SCALO Architecture Overview

Muhammed Ugur

August 2023

Contents

1 Overview

SCALO [17] is a distributed system composed of up to 60 individual HALO-like [5] nodes. The goal is to support a wide range of multi-region BCI applications (§2) which adhere to power and latency constraints (§3). Each node consists of a RISC-V microcontroller (§4), analog-to-digital/digital-to-analog converters (§5), a heterogeneous set of coarse-grained processing elements (PEs) (§6), NVM-based storage (§7), intra-BCI/external radios (§8), a reconfigurable interconnect between those PEs (§9), an ILP-based scheduler (§10), and battery (§11).

SCALO nodes can loosely be classified as a mix between a heterogeneous coarse-grained reconfigurable array (CGRA) and a dataflow architecture, where each PE runs in its own clock domain to achieve low power and high throughput. There are ~30 different PEs each with their own unique configurations, input formats, and output formats, resembling a collection of interconnected accelerators which continuously process streams of data. SCALO nodes can run independently, communicate with one another, or communicate with external devices.

2 Applications

SCALO is designed to support different classes of applications: internal closed-loop, external closed-loop, and human-in-the-loop. Within these classes are applications such as seizure detection/propagation, movement intent, spike sorting, and data compression. PEs (§6) are designed to support a handful of application pipelines with a degree of reconfigurability across pipelines. The distributed nature of SCALO enables the different classes of applications described above and opens the door for novel multi-site applications in the future.

2.1 Seizure Detection

2.2 Movement Intent

2.3 Spike Sorting

2.4 Data Compression

3 Constraints

There is a system-wide power budget of 15 mW across all components of an individual SCALO node. This is to ensure strict safety requirements due to overheating. Applications may also have latency constraints, such as during seizure detection and stimulation.

For safe operation, there are ultra-low-power Vdd comparator circuits, running at low frequencies, to identify power overshoot. When overshoot happens, this circuit will interrupt the MC (§4), enabling the MC to shut off PEs.

4 Microcontroller

The RISC-V microcontroller (MC) configures pipelines (via programmable circuit switches §9), configures PE parameters (§6), and runs arbitrary computation that is not supported by the PEs. The MC also controls the parameters and algorithms for stimulation, which is ultimately done through the DACs (§5.2). HALO [5] runs the MC at 25 MHz and SCALO [17] runs the MC at 20 MHz, both with 8 KB SRAM.

This MC is based off of Ibex [6], an open-source two-stage in-order 32-bit RISC-V embedded CPU, with the RV32EC ISA (an embedded version of RV32I). It has 16 general-purpose registers and a ”compression” feature to reduce memory requirements for storing programs. Changes were made to the implementation to allow for the PE-specific features mentioned above.

5 Sensors

SCALO [17] and HALO [5] assumes a microelectrode array with 96 channels. This is mainly because a standard Utah array has 96 channels. Some of the additional throughput studies in [17] consider higher channel counts, but they are currently not supported.

5.1 Analog-to-Digital Conversion

Output PE	Input
BBF (§6.1.6)	16-bits
HCONV (§6.3.1)	16-bits
INTLVR (§6.7.4)	16-bits
LZ (§6.5.1)	8-bits
NEO (§6.1.2)	16-bits
SBP (§6.1.1)	16-bits
SC (§6.7.2)	16-bits
XCOR (§6.1.5)	16-bits

Table 1: PEs (§6) which receive data directly from the ADCs.

Each channel produces samples at a frequency of 30 KHz with 16 bits per sample. To reduce power, these 16-bits may be split into two byte chunks and sent over the on-chip network (§8). This requires changes to the implementation of the PEs (§6). This means that each channel generates 480 Kbps, and given 96 channels, the ADCs produce data at ~46 Mbps. The ADC requires 2.88 mW of power for a single sample across all 96 electrodes.

The system assumes ”windowed” samples where samples for each channel is grouped into fixed size lengths, in this case 120 samples. This length enables a window to be generated every 4 ms. However, some of the PEs in the system are designed for shifted windows, specifically a shift of 45 samples or 1.5 ms. This shift implies that there is an overlap between successive windows, but data duplication will impact the performance of the system. Therefore, PEs need to be designed with the illusion of shifted windows. Figuring out how to design the PEs in this way is still an open problem.

The window shift feature was introduced due to datasets consisting of down-sampled recordings (i.e. 4 KHz) and PEs, like XCOR (§6.1.5), requiring comparison of signals across window boundaries. Introducing this shift increases the amount of samples, effectively upsampling the dataset to 30 KHz.

5.2 Digital-to-Analog Conversion

These are mainly used for stimulation, whose logic is supported by the MC (§4). HALO [5] uses only 16 channels for stimulation which gives a 0.48 mW upper bound for chronic stimulation. In SCALO [17], the DAC can consume ~0.6 mW of power.

6 Processing Elements

Processing elements, or PEs, are compute units which typically implement a kernel based on the needs of a BCI application (§2). They can be classified based on their high-level functionality. These classes include feature generation (§6.1), feature classification (§6.2), hashing (§6.3), linear algebra (§6.4), compression (§6.5), networking (§6.6), and miscellaneous (§6.7). Each PE has a set of parameters, an input interface, output format, and a specific clock frequency that it is running at.

Each PE is typically clocked at the lowest frequency necessary to process the incoming data rates. These frequencies are typically 3-180 MHz [16]. Intra-PE synchronization (§9.2) is based on per-PE pausable clock generators and clock control units. The clock generators use ring oscillators with a delay line which is extracted from the critical path. The ring oscillator is designed so that its frequency variation tracks the critical path.

PEs are connected to one another through the on-chip network (§9). Datapaths (§9.3) are fixed but reconfigurable (i.e. the output of a source PE may have different target PE options). The output of a source PE must match the input interface of a target PE however; this is currently enforced at design-time. The outputs of PEs can also be routed to the MC (§4). The interface between the MC and PEs for sharing data still needs to be determined. The MC also configures the parameters of each PE. These parameters live in the internal memory of the PE.

