November 11, 2019

AKV Artificial Neural Network Co-Processing Core
Microarchitecture Specification

Prepared by: Vladimir Milicevic {Vladimir.Milicevic@carleton.ca}

1. Introducing the AKV Co-Processor

The AKV is a highly integrated machine learning co-processing core that implements a systolic array architecture. This core targets machine learning applications such as neural network back-propagation training and real-time neural network processing.

The AKV Co-Processor contains two functional blocks: i.) a Host Controller and ii.) a scaleable, reconfigurable Neural Fabric. The Neural Fabric is a collection of "Layers". Neuron Units are the atomic units which comprise the Neural Fabric, consisting of a Fused-Multiply-Add (FMA) unit, scratchpad registers and activation function logic. Neuron Units within a Layer are interconnected by a non-blocking network-on-chip (NoC). The routing algorithm for this non-blocking network is implemented by a Routing Engine, scheduling is performed by an Arbiter.

The programmable Host Controller inside the core is capable of maintaining a resizable Neural Fabric given an appropriate amount of memory. The main tasks of the Host Controller include Direct Memory Access (DMA), layer configuration and training supervision. This allows the AKV to target a wide range of Field-Programmable Gate Array (FPGA) platforms with different resource availability. FPGAs implement the Host Controller, then the Neural Fabric is sized proportionately to the remaining resources. This system architecture is favourable when targeting Application-Specific Integrated Circuits (ASICs), allowing the core to be upscaled as technologies mature.

2. Functional Description

The features of the AKV Co-Processor core include:

- Minimum; 8 Neuron Units per layer with an 8-port non-blocking interface; 1 port per Neuron Unit.
- Primary (Level 1) N-way Set Associative Cache per layer, shared by all 8 Neuron Units in that Nth layer.
- Secondary (Level 2) Cache, used by Host Controller.
- IEEE 754 compliant Fused-Multiply-Accumulate (FMA) unit, implemented in all Neuron Units.
- Direct Memory Access (DMA) engine to access Host CPU memory.
- 8-port non-blocking Beneš interconnect network, Routing Engine and Arbiter.

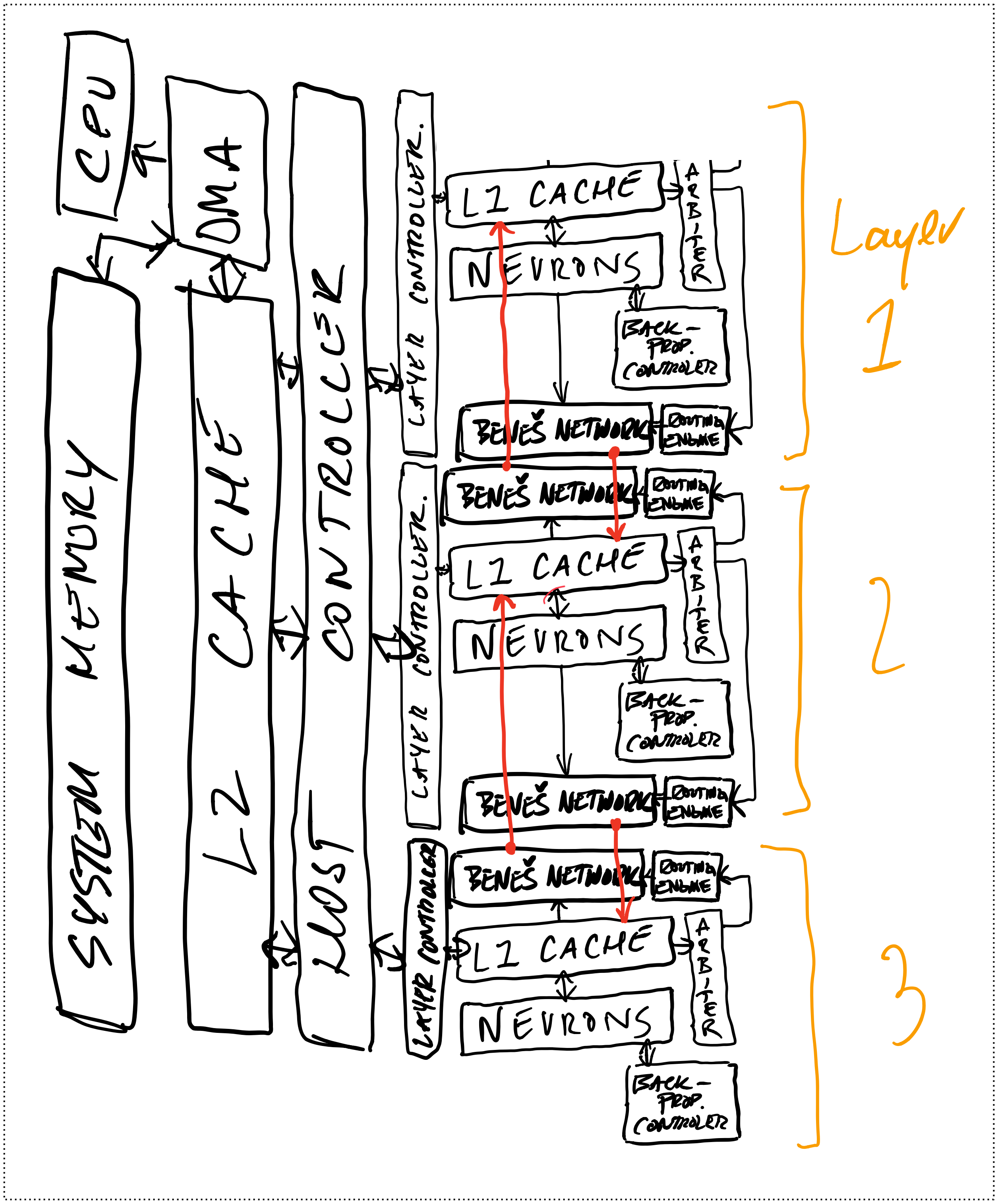


Figure 1: AKV Architecture Example, Host Controller and 3-Layer Neural Network Fabric

3. AKV Co-Processor Block Overview

3.1. Neuron Unit

The Neuron Unit integrates a FMA (Fused-Multipy-Accumulate) block with activation function logic. ReLU operation is described in section 5.2. The Neurons have access to their partition of the L1 Cache. There is a small amount of scratchpad memory for storing intermediate data. The Neuron Unit is capable of bypassing certain FMA logic when inputs are zero.

Capabilities which allow the FMA to anticipate truncated results approaching zero are in development as future work, this would allow the FMA to be bypassed if it anticipates an unneeded MAC operation, i.e.; MAC result strongly trends towards zero.

