

 Open access • Posted Content • DOI:10.1101/500280

Massively Parallel FPGA Hardware for Spike-By-Spike Networks — [Source link](#)

David Rotermund, Klaus Pawelzik

Institutions: University of Bremen

Published on: 19 Dec 2018 - bioRxiv (Cold Spring Harbor Laboratory)

Topics: Network on a chip, Field-programmable gate array and Massively parallel

Related papers:

- [Efficient Computation Based on Stochastic Spikes](#)
- [Deep learning in neural networks](#)
- [Efficient deep neural network acceleration through FPGA-based batch processing](#)
- [Throughput optimizations for FPGA-based deep neural network inference](#)
- [Scalable Network-on-Chip Architectures for Brain–Machine Interface Applications](#)

Share this paper:    

View more about this paper here: <https://typeset.io/papers/massively-parallel-fpga-hardware-for-spike-by-spike-networks-1sxww7vbit>

1

Massively Parallel FPGA Hardware for Spike-By-Spike Networks

David Rotermund^{1,*} and Klaus R. Pawelzik¹

¹*Institute for Theoretical Physics, University of Bremen, Bremen, Germany*

Correspondence*:

David Rotermund

davrot@neuro.uni-bremen.de

2 ABSTRACT

3 While inspired by the brain, currently successful artificial neural networks lack key features of
4 the biological original. In particular, the deep convolutional networks (DCNs) neither use pulses
5 as signals exchanged among neurons, nor do they include recurrent connections which are
6 both core properties of real neuronal networks. This not only puts to question the relevance of
7 DCNs for explaining information processing in nervous systems but also limits their potential for
8 modeling natural intelligence.

9 Spike-By-Spike (SbS) networks are a promising new approach that combines the
10 computational power of artificial networks with biological realism. Instead of separate neurons
11 they consist of neuronal populations performing inference. Even though the underlying equations
12 are rather simple implementations of such networks on currently available hardware are several
13 orders of magnitude slower than for comparable non-spiking deep networks.

14 Here, we develop and investigate a framework for SbS networks on chip. Thanks to the
15 communication via spikes, already moderately sized deep networks based on the SbS approach
16 allows a parallelization into thousands of simple and fully independent computational cores. We
17 demonstrate the feasibility of our design on a Xilinx Virtex 6 FPGA while avoiding proprietary
18 cores (except block memory) that can not be realized on a custom-designed ASIC. We present
19 memory access optimized circuits for updating the internal variables of the neurons based on
20 incoming spikes as well as for learning the connection's strength. The optimized computational
21 circuits as well as the representation of variables fully exploit the non-negative properties of all
22 data in the SbS approach. We compare the sizes of the arising circuits for floating and fixed point
23 numbers. In addition we show how to minimize the number of components that are required for
24 the computational cores by reusing their components for different functions.

1 INTRODUCTION

25 Nowadays, deep neuronal networks (Schmidhuber, 2015) are a basis for successfully applying neuronal
26 networks on problems from artificial intelligence research (Azkarate Saiz, 2015; Silver et al., 2016;
27 Guo et al., 2016; Gatys et al., 2016). The revival of using neuronal networks was provoked by the increase
28 of computational powers in modern computers and boosted even more through modern 3D graphic cards
29 as well as specialized application specific integrated circuits (ASICs) (Sze et al., 2017; Jouppi et al., 2018)
30 and field programmable gate arrays (FPGAs) (Lacey et al., 2016). The most successful type of networks
31 is based on multilayer perceptrons (Rumelhart et al., 1986; Rosenblatt, 1958) and consists of several so

32 called hidden layers. Typically information is processed and feed forward from one hidden layer to the
33 next, beginning at the input layer and ending at the network's output layer. In theory such a network is
34 able to calculate arbitrary functions. However, for doing so the weights – describing the connection of
35 elements of one layer to the next – must be learned based on the intended task. During learning these
36 weights, the error between the desired outcome and the actual outcome of the network is minimized.

37 In the realm of brain research, more detailed and biologically realistic spiking neuron models
38 are used (Maass and Bishop, 2001; Davies et al., 2018; Thakur et al., 2018; Pfeiffer and Pfeil, 2018;
39 Izhikevich, 2004) which require a vast amount of computational power (Izhikevich, 2004). There is
40 a large engineering community that constructs neuromorphic hardware for accelerating the necessary
41 computation for simulating these type of neurons, e.g. (Thakur et al., 2018; Davies et al., 2018;
42 Furber et al., 2014; Moore et al., 2012; Wang and van Schaik, 2018). Also, networks in the brain have
43 no simple feed-forward architecture but instead recurrent connections are ubiquitous implying that real
44 information processing is dynamic.

45 In (Ernst et al., 2007) we presented a different type of neuronal network (called Spike-by-Spike network,
46 SbS) based on the family of generative models (Lee and Seung, 1999, 2001; Salakhutdinov, 2015; Hinton,
47 2012). Instead of separate neurons they consist of neuronal populations performing inference and the
48 neurons exchange information via stochastic spikes. In terms of computational requirements the SbS
49 network lies in between the traditional perceptron based non-spiking networks and typical spike-based
50 networks.

51 In terms of biological plausibility, the SbS network is placed in between non-spiking networks (e.g.
52 deep convolutional networks) and networks of spiking neurons with realistic models (e.g. leaky integrate-
53 and-fire neurons, IaF neurons). Comparing it with networks of IaF neurons, the SbS network removes
54 large parts of biological plausibility. However, this comes with a reduction in parameters. There is no
55 need to optimize e.g. time parameters, firing thresholds, or membrane constants because there is no real
56 time left in a SbS network. Instead of many (sometimes thousands) computational steps that are required
57 to get the next spike in a IaF population, a SbS IP requires only one update per neuron to determine
58 the next spike. Such an update requires $3N$ multiplications, $2N$ summations, and the inversion of one
59 value if N is the number of neurons in the SbS IP. It is also possible to build recurred networks with IPs
60 (Rotermund and Pawelzik, 2019b). This all together allows SbS IPs to be used for building larger models
61 for understanding information processing in the brain.

62 Furthermore all entities in SbS networks are described by positive numbers which leads to
63 sparse representations Bruckstein et al. (2008). A recent discovery in the field of machine learning
64 is compressed sensing (Candes et al., 2006), which allows to reconstruct underlying causes from
65 incomplete measurements if the underlying causes are sparse. This lead to applications in many fields
66 from reducing measurement time for Magnetic Resonance Imaging (Lustig et al., 2008) to building
67 swarms of robots for efficiently exploring other planets (Wiedemann et al., 2018). Sparseness and its
68 benefits for information processing is also an important topic in brain research (Olshausen and Field,
69 2006; Spanne and Jörntell, 2015; Ganguli and Sompolinsky, 2012). In the context of this work, the
70 finding that non-negative representations in a network can be sufficient to induce some degree
71 of sparsity is particularly interesting (Ganguli and Sompolinsky, 2010; Bruckstein et al., 2008). In
72 (Rotermund and Pawelzik, 2019a) we extended the old SbS approach for shallow networks to deep
73 networks consisting of large numbers of inference populations (IPs). In (Rotermund and Pawelzik, 2019b)
74 we show how to use bi-directional information exchange between the layers of deep SbS network for
75 biologically realistic learning.

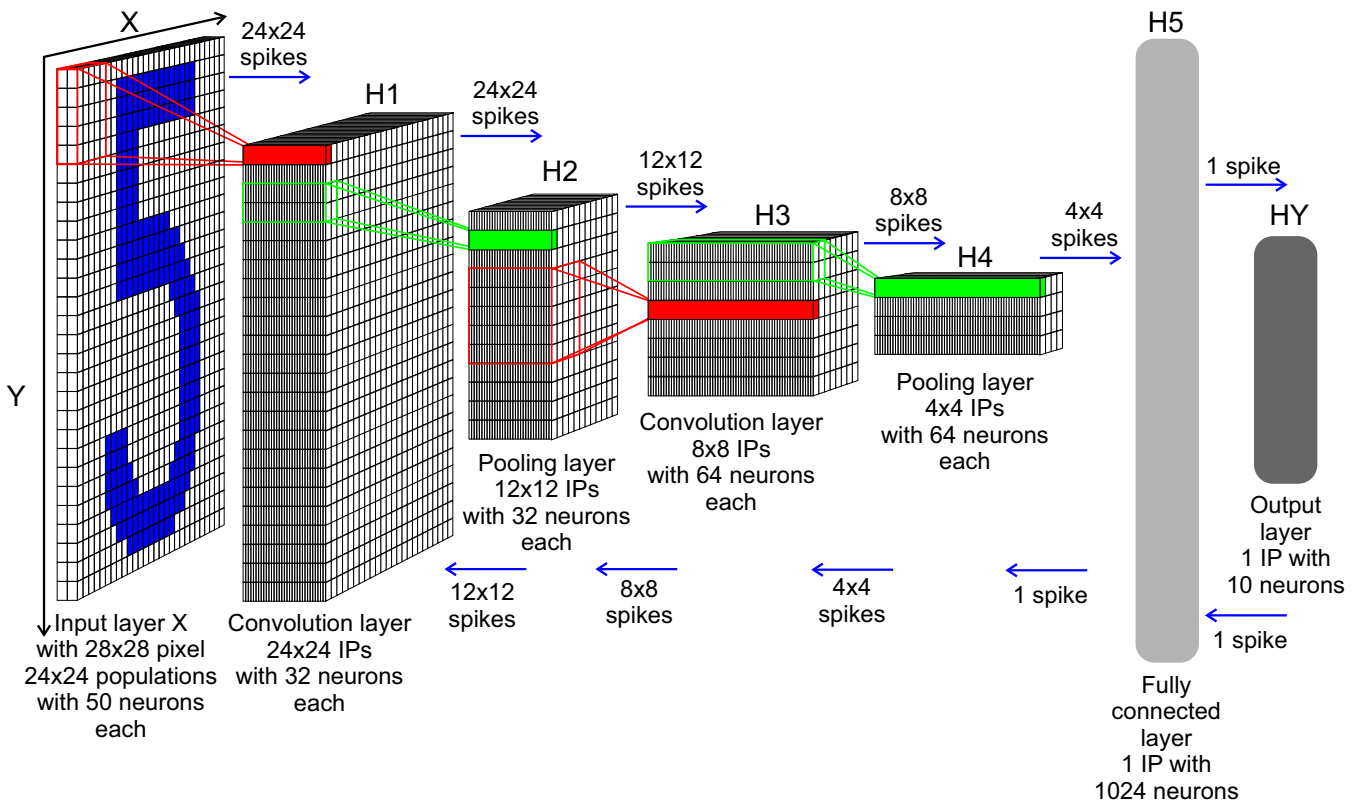


Figure 1. Example SbS network (see (Rotermund and Pawelzik, 2019a) for more details on this network) for analyzing handwritten digits (MNIST benchmark, <http://yann.lecun.com/exdb/mnist/>). The input image (28x28 gray value pixels) is represented by a 576 node input population and 802 parallel SbS IPs with different number of neurons (with 10, 32, 64, or 1024 neurons per SbS inference population IP, colored columns in layers H1-H4). Thus this network consists of 1378 independent computational elements. The network is organized in one input layer, two convolution layers, two pooling layers, one fully connected layer, and one output layer. All the layers, except the input layer, use the same update dynamic (i.e. updating the internal state of a SbS inference population with an incoming spike). In contrast to usual deep convolutional networks, there is no algorithmic difference for pooling layers, they only have a special arrangement of their weight matrices. Information between elements (SbS inference population and input populations) are exchanged only by spikes. Depending on the architecture of the network, the information can flow either only forward or bi-directionally (as indicated by the blue arrows). The latter can be used to implement local learning rules. In this specific network there are no interactions in between SbS inference populations within the same layer, but in a more complex network they could be helpful.

76 Figure 1 shows an example SbS network architecture for classifying MNIST benchmark handwritten
 77 digits. The shown network consists of several layers. Each layer can perform different computations
 78 like convolution, pooling or other learned functions. These layers are all constructed from many IPs
 79 and all the IPs realize the same algorithm for updating the latent variables based on arriving spikes as
 80 well as learning the weights. The spikes in such a network can flow either only forward, from input to
 81 output, (Rotermund and Pawelzik, 2019a) or spikes can flow in both direction to e.g. neighboring layers
 82 (Rotermund and Pawelzik, 2019b). In this example, spikes are only exchanged between IPs from different
 83 layers but not between IPs in the same layer. On a more abstract level such a network can be understood
 84 as pools of IPs and input populations, an architecture that defines which of these populations are allowed
 85 to interact, and spikes traveling between IPs.

86 In the supplemental materials, we have summarized the stylized facts of the SbS algorithm. The
 87 important equation for the design of the hardware are recapitulated in section 2.1.

88 While SbS networks are far less computationally demanding than networks with more detailed spiking
89 neuron models, e.g. leaky integrate-and-fire neurons (IaF), it still requires high computational effort,
90 especially if it is compared to a perceptron-based deep neuronal network. The reduction of the required
91 computing power compared to an IaF is a direct result of time progressing in a SbS network only from
92 one spike to the next. In between spikes, there are no operations happening in a SbS network; formally
93 this time does not exist in such a network. Thus biological realism is traded in for bigger networks that
94 require less computational effort for running them.