Many of the PEs perform fixed-point arithmetic. This is largely because implementing floating-point arithmetic is expensive in hardware. Samples are also limited to 16-bits which disincentivizes high degrees of precision provided by floating-point arithmetic. The amount of digits after the radix point for fixed-point arithmetic is dependent on the PE implementation. For the remainder of this section, we assume that raw samples are represented in 16-bits. The hardware implementation may represent each sample as two bytes, as discussed in (§5.1).

6.1 Feature Generation

BCI applications require different information features from the neurological signals to realize their goals. The PEs in this section are designed to generate these features from raw signal data. Raw signal data can arrive in two different ways. The first way is through the ADCs (§5.1) in channel-major order (see Figure 1). The other way is through the INTLVR (§6.7.4) in sample-major order (see Figure 2). Each PE assumes that data is in one of these orders.

In channel-major order, this means that **the maximum number of channels must be fixed at design time**. In sample-major order, there are an arbitrary number of channels which can be supported, but **the maximum number of samples (window size) must be fixed at design time.** Supporting an arbitrary number of channels is not entirely true in this case since the INTLVR (§6.7.4) requires that samples be buffered before reordering them, which requires fixing the maximum number of channels in the buffer size. Therefore, many of the situations are when **both the maximum number of channels and samples must be specified.**

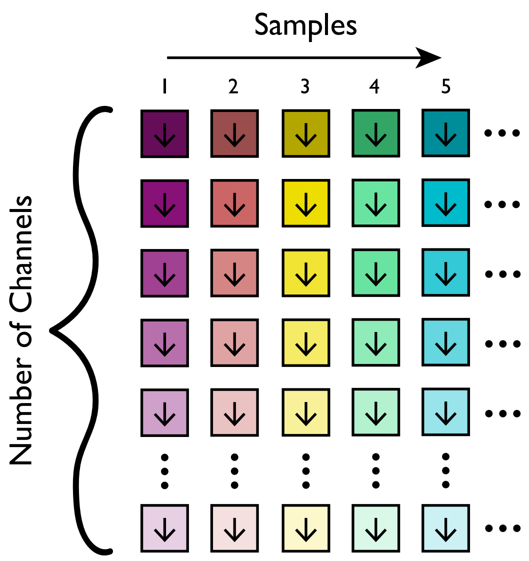

Figure 1: Channel Major Order. This is when a sample at a given time is grouped across all channels.

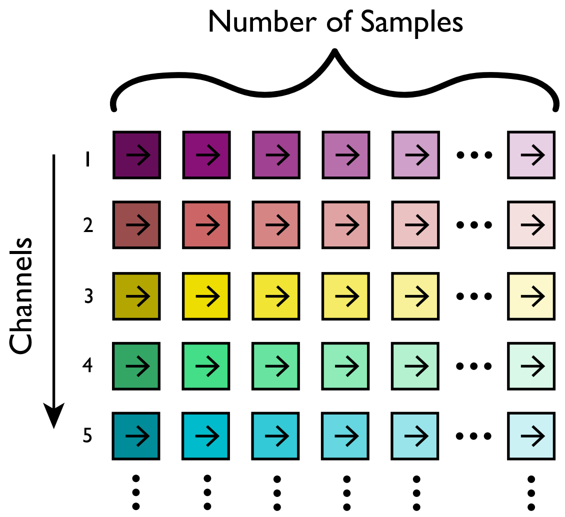

Figure 2: Sample Major Order. This is when the samples in the same channel are grouped together.

The output of feature generation PEs will either continue to represent the raw signals in their time-domain with adjusted values (e.g. BBF §6.1.6), represent them in the frequency-domain (e.g. FFT §6.1.7), or output new feature representations (e.g. NEO §6.1.2). For the remainder of this section, let $x_c[n]$

be the sequence of samples for channel c at a given time-point n. This sequence represents the discrete-time signal for a given channel from the sensors (§5).

6.1.1 SBP: Spike Band Power

This is the power of a signal for a given channel in the 300-1,000 Hz frequency band, as defined in [18]. SBP has been shown to be a better alternative to threshold crossings for spike detection [15, 11].

6.1.1.1 Algorithm

SBP is simply an average magnitude of filtered signal values (i.e. between 300-1,000 Hz) over a window of time. This filtered signal comes from the output of BBF (§6.1.6) and the SBP of this filtered signal $x_c[n]$ is

$$SBP_x = \frac{1}{N} \sum_{n=0}^{N-1} |x_c[n]| \tag{1}$$

Traditionally, in digital signal processing, the power of a signal is defined as the average energy over time:

$$P_x = \frac{1}{N} \sum_{n=0}^{N-1} |x[n]|^2 \tag{2}$$

6.1.1.2 Parameters

Parameter	Algorithm	Variable	Value(s)	Configurable?
Number of samples	$N \in \mathbb{N}$	max_samples	[1, 2048 (why?)]	?
Fixed-point shift	–	–	–	?
Channels	–	num_channels	96	?

6.1.1.3 Implementation

Input PE	Output
ADCs (§5.1)	16-bits
BBF (§6.1.6)	32-bits

Output PE	Input
BMUL (§6.4.2)	32-bits
NPACK (§6.6.1)	8-bits
SUB (§6.4.4)	32-bits

The current implementation computes the SBP in channel-major order. It outputs an average over all samples for each channel up to max_samples number of samples.

```
def sbp(input: i16) -> i32:
    static sbp[NUM_CHANNELS]: i32 = {0}
    static chan_counter: i32 = 0
    static sample_counter: i32 = 0

    sbp[chan_counter] += input
    prev_counter: i32 = chan_counter
    chan_counter += 1

    if chan_counter == NUM_CHANNELS:
        sample_counter += 1
        if sample_counter == MAX_SAMPLES:
            sample_counter = 0

    if sample_counter == 0:
        return sbp_[prev_counter] >> MAX_SAMPLES_LOG
        sbp[prev_counter] = 0
```

6.1.2 NEO: Non-Linear Energy Operator

This is a simple algorithm for computing the energy in a signal [4, 1, 10]. The relative error of this energy value is always below 11% if the sampling rate is greater than eight times the frequency of the original signal [4].

