Fixed-Point Implementations for Feed-forward Artificial Neural Networks

Daniel Llamocca
Electrical and Computer Engineering Department
Oakland University
Rochester, MI, USA
llamocca@oakland.edu

***Abstract*—We present scalable and generalized fixed-point hardware designs (source VHDL code is provided) for Artificial Neural Networks (ANNs). Three architectures are presented: multiply-and-add, multiplier-less, fully pipelined. In addition, we include two approaches for ANN binary layers: accumulation-based and fully pipelined. The fully customized hardware architectures allow for design space exploration to establish trade-offs among numerical format, processing time, resource usage, and numerical accuracy. Users can select the ANN architecture, ANN parameters (structure, weights, biases), the numerical format for both the input/output data in every layer and the network parameters (weights and biases). Results are presented in terms of resources, processing time, and numerical accuracy. The proposed architectures were implemented on modern FPGAs. These hardware designs are expected to be used as building blocks on a variety of applications such as CNNs and SNNs, as well as a platform for educational purposes.**

Keywords*— *Artificial Neural Networks, Register Transfer Level, field-programmable gate array (FPGA).

I. Introduction

ANNs are fundamental components in many applications such as computer vision, speech recognition, natural language processing, and automatic driving [1].

Compared to software implementations, ANN hardware implementations can fully exploit the parallel operation of neurons and reduce the cost of implementation. In addition, they can be embedded in a wide range of systems. An extensive overview of hardware implementations (analog, digital, hybrid) for several ANN models is presented in [2].

Our focus is on digital hardware implementations at the Register Transfer Level (RTL). This type of hardware-level design can improve the efficiency and achieve greater acceleration, but it requires an in-depth understanding of the algorithm structure [3]. This is an active area of research and FPGA-based implementations have attracted attention due to its reconfiguration capability. A good summary of implementation schemes (targeted to FPGAs) is presented in [4].

An FPGA-based implementation is presented in [5], where one layer implements all the network layers via a time-multiplexing scheme; neurons are implemented with LUTs. A compact hardware implementation where each neuron is treated as a Boolean function is presented in [6]; the method is more efficient for low number of input bits. An implementation with fast carry look-ahead adder and Booth multiplier is presented in [7]. The work in [8] presents a reconfigurable feed forward neural networks: nodes can be reorganized for a particular application.

Lately, high-level frameworks for the implementation of ANNs have become popular. The work in [9] presented a platform for the generation of parameterized FPGA-based architecture for feed forward ANN with backpropagation learning algorithms. The work in [10] features a design automation tool that generates RTL-based ANN accelerators. Layer implementation is straightforward: neurons as implemented via an adder tree. It allows for time-multiplexing of resources when there are logic constraints. The work in [11] presented an ANN designed with Xilinx System Generated and incorporated into an ECG classifier.

CNN implementations are also worth mentioning. Within a convolutional layer, each feature map is the result of a sum of convolutions, the pooling operation and activation function. Each pixel of a feature map is the result of a sum of products. This resembles an ANN layer (with fewer computations). A review of techniques for accelerating deep learning networks on FPGAs is provided in [12]. Stochastic computing can simplify the hardware complexity in these implementations [13]. The work in [14] presents an implementation of LSTM (long short term memory) based on Stochastic Computing.

High-level frameworks for CNN implementations are worth mentioning. In [15], a systolic array architecture is presented and a process by which a CNN is mapped onto a systolic array architecture. A related work is presented in [16], where symbolic descriptions of CNNs are converted into FPGA-based accelerators using RTL-HLS hybrid templates. Another high-level framework is presented in [17].

The high-level frameworks are very useful in many applications as they allow for rapid prototyping, design space exploration, and the testing of optimization techniques at higher abstraction layers. However, this creates dependency on software tools (commercial and custom-built) that might not be typically available and that might lack future support.

Though some frameworks present innovations in terms of hardware implementation of ANN components (e.g.: activation function, time multiplexing of resources), RTL architecture details are usually scarce, such as numerical format across the

datapath, implementations (folded, unfolded, systolic, etc.), cycle-accurate analysis of latency).

We believe that there is room for architecture improvements at the pure RTL description domain targeted to FPGAs as well as ASICs. As a result, we propose a unified set of parameterized architectures that can be directly incorporated in the design of various ANN-based applications that use fixed-point arithmetic. The RTL design we provide is written in VHDL, it is self-contained (no need of libraries) and it can be used directly in any FPGA-based tool or ASIC-based tool. Our dedicated fixed-point hardware designs let the user specify specific hardware design parameters (e.g.: datapath numerical format, weights/bias numerical formats, ANN architecture, implementation style) thereby allowing the user to select what to optimize for: resource usage, numerical accuracy, and processing time. We include designs for real-valued and binary-valued ANN layers. For the architectures, results in terms of accuracy, and hardware design parameters (e.g.: datapath bit-width, implementation approach) are provided.

The remainder of the paper is organized as follows. Section II briefly describes a standard ANN. Section III describes the hardware implementation approaches for an ANN layer. Section IV describes hardware implementations for an ANN. Section V presents the results in terms of processing time, accuracy, and resources. Finally, conclusions are given in Section VI.

II. Artificial Neural Network

A. Notation

ANNs are organized into layers, where all the neurons in one layer receive their inputs from neurons in a prior layer and provide outputs to the subsequent layer. We let L denote the number of layers, and l the layer index (0 to $L-1$), where $l = 0$ is the input layer.

Fig. 1(a) depicts a 3-layer neural network (also called Fully Connected Layer). Fig. 1(b) depicts the inputs and outputs of the first neuron in layer $l = 2$. Fig. 1(c) depicts the artificial neuron model. The neuron output (action potential a_j^l) results from applying an activation function to the membrane potential (z_j^l). The index corresponds to neuron j in layer l. The membrane potential z_j^l is a dot product between the inputs and the associated weights, to which a bias is then added [1].

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l, \; l \geq 1 \qquad (1)$$

Note that w_{jk}^l represents the synaptic gain factor from neuron k in layer $l-1$ (previous layer) to neuron j in layer l. The action potential intensity of a neuron is denoted by a_j^l, and it is modeled as a scalar function (activation function) of z_j^l.

$$a_j^l = \sigma\left(z_j^l\right) = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right), \; l \geq 1 \qquad (2)$$

Common activation functions include these non-linear operations:

- Rectified Linear Unit (ReLU): $\sigma\left(z_j^l\right) = max\left(0, z_j^l\right)$.
- Hyperbolic Tangent: $\sigma\left(z_j^l\right) = tanh\left(z_j^l\right)$.
- Sigmoid function: $\sigma\left(z_j^l\right) = 1 \Big/ \left(1 + e^{-z_j^l}\right)$

The output of a layer l can be described using a vectorized notation (Eq. (3)). Fig. 2 depicts the matrix operation for z^l. We let NO denote the number of outputs (or number of neurons), and NI the number of inputs. Here, z^l is a column vector that includes all the membrane potentials of layer l, a^{l-1} is a column vector that includes all the input signals to layer l, w^l is the weight matrix containing all the synaptic gain factors from layer $l-1$ to layer l, and b^l is a column vector that includes the biases of all neurons of layer l. The result a^l is a column vector that includes all the action potentials of layer l.

$$a^l = \sigma(z^l), \; z^l = w^l a^{l-1} + b^l, \; l \geq 1 \qquad (3)$$

As for activation functions, there are different nonlinear functions, e.g.: *tanh*, *sigmoid*, *softmax*, *ReLU*. Due to its simplicity, ReLU is preferred. Using specialized circuitry for the computation of the other functions is impractical as there are many neurons operating in parallel per layer. Approximation methods and LUT-based approaches are preferred. The work in [18] describes an approximate method to implement *tanh*, while the work in [7] proposes a piecewise linear approximation for the *sigmoid* function. The work in [19] presents a twofold LUT generic approach, while the work in [9] implements the *sigmoid* function using an LUT combined with linear interpolation.

B. Implementations

This standard ANN can be implemented in multiple domains (e.g.: microprocessors, GPUs, FPGAs, ASICs). We focus on RTL-based hardware designs using fixed-point arithmetic.

III. Hardware Implementations for an ANN Layer

In this section, we describe our parameterized, fixed-point architecture implementations for a generic ANN layer. They are designed at the Register Transfer Level (RTL) and described in VHDL. We let L denote the number of layers, and l to denote the layer index (from 0 to $L-1$), where $l = 0$ is the input layer.

Note that we refer to an ANN hardware architecture (or implementation) as the RTL hardware design, while the ANN architecture is what is commonly understood as the ANN structure (number of layers, neurons per layer, etc.)

The hardware designs are classified into two categories: real-valued inputs and binary-valued inputs. For each category, we present different implementation approaches, that are described in detail (datapath, control mechanism, design parameters).

A. Real-valued inputs: Generic Layer

We let XI to denote the number of inputs, and XO the number of outputs (or number of neurons). The inputs are grouped as a vector a^{l-1}, and the outputs are a^l.

We use signed fixed-point representation for the datapath, where $[NO\ NQ]$ represents a fixed-point format with NO integer bits and NQ fractional bits. Then, we let the input data format be $[ni\ pi]$, the weights/biases format be $[nw\ pw]$, the internal datapath format be $[nt\ pw + pi]$, and the output format be $[no\ po]$. For full accuracy, the output format is given by $[nt\ pw + pi]$, where $nt = ni + nw + \lceil \log_2(XI + 1) \rceil$. The user can truncate the output to the format $[no\ po]$. The weight matrix (w^l) and bias vector (b^l) are loaded as constant parameters via text files.

Three implementation approaches are presented. The signal k ($k = 0, \dots, XI-1$) indexes an element of the input vector a^{l-1}. The index j ($j = 1, \dots, XO$) refers to a neuron in the layer.

1) Multiply-and-Accumulate: Fig. 3 depicts the generic architecture for an ANN Layer with its parameters and I/O ports. Fig. 4 depicts a neuron: its components (adder, multiplier, register, mux), and the FX format at every stage. The computation starts by loading the bias in the register, and then accumulating the products (weight by input signal: $w_{jk}^l \times a_k^{l-1}$) at every clock cycle.

This architecture processes an input sample in $XI+1$ cycles. We can feed input samples to the ANN layer every $XI+1$ cycles. Fig. 5 depicts a timing diagram for $XI = 5$.

2) Fully parallel/pipelined: Fig. 6 depicts the generic architecture for the ANN Layer with its parameters and I/O ports. The neuron architecture uses XI multipliers, an adder tree [20], and a register. All the products, computed in parallel, are added up by the adder tree. An input sample is processed in $\lceil log_2 XI + 1 \rceil + 2$ cycles. Its fully pipelined nature allows us to feed input samples at every clock cycle, i.e., a new output a^l can be computed per clock cycle.