95 However, simulating a SbS network, in particular deep ones with many inference populations for large
96 data sets, is a problem when only normal computer CPUs, GPUs or even computer clusters are available.
97 Optimally simulating a SbS network benefits from a large number of parallel cores with a medium amount
98 of non-shared & directly accessible memory. Every IP in a SbS network is a compact local module that
99 only communicates via spikes with its environment. This allows to parallelize the whole network into
100 arbitrarily many parallel IPs. In the example of the MNIST network, shown in figure 1, it is best realized
101 by 802 individual threads (In bigger network, this number will significantly increase.). One thread per
102 IP would be the optimal solution. The information processing within IPs can even be asynchronous to
103 the rest of the network as long as the spikes between the populations can be exchanged successfully. The
104 design goal of the presented hardware is to create small and optimized computational units (SbS inference
105 populations) that simulate an IP. While it is possible to realize a few of these SbS inference populations
106 on a FPGA, the long term goal is to build ASICs (or networks of these ASICs) that allow to realize such
107 a network where every IP is represented by its own non-shared SbS inference population.

108 In the following we investigated how the algorithm can be realized as an ASIC. We tested the resulting
109 VHDL implementation with Xilinx Virtex 6 LX550T-1FF1759 FPGA on 4DSP (now abaco systems)
110 FM680 cards. The long term goal is to use this design for an ASIC, thus all FPGA specialized circuits
111 (e.g. hardware DSP cores) or other intellectual property cores were not used. The only exception to this
112 rule is block RAM (BRAM).

2 MATERIALS AND METHODS

113 2.1 Design goals

114 There are multiple ways to realize the required circuits. For example, the number of components
115 necessary for the arithmetic units can be minimized by using sequential processing as well as reusing
116 the arithmetic units as much as possible. Another approach would be to design the circuits such that
117 the calculations are done as fast as possible (measured in clock cycles) under the penalty of using much
118 more components in the process. The design goal for the results of this study is leaning strongly in the
119 latter direction by using pipeline designs for the arithmetic units. The reason for this decision lies in the
120 low clock speeds. Using FPGAs or ASICs allows us to design a computational environment for these
121 SbS networks that allow fast updates of the latent variable. For this task, this architecture allows better
122 parallelization than commercial general purpose CPUs.

123 For implementing a flexible multi-layer Spike-by-Spike network on chip, the following main ingredients
124 are required:

125 a) SbS inference population: Each such population consists of a population of neurons that are in a
126 competition which is realized by normalization. A inference population receives spikes as indices s of

127 sending populations and uses them to update internal states $h(i)$ of each population member i and weights
 128 $p(s|i)$ as well as produce outgoing spikes which reflect its internal latent variable $h(i)$.

129 b) Input populations: Input into a SbS network is given by probability distributions $p_{\mu,g}(s)$ represented
 130 by populations of neurons s , where μ denotes the actual pattern and g enumerates the input population.
 131 These probability distributions are used to generate spikes which are sent to SbS inference populations
 132 for further processing. Input patterns can be e.g. pixel images, time series, or waveforms. The input pattern
 133 is interpreted as a vector of numbers, which is transformed into a probability distribution by normalizing
 134 it according to the L1 norm. Every one of these normalized numbers is represented by an input neuron. The
 135 higher the value which is stored in the neuron, the higher is the probability that this neuron produces the
 136 next spike. The input pattern is provided from outside of the FPGA / ASIC from sensors (e.g. camera) or
 137 from data storage devices and programmed into the input neurons via a data bus. Typically the probability
 138 distribution stored in the input neurons doesn't change for an externally defined number of spikes (until
 139 the computation of this input pattern is done, then it is usurped by a new probability distribution). The
 140 neurons in the input populations don't react to any spikes produced in the SbS network. Thus they can be
 141 considered simplified versions of the inference population where only the spike generating part remained
 142 but without any weights, learning or updating of internal variables through spikes.

143 c) Network communication fabric: Spikes need to be exchanged between populations.

144 Focusing on the calculations that a SbS inference population needs to perform, a set of equations have
 145 to be realized: The basic equation realizes the update dynamic for the latent variables $h(i)$ based on
 146 an incoming spike. In (Ernst et al., 2007) it was shown that only the identity (i.e. the index) s^t of the
 147 subsequently active neuron needs to be taken into account:

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

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

154 For online learning we focus on a multiplicative learning rule for the weights $p(s|i)$:

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

155 where γ is a learning parameter which can change during learning.

156 For batch learning, a variety of implementations exist, we focus on batch learning rules that base on

$$\Delta p(s|i) = \sum_{\mu} \frac{h_{\mu}^t(i) \hat{p}_{\mu}(s) p(s|i)}{\sum_j h_{\mu}^t(j) p(s|j)} \quad (3)$$

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

$$p^{new}(s|i) = \mathcal{U} [p(s|i), \Delta p(s|i)]. \quad (4)$$

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

$$p^{new'}(s|i) = (1 - \alpha)p(s|i) + \alpha \sum_{\mu} \frac{h_{\mu}^t(i)\hat{p}_{\mu}(s)p(s|i)}{\sum_j h_{\mu}^t(j)p(s|j)} \quad (5)$$
$$p^{new}(s|i) = \frac{p^{new'}(s|i)}{\sum_r p^{new'}(r|i)}.$$

163 \mathcal{U} could also be realized by Adam (Kingma and Ba, 2014) or L4 (Rolinek and Martius, 2018) using
164 mini-batches. Since $\Delta p(s|i)$ is based on a multitude of patterns, offloading $\Delta p(s|i)$ to a separate CPU
165 and performing the weight update on the CPU occurs at a lower rate than all other operations. Thus the
166 reduction in performance by using a CPU for handling the weight updates is outweighed by the gain of
167 flexibility.

168 Furthermore, we need the means to generate spikes from probability distributions or the latent variables
169 using random numbers. Also specialized circuits are required for observing the spikes entering a
170 SbS inference population and calculating the corresponding probability distribution $\hat{p}_\mu(s)$ from these
171 observations.

172 **2.2 Non-negative numbers**

173 Investigating the three main equations 1, 2 and 3 reveals that no subtractions are required and that
174 no negative numbers appear. Furthermore most numbers (especially $h(i)$ and $p(s|i)$) are in the range
175 of $[0, \dots, 1]$. Incorporating these facts into the arithmetic units allows to simplify to the usual designs
176 (Shirazi et al., 1995).

177 Taking the memory structure of the Xilinx Virtex 6 FPGA into account, where the block memory is
178 organized into blocks of multiples of 1024 words with 18 bits each, we decided to used a custom variant
179 of 36 bit floating point numbers as well as 18 bit fixed point numbers ($X = \frac{X_{Int}}{2^{18-1}}$).

180 In more detail, we designed the 36 bit floating point numbers as follows: Since only non-negative
181 numbers are used, the usual sign bit was not necessary anymore. As part of batch learning, calculations on
182 a typical CPU occur. This requires an easy way to convert our floating point numbers into their IEEE 747
183 counterpart. Thus we keep the number of bits for exponents, removed the sign bit and appended 5 extra bits
184 to the lower significant bits of the mantissa. This allows conversion to be performed just by removing the
185 not required bits of the mantissa or filling the extra bits of the mantissa up with zeros. We also introduced
186 a similar derivative for 64 bit floating numbers (double precision) with 72 bits. Here 9 additional bits were
187 added to the mantissa. These 72 bit floating point numbers are only used for representing $\Delta p(s|i)$ because
188 summing over the contributions from larger amounts of patterns may otherwise lead to a degeneration of
189 precision. For this reason, a similar extension from 18 bits to 36 bits was done for the fixed point numbers.

190 Figure S1 shows the coding for the two types of floating point numbers used in the design. The bits
191 marked with gray boxes as well as the not available sign bit are different from the IEEE 747 standard. To
192 simplify the floating point arithmetic units even more, only 0 as a sub-normalized number is allowed. In
193 the supplemental materials, an investigation is presented where the impact of different representations on
194 the performance of the MNIST SbS example network is analyzed.

195 **2.3 Analyzing the equation's structure**

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

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

198 ζ can be ϵ , γ or $\hat{p}(s)$. All three variants of ζ have in common that they change slower than i , which means
199 that i can change many times before ζ is changed once. Also the s in $p(s|i)$ changes slower than i . s can
200 be selected through an observed spike s^t and only changes after all latent variables of that SbS inference
201 population have been updated. Furthermore, for batch learning s should be selected as the slower changing
202 index because this reduces the amount of times $\hat{p}(s)$ needs to be calculated.

203 Since memory is a scarce source on FPGAs and ASICs, it is advantageous to recall all the corresponding
204 $h(i)$ and $p(s|i)$ pairs twice from memory during calculating $\Omega(i; s; \zeta)$ while s is fixed. Since the arithmetic
205 operation Y/X is known to be computational demanding, the following approach was chosen:

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

209 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
210 the output from the multiplication pipeline. In the case of floating point numbers, adding two numbers
211 take longer than two clock cycles. Figure S5a shows the steps necessary for adding these numbers and
212 every step requires one clock cycle in our design. Thus we found that it is beneficial to implement the
213 summation operation via an addition pipeline, where the output of the addition pipeline is feedback to its
214 input (this procedure will be explained in detail later).

215 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
216 required amount of components for this arithmetic operation lower while a pipeline approach wouldn't
217 lead to a faster calculation anyhow.

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

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

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

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

3 RESULTS

231 3.1 Computational building blocks

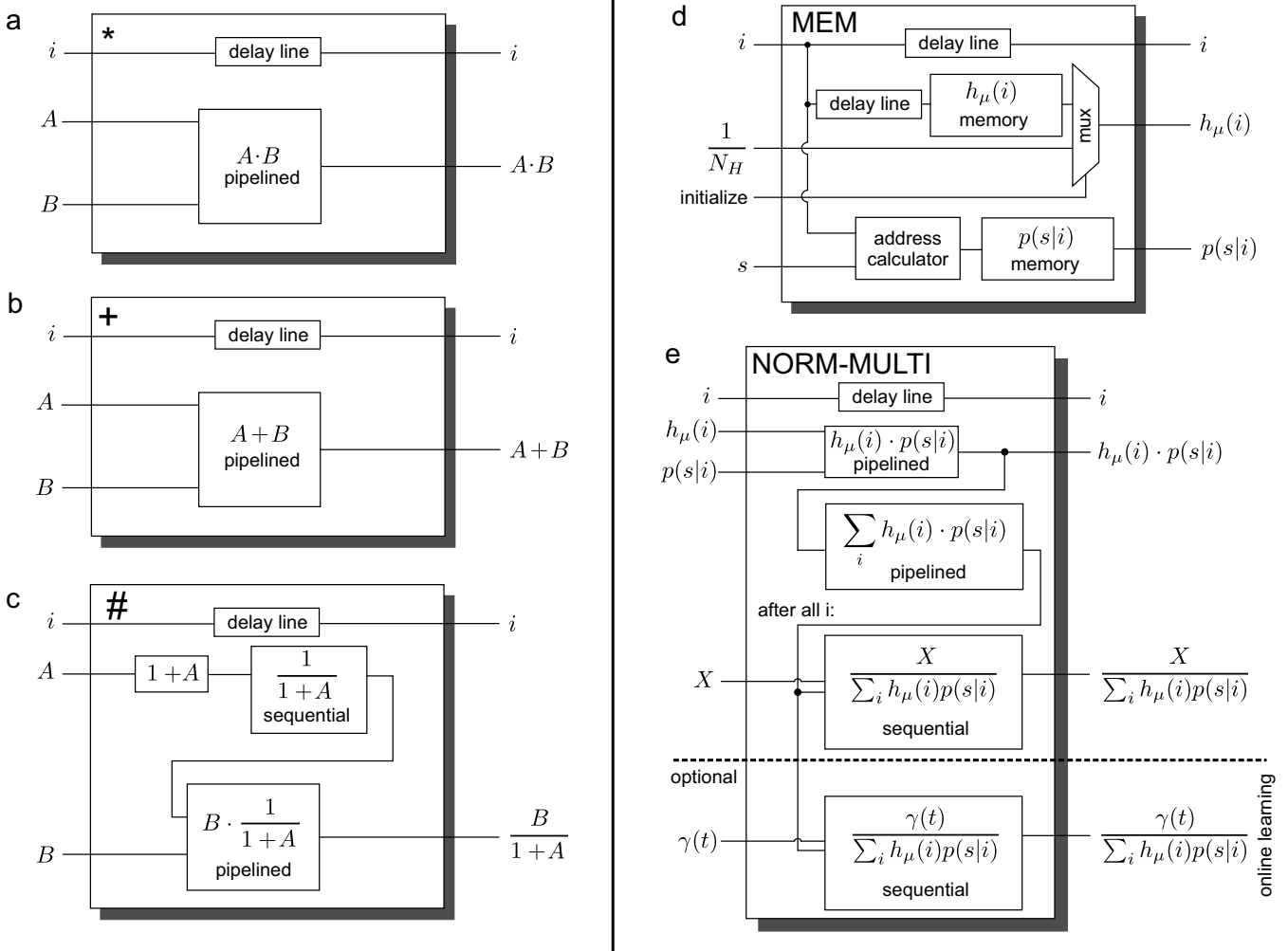


Figure 2. Basic computational building blocks: a.) **Module ***, b.) **Module +**, c.) **Module #**, d.) **Module MEM**, and e.) **Module NORM-MULTI**.