6.1.2.1 Algorithm

For a given sample, this operation will multiply the sample ahead and before then subtract that product from the sample's square. For a given channel and sample $x_c[n]$, NEO performs:

$$NEO_x = x_c[n]^2 - (x_c[n-1] * x_c[n+1]) \tag{3}$$

An extension to this operation is by parameterizing the number of samples ahead and before a given sample $x[n]$.

$$x_c[n]^2 - (x_c[n-k] * x_c[n+k]) \tag{4}$$

By skipping k samples ahead and before, the precision of the energy calculation decreases. This may be a more suitable feature for neuroscientists for classification later in the pipeline.

6.1.2.2 Parameters

Parameter	Algorithm	Variable	Value(s)	Configurable?
Samples ahead (citation)	$k \in \mathbb{N}$	?	?	?
Fixed-point shift	–	–	–	–
Channels	–	num_channels	96	?

6.1.2.3 Implementation

Input PE	Output
ADCs (§5.1)	16-bits
BBF (§6.1.6)	32-bits

Output PE	Input
GATE (§6.7.1)	8-bits
THR (§6.2.2)	32-bits

The current implementation is in channel-major order and outputs a value for every given input.

```
def neo(input: u16, num_channels: i32) -> i32:
    TODO
```

6.1.3 DWT: Discrete Wavelet Transform

Computes the discretized version of the Haar wavelet transform using a lifting scheme. The output is the scaling function (averages) and the wavelet function (coefficients) of the wavelet transform. For compression, DWT with an order of 1 is used and for feature generation, up to order 5 is used.

6.1.3.1 Algorithm

Given an input vector $\vec{x} \in \mathbb{R}^N$ representing a signal, the discrete Haar wavelet transform will transform $\vec{x}$ into a set of averages $\vec{a}_i \in \mathbb{R}^{N/2^i}$ and a set of coefficients $\vec{c}_i \in \mathbb{R}^{N/2^i}$ for $i = 1, \dots, \log_2 N$ where i represents the order of the transform. For $i = 1$, the averages/coefficients are computed using $\vec{x}$ but for $i \geq 2$, $\vec{a}_i$ and $\vec{c}_i$ are computed using $\vec{a}_{i-1}$.

Given two consecutive samples $x[j]$ and $x[j+1]$ in an input vector (i.e. $\vec{x}$ or $\vec{a}_i$), the average is computed as

$$a[j] = \frac{x[j] + x[j+1]}{2} \tag{5}$$

Given the same two samples, the coefficient is computed as

$$c[j] = \frac{x[j] - x[j+1]}{2} \tag{6}$$

For a DWT of order 1, the first set of averages are

$$\vec{a}_1 = [\frac{x[0] + x[1]}{2}, \frac{x[2] + x[3]}{2}, \dots, \frac{x[N-2] + x[N-1]}{2}] \tag{7}$$

and the coefficients are

$$\vec{c}_1 = [\frac{x[0] - x[1]}{2}, \frac{x[2] - x[3]}{2}, \dots, \frac{x[N-2] - x[N-1]}{2}] \tag{8}$$

For an order of 2, the averages are

$$\vec{a}_2 = [\frac{a_1[0] + a_1[1]}{2}, \frac{a_1[2] + a_1[3]}{2}, \dots, \frac{a_1[N/2-2] + a_1[N/2-1]}{2}] \tag{9}$$

and the coefficients are

$$\vec{c}_2 = [\frac{a_1[0]-a_1[1]}{2}, \frac{a_1[2]-a_1[3]}{2}, \ldots, \frac{a_1[N/2-2]-a_1[N/2-1]}{2}] \tag{10}$$

This process is recursive up to $\log_2 N$ steps which is the maximum order for the transform. Additionally, this transform is for the Haar wavelet, but there are other wavelets that can be used in a lifting scheme. However, these other wavelets may require more than 2 consecutive samples per computational step which will complexity.

6.1.3.2 Parameters

Parameter	Algorithm	Variable	Value(s)	Configurable?
Number of samples	$N \in \mathbb{N}$	–	?	?
Order	$O \in \mathbb{N}$	–	$\log_2 N$? 4?	?
Wavelet function	–	–	[Haar,?]	?
Fixed-point shift	–	–	–	–
Channels	–	–	–	–

6.1.3.3 Implementation

Input PE	Output
INTLVR (§6.7.4)	16-bits

Output PE	Input
TOK (§6.5.3)	16-bits
THR (§6.2.2)	32-bits

In the current implementation, the PE expects that a 16-bit sample be split in two bytes. Therefore, the PE will perform the following two operations for an input byte sequence of $(0x11, 0x22, 0x33, 0x44)$ and output them one after the other.

$$0x2211 - 0x4433$$

$$0x4433 + ((0x2211 - 0x4433) >> 1)$$

The first computation is the predict step corresponding to the coefficients in the algorithm. The second computation is the update step corresponding to the averages in the algorithm.

```
def dwt(input: u8) -> i16:
    TODO
```

6.1.4 DTW: Dynamic Time Warping

This is an algorithm for measuring the similarity between two signals, mainly for those that are misaligned in time. DTW is equivalent to minimizing Euclidean distance between signals across all temporal alignments.

6.1.4.1 Algorithm

Assume that there are two sequences of values $(a_1, a_1, \ldots, a_n)$ and $(b_1, b_1, \ldots, b_m)$. The distance between values in these sequences is computed using a distance function $d : \mathbb{R} \times \mathbb{R} \to \mathbb{R}^+$ which takes in a value from each sequence and outputs a positive real value corresponding to "how close" the two values are (lower is closer). The standard distance function is $d(a_i, b_j) = |a_i - b_j|$ (or Euclidean distance).

The algorithm computes a distance a matrix whereby each entry is defined by the following recurrence relation:

$$D_{min}(a_i, b_j) = \min \begin{Bmatrix} D_{min}(a_{i-1}, b_{j-1}) \\ D_{min}(a_i, b_{j-1}) \\ D_{min}(a_{i-1}, b_j) \end{Bmatrix} + d(a_i, b_j) \tag{11}$$

The final DTW value, representing the similarity between the two sequences, is the corner of that matrix:

$$DTW = D_{min}(a_n, b_m)$$

An addition to this algorithm is the Sakoe-Chiba's window optimization [14]. This introduces a constant-width band parameterized by a radius $r \in \mathbb{N}$ which restricts how many matches can occur for every sample in the signal.