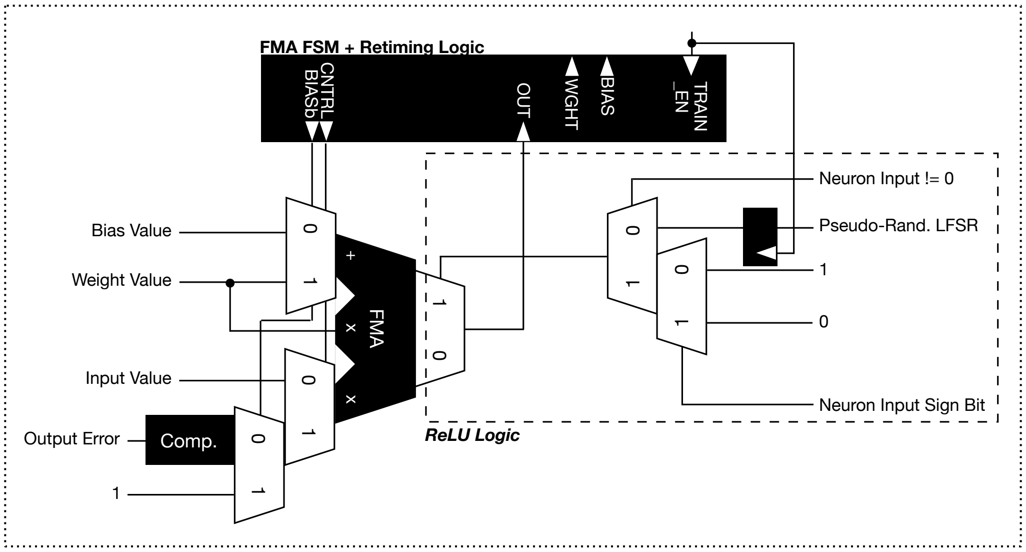


Figure 1: The Atomic Processing Unit, The Neuron Unit

3.2. ReLU Activation Function

The ReLU activation function provides a low computational-cost slope calculation:

$0,$	Neuron Input < 0	(Eq. 1)
$Rand(0,1),$	Neuron Input $= 0$	(Eq. 2)
$1,$	Neuron Input > 0	(Eq. 3)

3.3. L1 Cache

There is an L1 Cache included inside each layer. This is an N-way set associative cache which is a subset of the L2 Cache, where N is the number of layers present in the neural network. Working values for each of the 8 neuron units are stored inside the L1 Cache including current neuron output value, current neuron bias value. Each L1 Cache is comprised of 320B of memory.

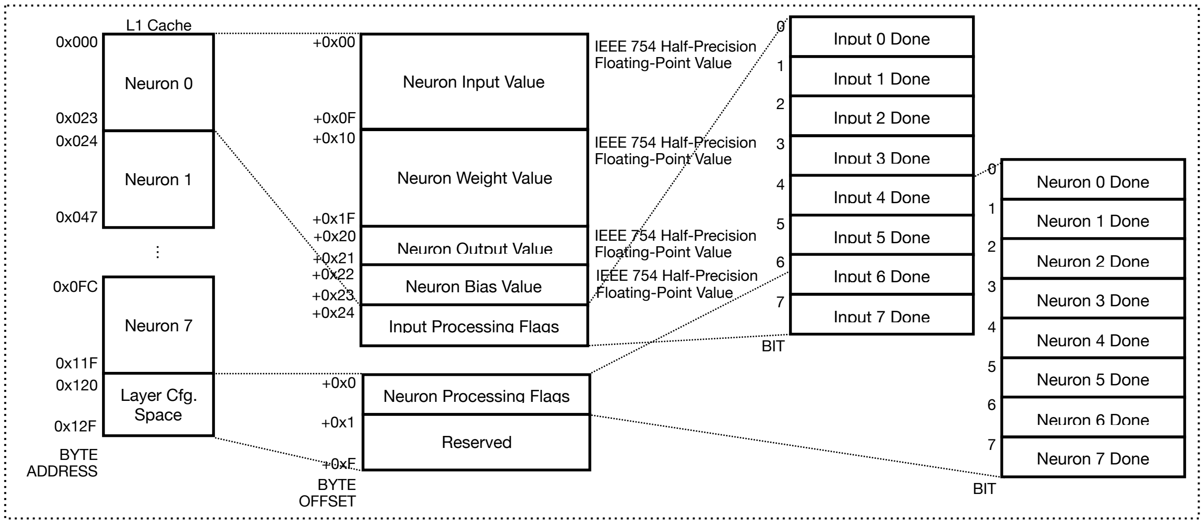


Figure 2: L1 Cache Memory Map

3.4. Layer Controller

The Layer Controller is a wrapper providing control signals and retiming to internal logic. The Layer Controller also provides extended capabilities to the Host Controller through the DATA bus, providing L1 Cache access to the Host Controller. During neural network training, the AKV can be put into a debug mode, whereby the Layer Controller halts after each weight recalculation, allowing the Host Controller to be programmed to review and modify weights of individual neurons in a layer. Programming the Layer Controller configuration is done by the Host Controller via the L1 Cache contents.

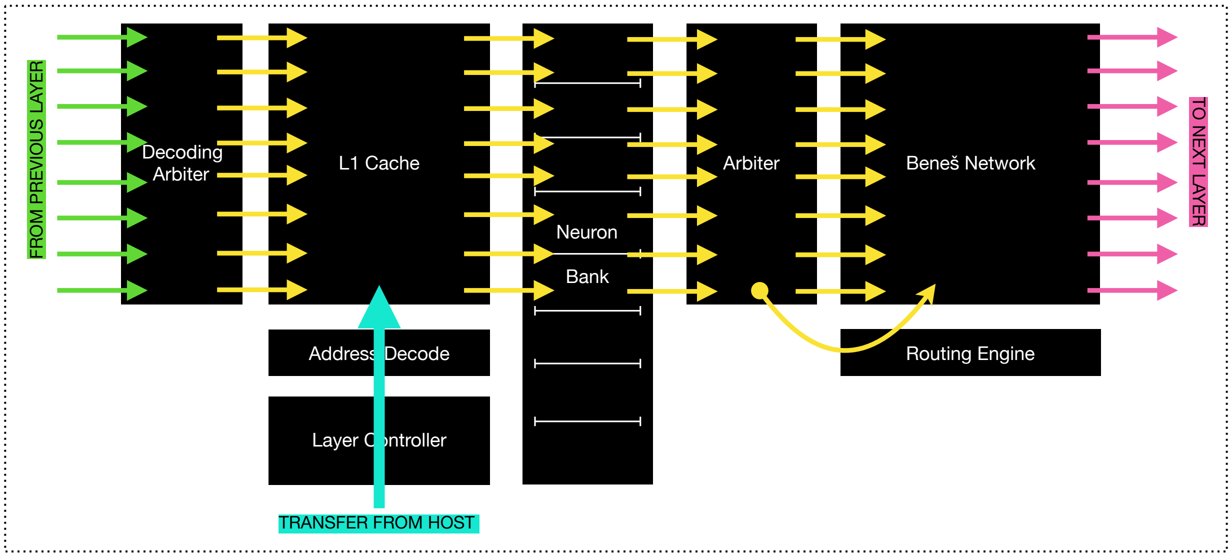


Figure 3: Layer Controller Wrapper

3.5. Host Controller

The Host Controller is a programmable block based on the AKV Instruction Set. The Host Controller contains the DMA engine and L2 Cache, as well as connections to each individual layer. The Host Controller is responsible for configuring each individual layer and supervising network training. The Host Controller is able to target System Memory using the Direct Memory Access engine to load data into the L2 Cache.