3) Multiplier-less: Fig. 7 depicts the generic architecture for the ANN Layer with its parameters and I/O ports. Fig. 8 depicts a neuron with its components (adder/subtractor, register, MUX); the datapath format is indicated at every stage. The computation starts by loading the bias in the register, and then accumulating the products (weight by input signal). The product $w_{jk}^l \times a_k^{l-1}$ is implemented by accumulating (we add or subtract based on the sign of a_k^{l-1}) w_{jk}^l for $\left|a_k^{l-1}\right|$ times (a_k^{l-1} treated as an integer value). The result includes pi extra fractional bits to account for the fractional bits of a_k^{l-1}.

The computation of $w_{jk}^l \times a_k^{l-1}$ takes $\left|a_k^{l-1}\right|$ cycles. This means that the number of cycles can range from 0 to 2^{ni-1}. As a result, while this may be an effective approach for small ni, it can also be very impractical for large ni. Also, the number of cycles is non-deterministic as it depends on the input values.

This architecture processes an input sample (and can be fed a new input sample) in $\sum_{k=0}^{XI-1}\left|a_k^{l-1}\right| + 1$ cycles.

B. Binary-valued inputs: Generic Layer (or Binary layer)

This is a special case where the input elements of vector a^{l-1} are binary-valued (0 or 1). However, we note that the elements of output vector a^l are real-valued. This binary layer is different than the real-valued input Layer with $ni = 1$, where data is treated as signed FX. A binary layer is useful in applications such as Spiking Neural Networks [21], or an ANN whose first layer processes binary-valued inputs.

We let the weights/biases format be $[nw\ pw]$, the internal datapath format be $[nt\ pw]$, and the output format be $[no\ po]$. For full accuracy, the output format is given by $[nt\ pw]$, where $nt = nw + \lceil \log_2(XI+1) \rceil$. The user can truncate the output to the format $[no\ po]$. The weight matrix (w^l) and bias vector (b^l) are loaded as constant parameters via text files. Two implementation approaches are presented:

1) Accumulator-based: This is based on the multiply-and-accumulate hardware for real-valued inputs. Given the binary inputs, the multipliers are not needed, hence resembling the multiplier-less approach applied to binary inputs.

The generic architecture for the Binary Layer is depicted in Fig. 9 with its parameters and I/O ports. The neuron uses an adder, multiplier, MUX. The computation starts by loading the bias in the register, and then accumulating the products (weight by input signal) at every clock cycle. The product $w_{jk}^l \times a_k^{l-1}$ is either *0* or w_{jk}^l. This architecture processes an input sample in $XI+1$ cycles. And we can feed input samples to the ANN layer every $XI+1$ cycles.

2) Fully parallel/pipelined: Fig. 10 depicts the generic architecture for the Binary Layer, its parameters and I/O ports. In the neuron architecture, each product $w_{jk}^l \times a_k^{l-1}$ is either 0 or w_{jk}^l. The adder tree adds up all the products (including the bias). The architecture processes an input sample in $\lceil log_2 XI + 1 \rceil + 2$ cycles. Its fully pipelined nature lets us feed input samples at every clock cycle, i.e., a new output a^l can be computed per clock cycle.

Table I lists the five architectures along with their processing delay (latency), minimum number of cycles between input samples, and hardware resources. Note that the multipliers we employ have one constant operand.

C. Weight matrix and bias vector

These design parameters of the RTL hardware designs are specified in `.txt` files, from which the VHDL code access them. The weight matrix is stored in a raster scan fashion. The generation of the `.txt` file can be carried out by any software application (MATLAB, C++, Pytorch) after an ANN is trained. More details are provided in the next section.

IV. Hardware Implementation for ANNs

Here, we present hardware implementations of ANNs with real-valued inputs. As for binary-valued layers, they can be used as part of an SNN, or as part of an ANN whose first layer processes binary-valued data.

A. Generic ANN with L layers

For L layers, l denotes the layer index ($0, \dots, L-1$), where $l = 0$ is the input layer (with XO^0 outputs). For $l \geq 1$, XI^l denotes the number of inputs, and XO^l the number of outputs. Fig. 11 depicts a generic architecture for an ANN Layer with its parameters and I/O ports. The design parameters are '*name*' (ANN name), '*type*' (layer type: multiply-and-add, fully parallel/pipelined, and multiplier-less), and $[ni^l\ pi^l]$ (user-selectable FX format per layer, $l = 0, \dots, L-1$).

An ANN is implemented by cascade connecting the individual layers (one after the other), as in Fig. 11. The start ('*s*' or '*E*') and 'v' signals are also connected in cascade.

The ANN name ('*name*') provides a reference to the ANN structure as well as the weight matrix and bias vector per layer. Specifically, it references these design parameters:

- FX format for weight matrix and bias vector elements for all layers: $[nw\ pw]$.
- Input and output sizes for every layer, specified in parameter $XL = [XO^0, XO^1, \dots, XO^{L-1}]$. The input layer ($l = 0$) only has outputs whose size is $XO^0 \times 1$. For layer l ($l \geq 1$), we have that $XI^l = XO^{l-1}$, the input size is $XI^l \times 1$, the output size is $XO^l \times 1$, the weight matrix size is $XO^l \times XI^l$, and the bias vector size is $XO^l \times 1$.
- Weight matrix (w^l) and bias vector (b^l) values per layer ($l \geq 1$), available in a set of `.txt` files. For layer l, the `.txt` filenames for the weight matrix and bias vector are:

  Weight matrix: `w_<name>_L<Layer #>_XO`^{l}`xXI`^{l}`.txt`
  Bias vector: `b_<name>_L<Layer #>_XO`^{l}`x1.txt`

B. Examples

A network can be built in a high-level platform like Pytorch or MATLAB, where we train the network and then extract the weights and matrices, and then generate the `.txt` files. We show two ANN examples:

1) 'test1': This 4-layer ANN is depicted in Fig. 12. Table II provides a summary. The structure (including the weight matrix and bias vector per layer) is fixed. The user can select the FX format for every layer ($l \geq 0$). Table II lists the largest FX format for full accuracy; however, a user can select shorter FX formats that will incur in truncation and precision loss. The `.txt` filenames are the following:

- $l = 1$: `w_test1_L1_3x2.txt, b_test_L1_3x1.txt`
- $l = 2$: `w_test1_L2_3x3.txt, b_test_L2_3x1.txt`
- $l = 3$: `w_test1_L3_1x3.txt, b_test_L3_1x1.txt`

TABLE II. STRUCTURE AND DESIGN PARAMETERS FOR NETWORK 'TEST1'. L=4. WEIGHTS/BIAS FORMAT: [NW PW] = [8 2].

Layer	Input Size	Output Size	Weight matrix	Bias vector	FX Format	
					Input	Output
0		2x1				[6 0]
1	2x1	3x1	3x2	3x1	[6 0]	[16 2]
2	3x1	3x1	3x3	3x1	[16 2]	[26 4]
3	3x1	1x3	1x3	1x1	[26 4]	[36 6]

2) 'mlosh': This 3-layer ANN is depicted in Fig. 13. Table III provides a summary and lists the largest FX format that allows for full accuracy; a user can select a shorter FX format that will incur in truncation and precision loss. The `.txt` filenames are the following:

- $l = 1$: `w_mlosh_L1_16x196.txt`, `b_test_L1_16x1.txt`
- $l = 2$: `w_mlosh_L2_10x16.txt`, `b_mlosh_L2_10x1.txt`

TABLE III. STRUCTURE AND DESIGN PARAMETERS FOR NETWORK 'MLOSH'. L=3. WEIGHTS/BIAS FORMAT: [NW PW] = [8 7].

Layer	Input Size	Output Size	Weight matrix	Bias vector	FX Format	
					Input	Output
0		196x1				[9 0]
1	196x1	16x1	16x196	196x1	[9 0]	[25 7]
2	16x1	10x1	10x16	10x1	[25 7]	[38 14]

C. Feeding data to the ANN Architecture

The design parameter '*type*' allows the user to select the hardware implementation approach: multiply-and-add, fully parallel/pipelined, and multiplier-less. The way input samples are fed varies depending on the approach.

1) Multiply-and-accumulate: Each layer l has a processing delay given by $p_l = XI^l + 1$. The total processing delay (latency) is $P = \sum_{l=1}^{L-1} p_l$ cycles. A naïve approach is to issue a new sample every P cycles. An optimized approach is to think of layers ($l = 1, \dots, L-1$) as pipeline stages. Here, we can issue a new sample every $F = max(p_1, p_2, \dots, p_{L-1})$ cycles. Fig. 14(a) illustrates this for the '*test1*' ANN (see Fig. 12). Fig. 14(b) illustrates why F is the largest delay among all stages.

Fig. 15 shows a timing diagram that illustrates the behavior of the '*s*' and '*v*' signals for every stage (for ANN '*test1*'). For the first input sample, the '*s*' and '*v*' pulses of every stage are shaded in gray. For the second input sample, the associated '*s*' and '*v*' pulses are shaded in orange. For the third input sample, the associated '*s*' and '*v*' pulses are shaded in green. For the fourth input sample, the associated '*s*' and '*v*' pulses are shaded in cyan. Note how we can feed a new input sample every $F = 4$ clock cycles.

2) Fully parallel/pipelined: Each layer l has a processing delay $p_l = \lceil log_2(XI^l + 1) \rceil + 2$. The total processing delay (latency) is $P = \sum_{l=1}^{L-1} p_l$ cycles. Here, the nature of the architecture allows us to issue a new input sample every clock cycle ($F = 1$). Note that instead of '*s*', we call the signal '*E*' (enable) as this is more standard for pipelined architectures.

3) Multiplier-less: Each layer l has a processing delay $p_l = \sum_{k=0}^{XI^l-1} |a_k^{l-1}| + 1$. The total processing delay (latency) is $P = \sum_{l=1}^{L-1} p_l$ cycles. Here, we can use a pipelined approch and issue a new sample every $F = max(p_1, p_2, \dots, p_{L-1})$ cycles. In practice, it is not possible to compute p_l. The best we can do is to establish a bound (assuming the largest data values) to compute p_l. Usually, as the number of bits grow per layer, $F = p_{L-1}$. This approach can be helpful when the number of bits is small (say less than 8), otherwise it is impractical.

Table IV lists the three ANN architecture types along with their processing delay, and the minimum number of cycles between input samples (using a pipelined feeding approach).

In addition, when processing N input samples (images) the throughput is given by:

$$Throughput = \frac{\#\ of\ samples\ (N)}{Time\ to\ process\ N\ samples} = \frac{N}{\sum_{l=1}^{L-1} p_l + F \times (N-1)} \quad (4)$$

Given Eq. (4), we can refer to the formulas for p_l and F for each implementation approach. For sufficiently large N, the throughput results in $1/F$ (input sample per F clock cycles). F is known for the multiply-and-accumulate approach, $F = 1$ for the fully parallel/pipelined approach, and F is a non-deterministic quantity in the multiplier-less approach.