6.1.4.2 Parameters

Parameter	Algorithm	Variable	Value(s)	Configurable?
First sequence length	$N \in \mathbb{N}$	?	120	?
Second sequence length	$M \in \mathbb{N}$	?	120	?
Warping window size	$r \in \mathbb{N}$	band_size	12?	Yes
Distance function	$d : \mathbb{R} \times \mathbb{R} \to \mathbb{R}^+$	–	–	No
Fixed-point shift	–	–	–	–
Channels	–	–	–	–

6.1.4.3 Implementation

Input PE	Output
SC (§6.7.2)	xx-bits
UNPACK (§6.6.2)	xx-bits

Output PE	Input
THR (§6.2.2)	32-bits

The current implementation is in sample-major order.

```
def dtw(input: i16) -> u32:
    TODO
```

6.1.5 XCOR: Cross Correlation

Computes the cross correlation, which is a measure of similarity between two sequences/signals. with a displacement parameter of one relative to the other (i.e. lag). Does this computation between all pairs of channels for a fixed signal size. Uses fixed-point arithmetic.

6.1.5.1 Algorithm

Given two signals $x[n]$ and $y[n]$, the goal is compute a metric which quantifies their similarity. The following is the common definition of the cross-correlation/correlation with lag (denoted τ) for digital signal processing:

$$R_{xy}[\tau] = \sum_{n=0}^{N-1} x[n]y[n-\tau], \tau = 0, 1, \ldots, k \tag{12}$$

This can be augmented with the normalized correlation to get a value which does not exceed 1:

$$N_{xy}[\tau] = \frac{R_{xy}[\tau]}{\sqrt{R_{xx}[0]}\sqrt{R_{yy}[\tau]}} = \frac{\sum_{n=0}^{N-1} x[n]y[n-\tau]}{\sqrt{\sum_{n=0}^{N-1} x[n]^2}\sqrt{\sum_{n=0}^{N-1} y[n-\tau]^2}} \tag{13}$$

Subtracting the mean values will help reduce any bias in the sequences and give a fair comparison:

$$XCOR_{xy}[\tau] = \frac{\sum_{n=0}^{N-1}(x[n]-\bar{x})(y[n-\tau]-\bar{y}[\tau])}{\sqrt{\sum_{n=0}^{N-1}(x[n]-\bar{x})^2}\sqrt{\sum_{i=0}^{N-1}(y[n-\tau]-\bar{y}[\tau])^2}} \tag{14}$$

where $\bar{x} = \frac{1}{N}\sum_{n=0}^{N-1} x[n]$ and $\bar{y}[\tau] = \frac{1}{N}\sum_{n=0}^{N-1} y[n-\tau]$. This equation is also the standard definition of Pearson's correlation coefficient used to correlate two sets of data, mainly used by statisticians and mathematicians. However, this algorithm adds the lag parameter to this definition.

Additionally, our algorithm computes $XCOR_{xy}[\tau]$ across all possible pairs of channels. In other words, let C be the set of all unique pairs of channels up to some maximum number of channels $M \in \mathbb{N}$. Then, $|C| = \frac{M(M-1)}{2}$ and $\forall (i,j) \in C$, this algorithm will compute $XCOR_{ij}[\tau]$. The output will be a vector $[\ldots, XCOR_{ij}[\tau], \ldots] \in \mathbb{R}^{|C|}$.

6.1.5.2 Parameters

Parameter	Algorithm	Variable	Value(s)	Configurable?
Number of samples	$N \in \mathbb{N}$	window_size	120	?
Lag	$\tau \in \mathbb{N}$	lag	[0, 119]	Needs testing
Fixed-point shift	–	–	8	?
Channels	$M \in \mathbb{N}$	num_channels	96	?

6.1.5.3 Implementation

Input PE	Output
ADCs (§5.1)	16-bits

Output PE	Input
SVM (§6.2.1)	32-bits

This implementation is in channel-major order. It computes the cross correlation for each channel across all other channels.

```
def xcor(input: i16) -> i32:
    TODO
```

6.1.6 BBF: Butterworth Bandpass Filter

Filters the signal using the Butterworth Bandpass method. Uses fixed-point arithmetic.

6.1.6.1 Algorithm

The first step is to apply gain to the signal to deal with attenuation, so let $g_c[n] = x_c[n]/G$ for some gain G. The output of the filter for a given channel is

$$y_c[n] = \sum_{k=0}^{2*O} v_k g_c[n-k] + \sum_{k=1}^{2*O} u_k y_c[n-k] \tag{15}$$

where $\vec{v}, \vec{u} \in \mathbb{R}^{2*O+1}$ are coefficients for the filter computed for the order O and the cutoff frequencies (i.e. a low frequency and high frequency).

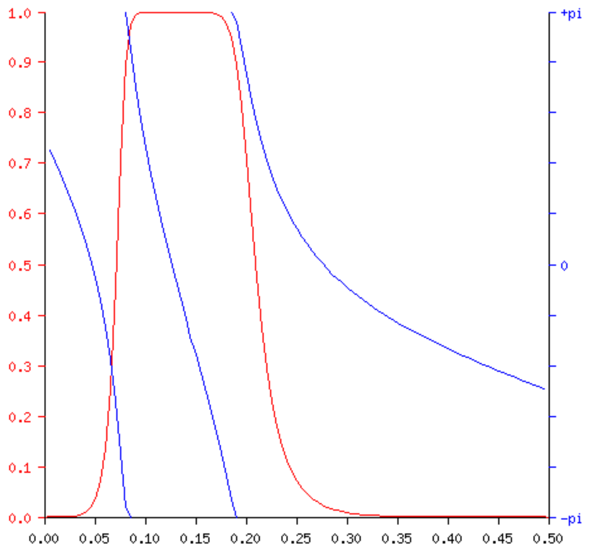

Figure 3: Magnitude (red) and phase (blue) vs. frequency. The x-axis is frequency as a fraction of the sampling rate (30000 Hz) and 0.5 represents the Nyquist frequency. The red y-axis is magnitude (linear, normalized) and the blue y-axis is phase. The cutoff frequencies for this graph are 2250 Hz and 6000 Hz respectively. Created using the mkfilter online plot generator.