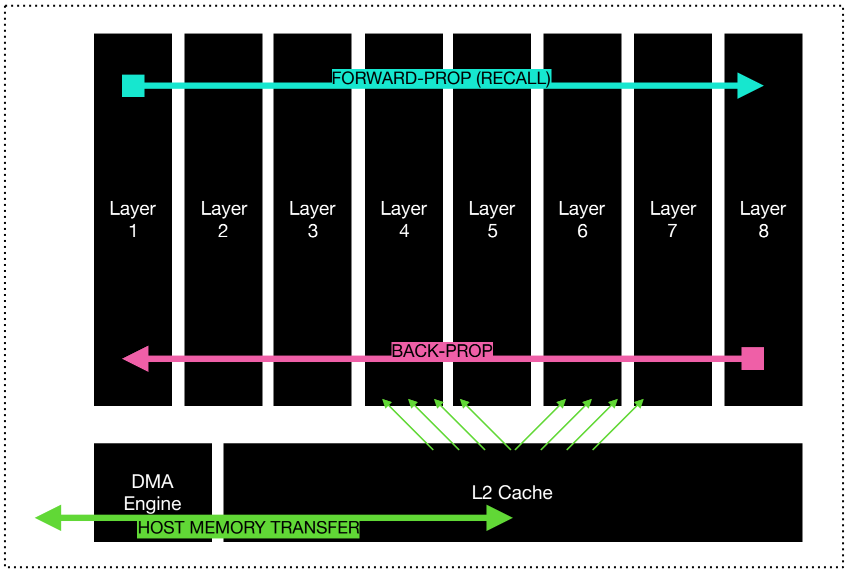


Figure 4: Host Controller Wrapper

3.6. Host Controller Layer Controller Interface

The 147b asynchronous interface is composed of a 128b bidirectional DATA bus, S_RDY and D_RDY handshake signals, a 16b address bus, and a WR write signal. Data encoding and decoding is done on a per-address basis. Logic accessing the memory must be able to evaluate the memory contents on its own. Maximum bus bandwidth is achieved when accessing 128b of data listed in adjacent memory locations.

3.7. L2 Cache

L2 Cache contains each layers training information, network configuration, and input/output values. The Host Controller is responsible for maintaining L1/L2 Cache coherency. In practice, the Host Controller will likely synchronize caches after a complete epoch has elapsed.

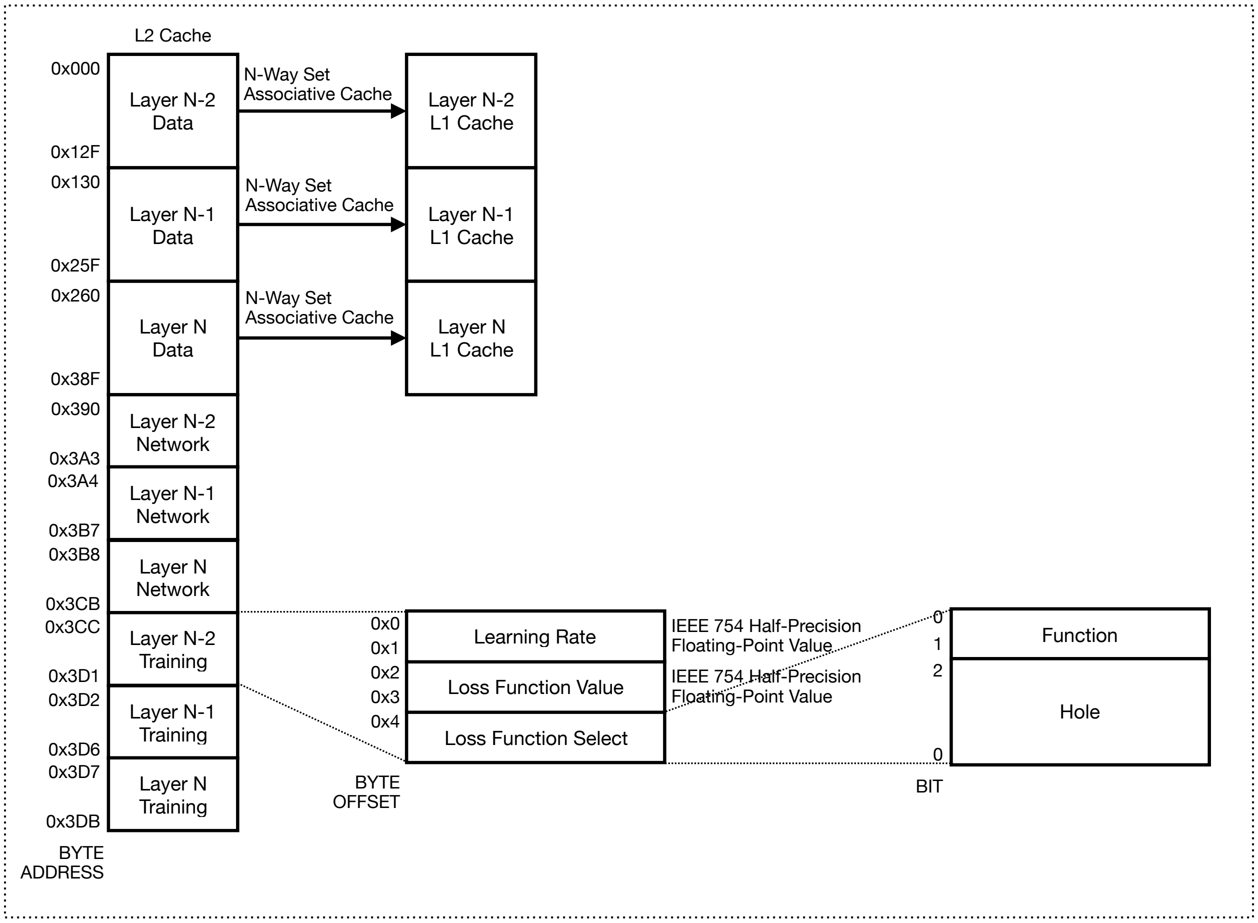


Figure 2: L2 Cache Memory Map

When data is loaded into the L2 Cache from the Host Memory by means of the DMA Engine, the Host Controller decodes the layer configuration, weights, initial values, and transfers this to the Layer Controllers. Rapid asynchronous transfer of decoded L2 Cache data to the constituent L1 Caches is provided by individual 147b interfaces to each Layer Controller. The Layer Controller is responsible for decoding data based on which address is being targeted, then partitioning the L2 Cache contents it into the L1 Cache for each Neuron to access.