232 Five basic building block can be identified for implementing the three equations. All five modules have
 233 in common that input entering the module is accompanied by an organizational index i . Delay lines in
 234 the modules ensure that this index labels its corresponding output. Figure 2 shows an overview of the
 235 computations performed in these modules. Table S1 lists the components required to map the circuits
 236 onto Xilinx Virtex 6 FPGA hardware. Resources listed are Slice Registers (687360 available on the used
 237 LX550T version), Look-Up-Tables (LUT, 343680 available), LUT Flip Flop pairs used (85920 available),
 238 and block Random-Access Memory (BRAM, 1264 blocks with 18k bits each or organized as 632 blocks
 239 with 36k bits). Every type of these resources represent highly complex circuitry (see Xilinx user guide

240 UG364 'Virtex-6 FPGA Configurable Logic Block Table' for more details). Every pipelined operation is
241 designed such that one new operation can be started per clock cycle. However, it typically takes more
242 than one clock cycle (also called latency) before the result of an operation leaves its pipeline. Furthermore
243 some modules contain sequential components that need a given number of clock cycles before they are
244 initialized for their task and can perform their actual computation. Table S2 gives details concerning the
245 latencies or how long the operations take.

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

249 **Module +** (figure 2b) represents a simple adding pipeline. Two inputs A and B enter the module and
250 $A + B$ leaves the module. In the case of fixed point numbers, the output has one bit more than the input.

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

257 **Module MEM** (figure 2d) stores the latent variables $h(i)$ and the corresponding weights $p(s|i)$. The
258 main components of this module are two **block RAM modules** (which can have different sizes), which
259 provide a vector of memory values each. For each of the **block RAM modules**, only one reading operation
260 and one writing operation can be done in parallel during one clock cycle. While the access of the $h(i)$
261 memory could be done directly, for $p(s|i)$ the memory location needs to be calculated via an address
262 calculator from i and s first (this is done according to the equation: Linear memory position(s, i) =
263 $s + N_S \cdot i$ with N_S as the biggest possible s plus one if s and i are zero-based variables.). Thus the module
264 contains two address calculators, one for writing and one for reading operations. Since the calculation of
265 the memory address takes some clock cycles (one multiplication and one addition) and the output of the
266 module are pairs of $h(i)$ and $p(s|i)$ based on the same i , read requests to $h(i)$ memory are appropriately
267 delayed. Figure 2d shows in a simplified fashion the reading part of the module, for which a multiplexer
268 is placed at the output of the $h(i)$ memory. This multiplexer allows to usurp the $h(i)$ memory output by
269 an initialization value without writing it into $h(i)$ memory first. For floating point numbers, at the input
270 and output of the module, the first bit of the mantissa is removed or re-added (see figure S1). This bit is
271 necessary for calculations but is directly defined by the exponent and hence doesn't need to be stored in
272 memory.

273 **Module NORM-MULTI** (figure 2e) is a modification of **Module ***. In addition to the multiplication
274 functionality, the cumulative sum over the output of the multiplication pipeline is calculated. In case of
275 fixed point numbers, adding two numbers can be done in one clock cycle. However, for floating point
276 numbers it takes several clock cycles for adding two numbers. In combination with receiving one new
277 output from the multiplication pipeline in every clock cycle, this poses a problem. As a solution, we used
278 an adder pipeline feedback on itself (see figure S5). The cumulative sum goes through three stages for
279 floating point numbers: For stage 1 and 2, input A into the adder pipeline is defined by the output of the
280 multiplier pipeline. In stage 1, which lasts as many clock cycles as required to pass through the adder
281 pipeline, B are set to 0. Then stage 2 is entered where B is set to the output of the adder pipeline and thus
282 creates a kind of circular buffer. After the last input value from the multiplication pipeline's output was

283 received (defined by an external constant), stage 3 is entered. In stage 3 A and B is set to 0 by default.
284 The output of the adder pipeline is collected until two valid outputs are available. These two values are
285 send as an A - B -pair through the adder pipeline. This is done several times and thus combines more and
286 more remaining pairs until one value is left, which is the desired output of the cumulative sum. Finally,
287 an external value (in figure 2e X is a placeholder for ϵ or $\hat{p}(s)$) is divided by the actual cumulative sum in
288 a sequential way. In the case of online learning the weights it is beneficial to perform a second division
289 with the learning parameter γ in parallel. A note concerning the number of bits for the fixed point case:
290 the cumulative sum can not exceed 1 since $h(i)$ is normalized to 1 and the values of the weights obey
291 $0 \leq p(s|i) \leq 1$. Since ϵ values which are larger than 1 can be interesting for some applications, the design
292 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.

293 Besides the basic computation building blocks for calculating the three main equations, additional
294 modules are required for realizing the network. In particular a module that allows to generate spikes from
295 a probability distribution and a random number, a module that analyzes spikes and calculates a rate $\hat{p}(s)$
296 out of them, and a module that offsets the weights in a normalized fashion during learning. The figures
297 S2, S3 and S4 show simplified schematics for these modules. Table S3 shows the amount of components
298 necessary to build them as well as the number of clock cycles they require.

299 **Module Spike Generator** (figure S2a) converts a probability distribution and one random number into
300 one spike. A spike is an index describing a position in the probability distribution that elicited the spike.

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

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

311 **Module Spike Generator with offset** (figure S4a): Sometimes, especially during annealing while
312 learning, it is helpful to add an offset α to a probability distribution and to fade out this offset over time.
313 One way to generate spikes with an offset is to add the offset to the probability distribution, re-normalize
314 it and then draw spikes from it. In the context of these circuit designs, a more efficient way is to rely on a
315 double stochastic process using two random numbers. One random number is used to draw one spike from
316 the original probability distribution without offset and one spike from a uniform probability distribution.
317 The second random number is compared to α . The outcome of this comparison decides which one of the
318 two spikes is used.

319 **Random number generator** (figure S3): For generating spikes, we need random numbers. For generating
320 those, we use a Mersenne twister (MT 19937) which produces 32 bit random numbers (with a period
321 of 2^{19937}). A requirement for the MT 19937 implementation was that it needs to produce one random
322 number in every clock cycle. Several different designs for such MT circuits are available. We decided to
323 combine the designs from (Tian and Benkrid, 2009) and (Saraf and Bazargan, 2017). Our design uses two
324 block RAM (624x 32bit words each) and 624bit logic RAM as memory. While it uses one more block
325 RAM than such an implementation ultimately needs, it allows us to produce exactly the random number

326 series in the order produced by Matlab or C++. Our **MT modules** also contains a computational unit that
327 calculates the initial table after reset given a seed value.

328 **Module Rate Calculator** (figure S2b): For batch learning the weights, the input rate $\hat{p}(s)$ is necessary.
329 For calculating the rate $\hat{p}(s)$, the following is required: The total number of spikes c_{All} observed and the
330 number of spikes seen in each channel s counted as $c(s)$. After an externally defined number of spikes
331 has been counted, the rate is calculated and recalled by an external module. This is realized by calculating
332 $\frac{1}{c_{All}}$ first and then multiplying it by $c(s)$. Depending on the required representation of the rates, conversion
333 into other number formats may be necessary.

334 **Module Normalized Weight Offset** (figure S2b): During learning it may be necessary to keep the
335 weights $p(s|i)$ away from 0 (otherwise the multiplicative learning algorithm may get stuck). Hence this
336 module adds an offset $\frac{\varphi}{N_S}$ to each weight while they pass through this module. However, the module
337 must also ensure that the normalization of the weights $\sum_s p(s|i) = 1$ is still kept intact. Thus the module
338 also divides the intermediate weight by $\frac{1}{1+\varphi}$ before it exits this module. In addition, the corresponding
339 $h(i)$ values are delayed to keep them in sync with their $p(s|i)$ counterparts. The **module Normalized**
340 **Weight Offset** is directly placed in series after the **MEM module**. To keep the figures 3, 4, and 5 clear,
341 we omit showing the **Normalized Weight Offset module** and show only the **MEM module** and not the
342 combination of both modules.

343 **3.2 Circuits for updating $h(i)$, online and batch learning $p(s|i)$**

344 With the presented building blocks it is now possible to implement the three equations 1, 2 and 3 as
345 circuit diagrams. Concerning the update of $h(i)$ based on an observed spike s , figure 3 shows how the
346 building blocks are used. The update is done in two stages, during which the spike s is constant. In the
347 first stage, the index i counts through all allowed values $1, \dots, N_H$. This recalls N_H pairs of $h(i)$ and
348 $p(s|i)$ values. After an optional normalized offset is added to the weights $p(s|i)$, these pairs a feed into
349 a **NORM-MULTI module**. As a result $\frac{\epsilon}{\sum_j h(j)p(s|j)}$ is calculated. Now stage two begins, with recalling
350 the $h(i)$ and $p(s|i)$ a second time from the memory. In this stage, the output of the multiplier pipeline
351 $h(i) \cdot p(s|i)$ of the **module NORM-MULTI** is combined with the previous calculated $\frac{\epsilon}{\sum_j h(j)p(s|j)}$ via
352 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
353 pipeline to its $h(i)$ which was delayed to be available at the right moment in time. As a last processing step,
354 the output of the addition pipeline is divided by $\frac{1}{1+\epsilon}$ via a **Module #**. The output of the **module #** delivers
355 the new $h(i)$ values. These are then written into the $h(i)$ memory of the **module MEM**. Furthermore a
356 **spike generation module** (we used the 'with offset' variety in our design) observes the new $h(i)$ values
357 in parallel and, if provided with a random number, draws a spike out of the new $h(i)$ distribution. Table
358 S4 shows the number of components required to map this circuit onto the Xilinx Virtex 6 FPGA as well
359 as how many clock cycles of latency the **modules** *, + and # adds to the processing time.

360 Measured from the time then the first pair of $h(i)$ and $p(s|i)$ values are requested from **module MEM**
361 until the clock cycle when the last updated $h(i)$ values is written into memory, the $h(i)$ update takes
362 $179 + 2 \cdot N_H$ clock cycles for floating point numbers and $118 + 2 \cdot N_H$ clock cycles for 18bit fixed point
363 numbers.

364 In parallel to the $h(i)$ update, an online update step for the weights $p(s|i)$ can be performed. Figure
365 4 shows the three stage process for doing so. Stage one and two of the $h(i)$ -update and online $p(s|i)$ -
366 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

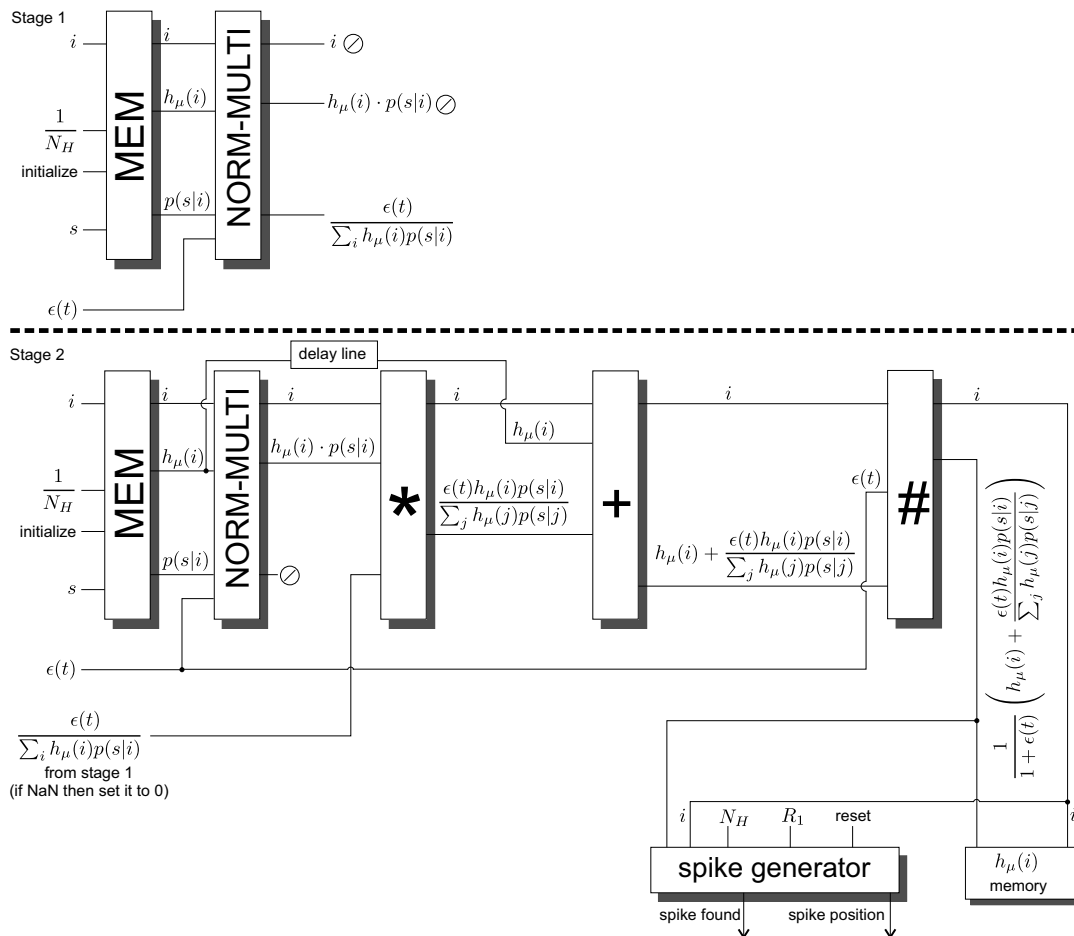


Figure 3. Circuits for updating $h(i)$ based on a spike s . The update is done in two stage.