D. On Finite Precision effects

Throughout the datapath, we have truncated the fixed-point format of the data in our ANN examples (see Tables II and III). A fixed-point format that considers full numerical precision requires an impractical number of bits. RTL-based ANN hardware designs are affected by the finite wordlength of the fixed-point format in the datapath, the weights/bias format, and the input data format (if truncated). A survey of recent methods of ANN quantization techniques is available in [22].

Among the Post-Training quantization techniques, we can mention the work in [23] that converts a pre-trained single floating-point ANN to an 8-bit fixed-point ANN using various techniques: quantization range setting, cross layer equalization, bias correction, and AdaRound. AdaRound [24] is a weight-rounding mechanism that adapts to the data and the task loss; this improves significantly over rounding-to-nearest. Other methods include Ternary Quantization [25] and Loss-Aware Post-Training quantization [26].

V. Results and Analysis

This section details the experimental setup for the proposed hardware designs and then provides results in terms of arithmetic precision, hardware resources, and performance.

A. Experimental Setup

We test both the real-valued ANN examples (‘*test1*’, ‘*mlosh*’) and the Binary Layers. A MATLAB model (in double floating-point arithmetic) is used as a basis for comparison.

For the ‘*mlosh*’ ANN, we specifically trained this network using a downsampled version (14x14 images) of the 60,000-element MNIST database [27]. Training was carried out via MATLAB using the standard back-propagation method (‘mse-loss’ as cost function) with a learning rate of 0.5 and 2 epochs.

Results in terms of arithmetic precision and hardware resources are obtained by evaluating the proposed hardware designs via both synthesis (on Xilinx® FPGAs) and cycle-accurate simulation on Xilinx® Vivado software.

For the real-valued ANN examples (‘*test1*’, ‘*mlosh*’), Tables II and III provide ANN structure details, the weights/bias format, and the I/O FX format per layer that allows for full arithmetic precision (based on the given weight/bias format). This largest FX format is an artifact of the FX formulas; in practice, data values might need smaller FX formats.

We evaluate how different I/O FX formats (per layer) and weights/bias FX format affect arithmetic precision. Table V lists 3 different I/O FX formats (including ‘a’, the largest FX format) for the network ‘*test1*’. Table VI lists 3 different I/O FX formats for the network ‘*mlosh*’. We compare all the three cases (‘a’, ‘b’, ‘c’) against our MATLAB model.

In addition, we compare the arithmetic precision of the cases ‘b’ and ‘c’ with those of the ‘a’ case. This will provide insight as how the effect of I/O FX formats differ from that of the weights/bias format.

The parameterized VHDL implementations of the proposed designs allow us to get results in terms of arithmetic precision and performance. A space of hardware configurations is generated by varying the design parameters (I/O FX formats, weights/bias format, input size). For ‘*test1*’, we use a 100-element dataset, while for ‘*mlosh*’, we use a downsampled version (14x14) of the 10,000-element MNIST database.

TABLE V. Different Sets of FX formats for ‘test1’ ANN. L=4. Weights/bias format: [NW PW] = [8 2].

	Approach	Layer 1	Layer 2	Layer 3
In / Out	a	[6 0] / [16 2]	[16 2] / [26 4]	[26 4] / [36 6]
	b	[6 0] / [14 2]	[14 2] / [22 4]	[22 4] / [30 6]
	c	[6 0] / [12 0]	[12 0] / [18 0]	[18 0] / [24 0]

TABLE VI. Different Sets of FX formats for ‘Mlosh’ ANN. L=3. Weights/bias format: [NW PW] = [8 7].

	Approach	Layer 1	Layer 2
In / Out	a	[9 0] / [25 7]	[25 7] / [38 14]
	b	[9 0] / [24 6]	[24 6] / [36 12]
	c	[9 0] / [22 4]	[22 4] / [32 8]

B. Arithmetic Precision Assessment

The precision of a fixed-point architecture is affected by the FX format selected by the design parameters.

We evaluate arithmetic precision using the mean squared error (MSE) between the ANN the results from the model in MATLAB. We also report the ANN accuracy to show how different FX formats affect this accuracy.

TABLE VII. ‘test1’ ANN. L=4. For quantized approaches (a,b,c), the weights/bias format is [NW PW] = [8 2]. The MSE is measured on the output of the FCN.

Approach	a	b	c
MSE w.r.t. ideal	1.0298×10^{6}	1.0298×10^{6}	1.3981×10^{6}
MSE w.r.t. ‘a’		0	1.7789×10^{5}

TABLE VIII. ‘Mlosh’ ANN. L=3. For quantized approaches (a,b,c), the weights/bias format is [NW PW] = [8 7]. The MSE is measured on the 10 outputs of the FCN. The accuracy of the ideal case is 92.87%.

Approach	a	b	c
MSE w.r.t. ideal	216.8886	216.9236	217.1275
	230.2636	230.3180	230.6217
	1073.9	1074.1	1075.2
	894.4896	894.6366	895.56
	178.4973	178.5286	178.7447
	1059.6	1059.8	1060.9
	51.9337	51.9435	52.0091
	299.6514	299.7131	300.1101
	297.3214	297.3657	297.6845
	448.9770	449.0701	449.6818
MSE w.r.t ‘a’		6.9271×10^{-6}	2.0295×10^{-4}
		1.3485×10^{-5}	3.6790×10^{-4}
		2.7388×10^{-5}	8.3988×10^{-4}
		2.3380×10^{-5}	7.1864×10^{-4}
		1.3938×10^{-5}	3.5657×10^{-4}
		2.3494×10^{-5}	7.3138×10^{-4}
		6.0845×10^{-6}	1.6207×10^{-4}
		1.5755×10^{-5}	4.5142×10^{-4}
		2.9504×10^{-5}	7.3988×10^{-4}
		3.1058×10^{-5}	8.8308×10^{-4}
ANN Accuracy	92.83%	92.83%	92.83%

For ‘*test1*’, Table VII shows the MSE for the 100-element dataset, a different MSE for each FX case.

For '*mlosh*', Table VIII shows 10 MSEs (one for each output) for all the samples. This is shown for each FX case. Here, we also show ANN accuracy for each FX case.

When computing the MSE of the cases 'b' and 'c' with respect to 'a', keep in mind that the ANN weights and biases are quantized. As such, these MSE results are useful to determine how the circuit structure affect the precision of the results.

We also computed the MSE of the cases 'a', 'b', and 'c' with respect to an ideal MATLAB implementation with unquantized weights/biases. The results are mixed: it looks like we need more fractional bits for weights/biases as the MSE is large, though ANN accuracy does not change.

All in all, there is not much variation when MSE is computed with respect to the case 'a', i.e., the effect of weights/bias FX format is not pronounced. However, there is large variation when computing the MSE with respect the ideal MATLAB implementation (that has quantized weights/bias). These results suggest that quantizing the weights/bias has a larger effect than quantizing the I/O FX format per layer.

C. Hardware Resource Utilization

Tables IX and X show the hardware resource utilization of the different architectures in terms of slice registers, 6-input LUTs, and DSP Slices for the design parameters specified in Table V and Table VI respectively. The results were obtained using Vivado software tool targeted to Artix-7 FPGA device.

TABLE IX. 'TEST1' ANN. L=4. WEIGHTS/BIAS FORMAT: [NW PW] = [8 2]. THE QUANTIZED APPROACHES (REFER TO TABLE V) ARE LABELED 'A', 'B', 'C'.

	Multiply-and-accumulate			Fully parallel/pipelined			Multiplier-less	
	LUT	FF	DSP	LUT	FF	DSP	LUT	FF
a	485	301	4	732	776	9	436	349
b	435	279	4	548	633	12	445	321
c	419	253	4	503	574	12	396	289

TABLE X. 'MLOSH' ANN. L=3. WEIGHTS/BIAS FORMAT: [NW PW] = [8 7]. THE QUANTIZED APPROACHES (REFER TO TABLE VI) ARE LABELED 'A', 'B', 'C'.

	Multiply-and-accumulate			Fully parallel/pipelined			Multiplier-less	
	LUT	FF	DSP	LUT	FF	DSP	LUT	FF
a	3559	3056	10	75108	76828	115	2670	3095
b	3516	3030	10	74900	76579	115	2633	3068
c	3372	2978	10	74314	76255	115	2561	3014

As for the ANN Binary Layers, Table XI shows some resource results for the Layer 1 of both the '*test1*' and '*mlosh*' networks. We use the largest output FX format for Layer 1. The fully pipelined case uses more resources, and this is evident the larger the ANN is.

TABLE XI. RESOURCE UTILIZATION FOR BINARY LAYERS. 'TEST1' LAYER 1: XI=2, XO=3, [NW PW] = [8 2]. 'MLOSH' LAYER 1: XI=196, XO=16, [NW PW] = [8 7]

Output FX Format	Accumulation-based		Fully pipelined	
	LUT	FF	LUT	FF
Layer 1 'test1': [10 2]	48	33	33	40
Layer 1, 'mlosh': [16 7]	910	574	12324	15651

For fixed input data wordlength, the I/O FX formats have a small effect on hardware resources. Implementation-wise, the fully parallel/pipelined approach uses significantly more resources than the multiply-and-add or the multiplier-less approach. The multiply-and-add approach uses slightly more resources than the multiplier-less.

Unsurprisingly, the ANN architecture (number of layers, neurons per layer) plays a significant role. Finally, the weight/bias FX format does not play a significant role unless the wordlength is modified.

D. Hardware Performance

Performance-related formulas are available in Table I (for individual layers) and Table IV (for ANNs with real-valued inputs).

Table XII lists the actual processing cycles for '*test1*' and '*mlosh*' ANNs (with real-valued inputs). For each case, the wordlenghts for full accuracy are employed. Note that the results from multiplier-less implementation are data dependent (we tried with one specific sample for each case). We can see how the multiplier-less can only be useful for very small networks.

TABLE XII. PROCESSING CYCLES FOR ANN EXAMPLES 'TEST1' AND 'MLOSH'

ANN	Metric	Multiply-and-accumulate	Fully Parallel/ Pipelined	Multiplier-less
'test1'	Latency (1 sample)	11	12	$2.65*10^{14}$
	Min. # of cycles between input samples (F)	4	1	$2.637*10^{14}$
	Throughput (samples/cycle)	0.25	1	$1/2.637*10^{14}$
'mlosh'	Latency (1 sample)	214	17	$4.6385*10^{14}$
	Min. # of cycles between input samples (F)	197	1	$4.531*10^{14}$
	Throughput	1/197	1	$1/4.531*10^{14}$

As for throughput, Table XII expresses it in samples per clock cycle. Another way to look at it is by considering a 100 MHz operating frequency. Here, the multiply-and-accumulate architecture has a throughput of 25 Msamples per second for 'test1' and 0.507 Msamples for second for 'mlosh'. For the fully parallel/pipelined case, the throughput is 100 Msamples per second for all cases. We do not include the multiplier-less results as they are too low.