To transfer the current Network configuration to the Host Memory, the Host Controller updates the L2 cache by accessing the L1 Cache contents through the Layer Controller. The Host Controller then programs the DMA Engine to transfer the L2 Cache contents to the Host Memory.

4. Interfacing

4.1. Routing Engine L1 Cache Interface

The Routing Engine accesses the L1 Cache to view the current layers interconnect between layers. If the layer in which the Routing Engine exists is Layer N: the L1 Cache contains the interconnect information between Layer N -> Layer N-1 for back propagation, and interconnect information between Layer N -> Layer N+1 for forward propagation. The interconnect data for one neuron is stored as a 32b value in the L1 Cache. The 32b value is a list containing 8x4b fields containing the Neuron's destination Neurons. Each 4b field contains the destination neuron number. If the value in the nibble is 4'd0, the Neuron is connected to Neuron 0 of the destination Layer. Similarly, if the value is 4'd7, Neuron 0 is connected to Neuron 7 of the destination layer. Value 4'hF is signals no connection to the destination Layer.

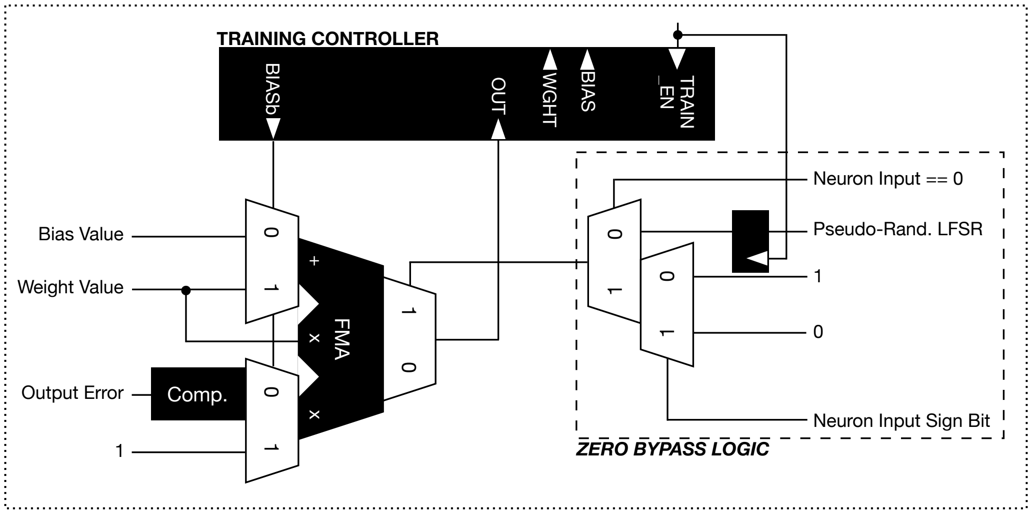
4.2. Layer Controller L1 Cache Interface

When data is transferred from the Host Controller into the Layer Controller it is written into local L1 Cache. Any writes to the Neuron inputs (input value, bias, weight) clear the Processing Done sticky bit in the L1 Cache. The Arbiter snoops the L1 Cache line containing the Neuron Done sticky bits. When a sticky bit is set, the Arbiter buffers the transmission request until a connection is available. When a port in the network is available, the output contents from the L1 Cache are tagged and the Arbiter communicates with the Routing Engine to transfer data through the switching fabric in round-robin fashion. The Arbiter avoids head-of-line blocking by scanning the transmit buffer contents prior to servicing another neuron's buffer. Pairing round-robin arbitration and the head-of-line bypass logic allows for fair network used and avoids greed. Downstream, (N+1 layer), the adjacent Layer Controller decodes tagged Neuron outputs from the switch matrix. It will read it's L1 Cache to find the transmitting layer (N-1 Layer) Network Configuration, and will copy the incoming data into the appropriate L1 Cache line. This begins the process of recalculating the outputs for the adjacent layer.

5. Back Propagation

Back propagation may begin when forward-propagated calculations are complete. The Host Controller will poll and update the L2 Cache network output values from the final layer (Layer L), calculate the output error, and broadcast these values to each layer controller. Back propagation calculations begin when the Host Controller sets the Training bit in the individual layer controls. Beginning with the final layer (layer L), and enable training on each subsequent layer (Layer L-1, L-2…) by setting the training enable flag in the L1 Cache. The L-1 layer will begin re-calculating weights and bias' based on the network output values. For each output vector there will need to be a complete round of recalculations. That is to say, the weight and bias for each Neuron will be re-calculated $N_{L_{out}}$ times.

Figure 5: Weight and Bias Recalculation Logic



6. Layer Interconnect

6.1. Beneš Switching Fabric

The implemented switching fabric interconnecting each layer consists of 20 crossbar switches in a Beneš configuration, shown in Figure 6. The crossbars may be crossed: $[I_{top}, I_{bot}] \rightarrow [O_{bot}, O_{top}]$ or the switches may be barred: $[I_{top}, I_{bot}] \rightarrow [O_{top}, O_{bot}]$. When controlled by the Routing Engine the 8-port Beneš network operates as a nonblocking minimal spanning switch. There are 8 input ports connected to 4 two-port crossbar switches in the ingress stage, ($N = 8$, $n = m = 2$). Each ingress switch outlets into 2 ($r = 2$) similar crossbar switches. Each ingress switch has a dedicated connection to a crossbar switch in the intermediate ("middle") switching layer. With $m \geq n$, we realize a rearrangeable nonblocking network. There is no buffering — this is strictly a circuit-switched network.

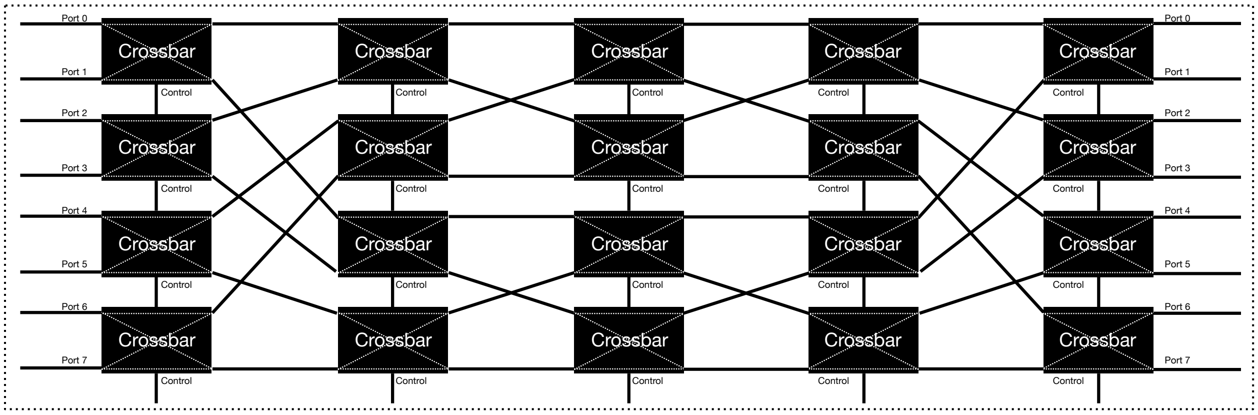


Figure 6: Inter-Layer Beneš Switching Fabric

Scaling the switch network can be done using the following resources:

$n \rightarrow$ Crossbar input ports

$m \rightarrow$ Crossbar output ports

$R \rightarrow$ Number of crossbars per-layer.