367 in parallel. In stage two, the output $h(i) \cdot p(s|i)$ of the **module NORM-MULTI** from the $h(i)$ -path is
 368 multiplied via another multiplication pipeline with $\frac{\gamma h(i)p(s|i)}{\sum_j h(j)p(s|j)}$ in parallel. This results in $\frac{\gamma h(i)p(s|i)}{\sum_j h(j)p(s|j)}$ as
 369 an intermediate result which is stored in a temporary memory $U(i)$. Furthermore the spike s is stored as s_t
 370 for stage three. In stage three the old weight values need to be updated with $U(i)$ and stored as new weight
 371 values into their memory in **module MEM**. For this process all $p(s|i)$ have to be recalled from **module**
 372 **MEM** (and optionally modified with an normalized offset via the corresponding **offset module**) but with
 373 i as the slower changing index. For a given index i the **module #** is prepared with the corresponding
 374 $U(i)$. After this preparation is finished, the index s counts from 1 to N_S . In a first step, it is checked if
 375 s equals s_t . If this comparison is true, an addition pipeline adds $U(i)$ to $p(s|i)$. Otherwise 0 is added to
 376 $p(s|i)$. The output of the addition pipeline is then divided by $1 + U(i)$ via the already prepared **module**
 377 **#**. This result is written as the new $p(s|i)$ values into their memory in **module MEM**. The components
 378 required to realize state three and the temporary memory $U(i)$ is listed in table S4.

379 Since stage one and two of the online $p(s|i)$ update are done in parallel to the $h(i)$ -update and with less
 380 complex calculations, the number of clock cycles required for stage one and two are defined by the $h(i)$ -
 381 update. Concerning stage three, we measured $N_H \cdot (124 + N_S)$ clock cycles for floating point numbers and
 382 $N_H \cdot (91 + N_S)$ clock cycles for 18bit fixed point numbers. It needs to be noted that the reported
 383 clock cycle count for the $h(i)$ -path and one $p(s|i)$ -path were measured with the **normalized weight offset**
 384 **module** in place. Furthermore, it is important to point out that the listed 124 and 91 clock cycles are

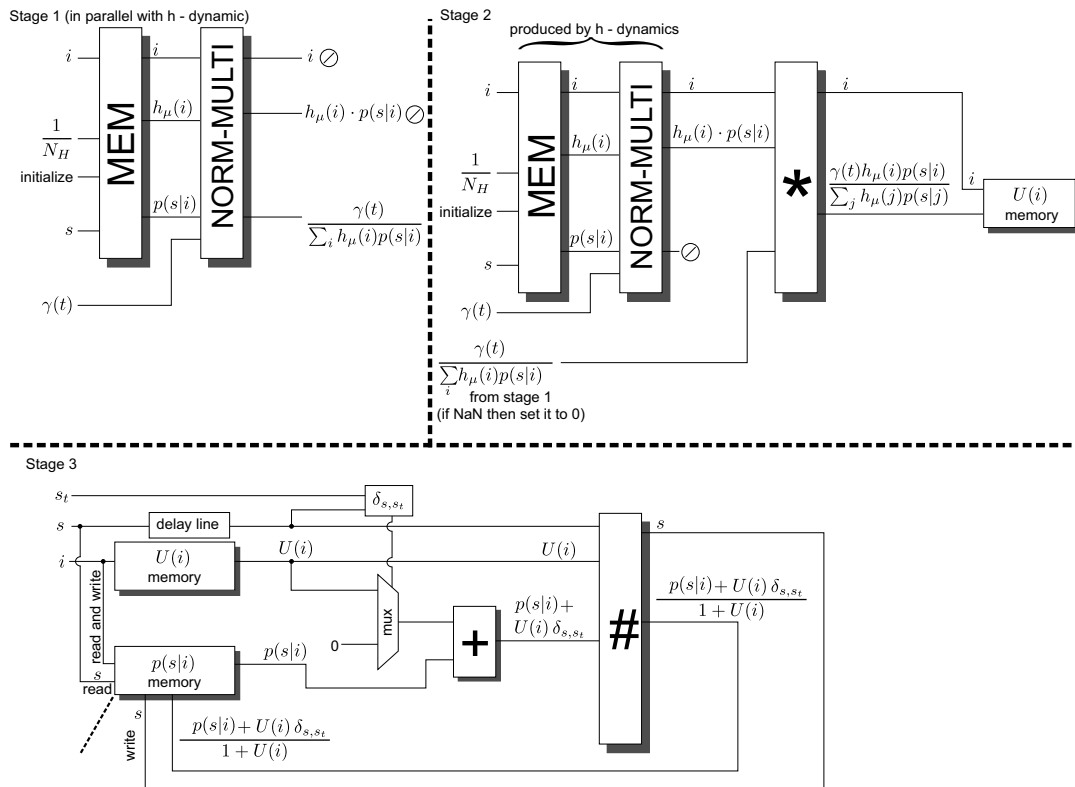


Figure 4. Circuits for updating the weights $p(s|i)$ via online learning. The update of $p(s|i)$ is done in a three stage process.

385 mainly a result of the time during which the **module** # needs to be prepared with the actual $U(i)$. Using
 386 additional **modules** # and switching between them would reduce these clock cycle counts significantly,
 387 however, would cost much more components.

388 Another approach to learn weights is to perform batch learning. In this learning mode, for a given
 389 input pattern $p(s)$, $h(i)$ is updated with many spikes s . Based on the final $h(i)$ and the probability
 390 distribution $p(s)$, for generating the observed spikes s in the first place, a contribution to the update of the
 391 weights from the actual pattern is calculated. Since $p(s)$ is not available to this circuit, an estimate $\hat{p}(s)$
 392 is calculated via the **module Rate Calculator** from the observed spikes s . The circuit processes several
 393 patterns and accumulates these individual contributions into a matrix $W(s|i)$. After the scheduled patterns
 394 are processed and their contributions are collected, $W(s|i)$ is sent to an external CPU for updating the
 395 weights $p(s|i)$. Using an external CPU for this purpose allows for a high flexibility on the realized update
 396 rule. Since the collected contributions can stem from a large number of patterns, $W(s|i)$ is realized with
 397 twice the bits as $p(s|i)$ for accommodating much larger numbers. The circuit for calculating $W(s|i)$ in
 398 a two stage process is shown in figure 5. The required amount of components for stage two are listed in
 399 table S4.

400 For stage one of adding a pattern's information to $W(s|i)$, the circuitry of the $h(i)$ -update can be
 401 reused, since there are no $h(i)$ -update during this time. First, $\hat{p}(s)$ is calculated for one s by **module**
 402 **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
 403 to stage two of the online $p(s|i)$ update path, a multiplexer pipeline is used to create the intermediate
 404 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

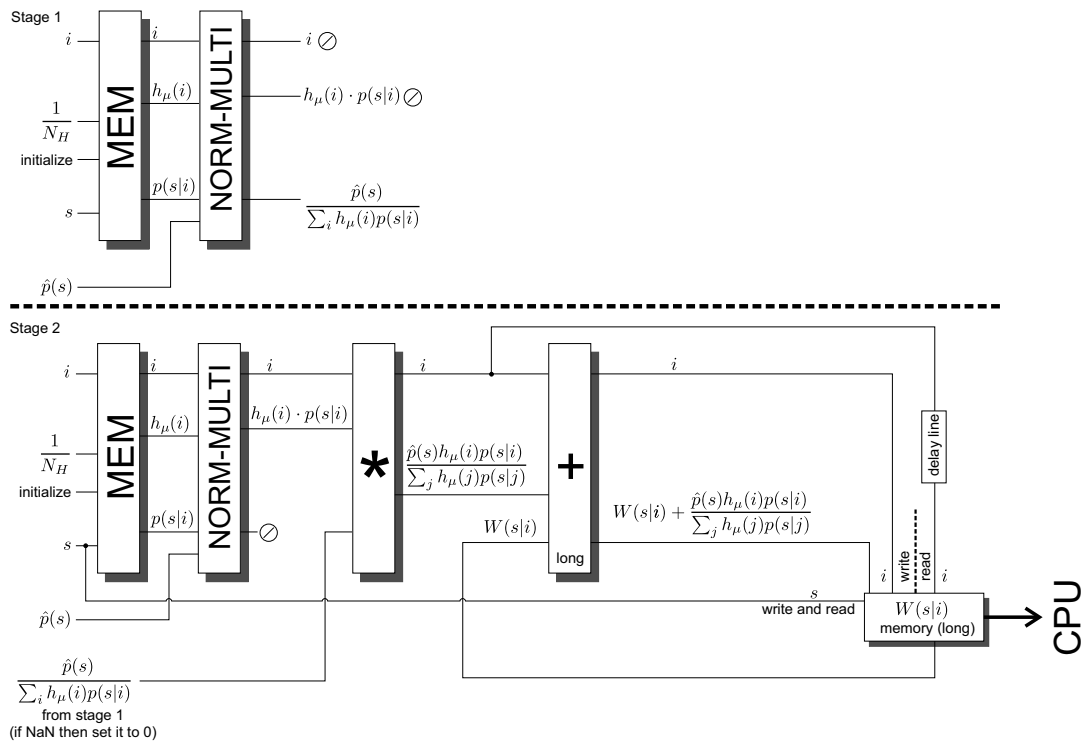


Figure 5. Circuits for batch learning. The matrix $W(s|i)$ is calculated from many spikes and input patterns. The $W(s|i)$ is send to an external CPU and used to calculate the new weights $p(s|i)$. The calculation of $W(s|i)$ is done in two stages.

405 corresponding number format with more bits and added to the content of the matrix $W(s|i)$. After all N_H
 406 indices i are processed, s is incremented by one and the next $\hat{p}(s)$ is recalled. For this new s the process
 407 starts at stage one again. This sequence of fetching $\hat{p}(s)$, stage one and stage two is repeated until it went
 408 through all N_S index values of s . Altogether this takes $51 + N_S \cdot (185 + 2 \cdot N_H)$ clock cycles for floating
 409 point numbers and $48 + N_S \cdot (124 + 2 \cdot N_H)$ clock cycles for 18bit fixed point numbers.

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

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

421 For floating point numbers we found a maximum relative (difference divided by the Matlab simulation
 422 result) error around $3 \cdot 10^{-8}$ for $h(i)$, $p(s|i)$ and $W(s|i)$. For the 18bit floating point numbers, calculating
 423 the maximum absolute difference for $h(i)$, $p(s|i)$ and $W(s|i)$ between the FPGA implementation and the
 424 corresponding Matlab simulation resulted in single digit differences, typically in the range from 0 to 6 for

425 numbers in the range of $[0, 2^{18} - 1]$. These differences are a result of combining many differences created
 426 by neglecting rounding the results for divisions.

