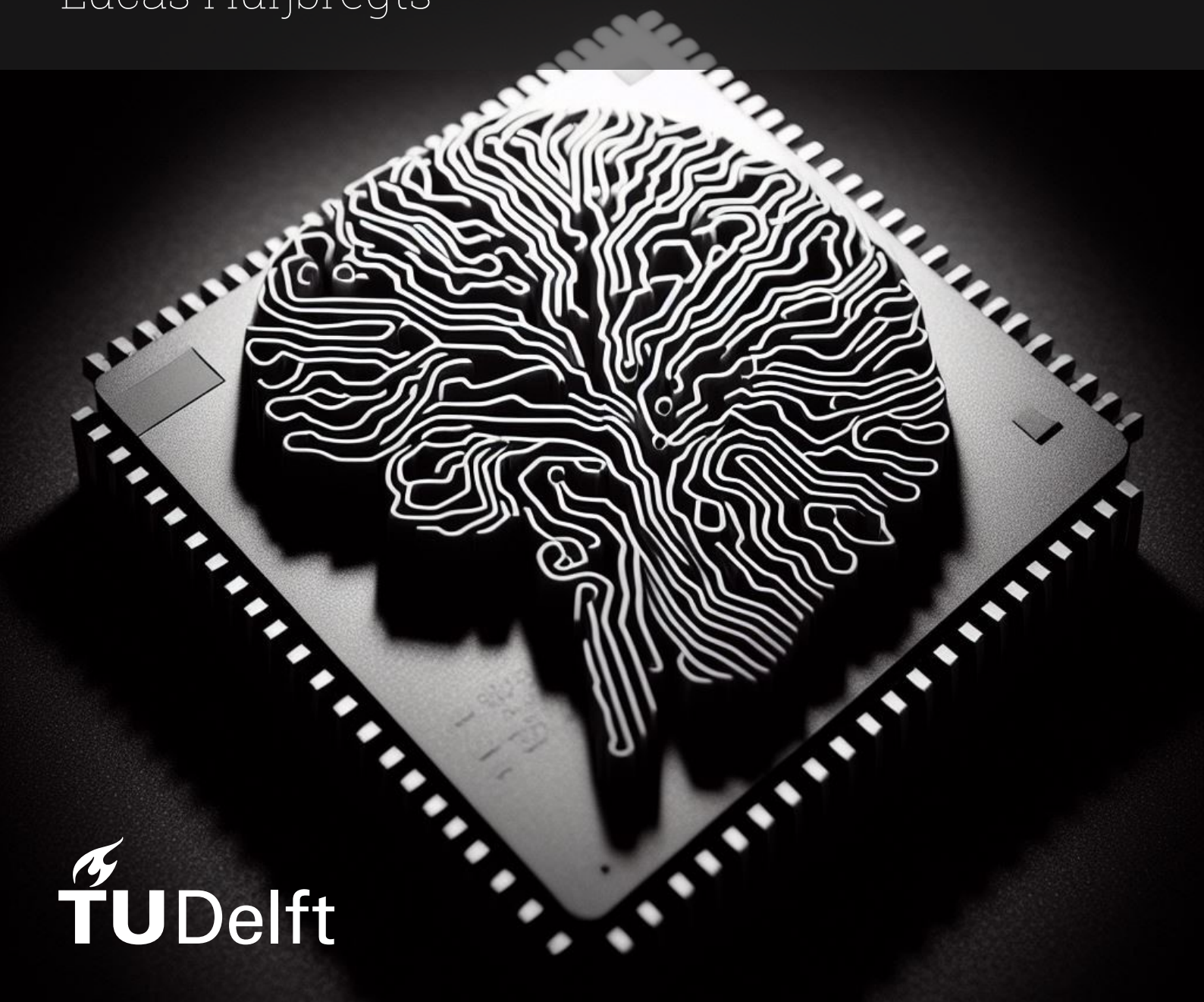
Transposable Multiport SRAM-based In-Memory Compute Engine for Binary Spiking Neural Networks in 3nm FinFET

Lucas Huijbregts



TU Delft

Transposable Multiport SRAM-based In-Memory Compute Engine for Binary Spiking Neural Networks in 3nm FinFET

by

Lucas Huijbregts

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday October 20, 2023 at 14:30.

Student number:	4717708
Project duration:	November 21, 2022 – October 20, 2023
Thesis committee:	Dr. ing. R. Bishnoi; *QCE, TU Delft, supervisor*
	Prof. dr. ir. S. Hamdioui; *QCE, TU Delft*
	Dr. ing. C. Gao; *Microelectronics, TU Delft*
	Dr. ing. A. Yousefzadeh; *IMEC, The Netherlands*
Faculty:	EEMCS
Degrees:	MSc Computer Engineering, MSc Embedded Systems

TU Delft

Abstract

Ultra-low power Edge AI hardware is in increasing demand due to the battery-limited energy budget of typical Edge devices such as smartphones, wearables, and IoT sensor systems. For this purpose, this Thesis introduces an ultra-low power event-driven SRAM-based Compute In-Memory (CIM) accelerator optimized for inference of Binary Spiking Neural Networks (B-SNNs). In this Thesis, a custom-designed 3nm SRAM cell is developed, with up to four read ports to improve inference performance and one transposable read/write port for efficient on-chip learning functionality. The event-based nature of SNNs is exploited to minimize the computation and memory cost. The design benefits from technology scaling of fully digital design by synthesizing the accelerator in the *imec* 3nm FinFET technology node. The proposed accelerator's performance is evaluated by running MNIST inference at 97.6% accuracy, achieving an impressive throughput of 44M inferences/s at 607 pJ/inference (3.2 fJ per synaptic operation) while running at 29 mW. The results demonstrate that the proposed accelerator provides an energy-efficient and high-performance solution for inference of Binary SNNs, opening up new possibilities for Edge AI applications.

Acknowledgement

First and foremost I would like to thank my TU Delft supervisor, Rajendra Bishnoi, and my supervisor at my internship company *imec*, Amirreza Yousefzadeh. Both supervisors were extremely helpful and hands-on, meaning I never sat with any questions for long. Additionally, they helped tremendously in perfecting the paper submission I did for the DATE Conference. I would also like to thank my chair, Said Hamdioui, for giving me advice, mainly on how to structure my story and properly argue for my design choices.

I would also like to thank Samantha (Hsiao-Hsuan) Liu from *imec* for spending a lot of time helping me with the SRAM cell design. Without her time, I would not have been able to share such detailed results for the memory. Similarly, I would like to thank Paul Detterer from *imec* for helping me design the Arbiter system and Neural Network architecture. Also important to mention are Sumit Diware and Yash Biyani for helping with more technical issues with the wide range of tools I had to learn to use.

Finally, of course I want to thank my friends and family for always being supportive, as well as distracting and entertaining when I needed them to be.

Lucas Huijbregts
Delft, October 2023

Contents

List of Figures

List of Tables

1 Introduction

This Chapter introduces the Thesis. First, the Motivation and Problem Statement for the project are discussed in Section 1.1. Next, in Section 1.2 the state-of-the-art solutions to to this problem are given, along with the challenges these solutions face. This is followed by the proposed solution of this Thesis to these challenges in Section 1.3, the main contributions of the Thesis in Section 1.4, and finally an outline of the Thesis in Section 1.5.

1.1. Motivation and Problem Statement

The demand for Artificial Intelligence applications to run on battery-powered Edge devices like smartphones, wearable devices, and various IoT systems is increasing rapidly. These devices are now dealing with a growing amount of data that needs to be processed using AI algorithms, such as facial and speech recognition, defect detection in factory supply chains, or traffic interpretation for autonomous vehicles. Communicating all this raw data to some central processing server, often called *the cloud*, is not an option.

Firstly, there is too much data to communicate, meaning the cost of transmitting it quickly becomes prohibitive. Additionally, most Edge devices are not guaranteed to always be connected to the Internet in the first place. Secondly, communicating the data to a central server and waiting for a response induces too much latency; systems that need to react in real time typically cannot wait for so long. Finally, communicating all this data brings privacy issues with it. Communicating visual or speech recordings of people to *the cloud* significantly increases the risk of sensitive information being leaked. Instead, by immediately processing all data on the Edge device itself, privacy risk is minimized [1].

Thus, there is a growing demand for execution of AI applications fully on the Edge devices. The main problem this demand brings is Energy consumption. Most Edge devices are wireless, meaning unlike the servers that make up *the cloud*, they do not have easy access to the Power grid. Instead they must rely on battery power. Therefore Edge AI is expected to run at much lower Power and Energy budgets than AI run on *the cloud*. This means existing solutions to accelerating AI applications, such as high-performance CPUs, GPUs, TPUs, or FPGAs cannot be applied here, and new computing paradigms need to be explored [2].

1.2. State-of-the-Art Solutions and their Challenges

The primary solutions explored for low-power Edge AI are neuromorphic computing and the use of Spiking Neural Networks (SNNs). Specifically, most solutions adopt some form of large-scale parallel operation, perform Computation In-Memory (CIM), exploit event-based computation, and reduce parameter precision. The main challenge for neuromorphic accelerators lies in how to implement the essential Multiply-and-Accumulate (MAC) Operation quickly and efficiently. Research is being performed into solutions to this challenge in both the analog and digital domain. For this Thesis, a digital solution is chosen due to its robustness, scalability, and portability across technology nodes [3].

In the digital domain, where synaptic weights are stored in SRAM, performing the MAC operation using CIM (CIM-MAC) requires additional hardware. Two main methods of CIM-MAC are typically utilized:

Adder Trees and their variants [2, 4, 5, 6] or Multiplication In-Memory with Sequential Accumulation in the SRAM Periphery [7, 8, 9, 10]. Adder trees allow for a higher degree of parallelism at the cost of breaking up the SRAM structure and adding a lot of hardware overhead. On the other hand, Sequential Accumulation in the Periphery minimizes hardware overhead and efficiently exploits SNN sparsity at the cost of lower parallelism in the pre-synaptic neuron dimension. This is because, for typical SRAM arrays, only one row may be accessed at a time, meaning only one pre-synaptic neuron can fire per timestep. Implementations such as [10] aim to mitigate this issue through approximate computing, but this degrades classification accuracy. Another issue for Sequential Accumulation is spike arbitration; ensuring only one spike enters the SRAM per timestep. Typically such arbitration systems are large and require multiple clock cycles per spike [7].

Additionally, on-chip learning is a popular practice for SNNs, allowing the SNN to adapt to changing environments and to be trained with smaller data sets. However, to efficiently perform on-chip learning, transposable access to the SRAM is essential. This means being able to access the SRAM cells both per row and per column, as opposed to just per row. Various methods have been explored to make SRAM transposable. However, most methods either require additional hardware components in the SRAM array [11], negatively influence cell stability [8, 12], result in slow, high-power Read/Write operations [5, 13], or add more transistors than necessary [14, 15].

1.3. Proposed Solution

In this Thesis, a Sequential Accumulation-based accelerator aimed at Binary-SNNs is presented. Utilizing Sequential Accumulation ensures minimal hardware overhead and full exploitation of the event-based nature of SNNs. The Binary network simplifies the MAC operation to just reading from memory and performing a popcount in the SRAM periphery. Pre-synaptic neuron parallelism is improved by utilizing newly designed 3nm SRAM cells with multiple decoupled read ports, inspired by [16, 17]. The decoupled ports additionally allow for Energy savings through local voltage scaling. Transposable read/write access for efficient online learning is provided through the original SRAM access ports. Additionally, a fully logic-based arbiter is employed to ensure multiport spike arbitration in just one clock cycle.

1.4. Contributions

The main contributions of this Thesis are:

- Design of a Binary-SNN hardware accelerator for ultra-low-power Edge AI applications using SRAM-based CIM.
- Design of four novel multiport SRAM bitcells in 3nm FinFET, adding one up to four decoupled read ports and a transposable read/write port to facilitate online learning.
- Design of a novel fully logic-based digital Arbiter for multiport SRAM read access.

1.5. Thesis Outline

The rest of this Thesis is Organized as follows:

- Chapter 2 explains the background information necessary for the rest of the Thesis. Topics covered are the Basics of Neural Networks, Binary Spiking Neural Networks, performing the MAC operation in a memory crossbar, the Basics of SRAM, and finally Online learning and Transposable SRAM.

- Chapter 3 presents the State of the Art in the field of performing the MAC operation in-SRAM, as well as how to add a Transposable port to SRAM.

- Chapter 4 gives an overview of the proposed system architecture, showing in detail the main building blocks: the SRAM Macro, the Arbiter, and the Neuron.

- Chapter 5 shows the simulation results of the proposed system. Results are shown at both circuit and system level, and an extensive comparison is made between using the unmodified SRAM

cell and the proposed multiport cells, both in terms of Online Learning efficiency and Inference performance. Additionally, the proposed design is compared to state-of-the-art systems in the field.

- Chapter 6 concludes the Thesis and lists recommendations for future work related to the Thesis.

2

Background

In this Chapter, the necessary background knowledge to understand this Thesis is discussed. It is assumed the reader has a basic understanding of circuit design, transistor mechanics, digital memories, and Neural Networks.

In order of occurrence, the following is explained. First, the basics of Neural Networks are given in Section 2.1, mainly as a refresher and to introduce the MAC operation. Next, Spiking Neural Networks (SNN) and the variant most relevant for this Thesis, Binary SNNs (B-SNN) are discussed in Section 2.2. In Section 2.3, it is shown how the MAC operation for a Binary SNN maps to a memory crossbar. In Section 2.4, the basics of SRAM, specifically in the 3nm FinFET technology node, are explained, which are necessary to understand the modifications to the SRAM cell that will be made later in this Thesis. Finally, in Section 2.5 the concept of Online Learning for SNNs is explained, how efficient online learning requires transposable SRAM, and what exactly transposable SRAM is.

2.1. Basics of Neural Networks

Neural Networks are a class of Machine Learning models inspired by the structure of animal brains. Neural Networks consist of neurons (nodes) and synapses (directed edges) connecting the neurons. Neurons transmit real numbers between each other via their synapse connections. Typically, these synapses have a weight associated with them, so that a transferred signal is multiplied by this weight before it arrives at the next neuron. The neuron performs two main functions; it accumulates the weighted input signals of its input synapses, and it applies an activation function to this accumulated value. The output of this function is the neuron's output signal, which it broadcasts on all its output synapses.

2.1.1. Fully Connected Neural Networks

The most basic and easy to understand Neural Network is the Fully Connected (FC) Neural Network, see Figure 2.1. An FC Neural Network groups neurons in layers, where signals are only transferred from one layer to the next, travelling from the Input Layer to the Output Layer via zero or more Hidden Layers in between. Every layer is fully connected to the previous and next layer, meaning all neurons of layer i are connected to all neurons of layer $i+1$ and layer $i-1$. In typical usage, an input is presented to the Neural Network by setting the outputs of the Input neurons. Signals propagate via the Hidden Layers to the Output Layer, where a decision is extracted from the neuron outputs. In classification tasks, each neuron is associated with a classification option, and the decision of the network is found by picking the neuron with the highest output.

As stated, each neuron accumulates the weighted signals it receives as its inputs. In Figure 2.1 neuron j in Hidden Layer 1 is highlighted. It receives the outputs of all the neurons in the previous layer: x_i where $i = \{1, 2, ..., N\}$, as indicated by the orange highlighted synapses. Before arriving at neuron j, these signals x_i are multiplied by the weights of their respective connections; w_{ij}. In FC networks the neuron then typically applies some bias b_j to this accumulation, after which the activation function f is applied. This can be summarized by Equation 2.1, giving the output x_j of neuron j:

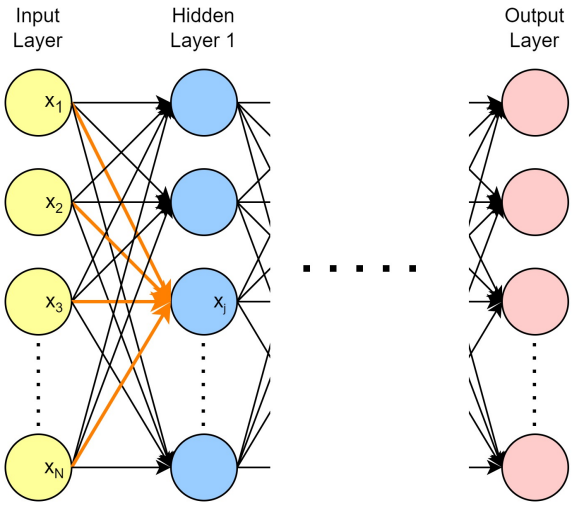


Figure 2.1: General structure of a Fully Connected Network

$$x_j = f\left(\sum_i (x_i \cdot w_{ij}) + b_j\right) \tag{2.1}$$

Here, x_i is the output from neuron i in the previous layer, w_{ij} is the weight of the connection between neuron i and neuron j, b_j is the bias of neuron j and f is its activation function. Typically the activation function is a simple nonlinear function such as a ReLU [18].

The most resource intensive operation in the FC network is the operation denoted by $\sum_i (x_i \cdot w_{ij})$, which consists of pointwise multiplication of all the input signals by their respective synapse weights and accumulating the results. This operation is called the Multiply-and-Accumulate (MAC) operation. It is the operation that requires the most arithmetic as well as the most accesses to memory, as all the weights w_{ij} need to be retrieved from it. Neural Network acceleration is therefore mainly concerned with making this MAC operation as efficient as possible [19].

2.1.2. Recurrent Neural Networks

It is worth noting that more complex Neural Network types exist. Most are outside scope of this Thesis, but Recurrent Neural Networks (RNN) [20] bare some explanation due to their relationship with Spiking Neural Networks, which are explained in Section 2.2. RNNs are a type of Neural Network used for input signals with a timing component. In RNNs, neurons do not only send signals to neurons in subsequent layers, like in a Fully Connected network, but also to neurons in 'previous' layers, as well as to themselves. Connections to themselves allow them to 'remember' their previous state, and use this information in subsequent timesteps. As such, the network can remember information over time and use this to extract temporal features of the input signal.

2.1.3. Training Neural Networks

In principle a Neural Network is nothing more than a multi-input, multi-output function with many internal parameters. What makes Neural Networks useful is that these internal parameters can be changed through training, allowing the network to 'learn'. By presenting the network with many training samples, observing the output, and making adjustments according to how well the network is performing, we can steer the network to perform better according to our metrics.

The most common method of training networks such as the FC network is through Backpropagation [21]. An in-depth explanation of this method is outside the scope of this Thesis, but a short explanation is useful to understand issues with training Spiking Neural Networks later on. Backpropagation works by first passing some training data through the network and observing the resulting output. An error value (or Loss) is calculated based on the difference between the observed and desired output. Then, starting at the Output Layer, for every trainable parameter in the layer, the gradient (or slope) of the Loss function is found with respect to the trainable parameter. Depending on the direction and magnitude

of the gradient, the parameter is adjusted in order to minimize the Loss function. This step is called Gradient Descent. Once this step is performed for a layer, the same can be done for the previous layer. The process is repeated until the Input Layer is reached, hence the name Backpropagation. This full Backpropagation operation is repeated, typically with a lot of different input data, until we are satisfied with the network performance.

2.2. (Binary) Spiking Neural Networks

2.2.1. Spiking Neural Networks in General

Spiking Neural Networks (SNN) are a special type of Neural Network, deemed by many to be the next generation of Neural Networks [22]. Their main innovation is the use of Binary spikes as the signals travelling between neurons. Where in conventional Neural Networks the signals are typically represented by floating point or multi-bit integer numbers, in SNNs the only signal transmitted is a spike with an amplitude of '1'. These spikes are no longer static signals, but are time-dependent. Where precision is lost in the amplitude of the signal, it is won by adding a temporal component to the signal. The main advantage of SNNs is the lower precision of the signals, and the fact that the network can work on an event-bases. Using low-precision signals mean most operations are simpler and thus cheaper to perform. Event-based operation means hardware only needs to operate when a spike arrives at its input; otherwise it can remain dormant. If hardware is designed to exploit this fact, this allows for significant Power savings.

As with conventional Neural Networks, the spikes travel over the synapses and are multiplied by the synapse weights before being accumulated in the neurons. However, as the spikes have a temporal component, this accumulation now happens over time. The amount accumulated by a neuron is represented by its membrane potential $V_{mem}(t)$. This potential needs to be remembered by the neuron between subsequent timesteps to continue the accumulation. As such, the neuron is self-recurrent, similar to in an RNN network.

The main distinguishing factor between different types of SNNs is which type of spiking neuron they utilize. The simplest of such neurons is the Integrate-and-Fire (IF) neuron. It accumulates the incoming weighted spikes as $V_{mem}(t)$. It has a threshold potential V_{th} (comparable to the bias in conventional Neural Networks), and if $V_{mem}(t)$ exceeds V_{th} after a timestep, it fires. This entails that $V_{mem}(t)$ is reset to '0', and that the neuron outputs a spike of its own to all its output synapses. This behavior is visualized in Figure 2.2 and can be summarized with the following equations, which are performed in the neuron in every timestep:

$$V_{mem,j}(t) = V_{mem,j}(t) + \sum_j w_{ij} \cdot x_i(t) \tag{2.2}$$

$$x_j(t) = \begin{cases} 1, & \text{if } V_{mem,j}(t) \geq V_{th} \\ 0, & \text{otherwise} \end{cases} \tag{2.3}$$

$$V_{mem,j}(t) = (x_j(t) - 1)V_{mem,i}(t) \tag{2.4}$$

Another neuron that is commonly implemented is the Leaky Integrate-and-Fire (LIF) neuron. It adds a leak factor to the model that leaks away part of $V_{mem}(t)$ in every timestep. This allows the neuron to 'forget' older information, which is especially useful in time-based tasks where accumulated spikes from a long time ago are no longer relevant. The mathematical description of the LIF neuron is identical to the IF neuron, except Equation 2.2 is replaced by:

$$V_{mem,j}(t) = \alpha V_{mem,j}(t) - \beta + \sum_j w_{ij} \cdot x_i(t) \tag{2.5}$$

Where α represents a proportional leak factor, while β represent a linear form of leakage.

More complex neuron models exist, such as (in order of increasing complexity) the QIF neuron, the LIF with exponential decay, the Izhikevich, and the Hodgkin-Huxley models. However, these are all much less suitable for digital hardware implementation due to their complexity and use of differential equations [23]. Therefore, most research for hardware implementations is focused on IF and LIF neurons.

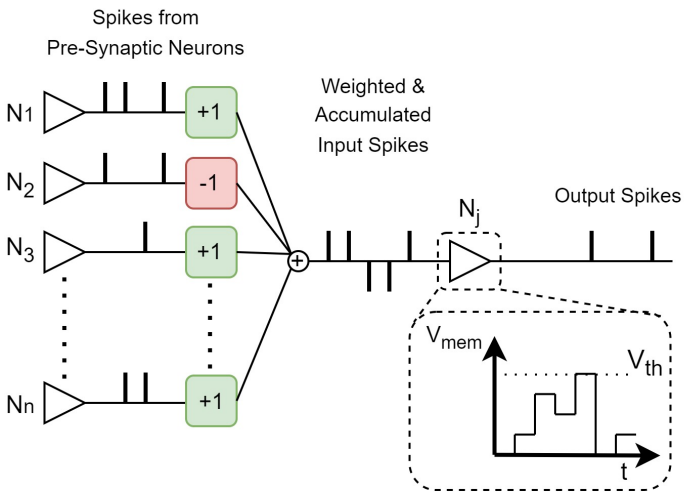


Figure 2.2: Diagram of a layer of IF neurons propagating spikes to another IF neuron, N_j. Spikes are multiplied by synapse weights and accumulated by the neuron as its membrane potential V_{mem}. Neuron N_j fires a spike itself when $V_{mem} \geq V_{th}$, after which it resets V_{mem} to 0.

2.2.2. Training SNNs

While conventional Neural Networks can be trained using Backpropagation, this method does not work for SNNs [24]. This is due to the training step where the gradient of the Loss function needs to be calculated. Since the activation function of an IF/LIF neuron, as described by Equation 2.3, is a step function, its gradient, or derivative, is the Dirac delta function. This function is '0' for all inputs except one, for which it approaches ∞. Therefore, training the network through gradual adjustments of the trainable parameters based on the Loss function and its gradient is not possible. Three main alternative methods are currently used to train SNNs: bio-inspired learning, Surrogate Gradient Descent, and ANN to SNN conversion. However, none of these methods is objectively better than the others, and all are still being researched and improved.

Bio-Inspired Learning: Due to SNNs' similarity to biological neural networks, a lot of research is being done into biologically inspired learning techniques. The most common bio-inspired learning model is Spike-Timing Dependent Plasticity (STDP) [25, 26]. In this model, synapse connections between sets of neurons are adjusted based on the relative timing of spike events of the two neurons. STDP is an unsupervised learning method, meaning that apart from the inputs themselves, no information is provided to the network. This makes training the network a simpler process; we simply need to feed in a lot of input data and the network will learn by itself. However, this also means we have little control over the network's learning process, meaning it may not learn exactly what or how we want it to. Because of this, bio-inspired algorithms tend not to perform as well in terms of accuracy as conventional Neural Networks. To overcome this, some researchers are looking into supervised STDP, where the STDP process is nudged in the right direction during training [27, 28].

Due to its unsupervised nature, bio-inspired learning has the added benefit of enabling Online Learning. This is a process where the network is allowed to keep learning after it has been pre-trained. As such, it can adapt to a changing environment, and the original training step can be performed with a smaller dataset; the inputs observed after deployment expand the pre-training dataset.

Surrogate Gradient Descent: As explained, Gradient Descent in its original form cannot be used to train SNNs. However, a workaround method is to use a surrogate activation function [24]. Instead of using the Step function, which has a derivative that is not useful, a function similar to the Step function is used, which does have a useful derivative. Figure 2.3 shows an example of such a surrogate gradient function $F(x)$ for the Step function $H(x)$. $F(x)$ approximates $H(x)$ quite well, but it is smooth and results in a smooth derivative as well.

In terms of accuracy achieved, Surrogate Gradient Descent tends to outperform bio-inspired learning, but it still does not tend to achieve the same numbers as conventional Neural Networks [29].

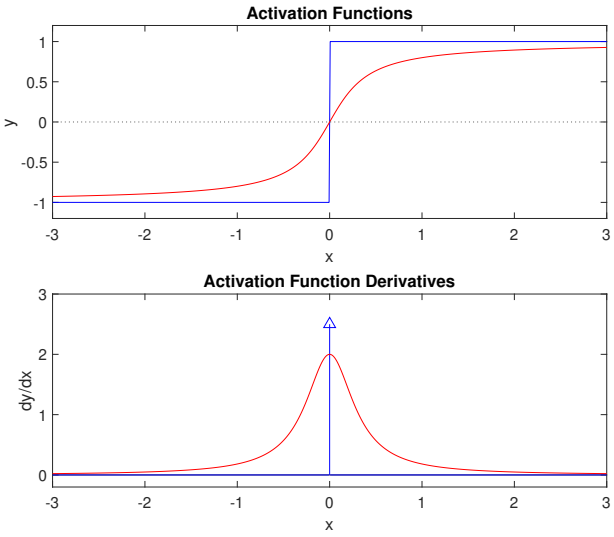


Figure 2.3: Step Function $H(x)$ and its derivative $H'(x)$, along with a Surrogate Gradient function $F(x)$ and its derivative $F'(x)$.

ANN to SNN Conversion: The method that achieves the highest accuracy is ANN to SNN conversion [30]. Here, a conventional Neural Network is trained as it normally would, after which it is converted to an SNN. Typically, the ANN is restricted to the use of a ReLU for the activation function, and to integer values for the signals on the synapses. To convert to an SNN, the activation ReLU activation is replaced by the IF neuron functionality, and the signals are replaced by rate-coded spikes. Through this method, the full accuracy of the original ANN is maintained. The main disadvantage of this method is that it generates a very large number of spikes. As such, the event-based nature of SNNs is not fully exploited [24].

2.2.3. Binary SNNs

In this Thesis, the focus will be on Binary Spiking Neural Networks (B-SNN). These are SNNs with Binary weights; for this Thesis the weights are restricted to {+1,-1}. The diagram in Figure 2.2 shows such a B-SNN.

There are two main advantages to B-SNNs. Firstly, weight storage is minimal; every weight can be represented by a single bit, and can thus be stored in a single bitcell. Secondly, the MAC operation is simplified as much as possible. By encoding {+1,-1} as bits {'1', '0'}, multiplication simplifies to an AND operation, and accumulation is achieved through a popcount. As will be shown in Section 2.3, this is very convenient for hardware implementation.

As B-SNN weights and activations are all Binary, for the ANN to SNN training technique a BNN should be used as the base ANN to convert. BNNs also cannot be trained simply with Backpropagation due to the binarization step in their activation function. Thus, training a B-SNN leaves just two methods; bio-inspired learning [31], or BNN to B-SNN conversion [32] where the BNN is trained with Surrogate Gradient Descent. In order to fairly compare to the State-of-the-Art in the field of SNN accelerators, BNN to B-SNN conversion is used in this Thesis.

BNN to B-SNN conversion: BNNs use {+1,-1} for both weights and activations, so conversion to a B-SNN means the activations should be converted to {1,0}. Doing so on its own will result in a bias in the accumulated value. Thus, the trained bias of the BNN must be transformed. Additionally, to conform to SNN standard practice, the bias should be converted to a threshold instead. Converting to a threshold is simply a question of multiplying the bias by -1; then, instead of subtracting the bias and comparing the result to 0 (check if $V_{mem} - bias \geq 0$), the accumulated value is directly compared to V_{th} (check if $V_{mem} \geq V_{th}$). The following formula describes how the BNN bias can be converted to a transformed threshold voltage for neuron j in one step [32]:

$$V_{th,j} = \lceil 0.5 \cdot -bias \rceil + 0.5 \cdot \sum_i w_i \tag{2.6}$$

where w_i are the weights of all the input synapses of neuron j.

2.3. MAC Operation in Memory Crossbar

For Fully Connected Neural Networks, storing the synapse weights and performing the MAC operation both map quite well to a standard memory crossbar. Figure 2.4(a) visualizes the MAC operation for neuron 2 in Layer $i+1$. It receives inputs from the four neurons in Layer i, which are multiplied by their respective synapse weights and accumulated in the neuron. Figure 2.4(b) shows the same network, but now the weights are placed in a crossbar structure. The weights of synapses at the input of neuron 2 in Layer $i+1$ correspond to the crossbar cells in the column above neuron 2.

This mapping makes the multiplication step in the MAC operation very convenient; input signals are entered into the memory from the left using the existing Wordlines, and they can be multiplied by the weights in the memory at the cells themselves. The main issues that remain for performing a MAC operation in-memory are how to multiply the input activations and the weights in-memory, and how to accumulate the results.

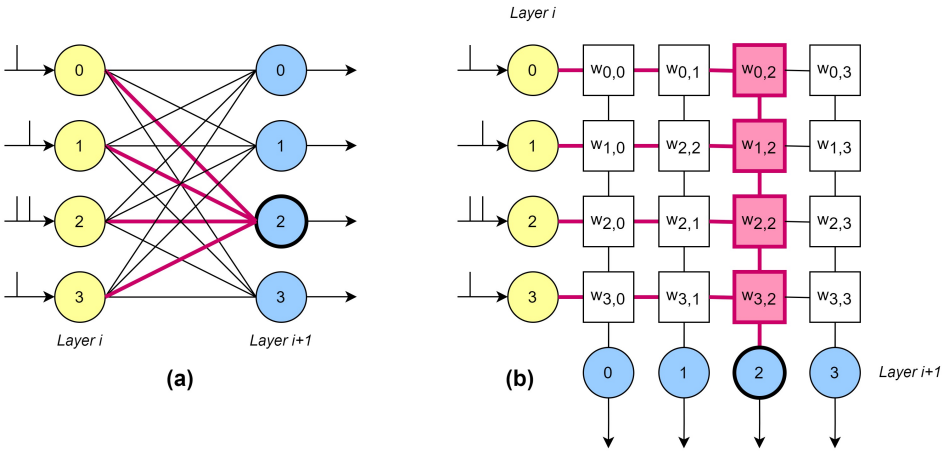


Figure 2.4: (a) MAC operation visualized for singular post-synaptic neuron; (b) MAC operation for same post-synaptic neuron, mapped to crossbar array.

In this Thesis, the first problem is solved by utilizing Binary spikes and Binary weights. By encoding the {+1,-1} weights as {1,0} in memory, the multiplication is performed implicitly when reading from memory; if there is no spike (input activation '0'), no Read operation happens, so '0' is accumulated in the post-synaptic neuron. If there is a spike (input activation '1'), a Read operation occurs, where either '1' is read ('+1' should be accumulated) or '0' is read ('-1' should be accumulated). The three cases are summarized in Table 2.1.

Table 2.1: Case Table showing implicit Binary multiplication in Read operation.

Input Activation	Memory Read	ΔV_{mem}
0	*None*	0
1	1	$+1$
	0	-1

2.4. Basics of SRAM

2.4.1. 6T SRAM Cell

The most popular form of on-chip memory is SRAM; Static Random Access Memory. The most common method of implementing an SRAM cell is depicted in Figure 2.5(a), where a two cross-coupled inverters

(M3-M6) form a latch, and two access transistors (M1-M2) provide Read/Write access to this latch. In this Thesis, this cell will be referred to as the 6T SRAM Cell. To form an SRAM array, the cell is simply duplicated and the Wordlines (WL) and Bitlines (BL/BLB) are connected. Using Figure 2.5(a) as a reference, the result is that a row of SRAM cells shares a single WL, and a column of cells shares BL/BLB.

