this paper we compared how different types of representations for these numbers result in different sizes of computational circuits as well as different required numbers of clock cycles. In particular we examined 18 bit positive fixed point numbers and 36 bit positive floating point numbers. The latter are a custom derivation of the IEEE 754 standard by assuming positivity and a higher bit count for the mantissa. The fixed point representation requires less components but was found usable only for smaller numbers of neurons per group.
If a spike s^t is received by a group of neurons, for every neuron in this group an update of it's latent variables is calculated through

$$h^{t+1}(i) = \frac{1}{1+\epsilon}\left(h^t(i) + \epsilon \frac{h^t(i)p(s^t|i)}{\sum_j h^t(j)p(s^t|j)}\right).$$

We presented computational circuitry for this equation that only requires to read the involved $h(i)$ and $p(s^t|i)$ value pairs twice during the update process. For floating point numbers a number of $179 + 2 \cdot N_H$ clock cycles and $118 + 2 \cdot N_H$ clock cycles for the fixed point numbers were measured. The fact that the number of clock cycles that don't scale with the number of neurons is a result of the information traveling through the pipeline structure of the computational circuitry as well as setting up the circuits for incoming data. This amount of clock cycles can be reduced if special features like e.g. adding an normalized offset to the weights are removed. Auxiliary circuits were designed that, given a random number, draw a new spike from $h(i)$ while updating the h-values. Furthermore it was shown that circuits for learning the weights (see equation 2 and 3) can be designed, while keeping the amount of required components low by partially reusing circuitry for the h-update.
Given a collection of groups of SbS neurons, the exchange between groups of spikes requires organization. In our framework we show that exchanging spike information can easily be done by a common data bus. A controller checks if all spike generating elements on that bus are ready for exchanging spikes and then calls every element on that bus to report its spike. Every SbS core has a programmable list which defines the spike producing elements it listens to. This allows user-defined changes of the network architecture on the fly. Also it allows to remove broken elements (which might be e.g. a result of production failures) from the network and to reallocate – if available – other resources to take its place. The lists contains additional information to allow for changes of the ϵ parameter for the h-dynamic and the γ parameter for online learning depending on the source of the spike. This feature is essential for more complex networks. In the same spirit, we introduced a 'mailbox system' into the framework. It allows to make processing a given pattern for a user-defined number of spikes independent from external sources. Sometimes it may be necessary to change the ϵ parameter after a given number of spikes. Without the mailbox system, an external controller would need to determine the state of the network (i.e. are the required number of spikes already processed?), then change the parameters, and tell the network to continue for another amount of spikes. This would obviously waste time. In comparison, with the mailbox system

the network distributes the pre-defined messages about the parameter change automatically in the moment a pre-defined number of spikes have been processed. Obviously, the same is true for the required random numbers for producing spikes. Thus it is also essential that random numbers are sent to the consumers of these random number with as small as possible pauses. For this reason it was necessary to add a random number generator (Mersenne Twister 19937) to the overall message controller with its mailbox system. The message controller also takes care that the correct number of spikes is processed and that any external system is informed that the network finished the pre-programmed number of spikes.

As a second type of spike producing element in the network, we designed input groups. These input groups take a probability distribution, where every value is represented by a 32 bit fixed point number, as well as 32 bit random numbers and produces spikes from this information. For programming these probability distributions as well as configuring any other parameter or variable in the whole framework (or network) two data buses (one for incoming and one for outgoing data) with a pre-defined set of commands are introduced into the design. The aforementioned message controller receives commands from the incoming data bus but can also introduce its own commands into this data stream that is seen by all the elements of the network.