427 **3.3 Connecting Spike-By-Spike inference populations and input populations**

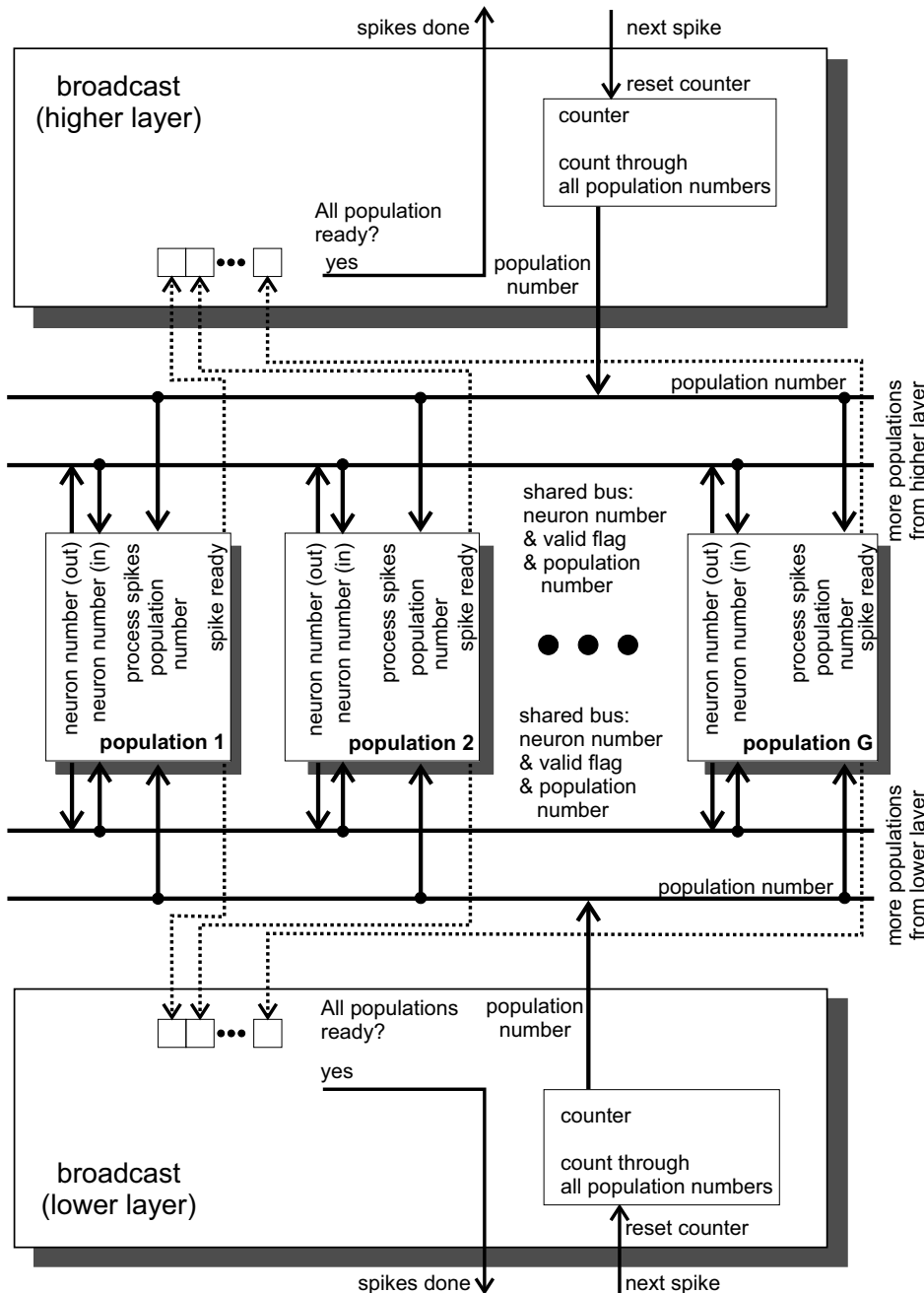


Figure 6. Communication between spike-processing elements in the network via broadcasts.

428 Typically, a network consists of more than one Spike-By-Spike inference population which need to
 429 exchange spikes. Or the task requires input patterns that need to be converted into spikes, which can
 430 be accomplished by an input population (a simple combination of block RAM for the input probability
 431 distribution and a **module Spike Generation** for drawing spikes; see table S5 for a component count for a
 432 probability distribution of up to 1024 32bit values). As result a communication fabric between these spike

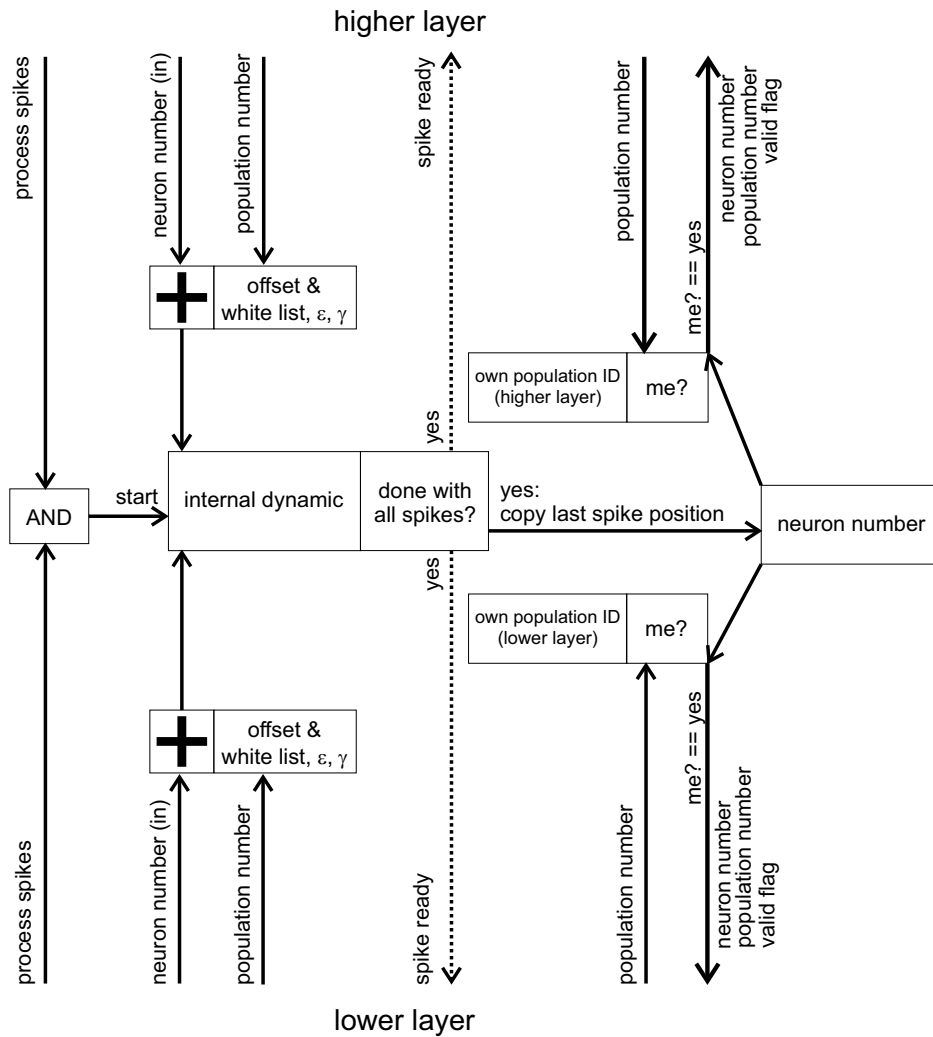


Figure 7. Handling incoming spike that enter a Spike-by-Spike inference population.

433 producing and spike received network elements is required. Figure 6 shows the communication fabric that
 434 was implemented into the presented design.

435 The communication fabric functions as follows: The input populations and Spike-By-Spike inference
 436 populations are connected to one or more shared data buses. The data bus conveys a unique identifier for
 437 the network element on that particular data bus, a neuron number for describing which neuron in that
 438 element produced a spike as well as a valid flag signaling that there is valid information on the bus at that
 439 moment. By default a network element is silent (all outputs are low). Every data bus owns a **broadcast**
 440 **module** (see table S5 for a component count) which coordinates the activity on its data bus. When all
 441 modules on the bus are ready for exchanging spikes, the **broadcast module** sequentially calls all unique
 442 identifiers of the elements connected to this bus. Every element that sees its identifier and has a spike to
 443 report, sends out its own identifier and neuron identity that caused the actual spike. After the exchange
 444 is performed, the **broadcast module** informs the connected network elements to process the information
 445 they received. The **broadcast module** itself is controlled from a controller on a higher level. The shared
 446 data bus is realized by XOR elements which ensures that only one network element at the time uses
 447 the data bus. For some applications it is helpful to use more than one data bus for selected groups of

448 network elements for keeping the required time to exchange spikes as low as possible. The presented
449 design realizes two data buses.

450 Concerning the bandwidth of this spike communication, for every spike three numbers are exchanged:
451 1) The question if a spikes was produced by population X (where X is a 10 bit number plus one valid
452 bit). 2) The answer of population X contains the 10 bit X and one valid bit itself as well as 3) the spike
453 s^t which is a 10 bit number too, which is indexing the neuron that produced the spike in population X .
454 However, population X only answers if it has a spike to report. The data bus is designed that transferring
455 the question as well as the answer uses only one clock cycle. The questions and answers are transmitted
456 non-blockingly in parallel on different parts of the bus. Assuming a population size of $N_H = 1024$, we
457 can expect the FPGA to process $\approx 50,000$ spikes per second and SbS inference population. For every
458 spike, a total of 4 Byte per connected population or 200 kByte per second and connected population are
459 exchanged.

460 While an input population doesn't react to spikes on the shared data bus, a Spike-By-Spike inference
461 population may need to update its internal variables according the observed spikes on the data bus. Figure
462 7 shows how incoming information on the data buses is handled by a Spike-By-Spike core. Assuming that
463 it sees a valid set of information on the data bus, it compares the network element's unique identifier with
464 a dynamically programmable white list (table S5 details the required component for such a list). In case
465 the unique identifier is on the list, the incoming information is processed further. Otherwise the incoming
466 information is ignored. For white-listed information, the neuron number for the spike is stored and a
467 flag that there was information received from this sending network is set. After the **broadcast module**
468 signals that all spikes have been exchanged and processing can begin (via a 'process spikes' line), the
469 Spike-by-Spike core goes through its white-lists (one per data bus) and processes all flagged entries. It
470 needs to be noted that the white list contains more information: An offset value that allows a dynamically
471 programmed offset on the spike index s for this source of spikes as well as individual ϵ and γ values for
472 every one of the white-listed spike sources.

473 All the spikes in the white-list are processed which causes an update of the Spike-By-Spike inference
474 population's $h(i)$. For every processed spike, automatically a new spike is drawn from the updated $h(i)$
475 probability distribution. However, only the last one of these spikes is stored. After all spikes are processed
476 a 'spike ready' line is raised, informing the **broadcasts modules** that a new spike is ready for exchange. In
477 the moment when the Spike-By-Spike inference population sees it's own unique identifier on the shared
478 bus, it puts its own identifier and the generated spike on the bus.

479 **3.4 Coordinating the networks and exchanging data**

480 While spikes are exchanged on specialized shared buses, also other data are required for operating
481 the network (e.g. input pattern distributions $p(s)$, weights $p(s|i)$, white-lists and random numbers).
482 Some data is generated inside the network, while other data needs to be provided from outside of the
483 network. Furthermore, data needs to be read out of the variables from the network (especially $h(i)$
484 distributions after processing input pattern as well as weight matrices $p(s|i)$ or $W(s|i)$ after learning)
485 at externally defined moments in time. Thus we implemented two shared 32bit data buses: The first one
486 for exchanging incoming external data as well as internally generated data (mainly random numbers and
487 control sequences). The second one for sending data (e.g. results or information about the status of issued
488 commands) to external receivers. See figure 8 for an overview.

489 This communication fabric was designed under the premise that information is loaded into the network
490 (weight matrices, parameters and input patterns) and then the network runs for a number of given spikes

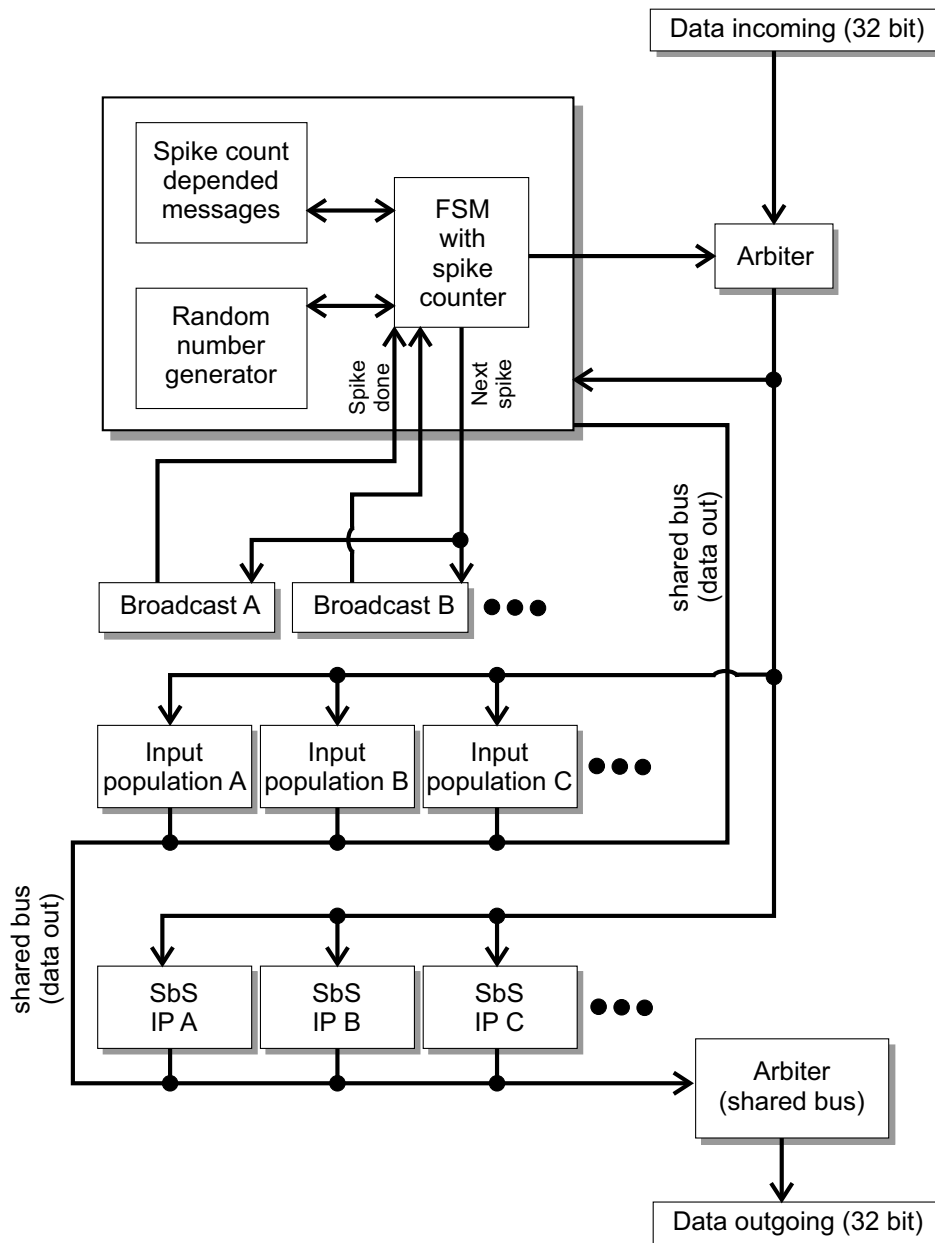


Figure 8. Data bus between the different elements as well as external participants.