6.1.6.2 Parameters

Parameter	**Algorithm**	**Variable**	**Value(s)**	**Configurable?**
Gain	$G \in \mathbb{R}$	gain	$[-2^{31}, 2^{31}-1]$	Yes
Order	$O \in \mathbb{N}$	order	$[1, 10]$	Yes
Filter coefficients for inputs	$\vec{v} \in \mathbb{R}^{(2*O)+1}$	xcoef[2O+1]	$[-2^{31}, 2^{31}-1]$	Yes
Filter coefficients for outputs	$\vec{u} \in \mathbb{R}^{(2*O)+1}$	ycoef[2O+1]	$[-2^{31}, 2^{31}-1]$	Yes
Low frequency	Replaces coeffs	?	$(0, 0.5)$	Software (ext.)
High frequency	Replaces coeffs	?	$(0, 0.5)$	Software (ext.)
Fixed-point shift	–	shift	$[0, 15]$	?
Channels	–	num_channels	96	?

6.1.6.3 Implementation

Input PE	Output
ADCs (§5.1)	16-bits

Output PE	Input
FFT (§6.1.7)	16-bits
NEO (§6.1.2)	16-bits
SBP (§6.1.1)	16-bits
SVM (§6.2.1)	32-bits

The implementation is based off of the mkfilter tool.

```
def bbf(input: i16) -> i32:
    TODO
```

6.1.7 FFT: Fast Fourier Transform

Performs the Discrete Fourier Transform. Uses fixed-point arithmetic. Movement intent requires 14-25 point FFTs and seizure prediction requires 1024-point FFTs [5].

6.1.7.1 Algorithm

Let $W_N^{nk} = e^{-i(2\pi/N)nk}$, then the DFT can be represented as the following:

$$X_c[k] = \sum_{n=0}^{N-1} x_c[n] W_N^{nk}, k = 0, \ldots, N-1 \tag{16}$$

where $X_c[k]$ is the Fourier coefficient at k for a given channel c, representing the magnitude and initial phase of the k-th sinusoidal component of the signal $x_c[n]$. In other words, each $X_c[k]$ represents "how much" oscillatory behavior is contained in the signal at the specified frequency $2\pi/k$. These Fourier coefficients are referred to as the *spectrum* of the initial signal $x_c[n]$. Additionally,

$$|X_c[k]|^2 \tag{17}$$

represents the energy of the signal at the frequency $(2\pi/N)k$ (up to a scale factor N). This enables determining how much of the total energy of a signal is distributed across different frequencies.

6.1.7.2 Parameters

Parameter	Algorithm	Variable	Value(s)	Configurable?
Signal length	$N \in \mathbb{N}$	?	?	Yes
Number of points	–	?	[1, N]	Yes
Lowest frequency	–	?	?	?
Highest frequency	–	?	?	?
Frequency bins	–	?	?	?
Fixed-point shift	–	–	[0, 15]	?
Channels	–	–	–	–

6.1.7.3 Implementation

Input PE	Output
INTLVR (§6.7.4)	16-bits
BBF (§6.1.6)	32-bits

Output PE	Input
SVM (§6.2.1)	32-bits
THR (§6.2.2)	32-bits

This is implemented in sample-major order.

Given a specified number of samples, the implementation will output a maximum number of frequency components capped at the number of samples. Further configurations will allow multiple adjacent frequency components to be summed. If none of these configurations are present, then there will be an output for each input.

```
def fft(input: i16) -> i32:
    TODO
```

6.2 Feature Classification

Once features are generated, they are typically run through a decision process which classifies their activity into different categories (e.g. seizure or no seizure).

6.2.1 SVM: Support Vector Machine

Performs the inference for a support vector machine with a fixed number of weights.

6.2.1.1 Algorithm

Given a vector of weights $\vec{w} \in \mathbb{R}^N$, and an input vector $\vec{x} \in \mathbb{R}^N$, the inference is computed as

$$SVM(\vec{x}) = \sum_{i=0}^{N-1} w_i x_i \tag{18}$$

Some of the inputs from PEs can be in fixed-point which requires that the weights be configured in fixed-point as well, with the same shift from the radix point as the input.

6.2.1.2 Parameters

Parameter	Algorithm	Variable	Value(s)	Configurable?
Model size	$N \in \mathbb{N}$	model_size	4960?	?
Model	$\vec{w} \in \mathbb{R}^N$	model[N]	$[-2^{31}, 2^{31}-1]$	Yes
Fixed-point shift	–	–	?	?
Channels	–	–	–	–

6.2.1.3 Implementation

Input PE	Output
FFT (6.1.7)	32-bits
BBF (6.1.6)	32-bits
XCOR (6.1.5)	32-bits
UNPACK (6.6.2)	8-bits

Output PE	Input
THR (§6.2.2)	32-bits
NPACK (§6.6.1)	8-bits

Outputs a value every model_size number of inputs.

```
def svm(input: i32) -> i32:
    static counter: i32 = 0
    static sum: i32 = 0
    sum += MODEL[counter] * input
    counter += 1
    if counter == MODEL_SIZE:
        return sum
        sum = 0
        counter = 0
```

6.2.2 THR: Thresholding

This PE will output a bit vector indicating if values are within a pre-specified range of values. There is typically a unique threshold for each independent channel.