The presented design was transferred into a FPGA firmware written in VHDL. It was taken care that no special FPGA vendor proprietary modules (except block RAM) were used, for allowing re-using the design for developing a custom ASIC. For proof of feasibility, we integrated the VHDL code into an example firmware for the 4DSP FM680 cards which hosts a Xilinx Virtex 6 FPGA from 2009. We successfully generated a binary firmware bit file for this card, tested it under Linux and compared it with the results from Matlab simulations. Small differences in our simple example simulation were found, which were expected due to the difference in precision (floating point numbers: 5 more bits representing the mantissa for the FPGA implementation compared to Matlab) as well as a different handling of rounding during the mathematical operation of division. We decided that the differences through rounding are not relevant, especially when compared to the required increase in the amount of components and processing stages necessary for reaching a perfect match with the Matlab simulations. While the pure SbS network would have been able to run with over 200MHz on an otherwise empty Virtex 6, combining it with the existing card manufacturer's firmware unexpectedly forced us to reduce the clock speed down to 125MHz. Isolating the network into a separate 200MHz clock domain didn't work out. Since the VHDL code was optimized for a 200MHz stand-alone implementation on a Virtex 6 LX550T FPGA, several buffering steps in the pipeline architecture were added which are unnecessary for a 125MHz operation. A second SbS core was added to the firmware's network and tested out fine. Extrapolating from the number of required components for one SbS core, we expected that we could increase the number of SbS cores to 7 or 8 for floating point number on this Xilinx Virtex 6 LX550T. However, the generation of these firmwares failed in the Xilinx ISE design tool. Our guess is that the reason lies in the fixed placement of the some components (especially block RAM) on the FPGA. Increasing the number of SbS cores produces distances between components that are too long for even 125MHz on this FPGA from 2009. We didn't pursued the increase in the number of SbS cores any further, because we found that the 4DSP firmware, driver and software framework for this card is optimized for continuously streaming of large bandwidth of data. However, in our application the data is send in a stop-and-go fashion which doesn't ensure that data is always available. Thus it was necessary to probe the cards for available data via blindly trying to read data from the card and to rely on slow timeouts of the Linux driver if there was no data available at that exact moment. Furthermore we had to fill up our data streams with zeros (up to 31x bit in zeros for filling up compared to the payload) to conform to the required data format. In the end we decided to focus on transferring the design onto a custom ASIC instead of optimizing the FPGA firmware.

The computationally attractive aspect of a SbS neuron based network is the property that groups of neurons only communicate via spikes. Thereby every group of neurons operates independently from all the other groups. The internal computation of any given group is based exclusively on the incoming spikes and its internal variables. After finishing its computations every group of neurons also only sends out spikes as a signal to other groups. This locality property is true for updating the $h(i)$ values as well as for learning the weights $p(s|i)$. Thus the locality of its groups is the key ingredient for massively parallelizing networks with such elements (groups). The optimal way for parallelizing such networks is to provide independent memory and computational circuitry for

every individual group of neurons. With the specialized processors proposed in this paper one update cycle in a large network of such groups does not scale significantly with the number of groups. This stands in stark contrast to running such networks on clusters of computers where one would experience latencies and transmission delays that scale much worse with the number of nodes due to the creation of network packages, their transmission and their analysis by soft- and hardware.

In the future we plan to design a custom ASIC with a network with larger numbers of SbS cores. As a preparation we made sure that the design doesn't use any special 3-party IP cores. For improving the processing speed of the SbS groups it might be interesting for a custom ASIC to use different clock domains for exchanging spike information between groups and doing the calculations inside the SbS group. This would also ensure that the SbS groups can retain high computational speeds even if the distances between the groups on the chips for exchanging spike information increase. This even allows to extend the data bus for exchanging the spikes over a multitude of chips. Furthermore, it is also not necessary that all SbS groups run at the same speed. It is enough that only the part of a group responsible for communicating with the data bus for exchanging spike information is in sync.

The area on chip required for memory may be an issue for the custom ASIC. Concerning the weights, in a convolution setting all groups in a layer use the same weight values. Thus in such an application it could be suitable to use a common RAM for all groups in that layer. Two approaches are possible: Going through all the weight values sequentially and the SbS groups extract their relevant part from the data stream. Or use a custom RAM that delivers the $p(s|i)$ values for all s during one read cycle in parallel which would create N_S streams of weight values in parallel. In this scenario, every SbS group would have a multiplexer connected to the output stream of the RAM and would, according to the value of s^t, switch between the different weight value streams.

References

[1] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[2] A. Azkarate Saiz, "Deep learning review and its applications," 2015.

[3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, p. 484, 2016.