491 (e.g. thousands of spikes). After that the results are readout from the network. New input pattern are
492 loaded into the system, the network processes a number of spikes again, and the results are readout from
493 the system again. It was not designed such that large amount of data is exchanged between every spike.
494 Scalability of the communication fabric can be archived by splitting the communication fabric into parallel
495 regions.

496 On the incoming data bus there are two sources for data: external data and internal data generated
497 by a **message control center**. The **message control center** has several tasks: a.) Based on externally
498 received information, it knows how many spikes the network needs to process as well as after which
499 amount of spikes it needs to start with online learning of the weights. Hence it controls the activity of
500 all the **broadcast modules**. b.) It contains a Mersenne twister random number generator. After a spike, it
501 provides all the network elements with new random numbers. It converts random numbers into messages

502 and puts these messages on the data bus which programs the network elements with these random numbers.
 503 c.) It contains a mailbox system that is controlled by the number of already processed spikes. This allows
 504 to store every possible – otherwise externally provided – data control sequence in this mailbox together
 505 with a number of processed spikes that will release the message at that moment on the data bus. E.g. this
 506 message system can be used to change parameters like ϵ during processing the pre-defined number of
 507 spikes without waiting of any external just-in-time changes of parameters.

508 One arbiter joins the two data streams from the incoming data. Another arbiter combines all data sources
 509 that are destined for an external data receiver. In table S6 the amount of components required to realize all
 510 the participants on these data buses are listed. The network’s parts know 17 commands (e.g. set parameters,
 511 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
 512 pattern probability distribution, set messages in the mailbox, and start the spike processing) that require
 513 up to nine 32bit words.

514 **3.5 Test with a Xilinx Virtex 6 FPGA**

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

Table 1. Amount of components required for a network with one Spike-By-Spike inference population and 25 input populations.

515 We designed circuits for the network (see table 1 for details) such that it can be mapped, placed and
 516 routed without any special FPGA components (e.g. DSP cores), except for **block RAM modules**, on
 517 a FPGA or ASIC. Our design, especially the number of buffering and pipeline stages, is optimized to
 518 generate a firmware that can be operated with 200+MHz clock speed on a Xilinx Virtex 6 LX550T-
 519 1FF1759 (speed grade: commercial 2). The Xilinx ISE 14.3 software is capable of producing such a
 520 firmware when the circuit is alone and the IO pins can be selected by the software.

521 However, we weren’t able to achieve these high clock rate when the network was embedded into the
 522 third party firmware which is required for communicating with the Virtex 6 FPGA on the 4DSP FM680
 523 PCIe card. We used the ’adder’ example from the training materials and simply replaced the ’adder’ core
 524 with our network. We allowed for a separate clock domain for the network by insulating the data-flow by
 525 dual clock FIFOs on the input and output side. Nevertheless, Xilinx ISE wasn’t able to provide us with a
 526 working firmware if we set the clock speed for the network to 200MHz. Thus we were forced to use the
 527 overall 125MHz clock provided by the 4DSPs example design also for our network.

528 With the 125MHz clock speed we were able to run the network on 4DSP FM680 cards under Linux
 529 (Centos 7.5 64bit with driver version 04.05.2018). Results were identically to the Xilinx ISim simulations.
 530 However, the overall design of the 4DSP FM680 doesn’t fit for our application very well. This type
 531 of card is designed for high data bandwidth applications where 64bit words in blocks of 1024bits are
 532 continuously exchanged. On one side we were forced to incooperate our 32bits into block of 1024bits. On
 533 the other side we had problems receiving data from the card because our network doesn’t produce data
 534 continuously. Thus the Linux driver and our software had to recover from trying to read from the card
 535 when it had no data for us. All this combined, slows down the communication with our network on the
 536 FPGA significantly.

537 In the supplemental materials we examine how the presented design fairs in comparison to Intel CPUs. In
538 addition, it is investigated how the performance in the SbS MNIST example network (see figure 1) reacts
539 to floating point and fixed point numbers with different number of bits. The conclusion of these tests is
540 that FXP IPs are an interesting option, especially for IPs with smaller number of neurons. However, it
541 needs to be tested if, in the given use-case, additional problems haven't been created by the use of FXP
542 IPs.

4 DISCUSSION AND CONCLUSION

543 The paper presents a design and an investigation of a circuitry optimized for implementing neuronal
544 networks based on Spike-By-Spike (SbS) neurons (Ernst et al., 2007). As in the brain, signaling is based
545 exclusively on spikes, interactions among local neuronal circuits are strictly non-negative (Dale's law), it
546 allows for recurrent interactions in cyclic architectures, and both, dynamics of the neuronal state variables
547 as well as the learning rules can be local (Rotermund and Pawelzik, 2019b). Thereby this approach
548 realizes a compromise between artificial neural networks and biologically realistic models employing
549 detailed models of spiking neurons.

550 In a typical spiking neuron models (Izhikevich, 2004) are simulated with fine temporal resolution. In
551 particular, simulations of networks of *noisy* leaky integrate-and-fire (IaF) neurons require representation
552 of real time. Typically, the membrane potential needs to become updated every time step dt which is often
553 in the range of sub-milliseconds. The number of updates between two spikes depends on the firing rate
554 of that neuron. If for example $dt = 0.1ms$ and the firing rate is 10Hz then this would translate roughly
555 into 1000 updates of the membrane potential. In contrast, the SbS-approach avoids simulations of real
556 time dynamics and would perform one update of a whole population between two input spikes. While the
557 different types of spiking neuron models (Izhikevich, 2004) have varying number of computations for one
558 update, in a SbS population with N neurons $3N$ multiplications, $2N$ summations, and one division are
559 used for one update of the whole population. This reduction in computational requirement is payed by a
560 decrease in biological realism.

561 In a SbS network time is only progressed with each spike received by an inference population (IP). Thus
562 no computations have to be performed in between spikes, which drastically reduces the computational
563 demand. An approach akin to the SbS's removal of real time is also known for integrate-and-fire neurons.
564 This so called event-based neuronal networks (e.g. Brette (2006, 2007); Serrano-Gotarredona et al. (2015);
565 Lagorce et al. (2015)) use analytic solutions of the neuron's dynamics to bridge the time between to spikes.
566 However, with stochastic neurons the event-based approach becomes problematic (Brette, 2007). This is
567 similar to the problem of finding an analytic solution for the first passage time (Burkitt, 2006a,b) for
568 neurons with stochastic inputs in a network, which is a hard problem. For SbS networks, the stochasticity
569 rather is feature because it corresponds to importance sampling of the input as well as the latent variables.
570 This acts as a filter for capturing the more dominant information in the network and suppress noise.

571 In SbS networks, neurons are organized in populations where the neurons within a population compete
572 with each other. Every neuron in a populations has a latent variable $h(i)$. The value of $h(i)$ is a positive
573 number and the competition is expressed by a normalization over all latent variables of a population
574 ($\sum h(i) = 1$). In between populations, neurons communicate exclusively via spikes (i.e. an index s_t
575 describing a single neuron's identity in a population at a time t). The connections between neurons from
576 different populations are described by weights $p(s|i)$ (with s as the emitting neuron and i as the receiving
577 neuron). Also the values of the weights are positive numbers and limited in size by a second normalization

578 condition ($\sum_s p(s|i) = 1$). Thus everything describing the state of the network is given by positive
579 numbers in the range of 0 to 1. In this paper we compared how different types of representations for these
580 numbers result in different sizes of computational circuits as well as different required numbers of clock
581 cycles. In particular we examined 18 bit positive fixed point numbers and 36 bit positive floating point
582 numbers. The latter are a custom derivation of the IEEE 754 standard by assuming positivity and a higher
583 bit count for the mantissa. The fixed point representation requires less components but it was found that
584 SbS inference populations can cause problems during learning, especially with larger numbers of neurons.

585 If a spike s^t is received by a population of neurons, for every neuron in this population an update of its
586 latent variables is calculated through

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

587 We presented computational circuitry for this equation that only requires to read the involved $h(i)$
588 and $p(s^t|i)$ value pairs twice during the update process. For floating point numbers a number of
589 $179 + 2 \cdot N_H$ clock cycles and $118 + 2 \cdot N_H$ clock cycles for the fixed point numbers were measured.
590 The constant numbers of clock cycles in these two equations don't scale with the number of neurons
591 because they represent the time which the information needs to travel through the pipeline structure of
592 the computational circuitry as well as setting up the circuits for incoming data. These amounts of clock
593 cycles can be reduced if optional features are removed, like e.g. adding an normalized offset to the weights.
594 Auxiliary circuits were designed such that, given a random number, they draw a new spike from $h(i)$ while
595 simultaneously updating the h-values. Furthermore it was shown that circuits for learning the weights (see
596 equation 2 and 3) can be designed, while keeping the amount of required components low by partially
597 reusing circuitry for the h-update.

598 Also for batch learning, circuits and memory are included into the presented design. Thus during batch
599 learning, the contributions from many training patterns are collected within the circuitry. In the design
600 phase of the system we decided against a hardware implementation of a specific learning rule that uses
601 this accumulated data and calculates updated weights from it. Instead the idea is to use a CPU for these
602 final weight update calculations, to allow large flexibility in choosing the 'right' variation of the learning
603 rule later and allow for changes in the learning rule if new learning methods arise in the community
604 without changing the hardware design. Combining FPGAs with CPUs has a long tradition, e.g. Intel
605 started to offer 2010 the Stellarton (Series E6x5C, Atom CPU with FPGA). The Xilinx Virtex-4 FPGA
606 from 2004 had versions with integrated PowerPC cores. At the same time, a soft core for the low-cost
607 Xilinx Spartan-3 FPGA (a 32 bit RISC processor called MicroBlaze) was released as intellectual property
608 core. This is also an approach which the neuromorphic community also adapted, e.g. (Wunderlich et al.,
609 2019; Naylor et al., 2013).

610 Given a collection of populations of SbS neurons, the exchange between populations of spikes requires
611 organization. In our framework we show that exchanging spike information can easily be done by a
612 common data bus. A controller checks if all spike generating elements on that bus are ready for exchanging
613 spikes and then calls every element on that bus to report its spike. Every SbS inference population has
614 a programmable list which defines the spike producing elements it listens to. This allows user-defined
615 changes of the network architecture on the fly. Also it allows to remove broken elements (which might
616 be e.g. a result of production failures) from the network and to reallocate – if available – other resources
617 to take its place. The lists contains additional information to allow for changes of the ϵ parameter for the

618 h-dynamic and the γ parameter for online learning depending on the source of the spike. This feature
619 is essential for more complex networks. In the same spirit, we introduced a 'mailbox system' into the
620 framework. It allows to make processing a given pattern for a user-defined number of spikes independent
621 from external sources. Sometimes it may be necessary to change the ϵ parameter after a given number
622 of spikes. Without the mailbox system, an external controller would need to determine the state of the
623 network (i.e. are the required number of spikes already processed?), then change the parameters, and tell
624 the network to continue for another amount of spikes. This would obviously waste time. In comparison,
625 with the mailbox system the network distributes the pre-defined messages about the parameter change
626 automatically in the moment a pre-defined number of spikes have been processed. Obviously, the same is
627 true for the required random numbers for producing spikes. Thus it is also essential that random numbers
628 are sent to the consumers of these random number with as small as possible pauses. For this reason it was
629 necessary to add a random number generator (Mersenne Twister 19937) to the overall message controller
630 with its mailbox system. The message controller also takes care that the correct number of spikes is
631 processed and that any external system is informed that the network finished the pre-programmed number
632 of spikes.