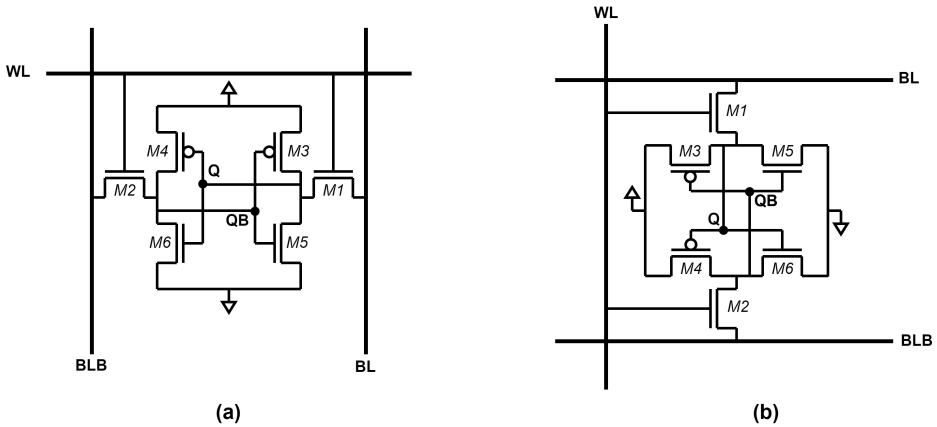


Figure 2.5: Standard 6T SRAM Cell; (a) Typical Depiction; (b) Rotated 90 degrees.

To write to the cell, the WL is driven high, and the BL is driven to the value to be written to the cell, while the Complementary Bitline (BLB) is driven to the inverse of BL. This will override the latch and set node Q to the value on BL. To read from the cell, BL and BLB are precharged to the supply voltage VDD, after which WL is driven high. Depending on the latch content, either BL or BLB is discharged to ground via M1 and M5 or M2 and M6 respectively. As a result, either {BL, BLB} = {1,0} or {BL, BLB} = {0,1}, which is sensed at the periphery of the array. In principle these procedures are simple, but designing SRAM is a delicate balancing act. The main challenge is balancing cell stability (maintaining the stored value) and writability (being able to override the stored latch value) [33]. Because of this challenge, write assistance techniques are sometimes utilized. The SRAM described in this Thesis uses the Negative Bitline Voltage technique, explained in Section 2.4.4. Additionally, decoupled Read ports may be implemented, as explained in Section 2.4.3.

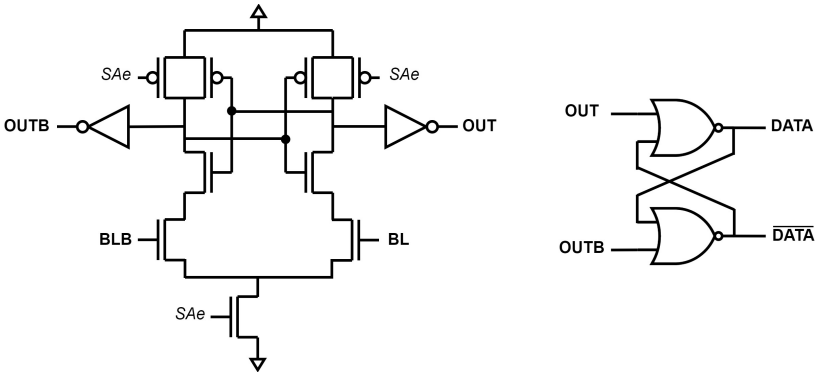


Figure 2.6: Differential Sense Amplifier Schematic [34]

When reading, the state of BL and BLB is sensed using a differential Sense Amplifier (SA). The most common SA is shown in Figure 2.6 [34]. It consists of a latch similar to the SRAM cell, but adds three Transistors and an SR latch, which in turn adds eight Transistors. This leads to an area quite a bit larger than the SRAM cell. As the SA does not fit within the width of the SRAM cell, it is common to use MUXs to sense BL/BLB. For instance, a 128-wide SRAM may be split into 4 sets of 32 columns using 4-to-1 MUXs, meaning only 32 SAs are needed. Despite its size, the differential SA is popular, as it saves in Power and Speed; due to the use of differential sensing it is able to determine the state BL/BLB before either Bitline has discharged to VDD/2, which is when typical logic would sense the

state. This saves in time, but also in Power, as the SA will stop the current flow as soon as the sensing has occurred [34].

The depiction of in Figure 2.5(a) is most commonly used for the 6T SRAM cell. However, in this Thesis the rotated variant shown in Figure 2.5(b) will also be shown. This schematic is the same on all accounts, only rotated by 90 degrees such that WL runs vertically and BL/BLB run horizontally.

2.4.2. 6T SRAM Cell Layout in 3nm FinFET

The layout of an SRAM cell depends heavily on the technology in which it is implemented, which for this Thesis is the 3nm FinFET technology from *imec* (Interuniversity Microelectronics Center). In this Subsection a brief explanation of 3nm FinFET layout is given, followed by an explanation of the layout of the 6T SRAM cell in this technology node. The most important SRAM layout design rules for the node can be found in Appendix A.1.

FinFET (Fin Field Effect Transistor) technology is based not on planar layout design, but instead on non-planar, 3-dimensional transistors. By wrapping the Gate around the Channel of the transistor, the surface area between the Gate and Channel is increased significantly, providing enhanced control over the Channel. This allows for less wide transistors and thus a higher integration density without deteriorating the transistor's qualities [35]. Figure 2.7 illustrates the difference between Planar FET and FinFET.

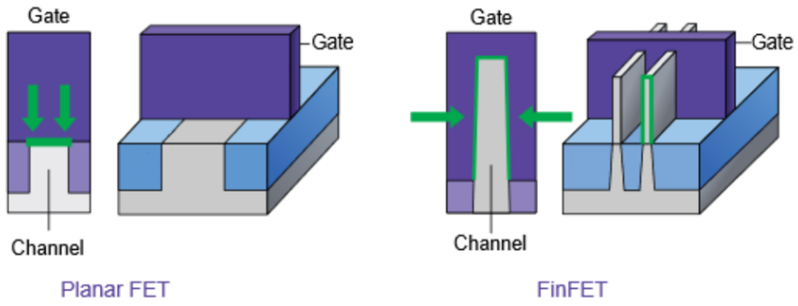


Figure 2.7: Planar FET and FinFET comparison [35].

Figure 2.8 shows schematically what the imec 3nm technology layout looks like (note: not to scale). Figure 2.8(a) shows the most important layers and how they are vertically connected. The GATE and FIN layers are approximately at the same height. Directly connected to the FIN layer is the M0; the first metal layer. MINT, M1, M2, and M3 are the subsequent metal layers, able to be connected to each other through vias VINTG, VINTA, V0, V1, and V2. Figure 2.8(b) shows from a Bird's Eye view how the aforementioned layers lie on top of each other. Especially important to note is how all layers consist of strips running in either the vertical or horizontal direction. The layout is changed by making cuts in these strips at different places. M0, GATE, M1, and M3 run in the vertical direction, while FIN, MINT, and M2 run in the horizontal direction. Finally, Figure 2.8(c) shows how a Transistor is formed; the FIN forms the Channel, and a GATE strip is deposited over the FIN to form the Gate. M0 strips are used as contact points to form the Drain and Source. By placing a different oxide below the Transistor, the designer can control whether it behaves as a PMOS or NMOS transistor.

Figure 2.9a shows the layout schematic of the 6T SRAM Cell (note: not to scale). A to-scale image of the layout in Virtuoso Layout Editor can be found in Appendix A.2. The metal layers MINT and M1 are shown to the sides; in reality they are deposited on top of the cell. Note that some metal strips of the MINT layer are shorter; this indicates that they do not stretch over the full cell width, and are only used for routing inside the cell itself. The longer strips do stretch across the full cell, and connect to neighboring cells. For example, the Q strip is only used for routing Q (the cell content) inside the cell, while the BL strip spans the entire cell and connects to both horizontally neighboring cells.

Figure 2.9b shows how multiple of such 6T cells are connected together. By mirroring the standard cell, neighboring cells can overlap slightly, maximizing integration density. For this to be possible, it is very important that the nodes on the M0 strips of neighboring cells are allowed to be shared. For instance, horizontally neighboring cells share BL and BLB, as these lines run over the entire row of cells. Thus, the M0 nodes BL and BLB can be shared between horizontally neighboring cells, allowing

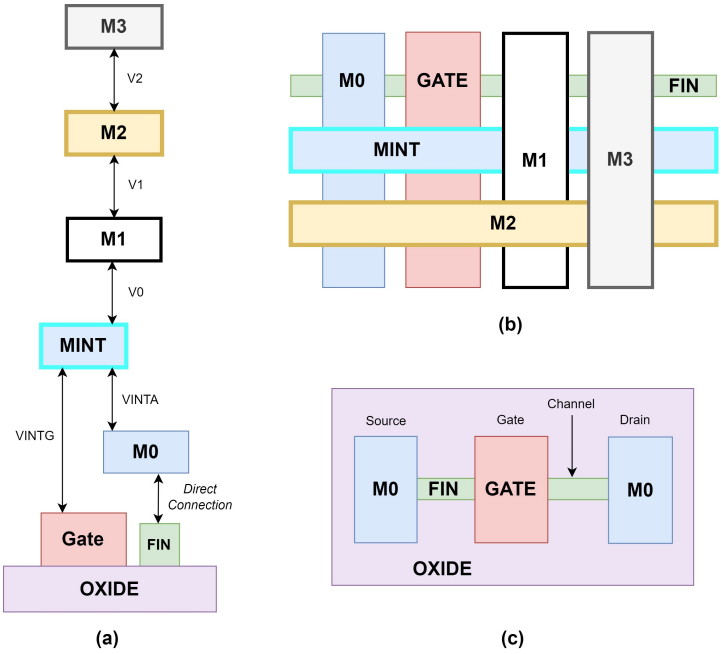


Figure 2.8: Schematic view of imec 3nm technology; (a) Vertical View; (b) Bird's Eye View; (c) Transistor Structure (note: not to scale).

the slight overlap.

2.4.3. 8T SRAM Cell

As explained earlier in this Section, the main challenge in SRAM cell design is balancing the data hold strength and writability of the cell. The data hold strength needs to be sufficient so that, when performing a Read operation, the cell content is not changed. On the other hand, if the data hold is too strong, it is not possible to perform a successful Write operation, as the latch content cannot be overridden.

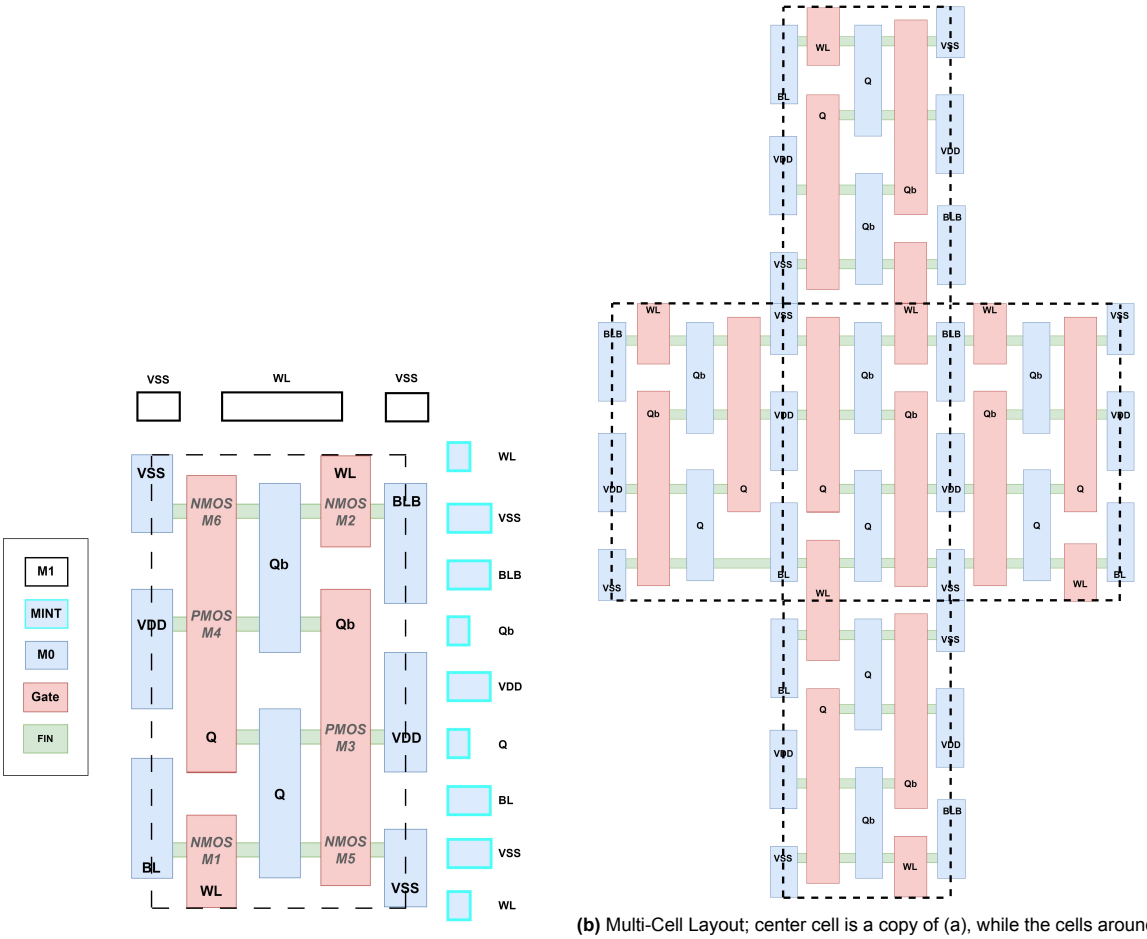
One solution to this problem comes in the form of the 8T SRAM cell [16]. Two Transistors are added to the cell; M7 connects with its Gate to the inverted cell content (QB), while M8 connects with its Gate to an additional Read Worldine (RWL). To perform a Read operation, the additional Read Bitline (RBL) is precharged. Then, RWL is driven high so that M8 conducts. RBL is then discharged if M7 also conducts, meaning if QB = '1' (Q = '0'). The cell content can then be sensed from RBL. M7 and M8 together form a Decoupled Read Port. It is called Decoupled because it only connects via a Gate to the cell content, resulting in minimal influence to the cell stability. By only Reading via this Decoupled Port, and using BL/BLB only for Writing, the two processes are made much more independent. Thus, the cell can be optimized for writability without risking low stability under Reading.

As a result, the cell stability is improved drastically at the expense of just 16% area overhead in the 65nm technology node [16].

2.4.4. Negative Bitline Voltage Assistance Technique

Another technique utilized to balance the writability and cell stability is the Negative Bitline Voltage (NBL) Assistance technique. It is especially useful at small technology nodes such as the 3nm FinFET node used in this Thesis, as at these sizes the Bitline Resistance is the main problem for the design [36]. Due to the high resistance, when writing to the cell and a Bitline is driven low, the Bitline does not fully discharge to Ground at the cell itself. Instead it remains at some voltage higher than VSS. As a result, the voltage differential between BL and BLB is not sufficient to override the latch content.

Figure 2.11(a) shows what happens for such technology nodes without NBL assistance. The WL is driven high, and BL is driven low (and BLB is driven high) to write '0' to the cell. However, the voltage on BL never reaches below $120mV$. Together with the other parasitics causing BLB to drop down to below $550mV$, the voltage differential is insufficient to write to the cell.



(a) Single Cell Layout. MINT and M1 shown to the side.

(b) Multi-Cell Layout; center cell is a copy of (a), while the cells around are mirrored to fit together.

Figure 2.9: 6T SRAM Cell Layout.

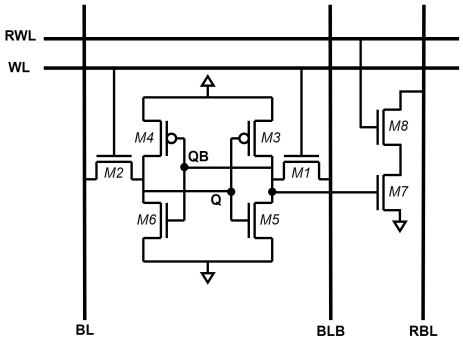


Figure 2.10: 8T SRAM Cell Schematic (from [16]).

Figure 2.11(b) shows the same operation, with the only difference being that BL is driven to $V_{WD} = -300mV$ (BLB is still simply driven to $VDD = 700mV$). Due to the line parasitics, BL never actually reaches $-300mV$, but rather to $-40mV$. However, the induced voltage differential is now sufficient to override the latch content and write to the cell. This is visible as Q and QB flipping.

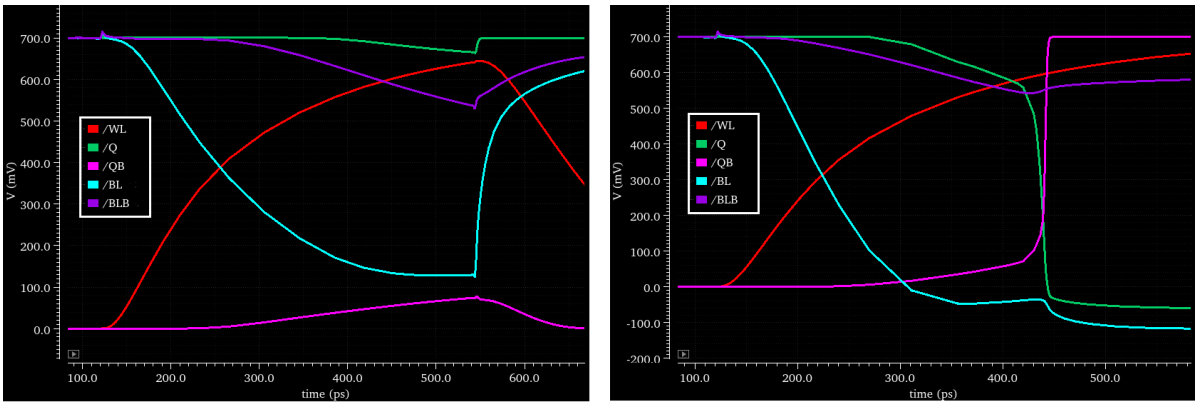


Figure 2.11: Comparison between: (a) Conventional Write operation; (b) NBL-assisted Write operation.

Typically, more severe parasitics in the SRAM array necessitate a lower V_{WD} to successfully write to the cell. It has been shown that for arrays for which $V_{WD} < -400mV$ is necessary to successfully write to the cell, expected yield diminishes quickly [36]. Hence, in this Thesis a limit of $V_{WD} = -400mV$ will be adhered to; an array needing a lower V_{WD} is considered invalid.

2.5. Online Learning and Transposable SRAM

On-chip learning is a popular practice in SNNs, where the network keeps learning even after deployment. This allows it to adapt to changing environments and be trained with smaller and more manageable data sets. For efficient on-chip learning, it is crucial to have access to the synapse weights in both the pre-synaptic and post-synaptic dimensions [31, 37, 38].

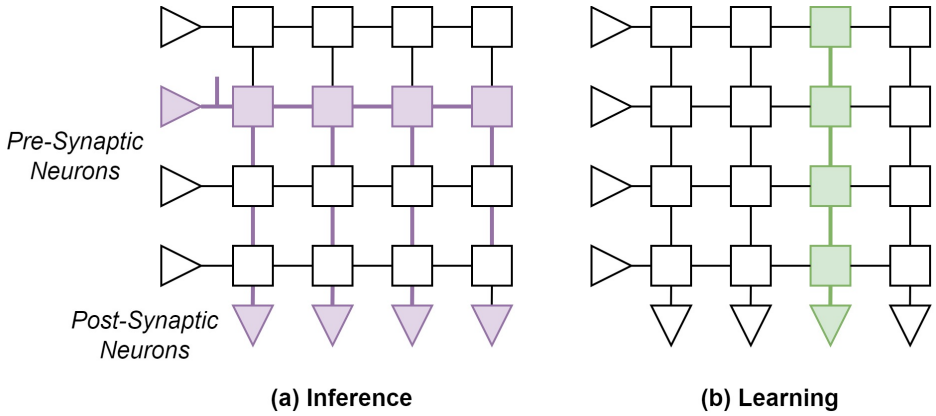


Figure 2.12: Illustration of row-wise and column-wise access to synaptic weight array for (a) Inference and (b) Learning respectively.

Access in the pre-synaptic dimension is necessary so that a spike can be sent from a pre-synaptic neuron to the SRAM crossbar and be multiplied by the weights between this neuron and all the connected post-synaptic neurons. The result should arrive at all the post-synaptic neurons. This process is illustrated in Figure 2.12(a) and corresponds to reading a memory Row. In contrast, learning in SNNs typically occurs when particular conditions arise in the post-synaptic neuron. Therefore, weight updates should occur to all the synapses before this post-synaptic neuron, corresponding to a memory Column, as shown Figure 2.12(b).

Standard SRAM allows Read/Write operations in either just the row or just the column direction, depending on the chosen orientation of the cell (see Figure 2.5). If we choose SRAM with only row-wise access, then reading and writing the weights in the column-wise direction would require dozens of row-wise operations, costing a lot of time and Power. In contrast, transposable SRAM provides access in the row- and column-wise directions [8, 39]. Specifically, for SNN inference and efficient on-chip learning, row-wise Read access (spikes) and column-wise Read/Write access (weight updates) are required.

3

Related Work

In this Chapter, the relevant Related Work for this Thesis is discussed. First, the three main methods of performing the MAC operation digitally in SRAM are shown in Section 3.1. Next, Section 3.2 discusses the two main methods of communicating spikes between layers in neuromorphic hardware. Then, in Section 3.3 the six most relevant methods of making SRAM transposable are discussed.

3.1. MAC Operation in-SRAM

The most critical operation to accelerate in order to increase the efficiency of Neural Network inference is the MAC operation, due to its prevalence but also due to its potential for parallelization. Computation In-Memory (CIM) has emerged as the primary method of accelerating the MAC operation [9]. Typically the main Power and Time draws when performing Neural Network computations are Reading and Writing from and to the memory, even for neuromorphic accelerators [40]. Synapse weights, once a network has been trained, remain the same during inference. Thus, instead of constantly retrieving the same synapse weights from the memory and performing calculations in a dedicated compute unit, it is much more efficient to enter any generated spikes into the memory and perform the computations there.

As explained in Section 2.3, for the Binary Spiking Neural Network that is being accelerated in this Thesis, the multiplication step in the MAC operation is performed implicitly when reading from the memory. Thus, the step that still needs accelerating is the accumulation; the summing of all the pointwise multiplication results. This Section will detail the three main methods found in literature to perform this operation.

3.1.1. Systolic Arrays

A currently popular computing architecture paradigm when dealing with array-like structures requiring high throughput is the use of systolic arrays [41]. A CIM in-SRAM implementation using systolic arrays can be found in [42]; an overview of this implementation is shown in Figure 3.1. Figure 3.1(a) shows the array of MAC elements, where inputs (spikes) are fed from the bottom Input Buffer, and where partial sums flow in the horizontal direction into the Accumulator. Note that this is a 90-degree rotated view with respect to the other images shown in this Thesis; typically spikes are entered from the left and accumulated at the bottom of the array.

Figure 3.1(b) shows the structure of a single MAC element. Each element stores a weight in an SRAM cell and multiplies the weight by the activation using an XNOR block. Note that this is already superfluous hardware; if a BNN is converted to a B-SNN, this XNOR block is no longer necessary [32]. Next, a Full Adder is used to add the multiplication result from the current block to the incoming partial Sum, and generate a partial Sum Output and Carry-Out.

Systolic arrays allow for high throughput due to high parallelism. There is parallel operation in both the pre- and post-synaptic neuron direction. However, the hardware overhead it introduces is enormous. Every single weight bit, stored in a single SRAM cell, is accompanied by two MUXs and a Full Adder. In the 3nm node, a single 6T SRAM cell is $0.01512\mu m^2$ [43], while the two 2-to-1 MUXs and the Full

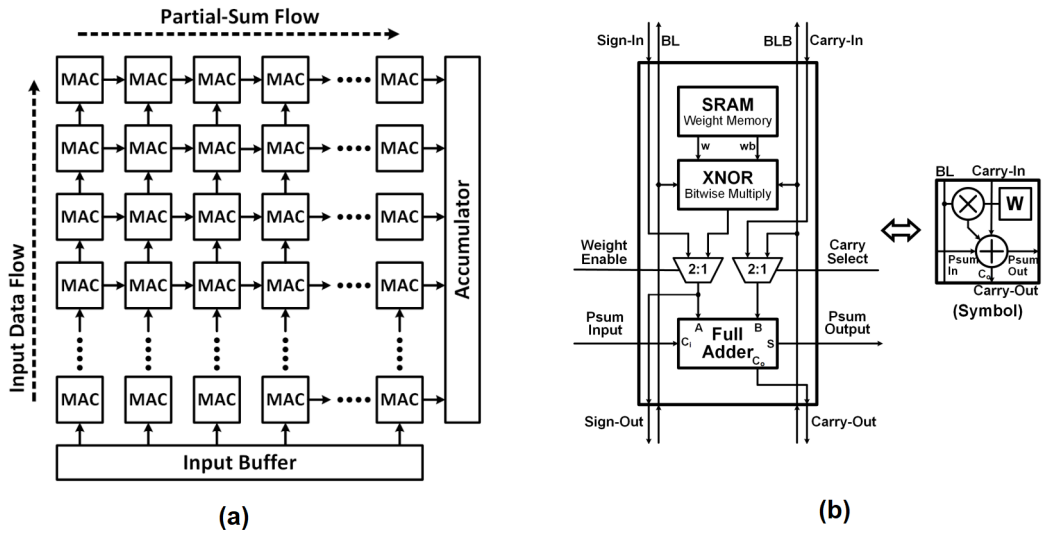


Figure 3.1: MAC Operation in-SRAM through the use of a systolic array; (a) Array-level overview; (b) Individual Systolic Block [42]

Adder together are $0.192\mu m^2$; adding the MUXs and FA increases the area by $13.7\times$. This indicates that adding hardware to every single cell results in too much area overhead.

3.1.2. Adder Trees

More CIM-MAC solutions focus on the use of Adder Trees to accumulate the MAC multiplication results [2, 4, 5, 6]. Figure 3.2 shows the structure presented in [4]. The synapse weights are stored in SRAM banks on the left. Weights in this system are four bits, hence the use of four-wide banks; for a Binary system such a bank would simply be a single column of SRAM cells. Inputs are entered into the system in a crossbar fashion using signal XIN. Using NOR gates and an adder tree per bank, the MAC operation is performed. In this implementation the MAC operation takes four clock cycles, as only one bit of XIN is entered into the system per timestep. However, for a fully binary system the MAC operation takes just one clock cycle.

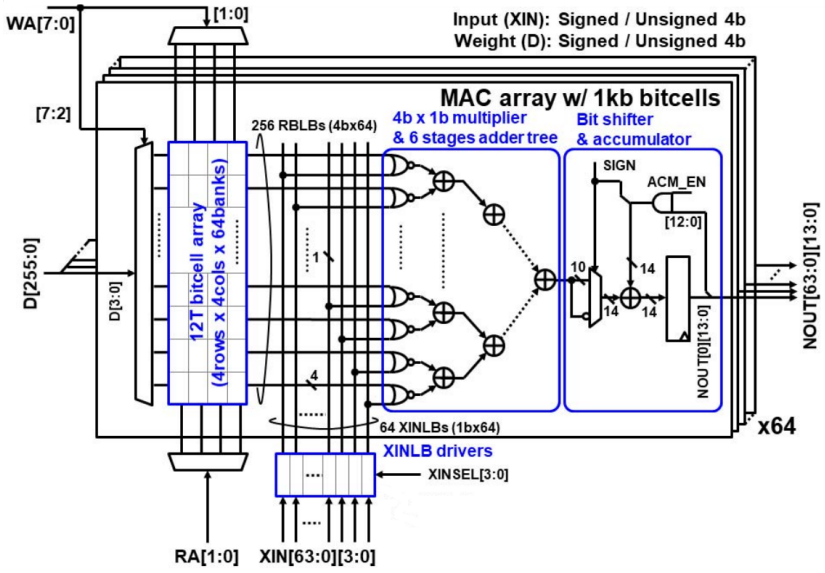


Figure 3.2: Adder Tree MAC structure used in [4].

Similar to the systolic array paradigm, adder trees allow for very high throughput; full parallelism is

maintained for the pre- and post-synaptic neurons. However, the issue of area and Power overhead persists, though less severe compared to the systolic array implementation. The following is a simplified overhead estimation.

Consider a simplified architecture where all weights and activations are Binary, and a 128×128 set of weights is used. Additionally, due to the use of a B-SNN, the NOR gates as used in [4] are not necessary; multiplication happens implicitly when Reading from a cell. Every column just has its own Adder Tree that reduces 128 Binary inputs to a single 7-bit number. In the 3nm node, a 128-cell column of SRAM cells is approximately $1.935\mu m^2$ [43]. An optimized Adder Tree, synthesized with Cadence Genus for the 3nm node, is approximately $12.909\mu m^2$. Thus, the area of a column of cells is increased by $7.7\times$. This is a significant reduction with respect to the systolic array system, but the overhead is still huge. Additionally, the structure of the SRAM array is broken up in order to add the adder trees. One of the main advantages of SRAM arrays is their high density; the cells are designed to fit as close together as physically possible, maximizing integration density (see Section 2.4). By inserting an adder tree between every column, this potential for density is not taken advantage of.

Finally, it is worth noting that both Adder Tree and Systolic Array solutions do not exploit one of the big advantages of SNNs; event-based computing. For SNNs, if no spikes arrive at a component, no computation is strictly necessary. However, Adder Trees and Systolic Arrays do not detect whether a spike arrives or not; they simply add all their inputs regardless of their values. All hardware in the system has to work in every timestep, regardless of how many spikes enter the system. By not exploiting event-based computing, potential Power savings are lost.

3.1.3. Sequential Accumulation

The CIM in-SRAM technique that minimizes hardware overhead and does exploit event-based computing in SNNs is Sequential Accumulation in the SRAM Periphery [7, 8, 9, 10, 11]. With this technique, no hardware is added inside the SRAM cell array. Instead, the array is kept exactly as-is, and accumulation hardware is added to the SRAM periphery, directly connected to the sensing systems.

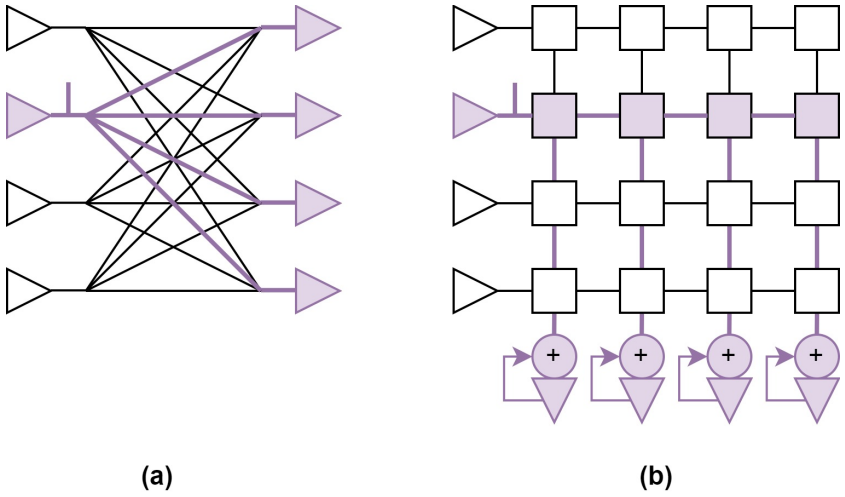


Figure 3.3: Sequential MAC operation visualized; (a) Spike Event in Fully Connected Layer; (b) Corresponding component utilization when mapped to Crossbar Array and using the Sequential MAC technique.

Figure 3.3 visualizes the Sequential MAC operation. Figure 3.3(a) shows what should happen in an SNN when a spike is sent by a pre-Synaptic neuron; it should be multiplied by all its output synapse weights and each result is added to the running V_{mem} of the post-synaptic neurons. Figure 3.3(b) shows how this operation is performed using the Sequential MAC technique; the spike corresponds to a Read of the memory row, which holds all the synaptic weights of the output connections of the spiking Pre-Synaptic Neuron. The spike-weight multiplication happens implicitly by reading from the cells. The results travel over the Bitlines to the sensing circuits of the SRAM at the bottom. This corresponds to a basic memory Read, without the need for any adjustments to the SRAM cells or array. The memory Read only happens when a pre-synaptic neuron actually spikes, making the system event-

based. Once sensed, the Bitline values are added to the V_{mem} registers of each neuron. This adder is the only hardware overhead necessary.