[4] Y. Guo, Y. Liu, A. Oerlemans, S. Lao, S. Wu, and M. S. Lew, "Deep learning for visual understanding: A review," *Neurocomputing*, vol. 187, pp. 27–48, 2016.

[5] L. A. Gatys, A. S. Ecker, and M. Bethge, "Image style transfer using convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2414–2423, 2016.

[6] G. Lacey, G. W. Taylor, and S. Areibi, "Deep learning on fpgas: Past, present, and future," *arXiv preprint arXiv:1602.04283*, 2016.

[7] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, p. 533, 1986.

[8] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.," *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[9] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[10] E. J. Candes, J. K. Romberg, and T. Tao, "Stable signal recovery from incomplete and inaccurate measurements," *Communications on Pure and Applied Mathematics: A Journal Issued by the Courant Institute of Mathematical Sciences*, vol. 59, no. 8, pp. 1207–1223, 2006.

[11] M. Lustig, D. L. Donoho, J. M. Santos, and J. M. Pauly, "Compressed sensing mri," *IEEE signal processing magazine*, vol. 25, no. 2, pp. 72–82, 2008.

[12] T. Wiedemann, C. Manss, and D. Shutin, "Multi-agent exploration of spatial dynamical processes under sparsity constraints," *Autonomous Agents and Multi-Agent Systems*, vol. 32, no. 1, pp. 134–162, 2018.

[13] B. A. Olshausen and D. J. Field, "What is the other 85 percent of v1 doing," *L. van Hemmen, & T. Sejnowski (Eds.)*, vol. 23, pp. 182–211, 2006.

[14] A. Spanne and H. Jörntell, "Questioning the role of sparse coding in the brain," *Trends in neurosciences*, vol. 38, no. 7, pp. 417–427, 2015.

[15] S. Ganguli and H. Sompolinsky, "Compressed sensing, sparsity, and dimensionality in neuronal information processing and data analysis," *Annual review of neuroscience*, vol. 35, pp. 485–508, 2012.

[16] S. Ganguli and H. Sompolinsky, "Statistical mechanics of compressed sensing," *Physical review letters*, vol. 104, no. 18, p. 188701, 2010.

[17] A. M. Bruckstein, M. Elad, and M. Zibulevsky, "On the uniqueness of nonnegative sparse solutions to underdetermined systems of equations," *IEEE Transactions on Information Theory*, vol. 54, no. 11, pp. 4813–4820, 2008.

[18] U. Ernst, D. Rotermund, and K. Pawelzik, "Efficient computation based on stochastic spikes," *Neural computation*, vol. 19, no. 5, pp. 1313–1343, 2007.

[19] D. D. Lee and H. S. Seung, "Learning the parts of objects by non-negative matrix factorization," *Nature*, vol. 401, no. 6755, p. 788, 1999.

[20] D. D. Lee and H. S. Seung, "Algorithms for non-negative matrix factorization," in *Advances in neural information processing systems*, pp. 556–562, 2001.

[21] R. Salakhutdinov, "Learning deep generative models," *Annual Review of Statistics and Its Application*, vol. 2, pp. 361–385, 2015.

[22] G. E. Hinton, "A practical guide to training restricted boltzmann machines," in *Neural networks: Tricks of the trade*, pp. 599–619, Springer, 2012.

[23] W. Maass and C. M. Bishop, *Pulsed neural networks.* MIT press, 2001.

[24] E. M. Izhikevich, "Which model to use for cortical spiking neurons?," *IEEE transactions on neural networks*, vol. 15, no. 5, pp. 1063–1070, 2004.

[25] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[26] M. Rolinek and G. Martius, "L4: Practical loss-based stepsize adaptation for deep learning," *arXiv preprint arXiv:1802.05074*, 2018.

[27] N. Shirazi, A. Walters, and P. Athanas, "Quantitative analysis of floating point arithmetic on fpga based custom computing machines," in *fccm*, p. 0155, IEEE, 1995.

[28] X. Tian and K. Benkrid, "Mersenne twister random number generation on fpga, cpu and gpu," in *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on*, pp. 460–464, IEEE, 2009.

[29] N. Saraf and K. Bazargan, "A memory optimized mersenne-twister random number generator," in *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pp. 639–642, Aug 2017.