$N = nR \rightarrow$ Number of input ports (Eq. 4)

Number of stages:

$2log_2N - 1$ (Eq. 5)

Total Crossbar Switches:

$Nlog_2N \ - (N \ / \ 2)$ (Eq. 6)

The configuration used within AKV:

$N = nR = 2x4 = 8$

Number of stages:

$2log_2(8) - 1 = 5$

Total Crossbar Switches:

$(8)log_2(8) \ - (8 \ / \ 2) = 20$

6.2. Interconnect Scheme

With 5 crossbar stages, there are 4 interconnect stages accompanied by 4 interconnect schemes: the perfect shuffle, the reverse perfect shuffle, the butterfly, the reverse butterfly.

We define a set P of "ports" such that n is the total number of ports: $P = [p_{n-1}, p_{n-2}, \ldots, p_1, p_0]$.

When $n = 8$, as is the case with the AKV Beneš network, we find the set $P = [p_7, p_6, \ldots, p_1, p_0]$.

Using the base 2 numeral system: $P = [p_{111_2}, p_{110_2}, \ldots, p_{001_2}, p_{000_2}]$

6.2.1. The Perfect Shuffle

The perfect shuffle interconnect scheme is described by the sigma function:

$$\sigma(P) = [p_{111_2<1}, p_{110_2<1}, \ldots, p_{001_2<1}, p_{000_2<1}] \quad \text{(Eq. 7)}$$

where < 1 describes a logical shift left.

Given:

$$P = [p_7, p_6, p_5, p_4, p_3, p_2, p_1, p_0]$$

we find that:

$$P_{base_2} = [p_{111_2}, p_{110_2}, p_{101_2}, p_{100_2}, p_{011_2}, p_{010_2}, p_{001_2}, p_{000_2}]$$

$$\sigma(P_{base_2}) = [p_{111_2}, p_{101_2}, p_{011_2}, p_{001_2}, p_{110_2}, p_{100_2}, p_{010_2}, p_{000_2}]$$

The perfect shuffle connection implemented in the Beneš network can be seen in Figure 7, below:

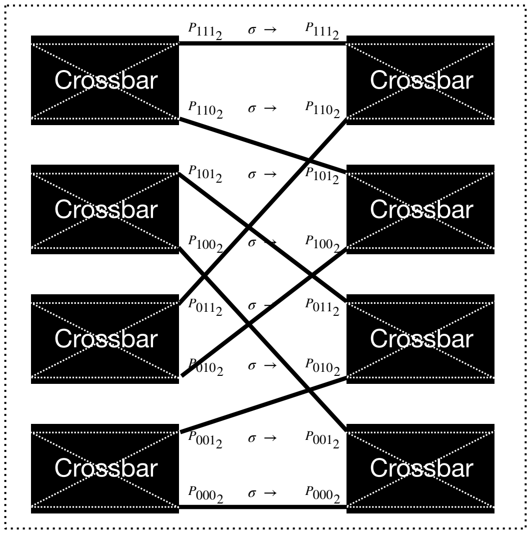

Crossbar
p_{111_2} $\sigma \rightarrow$ p_{111_2}
p_{110_2} $\sigma \rightarrow$ p_{110_2}
p_{101_2} $\sigma \rightarrow$ p_{101_2}
p_{100_2} $\sigma \rightarrow$ p_{100_2}
p_{011_2} $\sigma \rightarrow$ p_{011_2}
p_{010_2} $\sigma \rightarrow$ p_{010_2}
p_{001_2} $\sigma \rightarrow$ p_{001_2}
p_{000_2} $\sigma \rightarrow$ p_{000_2}

Figure 7: Beneš Perfect Shuffle

6.2.2.The Reverse Perfect Shuffle

The reverse perfect shuffle interconnect scheme is described by the inverse sigma function:

$$\sigma^{-1}(P) = [p_{111_2>1}, p_{110_2>1}, \ldots, p_{001_2>1}, p_{000_2>1}] \qquad \text{(Eq. 8)}$$

where > 1 describes a logical shift right.

Given:

$$P = [p_7, p_6, p_5, p_4, p_3, p_2, p_1, p_0]$$

we find that:

$$P_{base_2} = [p_{111_2}, p_{110_2}, p_{101_2}, p_{100_2}, p_{011_2}, p_{010_2}, p_{001_2}, p_{000_2}]$$
$$\sigma^{-1}(P_{base_2}) = [p_{111_2}, p_{011_2}, p_{110_2}, p_{010_2}, p_{101_2}, p_{001_2}, p_{100_2}, p_{000_2}]$$

The reverse perfect shuffle connection implemented in the Beneš network can be seen below:

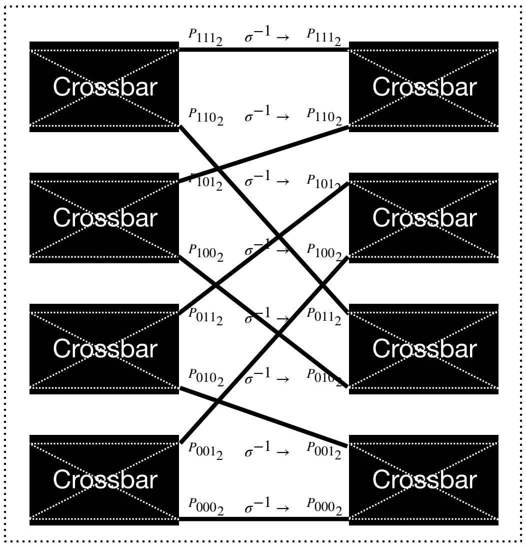

Crossbar
p_{111_2} $\sigma^{-1}\rightarrow$ p_{111_2}
p_{110_2} $\sigma^{-1}\rightarrow$ p_{110_2}
p_{101_2} $\sigma^{-1}\rightarrow$ p_{101_2}
p_{100_2} $\sigma^{-1}\rightarrow$ p_{100_2}
p_{011_2} $\sigma^{-1}\rightarrow$ p_{011_2}
p_{010_2} $\sigma^{-1}\rightarrow$ p_{010_2}
p_{001_2} $\sigma^{-1}\rightarrow$ p_{001_2}
p_{000_2} $\sigma^{-1}\rightarrow$ p_{000_2}