Sequential MAC induces minimal hardware overhead, does not break the SRAM array structure (maximizing the integration density), and it works on an event-basis. However, existing implementations still face two issues, both caused by the fact that multiple spikes from pre-synaptic neurons may arrive at a layer in a single timestep. First, this technique only allows a single spike to be processed in a layer per timestep. This lack of pre-Synaptic parallelism can have significant effects on the throughput. In [10] this issue is addressed through a form of approximate computing. However, this has negative effects on the SNN accuracy. The second issue is that some form of arbitration is needed to ensure that only one spike enters the SRAM per timestep, and that the non-processed spikes are remembered for later processing. In existing implementations these arbitration schemes are implemented using large Finite State Machine (FSM) systems, introducing hardware overhead and necessitating multiple clock cycles per spike arbitration.

3.1.4. Summary

In this Section, three methods of performing the MAC operation in-SRAM have been discussed: Systolic Arrays, Adder Trees, and Sequential Accumulation in the Periphery. Systolic Arrays introduce enormous hardware and therefore Power overhead. Adder Trees reduce this overhead, but still increase SRAM area by over $7\times$. Additionally, neither solution exploits the advantages of event-based computing that are offered by SNNs.

The Sequential Accumulation technique adds minimal hardware overhead and perfectly exploits event-based computing, making it the most suitable MAC in-SRAM solution for SNNs. However, existing solutions face the issue of low throughput caused by two factors: low Pre-Synaptic parallelism and slow, complex spike arbitration systems.

3.2. Inter-Layer Communication

Most Neural Networks consist of more than two layers of neurons. While two layers of neurons can be directly connected via just one synapse array [8], when more than two layers of neurons are implemented in hardware, some form of communication fabric is needed to transfer spikes from one synapse array to another. In this Section, the two most prevalent methods of communicating spikes on a chip are discussed: a Network-on-Chip (NoC) and Dedicated Wiring.

3.2.1. Network-on-Chip

A Network-on-Chip (NoC) is a form of shared interconnect used to connect all synapse arrays to all other synapse arrays. Figure 3.4 shows an example of a grid-based NoC interconnecting nine synapse arrays. Generally, spikes are encoded, mainly to include addressing information, and communicated between synapse arrays in a serial fashion using some form of bus and router system [44].

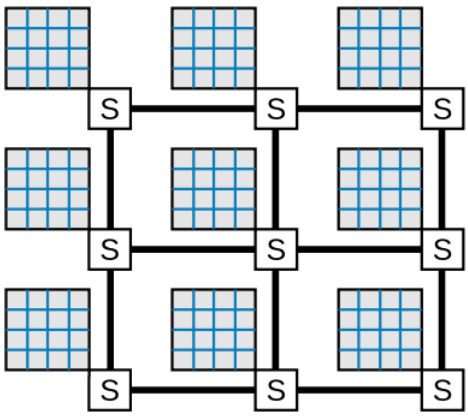


Figure 3.4: Example of a Network-on-Chip. Shown are nine synapse arrays, connected via routing blocks “S” to the global interconnect in a grid formation [44].

NoCs allow for flexible programming of a Neural Network onto a chip. This is essential for ex-

tremely large-scale neuromorphic accelerators such as Loihi [45] (130,000 neurons) or TrueNorth [46] (1,000,000 neurons). Without an NoC, it would be impossible to make full use of all these neurons, and the variations of Neural Networks that could be programmed would be very limited, not justifying the cost of these accelerators.

However, in this Thesis the focus is on small-scale ultra low-Power Neural Network accelerators. NoCs have been used for such systems [10], but the flexibility they offer is not worth the the encoding/decoding, routing, and buffering hardware overhead.

3.2.2. Dedicated Wiring

Instead of NoCs, use of Dedicated Wiring is more prevalent among small-scale ultra low-Power Neural Network accelerators [7, 8, 11]. Figure 3.5 illustrates what this typically looks like. Parallel wires travel directly from the outputs of the neurons belonging to one synapse to the next block. There is no encoding, decoding, or routing necessary, and since spikes are binary signals, buffers need to store at most a single bit.

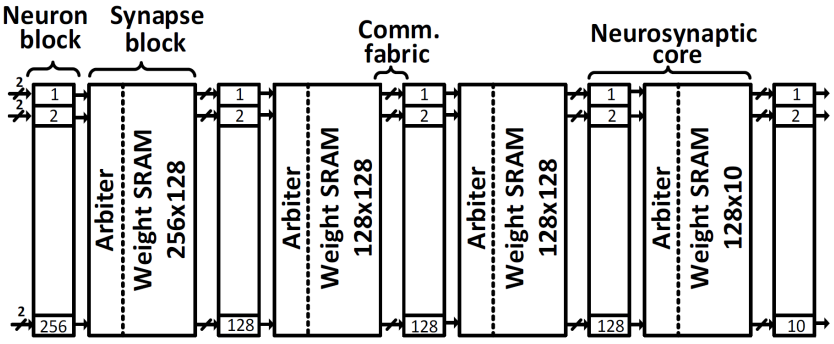


Figure 3.5: High-level overview of Dedicated Wiring for an SNN in the form of parallel connections between every synapse block [7].

Using Dedicated Wiring makes the communication fabric as simple as possible, at the cost of limited flexibility. The architecture now fully defines the maximum number of neurons for each layer, as well as the maximum number of layers that can be implemented. The only freedom left in programming the shape of a Neural Network onto the chip, is the use of less neurons than provided for a layer, or skipping a layer altogether by setting all synapse weights to '1'. Despite this lack of flexibility, when aiming at extremely low-Power applications, the simple, light-weight nature of Dedicated Wiring is the preferred solution.

3.3. Transposable SRAM

In this Section, various implementations of Transposable SRAM are discussed. As explained in Section 2.5, the minimum requirements for efficient on-chip learning are row-wise Read access (to send activations into the SRAM) and column-wise Read/Write access (to adjust synapse weights). Note that row- and column-wise access are relative terms; one can simply rotate an array to change rows into columns and vice versa.

3.3.1. Double Access Cell

Probably the most obvious way to add Transposable access to the original 6T SRAM cell is by duplicating the existing access ports [8, 12]. This is illustrated in Figure 3.6. Original access transistors M1 and M2 are duplicated as M7 and M8. These new access ports function identically to M1 and M2; if TWL is high, they conduct, giving access to the cell contents Q/QB via TBL/TBLB. The added ports allow Read/Write access in both the column- and row-wise directions.

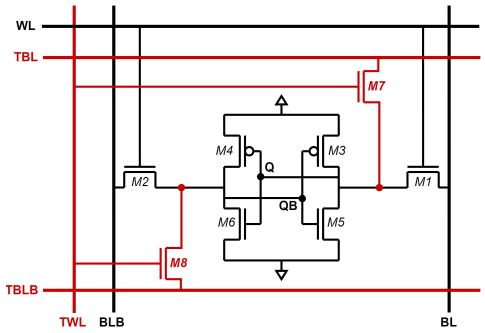


Figure 3.6: *Double Access* Transposable SRAM cell (from [12]). Circuitry added with respect to original 6T cell is highlighted in red.

The biggest issue with this cell is degraded cell stability. As explained in Section 2.4, SRAM cell design is mainly concerned with maintaining data hold stability under Read conditions while also maintaining writability. The data hold issues are caused by the access transistors M1 and M2. By adding even more of these problematic access transistors, the cell stability is made even worse, making the SRAM design balancing act more difficult.

3.3.2. 7T Cell

The most important realization in Transposable cell design for SNNs is that there is no need for Read-/Write access in both the column- and row-wise directions. One of these directions just needs to have Read access. The original 6T cell provides Read/Write access in one direction, so in the other direction only a Read port needs to be added.

The big advantage of a Read-only port is that the cell content can be read out in a *decoupled* manner, namely by only connecting the *Gate* of a Transistor to the content of the cell. If one would need to write to the cell, a Gate connection would be insufficient, but to extract information from the cell it is perfectly suitable. Connecting only an additional Gate to the content of the cell influences the cell stability as little as possible. An additional advantage of decoupled Read Ports is that local voltage scaling can be applied. The supply voltage used for the decoupled circuitry can be different from the voltage used for the rest of the SRAM. Thus, one may for example precharge the decoupled Bitline to a voltage $V_{prech} < VDD$ in order to save Power.

The most minimalist Decoupled Read implementation is found in [13], shown in Figure 3.7. It adds only Transistor M7, with its gate connected to Q. To Read the cell, a current is produced in TWL. If Q = '1', M7 conducts and the current is passed to TBL, pulling it high, which can be sensed at the edge of the array. If Q = '0', M7 does not conduct, and TBL remains discharged.

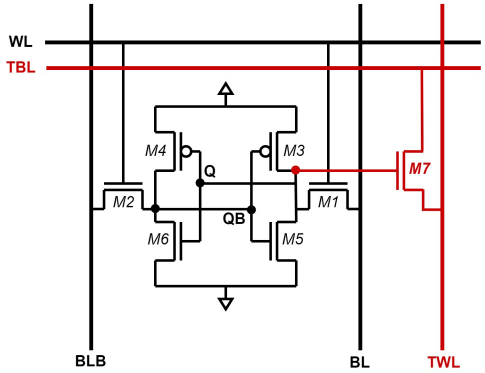


Figure 3.7: *7T* Transposable SRAM cell (from [13]). Circuitry added with respect to original 6T cell is highlighted in red.

To read from the cell, a current has to travel all the way over TWL, through M7, and then all the way over TBL to the sensing circuit. This is a very long, high-resistance and high-capacitance path. Not only would a Read operation take a long time, it would also require a lot of Power, as a current needs to be

driven high enough to conduct over this high resistance and capacitance path. To make the operation feasible, M7 would have to be very wide in order to carry enough current for it to be sensed at the end of TBL. Thus, the decoupled Read will consume a lot of Power, be very slow, and necessitate a very wide M7 transistor.

3.3.3. 6T-AND Cell

From the *7T* cell it can be concluded that adding a single Transistor is insufficient for creating an efficient decoupled Read port. In [47] an extra Transistor is added, as shown in Figure 3.8. Now, M7 connects with its Gate to QB, while M8 connects with its Gate to Q. The idea is that M7 and M8 together form an AND gate between the cell content and TWL. If QB = '1' (Q = '0'), M7 conducts and pulls TBL to Ground. If Q = '1' (and thus QB = '0'), M8 conducts and TBL is instead connected to TWL.

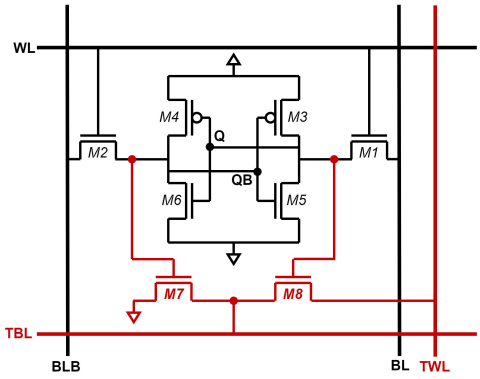


Figure 3.8: *6T-AND* Cell (from [47]). Circuitry added with respect to original 6T cell is highlighted in red.

Similar to the *7T* cell, though now only if Q = '1', a current must be generated on TWL, travel over all of TWL, through M8, and over all of TBL to be sensed. This allows Pulse With Modulated signals to be generated on TWL, be AND-ed with the cell content, and sensed on TBL. However, for a B-SNN, such Pulse Width Modulation is not necessary. Without this advantage, the implementation just suffers from same issues as the *7T* cell; the path over TWL, M8, and TBL is long and has high resistance and capacitance, resulting in a slow, Power-hungry Read, and necessitating a wide M8 transistor.

3.3.4. 6T + 2 PMOS Cell

The issue of the long Read path is addressed in [9, 17], shown in Figure 3.9. M7 connects with its Gate to QB, while M8 connects with its Gate to TWL. TWL is active low, meaning a Read operation occurs when TWL is pulled to Ground, in which case M8 conducts. If QB = '0' (Q = '1') M7 conducts as well, opening a path from the cell's VDD connection to TBL and charging up TBL, which can be sensed at the end of the line. Now the current path is split in two; first a path from the Wordline Driver which discharges TWL, followed by a separate path from the cell's VDD, over TBL, to the sensing circuit. Thus, the Wordline Driver no longer needs to provide an enormous current, as was the case for the *7T* and *6T-AND* Cells. Additionally, the authors in [17] claim that the cell is event-driven; as the cell's VDD charges TBL, there is no precharging necessary, so the line is only charged/discharged when absolutely necessary.

The cell improves over earlier designs, but still has some flaws. Firstly, the argument regarding its event-driven nature is flawed. Using a precharge technique, as is common in conventional SRAM, precharging only happens after a Bitline is discharged, meaning when a '0' is read. If no Read happens, or if '1' is read, the line remains at VDD and no charging is needed. For the technique in [17] the inverse is true; no charging/discharging happens if no Read happens or a '0' is read, but when a '1' is read, the Bitline needs to be discharged after to prepare it for the next Read operation. This means in effect the line is charged/discharged approximately the same number of times, regardless of the technique.

Second, the use of PMOS transistors and charging to VDD is less efficient than using NMOS transistors and discharging to Ground. This is because the existing SRAM cell Ground connection is stronger than the connection to VDD, as it already needs a strong Ground connection to discharge BL/BLB during the conventional Read operation. Additionally, NMOS transistors have a higher current carrying

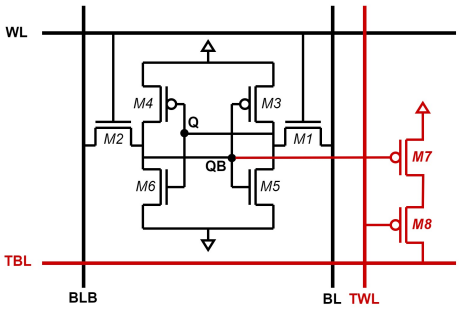


Figure 3.9: *6T+2PMOS* Cell (from [17]). Circuitry added with respect to original 6T cell is highlighted in red.

capacity than PMOS transistors and are thus more suitable for charging/discharging the bitlines. This is also supported by the architecture of the 8T standard SRAM cell [16]. To conclude, the principle of the *6T+2PMOS* cell is sound, but the implementation would be improved by utilizing NMOS transistors instead of PMOS transistors and by precharging the Bitline before reading.

3.3.5. 9T Cell

The SRAM cell presented in [14] follows the 8T standard SRAM cell architecture more closely, as can be seen in Figure 3.10. As opposed to the *6T+2PMOS* cell, it utilizes a precharge technique and NMOS transistors. M7 connects with its Gate to Q. It then connects two decoupled Read ports to M7 using M8 and M9; M8 provides Transposable access, while M9 provides decoupled Read access in the same direction as the original Read/Write ports, similar to the standard 8T SRAM cell. A decoupled Read can happen via RWL or TWL, giving results on RBL or TBL respectively.

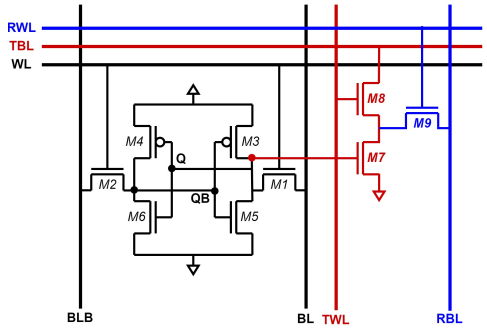


Figure 3.10: *9T* Cell (from [14]). Circuitry added with respect to original 6T cell is highlighted; red highlights the Transposable circuitry, blue highlights the additional circuitry giving non-Transposable decoupled Read access.

There are two important things to note about the *9T* cell. First, M7 connects to Q and thus discharges TBL/RBL if Q = '1', meaning the inverted content of the cell is read. For the B-SNN it is more convenient to read the non-inverted content, which is easily achieved by connecting M7 to QB instead. Second, if both RWL and TWL are activated, meaning a decoupled Read happens in both column- and row-wise directions, M7 would need to discharge both RBL and TBL. Discharging two Bitlines means M7 needs to be able to handle all the current from both. Thus, either M7 needs to be very wide, or the design needs to be limited such that only RWL *or* TWL is active at any time.

3.3.6. Barrel Shifter

A final design worth noting is presented in [11, 48], and is shown in Figure 3.11. It takes a completely different approach from the other designs presented in this Chapter. With this design, no adjustments are made to the SRAM cells themselves. Instead, a MUX is placed before every memory block, which are indicated by $w_{x,y}$ in Figure 3.11. By intelligently placing the weights in the array and using the MUXs to address these weights with specific patterns, one can Read the memory blocks in both row- and column-wise directions. Due to the placement of weights in the memory, a logic-based Barrel Shifter is needed to shift the memory blocks such that they are in the correct order.

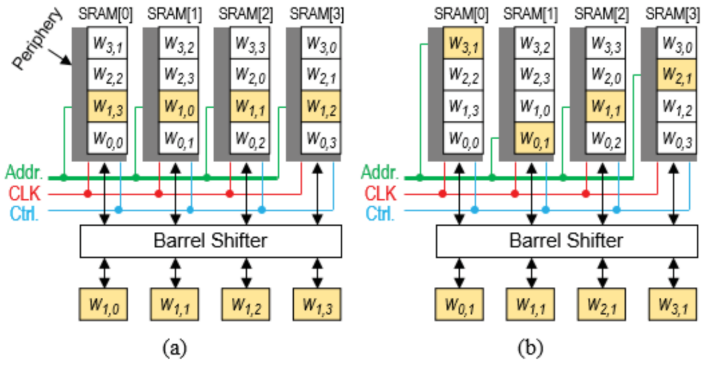


Figure 3.11: *Barrel Shift* Architecture [48]. (a) Row-wise access; (b) Column-wise access.

Though this design may be advantageous for certain systems, it is not efficient when storing Binary weights. For Binary weights, the memory blocks would become single SRAM cells. As such, a MUX is needed for every single SRAM cell. A MUX is larger than an SRAM cell, and every MUX requires separate control wires. Additionally, the MUXs and wires break up the efficient, regular structure of the SRAM, which is one of the key strengths of SRAM as discussed in Section 3.1. Thus, the *Barrel Shifter* architecture is not suitable for single-bit weights.

3.3.7. Summary

Table 3.1: Comparison Table for various Transposable SRAM schemes.

Cell Name	References	Cell Stability	Read Power	Area Overhead	Read Speed
Double Access	[8, 12]	-	+	+	++
7T	[13]	+	--	++	--
6T-AND	[47]	+	-	+	--
6T+2PMOS	[9, 17]	+	++	+	+
9T	[14]	+	+++	-	++
Barrel Shifter	[11, 48]	++	+	--	--

Table 3.1 shows an overview of the discussed transposable schemes. For each scheme, the influence of the transposable addition is shown with respect to the cell stability and area, and reported are the transposable Read Power and speed.

For cell stability the main concern is connections to the cell content. The *Double Access* cell performs badly here, connecting two extra access transistors. The *7T*, *6T-AND*, *6T+2PMOS* and *9T* cells each utilize a decoupled Read port so they perform quite well. The *Barrel Shifter* scheme does not alter the cell, maximizing cell stability.

Read Power consumption is poor for the *7T* and *6T-AND* cells due to their long Read Paths. The *Double Access* and *Barrel Shifter* schemes simply use the conventional SRAM Read method. *6T+2 PMOS* and *9T*, however, implement a decoupled Read port that has the potential for Power savings from voltage scaling. The Read Power is slightly better for the *9T* cell compared to the *6T+2PMOS* cell, as it uses NMOS transistors.

Area overhead is similar for the *Double Access*, *6T-AND*, and *6T+2 PMOS* cells; they each add two Transistors. The *7T* cell adds just a single transistor and The *9T* cell adds three transistors. The *Barrel Shifter* adds by far the most area, necessitating a MUX and individual wiring for every single cell.

Finally, most notable in terms of Read speed are the *7T* and *6T-AND* cells; due to their long Read paths they are very slow. The *Barrel Shifter* is also slow, as address signals travel through every MUX in a memory row, so the final column is addressed with significant delay. The remaining cells are similar in speed, though the *6T+2 PMOS* cell is estimated to be somewhat slower than the other cells due to its use of PMOS transistors, which have a lower current-carrying capacity.

From this qualitative comparison it can be concluded that the best method for transposable access when storing Binary weights is the use of a decoupled port. It can also be concluded that the Read

operation should not be performed by pushing a current over the Wordline, a transistor, and the Bitline, but should instead be based on charging/discharging the Bitline via the cell. Finally, it is best to precharge the Bitline and discharge via NMOS Transistors to Ground.

4

Proposed Solution

In this Chapter, the proposed solution for a B-SNN accelerator using SRAM-based CIM tailored for Edge AI applications is shown. First, Section 4.1 gives an overview of the overall architecture of the proposed system. Then, the most important components of the system are discussed: the SRAM Macro (Section 4.2), the Arbiter (Section 4.3), and the Neuron Array (Section 4.4).

4.1. Architecture Overview

In this Section, a full overview of the proposed architecture is given. The overview starts with a system-level view, after which a high-level overview is given of a single Tile. Following this, the Tile is explained in full detail. Finally, the behavior of the system in time is explained.

4.1.1. System Overview

Figure 4.1 shows a high-level overview of the developed Accelerator architecture. It is compromised of a cascade of CIM Tiles. Each Tile represents a Fully Connected B-SNN layer. Spikes are transmitted as binary signals through a set of parallel connections from Tile i to Tile $i+1$. Note how this mimics the architecture presented in [7]. Spikes are presented at the input of a Tile as a Spike Request Vector R_{i-1}. The Spike Request Vector of Tile 1, R_0, is stored in a register array before Tile 1. Spike Requests are served by the respective Tiles, and once all spikes are served in all Tiles, the next Tile Timestep is started. At this point, a new set of input spikes is stored in the R_0 register array, and every Tile will have a new set of Spike Requests to serve.

The connections between Tiles are fully parallel. It is very common for Neural Network accelerators to utilize encoding and decoding schemes and a Network on Chip (NoC) to transfer spikes between CIM Tiles [10, 45, 46, 49]. This allows for a more flexible system, where the user has more freedom in programming a Neural Network onto the system. For very large Neural Network accelerators, such as Loihi [45] and TrueNorth [46], this flexibility is necessary, as these systems are large and costly, and therefore need to be flexible to be practical. In the case of this Thesis, the aim is to create a very small, lightweight Neural Network accelerator. Therefore, the choice fell on the proposed minimalist communication fabric, as it results in the highest energy efficiency [7] and throughput for a small Neural Network.

The architecture presented here simulates a 768:256:256:256:10 Neural Network, specifically tailored for MNIST digit classification inference [50], as will be explained in more detail in Section 5.1.2. This is because MNIST is the typical benchmark for these types of systems. However, the presented architecture can be seen as a framework for general design of B-SNN accelerators.

4.1.2. High-Level Tile Overview

Inside each Tile there are three main components: the Arbiter, the SRAM Macro, and the Neuron Array. The SRAM Macro is a fully working SRAM system, containing Write Drivers, Wordline Drivers, sensing circuitry, precharge circuitry, timing control, and an array of multiport transposable SRAM cells. More specifically, each cell has p ($1 \leq p \leq 4$) Read ports which are used for row-wise Read access to the SRAM (purple in Figure 4.1), as well as one Read/Write port which is used for column-wise Read/Write

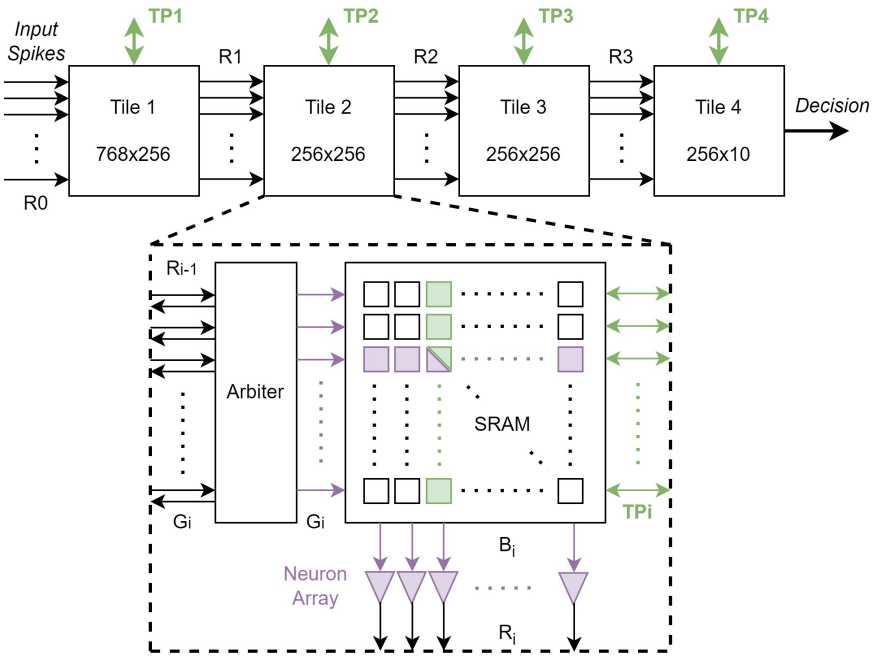


Figure 4.1: High-level overview of the developed macro architecture. Purple indicates inference Read access, while Green indicates transposable Read/Write access.

access (green in Figure 4.1). A more in-depth explanation of what the SRAM Macro and its cells look like is given in Section 4.2.

In Tile i, the Arbiter takes Spike Request Vector R_{i-1} as its input. Every clock cycle, the Arbiter serves a set of requests from R_{i-1}, which are allowed to enter the SRAM Macro. These selected requests are called the Granted Requests, represented by G_i. These Granted Requests will activate their corresponding Wordlines in the SRAM Macro simultaneously using the multiple Read ports. This results in a set of values being read on the SRAM Macro Bitlines, represented by B_i. B_i is presented to the Neuron Array, where it determines the change in the membrane potentials V_{mem} of the Neurons. G_i is also sent back to the previous Tile, where it is used to remove the Granted spikes from the Request Vector.

After serving all the spikes in R_{i-1}, for every neuron j if $V_{mem,j} \geq V_{th,j}$, the neuron sets its output to '1', indicating that it wants to send a spike to the next Tile. All neuron outputs r_j together form R_i, the Request vector for the next Tile. The final Tile behaves slightly differently. Instead of comparing to a V_{th} and creating a Request Vector, all $V_{mem,j}$ are compared to each other and the index of the neuron with the highest V_{mem} is spit out as the *Decision* of the network.

4.1.3. Detailed Tile Overview

The overview given in the previous Section misses some key details to describe a Tile in full detail. The main piece of information missing from the overview presented in Figure 4.1 is that the maximum SRAM array size, for any number of ports p, is 128×128. The reason for this limit is explained in Section 4.2. Because of this, each Tile contains multiple smaller SRAM Macros:

- Tile 1: $6 \times 2 = 12$ Macros of size 128×128
- Tile 1,2: $2 \times 2 = 4$ Macros of size 128×128
- Tile 4: 2 Macros of size 128×10

Figure 4.2 gives a more detailed view of what Tile 2 and 3 look like. This structure can be extrapolated to Tiles 1 and 4. Note that only the Inference side of the system is shown (purple in Figure 4.1); the transposable Read/Write access (green in Figure 4.1) is not shown, but is the same for every 128×128 SRAM Macro. Note also that the connection lines representing p-bit connections are made thicker to distinguish them from the thinner 1-bit connections (R_{i-1} and R_i).

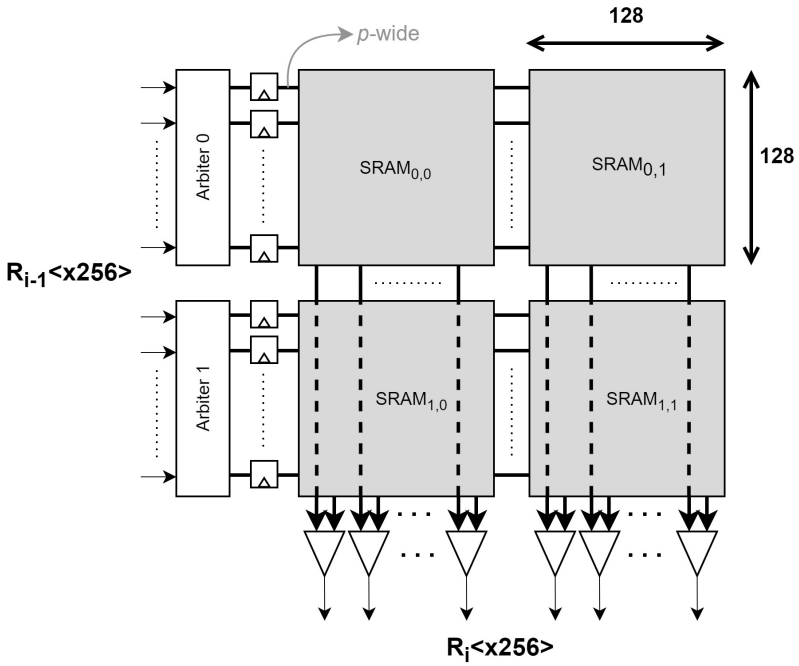


Figure 4.2: Detailed view of Tile architecture for 256×256 Tile. Thin connecting lines represent single-bit connections, thick lines represent p-bit connections.

Figure 4.2 shows how four SRAM Macros are used to simulate a 256×256 layer. Every Macro has p Read ports. Every Row of Macros can have its own p-port Arbiter. Therefore, a total of $2 \times p$ spikes can be served per clock cycle. The same holds for Tile 4. For Tile 1, as there are six rows of Macros, up to $6 \times p$ spikes can be served. Important to note is that it is not guaranteed that this many spikes are actually served in a clock cycle. Arbiter 0 and Arbiter 1 connect to 128 elements of R_{i-1} each. If for instance in the first 128 elements of R_{i-1} no spike requests are waiting, with all requests in the second 128 elements, then Arbiter 0 will serve no spikes, and only Arbiter 1 serves any spikes.

Also of note in Figure 4.2 is that the output Bitlines of all the SRAM Macros in a column feed into the same Neuron Array. Thus, each neuron must have $2 \times p$ input ports. The same holds for Tile 4, while for Tile 1 each neuron must have $6 \times p$ input ports. In Section 4.4 it is explained how the neurons deal with this many inputs.

4.1.4. System Timing and Pipelining

In this Subsection the timing of the circuit is explained. As can be seen in Figure 4.2, an array of registers is placed between the Arbiters and the SRAM Macros. Additionally, the outputs of the SRAM Macros, after decoding, are stored in V_{mem} registers. This creates two pipeline stages inside a Tile; the Arbiter stage and the SRAM+Neuron stage. The longest of these two stages determines the maximum clock frequency of the full system.

Additionally, each neuron j contains a register to hold r_j (its request for a spike, all neurons together forming R_i) until its request has been granted. Every Tile serves its input Request Vector R_{i-1} until it is empty. This results in updates to V_{mem} of all neurons in the Tile. Once every Tile has emptied its input Request Vector (i.e. the slowest Tile has finished), all the neurons will set their r_j for the next Tile Timestep based on whether $V_{mem} \geq V_{th}$. This updates R_{i-1} for each Tile, after which the Tiles can start serving again.

Figure 4.3 shows this process per clock cycle (cc). Shown is a single Tile i and the neurons of Tile $i-1$. Assumed is that each Tile can process one spike per clock cycle. The image shows the state of R_{i-1} and R_i for several clock cycles. As the Figure shows, both R_{i-1} and R_i are emptied one by one by the Arbiter (served spikes are highlighted green), and once they have been served they are removed from the Request Vectors. At Time = 2 cc R_{i-1} is empty, but R_i is not yet. In the next clock cycle both are empty. Assuming R_i was the last of all the Tiles to be emptied, this means the Tile

Timestep is complete. Subsequently, all neurons in all Tiles compare the end result for V_{mem} with their V_{th} and based on this send out new spike requests. This is visible in the last image of Figure 4.3, where the next Tile Timestep starts and all neurons have new requests pending at their outputs, forming new R_{i-1} and R_i vectors.