E. Comparison with other implementations

We note that hardware comparisons are very difficult to carry out fairly as the test cases presented vary greatly in terms of implementations: ANN architecture, datawidth of the datapath, weights/bias datawidth pooling mechanism, activation function. When comparing different works, if comparable designs are not found, it might be preferable to compare the method/architecture rather than raw resources.

Table XIII provides some comparisons with related works. We include a comparable RTL design [5], and designs generated by high-level frameworks [9],[10],[11]. We note that some RTL details are difficult (sometimes not found) to extract.

Our proposed approach generates less overall hardware consumption and it takes less processing cycles than the others (for comparatively similar ANN architectures and hardware implementation approaches). We note that other implementations usually implement activation functions other than ReLU: this is an area where our proposed architectures can improve and for which there are remarkable approaches [9][19].

F. *Selection of design paramters for given ANN applications*

Based on our results, we provide general guidelines on the selection of the optimal designs for given ANN applications.

First, numerical precision of the datapath does not play a major role in the final ANN accuracy. So, we can truncate the fractional bits in the datapath to improve resource utilization without affecting ANN accuracy. However, the weights/bias FX format greatly affects the numerical precision, and it should not be truncated further.

Second, the multiplier-less implementation should be avoided except for applications that need small ANNs (e.g.: 8-5-3, 3-4-1). For larger ANNs, the multiply-and-accumulate implementation should be chosen if resources are to be optimized. If resources are not a big concern, the fully parallel/pipelined implementation must be selected, as it allows for output data to be used at every new clock cycle.

Third, to take advantage of multi-stage pipelining, the number of neurons of the ANN layers should be relatively similar, otherwise the throughput will be dominated by the largest layer (e.g.: 'mlosh' ANN: 196-16-10).

Finally, for applications that require binary layers (e.g.: ANNs with binary input data, SNNs) and the proposed binary layer implementations are the most optimized: accumulator-based (to save resources), fully parallel/pipelined (to maximize throughput).

VI. Conclusions

We presented and successfully validated a set of hardware architectures for Artificial Neural Networks, both with real-valued inputs and with binary-valued inputs. The fixed-point implementations can handle varied numerical ranges with reasonable resource requirements, while providing the same levels of ANN accuracy. Overall, the multiplier-and-accumulate and the fully parallel/pipelined approaches are the most optimal designs, whereas the multiplier-less architecture could only provide acceptable results for small number of bits. These architecture implementations can be used in varied settings: as part of CNNs and SNNs.

References

[1] Nielsen, M.A., *Neural Networks and Deep Learning*; Determination Press: San Francisco, CA, USA, 2015.

[2] J. Misra, I. Saha, "Artificial neural networks in hardware: A survey of two decades of progress", *Neurocomputing*, vol. 74, pp. 239-255, 2010.

[3] Y. Ma, N. Suda, Y. Cao, S. Vrudhula, J. Seo, "ALAMO: FPGA acceleration of deep learning algorithms with a modularized RTL compiler", *Integration, the VLSI Journal*, vol. 62, pp. 14-23, 2018.

[4] A. R. Ormondi, J. Rajapakse, *FPGA Implementations of Neural Networks*, New York: Spring, 2006.

[5] S. Himavathi, D. Anitha, A. Muthuramalingam, "Feedforward neural network implementation in FPGA using Layer Multiplexing for Effective Resource Utilization", *IEEE Transactions on Neural Networks*, vol. 18, no. 3, pp. 880-888, May 2007.

[6] A. Dinu, M. Cirstea, S. Cirstea, "Direct Neural-Network Hardware-Implementation Algorithm", *IEEE Transactions on Industrial Electronics*, vol. 47, no. 5, pp. 1845-1848, May 2020.

[7] S. Hariprasath, T.N. Prakabar, "FPGA implementation of multilayer feed forward neural network architecture using VHDL", in *Proceedings of the 2012 International Conference on Computing, Communication and Applications*, Feb. 2012.

[8] K. Khalil, O. Eldash, B. Dey, A. Kumar, M. Bayoumi, "A Novel Reconfigurable Hardware Architecture of Neural Network", in *Proceedings of the IEEE 62nd International Midwest Symposium on Circuits and Systems (MWSCAS)*, Dallas, TX, Aug. 2019.

[9] A. Gomperts, A. Ukil, "Development and Implementation of Parameterized FPGA-Based General Purpose Neural Networks for Online Applications", *IEEE Transactions on Industrial Informatics*, vol. 7, no. 1, pp. 78-89, Feb. 2011.

[10] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li, "DeepBurning: Automatic generation of FPGA-based learning accelerators for the neural network family", in *Proceedings of the 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2016.

[11] N. B. Gaikwad, V. Tiwari, A. Keskar, N.C. Shivaprakash, "Heterogeneous Sensor data Analysis Using Efficient Artificial Neural Network on FPGA Based Edge Gateway", *KSII Transactions on Internet and Informational Systems (TIIS)*, vol. 13, no. 10, pp. 4865-4885, 2019.

[12] A. Shawahna, S.M. Sait, and A. El-Maleh, "FPGA-based accelerators of deep learning networks for learning and classification: A review", *IEEE Access*, vol. 7, pp. 7823-7859, 2019.

[13] Z. Li, A. Ren, J. Li, Q. Qiu, Y. Wang, and B. Yuan. "Dscnn: Hardware-oriented optimization for stochastic computing based deep convolutional neural networks", in *Proceedings of the IEEE 34th International Conference on Computer Design (ICCD)*, pp. 678-681, 2016.

[14] G. Maor, X. Zeng, Z. Wang, and Y. Hu. "An FPGA implementation of stochastic computing-based LSTM", in *Proceedings of the IEEE 37th International Conference on Computer Design (ICCD)*, pp. 38-46. 2019.

[15] X. Wei, C.H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs", in *Proceedings of the 54th Annual Design Automation Conference*, 2017.

[16] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong. "FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates", in *Proceedings fo the IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017.

[17] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. "Optimizing FPGA-based accelerator design for deep convolutional neural networks." *in Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*, pp. 161-170. 2015.

[18] B. Zamanlooy, M. Mirhassani, "Efficient VLSI implementation of neural networks with hyperbolic tangent activation function", *IEEE Transactions on VLSI systems*, vol. 22, no. 1, pp. 39-48, Jan. 2014.

[19] Y. Xie et al, "A Twofold Lookup Table Architecture for Efficient Approximation of Activation Functions", *IEEE Transactions on VLSI systems*, vol. 28, no. 12, pp. 2540-2550, Dec. 2020.

[20] C. Carranza, D. Llamocca, and M. Pattichis, "Fast and scalable computation of the forward and inverse discrete periodic radon transform", *IEEE Transactions on Image Processing*, vol. 25, no. 1, pp. 119-133, Jan. 2016.

[21] Tavanei, A., Ghodrati, M., Kheradpisheh, S.R., Masquelier, T., Maida, AS., "Deep Learning in Spiking Neural Networks", *Neural Networks*, 2019, 111, pp. 47-63.

[22] O. Weng. "Neural Network Quantization for Efficient Inference: A Survey." *arXiv preprint arXiv:2112.06126* (2021).

[23] M. Nagel, M. Fournarakis, R.A. Amjad, Y. Bondarenko, M. Van Baalen, and T. Blankevoort. "A white paper on neural network quantization." *arXiv preprint arXiv:2106.08295* (2021).

[24] M. Nagel, R.A. Amjad, M. Van Baalen, C. Louizos, and T. Blankevoort. "Up or down? adaptive rounding for post-training quantization", in *Proceedings of the International Conference on Machine Learning*, pp. 7197-7206, 2020.

[25] F. Li, B. Zhang, and B. Liu. "Ternary weight networks." *arXiv preprint arXiv:1605.04711* (2016).

[26] Y. Nahshan, B. Chmiel, C. Baskin, E. Zheltonozhskii, R. Banner, A.M. Bronstein, and A. Mendelson. "Loss aware post-training quantization." *Machine Learning*, vol. 110, no. 11 (2021): 3245-3262.

[27] L. Deng, “The MNIST database of handwritten digit images for machine learning research [best of web]”, *IEEE Signal Processing Magazine*, vol. 29, pp. 141-142, 2012.

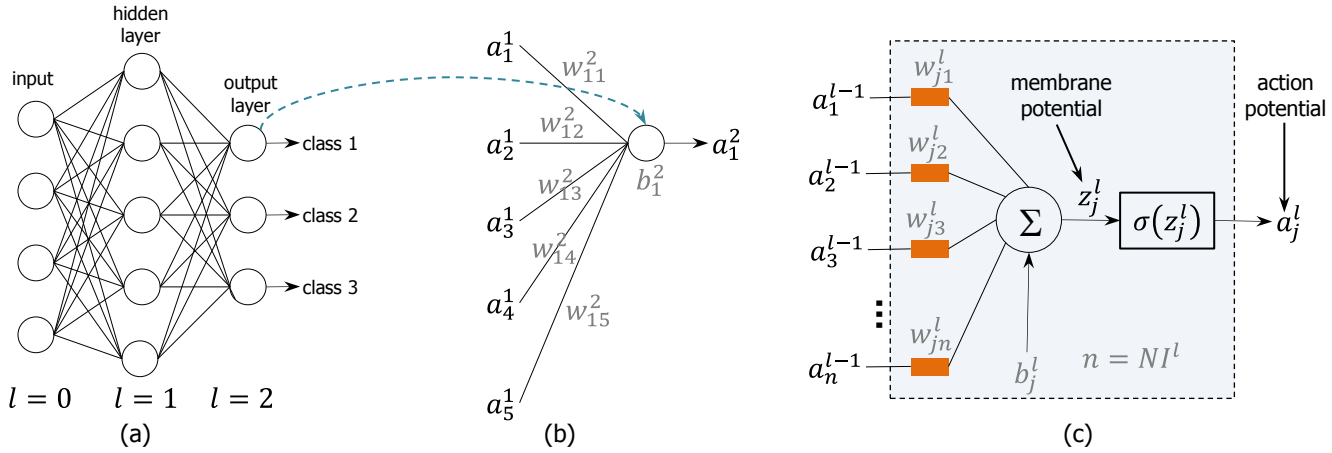


Figure 1. (a) ANN with 3 layers ($L = 3$). (b) First neuron (index '1') in layer $l = 2$. (c) Artificial neuron model. The membrane potential is a sum of products (input activations by weights) to which a bias term is added. j represents the neuron index in layer l. The input activations come from a previous layer ($l - 1$).