Figure 8: Beneš Reverse Perfect Shuffle

6.2.3.The Butterfly

The butterfly interconnect scheme is described in base 2 numeral system by the beta function:

$$\beta(P) = [p_{111_2}, p_{011_2}, p_{101_2}, p_{001_2}, p_{110_2}, p_{010_2}, p_{100_2}, p_{000_2}] \quad \text{(Eq. 9)}$$

Simply, the bufferfly switches to the destination port by swapping end bits [2] and [0]. The Beneš configuration utilizes a set P of length $n/2$ when using the butterfly interconnect scheme. As such an appropriate beta function prototype is:

$$\beta(P) = [p_{11_2}, p_{01_2}, p_{10_2}, p_{00_2}]$$

Only with $n = 8$ in the Beneš network, and set length of 4, we conclude the butterfly function is described by a logical shift left.

$$IFF\ n = 8\ :\rightarrow\ \beta(P) = [p_{11_2<1}, p_{10_2<1}, p_{01_2<1}, p_{00_2<1}] \qquad \text{(Eq. 10)}$$

Given:

$$P = [p_3, p_2, p_1, p_0]$$

we find that:

$$P_{base_2} = [p_{11_2}, p_{10_2}, p_{01_2}, p_{00_2}]$$
$$\beta(P_{base_2}) = [p_{11_2}, p_{01_2}, p_{10_2}, p_{00_2}]$$

The butterfly connection implemented in the Beneš network can be seen in Figure 9, below:

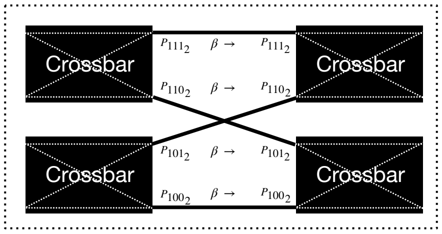

Crossbar
Crossbar
Crossbar
Crossbar
p_{111_2} $\beta \rightarrow$ p_{111_2}
p_{110_2} $\beta \rightarrow$ p_{110_2}
p_{101_2} $\beta \rightarrow$ p_{101_2}
p_{100_2} $\beta \rightarrow$ p_{100_2}

Figure 9: Beneš Butterfly

6.2.4.The Reverse Butterfly

Similar to The Butterfly.

6.2.1.Implementation

The implementation of the perfect shuffle, reverse perfect shuffle, butterfly, and reverse butterfly are shown in Figure 10, below. Note when all crossbar switches are barred (straight through), the reader can convince themselves this network directly passes data straight through without any circuit switching.

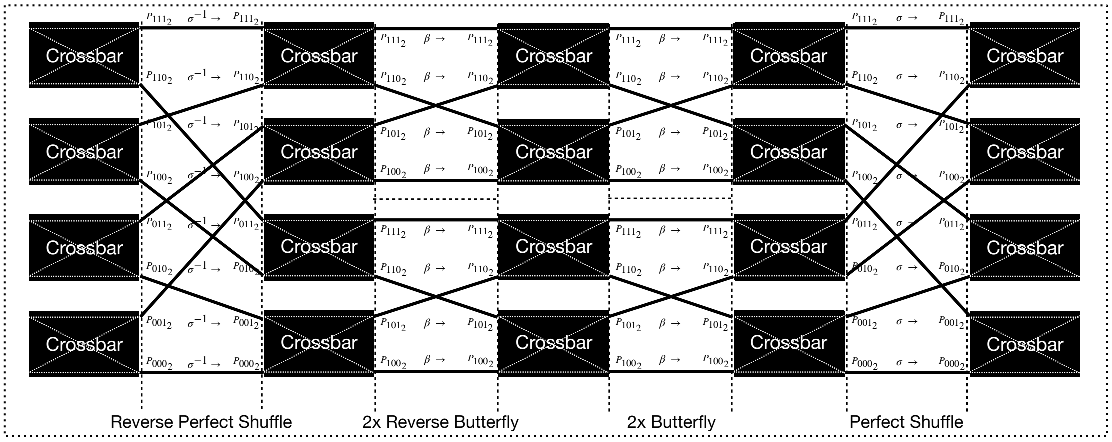

Crossbar
p_{111_2} $\sigma^{-1} \rightarrow$ p_{111_2}
p_{111_2} $\beta \rightarrow$ p_{111_2}
p_{111_2} $\sigma \rightarrow$ p_{111_2}
Reverse Perfect Shuffle
2x Reverse Butterfly
2x Butterfly
Perfect Shuffle

Figure 10: Beneš Interconnections

6.3. Routing Engine

The routing engine allows the Beneš network to provide an output connection to any input port for every permutation of I/O combinations. With 8 input ports we realize $8! = 40'320$ network permutations. We now understand the need for an efficient routing algorithm. The routing algorithm used is two staged, see Figure 11 below. The first: an algorithm stage, the second: a bit-controlled stage.

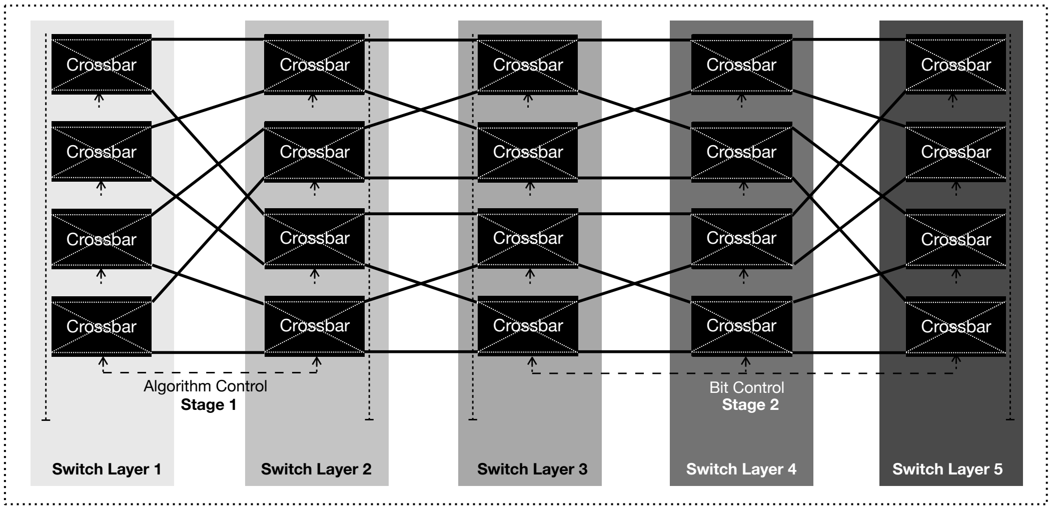


Figure 11: Algorithm Control

6.3.1. Algorithm Stage

'Input ports' refers to the 8 inputs on the LHS of Figure 6, not shown but implied in Figure 8, 10, 11. The set P contains all input ports, $P_{base_2} = [000_2, 001_2, 010_2, 011_2, 100_2, 101_2, 110_2, 111_2]$.