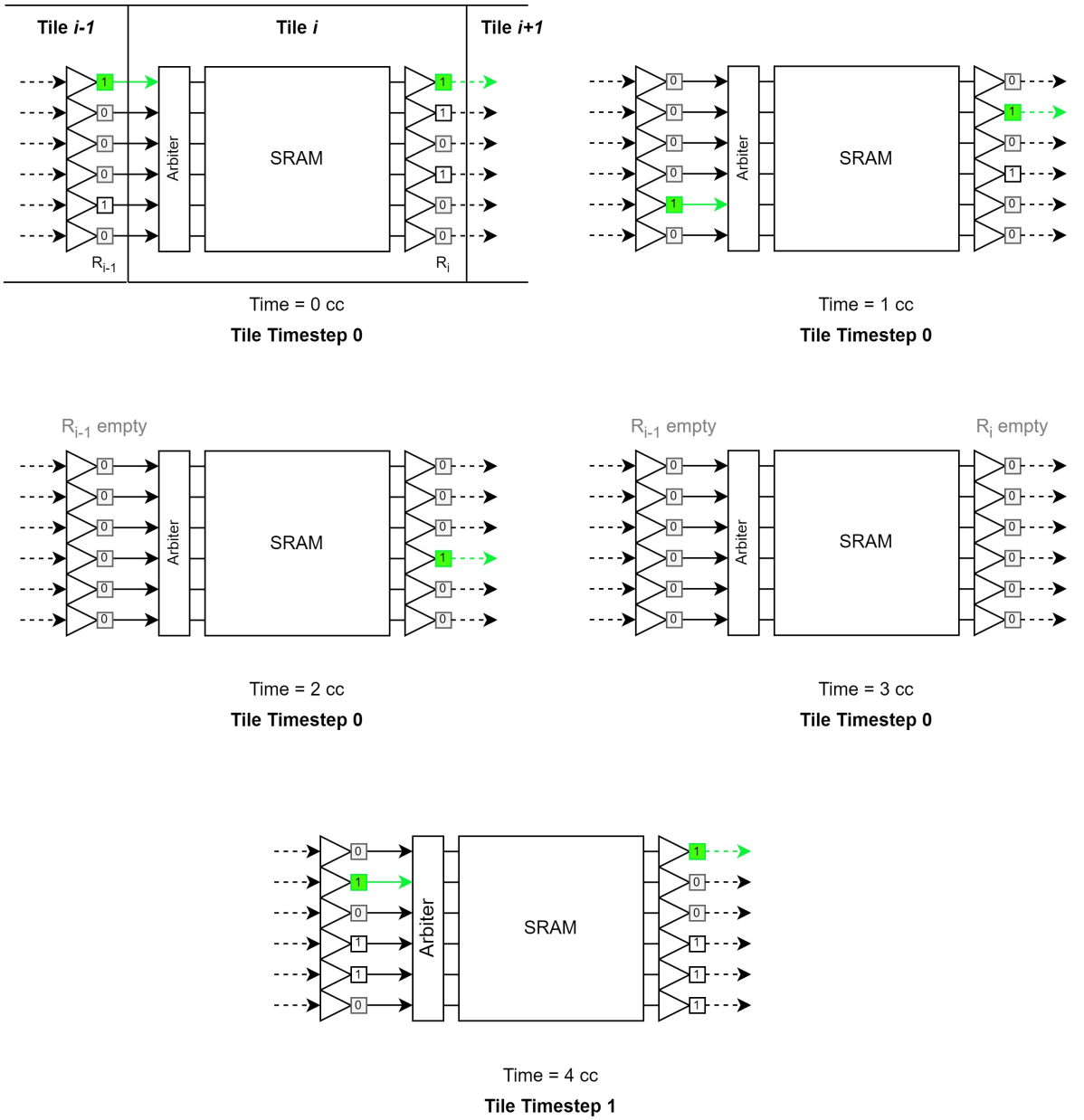


Figure 4.3: Visual Representation of spikes propagating through Tile. Indicated are Clock Cycles (cc) and Tile Timesteps.

As the Figure shows, all Tiles serve their input Request vector simultaneously. In effect this means the layers of the Neural Network are pipelined, with every Tile forming a pipeline stage. The stage lasts for a full Tile Timestep, which is the time it takes for the longest of the Request Vectors currently in the system to be emptied. Tile Timestep 0 shown in Figure 4.3 lasted four clock cycles.

Every Tile Timestep a new set of inputs can be presented at the input of the Accelerator. As a result, the average throughput can be computed as one Inference per average Tile Timestep.

4.2. SRAM Macro

In this Section, the SRAM Macro is introduced. It is explained what the multiport transposable SRAM cell looks like, first at schematic level and then at layout level. Then, an overview of the full SRAM Macro is given.

4.2.1. Cell Schematic

As explained in Section 2.5, in order to significantly increase the efficiency of online learning, it is essential to make the SRAM transposable. Additionally, to improve the inference throughput, having multiple Read ports would allow the system to process multiple spikes in parallel in every SRAM Macro. Therefore, the proposed SRAM cell has one dedicated column-wise Read/Write port and multiple row-wise Read ports. The most efficient way to achieve this is to maintain the original 6T SRAM cell structure, but rotate it 90 degrees, as shown in Figure 4.4; transistors M1-M6 compose the original 6T cell. The WL now runs vertically and the BL/BLB run horizontally. This provides column-wise Read/Write access to update the weights of the SRAM.

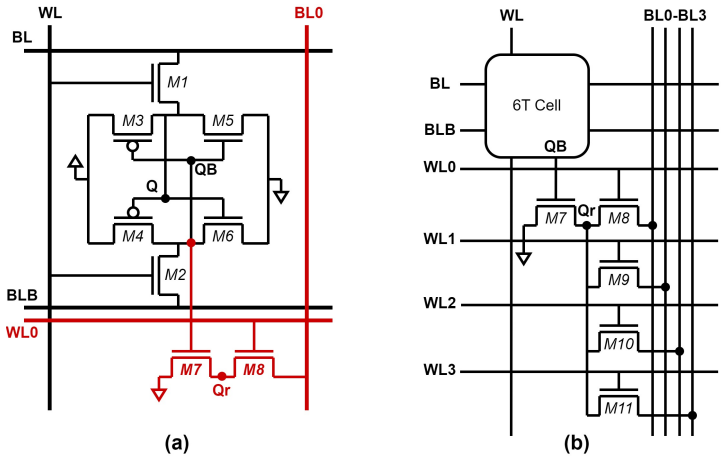


Figure 4.4: Proposed transposable SRAM bitcell schematic; (a) Single Port (added circuitry highlighted in Red); (b) Four Ports.

Then, in order to add the first row-wise Read port, two transistors M7 and M8 are added to this cell, similar to the technique used in the *6T+2PMOS* transposable cell [17]. However, instead of adding PMOS transistors, NMOS transistors are added, similar to the *9T* transposable cell [14], as well as the standard 8T SRAM cell [16]. The resulting cell is transposable, as the newly added WL0 runs horizontally and BL0 runs vertically, giving row-wise Read access.

The Read port works as follows: M7 connects with its Gate to QB, the inverted content of the cell. Before a Read operation, BL0 is precharged to V_{prech}. When WL0 is driven to '1', M8 conducts and connects BL0 to node Qr. If QB = '1' (Q = '0'), Qr connects to VSS, and BL0 is discharged to VSS. If QB = '0' (Q = '1'), M7 does not conduct, and BL0 is not discharged and instead remains at V_{prech}. A sensing circuit must then be able to distinguish whether BL0 is at VSS or V_{prech} to complete the Read operation.

By connecting M7 only with its Gate to the cell content, it forms a decoupled Read port, minimizing its influence on the cell stability and allowing for voltage scaling on the decoupled portion of the SRAM. Specifically, V_{prech} can be made lower than VDD. As Power consumption scales with the square of the Voltage, this can lead to significant Power savings. However, a lower V_{prech} also results in slower precharging of BL0, slowing down the system as a whole. This presents a trade-off between Power savings and Read speed.

The proposed cell design has the advantage of allowing the addition of more Read ports at minimal cost, as can be seen in Figure 4.4(b). To add another Read port, all that is required is to replicate WL0, BL0, and M8. The added transistor should connect Qr to the newly added BLx, and its gate should be connected to WLx. Figure 4.4(b) shows how four Read ports are added in this way. Important to note is that if WL0-WL3 could all be active at the same time, M7 would have to be able to drain the charge of four Bitlines at once. This means M7 would have to be extremely wide to carry a lot of current. However, due to the way the Arbiter functions, as will be explained in Section 4.3, per SRAM row only

one of WL0-WL3 is active at a time. Thus, M7 only needs to drain a single Bitline and can remain relatively small.

4.2.2. Cell Layout

Layout Explanation: Figure 4.5 shows the designed layouts of the proposed transposable multiport SRAM cells for 1, 2, 3, and 4-port cells in the imec 3nm FinFET technology node. For an explanation of layout design in this node, as well as the layout of a 6T SRAM cell, see Section 2.4.2. For images of the actual cell layouts in the *Virtuoso Layout Editor*, see Appendix A.3.

The layouts are shown with the FIN, GATE, and M0 layers in their actual place. MINT, M1, M2 and M3 are shown to the side of the layout to retain visibility. In reality these layers are deposited as strips of metal over the cell. If a strip to the side is wider, this indicates that the real metal strip would span across the full cell; if it is more narrow, this indicates that the strip is only used for routing inside the cell, and does not connect to the outside. For instance, BL/BLB in the MINT layer are wide and span across the cell, connecting to neighboring cells. On the other hand, Q/QB in the MINT layer are only used to connect a GATE and M0 portion internally in the SRAM cell.

Cell Borders: For every cell, a dashed border is shown, indicating the bounding box of the cell. Along with this bounding box, the height and width of the cells is shown. The layouts of the 1- and 2-port cells are slightly more complex; they are shown as two dual cells, with an A and a B-variant. Normally, SRAM cells are symmetric. However, in order to maximize the integration density, for the 1- and 2-port cells it was more efficient to design two variants that fit together exactly. The variants fitted together form a Dual Cell, which is symmetric and can be replicated like a normal SRAM cell.

The reason for the dual cell design, as well as the rather large bounding boxes for the 3- and 4-port designs, is that BL0-BL3 are not shared between horizontally neighboring cells. For the 6T SRAM cell, BL/BLB is shared between horizontally neighboring cells. This allows the cell to have its outside M0 strips overlap with its neighbors, as the only nodes on these outside strips are BL/BLB, VDD, and VSS, which can all be shared. However, BL0-BL3 are shared between *vertically* neighboring cells, and must be kept separate between horizontally neighboring cells. For the 3- and 4-port designs, the best way to ensure this was to simply take a larger bounding box and leave one empty strip between every cell. For the 1- and 2-port cells, the most efficient method was to design the presented asymmetric dual cells that fit together exactly to ensure no overlap of BL0.

A final note is that one might wonder how the 1P Dual Cell is as efficient as it can be, given that there are empty spaces in the layout. An alternative layout was designed, where these empty spaces are not present. The schematic for this layout can be found in Appendix A.4. However, this design runs into the issue of horizontally neighboring cells, meaning an empty strip is necessary next to the cell, similar to the 3P and 4P cells. This empty strip causes the alternative cell in Appendix A.4 to have a slightly larger area (about 4%) than the the proposed cell. Therefore the proposed cell in Figure 4.5 was chosen.

Port Number Comparison: Before comparing the presented cells, it is important to explain why the number of ports was limited to four. This can be explained by the 4-port cell in Figure 4.5. BL0-BL3 are routed in M1 and M3, running vertically, along with VSS and WL. All these lines together exactly fill up M1 and M3, leaving no room for more lines to be routed. Metal layers above M3 are reserved for global routing, and cannot be used in the cell design. Thus, if one were to add more Read ports, the newly added BL4+ would have to be routed next to the cell. This means the cell needs to be made wider than it already is. The extra area per Bitline would amount to almost 90% of the area of the original 6T SRAM cell. In this extra area, no transistors are added, creating a lot of empty, wasted space in the SRAM array. The larger the array, the more parasitics it induces, causing it to become slower and more Power-hungry. During the cell design it was estimated that adding more than four Bitlines would induce enough additional parasitics that diminishing returns would be observed at the system level. In the measurement Results in Section 5.2 this hypothesis is confirmed.

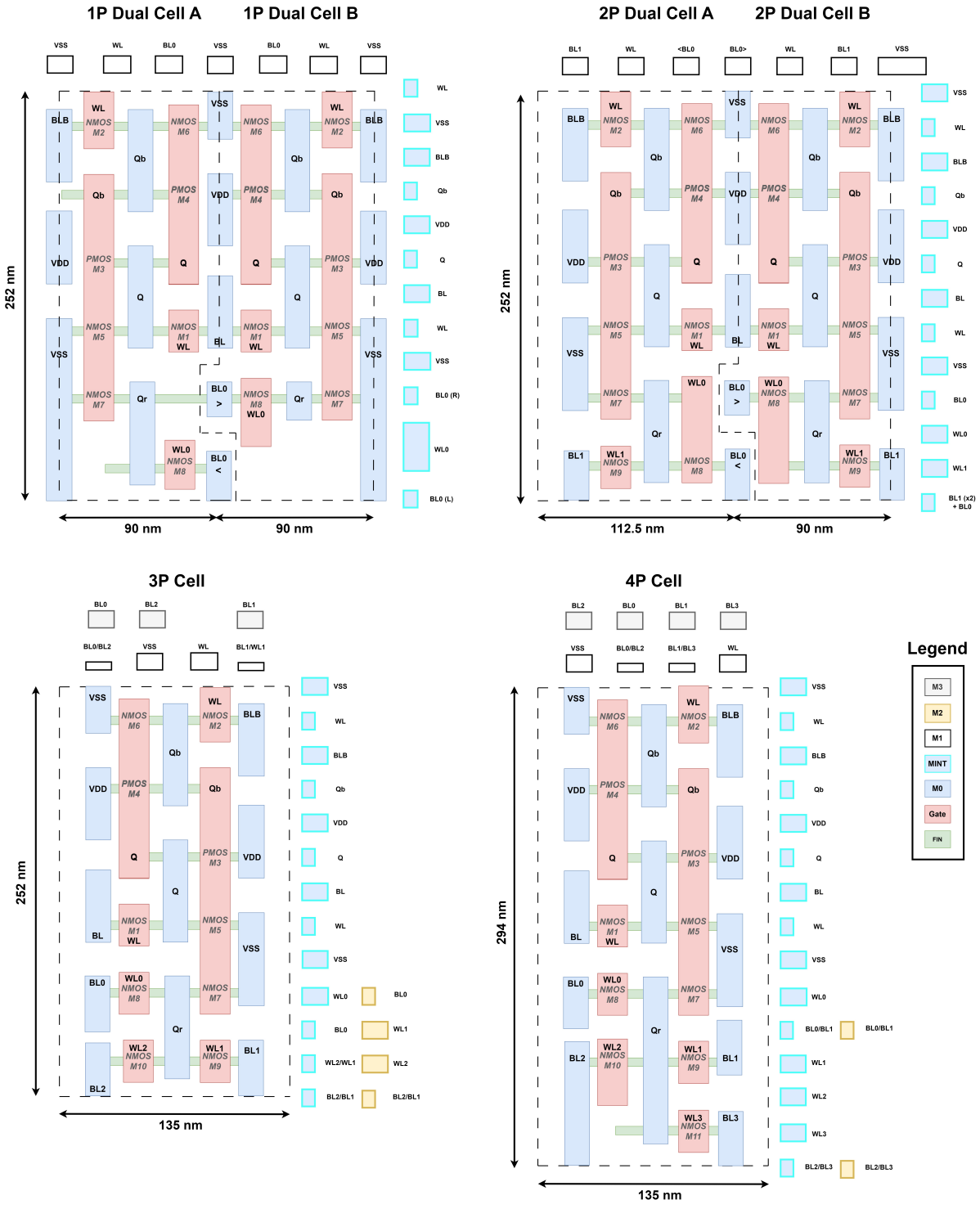


Figure 4.5: Schematic layouts of proposed 1, 2, 3, and 4-port SRAM cells (not to scale). MINT, M1, M2, and M3 layers shown to side of layouts.

4.2.3. Full Macro

Figure 4.6 shows an overview of the full Macro architecture. The main component is the array of SRAM cells, which are described earlier in this Section. The other components can be split into the Inference Read components (Purple) and Transposable Read/Write components (Green). Both have their own Timing Control circuitry. The timing control is a logic block that uses the CLK signal and R/W input signals to set the various SRAM control signals, which are the precharge, WL, Write, and Sense Amplifier enable signals. The exact logic in the block is technology dependent, as the timing for these signals can be very different for different technologies.

Transposable Read/Write Components: The Transposable Read/Write circuitry consist of two blocks; the Address Port and the Data Port. Inside the address port, a memory address is decoded to enable a single WL, which is driven high using its WL driver. This is a Buffer with an enable input.

Inside the Data Port there are three subcomponents. The Sense Amplifier Array consists of an array of SAs as shown in Figure 2.6, which sense the data on the Bitlines and present the sensed data to the output. Before the Sense Amplifiers, a set of 4-to-1 MUXs is placed, which reduces the number of sensed Bitlines to $N/4$. This is because the utilized SAs have a wider footprint than the SRAM columns themselves. Therefore, N SAs would not fit within the width of the SRAM array, which would make for very complex routing and inconsistent behavior between columns. Thus, only $N/4$ SAs are used. This also means only $1/4$ of a column can be read per Transposable Read operation. The Precharge Array consists of a set of single-fin PMOS transistors per BL/BLB pair. These connect the BL/BLB to VDD when precharging. Finally, the Write Driver Array consists of an array of Buffers, similar to the Address Port, which drive the BL/BLB to drive the input data to the cells. However, these Buffers do not drive to VDD and VSS, but instead to VDD and $V_{WD} < VSS$, to write to the cells using the Negative Bitline Voltage assistance technique, as explained in Section 2.4.4.

Inference Read Components: The Inference Read circuitry consist of two blocks; the WL Drive and the Data Port. Inside the WL Drive, Wordlines are driven high using Buffers, similar to the Transposable Address Port. The WL drivers are replicated p times for the p Wordlines per row.

The Data Port contains a Precharge Array similar to the Transposable Data Port, as well as the Sensing Circuitry. The Sensing Circuitry used for the decoupled Inference Read is not the same as for the Transposable Read. This has two reasons. First, the decoupled Bitlines do not have a complementary counterpart, while the Transposable Bitlines do. Thus, a differential Sense Amplifier cannot be used. Second, as explained above, SAs are larger than SRAM cells. A single SA already does not fit within the width of an SRAM cell, let alone four SAs as would be necessary for the 4P cell. Therefore, a simpler but slower method is chosen: sensing with Inverters. Two Inverters are placed back-to-back per Bitline. The first Inverter's VDD port is connected to V_{prech}, while the second is connected to VDD, so that the eventual output is a full-strength Binary signal to be sent to the Neuron Array.

4.3. Arbiter

In this Section, the Arbiter is described in detail. First, the functional requirements are explained, after which the implementation is shown. The VHDL code used for the synthesis of the Arbiter can be found in Appendix B.1.

4.3.1. Functional Requirements

The Arbiter receives a Spike Request Vector R as its input. The vector contains '1's to indicate which Pre-Synaptic neurons want to send a spike; the other elements are '0'. The Arbiter should produce a set of p vectors G. Each vector G should be one-hot coded to select a different Wordline to be activated in the next clock cycle. If less than p requests are pending, the leftover vectors G should be all '0'.

4.3.2. Implementation

Priority Encoder: The basic building block of the Arbiter is the Priority Encoder, highlighted at the top of Figure 4.7. It consists of a set of identical logic blocks, each performing the logic operations shown in the zoomed in portion of the Figure. If a logic block receives a request ($r = '1'$), and it is not blocked by a block to its left ($s[n] = '0'$) then this request is granted ($g = '1'$). If a block grants a request, it will stop all blocks to its right from granting a request by setting $s[n-1] = '1'$; this signal propagates to all

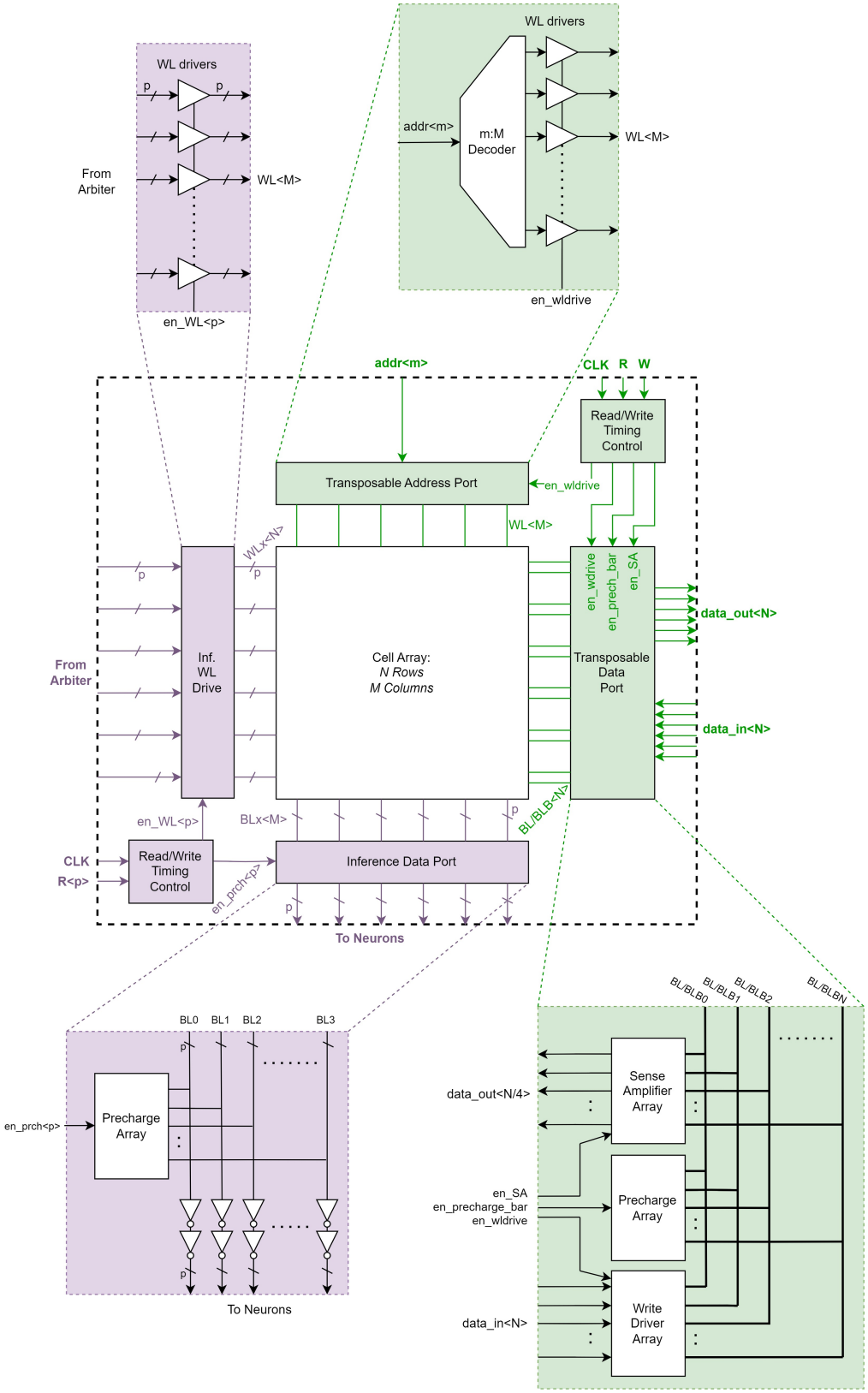


Figure 4.6: Overview of full SRAM Macro. Purple components relate to Inference Read access, Green components to Transposable Read/Write access.

blocks to the right. As a result, the Priority Encoder selects the leftmost request in R. The rightmost block outputs $s[n-1]$ as signal noR; if R is empty, noR is '0'. Each block also outputs r'. This output is the same as r, except if r was '1' and it was granted, in which case it is made '0'. As such, R' forms a masked version of R, with all non-granted requests still '1', but with the granted spike having been removed. This means R' can be propagated to a subsequent Priority Encoder, which can grant another request. By cascading Priority Encoders in this fashion, a multiport Priority Encoder can be created.

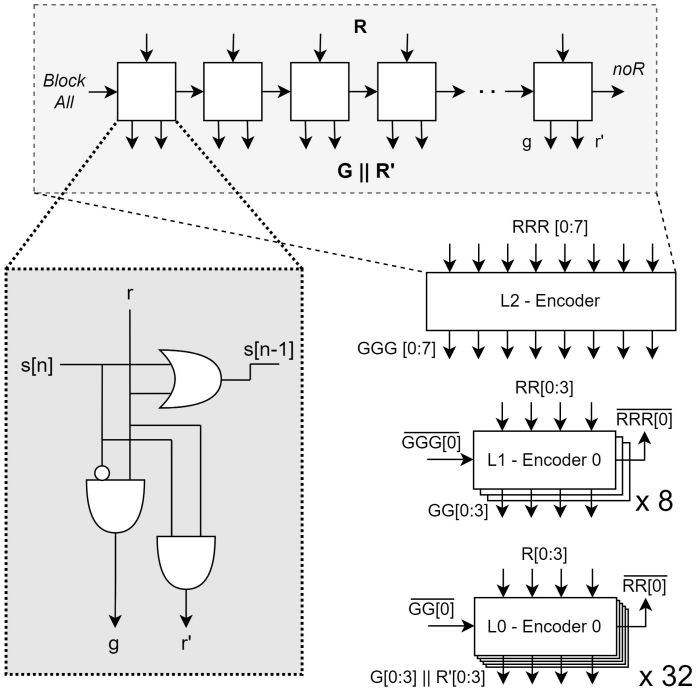


Figure 4.7: Proposed logic-based Arbiter consisting of a 3-layer tree structure. Highlighted is the main building block, the Priority Encoder, as well as its internals.

Tree Encoder: As explained in Section 4.1, the Arbiter should should work on a 128-wide vector R. From synthesis results it was found that building a 128-wide Arbiter just using the presented basic Priority Encoder structure resulted in an excessively long critical path, even for a single-port SRAM. To solve this, multiple smaller Priority Encoders are combined in a tree structure as shown in Figure 4.7:

- The 128-bit vector R is split into 32 chunks, which are each entered into 4-wide Priority Encoders at Level 0 (L0). Each L0 Encoder determines whether it has a request in its chunk, and based on that creates a higher-level Request Vector of 32 bits; RR.
- This 32-bit vector RR is split into 8 chunks and entered into 4-wide Priority Encoders at Level 1 (L1). Each L1 Encoder determines whether it has a request in its chunk, and creates a higher-level Request Vector of 8 bits; RRR.
- This 8-bit vector RRR is passed to a final 8-wide Priority Encoder at Level 2 (L2), which generates a high-level Grant vector GGG to Grant one of the L1 Encoders.
- The Granted L1 Encoder in turn generates Grant vector GG to Grant one of the L0 Encoders.
- The Granted L0 Encoder finally is allowed to Grant the highest-priority Spike in its 4-bit chunk. All the other L0 Encoders will output only '0'.

Effectively the Tree Encoder has the same input-output behavior as the original Priority Encoder, but with a shorter critical path. Additionally, as before, the L0 Encoders all generate R', containing all non-granted spikes. Multiple Trees can then be cascaded, creating a multiport Tree Encoder.

As explained in Section 4.1, each Tile is pipelined so that there are two stages; the Arbiter and the SRAM+Neuron. There is little control over the Read time of the SRAM, so it is taken as the limiting factor for the global clock frequency. All that is then necessary to not slow the system down any further,

is to ensure that the Arbiter takes less time than the SRAM+Neuron stage. Thus, different variations of the tree structure were synthesized until a variant had a shorter critical path than the SRAM+Neuron stage. In the end, the critical path of the Arbiter stage is reduced from $> 1100ps$ for a single port to $< 800ps$ for 4 ports, at the cost of at most 8.0% area overhead when comparing the basic Priority Encoder structure and the presented Tree Encoder. As will be shown in Section 5.5, the Arbiter takes up at most 5% of the overall system area, so the Tree Encoder's area overhead is negligible compared to the gains in throughput.

4.4. Neuron Array

In this Section, the Neuron Array is described in detail. First, the functional requirements are explained, after which the implementation is shown. The VHDL code used for the synthesis of the Neuron Array can be found in Appendix B.2.

4.4.1. Functional Requirements

Spiking Neuron Type: First, the choice of spiking neuron type needs to be discussed. As explained in Section 2.2, there are two main spiking neurons used in hardware implementations of SNNs: the IF and LIF neurons. More complex neurons exist, but they are much less convenient for hardware implementation. For this Thesis the IF neuron is chosen. This is because the LIF neuron provides benefits for time-based tasks, where a constant stream of data is entering the system. The LIF neuron can 'forget' old data and therefore 'focus' more on newer data. However, for this Thesis the time-static task of MNIST digit classification [50] is used as the benchmark application. Hence, the LIF neuron does not provide any significant advantage. However, it is worth noting that adding a linear leak factor to the presented IF neuron is a trivial task which can be done without incurring any substantial hardware cost.

Behavioral Requirements: The main neuron input is a set of Bitlines B_i. Every clock cycle, the Arbiters of Tile i select spikes from R_{i-1} to be sent into the SRAM Macros. It is not guaranteed that an Arbiter actually selects p spikes. Therefore, each Arbiter also outputs a vector v of length p to indicate which of the p SRAM ports are utilized and are therefore valid for reading; the non-valid ports should be ignored by the neuron. Thus, the neuron should read a set of Bitlines B_i and the vector v indicating their validity. All the valid Bitlines should be decoded from their {1,0} values to {+1,-1} values, summed together, and added to V_{mem}. This should happen every clock cycle until the Tile controller indicates that R_{i-1} is empty, at which point the neuron should compare V_{mem} to its personally stored V_{th}, and if $V_{mem} \geq V_{th}$ it should set its output register r to '1'. It should also reset V_{mem} to 0 so the process can start again in the next clock cycle.

The output registers r of all neurons in Tile i together form R_i, to be processed by Tile $i+1$. The Arbiter of Tile $i+1$ selects spikes from R_i and Grants them access to its SRAM Macros. When a spike is Granted, it will be made '1' in G_{i+1}. This G_{i+1} is not only used for SRAM Macro access on Tile $i+1$, but it is also sent back to Tile i and used as input g to each of the neurons. Whenever a neuron's spike is Granted ($g = '1'$) it should reset its output register r back to '0'.

An overview of how the Neurons of Tile i are connected between the SRAM Macros of Tile i and the Arbiters of Tile $i+1$ is shown in Figure 4.8. It shows the most important signals; the Bitlines B_i, the neuron's output requests R_i and the Granted vector from Arbiter $i+1$; G_{i+1}.

4.4.2. Implementation

Figure 4.9 shows a schematic view of an individual neuron. First, the input Bitlines b (width *w*) are decoded and added together, only counting the valid Bitlines as indicated by v (width *w*). This is a process optimized by the synthesizer. An example of a Decode & Add operation for a 16-bit input is shown in Table 4.1. In this example, a total of five Bitlines are unused and are thus marked invalid using an `X`. They are not counted towards the total sum.

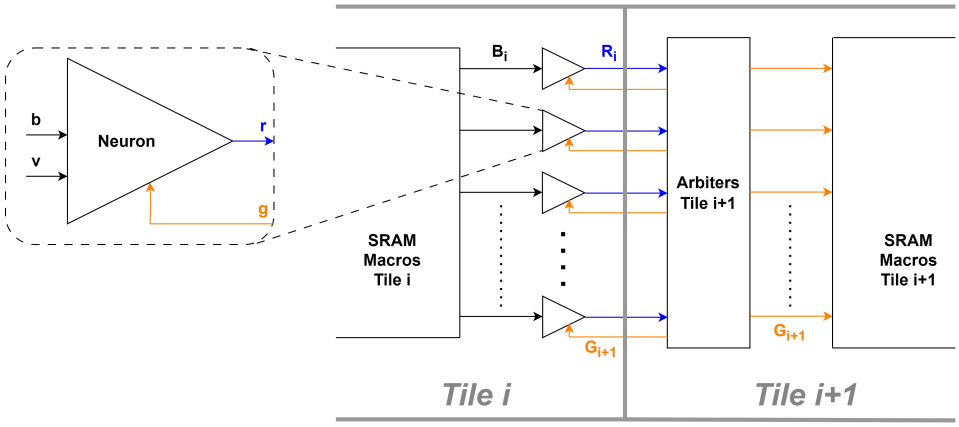


Figure 4.8: High-level view of neuron placement inside tile, and most important signals it interacts with.

Table 4.1: Example of a Decode & Add operation inside the neuron.

b	0011 1111 0001 0101
v	1100 1000 1111 1111
valid only	00XX 1XXX 0001 0101
decoded	--XX +XXX ---+ -+-+
summed	-3

The Decode & Add result is added to the content of the V_{mem} register (width *m*). Next to the V_{mem} register is a latch register storing V_{th} (width *t*). V_{mem} is compared to V_{th}, and if *R_empty* = '1' and $V_{mem} \geq V_{th}$, a '1' is written to the output register. If *R_empty* = '1' and $V_{mem} < V_{th}$ a '0' is written, and if *R_empty* = '0' the register content remains the same. *R_empty* is a control signal indicating whether the input request vector R_{i-1} of this Tile is empty. Finally, if the neuron has a pending spike ($r = '1'$) then it can be reset by the Grant signal g, originating from the Arbiter of the next Tile.

The Neuron Array is formed by replicating the presented neuron as many times as necessary. V_{th} of the neurons is written using a single global *t*-bit signal, and the enable port is connected to a decoder to select a single neuron to write V_{th}. Thus, V_{th} of only one neuron can be changed at a time. However, this is not a problem, seeing as replacing the column of weights associated with a neuron takes more than one clock cycle, and changing V_{th} can happen simultaneously.