6.2.2.1 Algorithm

Given an input vector $\vec{x} \in \mathbb{R}^N$, the algorithm will check if each element in that vector is within a certain range specified by a lower bound $L \in \mathbb{R}$ and an upper bound $H \in \mathbb{R}$. The output will be the following:

$$THR_{L,H}(\vec{x}) = [f_{L,H}(x_0), f_{L,H}(x_1), \ldots, f_{L,H}(x_{N-1})] \in \mathbb{R}^N \tag{19}$$

where

$$f_{L,H}(x) = \begin{cases} 1 & L \leq x \leq H \\ 0 & otherwise \end{cases}$$

6.2.2.2 Parameters

Parameter	Algorithm	Variable	Value(s)	Configurable?
Number of inputs	$N \in \mathbb{N}$	?	[1, 16]	Yes
Lower bound	$L \in \mathbb{R}$	thr_low	$[-2^{31}, 2^{31}-1]$	Yes
Upper bound	$H \in \mathbb{R}$	thr_high	$[-2^{31}, 2^{31}-1]$	Yes
Fixed-point shift	–	–	–	–
Channels	–	–	–	–

6.2.2.3 Implementation

Input PE	Output
DTW (§6.1.4)	32-bits
DWT (§6.1.3)	16-bits
FFT (§6.1.7)	32-bits
NEO (§6.1.2)	32-bits
SVM (§6.2.1)	32-bits

Output PE	Input
GATE (§6.7.1)	1-bit

This PE will be paired with GATE (§6.7.1) to select which data to forward and which data to drop.

```
def thr(input: i32, thr_low: i32, thr_high: i32) -> bool:
    if thr_low <= input <= thr_high:
        return 1
    return 0
```

6.3 Hashing

Hashing is used as a mechanism to lower latency of communication across nodes in SCALO.

6.3.1 HCONV: Hash Convolution

This performs a sliding window sketch as described in [7]. It emits a value for each inner window, instead of a single bit, resembling a convolution. Uses fixed-point arithmetic.

6.3.1.1 Algorithm

Given an input vector $\vec{x} \in \mathbb{R}^N$, a randomly generated filter vector $\vec{r} \in \mathbb{R}^W$, and a step size $\delta \in \mathbb{N}$, the algorithm computes the following:

$$HCONV(x) = [\sum_{i=0}^{W-1} r_i X_i^0, \sum_{i=0}^{W-1} r_i X_i^1, \ldots, \sum_{i=0}^{W-1} r_i X_i^{(N-W)/\delta}] \tag{20}$$

where

$$X^k = [x_{k*\delta}, x_{k*\delta+1}, \ldots, x_{k*\delta+W-1}] \in \mathbb{R}^W \tag{21}$$

6.3.1.2 Parameters

Parameter	Algorithm	Variable	Value(s)	Configurable?
Number of inputs	$N \in \mathbb{N}$	?	?	?
Inner window size	$W \in \mathbb{N}$	inner_len	?	?
Random vector	$\vec{r} \in \mathbb{R}^W$	r_vec[W]	$[-2^{15}, 2^{15}-1]$	?
Step size	$\delta \in \mathbb{N}$	?	?	?
Fixed-point shift	–	?	12	?
Channels	–	num_channels	96	?

6.3.1.3 Implementation

Input PE	**Output**
ADCs (5.1)	16-bits

Output PE	**Input**
EMDH (§6.3.3)	64-bits
NGRAM (§6.3.2)	1-bit

This implementation assumes channel-major order where samples arrive one-by-one for each channel, i.e.

$$(x_1[0], x_2[0], \dots, x_n[0], x_1[1], x_2[1], \dots, x_n[1], x_1[2], \dots)$$

where n corresponds to num_channels. For each incoming sample in this sequence, the PE will keep track of per-channel state to compute a running convolution. This means that the PE will output num_channels number of outputs after every num_channels $*$ (inner_len $- 1$) inputs. The outputs will be the dot product between r_vec and a sequence of inner_length values for each channel. For example, let inner_len $= 2$ and r_vec $= [r_1, r_2]$. Then the sequence of outputs will be the following:

$$(r_1x_1[0] + r_2x_1[1], r_1x_2[0] + r_2x_2[1], \dots, r_1x_{96}[0] + r_2x_{96}[1],$$
$$r_1x_1[2] + r_2x_1[3], r_1x_2[2] + r_2x_2[3], \dots, r_1x_{96}[2] + r_2x_{96}[3],$$
$$r_1x_1[4] + r_2x_1[5]), \dots)$$

Notice here that the step size of the convolution window is the same as inner_len. The current implementation does not support a sliding window with overlapping values after each step. Ideally, step_sz will be parameterized to configure the step size to resemble the original paper.

Additionally, HCONV needs to take into account the shift of windows. After each 45 samples, the implementation needs to compute the sliding window convolution for 45 new samples and 75 old samples.

```
def hconv(
    input: i16,
    num_channels: i8,
    r_vec: [i16],
    inner_len: i8
) -> u32:
    static results[num_channels]: [i32] = {0}
    static curr_channel: i8 = 0
    static curr_inner: i8 = 0

    results[curr_channel] += input * r_vec[curr_inner]

    if curr_inner == inner_len - 1:
        return results[curr_channel]
        results[curr_channel] = 0

    curr_outer += 1

    if curr_channel == num_channels:
        curr_channel = 0
        curr_inner += 1

    if curr_inner == inner_len:
        curr_inner = 0
```

6.3.2 NGRAM: Hash Ngram Generation

Performs the shingle (n-gram) generation described in [7] and then does a weighted minwise hash computation described in [3]. Uses fixed point arithmetic.

6.3.2.1 Algorithm

Given an input vector $\vec{x} \in \mathbb{R}^N$, the first step is to convert this into a bit vector by applying $sign : \mathbb{R} \to \{1, 0\}$ to each element in $\vec{x}$:

$$sign(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{22}$$

The next step is to perform shingle (n-grams) generation which emits a weighted set S_x describing the number of occurrences of sub-sequences in the input sequence (i.e. $\vec{x} \in \{1, 0\}^N$). The sub-sequence length is parameterized by $1 \leq M \leq N$.

$$S_x = \{(w_i, s_i) | s_i = \{x_i, x_{i+1}, \ldots, x_{i+M-1}\}, 0 \leq i \leq (N - M)\} \tag{23}$$

where w_i indicates how many times the sub-sequence s_i has been seen across the entire input sequence.