The number of algorithm control stages is:

$$m_a = log_2 N - 1 \qquad \text{(Eq. 11)}$$

The aim of the algorithm stage is to create m_a sets of port groupings.

Root set $M_0 \in P$ (Eq. 12)

Stage 1 set $M_1 \in M_0$ (Eq. 13)

All subsets are composed of $l = \frac{N}{2^i}$ equal-length subsets.

$$M_1 \supseteq M_{1_0} + M_{1_1} \text{ such that } M_{1_0} + M_{1_1} \in M_1 \qquad \text{(Eq. 14)}$$

We define i as the distance from the root set, $i_{M_1} = 1$.

$M_{1_0} \cap M_{1_1} = 0$, mutually exclusive. (Eq. 15)

Subsets of M_i contain $M_i \wedge \{i\{X\}\}$ (Exp. 1)

→ Subsets of M_1 contain $\{X00_2, X01_2, X10_2, X11_2\}$, subsets of M_2 contain $\{XX0_2, XX1_2\}$.

In general for M_1:

$\{(M_{1_0} \cup M_{1_1}) \cap (M_{1_0} \cap M_{1_1} = 0)\} \in M_1$ (Exp. 2)

For M_2 we find subsets M_{2_0} and M_{2_1}, each with properties similar to M_1 described in Expression 2.

The algorithm creates a permutation of each set by controlling crossbar switches in each Switch Layer. Looking at each input subset, the algorithm controls switches to satisfy Expressions 1 and 2.

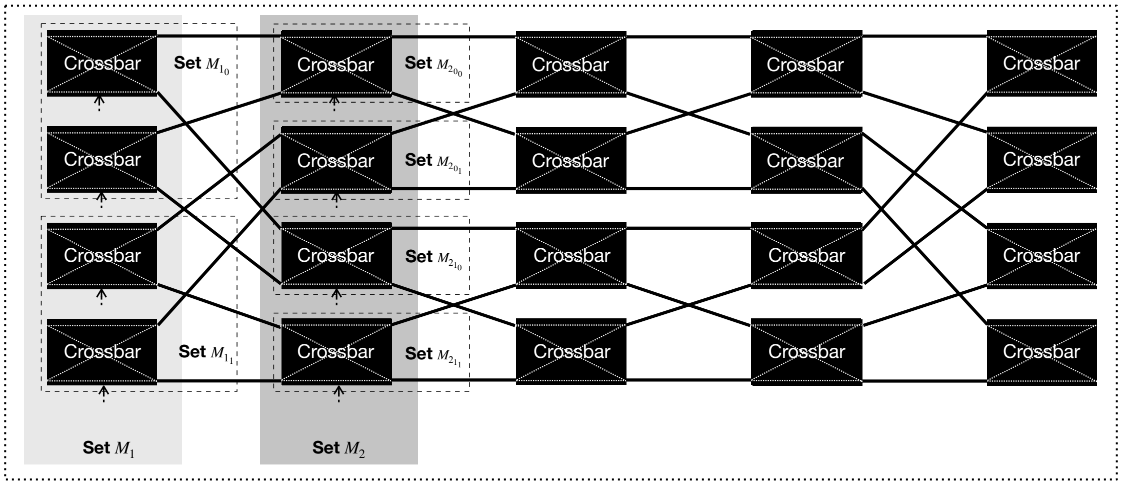


Figure 12: Control Set Domains

6.3.2.Algorithm Stage Example

Take the north-western (top left) crossbar to be $C_{ij} = C_{00}$, the one south of it to be $C_{i(j+1)} = C_{01}$, the one to the east of initial crossbar C_{00} to be $C_{(i+1)j} = C_{10}$.

Given:

$P_{base_2} = [010_2, 011_2, 100_2, 101_2, 111_2, 110_2, 000_2, 001_2]$

This can be read as:

$$P = \quad [Port_0 \rightarrow Port_2, Port_1 \rightarrow Port_3, Port_2 \rightarrow Port_4, Port_3 \rightarrow Port_5,$$
$$Port_4 \rightarrow Port_7, Port_5 \rightarrow Port_6, Port_6 \rightarrow Port_0, Port_7 \rightarrow Port_1]$$

The routing engine may builds the following sets:

$$M_{1_0} = [011_2,101_2,110_2,000_2]$$

$$M_{1_1} = [010_2,100_2,111_2,001_2]$$

$$M_{2_{0_0}} = [011_2,110_2]$$

$$M_{2_{0_1}} = [101_2,000_2]$$

$$M_{2_{1_0}} = [010_2,111_2]$$

$$M_{2_{0_0}} = [100_2,001_2]$$

This corresponds to the following switch controls:

$$C_0 = \{cross, cross, cross, bar\}$$

$$C_1 = \{bar, bar, bar, bar\}$$

The final output of the switch network after the algorithm stage is shown in Figure 13, below:

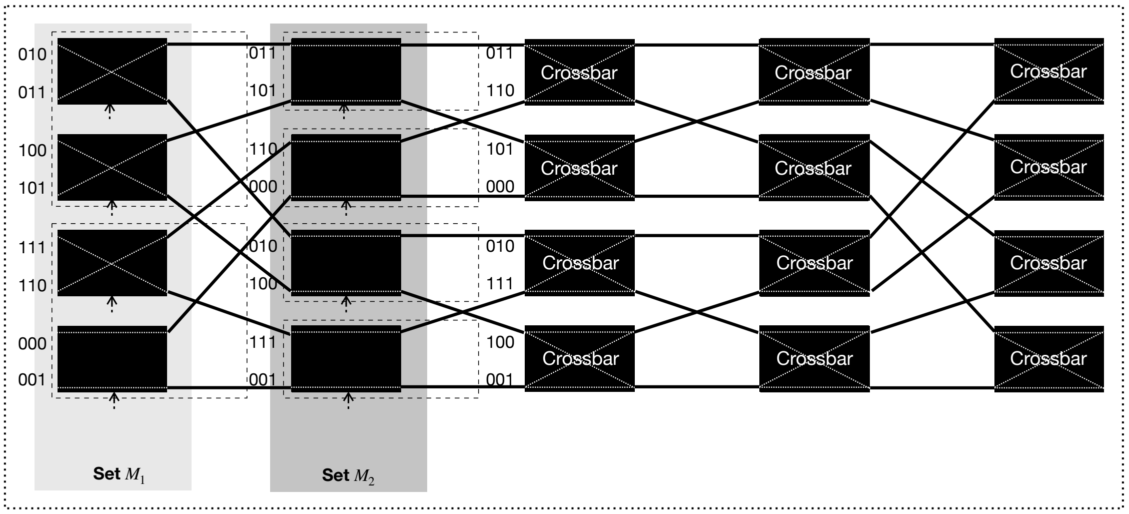


Figure 13: Algorithm Control Result

6.3.3.Bit-Controlled Stage

Number of bit control stages:

$m_b = log_2N$ (Eq. 12)

The remaining work left in the bit-controlled stage is trivial. We control the crossbar given the nth bit-control stage top crossbar port nth bit. We find that:

$$C_2 = \{C_{20_{top}}[0], C_{21_{top}}[0], C_{22_{top}}[0], C_{23_{top}}[0]\}$$

$$C_3 = \{C_{30_{top}}[1], C_{31_{top}}[1], C_{32_{top}}[1], C_{33_{top}}[1]\}$$

$$C_4 = \{C_{40_{top}}[2], C_{41_{top}}[2], C_{42_{top}}[2], C_{43_{top}}[2]\}$$

And we see the correct routing implemented in the network in Figure 14 below:

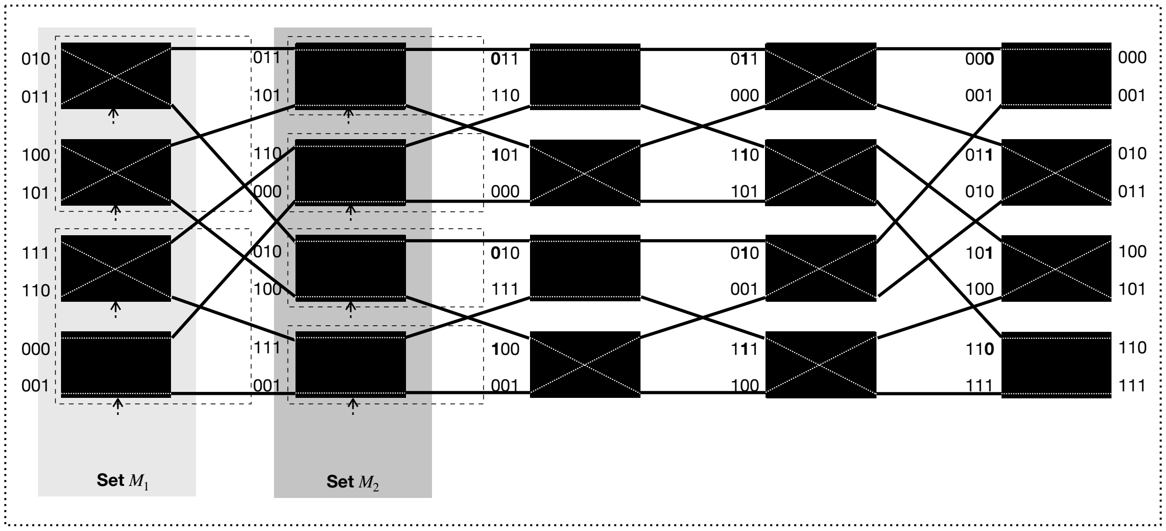


Figure 14: Bit Control Result

6.4. The Arbiter

The asynchronous Arbiter, seen in Figure 15 below, is fitted to the inputs of the switching fabric. It provides local buffering (viz re-timing), mechanisms for fair-use, and flow control when accessing the switching fabric. The Arbiter is capable of avoiding congestion caused by head-of-line blocking in the buffers by scanning beyond the first element targeted by each read pointer.

The Arbiter is mainly concerned with the efficient selection of source-destination port connections through the switching network. Once it selects a set of source — destination port pairs, a request is sent to the routing engine to provide the switch matrix configuration to achieve network configuration which results in the correct connections. The Arbiter avoids greedy neuron behaviour by implementing round-robin source selection. This is to say the Arbiter gives initial port selection priority to a different neuron each network configuration cycle.

The asynchronous handshaking method not only provides a mechanism for asynchronous data transfer between logic blocks, but also provides a flow control to downstream logic. If the downstream wants to disable upstream data transmission it simply does not negotiate a successful handshake. This method of flow control will not cause erroneous behaviour within the Arbiter. The Arbiter contains FIFOs sized for worst-case scenarios. Any updates from the neurons while idling overwrites stale neuron data, avoiding the need for complex eviction policies within the FIFOs.

Since the Arbiter may transmit data in out-of-order fashion, source data is tagged to allow correct identification at the destination end.

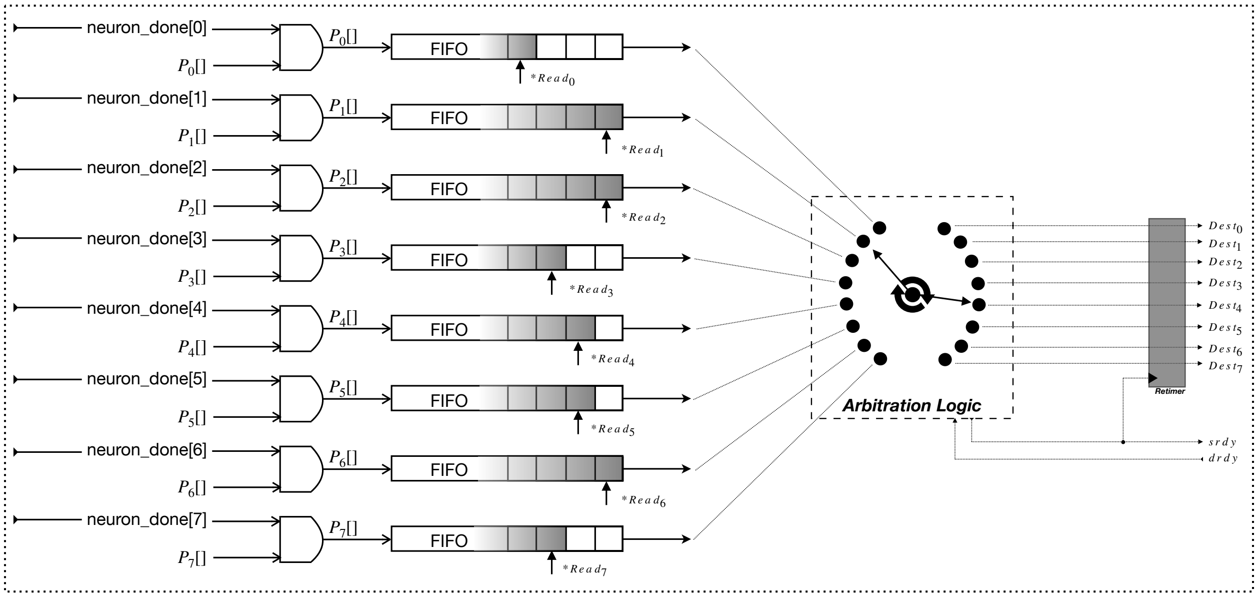


Figure 15: The Arbiter

6.5. Automatic Pruning
6.6. Automatic Floating-Point Truncation Bypass
6.7. Host Controller Instruction Set
6.8. Asynchronous Handshaking, Explained
6.9. Back Propagation Weight Re-calculation
6.10. Network Output Error Calculation