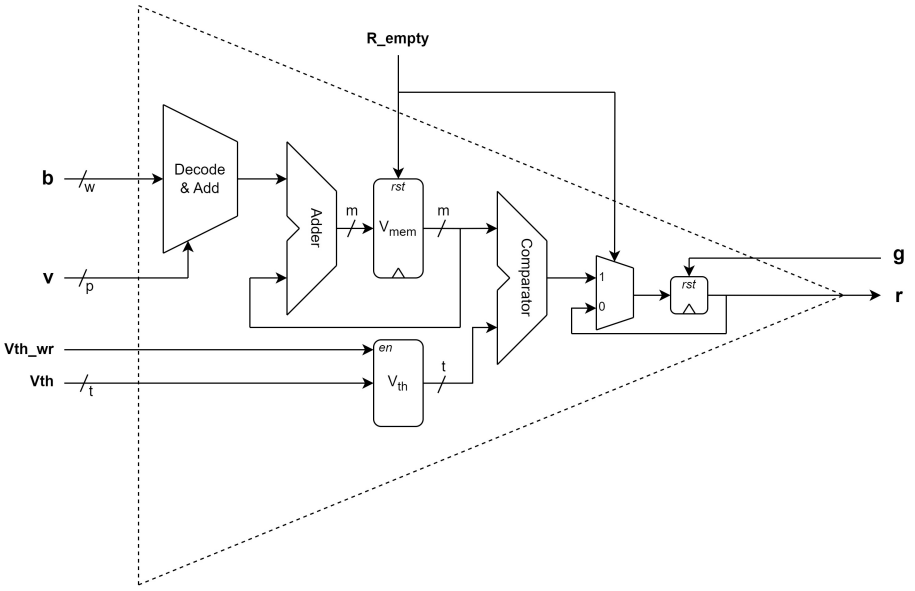


Figure 4.9: Schematic view of individual neuron architecture.

5

Simulation Results

In this Chapter, the simulation results for the proposed accelerator architecture are presented. First, Section 5.1 explains the simulation setup, at circuit-level, application-level, and at system-level. Then the circuit-level results for the SRAM Macro are given in Section 5.2, for the Arbiter in Section 5.3, and for the Neuron Array in Section 5.4. Following this, the system-level results are given in Section 5.5. Finally, Section 5.6 provides a comparison between the presented work and state-of-the-art systems in the field.

5.1. Simulation Setup

In this Section, the simulation setup for the measurements presented in this Chapter is described. The setup can be split into three parts: the circuit-level setup, the application setup including the Neural Network that is simulated, and the system-level setup.

5.1.1. Circuit-Level Setup

The full system is implemented in the *imec* 3nm FinFET technology node at transistor level, with a global supply voltage of 700 mV. To obtain the reported SRAM Macro results, SPICE-level simulations of the full 128×128 macros were performed using Cadence Spectre. In order to simulate the parasitics on the Bitlines and Wordlines of the SRAM array, resistance and capacitance parasitics were extracted from the presented SRAM cell Layouts. Capacitance parasitics were extracted using Calibre PEX, which is an automated tool built into Cadence Virtuoso. Resistance parasitics were determined based on the line geometries and *imec*'s restricted in-house datasheets. All simulation measurements are based on accessing the worst-case row, column, or cell. The worst-case row/column is the row/column furthest from the Sensing Circuitry. The worst-case cell is the cell in the worst-case row that is furthest from the Wordline Drivers.

Simulations were performed at the SS (Slow-Slow) Process Corner [51]. This means both NMOS and PMOS devices are simulated with lowered carrier mobilities, hence the Slow-Slow designator. As such, the SS corner represents the lower bound for speed of operation in the circuit, meaning typically the devices will operate at higher speeds. This makes it so that the presented choices for clock frequencies can be considered safe.

The Arbiter and Neuron Arrays are described at RTL level using VHDL. They were synthesized using Cadence Genus to obtain SPICE-level descriptions, which are simulated with Cadence Spectre.

5.1.2. Application & Neural Network Architecture

The most common benchmarking application for low parameter resolution SNN accelerators aimed at low-power operation [7, 10, 12] is the MNIST handwritten digit classification data set [50]. MNIST is a collection of 60,000 training images and 10,000 testing images of handwritten digits (0-9) scaled to 28×28 pixels and colored in grayscale (256 levels; 8 bits).

To compare the proposed system in this Thesis to existing designs, the proposed system had to be designed to run MNIST classification. To do so, a Binary Neural Network (BNN) was trained for this task, after which it was converted to a B-SNN [32]. The accuracy results and system-level results

are derived from simulating running this B-SNN on the presented architecture. The BNN was trained in Python using the *PyTorch* and *Brevitas* packages. *PyTorch* provides an easy-to-use framework for Neural Network training, while *Brevitas* is a *PyTorch* library for Neural Network quantization. *Brevitas* allows the user to directly train a BNN in *PyTorch* using Quantization-Aware Training.

The MNIST images were binarized at a threshold of 0.3. The 28×28 MNIST images result in 784 input elements. However, as the presented system uses 128×128 SRAM macros, a multiple of 128 is much more convenient. Thus, to create an input vector $6 \times 128 = 768$ elements, a 4×4 patch of pixels was cut off all four corners of each image.

Every neuron stores its own Bias, which is converted to a Threshold Potential for the B-SNN. The activation function of the neuron is implemented using a custom class where the Forward Pass behavior is a simple Heaviside function:

$$H(x) = \begin{cases} 1, & \text{if } x > 0 \\ -1, & \text{otherwise} \end{cases} \tag{5.1}$$

while for the Backward Pass a surrogate gradient $F'(x)$ is used [52]:

$$F(x) = \frac{\arctan\left(\sqrt{10}x\right)}{\sqrt{10}} \rightarrow F'(x) = \frac{1}{1 + 10x^2} \tag{5.2}$$

A brief exploration of BNN architectures was performed, keeping all hidden layers to multiples of 128, resulting in the following Fully Connected network architecture: 768:256:256:256:10. The network was trained for 240 epochs with the Adam optimizer using a batch size of 200 and variable learning rate as can be found in Table 5.1.

Table 5.1: Learning Rates used in Adam Optimizer for training the BNN.

Epochs	1-59	60-79	80-99	100-120	121-179	180-199	200-219	220-240
LR	1e-3	1e-5	1e-7	1e-9	1e-5	1e-7	1e-9	1e-11

After every epoch, the new network was converted to a B-SNN and its accuracy was tested. Over the whole training period the B-SNN with the highest accuracy was saved, giving a final accuracy of 97.64%.

Possible Optimizations: As the benchmarking application was not the focus of this Thesis, no further optimizations were done. However, it is very likely that a higher accuracy can be achieved with, among others, the following optimizations:

- Hyperparameter Tuning (such as Network Architecture, Surrogate Gradient function etc.)
- Batch Normalization
- Training for more epochs and/or re-training the network many times
- Data augmentation [53]

Additionally, to improve the Energy efficiency of the system, sparsification, or regularization [32], could be performed. This means training the network as normal, but adding a measure of the number of ‘1’s in the network activations to the Loss function. In doing so, the optimizer can co-optimize for less spikes in the network, increasing sparsity. Due to the event-driven nature of the proposed system this will reduce the overall Energy consumption and possibly also the throughput.

5.1.3. System-Level Setup

System-level results were collected for the proposed architecture using five different SRAM cells: the original 6T cell, and the four proposed multiport transposable cells (1P, 2P, 3P, and 4P). This gives five sets of results.

The clock frequency for each of the five designs was found from the longest pipeline stage inside a Tile. There are two stages; the Arbiter stage and the SRAM Read + Neuron stage. From the critical path measurements of the Arbiters as well as the Read Delay measurements of the SRAM Macros the minimum time for each stage is found. To these minimal times some slack was added to account for

process and environmental variations. Specifically, $300ps$ was added to the Arbiter stage, and $400ps$ was added to the SRAM+Neuron stage. The latter is slightly longer to include additional hold time for the neurons to sample the SRAM output.

Overall system-level results are found by simulating the network spike-by-spike in Python. For every subcomponent, the number and type of uses is recorded, and an average is taken over the 10,000 MNIST test images. The type of use relates mainly to the level of utilization of a component. For example, a 4-port SRAM Macro can process four spikes per clock cycle, but if the Arbiter selects less than four spikes, the SRAM Energy consumption is different. Another example of different use types is the Arbiter; after a Tile Timestep, when input vector R has been emptied, a new vector R of '1's and '0's is presented to the Arbiter. This means many of its inputs change in one clock cycle, giving a much larger Energy consumption than in a normal clock cycle. These different use types are all measured and used to calculate the system-level results as accurately as possible.

The system can take in a new MNIST image every Tile Timestep, which determines the overall throughput as explained in Section 4.1.4. To find the throughput, for every MNIST image forward pass, for every Tile i, the number of clock cycles necessary to process R_i is recorded during simulation. Then, for every image the maximum R_i is determined, and this is taken as the length of the Tile Timestep for that image. The average Tile Timestep length is then computed over all 10,000 MNIST test images, from which it can be computed how many clock cycles on average are needed per MNIST inference. From this and the clock frequency the throughput is calculated.

Additionally, system-level area estimates are made. For the SRAM Macros, the area of the cell array is found from the areas of the cells and the area of the periphery is found from the standard cell sizes in the *imec* 3nm node. The areas of the cells are found from the area of the standard 6T SRAM cell [43], the layouts shown in Figure 4.5, and the design rules of the technology node, as can be found in Appendix A.1. For the Arbiters and Neuron Arrays, the areas reported by the synthesis tool are used. The overall area estimate is then made based on the number of occurrences of each component and their respective sizes. It is important to note that this does not include area overhead for the interconnect *between* components, and thus does not represent an accurate enough result to compare to other implementations outside this Thesis. However, it does allow a comparison between the five variants of the proposed system.

5.2. SRAM Macro

In this Section, the circuit-level results for the SRAM Macro are presented. The results will mainly be presented in graph form; the Tables with full results can be found in Appendix C. The results presented for full SRAM Macros will all relate to the 128×128 Macro, as it is by far the main contributor to the Energy consumption, as well as the limiting factor for the clock frequency. However, the results for the 128×10 Macro, as used in Tile 4, can be found in Appendix C.

The Section will start with an evaluation of the SRAM cell areas, followed by the results of the parasitics extraction for the SRAM cells. Next, the NBL Assistance measurements are shown, explaining the array size limit to 128×128. Then, the Energy and Timing results for the Transposable Read and Write operations of the full Macro are shown. Finally, the results for the multiport Inference Read operation of the full Macro are presented.

5.2.1. Cell Area Evaluation

Table 5.2 shows the areas of the SRAM cells, where for the dual cells (1P and 2P) the average area of the two halves is taken. As the Table shows, adding 1 port increased the area by 50% of the original 6T cell area; every additional port increases the area by 37.5% of the 6T cell. Due to the metal layers M1 and M3 being exactly filled up for the 4P cell, adding just one more port to create the 5P cell would instead increase the area by 87.5% of the 6T cell compared to the 4P cell. This was deemed an excessive increase in area for the benefit it provides. Later in this Section, the Multiport Inference Read results will confirm that adding more ports is not beneficial due to the excessive parasitics induced.

5.2.2. Parasitics Extraction

To accurately evaluate the SRAM Macro, more detailed parasitics are needed than the ones built into the existing Transistor models of the technology library. Specifically, the Resistance and Capacitance parasitics on the Bitlines and Wordlines need to be extracted and added to the SRAM cell schematics

Table 5.2: Absolute and relative areas of the presented SRAM cells

	6T	1P	2P	3P	4P
Area [μm^2]	0.01512	0.02268	0.02835	0.03402	0.03969
Relative Area	$1\times$	$1.5\times$	$1.875\times$	$2.25\times$	$2.625\times$

manually. Note that for the dual cells (the 1P and 2P cells) a set of parasitics is extracted for both halves of the cell. The highest of the two values is chosen as the parasitic Resistance or Capacitance to report and to simulate with.

As these are intermediate results, with only indirect effect on the system-level results, they are not presented here. However, the values found for the Bitline and Wordline Resistances and Capacitances can be found in Appendix C.1.

5.2.3. Negative Bitline Voltage Assistance

As explained in Section 2.4.4, the Negative Bitline Voltage Assistance technique is applied to perform successful write operations to the cells. As also mentioned in Section 2.4.4, if it is necessary for the assistance voltage V_{WD} to be lower than $-400mV$, a low yield is expected for the SRAM and the array should be considered invalid.

To determine which SRAM arrays would yield valid results, measurements were performed simulating only a single Row and Column of an array, measuring the necessary V_{WD} to write to the worst-case cell according to the method described in [36, 54]. In short, a Write operation is attempted, and the V_{WD} is gradually lowered until the Write operation is successful. The V_{WD} at which this happens is taken as the necessary voltage. Simulating only a Row and Column allowed for quicker validation of the array sizes. All arrays marked as valid from these measurements are checked again during the full array measurements later in this Chapter.

The measurements were done for the following array sizes: 256×256, 256×128, 128×128, 256×10 and 128×10. The results can be found in Figure 5.1, along with the limit of $V_{WD} = -400mV$. The same results can be found in Appendix C.2. As the Figure shows, a 256×256 array is not valid for any cell. The 6T cell gives a valid result for the 256×128 and 256×10 arrays, but all the other cells do not. The only sizes where all cells pass the test are 128×128 and 128×10. Further circuit-level and system-level results are therefore only collected for these two array sizes.

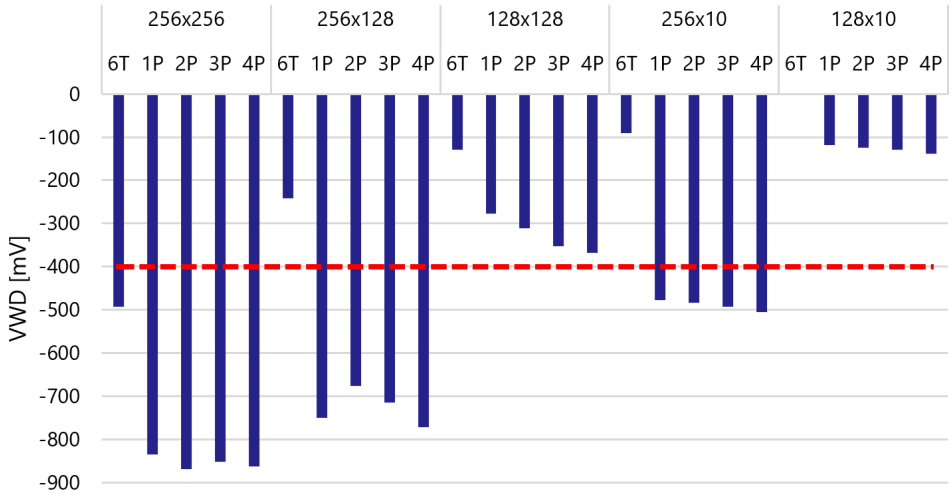


Figure 5.1: Results showing the necessary V_{WD} for a successful Write operation. Added is the limiting line of $-400mV$.

5.2.4. Transposable Read and Write

Figure 5.2 shows the Time and Energy measurements for Writing to and Reading from the worst-case cell in a 128×128 array using the Transposed port (i.e. the WL and BL/BLB). The results can also be found in Appendix C.3, along with the full array measurements of V_{WD}, which confirm that all arrays

maintain a $V_{WD} > -400mV$.

The Write Time is the time between the start of the Write process and the cell content flipping to 90% of its intended value. The Read Time is the delay between the Read process starting and the data output of the Sense Amplifier flipping to the correct value. Write Energy is the Energy consumed by the SRAM Macro during the Write Time period, and Read Energy is the Energy consumed by the Macro during a full clock cycle, which includes precharging of the BL/BLB [54], as this also significantly contributes to the Energy consumption.

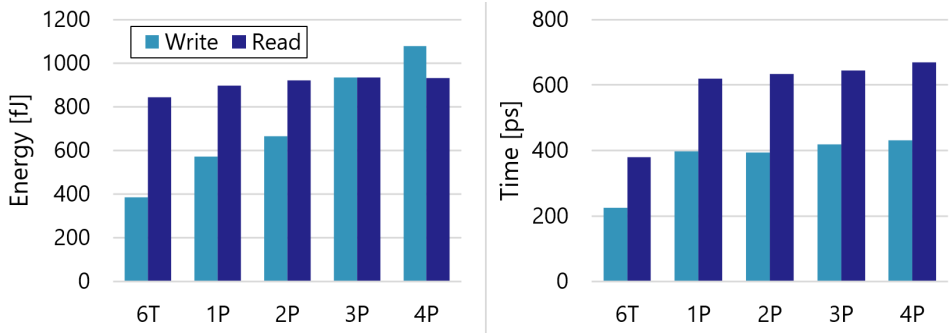


Figure 5.2: Write and Read Energy and Time for a 256×256 SRAM array via the Transposed port for all tested SRAM cells.

As expected, both the Write and Read operation results scale with the addition of more Inference ports due to the parasitics these ports introduce. The effect is stronger for the Write operation, as the parasitics also cause a lower required V_{WD} when more ports are added, increasing the voltage differential and thereby the power consumption. It is also worth noting that when just one Inference Port is added (going from 6T to 1P), there is immediately a significant increase in Write and Read times via the transposed port. This is because the WL wire in each of the proposed cells is narrower and therefore more resistive than in the original 6T cell, as can be seen in Figure 4.5. This is made necessary by the addition of BL0-BL3 that have to be routed in the same metal layers. The higher resistance causes WL to charge up more slowly.

5.2.5. Multiport Inference Read Operation

For the Inference Read operation using the decoupled multiport ports, many different measurements had to be taken due to the amount of variables. First, measurements are taken for the different numbers of ports; one up to four, and for the different array sizes; 128×128 and 128×10. As explained in Section 4.2, due to the fact that the added ports are decoupled from the cell, V_{prech} can be scaled to lower values than VDD. Thus, all measurements are taken at four levels: $V_{prech} = \{700, 600, 500, 400\}mV$ to show the effect of voltage scaling over a reasonable range. Finally, the number of Read operations is varied. As explained in Section 4.3, the Arbiter is not always able to serve p spikes for a p-port SRAM. Thus, for every number of ports p, measurements are taken for x Read operations, where $1 \leq x \leq p$. The full set of results can be found in Appendix C.4.

Precharge Voltage Scaling: Before inspecting the final results, first the trade-off resulting from the choice of V_{prech} must be discussed. Figure 5.3 shows this trade-off from the results for a 128×128 array. Shown are the average access energy per port and average access time per port. This means the values reported are normalized to the number of ports, meaning access Energy and Time for a p-port SRAM are divided by p. Additionally, only results for full port utilization are shown, meaning for a p-port design, p Read operations are done.

As expected, lowering V_{prech} increases the overall access Time, but also lowers the Energy consumption. Lowering V_{prech} from $700mV$ to $500mV$ results in at least 43% reduction in Energy consumption for at most 19% higher access time. Lowering V_{prech} further from $500mV$ to $400mV$ saves at most 10% more Energy for the 1- and 2-port designs, but for the 3- and 4-port designs the Energy consumption actually increases. This is due to the significantly slower precharging. Therefore, for the rest of this Thesis, the precharge voltage for the Inference ports is taken as $V_{prech} = 500mV$ as an optimal point in the trade-off. Note that further searching for a more optimal V_{prech} could be done, but this is deemed outside the scope of this Thesis.

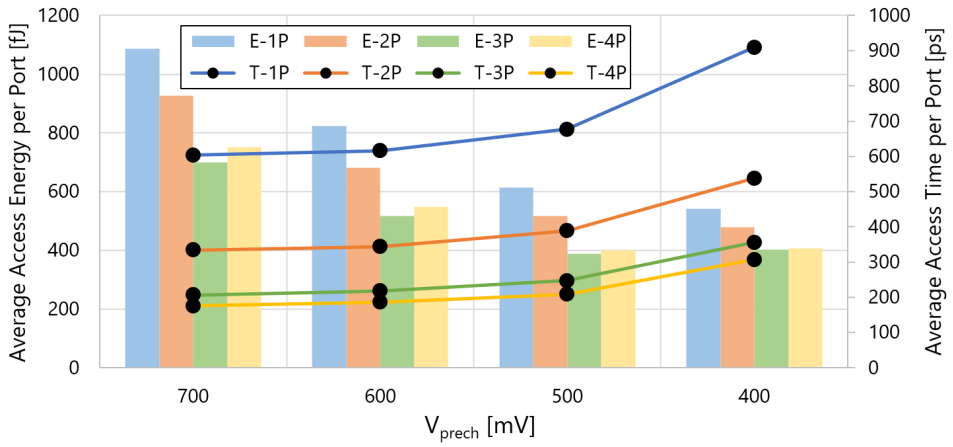


Figure 5.3: Average access Energy and Time per port for a 128×128 array for different levels of V_{prech}.

Results for $V_{prech} = 500mV$: Figure 5.4 shows the average access energy per port and average access time per port for the four multiport cells just when using $V_{prech} = 500mV$. Again, results are normalized for the number of ports and full port utilization is assumed. This is a subset of the results presented in Figure 5.3.

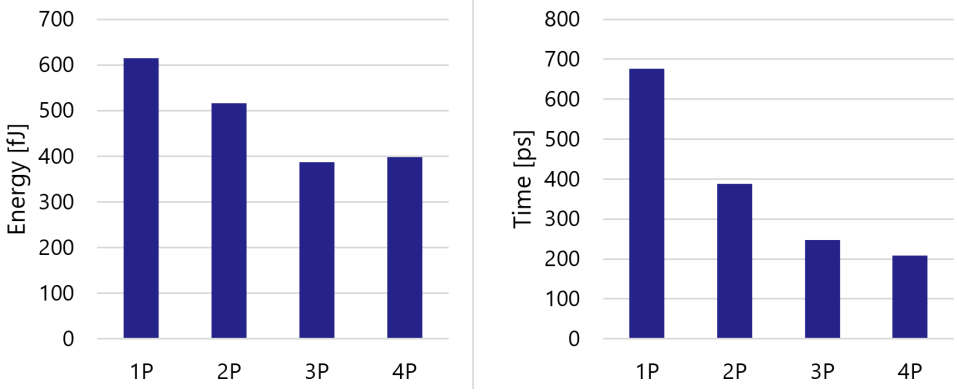


Figure 5.4: Inference Read average Access Time and Energy normalized to number of ports for $V_{prech} = 500mV$ and a 128×128 array.

The average access Time reduces as expected with the number of multiport ports. However, there are clearly diminishing returns in the access Time when increasing the number of ports, where the increases in parasitics start to outweigh the benefits of increased parallelism. For the Energy consumption this effect is even stronger, where the 4-port cell actually consumes more Energy per Read than the 3-port cell, despite the higher degree of parallelism. This shows that increasing the number of ports to 5+ would indeed lead to little to no return, or possibly even lowered performance, as was hypothesized based on the cell layouts in Section 4.2.

5.3. Arbiter

Table 5.3 shows the results for the Arbiter component, including the pipelining registers that store G. The Area, Leakage Power, and Critical Path figures come directly from Genus. More complex are the E_{avg} and E_{max}. The Energy consumption of the Arbiter in a clock cycle differs based on how its inputs change. The more inputs change, the more Energy it will typically consume, as more logic gates have changing inputs. To account for this, two types of changes in inputs are measured:

- E_{avg}: All inputs R are '0' except for the top p, which are '1'. These p requests are then served, meaning all inputs R become '0', which propagates all the way to the noR output.
- E_{max}: All inputs R are '0' (all requests have just been served), then all inputs R are made '1' (the

next Tile Timestep starts and all neurons in the previous Tile now want to send a spike).

E_{avg} represents the typical cycle, where the Arbiter picks p inputs from its request vector R. E_{max} represents the arrival of a completely new request vector R. Both E_{avg} and E_{max} are overestimations of these situations. This is so that the system-level results will err on the high side in terms of Energy.

Table 5.3: Arbiter results from synthesis and simulation of synthesized design. Reported for a 1 up to 4-port Arbiter.

Ports	Area [μm^2]	P_{Leak} [μW]	Critical Path [ps]	E_{avg} [fJ]	E_{max} [fJ]
1	20.12	1.69	706	66.5	90.4
2	44.18	3.75	734	137.5	213.7
3	67.30	5.79	740	207.1	340.8
4	90.20	7.72	707	273.2	455.1

The results in Table 5.3 are mostly as expected; the multiport Arbiter variants are made by duplicating the 1-port Arbiter, so the overall Area, Power, and Energy scale with the number of ports. The only thing that stands out is the fact that the Critical Path remains almost constant. This is due to the synthesis tool optimizing for a critical path below $750ps$ in order to keep the Arbiter times shorter than the SRAM Read times. As explained in Section 5.1, the Arbiter and SRAM + Neuron are pipelined, and since the SRAM Read times are much more difficult to improve, they determine the maximum clock frequency. By ensuring all Arbiter times are below this Read Time, the Arbiter is as optimal in timing as it needs to be. This comes at the cost of some extra area, but since the Arbiter is relatively small compared to the other components, this is not a problem.

5.4. Neuron

5.4.1. Bit Width Choices

There are two main parameters to decide for the final Neuron implementation: the number of bits m used to store V_{mem} and the number of bits t used to store V_{th}. These values are chosen based on running MNIST inference with the previously described B-SNN.

Figure 5.5 shows the distribution of all $3 \times 256 = 768$ V_{th} values to be stored. The minimum and maximum V_{th} are -28 and 24 respectively. Also shown are two sets of lines; the red lines show the range of a signed 5-bit integer, the green lines of a signed 6-bit integer. Choosing $t = 6$ would cover all V_{th}, even the largest outliers. $t = 5$ means a not insignificant portion of the distribution would not fit. This means quite a lot of nuance in the comparison between V_{mem} and V_{th} could be lost. Therefore, $t = 6$ was chosen.

Figure 5.6 shows the distribution of all *final* values of V_{mem}, meaning the value V_{mem} has reached after all the input spikes to a tile (R_{i-1}) have been processed, i.e. at the end of a Tile Timestep. Data was collected for all layers and all 10,000 MNIST test images. This yields 7,780,000 data points, which should cover almost any possible value V_{mem} could attain, even in the middle of a Tile Timestep. The minimum and maximum measured values are -124 and 261 respectively. Also shown are two sets of lines; the red lines show the range of a signed 7-bit integer, the green lines of a signed 8-bit integer. A 7-bit integer would cover most of the main distribution, but miss out on the outer edges, as well as any outliers. An 8-bit integer ensures that the main distribution is easily covered, as well as most of the outliers. Therefore $m = 8$ was chosen to err on the safe side. Note that missing out on the most extreme outliers is not a big problem; V_{th} covers a smaller range so any extreme outlier will give the same result for the V_{mem} with V_{th} comparison, regardless of whether it is stored as its actual value or the edge values of V_{mem} (i.e. $[-128, 127]$).

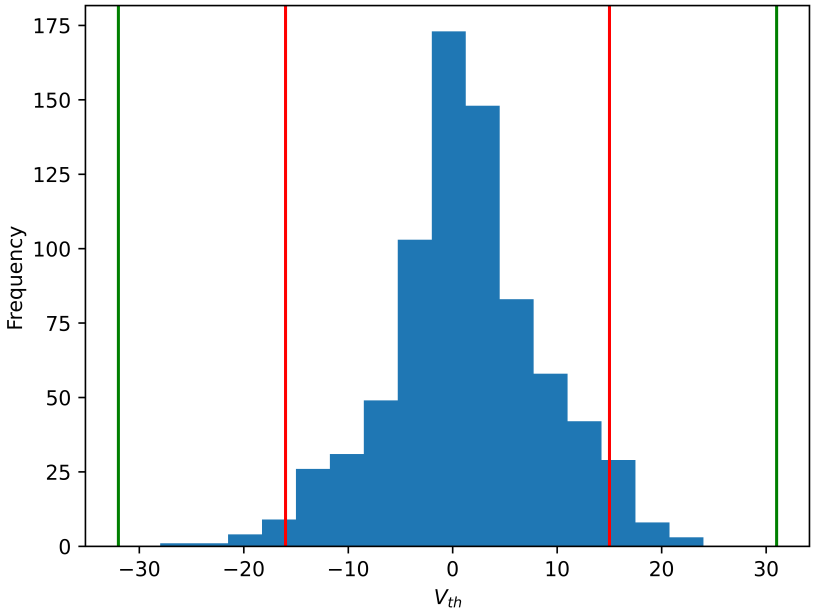


Figure 5.5: Distribution of all V_{th} values to be stored when running MNIST inference on the proposed B-SNN. Red line indicates signed integer range for 5 bits, green line indicates signed integer range for 6 bits.

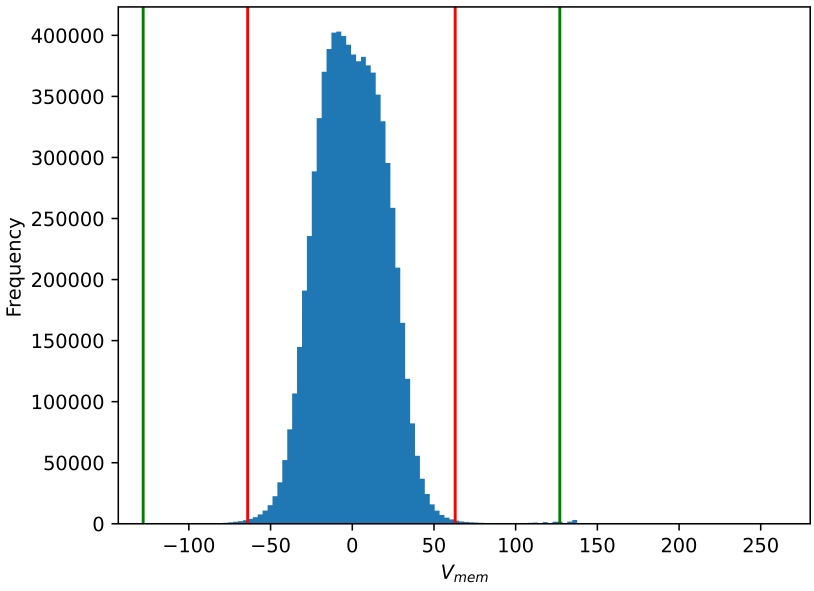


Figure 5.6: Distribution of all V_{mem} values after a when running MNIST inference on the proposed B-SNN. Red line indicates signed integer range for 7 bits, green line indicates signed integer range for 8 bits.

5.4.2. Measurements

Table 5.4 shows the results for a 128-wide Neuron Array. Each neuron has an 8-bit register to store V_{mem} ($m = 8$) and a 6-bit register to store V_{th} ($t = 6$). Based on which Tile the neuron is in, and how many access ports the SRAM Macros on that Tile have, the neuron can have many different numbers of input ports. Thus, results are reported for all possible numbers of ports in the architecture. The Area and Leakage Power come directly from Genus. The critical path is very short for each, so it is not reported here; it is taken into account by the slack for the SRAM Read time. More complex are the different Energy numbers reported:

- E_{Vth} is the Energy to change the latch register holding V_{th}.
- E_{avg} is the average clock cycle Energy when the neuron is accumulating V_{mem}, where the average is taken over a Neuron when: half of its inputs are '1', all inputs are '1', and all inputs are '0', to cover the full range of possibilities.
- E_{show} is the Energy in the cycle where R has been emptied, indicated by control signal *R_empty* and the neuron changes its output r to indicate whether $V_{mem} \geq V_{th}$.
- E_{grant} is the Energy in a cycle when the neuron has a spike pending at its output r, which is granted by the Arbiter of the following Tile, meaning r is reset to '0'.

Table 5.4: Neuron results from synthesis and simulation of synthesized design. Reported for all possible numbers of input ports for the proposed architecture.

Ports	Area [μm^2]	P_{Leak} [μW]	E_{Vth} [pJ]	E_{avg} [pJ]	E_{show} [pJ]	E_{grant} [pJ]
1	744.57	74.97	0.234	3.478	1.666	1.631
2	759.42	75.86	0.234	2.664	1.517	1.624
3	829.05	84.90	0.235	4.599	1.698	1.631
4	829.05	81.89	0.235	3.397	1.524	1.627
6	1008.25	98.41	0.235	5.621	1.546	1.628
8	1042.81	101.21	0.235	5.862	1.440	1.609
12	1355.64	129.10	0.235	6.054	1.502	1.689
18	1702.52	155.53	0.235	9.120	1.535	1.702
24	2050.68	186.32	0.236	12.123	1.560	1.713

Again, the Neuron Array Area and Leakage Power scale with the number of input ports. This is expected, as the Decode & Add block becomes larger. E_{Vth} remains almost constant for the number of input ports. This is also expected; the width of this register remains unchanged. Similarly, E_{grant} and E_{show} do not really scale with the number of ports, as these operations are almost identical regardless of how many inputs there are.

5.5. System-Level

In this Section, the system-level simulation results of the proposed architecture are presented. Results are shown for five versions of the design; using the 6T, 1P, 2P, 3P, and 4P cells and their associated Arbiters and Neurons. First, a Timing evaluation is done to determine the clock frequency for the designs. Next, an Area evaluation is done, comparing the five designs with each other. Then, the Online Learning performance is evaluated, showing the advantage of the transposable Read/Write access. Finally, the Inference performance of the system is shown.

5.5.1. Timing Evaluation

There are two pipeline stages in each Tile: the Arbiter stage and the SRAM + Neuron stage. The longest of these two stages determines the maximum global clock frequency. For the Arbiter stage, the critical path as reported in Table 5.3 is taken, and $300ps$ slack is added to account for process and environmental variations. For the SRAM + Neuron stage, the SRAM Read time as reported in Appendix C.3 and Appendix C.4 is taken, and $400ps$ slack is added to account for variations, and to add some time for the Neuron to perform its Decode & Add operation and latch the data.

The results are reported in Table 5.5 for all five SRAM cell options; the 6T cell and the 1P, 2P, 3P, 4P cells. For each cell, the longest of all the possible Read times is taken. The Arbiter stage and the

SRAM + Neuron stage times are shown. The longest of the two stages is highlighted to indicate that its duration determines the clock period. In the final row of the Table the clock frequency is reported, based on the highlighted clock period. These frequencies will be used for further system-level results.