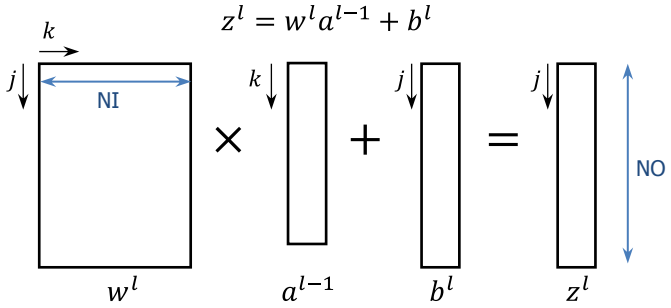


Figure 2. Matrix computation for membrane potentials (z^l) in layer l. The index k is for the neurons in layer $l - 1$, while the index j is for the neurons in layer l. The weight matrix w^l has NO rows by NI columns, the action potential vector a^{l-1} has NI rows, and the bias vector b^l has NO rows. The membrane potential z^l and the action potential a^l have NO rows.

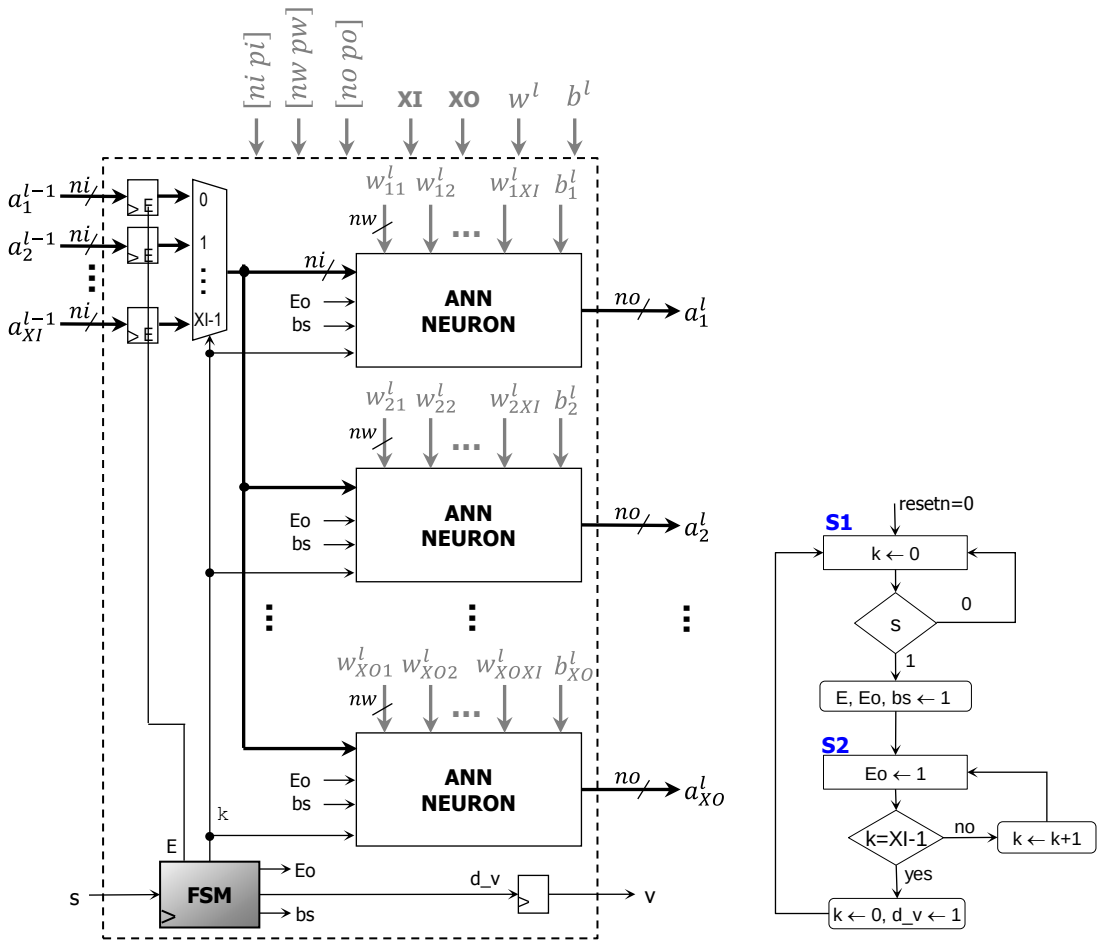


Figure 3. ANN Layer for real-valued inputs: Multiply-and-add architecture. The design parameters are indicated. XI: number of inputs, XO: number of outputs. Input data: $\vec{a}^{l-1} = [a_1^{l-1}\ \ a_2^{l-1}\ ...\ a_{XI}^{l-1}]^T$. Output Data: $\vec{a}^{l} = [a_1^{l}\ \ a_2^{l}\ ...\ a_{XO}^{l}]^T$. Weight matrix: w^l, bias vector: b^l. The FX format of the inputs, weights/biases, and output is specified as $[ni\ pi]$, $[nw\ pw]$, and $[no\ po]$ respectively. The FSM ensures input data is captured in registers and then processed sequentially.

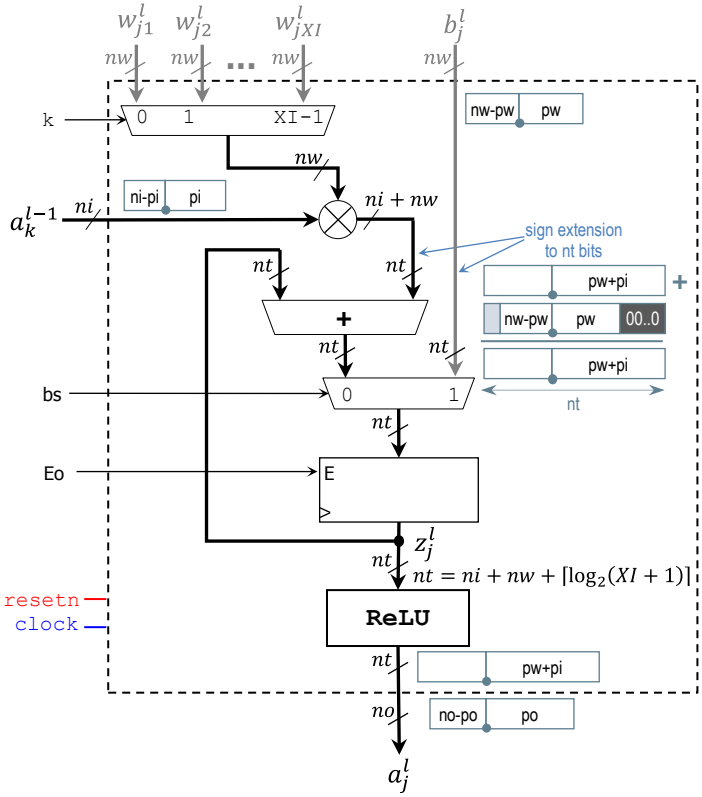


Figure 4. Neuron architecture for ANN Layer in Fig. 3. Each neuron only receives row $\boldsymbol{j}$ of the weight matrix $\boldsymbol{w^l}$, as well as the element $\boldsymbol{j}$ in the bias vector $\boldsymbol{b^l}$. As there are $\boldsymbol{XO}$ neurons in a layer, $\boldsymbol{j}$ is the index ($\boldsymbol{j = 1, ...XO}$).

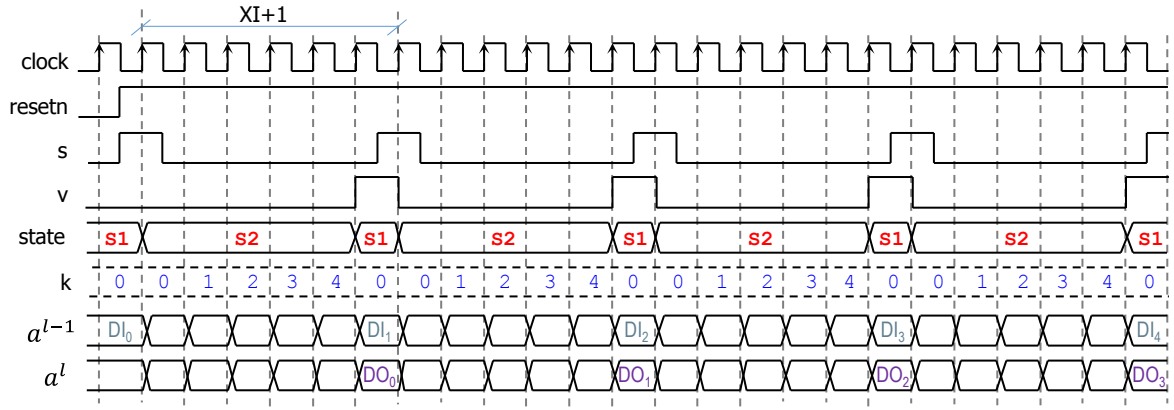


Figure 5. Timing diagram for the ANN Layer in Fig. 3. $XI = 5$. The input data $\vec{a}^{l-1}$ is fed to the ANN Layer and the result $\vec{a}^{l}$ appears on the output after $XI + 1$ cycles. We can feed another input sample right after it.