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

641 The presented design was transferred into a FPGA firmware written in VHDL. It was taken care that
642 no special FPGA vendor proprietary modules (except block RAM) were used, for allowing re-using the
643 design for developing a custom ASIC. For proof of feasibility, we integrated the VHDL code into an
644 example firmware for the 4DSP FM680 cards which hosts a Xilinx Virtex 6 FPGA from 2009. We
645 successfully generated a binary firmware bit file for this card, tested it under Linux and compared it
646 with the results from Matlab simulations. Small differences in our simple example simulation were found,
647 which were expected due to the difference in precision (floating point numbers: 5 more bits representing
648 the mantissa for the FPGA implementation compared to Matlab) as well as a different handling of
649 rounding during the mathematical operation of division. We decided that the differences through rounding
650 are not relevant, especially when compared to the required increase in the amount of components and
651 processing stages necessary for reaching a perfect match with the Matlab simulations. While the pure SbS
652 network would have been able to run with over 200MHz on an otherwise empty Virtex 6, combining it
653 with the existing card manufacturer's firmware unexpectedly forced us to reduce the clock speed down to
654 125MHz. Isolating the network into a separate 200MHz clock domain didn't work out. Since the VHDL
655 code was optimized for a 200MHz stand-alone implementation on a Virtex 6 LX550T FPGA, several
656 buffering steps in the pipeline architecture were added which are unnecessary for a 125MHz operation. A
657 second SbS inference population was added to the firmware's network and tested out fine. Extrapolating
658 from the number of required components for one SbS inference population, we expected that we could
659 increase the number of SbS inference populations to 7 or 8 for floating point number on this Xilinx Virtex
660 6 LX550T. However, the generation of these firmwares failed in the Xilinx ISE design tool. Our guess
661 is that the reason lies in the fixed placement of the some components (especially block RAM) on the
662 FPGA. Increasing the number of SbS inference populations produces distances between components that

663 are too long for even 125MHz on this FPGA from 2009. We didn't pursued the increase in the number
664 of SbS inference populations any further, because we found that the 4DSP firmware, driver and software
665 framework for this card is optimized for continuously streaming of large bandwidth of data. However, in
666 our application the data is send in a stop-and-go fashion which doesn't ensure that data is always available.
667 Thus it was necessary to probe the cards for available data via blindly trying to read data from the card
668 and to rely on slow timeouts of the Linux driver if there was no data available at that exact moment.
669 Furthermore we had to fill up our data streams with zeros (up to 31x bit in zeros for filling up compared
670 to the payload) to conform to the required data format. In the end we decided to focus on transferring the
671 design onto a custom ASIC instead of optimizing the FPGA firmware.

672 The computationally attractive aspect of a SbS neuron based network is the property that populations
673 of neurons only communicate via spikes. Thereby every population of neurons operates independently
674 from all the other populations. The internal computation of any given population is based exclusively
675 on the incoming spikes and its internal variables. After finishing its computations every population of
676 neurons also only sends out spikes as a signal to other populations. This locality property is true for
677 updating the $h(i)$ values as well as for learning the weights $p(s|i)$. Thus the locality of its populations is
678 the key ingredient for massively parallelizing networks with such elements (populations). The optimal way
679 for parallelizing such networks is to provide independent memory and computational circuitry for every
680 individual population of neurons. With the specialized processors proposed in this paper one update cycle
681 in a large network of such populations does not scale significantly with the number of populations. This
682 stands in stark contrast to running such networks on clusters of computers where one would experience
683 latencies and transmission delays that scale much worse with the number of nodes due to the creation of
684 network packages, their transmission and their analysis by soft- and hardware.

685 In future we will design a custom ASIC for networks with larger numbers of SbS inference populations.
686 The desire for using ASICs compared to continuing using FPGAs stems from several reasons: The goal is
687 to realize as many SbS inference populations as possible and give every SbS population in a network its
688 own SbS inference population. However, this might require a network of ASICs. The main limiting factor
689 with FPGAs is the amount of available memory (block RAM) and the corresponding routing problems.
690 Every SbS inference population needs its own dual port memory, at least for its latent variables. On
691 a FPGA this kind memory comes in block RAM modules and these are fixed resources which are at
692 certain manufacturer defined positions. These RAM blocks are spread over the whole FPGA die. If a
693 SbS inference population needs more memory than one of these RAM modules can provide, several of
694 these RAM modules are connected. This is a big problem for routing and timing which can be strongly
695 experienced in our Virtex 6 FPGA design. It is a major factor in limiting the clock speed if several SbS
696 inference populations are realized on the FPGA. The required logical components and the block RAMs
697 are still available but the timing is too problematic due to the long distances. Furthermore, FPGAs are
698 general purpose chips and the SbS inference populations have different requirements: The ration of logic
699 circuitry to block RAM seems not to fit into the exception for general purpose case. Using ASICs allows
700 us to place what we want in such a way that everything that needs to interact in a fast way is in close
701 proximity (i.e. everything that is part of one SbS inference population) and everything that communicates
702 over the spike bus can be placed further apart (and this communication can be on a much slower clock
703 rate).

704 As a preparation we made sure that the design doesn't use any special 3rd-party intellectual property
705 cores. For improving the processing speed of the SbS inference population, it might be interesting for a
706 custom ASIC to use different clock domains for exchanging spike information between populations and

707 doing the calculations inside the SbS inference population. This would also ensure that the SbS inference
708 populations can retain high computational speeds even if the distances between the populations on the
709 chips for exchanging spike information increase. This even allows to extend the data bus for exchanging
710 the spikes over a multitude of chips. Furthermore, it is also not necessary that all SbS inference population
711 run at the same speed. It is enough that only the part of a population responsible for communicating with
712 the data bus for exchanging spike information is in sync.

713 The area on chip required for memory may be an issue for the custom ASIC. Concerning the weights,
714 in a convolution setting all populations in a layer use the same weight values. Thus in such an application
715 it could be suitable to use a common RAM for all populations in that layer. The requirement with shared
716 convolutional weights is as follows: For every update of the latent variables h , every SbS inference
717 population gets a spike s^t . For performing the required computations for the update, the weight vector
718 $p(s^t|i)$ for all $i \in [1, \dots, N_H]$ is required. For example in the MNIST example, the convolutional weight
719 matrix between the input and the first hidden layer has the size $N_S = 50 \times N_H = 32$. Thus $p(s^t|i)$
720 is a vector with 32 numbers. However, there are only 50 different versions of this 32 number vector.
721 On the other side, we have 576 individual SbS inference populations in the hidden layer 1, hence, 576
722 different spikes to process in every time step of the simulation with only 50 allowed values. The idea
723 for the shared RAM is that all 50 vectors are sequentially recalled from RAM and broadcast to the 576
724 SbS inference populations. Every one of these 576 SbS inference populations know their own spike s^t
725 and filter their 32 number vector from this common Sb data stream. Obviously it takes 50x longer to do
726 this broadcast procedure compared to the case where every SbS inference population has its own weight
727 matrix. However, this reduction in speed is traded in for reducing the requirement for weight memory
728 by a factor of 576. Furthermore, this would allow to use faster external memory modules which would
729 allow a broader broadcast bus (e.g. two or more vectors are broadcast in parallel). Or using custom RAM
730 that delivers the $p(s|i)$ values for all s during one read cycle in parallel would be beneficial, which would
731 create N_S streams of weight values in parallel. In this scenario, every SbS inference population would
732 have a multiplexer connected to the output stream of the RAM and would, according to the value of s^t ,
733 switch between the different weight value streams.

734 The performance we can achieve in the MNIST benchmark (Rotermund and Pawelzik, 2019a) is 99.3%
735 classification correct by using an error-back-propagation & momentum based learning rule developed for
736 SbS networks which takes the requirements non-negativity and normalization into account. The learning
737 rate was modulated by the expected remaining error estimated from the training data set. In addition,
738 the fully connected layer H5 was subjected to a drop-out variant during learning. This learning rule is
739 compatible with the presented hardware design. Using the same network architecture for a non-spiking
740 neural network (Rotermund and Pawelzik, 2019a) we found 99.2%. These performance values are also
741 reported to be comparable with traditional non-spiking convolutional neuronal networks (Tavanaei et al.,
742 2018).

743 Using the presented SbS MNIST network with a local learning rule which is directly based on the
744 gradient – like it is used in the supplemental materials – we still yield a performance of 97.3%. This
745 is roughly the same performance we got using a classical non-spiking convolutional neuronal networks
746 using the same network structure and as well with only the gradient (without any additional optimizers)
747 (Rotermund and Pawelzik, 2019a).

748 Comparing our performance values with other networks (e.g. see for a list of MNIST networks
749 (Tavanaei et al., 2018) and <http://yann.lecun.com/exdb/mnist>) is not as simple as comparing
750 the values. In our case we didn't optimized our network structure for the use of SbS inference populations.

751 We rather choose to re-use the network structure associated with a Tensor Flow Tutorial because this
752 gave us a base-line for a network design which our computer cluster was just been able to simulate.
753 Furthermore, we didn't used any input distortion methods (e.g. shifting, scaling, or rotating the input
754 pictures) for increasing the size of the training data set. The reason was that this would have been too much
755 for our computer cluster, like it would have been to optimize the parameters used in the SbS MNIST
756 network. Or in other words: The performances shown for the MNIST SbS network doesn't reflect what
757 a fully optimized SbS network might be capable to deliver. Typically the performance values shown for
758 neuronal networks are for an fully optimized network, like performances of 99.8% (Wan et al., 2013).

759 Due to the communication problem with the FPGA cards and the limited number of FPGA cards we
760 have available (we own two of those cards and their price was 16,000+ Euros each), we had to perform
761 the analysis of the MNIST network on a cluster of computer with Intel CPUs. It took our cluster 19 days
762 with shy of 400 cores to perform the presented MNIST simulations. On the two FPGA cards with one SbS
763 IP each, this would have taken approximately over a decade. This shows that the number of parallel cores
764 is key in operating SbS networks in a fast fashion.

765 Obviously this raises the question why we are not just continue to use normal CPUs or migrate to GPUs
766 instead of developing a special ASIC for that task. The two main aspects of the answer are parallelization
767 (i.e. as many cores as possible) and memory bottleneck.

768 In the case of CPUs, the results of the timing measurements showed that as long as the required
769 memory stays within the CPU's L1/L2/L3 cache, the multitude of available cores can be efficiently used
770 to simulating SbS IPs. The moment when the amount of memory required exceeds the CPU's cache,
771 the situation starts to change. Now the memory bandwidth to the external memory modules starts to
772 limit the SbS performance and the effective number of cores usable in every point of time dwindles.
773 Thus even if a large number of cores is available on a CPU, only a few cores can be supplied with the
774 required information continuously while the rest of the cores wait. This problem is amplified if one core is
775 responsible for simulating several SbS IPs. In this use-case the information for the different SbS IPs (latent
776 variables and weight matrices) need to be switched and loaded into the CPU's cache continuously. Thus
777 the memory bandwidth becomes the limiting factor again. In summary, CPUs are an excellent solution to
778 simulate SbS IPs as long as the number of SbS IPs don't exceed the core count of the CPU and as long as
779 the required data fits into the CPU's cache. For SbS IPs with large number of neurons and/or large weights
780 matrices CPUs are especially a good choice because of the integration of CPU caches with several dozens
781 of MByte.

782 There are special multi-core CPU ASICs with large core counts available. Examples are the Epiphany-
783 IV (Olofsson et al., 2014) 64 RISC cores with 2 MB on-chip distributed memory (presumably 32 kByte
784 per core) or the Epiphany-V (Olofsson, 2016) 1024 RISC cores with 64 MB on-chip distributed memory
785 (presumably 64 kByte per core). As long as the latent variables and the corresponding weight matrix fit
786 into the core's local memory such an approach can be an interesting alternative for simulating SbS IPs.
787 However, these types of ASICs are optimized for their amount of cores but were not designed to access
788 external memory with high memory bandwidth. Thus, these multi-core CPU ASICs are a good alternative
789 as long as everything necessary can be stored within the core's memory. In the case of 32 bit floating
790 point numbers this allows to store only one or two thousand values per core. Assuming 128 neurons in
791 an IP, this would allow to connect every one of these neurons to only 15 input neurons. This back-on-the-
792 envelope calculation excluded all the required auxiliary memory and space for program code necessary
793 for an efficient computation which needs to be deducted from the available storage for the weights and

794 latent variables too. However, the next generation of CPU from AMD are expected to contain core counts
795 in the region of the Epiphany-IV but with much more on-chip memory.

796 Modern GPUs on the other hand are optimized for the use in memory and computational demanding
797 applications, like e.g. deep non-spiking networks. The nvidia TU 102 GPU (see nvidia turing GPU
798 architecture whitepaper WP-09183-001 v01 on <http://www.nvidia.com>) contains 72 (or 68 when
799 on a RTX 2080 Ti graphics card) streaming multi-processors (SM) with 96 KByte of L1 cache / shared
800 memory each. These 96 KByte can be configured as 64 KByte L1 cache & 32 KByte shared memory
801 or 32 KByte L1 cache & 64 KByte shared memory. This is similar to the specifications of the Epiphany
802 CPUs. However, the TU 102 has – compared to the Epiphany chips – a 384 bit wide memory bus with a
803 clock rate of up to 14 gigabit per second to external GDDR6 memory modules (672 gigabyte per second
804 or 168 giga-words per second for 32 bit floating point numbers). It needs to be noted that these 2.3
805 giga-words per second of 32 bit floating point numbers per SM can only be achieved if special care is
806 taken in correctly aligning the memory access (see CUDA Toolkit Documentation – Best Practices Guide
807 v10.1.168 <https://docs.nvidia.com/cuda/index.html>).