Table 5.5: Required time for each pipeline stage for the five versions of the architecture. Highlighted is the longest of the two stages, indicating the clock period. Also reported is the clock frequency following from that clock period.

	6T	**1P**	**2P**	**3P**	**4P**
Arbiter Stage [ns]	***1.007***	1.007	1.040	1.034	1.006
SRAM + Neuron Stage [ns]	0.685	***1.077***	***1.176***	***1.141***	***1.234***
f_{clk} **[MHz]**	993	929	850	876	810.3

5.5.2. Area Evaluation

In this Subsection an evaluation of the Area of the full system is reported. Note that this is an estimate; it does not include interconnects or global control circuitry. The Arbiter and Neuron areas are taken from synthesis results; the SRAM Macro areas are calculated based on the array size and the sizes of the utilized standard cells. Figure 5.7 and Table 5.6 show the total area estimations and their breakdown over the three main components.

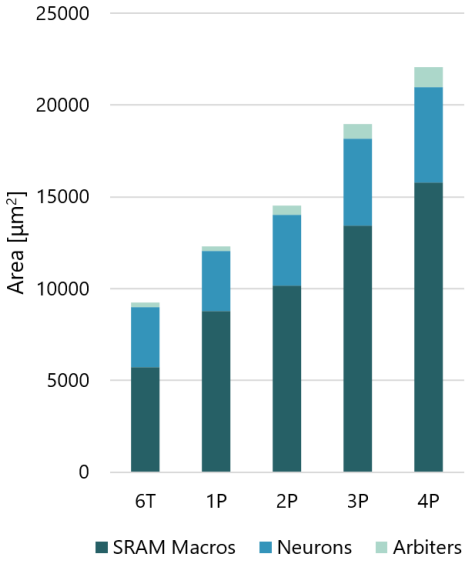


Figure 5.7: Area estimates for the five versions of the architecture.

Table 5.6: Total Area estimates for the five versions of the architecture, along with a breakdown over the three main components.

	Area [μm^2]	**SRAM Macros**	**Neurons**	**Arbiters**
6T	9244	61.8%	35.5%	2.6%
1P	12286	71.3%	26.8%	2.0%
2P	14546	69.9%	26.4%	3.6%
3P	18983	70.8%	24.9%	4.3%
4P	22050	71.6%	23.5%	4.9%

The overall area scales close to linearly with the number of added ports. The more ports added, the larger the portion of the area taken up by the Arbiters. This makes sense, as the Arbiter scales linearly with the number of ports, while the SRAM and Neuron sizes do not.

5.5.3. Online Learning

Online Learning becomes significantly more efficient with transposable access ports. Without the transposable port, for an array of size 128×128, it would take $2 \times 128 = 256$ clock cycles to read and write all the weights before a post-synaptic neuron. For the proposed transposable SRAM cells, reading and writing all these weights takes just $2 \times 4 = 8$ cycles. The factor 4 comes from the use of 4-to-1 MUXs, meaning only $1/4$ of a column can be accessed at once. The resulting total Energy and Time to update the weights of a single column in a 128×128 SRAM Macro is shown in Figure 5.8, derived from the Read/Write figures reported in Appendix C.3 and the clock periods reported in Table 5.5.

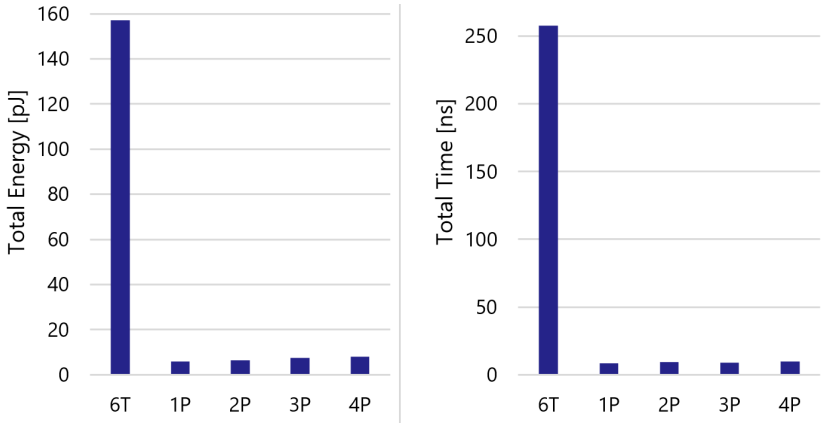


Figure 5.8: Energy and Time consumed to Read and Write all weights in a 128×128 SRAM Macro for the five cells discussed.

As the Figure shows, the efficiency of Online Learning is improved dramatically. The worst performer of the transposable cells is the 4P cell, but it still requires $19.5\times$ less Energy $26.0\times$ less Time to Read and Write all the column weights than the 6T cell.

5.5.4. Inference

Figure 5.9 shows the system-level results from running MNIST digit classification for the 10,000 test images using the five architecture variants. As stated before, the achieved accuracy on the test images is 97.64%.

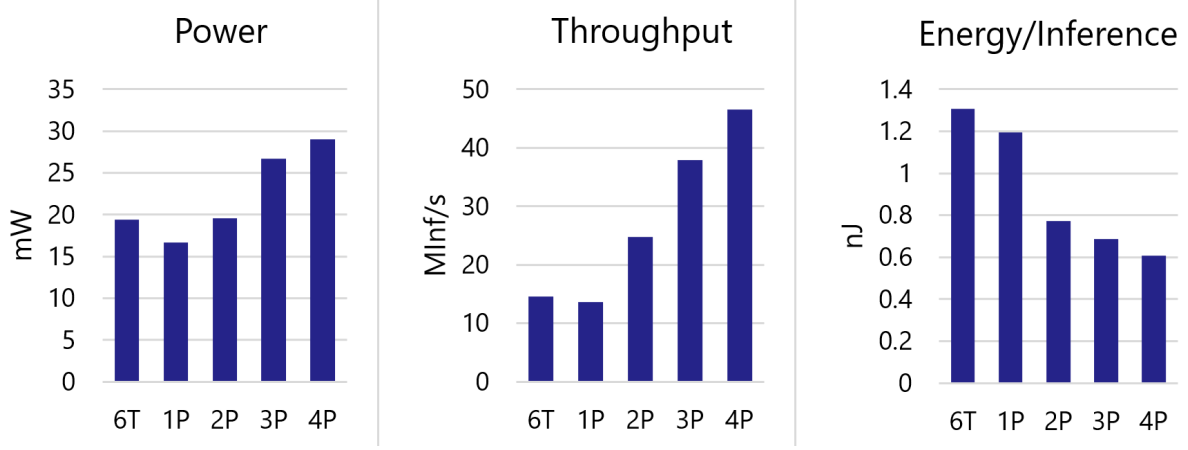


Figure 5.9: System-Level comparison between the five architecture variants, comparing Power, Throughput, and Energy/Inference.

Generally, the Inference Power consumption of the system increases with the number of added ports. However, the system Power when implemented with 6T cells is actually higher than that of the 1P and 2P cells. This is due to the active power savings from the voltage scaling of V_{prech} of the decoupled read ports. This voltage scaling is not possible for the 6T cell.

The throughput of the 1P system is slightly lower than the 6T system. This is to be expected; they

have the same level of parallelism, but the 6T system allows a slightly higher clock frequency. This is because the Read operation of the 6T cell array is faster than of the 1P decoupled port. However, at 2+ ports, the added parallelism easily compensates for this, and the throughput using 4P cells is $3.2\times$ higher than for 6T cells.

Finally, even though the overall Power consumption increases when adding ports, the increased throughput makes it so that the Energy/Inference reduces significantly. As such, the 4P-based system has the best Energy/Inference performance, improving by $2.2\times$ with respect to the 6T cell.

5.6. Comparison to State-of-the-Art

Table 5.7 shows a comparison between the proposed architecture using the 4P SRAM cell and state-of-the-art low parameter resolution SNN accelerators aimed at low-power applications.

The proposed architecture implements a similar number of neurons and synapses, but is unique in the field in that it uses both Binary weights and activations. Other works either used rate coding [7] to increase the activation bit width, or multi-bit weights [10, 11].

All systems are benchmarked on the MNIST digit classification data set. The proposed architecture runs at comparable Power to [10, 11], but two orders of magnitude higher than [7]. This can be explained by the fact that [7] is designed for extremely low-Power operation, running at just $70kHz$ and performing only 2 inferences per second. Though the low Power is certainly impressive, performing just 2 inferences per second is insufficient for the system to be useful in a realistic application; a typical sensor system will produce orders of magnitude more data.

Where the proposed system really stands out is in its throughput, measured in number of MNIST inferences performed per second, and the Energy per Synaptic Operation (Energy/SOP). By leveraging an extremely simple communication fabric (similar to [7]), a single clock cycle arbiter, and fully Binary weights, the throughput is increased by multiple orders of magnitude while the Energy/SOP is reduced by multiple orders of magnitude. As a result of these two significant improvements, the Energy/Inference is much lower than state-of-the-art solutions.

Table 5.7: Comparison between Proposed Architecture using the 4P cell and State-of-the-Art small-scale SNN Accelerators.

	[7]	**[10]**	**[11]**	***This Work***
Technology [nm]	65	10	65	3
Neuron Count	650	4096	1K	778
Synapse Count	67K	1M	256K	330K
Activation Bit Width	6	1	–***	1
Weight Bit Width	1	7	5	1
Transposable	No	No	Yes	Yes
Clock Frequency	70kHz	506MHz	100MHz	810MHz
MNIST				
Power	305nW	196mW*	53mW	29.0mW
Accuracy [%]	97.6	97.9	97.2	97.6
Throughput [inf/s]	2	6250	20	44M
Energy/Inference [nJ]	195	1000	–	0.607
Energy/SOP**	1.5pJ	3.8pJ	15.2pJ	3.2fJ

* Inferred from SOP/s/mm^2, Area, and pJ/SOP

**SOP: Synaptic Operations

*** 50 time steps per inference; Poisson spike train activations

6

Conclusion

This Chapter concludes the Thesis. First, a summary and general conclusions are presented, after which potential avenues for future work are discussed.

6.1. Conclusions

In this Thesis, an SRAM-based Compute In-Memory (CIM) accelerator designed for Binary Spiking Neural Networks (B-SNN) in 3nm FinFET was presented. The target use case for the accelerator was ultra-low Power Edge devices, meaning the aim was to minimize Energy consumption while maintaining classification accuracy and sufficient throughput to keep up with live sensor data. Since Spiking Neural Networks lend themselves very well to Online or On-Chip learning, an additional goal of increasing Online learning efficiency was set.

To accelerate a Neural Network using CIM, the main operation to target is the MAC operation. In this work, first the three main methods of performing the MAC operation in SRAM were discussed: systolic arrays, adder trees, and sequential accumulation. Due to the minimal hardware overhead and event-driven nature of the latter solution, it was chosen as the basis for the proposed architecture. The problem in existing works utilizing this solution was low throughput caused by low parallelism, complex spike arbitration, and superfluous activation or weight bit precision. This work tackled all three of these problems by introducing four multiport SRAM cell designs, allowing for anywhere between one and four parallel Read operations, a fully logic-based spike Arbiter, able to perform multiport spike arbitration in a single clock cycle, and finally a fully Binary SNN, minimizing MAC operation complexity. The proposed architecture was tested by running MNIST digit classification, and achieved 97.6% accuracy while running at $29.0mW$. It achieved a throughput of $44M$ MNIST images per second, consuming just $607pJ$ of Energy per Inference. Compared to state-of-the-art solutions, accuracy was maintained, Power was lowered slightly, and throughput and Energy per Inference are improved by multiple orders of magnitude.

For on-chip learning, synaptic weights in the memory need to be accessed on a per-column basis, while for inference, weights need to be accessed on a per-row basis. SRAM cells with access in both directions are called transposable. To design transposable SRAM cells, various methods were compared and discussed based on cell stability, Read Power, area overhead, and Read speed. In the end, a cell was proposed which maintains the original SRAM access ports for transposable access, and adds between one and four decoupled Read ports for row-wise Inference access. These decoupled ports have the added benefit of enabling local voltage scaling to save Power. As a result, updating a column of weights requires $26.0\times$ less time and $19.5\times$ less Energy than when utilizing a standard SRAM cell.

6.2. Future Work

The following are the main avenues of potential future work following from this Thesis:

- **Generalizing the Architecture:** The current design is custom-configured for MNIST classification. To increase its usefulness, the architecture should be made more general-purpose. This mainly means increasing the sizes of some layers, and making the final neuron layer programmable to allow for different decision outputs than just 10-digit classification.
- **Real-Time Data Classification:** The design is tested on time-independent data, but its realistic use case is real-time classification or recognition of sensor data on an Edge device. Thus, the architecture should be adjusted and tested to see how well it performs when data is supplied as a constant stream instead of in batches.
- **Hardware optimizations**: The following are the main methods of optimizing the hardware:
  - Global Voltage Scaling;
  - Clock Gating;
  - Power Gating;
  - Automatic Per-Tile Frequency Adjustment (see [7]).
- **B-SNN optimizations:** The following are the main avenues for improving the B-SNN performance:
  - *Improved accuracy:* To improve the accuracy of the network, various methods can be applied, such as hyperparameter tuning, batch normalization (requiring a slight adjustment to the neuron hardware), longer training times, and Data augmentation.
  - *Sparsification:* by training the network with spike sparsity as part of the Loss function, overall Energy consumption and timing can be improved due to the event-driven nature of the architecture.
- **Tapeout:** To compare the proposed architecture more fairly with state-of-the-art solutions, it should be implemented as a physical chip and taped out to perform real-world measurements.

References

[1] Yen-Lin Lee, Pei-Kuei Tsung, and Max Wu. “Techology trend of edge AI”. In: *2018 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. 2018, pp. 1–2. DOI: 10.1109/VLSI-DAT.2018.8373244.

[2] Yu-Der Chih et al. “16.4 An 89TOPS/W and 16.3TOPS/mm2 All-Digital SRAM-Based Full-Precision Compute-In Memory Macro in 22nm for Machine-Learning Edge Applications”. In: *2021 IEEE International Solid- State Circuits Conference (ISSCC)*. Vol. 64. 2021, pp. 252–254. DOI: 10.1109/ISSCC42613.2021.9365766.

[3] Sumon Kumar Bose, Jyotibdha Acharya, and Arindam Basu. *Is my Neural Network Neuromorphic? Taxonomy, Recent Trends and Future Directions in Neuromorphic Engineering*. 2020. arXiv: 2002.11945 [cs.ET].

[4] Hidehiro Fujiwara et al. “A 5-nm 254-TOPS/W 221-TOPS/mm2 Fully-Digital Computing-in-Memory Macro Supporting Wide-Range Dynamic-Voltage-Frequency Scaling and Simultaneous MAC and Write Operations”. In: *2022 IEEE International Solid- State Circuits Conference (ISSCC)*. Vol. 65. 2022, pp. 1–3. DOI: 10.1109/ISSCC42614.2022.9731754.

[5] Dengfeng Wanq et al. “All-Digital Full-Precision In-SRAM Computing with Reduction Tree for Energy-Efficient MAC Operations”. In: *2022 IEEE International Conference on Integrated Circuits, Technologies and Applications (ICTA)*. 2022, pp. 150–151. DOI: 10.1109/ICTA56932.2022.9963042.

[6] Dewei Wang et al. “DIMC: 2219TOPS/W 2569F2/b Digital In-Memory Computing Macro in 28nm Based on Approximate Arithmetic Hardware”. In: *2022 IEEE International Solid- State Circuits Conference (ISSCC)*. Vol. 65. 2022, pp. 266–268. DOI: 10.1109/ISSCC42614.2022.9731659.

[7] Dewei Wang et al. “Always-On, Sub-300-nW, Event-Driven Spiking Neural Network based on Spike-Driven Clock-Generation and Clock- and Power-Gating for an Ultra-Low-Power Intelligent Device”. In: Nov. 2020, pp. 1–4. DOI: 10.1109/A-SSCC48613.2020.9336139.

[8] Jae-sun Seo et al. “A 45nm CMOS neuromorphic chip with a scalable architecture for learning in networks of spiking neurons”. In: Sept. 2011, pp. 1–4. DOI: 10.1109/CICC.2011.6055293.

[9] Daehyun Kim et al. “MONETA: A Processing-In-Memory-Based Hardware Platform for the Hybrid Convolutional Spiking Neural Network With Online Learning”. In: *Frontiers in Neuroscience* 16 (Apr. 2022). DOI: 10.3389/fnins.2022.775457.

[10] Gregory K. Chen et al. “A 4096-Neuron 1M-Synapse 3.8-pJ/SOP Spiking Neural Network With On-Chip STDP Learning and Sparse Weights in 10-nm FinFET CMOS”. In: *IEEE Journal of Solid-State Circuits* 54.4 (2019), pp. 992–1002. DOI: 10.1109/JSSC.2018.2884901.

[11] Jinseok Kim et al. “Efficient Synapse Memory Structure for Reconfigurable Digital Neuromorphic Hardware”. In: *Frontiers in Neuroscience* 12 (2018). ISSN: 1662-453X. DOI: 10.3389/fnins.2018.00829. URL: https://www.frontiersin.org/articles/10.3389/fnins.2018.00829.

[12] Jingcheng Wang et al. “14.2 A Compute SRAM with Bit-Serial Integer/Floating-Point Operations for Programmable In-Memory Vector Acceleration”. In: *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*. 2019, pp. 224–226. DOI: 10.1109/ISSCC.2019.8662419.

[13] Hongwu Jiang et al. “CIMAT: A Compute-In-Memory Architecture for On-chip Training Based on Transpose SRAM Arrays”. In: *IEEE Transactions on Computers* 69.7 (2020), pp. 944–954. DOI: 10.1109/TC.2020.2980533.

[14] Sumon Kumar Bose and Arindam Basu. “A 389TOPS/W, 1262fps at 1Meps Region Proposal Integrated Circuit for Neuromorphic Vision Sensors in 65nm CMOS”. In: *2021 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. 2021, pp. 1–3. DOI: 10.1109/A-SSCC53895.2021.9634734.

[15] Zhiting Lin et al. “Two-Direction In-Memory Computing Based on 10T SRAM With Horizontal and Vertical Decoupled Read Ports”. In: *IEEE Journal of Solid-State Circuits* 56.9 (2021), pp. 2832–2844. DOI: 10.1109/JSSC.2021.3061260.

[16] Leland Chang et al. “An 8T-SRAM for Variability Tolerance and Low-Voltage Operation in High-Performance Caches”. In: *IEEE Journal of Solid-State Circuits* 43.4 (2008), pp. 956–963. DOI: 10.1109/JSSC.2007.917509.

[17] Daehyun Kim et al. “Processing-In-Memory-Based On-Chip Learning With Spike-Time-Dependent Plasticity in 65-nm CMOS”. In: *IEEE Solid-State Circuits Letters* 3 (2020), pp. 278–281. DOI: 10.1109/LSSC.2020.3013448.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Neural Information Processing Systems* 25 (Jan. 2012). DOI: 10.1145/3065386.

[19] Chuan-Jia Jhang et al. “Challenges and Trends of SRAM-Based Computing-In-Memory for AI Edge Devices”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 68.5 (2021), pp. 1773–1786. DOI: 10.1109/TCSI.2021.3064189.

[20] Robin M. Schmidt. *Recurrent Neural Networks (RNNs): A gentle Introduction and Overview*. 2019. arXiv: 1912.05911 [cs.LG].

[21] Jürgen Schmidhuber. “Deep learning in neural networks: An overview”. In: *Neural Networks* 61 (2015), pp. 85–117. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2014.09.003. URL: https://www.sciencedirect.com/science/article/pii/S0893608014002135.

[22] Wolfgang Maass. “Networks of spiking neurons: The third generation of neural network models”. In: *Neural Networks* 10.9 (1997), pp. 1659–1671. ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(97)00011-7. URL: https://www.sciencedirect.com/science/article/pii/S0893608097000117.

[23] Hélène Paugam-Moisy and Sander Bohte. “Computing with Spiking Neuron Networks”. In: vol. 1. Jan. 2012, pp. 335–376. ISBN: 978-3-540-92909-3. DOI: 10.1007/978-3-540-92910-9_10.

[24] Yujie Wu et al. “Spatio-Temporal Backpropagation for Training High-Performance Spiking Neural Networks”. In: *Frontiers in Neuroscience* 12 (May 2018). DOI: 10.3389/fnins.2018.00331. URL: https://doi.org/10.3389%2Ffnins.2018.00331.

[25] Wulfram Gerstner, Raphael Ritz, and Leo van Hemmen. “Why spikes? Hebbian learning and retrieval of time-resolved excitation patterns”. In: *Biological Cybernetics* 69 (Oct. 1993), pp. 503–515. DOI: 10.1007/BF00199450.

[26] Timothée Masquelier, Rudy Guyonneau, and Simon Thorpe. “Spike Timing Dependent Plasticity Finds the Start of Repeating Patterns in Continuous Spike Trains”. In: *PloS one* 3 (Feb. 2008), e1377. DOI: 10.1371/journal.pone.0001377.

[27] Dejan Pecevski, Wolfgang Maass, and Robert Legenstein. “Theoretical Analysis of Learning with Reward-Modulated Spike-Timing-Dependent Plasticity”. In: *Advances in Neural Information Processing Systems*. Ed. by J. Platt et al. Vol. 20. Curran Associates, Inc., 2007. URL: https://proceedings.neurips.cc/paper/2007/file/9b698eb3105bd82528f23d0c92dedfc0-Paper.pdf.

[28] Milad Mozafari et al. “First-Spike-Based Visual Categorization Using Reward-Modulated STDP”. In: *IEEE Transactions on Neural Networks and Learning Systems* 29.12 (Dec. 2018), pp. 6178–6190. DOI: 10.1109/tnnls.2018.2826721. URL: https://doi.org/10.1109%2Ftnnls.2018.2826721.

[29] Abhiroop Bhattacharjee et al. “Examining the Robustness of Spiking Neural Networks on Non-ideal Memristive Crossbars”. In: *ACM/IEEE International Symposium on Low Power Electronics and Design*. ACM, Aug. 2022. DOI: 10.1145/3531437.3539729. URL: https://doi.org/10.1145%2F3531437.3539729.

[30] Abhronil Sengupta et al. *Going Deeper in Spiking Neural Networks: VGG and Residual Architectures*. 2019. arXiv: 1802.02627 [cs.CV].

[31] Amirreza Yousefzadeh et al. “On practical issues for stochastic STDP hardware with 1-bit synaptic weights”. In: *Frontiers in neuroscience* 12 (2018), p. 665.

[32] Hyungjun Kim, Hyunmyung Oh, and Jae-Joon Kim. “Energy-efficient XNOR-free In-Memory BNN Accelerator with Input Distribution Regularization”. In: *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 2020, pp. 1–9.

[33] Waqas Gul, Maitham Shams, and Dhamin Al-Khalili. “SRAM Cell Design Challenges in Modern Deep Sub-Micron Technologies: An Overview”. In: *Micromachines* 13 (Aug. 2022), p. 1332. DOI: 10.3390/mi13081332.

[34] T. Kobayashi et al. “A current-controlled latch sense amplifier and a static power-saving input buffer for low-power architecture”. In: *IEEE Journal of Solid-State Circuits* 28.4 (1993), pp. 523–527. DOI: 10.1109/4.210039.

[35] Kamal Y. Kamal. “The Silicon Age: Trends in Semiconductor Devices Industry”. In: *Journal of Engineering Science and Technology Review* 15 (May 2022), pp. 110–115. DOI: 10.25103/jestr.151.14.

[36] Hsiao-Hsuan Liu et al. “Extended Methodology to Determine SRAM Write Margin in Resistance-Dominated Technology Node”. In: *IEEE Transactions on Electron Devices* 69.6 (2022), pp. 3113–3117. DOI: 10.1109/TED.2022.3165738.

[37] Amirreza Yousefzadeh et al. “Hardware implementation of convolutional STDP for on-line visual feature learning”. In: *2017 IEEE international symposium on circuits and systems (ISCAS)*. IEEE. 2017, pp. 1–4.

[38] Amirhossein Rostami et al. “E-prop on SpiNNaker 2: Exploring online learning in spiking RNNs on neuromorphic hardware”. In: *Frontiers in Neuroscience* 16 (2022), p. 1018006.

[39] Shimeng Yu et al. “Compute-in-memory chips for deep learning: Recent trends and prospects”. In: *IEEE circuits and systems magazine* 21.3 (2021), pp. 31–56.

[40] Amirreza Yousefzadeh et al. “SENeCA: Scalable Energy-efficient Neuromorphic Computer Architecture”. In: June 2022. DOI: 10.1109/AICAS54282.2022.9870025.

[41] C. Bagavathi and O. Saraniya. “Chapter 13 - Evolutionary Mapping Techniques for Systolic Computing System”. In: *Deep Learning and Parallel Computing Environment for Bioengineering Systems*. Ed. by Arun Kumar Sangaiah. Academic Press, 2019, pp. 207–223. ISBN: 978-0-12-816718-2. DOI: https://doi.org/10.1016/B978-0-12-816718-2.00020-8. URL: https://www.sciencedirect.com/science/article/pii/B9780128167182000208.

[42] Hyunjoon Kim et al. “Colonnade: A Reconfigurable SRAM-Based Digital Bit-Serial Compute-In-Memory Macro for Processing Neural Networks”. In: *IEEE Journal of Solid-State Circuits* 56.7 (2021), pp. 2221–2233. DOI: 10.1109/JSSC.2021.3061508.

[43] Hsiao-Hsuan Liu et al. “DTCO of sequential and monolithic CFET SRAM”. In: *DTCO and Computational Patterning II*. Ed. by Ryoung-Han Kim and Neal V. Lafferty. Vol. 12495. International Society for Optics and Photonics. SPIE, 2023, 124950Z. DOI: 10.1117/12.2657524. URL: https://doi.org/10.1117/12.2657524.

[44] Adarsha Balaji et al. “Mapping Spiking Neural Networks to Neuromorphic Hardware”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28.1 (2020), pp. 76–86. DOI: 10.1109/TVLSI.2019.2951493.

[45] Mike Davies et al. “Loihi: A Neuromorphic Manycore Processor with On-Chip Learning”. In: *IEEE Micro* 38.1 (2018), pp. 82–99. DOI: 10.1109/MM.2018.112130359.

[46] Filipp Akopyan et al. “TrueNorth: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.10 (2015), pp. 1537–1557. DOI: 10.1109/TCAD.2015.2474396.

[47] Dengfeng Wanq et al. “All-Digital Full-Precision In-SRAM Computing with Reduction Tree for Energy-Efficient MAC Operations”. In: *2022 IEEE International Conference on Integrated Circuits, Technologies and Applications (ICTA)*. 2022, pp. 150–151. DOI: 10.1109/ICTA56932.2022.9963042.

[48] Jongeun Koo et al. “Area-Efficient Transposable 6T SRAM for Fast Online Learning in Neuromorphic Processors”. In: *2019 IEEE Custom Integrated Circuits Conference (CICC)*. 2019, pp. 1–4. DOI: 10.1109/CICC.2019.8780165.

[49] Saber Moradi et al. “A Scalable Multicore Architecture With Heterogeneous Memory Structures for Dynamic Neuromorphic Asynchronous Processors (DYNAPs)”. In: *IEEE Transactions on Biomedical Circuits and Systems* 12.1 (Feb. 2018), pp. 106–122. DOI: 10.1109/tbcas.2017.2759700. URL: https://doi.org/10.1109%2Ftbcas.2017.2759700.

[50] Yann Lecun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[51] Neil H.E. Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective, 3rd Ed.* Addison-Wesley, 2005, pp. 231–235. ISBN: 0-321-14901-7.

[52] Wei Fang et al. *Deep Residual Learning in Spiking Neural Networks*. 2022. arXiv: 2102.04159 [cs.NE].

[53] Connor Shorten and Taghi Khoshgoftaar. “A survey on Image Data Augmentation for Deep Learning”. In: *Journal of Big Data* 6 (July 2019). DOI: 10.1186/s40537-019-0197-0.

[54] Hsiao-Hsuan Liu et al. “CFET SRAM DTCO, Interconnect Guideline, and Benchmark for CMOS Scaling”. In: *IEEE Transactions on Electron Devices* 70.3 (2023), pp. 883–890. DOI: 10.1109/TED.2023.3235701.

A

SRAM Layout

A.1. imec 3nm FinFET SRAM Design Rules

Table A.1: SRAM Design Rules for the *imec* 3nm FinFET technology node.

Parameters	**Value [nm]**
Gate Pitch	45
Gate Length	18
Metal Pitch	21
Gate Cut	18
Gate Extension	9.5
Fin-to-Well Spacing	13.5
Fin-to-Fin Separation	37
Fin Width	5

A.2. 6T SRAM Cell Layout

Figure A.1 shows screengrabs of the 6T SRAM layout in the Virtuoso Layout Editor. Shown are two images; the first shows layers and vias {FIN, GATE, M0A, VINTA, MINT, VINTG}, the second shows layers and vias {MINT, VINTA, VINTG, V0, M1}.

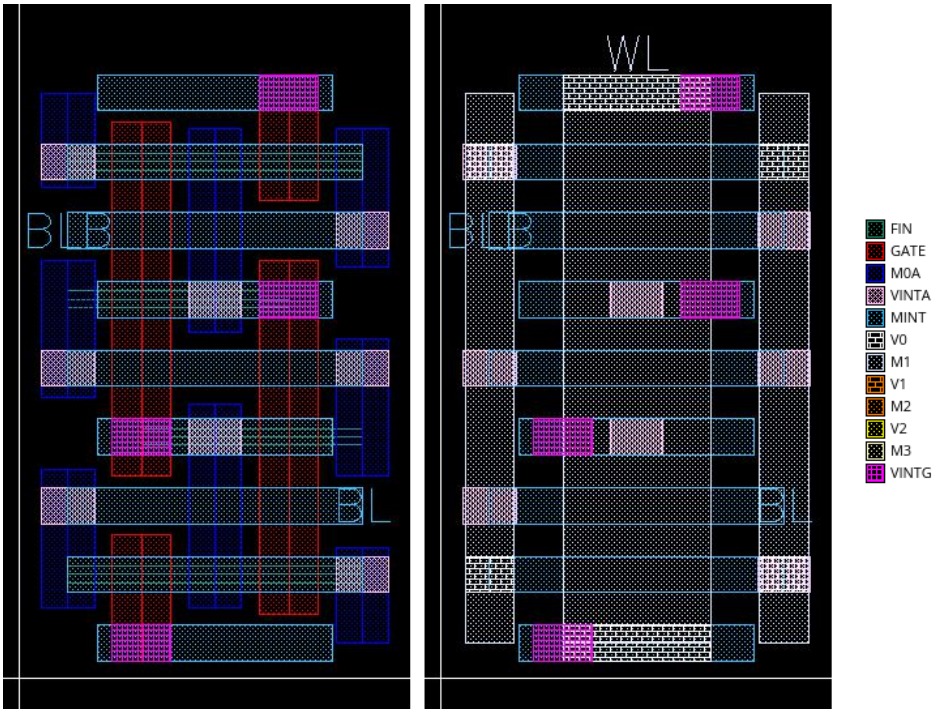


Figure A.1: Screengrabs of 6T SRAM layout in Virtuoso Layout Editor.

A.3. Multiport Transposable Cell Layouts

Figures A.2, A.3, A.4, A.5 show screengrabs of the 1P-4P SRAM layouts in the Virtuoso Layout Editor. Shown are at least two images; the first shows layers and vias {FIN, GATE, M0A, VINTA, MINT, VINTG}, the second shows layers and vias {MINT, VINTA, VINTG, V0, M1}. For the 3P and 4P cells a third image is shown to show the top metal layers, meaning layers and vias {V1, M2, V2, and M3}.

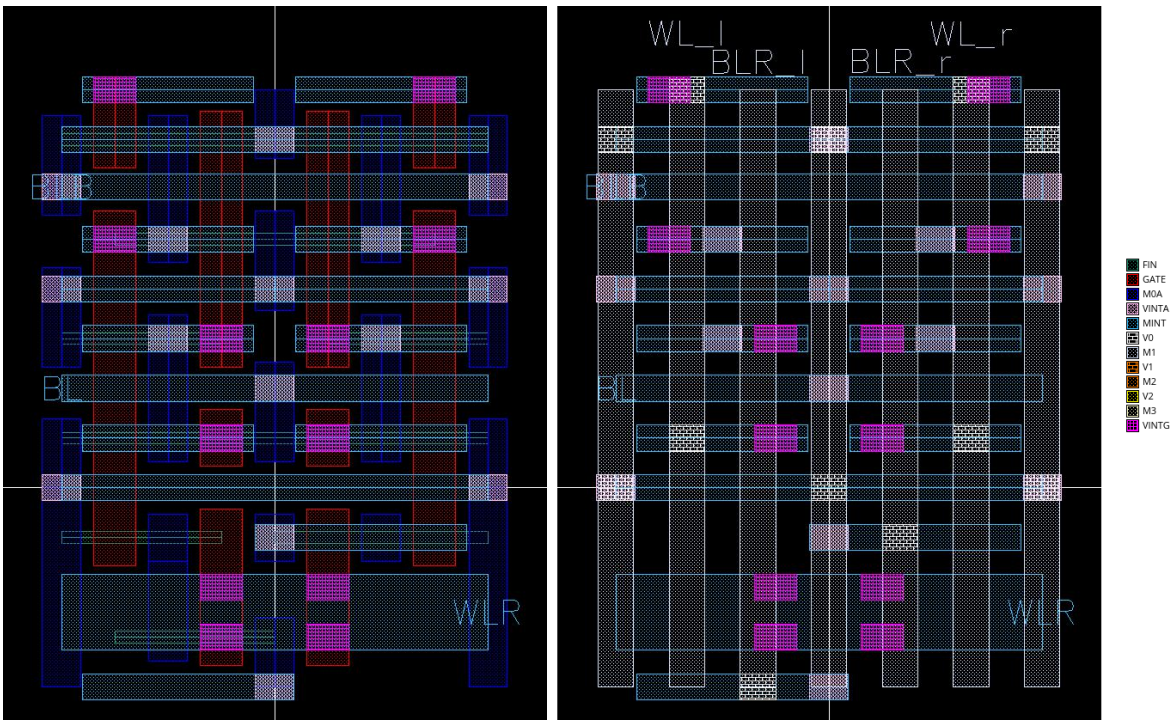


Figure A.2: Screengrabs of 1P SRAM layout in Virtuoso Layout Editor.