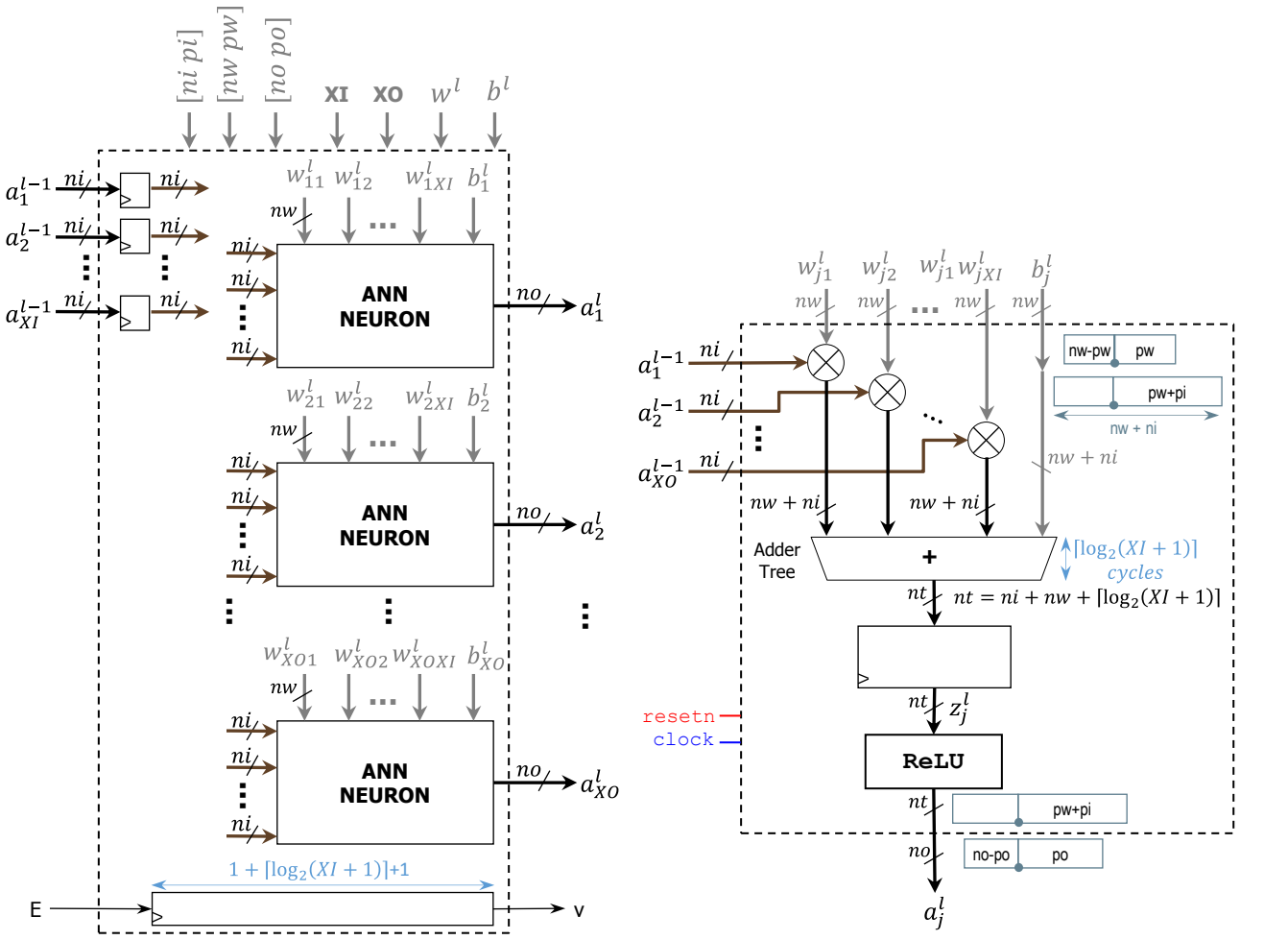


Figure 6. ANN Layer for real-valued inputs: Fully parallel/pipelined architecture. XI: number of inputs, XO: number of outputs. Input data: $\vec{a}^{l-1} = [a_1^{l-1}\ a_2^{l-1}\ \dots\ a_{XI}^{l-1}]^T$. Output Data: $\vec{a}^{l} = [a_1^{l}\ a_2^{l}\ \dots\ a_{XO}^{l}]^T$. Weight matrix: w^l, bias vector: b^l. The FX format of the inputs, weights/biases, and output is specified as $[ni\ pi]$, $[nw\ pw]$, and $[no\ po]$ respectively. The architecture of neuron j ($j = 1, \dots, XO$) is also shown. Each neuron only receives row j of the weight matrix w^l, as well as the element j in the bias vector b^l.

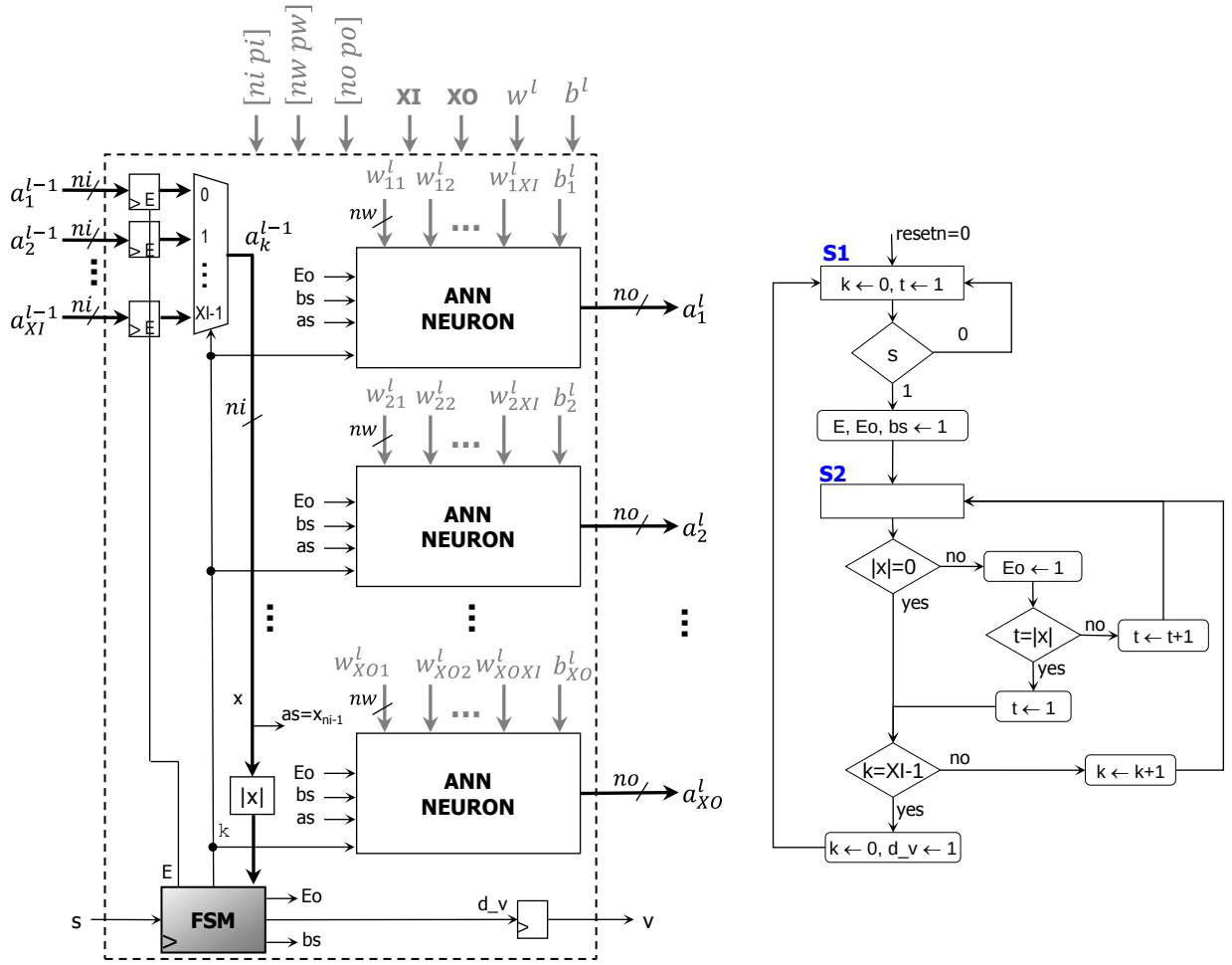


Figure 7. ANN Layer for real-valued inputs: Multiplier-less architecture. XI: number of inputs, XO: number of outputs. Input data: $\vec{a}^{l-1} = [a_1^{l-1}\ a_2^{l-1}\ ...\ a_{XI}^{l-1}]^T$. Output Data: $\vec{a}^{l} = [a_1^{l}\ a_2^{l}\ ...\ a_{XO}^{l}]^T$. Weight matrix: w^l, bias vector: b^l. The FX format of the inputs, weights/biases, and output is specified as $[ni\ pi]$, $[nw\ pw]$, and $[no\ po]$ respectively. The FSM orchestrates data capture in registers and the neuron output computations.

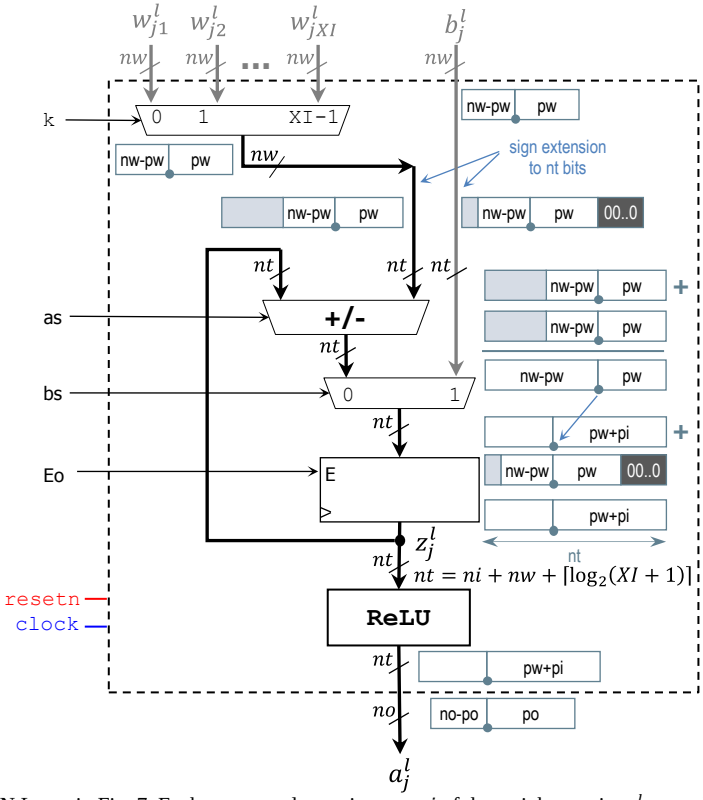


Figure 8. Neuron architecture for ANN Layer in Fig. 7. Each neuron only receives row j of the weight matrix w^l, as well as the element j in the bias vector b^l. As there are XO neurons in a layer, j is the index ($j = 1, ...XO$).

TABLE I. ANN LAYER HARDWARE FOR REAL-VALUED ($nt = ni + nw + \lceil log_2(XI+1) \rceil$) AND BINARY-VALUED ($nt = nw + \lceil log_2(XI+1) \rceil$) INPUTS: FEATURES.