This weighted set S_x will then go through a weighted minhash procedure, generating a hash. The idea is that comparing two of these hashes will indicate the similarity of the original sequences. The algorithm described in [3] assumes

- $\vec{r} \in \mathbb{R}^{2^M}$ s.t. $r_k \sim \text{Gamma}(2,\ 1)$ (i.e. $P(r_k) = r_k e^{-r_k}, r_k \geq 0$)
- $\vec{c} \in \mathbb{R}^{2^M}$ s.t. $c_k \sim \text{Gamma}(2,\ 1)$
- $\vec{b} \in \mathbb{R}^{2^M}$ s.t. $b_k \sim \text{Uniform}(0,\ 1)$

which means that there are three random variables for each possible sub-sequence s_i. The hash computation for a given sub-sequence is the following:

$$t_k = \left\lfloor \frac{\ln w_k}{r_k} + b_k \right\rfloor \tag{24}$$

To determine the minhash, the following two values are also computed for a given sub-sequence:

$$\begin{aligned} y_k &= e^{r_k(t_k - b_k)} \\ a_k &= \frac{c_k}{y_k e^{r_k}} \end{aligned} \tag{25}$$

Finally, the weighted minhash of S_x is (k^*, t_{k^*}) s.t.

$$k^* = \arg\min_k a_k \tag{26}$$

In this case, k^* represents the sub-sequence which was chosen for hashing and t_{k^*} is that sub-sequence's hash value. To simplify the computation of the algorithm, (25) can be re-written with log values for an equivalent result:

$$\begin{aligned} \ln y_k &= r_k(t_k - b_k) \\ \ln a_k &= \ln c_k - \ln y_k - r_k \end{aligned} \tag{27}$$

where $k^* = \arg\min_k \ln a_k$. This can further be simplified by substituting t_k.

6.3.2.2 Parameters

Parameter	Algorithm	Variable	Value(s)	Configurable?
Stream size	$N \in \mathbb{N}$	?	120	?
Ngram size	$M \in \mathbb{N}$	?	5	?
Random vector	$\vec{r} \in \mathbb{R}^{2^M}$	?	$[-2^{15}, 2^{15} - 1]$	?
Beta vector	$\vec{b} \in \mathbb{R}^{2^M}$	?	$[-2^{15}, 2^{15} - 1]$	?
Log vector	$\vec{c} \in \mathbb{R}^{2^M}$	?	$[-2^{15}, 2^{15} - 1]$	?
Fixed-point shift	–	?	11	?
Channels	–	–	–	–

6.3.2.3 Implementation

Input PE	Output
HCONV (6.3.1)	32-bits

Output PE	Input
GATE (§6.3.2)	8-bits
SC (§6.7.2)	8-bits

Given the maximum possible value for the weights, the hash value in addition to a Ngram size of 5 will be within 8-bits (i.e. the hash value is 3-bits).

```
def ngram(input: u32) -> u32:
    TODO
```

6.3.3 EMDH: Earth-Mover's Distance Hash

Performs the Earth-Mover's Distance Hash described in [2]. This is a locality-sensitive hash. It can be used to approximate other similar measures such as the chi-squared distance, euclidean distance, and the Pearson's cross correlation. Uses fixed-point arithmetic.

6.3.3.1 Algorithm

Given an input vector $\vec{x} \in \mathbb{R}^{+N}$ and a random vector $\vec{r} \in \mathbb{R}^{+N}$, where each entry is chosen from a Gaussian distribution with positive value, the hash is computed as follows:

$$h(\vec{x}) = \lfloor \frac{\sqrt{\frac{8*(\sum_{i=0}^{N-1} r_i x_i)}{W^2} + 1} - 1}{2} + b \rfloor \tag{28}$$

where $\forall \vec{x} \in \mathbb{R}^{+N}, h(\vec{x}) = n \iff I_{n-1} \leq \sum_{i=0}^{N-1} r_i x_i < I_n$ where $[I_{n-1}, I_n]$ represents an interval of possible values (i.e. hash bucket). Additionally, $\forall n \in \mathbb{N}, \sqrt{\frac{(I_{n-1}-I_n)^2}{I_{n-1}+I_n}} = W$ which represents the size of each of these intervals.

6.3.3.2 Parameters

Parameter	Algorithm	Variable	Value(s)	Configurable?
Number of inputs	$N \in \mathbb{N}$	–	–	–
Random vector	$\vec{r} \in \mathbb{R}^N$	-	–	–
Bucket size	$W \in \mathbb{N}$	?	2^7	?
Hash offset	$b \in \mathbb{R}$	fixed_b	765	Yes
Fixed-point shift	–	shift	14	?
Channels	–	–	–	–

6.3.3.3 Implementation

Input PE	Output
HCONV (6.3.1)	32-bits

Output PE	Input
GATE (§6.7.1)	8-bits
SC (§6.7.2)	xx-bits

This implementation will create a hash for every input. This assumes that the input is the sum from the algorithm (i.e. $\sum_{i=0}^{N-1} r_i x_i$). This is the case for the input connection from HCONV (§6.3.1) since that computes the convolution of a subsequence with a random vector $\vec{r}$.

```
#define FIXED_1 (1 << SHIFT_W)
#define FRAC_MASK (FIXED_1 -1)
#define MOD_MASK ((1 << SHIFT_W) - 1)

def emdh(input: i64) -> u16:
    x: i64 = (input >> (SHIFT_W2 - 3)) + FIXED_1
    x = sqrt64(x)
    h: i64 = (x - FIXED_1)) >> 1
    h_b: i64 = h + FIXED_B
    floor: i64 = h_b >> SHIFT_W
    ceil: i64 = 0
    if h_b & FRAC_MASK:
        ceil = floor + 1
    else:
        ceil = floor
    return ceil & MOD_MASK

// TODO
def sqrt64(x: i64) -> i64:
    pass
```

6.3.4 HFREQ: Hash Frequency

Sorts a set of input hashes by descending frequency. The implementation also forwards the original hash sequence.

6.3.4.1 Algorithm

Given an input sequence of hashes $\vec{x} \in \mathbb{R}^N$, the algorithm will output a sequence of descending hash/frequency pairs. In other words, it will output the following sequence after receiving N inputs:

$$(x_{a_0}, f(x_{a_0}), x_{a_1}, f(x_{a_1}), \ldots, x_{a_{U-1}}, f(x_{a_{U-1}})) \tag{29}$$

where $U \in \mathbb{N}$ is the number of unique hashes seen in $\vec{x}$ $(1 \le U \le N)$, $(a_0, a_1, \ldots, a_{U-1})$ are a subset of indices from the original input s.t. if $i < j$ then $f(x_{a_i}) \ge f(x_{a_j})$ where $f : \mathbb{R} \to \mathbb{N}$ returns the frequency of an input hash in $\vec{x}$.