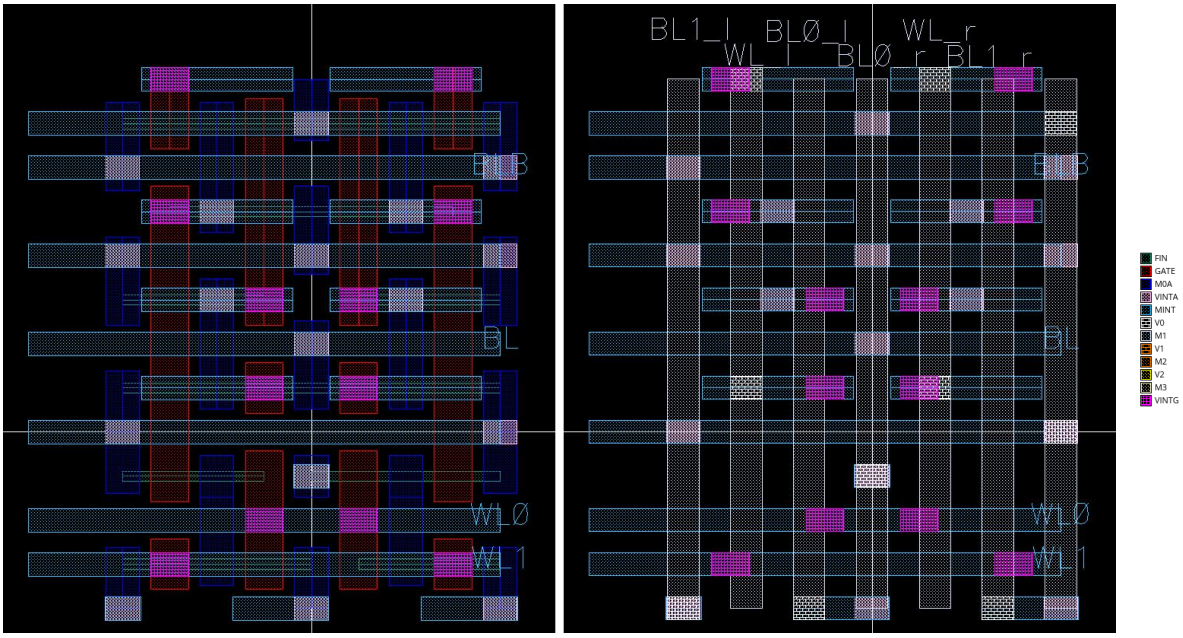


Figure A.3: Screengrabs of 2P SRAM layout in Virtuoso Layout Editor.

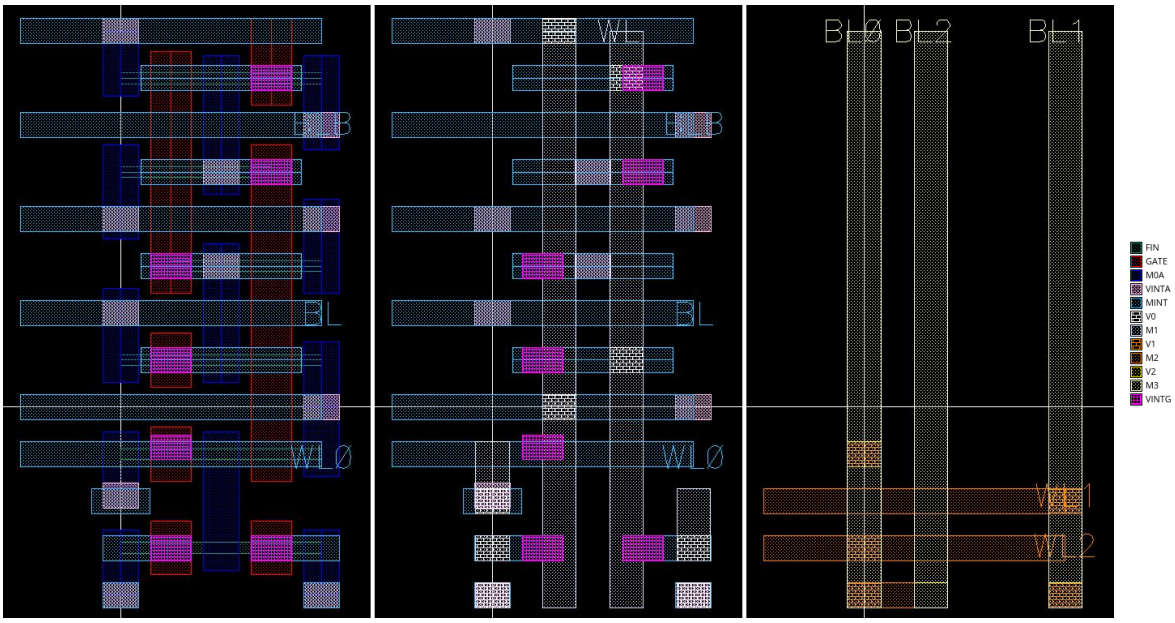


Figure A.4: Screengrabs of 3P SRAM layout in Virtuoso Layout Editor.

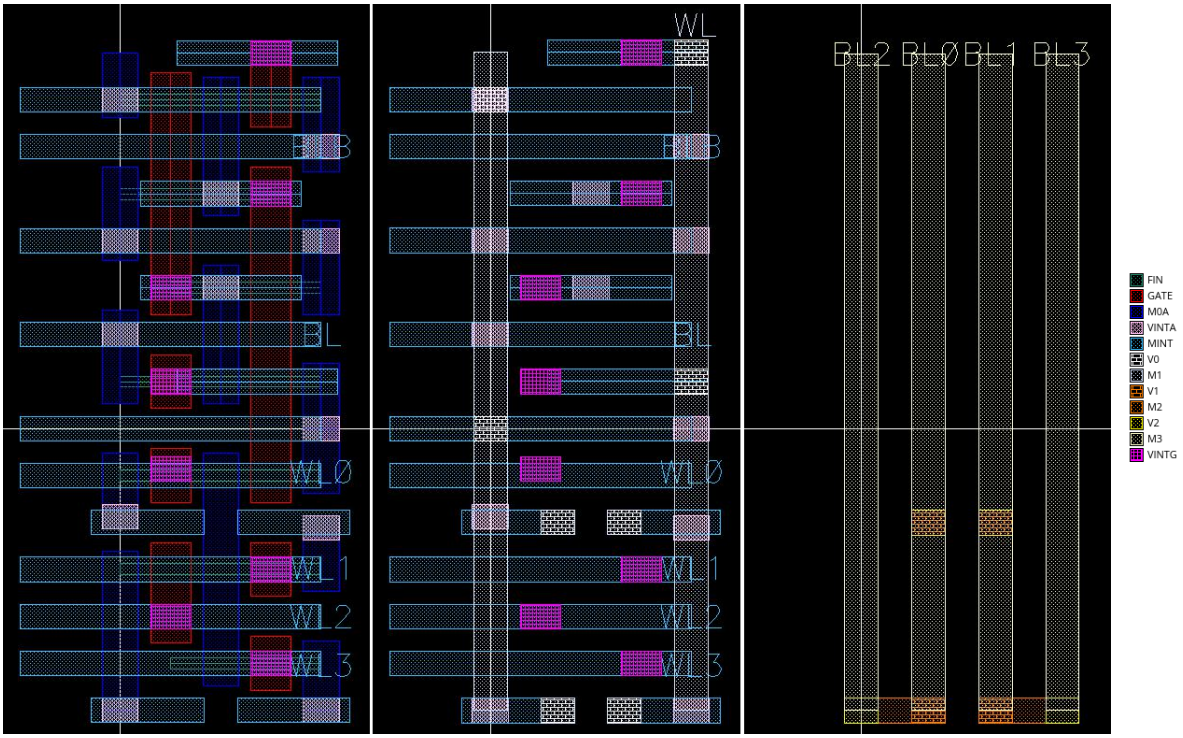


Figure A.5: Screengrabs of 4P SRAM layout in Virtuoso Layout Editor.

A.4. Alternative 1-port Transposable Cell Layout Schematic

Figure A.6 shows the alternative 1-port cell design as discussed in Chapter 4. This cell is 4% larger than the proposed cell as can be found in Figure 4.5, so the choice fell on that cell.

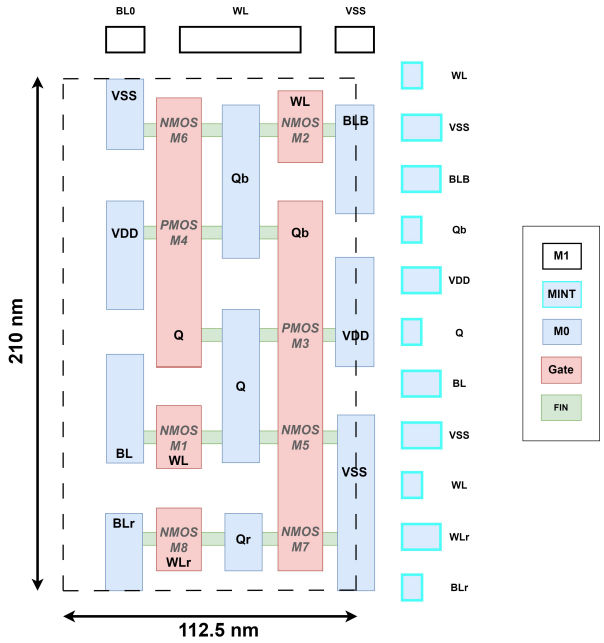


Figure A.6: Alternative 1-port cell layout.

B

VHDL Code

B.1. Arbiter

B.1.1. 4-Port Tree-Based Arbiter

VHDL description of 4-port tree-based Arbiter. Arbiters for less ports can be inferred from this description. The Arbiter uses four cascaded `prioEncoder_tree_Rout` components (described below), which are tree-based priority encoders. `Rout` (R' in Thesis text) is propagated from priority encoder to priority encoder. It contains the spikes in R that were not yet granted by an earlier priority encoder.

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.numeric_std.all;
4
5  entity arbiter_tree_4port is
6          generic (n: natural := 128);
7          port (  R:      in std_logic_vector(n-1 downto 0);
8                  clk:    in std_logic;
9                  rst:    in std_logic;
10                 G0:     out std_logic_vector(n-1 downto 0);
11                 G1:     out std_logic_vector(n-1 downto 0);
12                 G2:     out std_logic_vector(n-1 downto 0);
13                 G3:     out std_logic_vector(n-1 downto 0);
14                         no_R0:  out std_logic;
15                 no_R1:  out std_logic;
16                 no_R2:  out std_logic;
17                 no_R3:  out std_logic);
18 end arbiter_tree_4port;
19
20 architecture behav of arbiter_tree_4port is
21
22     signal Rout0, Rout1, Rout2: std_logic_vector(n-1 downto 0);
23
24     signal G0_int, G1_int, G2_int, G3_int: std_logic_vector(n-1 downto 0);
25
26     component prioEncoder_tree_Rout
27         generic (n: natural := 128);
28         port (  R:      in std_logic_vector(n-1 downto 0);
29                 G:      out std_logic_vector(n-1 downto 0);
30                 Rout:   out std_logic_vector(n-1 downto 0);
31                 no_R:   out std_logic);
32     end component;
33
34 begin
35
36     reg: process (clk, rst)
37     begin
38         if rst = '1' then
39             G0 <= (others => '0');
40             G1 <= (others => '0');
41             G2 <= (others => '0');
```

```
            G3 <= (others => '0');
        elsif (clk = '1' and rising_edge(clk)) then
            G0 <= G0_int;
            G1 <= G1_int;
            G2 <= G2_int;
            G3 <= G3_int;
        end if;
    end process;


    port0: prioEncoder_tree_Rout port map (       -- connect to overall input
        R => R,
        G => G0_int,
        Rout => Rout0,
        no_R => no_R0
    );

    port1: prioEncoder_tree_Rout port map (
        R => Rout0,
        G => G1_int,
        Rout => Rout1,
        no_R => no_R1
    );

    port2: prioEncoder_tree_Rout port map (
        R => Rout1,
        G => G2_int,
        Rout => Rout2,
        no_R => no_R2
    );

    port3: prioEncoder_tree_Rout port map (
        R => Rout2,
        G => G3_int,
        Rout => open,                       -- no need to propagate Rout
        no_R => no_R3
    );

end behav;
```

B.1.2. Tree-Based Priority Encoder

VHDL description of tree-based priority encoder. Contains a regular `prioEncoder` component for the tree trunk, and uses `prioEncoder_leaf_Rout` components for the two other levels of the tree. Both are described below.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity prioEncoder_tree_Rout is
generic (n: natural := 128);
        port (  R:      in std_logic_vector(n-1 downto 0);
                G:      out std_logic_vector(n-1 downto 0);
                Rout:   out std_logic_vector(n-1 downto 0);
                no_R:   out std_logic);
end prioEncoder_tree_Rout;

architecture behav of prioEncoder_tree_Rout is

    component prioEncoder_leaf_Rout is
        generic (n: natural := 16);
        port (  R:      in std_logic_vector(n-1 downto 0);
                en:     in std_logic;
                G:      out std_logic_vector(n-1 downto 0);
                Rout:   out std_logic_vector(n-1 downto 0);
                        yes_R:  out std_logic);
    end component;

    component prioEncoder is
```

```
25          generic (n: natural := 8);
26          port (  R:      in std_logic_vector(n-1 downto 0);
27                  G:      out std_logic_vector(n-1 downto 0);
28                  no_R:   out std_logic);
29      end component;
30
31      signal GGG, RRR: std_logic_vector(7 downto 0);
32      signal GG, RR: std_logic_vector(31 downto 0);
33
34  begin
35
36      master: prioEncoder generic map (8) port map (RRR, GGG, no_R);
37
38      gen: for i in 7 downto 0 generate
39          branches: prioEncoder_leaf_Rout generic map (4) port map ( RR( ((i+1)*4-1) downto (i
                *4) ) , GGG(i), GG( ((i+1)*4-1) downto (i*4) ) , open , RRR(i) );
40      end generate;
41
42      gen2: for j in 31 downto 0 generate
43          leaves: prioEncoder_leaf_Rout generic map (4) port map ( R( ((j+1)*4-1) downto (j*4)
                ), GG(j), G( ((j+1)*4-1) downto (j*4) ), Rout( ((j+1)*4-1) downto (j*4) ), RR(j)
                );
44      end generate;
45
46  end behav;
```

B.1.3. Priority Encoder

VHDL description of a standard Priority Encoder. Picks the highest-priority element from `R` and makes only that index '1' in `G`.

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.numeric_std.all;
4
5  entity prioEncoder is
6          generic (n: natural := 128);
7          port (  R:      in std_logic_vector(n-1 downto 0);
8                  G:      out std_logic_vector(n-1 downto 0);
9                          no_R:   out std_logic);
10 end prioEncoder;
11
12 architecture behav of prioEncoder is
13     signal S: std_logic_vector(n-1 downto 0);
14 begin
15
16     S(n-1) <= '0';
17
18     gen_most: for i in n-1 downto 1 generate
19         G(i) <= (not S(i)) and R(i);
20         S(i-1) <= S(i) or R(i);
21     end generate;
22
23     G(0) <= (not (S(0))) and R(0);
24     no_R <= not (S(0) or R(0));         -- report no R pending
25
26 end behav;
```

B.1.4. Priority Encoder as Tree Leaf

VHDL description of leaf Priority Encoder. Functions similar to `prioEncoder` component, but adds two features: an enable signal, `en`, that can make `G` be all '0', and an `Rout` signal (R' in Thesis text) that contains the spikes in `R` that were not granted.

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.numeric_std.all;
4
5  entity prioEncoder_leaf_Rout is
6          generic (n: natural := 16);
```

```
        port (  R:      in std_logic_vector(n-1 downto 0);
                en:     in std_logic;
                G:      out std_logic_vector(n-1 downto 0);
                Rout:   out std_logic_vector(n-1 downto 0);     -- R' in Thesis
                        yes_R:  out std_logic);
end prioEncoder_leaf_Rout;

architecture behav of prioEncoder_leaf_Rout is
    signal S: std_logic_vector(n-1 downto 0);
    signal G_int: std_logic_vector(n-1 downto 0);
begin

    S(n-1) <= '0';
    G <= G_int;

    gen_most: for i in n-1 downto 1 generate
        G_int(i) <= (not S(i)) and R(i) and en;            -- Grant
        --Rout(i) <= S(i) and R(i);               -- Requests not Granted
        Rout(i) <= R(i) and not G_int(i);
        S(i-1) <= S(i) or R(i);
    end generate;

    G_int(0) <= (not (S(0))) and R(0) and en;
    --Rout(0) <= S(0) and R(0);
    Rout(0) <= R(0) and not G_int(0);
    yes_R <= (S(0) or R(0));        -- report no R pending

end behav;
```

B.2. Neuron

B.2.1. Neuron Array

VHDL description of Neuron Array. The Neuron Array contains the `neuron` component and `decoder` components (described below). The `decoder` is used to address neurons to change `Vth` of individual neurons.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_misc.all;

entity neuronArray is
generic (   w: natural := 1;            -- Read ports
            M: natural := 8;            -- Number of neurons
            BLbits: natural := 2;       -- number of bits needed to count BL
            Vthbits: natural := 6;      -- number of Vth bits stored
            Vmembits: natural := 8);    -- number of Vmem bits stored
        port (  BLin        : in std_logic_vector(0 to (w*M)-1);
                yes_R       : in std_logic_vector(0 to w-1);
                rst         : in std_logic;
                clk         : in std_logic;
                Vth         : in signed(Vthbits-1 downto 0);
                Vth_wr      : in natural range 0 to M-1;
                R_empty     : in std_logic;
                Gnext       : in std_logic_vector(0 to M-1);
                Rout        : out std_logic_vector(0 to M-1));
end neuronArray;

architecture behav of neuronArray is

    component neuron is
        generic (   p: natural := 8;    -- number of multiport Ports
                    c: natural := 5;    -- number of bits needed to count BL
                    t: natural := 6;    -- number of Vth bits coming in
                    m: natural := 8);   -- number of Vmem bits maximally stored
        port (  BL :            in std_logic_vector(0 to w-1);
                v :             in std_logic_vector(0 to w-1);
                Vth:            in signed(t-1 downto 0);
                Vth_wr:         in std_logic;
```

```
                rst :           in std_logic;
                clk:            in std_logic;
                R_empty :       in std_logic;
                g:              in std_logic;
                r:              out std_logic);
    end component;

    component decoder is
        generic (   neu: natural := 8);   -- Number of neurons
        port (  neu_addr: in natural range 0 to neu-1;
                neu_sel : out std_logic_vector(0 to neu-1)
        );
    end component;

    signal neu_wr_sel: std_logic_vector(0 to M-1);

begin

    gen_neurons: for col in 0 to M-1 generate
        lbl: neuron generic map (w, BLbits, Vthbits, Vmembits) port map (
            BL => BLin(col*w to (col+1)*w-1),
            v => yes_R,
            Vth => Vth,
            Vth_wr => neu_wr_sel(col),
            rst => rst,
            clk => clk,
            R_empty => R_empty,
            g => Gnext(col),
            r => Rout(col)
        );
    end generate;

    neuron_addressing: decoder generic map (M) port map (
        neu_addr => Vth_wr,
        neu_sel => neu_wr_sel
    );

end behav;
```

B.2.2. Neuron

VHDL description of individual neuron. Takes `w` input ports, decodes and adds them from {1,0} to {+1,-1}, adds them to `Vmem`. When `R_empty = '1'`, neuron outputs on port `r` whether `Vmem >= Vth`

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity neuron is
    generic (   w: natural := 24;   -- number of multiport Ports
                c: natural := 6;    -- number of bits needed to count BL
                t: natural := 6;    -- number of Vth bits coming in
                m: natural := 8);   -- number of Vmem bits maximally stored
    port (  BL :            in std_logic_vector(0 to w-1);  -- input Bitlines
            v :             in std_logic_vector(0 to w-1);  -- which Bitlines valid
            Vth:            in signed(t-1 downto 0);        -- Vth to write
            Vth_wr:         in std_logic;                   -- enable Vth writing
            rst :           in std_logic;
            clk:            in std_logic;
            R_empty :       in std_logic;                   -- no spikes pending
            g:              in std_logic;                   -- spike granted
            r:              out std_logic);                 -- output spike pending
end entity neuron;

architecture behav of neuron is
    signal Vmem: signed(m-1 downto 0);
    signal Vth_stored: signed(t-1 downto 0);

    signal BLcount: signed(c-1 downto 0);
    signal r_int: std_logic;
```

```
27
28 begin
29
30     r <= r_int and R_empty;
31     Vth_stored <= Vth when Vth_wr = '1' else Vth_stored;
32
33     -- Decode & Add step
34     process(BL, v)
35         variable total: integer := 0;
36         variable inter: std_logic_vector(1 downto 0);
37     begin
38         total := 0;
39         for ii in 0 to w-1 loop
40             inter := (v(ii) and not(BL(ii))) & v(ii);
41             total := total + to_integer(signed(inter));
42         end loop;
43         BLcount <= to_signed(total, c);
44     end process;
45
46     -- process to add to Vmem, except if R_empty = '1'
47     -- if R_empty = '1', propagate whether V_mem >= Vth
48     process(rst, clk)
49         variable Vnew: integer := 0;
50     begin
51         if rst = '1' then
52             Vmem <= ( others => '0');
53             r_int <= '0';
54         elsif (clk = '1' and rising_edge(clk)) then
55
56             if R_empty = '1' then
57                 Vmem <= ( others => '0');
58                 if g = '1' then
59                     r_int <= '0';
60                 end if;
61             else
62                 Vnew := to_integer(Vmem + BLcount);
63                 if Vnew < -2**(m-1) then
64                     Vmem <= to_signed(-2**(m-1), m);
65                 elsif Vnew > 2**(m-1)-1 then
66                     Vmem <= to_signed(2**(m-1)-1, m);
67                 else
68                     Vmem <= to_signed(Vnew, m);
69                 end if;
70
71                 if Vnew >= to_integer(Vth_stored) then
72                     r_int <= '1';
73                 else
74                     r_int <= '0';
75                 end if;
76             end if;
77         end if;
78     end process;
79
80 end behav;
```

B.2.3. Decoder

VHDL description of simple decoder. Takes address and converts to a one-hot enable vector.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3  use IEEE.Numeric_Std.all;
4
5  entity decoder is
6      generic (   neu: natural := 128);   -- Number of neurons
7      port (  neu_addr: in natural range 0 to neu-1;
8              neu_sel : out std_logic_vector(0 to neu-1)
9      );
10 end entity decoder;
11
12 architecture behav of decoder is
```