	Architecture	Latency (cycles)	Min. # of cycles between input samples	Hardware per neuron	Largest output FX format	Comments
Real-valued Layer	Multiply-and-accumulate	$XI+1$	$XI+1$	MUX XI-to-1, multiplier, MUX 2-to-1 nt-bit adder, register, and ReLU	$[nt\ pw + pi]$	Multipliers: one constant operand.
	Fully Parallel/ Pipelined	$\lceil \log_2(\mathrm{XI}+1) \rceil + 2$	1	XI multipliers, adder tree (XI+1 inputs) nt-bit register and ReLU		
	Multiplier-less	$\sum_{k=0}^{XI-1} \lvert a_k^{l-1} \rvert + 1$	$\sum_{k=0}^{XI-1} \lvert a_k^{l-1} \rvert + 1$	MUX XI-to-1, MUX 2-to-1. nt-bit adder/subtractor, register, ReLU		Latency is non-deterministic
Binary Layer	Accumulator-based	$XI+1$	$XI+1$	MUX XI-to-1, MUX 2-to-1. nt-bit adder, register, and ReLU.	$[nt\ pw]$	No multipliers
	Fully Parallel/ Pipelined	$\lceil \log_2(XI+1) \rceil + 2$	1	XI MUXs 2-to-1, XI+1 input adder tree nt-bit register and ReLU		

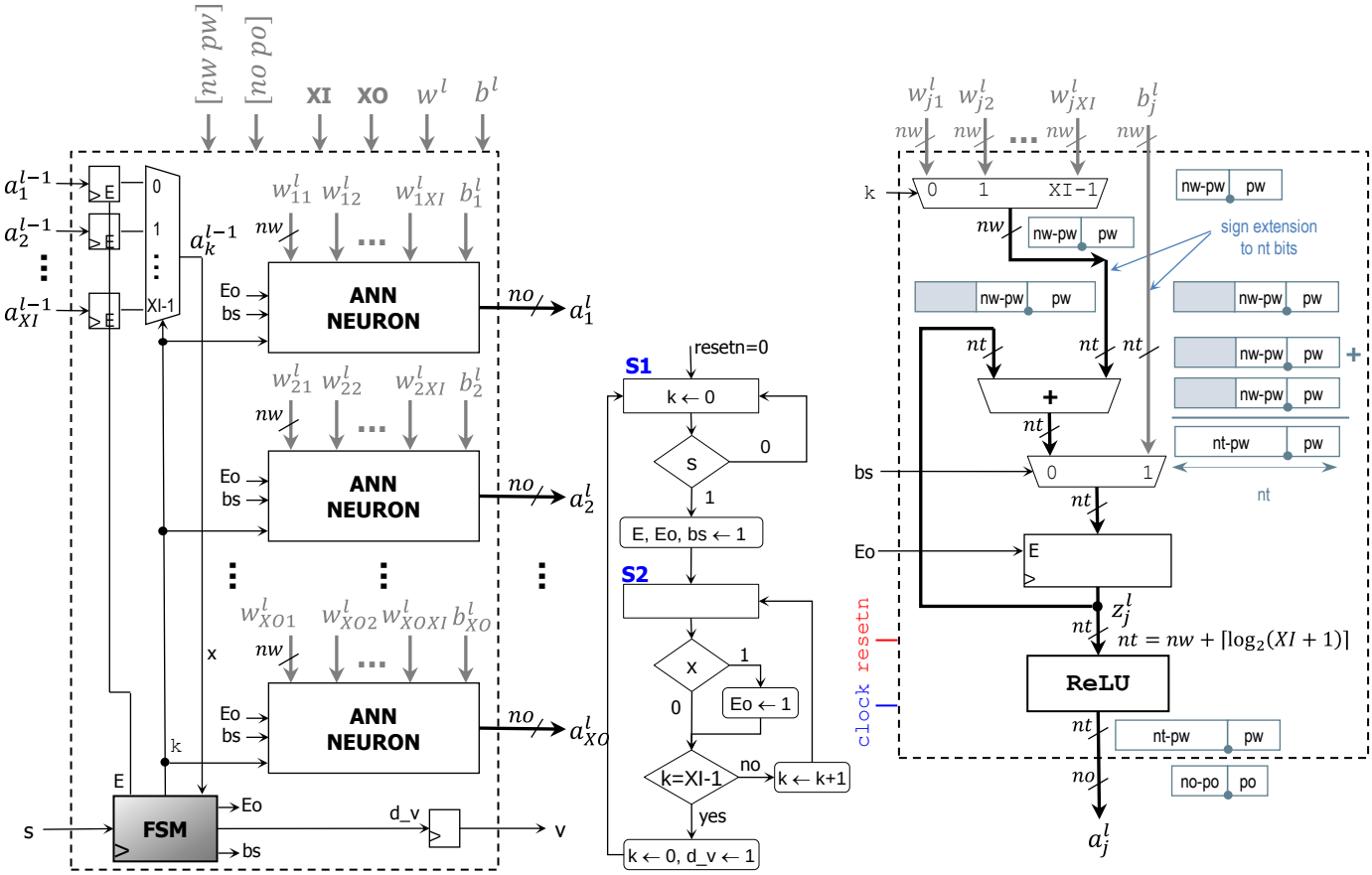


Figure 9. Binary Layer: accumulator-based architecture. XI inputs, XO outputs. Input data: $\vec{a}^{l-1} = [a_1^{l-1}\ \ a_2^{l-1}\ \ldots\ a_{XI}^{l-1}]^T$. Output Data: $\vec{a}^{l} = [a_1^{l}\ \ a_2^{l}\ \ldots\ a_{XO}^{l}]^T$. Weight matrix: w^l, bias vector: b^l. The FX format of the weights/biases and output is specified as $[nw\ pw]$, and $[no\ po]$ respectively. The FSM orchestrates data capture in registers and the neuron output computations. The architecture of neuron j ($j = 1, \ldots, XO$) is also shown. Each neuron only receives row j of the weight matrix w^l, as well as the element j in the bias vector b^l.

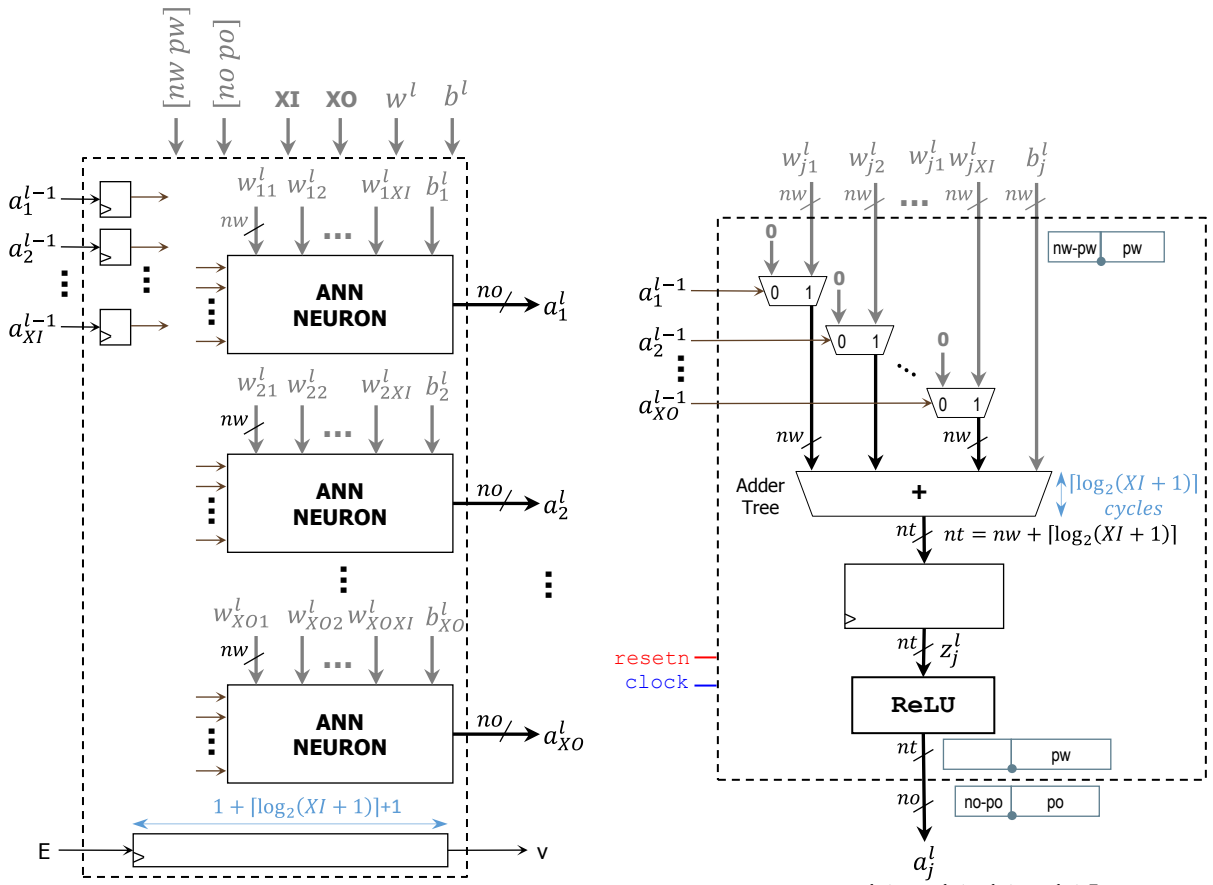


Figure 10. Binary Layer: Fully parallel/pipelined architecture. XI inputs, XO outputs. Input data: $\vec{a}^{l-1} = [a_1^{l-1}\ a_2^{l-1} \dots a_{XI}^{l-1}]^T$. Output Data: $\vec{a}^l = [a_1^l\ a_2^l \dots a_{XO}^l]^T$. Weight matrix: w^l, bias vector: b^l. The FX format of the weights/biases and output is specified as $[nw\ pw]$, and $[no\ po]$ respectively. The architecture of neuron j ($j = 1, \dots, XO$) is also shown. Each neuron only receives row j of the weight matrix w^l, as well as the element j in the bias vector b^l.

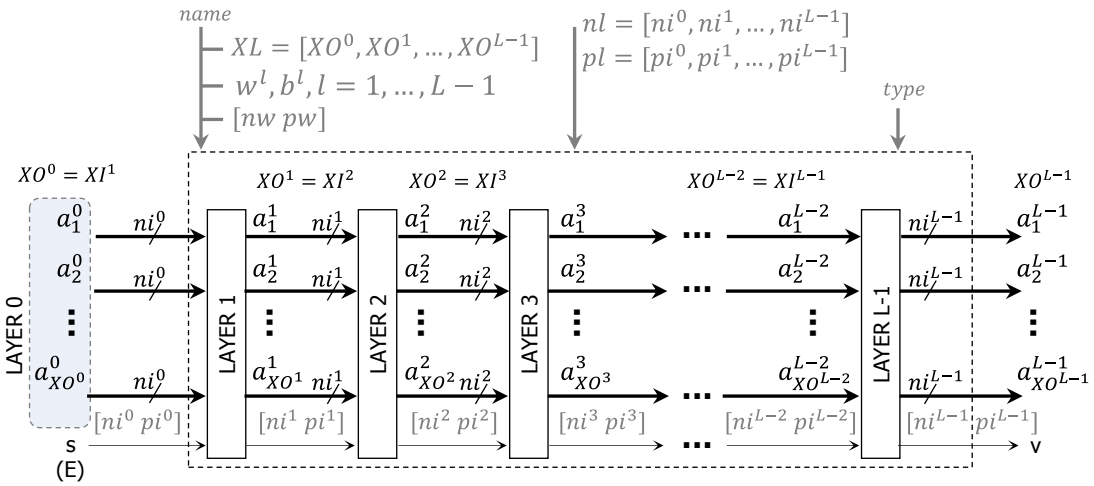


Figure 11. ANN with L layers. I/Os are indicated for every Layer. Layer 0 refers to input data. $XI^l = XO^{l-1}$ for $l = 1, \dots, L-1$. The parameter '*name*' specifies the weight matrix and bias vector for every layer. The parameter 'type' has three choices: 'multiply-and-add', 'fully parallel/pipelined', 'multiplier-less'.