808 For the timing tests in the results section, one of the used systems contains Intel Xeon E5-2640v3 CPUs.
809 Such a CPU has 8 cores with a 2.6 GHz clock rate (in a multi-core task) and 59 gigabyte per second of
810 memory bandwidth. A nvidia TU 102 GPU has 72 SMs. For the following calculation it is assumed that
811 this corresponds roughly to 9 x 8 CPU cores. Following this assumption, this can be translated into 9 x
812 Intel Xeon E5-2640v3 with a combined memory bandwidth of 531 gigabyte per second. Even though the
813 memory on the GPU is a bit faster, the problems encountered with the multi-core CPUs can be assumed
814 to occur with GPUs too. Even more so if the complex requirements of GPUs for memory alignment can
815 not be fulfilled for all 72 simulations processes at the same time.

816 The proposed design for the SbS IPs is optimized with respect to the number of memory accesses. The
817 pipelined structure of the computations requires only a minimum of memory read and write transactions.
818 For example the update of N latent variables in one IP requires to read $4N$ values from memory and to
819 write N values back to memory. In the case of a normal CPU, basic mathematical operations are combined
820 and for every mathematical operation transactions with the memory are required. First, for N sets of two
821 numbers are multiplied which need to read from memory. Then the resulting N numbers need to be stored
822 into N auxiliary variables within memory. The auxiliary variables are read from memory and summed
823 up. The result of this summation is inverted and multiplied with ϵ . Again the N auxiliary variables are
824 read from memory, multiplied with the former result and written back into memory. Now the N auxiliary
825 variables as well as the N latent variables are read from memory, summed and written back into memory.
826 These resulting N values are read from memory, multiplied by a constant, and written back into memory
827 as the result of the update of the latent variables. This sums up to roughly $7N$ read operations and $4N$ write
828 operations. In summary $5N$ memory transaction are required for the pipeline design and $11N$ memory
829 operations for the general purpose CPU.

830 The question remains for which number of neurons per SbS IPs, the propose design shines. We expect
831 that neuron numbers of 1024 and smaller are the optimal use-case. Larger number of neurons are more
832 prone to be simulated by modern CPU with their large & densely integrated caches. Optimally, for every
833 SbS IP with 1024 neurons we aim for a weight matrix with 100 x 1024 values. However, during the
834 planned transfer of the presented design to an ASIC, we will evaluate if it will be beneficial to have a
835 mixture of sizes of SbS IPs on that chip. It is expected that this question will be dominated by the area
836 required for the supporting circuitry for the IP's memory. It is not expected that one ASIC will be able to
837 accommodate a whole SbS network. Thus we focus on networks of ASICs as substrate for SbS networks,

838 where the individual ASICs communicate with spikes (i.e. the index of the firing neurons and identifier
839 for their IP). If this approach will work out and how the details will look like in the end will be the target
840 of our future research.

841 Once available the proposed ASICs will not only serve to investigate neuronal networks based on the
842 SbS framework for AI applications where it can serve to develop both, feed forward deep convolutional
843 networks as well as generative models (as e.g. deterministic auto-encoders: (Ghosh et al., 2019)) but also
844 allow investigations of large scale models of the brain (as e.g. the visual system in cortex where many
845 retinotopic areas are mutually connected (Van Essen et al., 1992).

ACKNOWLEDGMENTS

846 This manuscript has been released as a Pre-Print at www.biorxiv.org (Rotermund and Pawelzik,
847 2018).

REFERENCES

- 848 Azkarate Saiz, A. (2015). Deep learning review and its applications
- 849 Brette, R. (2006). Exact simulation of integrate-and-fire models with synaptic conductances. *Neural*
850 *Computation* 18, 2004–2027
- 851 Brette, R. (2007). Exact simulation of integrate-and-fire models with exponential currents. *Neural*
852 *Computation* 19, 2604–2609
- 853 Bruckstein, A. M., Elad, M., and Zibulevsky, M. (2008). On the uniqueness of nonnegative sparse
854 solutions to underdetermined systems of equations. *IEEE Transactions on Information Theory* 54,
855 4813–4820
- 856 Burkitt, A. N. (2006a). A review of the integrate-and-fire neuron model: I. homogeneous synaptic input.
857 *Biological cybernetics* 95, 1–19
- 858 Burkitt, A. N. (2006b). A review of the integrate-and-fire neuron model: II. inhomogeneous synaptic input
859 and network properties. *Biological cybernetics* 95, 97–112
- 860 Candes, E. J., Romberg, J. K., and Tao, T. (2006). Stable signal recovery from incomplete and inaccurate
861 measurements. *Communications on Pure and Applied Mathematics: A Journal Issued by the Courant*
862 *Institute of Mathematical Sciences* 59, 1207–1223
- 863 Davies, M., Srinivasa, N., Lin, T.-H., Chinya, G., Cao, Y., Choday, S. H., et al. (2018). Loihi: a
864 neuromorphic manycore processor with on-chip learning. *IEEE Micro* 38, 82–99
- 865 Ernst, U., Rotermund, D., and Pawelzik, K. (2007). Efficient computation based on stochastic spikes.
866 *Neural computation* 19, 1313–1343
- 867 Furber, S. B., Galluppi, F., Temple, S., and Plana, L. A. (2014). The spinnaker project. *Proceedings of*
868 *the IEEE* 102, 652–665
- 869 Ganguli, S. and Sompolinsky, H. (2010). Statistical mechanics of compressed sensing. *Physical review*
870 *letters* 104, 188701
- 871 Ganguli, S. and Sompolinsky, H. (2012). Compressed sensing, sparsity, and dimensionality in neuronal
872 information processing and data analysis. *Annual review of neuroscience* 35, 485–508
- 873 Gatys, L. A., Ecker, A. S., and Bethge, M. (2016). Image style transfer using convolutional neural
874 networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2414–
875 2423
- 876 Ghosh, P., Sajjadi, M. S. M., Vergari, A., Black, M., and Schölkopf, B. (2019). From variational to
877 deterministic autoencoders

- 878 Guo, Y., Liu, Y., Oerlemans, A., Lao, S., Wu, S., and Lew, M. S. (2016). Deep learning for visual
879 understanding: A review. *Neurocomputing* 187, 27–48
- 880 Hinton, G. E. (2012). A practical guide to training restricted boltzmann machines. In *Neural networks:
881 Tricks of the trade* (Springer). 599–619
- 882 Izhikevich, E. M. (2004). Which model to use for cortical spiking neurons? *IEEE transactions on neural
883 networks* 15, 1063–1070
- 884 Jouppi, N., Young, C., Patil, N., and Patterson, D. (2018). Motivation for and evaluation of the first tensor
885 processing unit. *IEEE Micro* 38, 10–19
- 886 Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint
887 arXiv:1412.6980*
- 888 Lacey, G., Taylor, G. W., and Areibi, S. (2016). Deep learning on fpgas: Past, present, and future. *arXiv
889 preprint arXiv:1602.04283*
- 890 Lagorce, X., Stomatias, E., Galluppi, F., Plana, L. A., Liu, S.-C., Furber, S. B., et al. (2015). Breaking
891 the millisecond barrier on spinnaker: implementing asynchronous event-based plastic models with
892 microsecond resolution. *Frontiers in neuroscience* 9, 206
- 893 Lee, D. D. and Seung, H. S. (1999). Learning the parts of objects by non-negative matrix factorization.
894 *Nature* 401, 788
- 895 Lee, D. D. and Seung, H. S. (2001). Algorithms for non-negative matrix factorization. In *Advances in
896 neural information processing systems*. 556–562
- 897 Lustig, M., Donoho, D. L., Santos, J. M., and Pauly, J. M. (2008). Compressed sensing mri. *IEEE signal
898 processing magazine* 25, 72–82
- 899 Maass, W. and Bishop, C. M. (2001). *Pulsed neural networks* (MIT press)
- 900 Moore, S. W., Fox, P. J., Marsh, S. J., Markettos, A. T., and Mujumdar, A. (2012). Bluehive-a
901 field-programable custom computing machine for extreme-scale real-time neural network simulation.
902 In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*
903 (IEEE), 133–140
- 904 Naylor, M., Fox, P. J., Markettos, A. T., and Moore, S. W. (2013). Managing the fpga memory
905 wall: Custom computing or vector processing? In *2013 23rd International Conference on Field
906 programmable Logic and Applications* (IEEE), 1–6
- 907 Olofsson, A. (2016). Epiphany-v: A 1024 processor 64-bit risc system-on-chip. *arXiv preprint
908 arXiv:1610.01832*
- 909 Olofsson, A., Nordström, T., and Ul-Abdin, Z. (2014). Kickstarting high-performance energy-efficient
910 manycore architectures with epiphany. In *2014 48th Asilomar Conference on Signals, Systems and
911 Computers* (IEEE), 1719–1726
- 912 Olshausen, B. A. and Field, D. J. (2006). What is the other 85 percent of v1 doing. *L. van Hemmen, & T.
913 Sejnowski (Eds.)* 23, 182–211
- 914 Pfeiffer, M. and Pfeil, T. (2018). Deep learning with spiking neurons: Opportunities and challenges.
915 *Frontiers in Neuroscience* 12, 774. doi:10.3389/fnins.2018.00774
- 916 Rolinek, M. and Martius, G. (2018). L4: Practical loss-based stepsize adaptation for deep learning. *arXiv
917 preprint arXiv:1802.05074*
- 918 Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in
919 the brain. *Psychological review* 65, 386
- 920 Rotermund, D. and Pawelzik, K. R. (2018). Massively parallel fpga hardware for spike-by-spike networks.
921 *bioRxiv* doi:10.1101/500280

- 922 Rotermund, D. and Pawelzik, K. R. (2019a). Back-propagation learning in deep spike-by-spike networks.
923 *bioRxiv* doi:10.1101/569236
- 924 Rotermund, D. and Pawelzik, K. R. (2019b). Biologically plausible learning in a deep recurrent spiking
925 network. *bioRxiv* doi:10.1101/613471
- 926 Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating
927 errors. *nature* 323, 533
- 928 Salakhutdinov, R. (2015). Learning deep generative models. *Annual Review of Statistics and Its*
929 *Application* 2, 361–385
- 930 Saraf, N. and Bazargan, K. (2017). A memory optimized mersenne-twister random number generator.
931 In *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*. 639–642.
932 doi:10.1109/MWSCAS.2017.8053004
- 933 Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural networks* 61, 85–117
- 934 Serrano-Gotarredona, T., Linares-Barranco, B., Galluppi, F., Plana, L., and Furber, S. (2015). Convnets
935 experiments on spinnaker. In *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*
936 (IEEE), 2405–2408
- 937 Shirazi, N., Walters, A., and Athanas, P. (1995). Quantitative analysis of floating point arithmetic on fpga
938 based custom computing machines. In *fccm* (IEEE), 0155
- 939 Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., et al. (2016). Mastering
940 the game of go with deep neural networks and tree search. *nature* 529, 484
- 941 Spanne, A. and Jörntell, H. (2015). Questioning the role of sparse coding in the brain. *Trends in*
942 *neurosciences* 38, 417–427
- 943 Sze, V., Chen, Y.-H., Yang, T.-J., and Emer, J. S. (2017). Efficient processing of deep neural networks: A
944 tutorial and survey. *Proceedings of the IEEE* 105, 2295–2329
- 945 Tavanaei, A., Ghodrati, M., Kheradpisheh, S. R., Masquelier, T., and Maida, A. (2018). Deep learning in
946 spiking neural networks. *Neural Networks*
- 947 Thakur, C. S. T., Molin, J., Cauwenberghs, G., Indiveri, G., Kumar, K., Qiao, N., et al. (2018). Large-
948 scale neuromorphic spiking array processors: A quest to mimic the brain. *Frontiers in neuroscience* 12,
949 891
- 950 Tian, X. and Benkrid, K. (2009). Mersenne twister random number generation on fpga, cpu and gpu. In
951 *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on* (IEEE), 460–464
- 952 Van Essen, D. C., Anderson, C. H., and Felleman, D. J. (1992). Information processing in the primate
953 visual system: an integrated systems perspective. *Science* 255, 419–423
- 954 Wan, L., Zeiler, M., Zhang, S., Le Cun, Y., and Fergus, R. (2013). Regularization of neural networks
955 using dropconnect. In *International conference on machine learning*. 1058–1066
- 956 Wang, R. and van Schaik, A. (2018). Breaking liebigs law: an advanced multipurpose neuromorphic
957 engine. *Frontiers in neuroscience* 12
- 958 Wiedemann, T., Manss, C., and Shutin, D. (2018). Multi-agent exploration of spatial dynamical processes
959 under sparsity constraints. *Autonomous Agents and Multi-Agent Systems* 32, 134–162
- 960 Wunderlich, T., Kungl, A. F., Müller, E., Hartel, A., Stradmann, Y., Aamir, S. A., et al. (2019).
961 Demonstrating advantages of neuromorphic computation: A pilot study. *Frontiers in Neuroscience*
962 13, 260. doi:10.3389/fnins.2019.00260