```
13
14 begin
15
16     process(neu_addr)
17     begin
18         neu_sel <= (others => '0');
19         neu_sel(neu_addr) <= '1';
20     end process;
21
22 end behav;
```

C

Full Simulation Results

C.1. Parasitics Extraction Results

. Table C.1 shows all the parasitic Resistances and Capacitances extracted from the SRAM cell layouts.

Table C.1: Parasitics Extraction Results for all Bitlines and Wordlines. Resistances reported in Ω, Capacitances reported in aF.

	6T		**1P**		**2P**		**3P**		**4P**	
	R	**C**	**R**	**C**	**R**	**C**	**R**	**C**	**R**	**C**
WL	24.07	107.00	122.01	121.00	122.01	121.00	122.01	123.00	142.35	105.00
BL	62.65	37.70	62.65	38.30	78.31	41.70	93.97	43.10	93.97	42.80
BLB	62.65	37.70	62.65	38.30	78.31	42.10	93.97	44.00	93.97	46.10
WL0			18.81	60.50	70.48	56.50	93.97	50.90	93.97	54.20
WL1					70.48	45.90	93.97	62.00	93.97	53.40
WL2							93.97	64.80	93.97	60.30
WL3									93.97	51.00
BL0			122.01	80.60	122.01	52.00	122.01	36.20	142.35	37.20
BL1					122.01	70.00	122.01	26.70	142.35	35.50
BL2							122.01	32.00	142.35	42.80
BL3									142.35	33.70

C.2. NBL VWD Results

. Table C.2 shows the necessary V_{WD} for various SRAM array sizes to perform a successful Write Operation.

Table C.2: V_{WD} measurements for various SRAM array sizes, reported in mV.

	6T	**1P**	**2P**	**3P**	**4P**
$\mathbf{256 \times 256}$	-493.0	-834.3	-868.9	-851.7	-862.4
$\mathbf{256 \times 128}$	-242.2	-750.2	-675.7	-714.9	-771.4
$\mathbf{128 \times 128}$	-129.5	-277.7	-310.4	-353.3	-368.4
$\mathbf{256 \times 10}$	-90.17	-477.7	-483.3	-492.9	-505.6
$\mathbf{128 \times 10}$	0	-119.0	-124.0	-129.7	-137.9

C.3. Transposable Read and Write Results

Table C.3 shows the Read and Write Energy and Time via the transposed port for full array simulations of 128×128 and 128×10 arrays. Added are the full array results for V_{WD}.

Table C.3: Write and Read Energy and Time via the Transposed port for all tested SRAM cells, for 128×128 and 128×10 arrays. Additionally, full array simulation results for VWD.

		6T	1P	2P	3P	4P
128×128	**Write Energy [fJ]**	384.2	571.5	666.1	934.0	1078.1
	Read Energy [fJ]	842.6	897.7	921.1	933.2	931.7
	Write Time [fJ]	224.3	397.2	394.3	418.5	431.3
	Read Time [fJ]	378.8	619.1	633.0	644.7	669.1
	VWD [mV]	-129.5	-277.7	-310.4	-353.3	-368.4
128×10	**Write Energy [fJ]**	207.4	284.1	285.5	288.7	294.2
	Read Energy [fJ]	353.5	394.6	398.0	400.7	404.4
	Write Time [fJ]	190.0	407.9	403.5	405.1	415.9
	Read Time [fJ]	270.5	481.2	483.1	486.0	496.5
	VWD [mV]	0	-119	-124	-129.7	-137.9

C.4. Inference Read Results

Tables C.4 and C.5 show the Inference Read Energy and Time for 128×128 and 128×10 SRAM arrays respectively, for the four multiport cell variants. Additional variables are the precharge voltage V_{prech} and the number of Read operations.

Table C.4: Read Energy and Time via the Inference Ports for all tested multiport SRAm cells, for a 128×128 array. Results given for all possible numbers of Read operations and the four tested values of V_{prech}.

			Vprech [mV]			
	Cell	**Reads**	**700**	**600**	**500**	**400**
Energy [fJ]	**1P**	**1**	1087.7	822.6	614.3	541.3
	2P	**1**	884.1	676.6	531.7	524.2
		2	1853.7	1361.0	1031.4	957.0
	3P	**1**	809.4	617.5	486.3	524.4
		2	1437.0	1072.5	814.4	855.2
		3	2097.1	1551.5	1162.7	1203.3
	4P	**1**	857.5	651.5	499.3	556.1
		2	1563.6	1156.0	858.5	904.2
		3	2326.8	1704.0	1245.1	1274.2
		4	3007.3	2192.1	1593.9	1623.0
Time [ps]	**1P**	**1**	603.7	615.6	676.7	909.4
	2P	**1**	602.3	621.8	701.6	956.3
		2	667.9	686.0	776.4	1074.6
	3P	**1**	597.0	626.4	705.7	983.7
		2	608.0	639.5	724.7	1029.2
		3	617.8	651.6	740.5	1067.3
	4P	**1**	645.7	681.0	761.4	1076.6
		2	657.6	694.8	779.0	1116.6
		3	690.5	729.1	817.3	1180.8
		4	699.5	741.1	834.0	1226.7

Table C.5: Read Energy and Time via the Inference Ports for all tested multiport SRAm cells, for a 128×10 array. Results given for all possible numbers of Read operations and the four tested values of V_{prech}.

			Vprech [mV]			
	Cell	**Spikes**	**700**	**600**	**500**	**400**
Energy [fJ]	**1P**	**1**	130.9	110.7	94.7	88.7
	2P	**1**	121.6	100.2	93.5	91.7
		2	196.8	160.5	134.4	125.8
	3P	**1**	120.0	106.7	96.1	96.8
		2	171.9	146.4	126.0	123.3
		3	226.3	188.1	156.9	151.1
	4P	**1**	130.0	116.1	103.8	104.8
		2	190.2	162.0	137.7	133.9
		3	237.3	211.2	173.3	158.1
		4	492.2	503.8	548.2	705.1
Time [ps]	**1P**	**1**	552.7	557.1	601.8	757.4
	2P	**1**	463.2	470.6	514.5	658.3
		2	528.9	535.8	582.3	738.5
	3P	**1**	427.9	438.2	478.7	615.2
		2	430.0	440.3	482.2	621.4
		3	430.5	440.9	483.0	623.9
	4P	**1**	464.5	476.8	519.2	666.6
		2	467.3	479.8	523.1	672.1
		3	489.4	500.8	545.9	698.0
		4	492.2	503.8	548.2	705.1

D

Conference Paper Submission

The findings of this Thesis have been submitted as a Conference Paper for the DATE (Design, Automation & Test in Europe) Conference in September 2023. As of the time of writing this Thesis, it is not yet known whether the paper is accepted.

The following six pages show the paper as it was submitted.

Transposable Multiport SRAM-based In-Memory Compute Engine for Binary Spiking Neural Networks in 3nm FinFET

Anonymous
Blinded for review

***Abstract*—Ultra-low power Edge AI hardware is in increasing demand due to the battery-limited energy budget of typical Edge devices such as smartphones, wearables, and IoT sensor systems. For this purpose, we propose an ultra-low power event-driven SRAM-based Compute In-Memory (CIM) accelerator optimized for inference of Binary Spiking Neural Networks (B-SNNs). In this paper, we introduce a custom-designed 3nm SRAM cell with up to four read ports to improve inference performance and one transposable read/write port for efficient on-chip learning functionality. We exploit the event-based nature of SNNs to minimize the computation and memory cost. We benefit from technology scaling of fully digital design by synthesizing our accelerator in the 3nm FinFET technology node. The proposed accelerator's performance is evaluated by running MNIST inference at 97.6% accuracy, achieving an impressive throughput of 44M inferences/s at 607 pJ/inference (3.2 fJ per synaptic operation) while running at 29 mW. Our results demonstrate that the proposed accelerator provides an energy-efficient and high-performance solution for inference of Binary SNNs, opening up new possibilities for Edge AI applications.**

***Index Terms*—Compute in memory, Spiking Neural Network, Neuromorphic, Digital accelerator, SRAM**

I. Introduction

The demand for Artificial Intelligence applications to run on battery-powered Edge devices like smartphones, wearable devices, and various IoT systems is increasing rapidly. These devices are now dealing with a growing amount of data that needs to be processed using AI algorithms. As communicating raw data can be power-expensive and impose higher latency and privacy risks, there is a growing demand for (partial) execution of AI applications on Edge devices. The primary solution explored for low-power Edge AI is neuromorphic computing and the use of Spiking Neural Networks (SNNs). Specifically, most solutions adopt some form of large-scale parallel operation, performing Computation In-Memory (CIM), event-based computation, and reduction in parameter precision. The main challenge for neuromorphic accelerators lies in how to implement the essential Multiply-and-Accumulate (MAC) Operation quickly and efficiently. Research is being performed into solutions in both the analog and digital domain. In this work, a digital solution is chosen due to its robustness, scalability, and portability across technology nodes [1].

In the digital domain, where synaptic weights are stored in SRAM, performing the MAC operation using CIM requires additional hardware. Two main methods of CIM-MAC are typically utilized: Adder Trees and their variants [2]–[5] or Multiplication In-Memory with sequential accumulation in the SRAM Periphery (CIM-P) [6]–[9]. Adder trees allow for a higher degree of parallelism at the cost of breaking up the SRAM structure and adding a lot of hardware overhead. CIM-P minimizes hardware overhead and efficiently exploits SNN sparsity at the cost of lower parallelism in the pre-synaptic neuron dimension. This is because, for typical SRAM arrays, only one row may be accessed at a time, meaning only one pre-synaptic neuron can fire per timestep. Implementations such as [9] aim to mitigate this issue through approximate computing, but this degrades classification accuracy. Another issue for CIM-P is spike arbitration; ensuring only one spike enters the SRAM per timestep. Typically such arbitration systems are large and require multiple clock cycles per spike [6].

Additionally, on-chip learning is a popular practice for SNNs, allowing the SNN to adapt to changing environments and to be trained with smaller data sets. However, to efficiently perform on-chip learning, transposable access to the SRAM is essential. Various methods have been explored to make SRAM transposable. However, most methods either require additional hardware components in the SRAM array [10], negatively influence cell stability [7], [11], result in slow, high-power Read/Write operations [12], [13], or add more transistors than necessary [14], [15].

In this paper, we propose an SRAM-based Binary SNN accelerator using CIM for low-power Edge AI applications. Utilizing CIM-P ensures minimal hardware overhead and event-based computation. The Binary network simplifies the MAC operation to just reading from memory and performing a popcount in the SRAM periphery. Pre-synaptic neuron parallelism is improved by utilizing novel 3nm SRAM cells with multiple Read ports. Transposable Read/Write access for efficient online learning is provided through the original SRAM access ports. Additionally, a fully logic-based Arbiter is employed to ensure multiport spike arbitration in just one clock cycle.

The main contributions of this paper are:

- Design of a Binary-SNN hardware accelerator for ultra-low-power Edge AI applications using SRAM-based CIM.
- Design of a novel multiport SRAM bitcell in 3nm FinFET with multiple decoupled Read ports and a transposable Read/Write port to facilitate online learning.
- Design of a novel fully logic-based spike Arbiter for multiport SRAM Read access.

The proposed design is evaluated at synthesis level by classifying the MNIST handwritten digit data set, achieving 97.6% accuracy. Running this application, the 4-port design achieves a throughput of 44 MInf/s at just 607 pJ/Inf, while consuming 29 mW of Power.

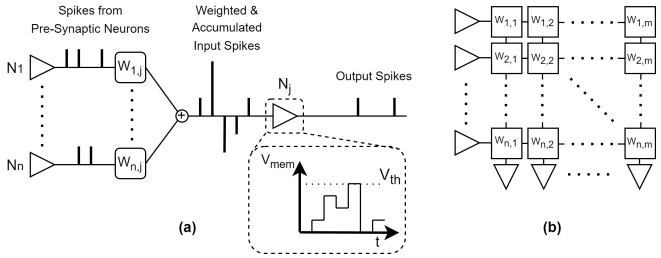


Fig. 1. (a) Illustration of a Binary SNN functioning; (b) Translation to memory crossbar.

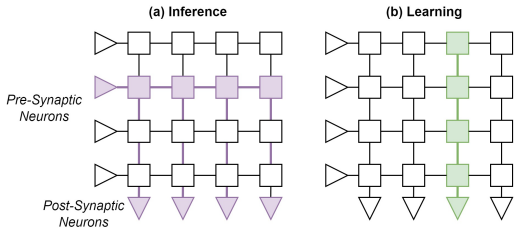


Fig. 2. Illustration of row-wise and column-wise access to synaptic weight array for Inference and Learning respectively.

The rest of this paper is organized as follows: Section II presents the basics of Binary-SNNs, as well as the concept of transposable crossbar memories. Section III presents a full overview the proposed accelerator architecture with different SRAM cell options, after which Section IV shows the performance of each option. Finally Section V concludes the paper.

II. Background

A. Binary Spiking Neural Networks In-Memory

Spiking Neural Networks (SNNs) represent a significant advancement from traditional Neural Networks [16] by utilizing self-recurrent neurons and Binary spikes with high temporal precision. These spikes result in a higher information-carrying capacity and event-based computing, minimizing Energy consumption. A Binary SNN (B-SNN) is an SNN where the synaptic weights are also represented by a Binary value; +1 or -1. Figure 1(a) illustrates such a B-SNN. Shown is the Integrate-and-Fire neuron, which accumulates weighted spikes as its membrane potential V_{mem} and sends out a spike itself when V_{mem} exceeds its threshold potential: $V_{mem} \geq V_{th}$. The other main spiking neuron variant is the Leaky IF neuron, which is suited better for time-based tasks. More complex models exist, but they are less convenient for hardware implementation [17].

Figure 1(b) shows how a B-SNN is mapped to a memory crossbar. The synaptic weights ($W_{i,j} = \{+1,-1\}$) are encoded as $\{1,0\}$. A spike corresponds to a memory row Read. A Read of '1' means the post-synaptic neuron state (V_{mem}) should be increased by 1, while a Read of '0' means V_{mem} should be decreased by 1. This means that, in contrast to conventional Binary Neural Networks (BNN), we only transmit spikes with +1 values, eliminating the need for XOR gates in the crossbar structure [18].

In order to train a B-SNN, different methods can be applied. These include converting a BNN into a B-SNN [19], or using bio-inspired Binary Spike-Timing-Dependent-Plasticity (STDP) models [20]. For the purpose of this study, we first trained a BNN and then converted it into a B-SNN, limiting the number of spikes to a maximum of one spike per neuron.

B. Transposable Crossbars for On-Chip Learning

On-chip learning is a popular practice in SNNs, where the network keeps learning even after deployment. This allows it to adapt to changing environments and be trained with smaller and more manageable data sets. For efficient on-chip learning, it is crucial to have access to the synapse weights in both the pre-synaptic and post-synaptic dimensions [20]–[22]. The pre-synaptic dimension is necessary so that a spike can be sent from a pre-synaptic neuron to the SRAM crossbar and multiplied by the weights between this neuron and all the connected post-synaptic neurons. The result should arrive at all the post-synaptic neurons. This process is illustrated in Figure 2(a) and corresponds to reading a memory Row. In contrast, learning in SNNs typically occurs when particular conditions arise in the post-synaptic neuron. Therefore, weight updates should occur to all the synapses before this post-synaptic neuron, corresponding to a memory Column, as shown Figure 2(b).

Standard SRAM allows Read/Write operations in either just the row or just the column direction. If we choose SRAM with only row-wise access, then reading and writing the weights in the column-wise direction would require dozens of row-wise Read operations. In contrast, transposable SRAM provides access in the row- and column-wise directions [7], [23]. Specifically, for SNN inference and on-chip learning, row-wise Read access and column-wise Read/Write access are required.

III. Proposed Solution

In this section, we present our proposed B-SNN accelerator. We first provide an overview of the full system architecture. Then we explain in more detail the most important subsystems: the SRAM Synapse array, the Arbiter, and the Neuron.

A. Architecture Overview

Figure 3 provides an overview of the macro architecture, which is comprised of a cascade of CIM-P Tiles. Each Tile represents a Fully Connected B-SNN layer. Spikes are transmitted as binary pulses through a set of parallel connections from Tile i to Tile $i+1$. Upon arrival, spikes are saved in Spike Request Vectors (R_i) before each Tile.

Inside each Tile, there are three main components: the Arbiter, the SRAM array, and the Neuron Array. For Tile i, the Arbiter takes Spike Request Vector R_{i-1} as its input. Every clock cycle, the Arbiter serves p requests from R_{i-1} that are allowed to enter the SRAM (p being the number of Read ports of the SRAM cells). These selected requests are called the Granted Requests, represented by G_i. The Granted Requests activate the corresponding word lines in the SRAM array simultaneously, and the SRAM outputs a set of bitlines, represented by B_i. These outputs are then presented to the Neuron Array, where they determine changes in the membrane potential V_{mem} of each neuron.

After serving all the spikes from R_{i-1}, if $V_{mem} \geq V_{th}$ (each neuron stores its own threshold potential), the neuron sets its output to '1', indicating that it wants to send a spike to the next Tile; all neuron outputs together form R_i. The final Tile behaves slightly differently; instead of comparing to a V_{th} and

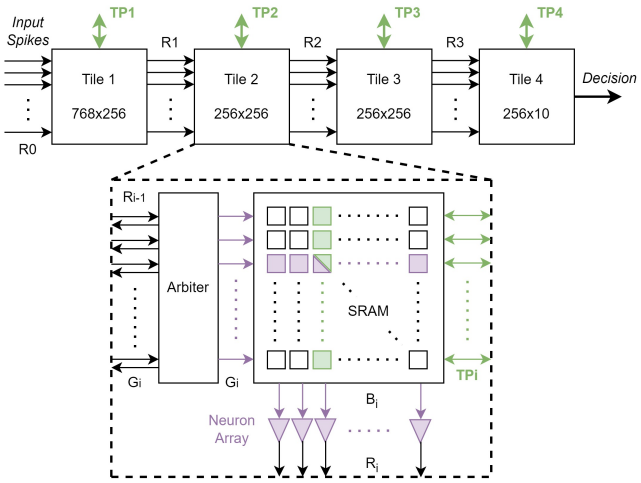


Fig. 3. Overview of proposed Macro Architecture. TP (green) indicates the transposable Read/Write access, with the Inference Read access in Purple.

creating R_i, it compares the V_{mem} of all neurons and outputs the index of the neuron with the highest V_{mem} as the *decision* of the network.

A register array is placed between the Arbiter and the SRAM Array. In effect this means each Tile can be divided into two pipeline stages; the Arbiter and the SRAM+Neuron. Additionally, the Tiles themselves are pipelined, meaning as soon as R_{i-1} has been served, the Tile forwards its spikes and can start serving a new vector R_{i-1}.

B. Transposable SRAM-based Synapse

As explained in Section II, in order to significantly increase the efficiency of online learning, it is essential to make the SRAM transposable. Additionally, to improve the inference latency, we would need several Read ports to be able to process multiple spikes in parallel. Therefore, our proposed SRAM has one dedicated column-wise Read/Write port and several row-wise Read ports. The most efficient method to achieve this is to maintain the original SRAM cell and its wordline (WL) and complementary bitlines (BL/BLB) for column-wise access. In order to add the first row-wise Read port, two NMOS transistors are added to the 6T SRAM cell, similar to the technique used for the standard 8T SRAM [24], [25]. To make this cell transposable, the newly added Read wordline (WL0) and Read bitline (BL0) should be rotated 90 degrees with respect to their WL and BL/BLB counterparts.

Figure 4(a) shows the resulting schematic. The original 6T SRAM cell is composed of M1-M6. Its WL runs vertically, providing column-wise Read/Write access. M7-M8 make up the added Read port, with WL0 running horizontally to allow row-wise Read access. The Read port works as follows: M7 is connected with its gate to QB (the inverted content of the cell). Before a Read operation, BL0 is precharged to V_{prech}. When WL0 is driven to '1', M8 conducts and connects BL0 to node Qr. If QB is '1', Qr connects to VSS, and BL0 is discharged. If QB is '0', BL0 is not discharged and remains at V_{prech}. By connecting the added Read port to cell using only the gate of M7, the port it is decoupled from the cell's content, enabling us to scale V_{prech} to lower values than VDD

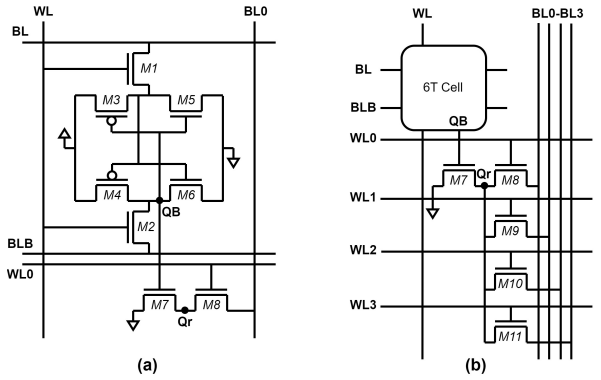


Fig. 4. Proposed transposable SRAM bitcell schematic; (a) Single Port; (b) Four Ports.

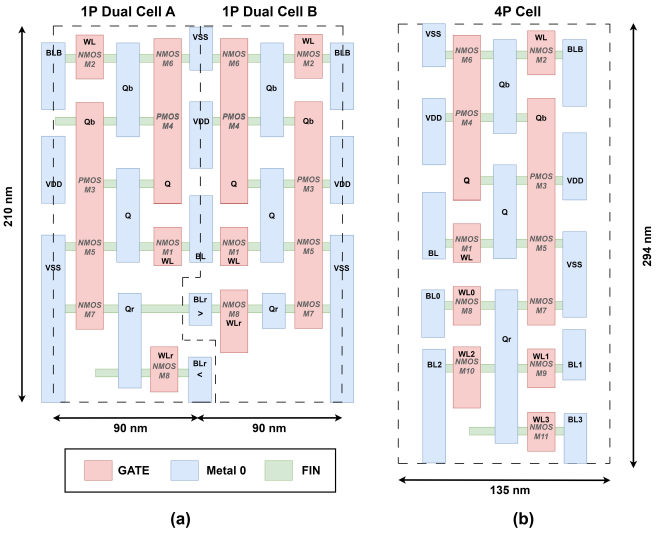


Fig. 5. Layout of the proposed SRAM bitcells; (a) Single Port; (b) Four Ports. Added are their bounding box and corresponding height and width.

with very little influence to the cell stability. This results in power savings at the cost of slower charging/discharging.

Our proposed transposable cell design has the advantage of allowing the addition of more Read ports at minimal cost, as can be seen in Figure 4(b). To add another Read port, all that is required is to replicate WL0, BL0, and M8. The added transistor should connect Qr to the newly added BLx, and its gate should be connected to the newly added WLx. The Arbiter system is designed to send multiple spikes into the SRAM, but never more than one spike in a single row. This means that no matter how many Read ports are added, M7 will only need to discharge one bitline.

The layouts of two of the designed cells are presented in Figure 5. The 1P and 2P cells are designed in dual form, as two non-symmetric cells are fitted together to minimize area. The result is a set of cells that together form a symmetric, repeatable dual cell. The area of the original 6T cell is $0.01512\mu m^2$ [26]. Adding each row-wise Read port increases the area by $1.25\times$, $1.4\times$, $1.9\times$ and $2.6\times$ respectively. We explored the possibility of adding 5+ ports, but only 4 bitlines could be fitted within the width of the 4-port cell. Adding another port would require further widening of the cell, increasing the area by 87.5% of the 6T cell, making it too area-inefficient.

To sense the BL/BLB of the transposed port, 4-to-1 Row MUXs and differential Sense Amplifiers (SA) are used. The BL0-BL3 of the Inference Read ports, however, do not have complementary counterparts and as such cannot use these SAs. In order to maximize inference throughput, it is beneficial to sense all BL0-BL3 of all Columns in a single cycle. Consequently, no MUXing should happen, and so all four sensing circuits must fit within an SRAM column width. To meet these requirements, we decided to use inverter gates to sense BL0-BL3, with their VDD port connected to the V_{prech} supply.

C. Arbiter Design

In Section III-A, it was explained that the Arbiter takes a Spike Request Vector R as input. The vector contains '1's which indicate for which SRAM wordlines there are pending spike requests. Figure 6 highlights the base form of the Priority Encoder at the top. The Priority Encoder takes R and selects the leftmost '1' in the vector, creating one-hot Grant vector G. The signal $s[n-1]$ is used to block any requests further to the right of the selection. If R does not contain any spike, noR is made '1'. The vector R' is the same as R except the selected spike is masked out. These non-granted spikes can be passed to subsequent priority encoders in a cascaded fashion to extend the system to multiple ports, generating multiple G-vectors within a single clock cycle.

Using this simple architecture for a Priority Encoder for a full SRAM results in an excessively long critical path, even for a single port. To solve this, multiple smaller Priority Encoders are combined in a tree structure as shown in Figure 6. Level 2 Priority Encoders take in the actual Request vector R in chunks. Using their *noR* signals they indicate to the levels above whether their chunk contains a request, forming a higher-level, shorter request vector; RR. The same happens on the next level to form RRR. The higher-level Priority Encoders then use their Grant signals to block/enable the lower-level Priority Encoders. The effective input/output behavior is unchanged, but at the cost of 8.0% area overhead

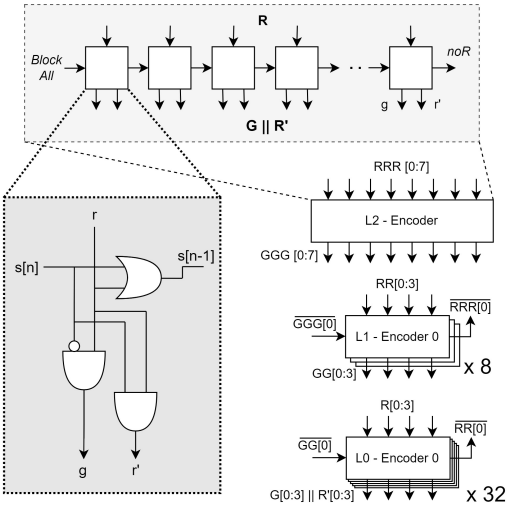


Fig. 6. The proposed logic-based Arbiter consisting of a 3-layer tree structure. Highlighted is the Priority Encoder building block and its internals.

the critical path is reduced significantly from $>1100ps$ to $<800ps$.

D. Neuron Design

For the purpose of this paper, IF neurons have been implemented. The reason for this choice is that the test setup involves the MNIST digit classification task. Hence, a more fine-tuned leak factor, as present in the LIF neuron, does not offer any significant advantage. However, it is worth noting that adding a leak factor to the presented neuron is a simple task and can be done without incurring any substantial hardware cost.

In each clock cycle, the neuron receives a set of {'1'/'0'} values on the bitlines it connects to, which it decodes to {+1/-1} and adds to its membrane potential V_{mem} stored in a register. V_{th} is stored in a latch register and can be changed at any time during operation, facilitating online learning. The V_{mem} accumulation continues until all spikes in R_{i-1} have been served. Then, if $V_{mem} \geq V_{th}$, the neuron output is set to '1' and V_{mem} is reset to zero before accumulating the next set of spikes. All neuron outputs together form Request vector R_i for the next Tile.

IV. Results

A. Experimental Setup

The entire system has been implemented using the 3nm FinFET technology node at the transistor level, with a global supply voltage of 700 mV. To obtain the reported SRAM Macro results, transistor-level simulations were carried out using Cadence Spectre. In order to simulate the SRAM parasitics, capacitance parasitics were extracted using Calibre PEX, and resistance parasitics were calculated based on the line geometries and material properties. All figures are based on accessing the worst-case cell or cell row, and all simulations were performed at the SS Process corner.

The SRAM Macro model is composed of the SRAM array, WL drivers, Write Drivers, BL sensing circuits, precharge circuits, and timing control. In order to improve the SRAM Write operation, the negative BL (NBL) assist technique is used, which creates a more negative voltage, $V_{WD} < VSS$, on the complementary bitline to force the cell to the desired state. This technique is necessary due to the high parasitics at smaller technology nodes. The required V_{WD} is determined for various array sizes, and if it is necessary for V_{WD} to be less than $-400mV$, the array size is considered non-valid for implementation due to low expected yield [27]. This restriction limits the array size to ≤ 128 Rows and Columns for all cell designs.

The rest of the components have been designed using RTL, and synthesis measurements have been conducted using Cadence Genus and Spectre. The synthesis results, combined with the SRAM Macro outcomes, are utilized to simulate the network in Python and determine the timing, power, and energy at the system level.

B. SRAM Results

Figure 7 shows the Time and Energy measurements for Writing to the cell and Reading from the cell using the transposed port (WL and BL/BLB). Write Time is the time between the start of the Write process and the cell content flipping to 90% of its intended value. Read Time is the delay

between the Wordline being driven and the data output of the Sense Amplifier flipping. Additionally, Write Energy is the Energy consumed during the Write Time, while Read Energy is the Energy consumed during a full clock cycle, which includes precharging of the BL/BLB [28].

As expected, both the Write and Read operation results scale with the addition of ports due to the parasitics that these ports introduce. The effect is stronger for the Write operation, as the parasitics also require a lower value of V_{WD} when more ports are added, increasing the voltage differential and, consequently, the power consumption. It is also worth noting that when just one extra Inference Port is added, there is immediately a significant increase in Write and Read times of the transposed port. This is because the WL wire in the proposed cells is narrower and thus more resistive, which is necessary due to the new BL0-BL3 that have to be routed in the same metal layer.

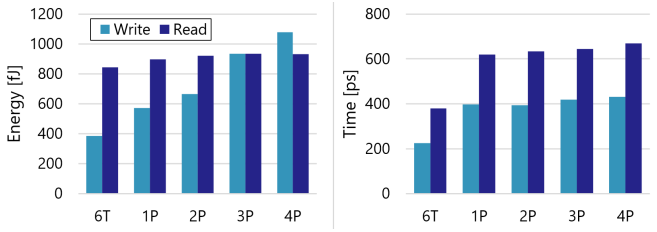


Fig. 7. Write and Read Energies and Timings via Transposed port for all tested SRAM Cells.

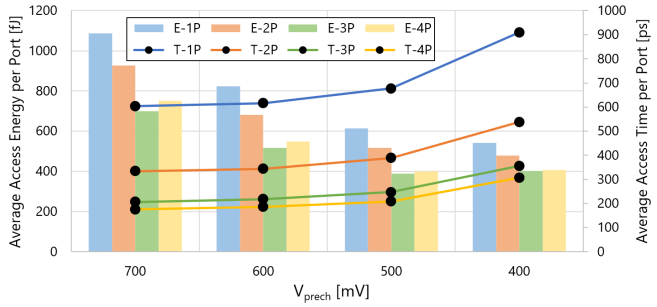


Fig. 8. Averaged access Energy and Time per each Inference port for various V_{prech} and various ports. Data is shown for 128 × 128 SRAM arrays.

Figure 8 demonstrates the relationship between access time and energy consumption for different V_{prech} levels and different numbers of Inference ports in SRAM. The results indicate full utilization of all available ports, meaning that if a cell has p ports, p read operations are performed. Total access time is calculated as the sum of the precharge time and the read time.

- **The effect of precharge voltage:** As Figure 8 indicates, there is a trade-off between access time and energy. We select $V_{prech} = 500mV$, as it leads to a reduction of at least 43% in energy consumption at the cost of at most 19% higher access time for all port numbers. Lowering V_{prech} from $500mV$ to $400mV$ saves up to 10% more energy for 1 and 2-port designs, but for 3 and 4-port designs, energy increases due to much slower precharging.
- **The effect of the number of Inference ports:** Adding extra Inference ports increases the parallelism and reduces the average access time. However, the average access energy starts increasing after adding the fourth port. This increase in energy consumption per port can be explained by the increased cell size that creates more parasitics. The results support our earlier claim that increasing the ports to more than four would not be beneficial due to the excessive area and resulting parasitics.

C. Timing Evaluation

Calculating the system clock period involves determining the duration of each pipeline stage. To account for process and environmental variations, we allocate $300ps$ of slack for the Arbiter stage and $400ps$ for the SRAM stage. The latter value is higher to ensure sufficient data hold time for the neuron registers to record the SRAM data. Table I shows the measured duration of each pipeline stage, including slack. The longest stage is highlighted to indicate that its duration determines the clock period.

TABLE I
REQUIRED TIME FOR EACH PIPELINE STAGE FOR THE DIFFERENT SRAM CELLS. HIGHLIGHTED IS THE LONGEST OF THE 2 STAGES, INDICATING THE CLOCK PERIOD.

	6T	1P	2P	3P	4P
Arbiter	1.007ns	1.007ns	1.040ns	1.034ns	1.006ns
SRAM + Neuron	0.685ns	1.077ns	1.176ns	1.141ns	1.234ns

D. System-Level Evaluations

1) Online Learning: Online learning becomes significantly more efficient with transposable access ports. Without a transposable port, for an array size of 128 × 128, it would take 2 × 128 clock cycles ($257.8ns$ and $157pJ$) to read and write all the weights. For our proposed SRAM cells, the synaptic weights of a post-synaptic neuron can be read and written in just 2 × 4 cycles, where the factor 4 is due to the use of 4-to-1 MUXs. The 4-port cell, which is the worst performer in the transposed Read/Write port metrics, has a clock period of $1234ps$. This means it requires only $9.9ns$ ($26.0\times$ less) and $8.04fJ$ ($19.5\times$ less) to read and write a full column.

2) Inference: To evaluate the system's inference performance, we have created a fully Connected B-SNN network for MNIST digit classification by placing multiple tiles in sequence. The network has a structure of *768:256:256:256:10*. In case a layer exceeds the maximum SRAM array size, multiple SRAM arrays are used in a single tile. Each SRAM has its own 128-wide Arbiter. This increases parallelism by another factor; a 256-wide layer will have two p-port Arbiters, meaning up to $2 \times p$ spikes can selected per clock cycle. In order to reduce the input images from 784 to 768 pixels, a 2 × 2 set of pixels is removed from every corner of the images. This ensures that the first layer corresponds to exactly 6 × 128 inputs. We have trained the network as a BNN with a sign activation function and per-neuron biases, which are converted to thresholds. The BNN is then converted to a B-SNN, as described in [19]. The resulting accuracy achieved by the network is 97.64%.

Figure 9 provides a comparison of system-level power, performance, energy, and area for the 5 discussed SRAM cell options. Generally, the power of the system increases with the

number of added ports. However, the system's power implemented with 6T cells is higher than that of the 1P and 2P cells. This is due to the active power savings from the voltage scaling of V_{prech} for the decoupled inference ports. When comparing the 6T and 1P cells, throughput decreases slightly, as the effective parallelism is the same, but read operations for the 1P cell are slower due to the added parasitics. However, at 2+ ports, the added parallelism compensates for this. Additionally, with every added port, the overall energy/inference decreases significantly due to the increased spike throughput.

Table II compares the 4P cell with state-of-the-art low parameter resolution SNN accelerators aimed at low-power applications. We achieved typical overall power consumption, but significantly improve energy/SOP (Synaptic Operation), energy/inference, and throughput.

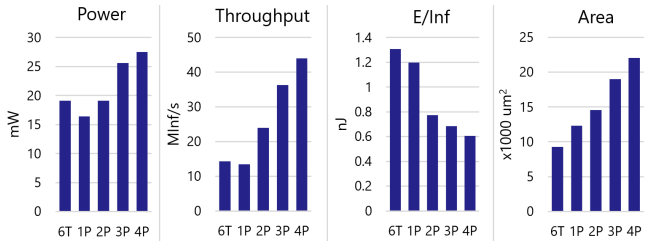


Fig. 9. System-Level comparison between using standard 6T SRAM Cell and our proposed cells.

TABLE II
COMPARISON WITH STATE OF THE ART SMALL-SCALE SNN ACCELERATORS.

	[6]	**[9]**	**[10]**	***This Work***
Technology [nm]	65	10	65	3
Neuron Count	650	4096	1K	778
Synapse Count	67K	1M	256K	330K
Activation Bit Width	6	1	–	1
Weight Bit Width	1	7	5	1
Transposable	No	No	Yes	Yes
Clock Frequency	70kHz	506MHz	100MHz	810MHz
MNIST				
Power	305nW	196mW*	53mW	29.0mW
Accuracy [%]	97.6	97.9	97.2	97.6
Throughput [inf/s]	2	6250	20	44M
Energy/Inference [nJ]	195	1000	–	0.607
Energy/SOP**	1.5pJ	3.8pJ	15.2pJ	3.2fJ

* Inferred from SOP/s/mm^2, Area, and pJ/SOP
**SOP: Synaptic Operations

V. CONCLUSION

In our research paper, we presented an SRAM-based CIM accelerator designed for B-SNNs in 3nm FinFET. The synthesis results indicate that our system exhibits highly competitive accuracy on the MNIST benchmark, while also reducing system-level power consumption and increasing the number of inferences per second by several orders of magnitude. These achievements were made possible through the use of an innovative SRAM cell that enables enhanced parallelism and voltage scaling, an improved spike arbitration system, a low-cost communication fabric, and technology scaling which is made possible by the fully digital design. We also utilized a fully binary neural network architecture, which further contributed to the success of the system by minimizing the operation complexity.

REFERENCES

[1] S. K. Bose, J. Acharya, and A. Basu, "Is my Neural Network Neuromorphic? Taxonomy, Recent Trends and Future Directions in Neuromorphic Engineering," in ACSSC 2019, pp. 1522–1527.
[2] H. Fujiwara et al. "A 5-nm 254-TOPS/W 221-TOPS/mm2 Fully-Digital Computing-in-Memory Macro Supporting Wide-Range Dynamic-Voltage-Frequency Scaling and Simultaneous MAC and Write Operations," in ISSCC 2022, pp 1–3.
[3] Y. Chih et al. "An 89TOPS/W and 16.3TOPS/mm2 All-Digital SRAM-Based Full-Precision Compute-In Memory Macro in 22nm for Machine-Learning Edge Applications," in ISSCC 2021, pp. 252–254.
[4] D. Wanq et al. "All-Digital Full-Precision In-SRAM Computing with Reduction Tree for Energy-Efficient MAC Operations," in ICTA 2022, pp. 150–151.
[5] D. Wang et al. "DIMC: 2219TOPS/W 2569F2/b Digital In-Memory Computing Macro in 28nm Based on Approximate Arithmetic Hardware," in ISSCC 2022, pp. 266–268.
[6] D. Wang et al. "Always-On, Sub-300-nW, Event-Driven Spiking Neural Network based on Spike-Driven Clock-Generation and Clock- and Power-Gating for an Ultra-Low-Power Intelligent Device," in A-SSCC 2020, pp. 1–4.
[7] J. Seo et al. "A 45nm CMOS neuromorphic chip with a scalable architecture for learning in networks of spiking neurons," in CICC 2011, pp. 1–3.
[8] D. Kim et al. "MONETA: A Processing-In-Memory-Based Hardware Platform for the Hybrid Convolutional Spiking Neural Network With Online Learning," in Frontiers in Neuroscience 16, 2022.
[9] G. Chen et al. "A 4096-Neuron 1M-Synapse 3.8-pJ/SOP Spiking Neural Network With On-Chip STDP Learning and Sparse Weights in 10-nm FinFET CMOS," in IEEE JSSC 54.4, 2019, pp. 992–1002.
[10] J. Kim et al. "Efficient Synapse Memory Structure for Reconfigurable Digital Neuromorphic Hardware," in Frontiers in Neuroscience 12, 2018.
[11] J. Wang et al. "A Compute SRAM with Bit-Serial Integer/Floating-Point Operations for Programmable In-Memory Vector Acceleration," in ISSCC 2019, pp. 224–226.
[12] H. Jiang et al. "CIMAT: A Compute-In-Memory Architecture for On-chip Training Based on Transpose SRAM Arrays," in IEEE TC 69.7, 2020, pp. 944–954.
[13] D. Wang et al. "All-Digital Full-Precision In-SRAM Computing with Reduction Tree for Energy-Efficient MAC Operations," in ICTA 2022, pp. 150–151.
[14] S. K. Bose and A. Basu, "A 389TOPS/W, 1262fps at 1Meps Region Proposal Integrated Circuit for Neuromorphic Vision Sensors in 65nm CMOS," in A-SSCC 2021, pp. 1–3.
[15] Z. Lin et al. "Two-Direction In-Memory Computing Based on 10T SRAM With Horizontal and Vertical Decoupled Read Ports," in IEEE JSSC 56.9, 2021, pp. 2832–2944.
[16] W. Maass, "Networks of spiking neurons: The third generation of neural network models," in Neural Networks 10.9, 1997, pp. 1659–1671.
[17] H. Paugam-Moisy and S. Bohte, "Computing with Spiking Neuron Networks," in Handbook of Natural Computing 1, 2012, pp. 335–376.
[18] R. Liu et al. "Parallelizing SRAM Arrays with Customized Bit-Cell for Binary Neural Networks," in IEEE DAC 2018, pp. 1–6.
[19] H. Kim, H. Oh, and J. Kim, "Energy-efficient XNOR-free In-Memory BNN Accelerator with Input Distribution Regularization," in ICCAD 2020, pp. 1–9.
[20] A. Yousefzadeh et al. "On practical issues for stochastic STDP hardware with 1-bit synaptic weights," in Frontiers in Neuroscience 12, 2018.
[21] A. Yousefzadeh et al. "Hardware implementation of convolutional STDP for on-line visual feature learning," in ISCAS 2017, pp. 1–4.
[22] A. Rostami et al. "E-prop on SpiNNaker 2: Exploring online learning in spiking RNNs on neuromorphic hardware," in Frontiers in Neuroscience 16, 2022.
[23] S. Yu et al. "Compute-in-memory chips for deep learning: Recent trends and prospects," in IEEE Circuits and Systems Magazine 21.3, 2021, pp. 31–56.
[24] L. Chang et al. "An 8T-SRAM for Variability Tolerance and Low-Voltage Operation in High-Performance Caches," in IEEE JSSC 43.3, 2008, pp. 956–963.
[25] D. Kim et al. "Processing-In-Memory-Based On-Chip Learning With Spike-Time-Dependent Plasticity in 65-nm CMOS," in IEEE LSSC 3, 2020, pp. 278–281.
[26] H.-H. Liu et al. "DTCO of sequential and monolithic CFET SRAM," in DTCO and Computational Patterning II, vol. 12495, 2023.
[27] H.-H. Liu et al. "Extended Methodology to Determine SRAM Write Margin in Resistance-Dominated Technology Node," in IEEE Transactions on Electron Devices 69.6, 2022, pp. 3113–3117.
[28] H.-H. Liu et al. "CFET SRAM DTCO, Interconnect Guideline, and Benchmark for CMOS Scaling," in IEEE Transactions on Electron Devices 70.3, 2023, pp. 883–890.