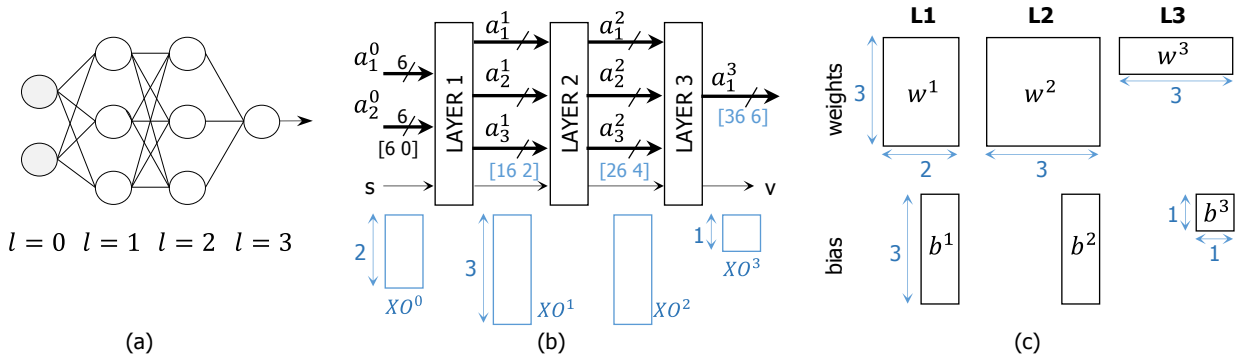


Figure 12. ANN '*test1*'. (a) ANN structure, $L = 4$ layers. (b) Bit-width (and FX format for full accuracy) per layer. Size of I/Os is depicted. (c) Weights and biases for every layer ($l \geq 1$).

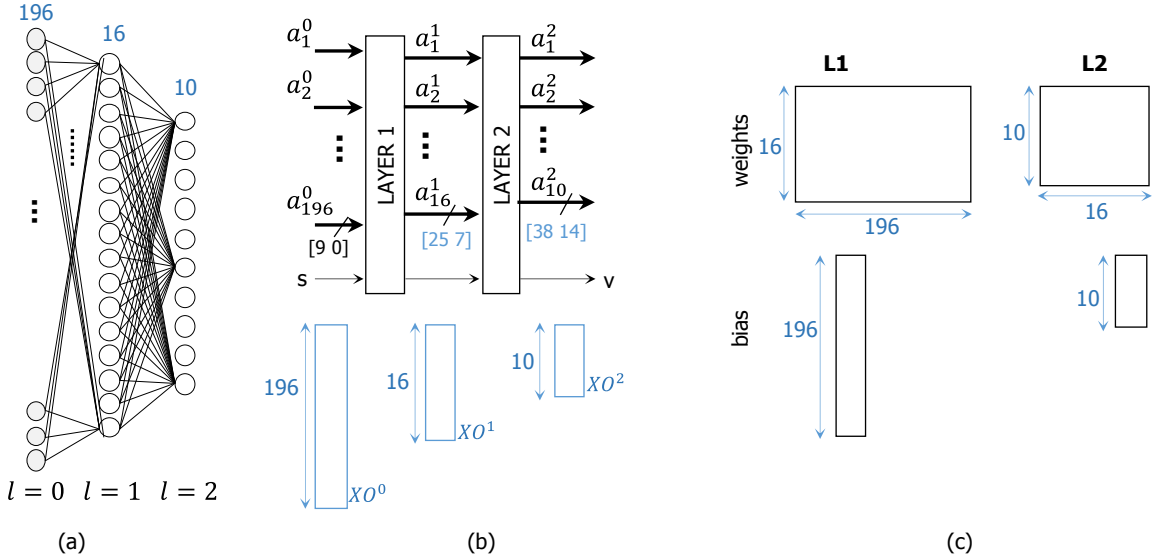
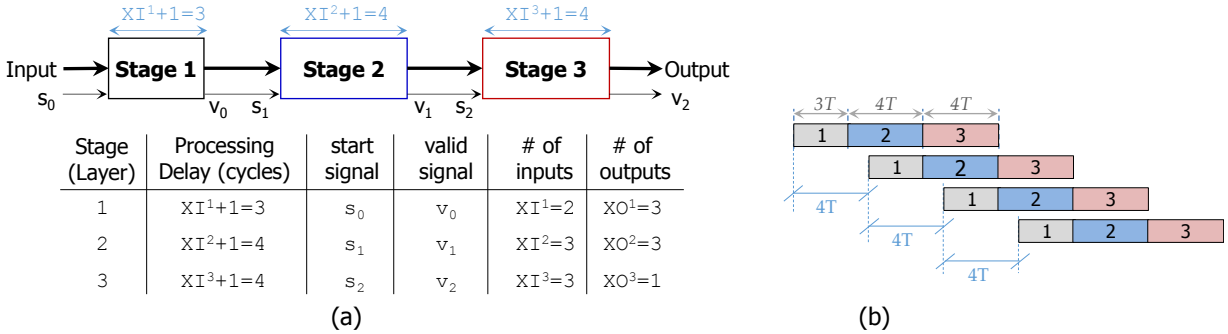


Figure 13. ANN '*mlosh*'. (a) ANN structure, $L = 3$ layers. (b) Bit-width (and FX format for full accuracy) per layer. Size of I/Os is depicted. (c) Weights and biases for every layer ($l \geq 1$).



Stage (Layer)	Processing Delay (cycles)	start signal	valid signal	# of inputs	# of outputs
1	$XI^1+1=3$	s_0	v_0	$XI^1=2$	$XO^1=3$
2	$XI^2+1=4$	s_1	v_1	$XI^2=3$	$XO^2=3$
3	$XI^3+1=4$	s_2	v_2	$XI^3=3$	$XO^3=1$

Figure 14. '*test1*' ANN: multiply-and-add approach. (a) Layers as stages along with their processing delays. An input sample is fed when the corresponding '*s*' signal is asserted. (b) Pipelined approach that illustrates why $F = max(3,4,4) = 4$.

TABLE IV. LATENCY OF ANN HARDWARE (REAL-VALUED INPUTS). TO FEED DATA ONTO THE ANNS, A PIPELINED APPROACH IS USED.

Architecture	Latency (cycles)	Min. # of cycles between input samples	Comments
Multiply-and-accumulate	$\sum_{l=1}^{L-1}(XI^l+1)$	$F=max(p_1,p_2,\dots,p_{L-1}),$ $p_l=XI^l+1$	Multipliers: one constant operand.
Fully Parallel/ Pipelined	$\sum_{l=1}^{L-1}(\lceil\log_2(XI^l+1)\rceil+2)$	1	
Multiplier-less	$\sum_{l=1}^{L-1}\left(\sum_{k=0}^{XI^l-1}\lvert a_k^{l-1}\rvert+1\right)$	$F=max(p_1,p_2,\dots,p_{L-1}),$ $p_l=\sum_{k=0}^{XI^l-1}\lvert a_k^{l-1}\rvert+1$	Latency is non-deterministic

TABLE XIII. COMPARISON OF DIFFERENT ANN ARCHITECTURES. VIRTEX-4: 4-INPUT LUTS, 18X18-BIT DSP MULTIPLIERS. VIRTEX-6, VIRTEX-7, ZYNQ-7000: 6-INPUT LUTS 25X18-BIT DSP MULTIPLIERS.

	Proposed			Gomperts et al. [9]	Himavathi et al. [5]	Wang et al. [10]	Gaikwad et al. [11]
Architecture	Multiply-and-accumulate	Fully parallel/ pipelined	Multiplier-less	Multiply-and-accumulate	Multiply-and-accumulate	Fully parallel	Multiply-and-accumulate
Activation Function	ReLU			sigmoid: LUT with interpolation	sigmoid/ReLU	ReLU	sigmoid
Numerical format	Fixed Point (variable, grows with the datapath)			Fixed Point (3 fractional bits, variable for activation function)	Fixed Point [29 16]	Fixed Point	Fixed-Point 8-bit precision
ANN	196-16-10 [9 0]: input data, [32 8] output data			10-3-1	8-5-5-3	4-layer	5-4-2
Device and frequency	Artix-7 XC7A100T @ 100 MHz			Virtex-5 XCVSX50T	XCV400 @ 73 MHz	Zynq-7045 @ 100 MHz	Artix-7 XC7A35T @ 10 MHz
FPGA Resources	2978 FFs 3372 LUTs 10 DSPs	76255 FFs 74314 LUTs 115 DSPs	3014 FFs 2561 LUTs	2243 FFs 8043 LUTs 246 DSP48E1s	2863 Slices (~5600 LUTs, 5600 FFs)	3621 LUTs 2532 FFs	1295 FFs 3244 LUTs 79 DSPs
Processing cycles	214 cycles	17 cycles	4.6×10^{14} cycles	-	186 cycles	-	31 us (~310 cycles)
Notes	self-contained VHDL design			Hardware generated by high-level platform	Only largest layer is implemented (and reused)	Hardware generated by high-level platform.	Hardware generated by Xilinx SysGen

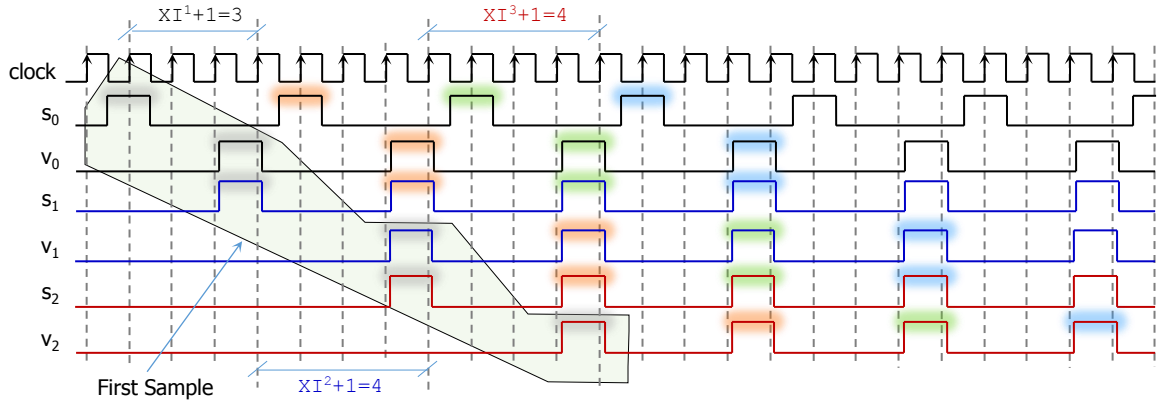


Figure 15. '*test1*' ANN: multiply-and-add approach. Timing diagram that illustrates how we can feed data every $F=4$ cycles. Note how we can assert the '*s*' signal of Layer 1 (s_0) every $F=4$ clock cycles. The behavior of the '*s*' and '*v*' signals for every layer is depicted.