6.3.4.2 Parameters

Parameter	Algorithm	Variable	Value(s)	Configurable?
Number of inputs	$N \in \mathbb{N}$	matrix_size	[?, 3 * 96?]	Yes
Fixed-point shift	–	–	–	–
Channels	–	num_channels	96	?

6.3.4.3 Implementation

Input PE	Output
GATE (6.7.1)	8-bits

Output PE	Input
HCOMP (§6.3.5)	32-bits

After receiving a certain number of hashes, this implementation will output the hashes and their frequencies in descending order. Additionally, as the PE reads these input hashes, it will forward the original hash that it read to the next PE. So up to the number of hashes it expects, the implementation will act as an identity function.

```
#define RETURN_FREQ 1
#define MATRIX_SIZE (NUM_CHANNELS * NUM_COLUMNS)
#define STREAM_SIZE (MATRIX_SIZE + NUM_UNIQUE)

entry {
    hash: u32
    count: u16
}

def hfreq(input: u32) -> u32:
    static table[TABLE_SIZE]: [entry] = {0}
    static num_inputs: u16 = 0

    result: entry = {INVALID, INVALID}

    if num_inputs < MATRIX_SIZE:
        table_put(input)
        num_inputs += 1
    else:
        result = table_max()

    if result.hash != INVALID:
        return result.hash
        if RETURN_FREQ:
            return result.count
```

6.3.5 HCOMP: Hash Compression

Compresses hashes to improve latency for cross-device communication and similarity checking. The compression algorithm will apply dictionary coding, run-length coding, and Elias-Gamma coding.

6.3.5.1 Algorithm

Given a input vector of hashes $\vec{x} \in \mathbb{R}^N$ and a list of unique hashes $\vec{u} \in \mathbb{R}^U$ that are present in $\vec{x}$, this algorithm will first create a dictionary encoding. This is a function $d : \mathbb{R} \to \mathbb{N}$ which maps a unique hash to its index in $\vec{u}$. i.e.

$$d(x_i) = \begin{cases} k & \text{if } \exists u_k \in \vec{u} \text{ s.t. } u_k == x_i \\ \infty & \text{otherwise} \end{cases} \tag{30}$$

The second step is to apply run-length encoding. This transforms all repetitive sub-sequences in $\vec{x}$ to a count/value pair. In other words, run-length encoding partitions the original input vector $\vec{x}$ into $\vec{x_{s_0}} \in \mathbb{R}^{n_0}, \vec{x_{s_1}} \in \mathbb{R}^{n_1}, \ldots, \vec{x_{s_j}} \in \mathbb{R}^{n_j}$ vectors s.t. $\vec{x_{s_0}}||\vec{x_{s_1}}||\ldots||\vec{x_{s_j}} = \vec{x}$ and $n_0 + n_1 + \cdots + n_j = N$ for some $j \in [0, N-1]$ where

$$\begin{aligned}
\vec{x_{s_0}} &= [x_0, x_1, \ldots, x_{n_0-1}] \\
\vec{x_{s_1}} &= [x_{n_0}, x_{n_0+1}, \ldots, x_{n_0+n_1-1}] \\
&\ldots \\
\vec{x_{s_j}} &= [x_{(\sum_i^{j-1} n_i)}, x_{(\sum_i^{j-1} n_i)+1}, \ldots, x_{(\sum_i^{j-1} n_i)+n_j-1}]
\end{aligned} \tag{31}$$

and for any two elements $x_q, x_p \in \vec{x_{s_i}}, x_q == x_p$ for all $0 \leq i \leq j$. Let this value be $x_{s_{i^*}}$.

Run-length encoding will replace each $\vec{x_{s_i}}$ with $(n_i, x_{s_{i^*}})$. This pair represents a the count/value pair mentioned earlier, and there will j such pairs in the output. To reduce the total number of bits, the dictionary function d will be applied on the hash value, i.e. the output will be $(n_i, d(x_{s_{i^*}}))$. This means that the value in the pair is now represented in $\lceil \log_2 U \rceil$ bits. Additionally, we can assume that n_i will be represented in $\lceil \log_2 N \rceil$ bits.

For the final coding step, the algorithm will use an Elias-Gamma coding function for some $S, T \in \mathbb{N}$ which compresses the bit-vector representation of a number (e.g. representing 5 as 00101 instead of 00000101, using 5-bits instead of a fixed 8-bits). Therefore, depending on the maximum bits needed to represent a count/value (i.e. $\lceil \log_2 U \rceil$ and $\lceil \log_2 N \rceil$ respectively), Elias-Gamma coding will reduce the total number of bits necessary for the pairs, especially for low count values and the most frequent hashes. Recall that the unique hashes will be in decreasing frequency from HFREQ (§6.3.4), so the most frequent hashes will have the lowest dictionary encoding which will maximize the benefit from Elias-Gamma coding.

For a given fixed bit-width input $x \in \{1,0\}^S$, let $M = \lfloor \log_2 x \rfloor$ be the highest power of 2 which can be subtracted from x (treat $x \in \mathbb{R}$ to get M). Furthermore, let $\{1,0\}^O$ be the optimal binary representation of x. Then, the Elias-Gamma coding function $\gamma : \{1,0\}^S \rightarrow \{1,0\}^T$ outputs

$$\gamma(x) = \{0\}^M || \{1,0\}^O \tag{32}$$

and $T = O + M = 2 * \lfloor \log_2 x \rfloor + 1$. Therefore, Elias-Gamma coding is only beneficial when $S > T$ which may happen when the bit-width used to represent x is high but the actual real number representation of x is low.

The final output of our algorithm will be the number of unique hashes, $\vec{u}$ (to recreate the inverse function d^{-1} for decompression), and a sequence of count/value pairs, i.e.

$$U, \vec{u}, \gamma(d(x_{s_0^*})), \gamma(n_0), \gamma(d(x_{s_1^*})), \gamma(n_1), \ldots, \gamma(d(x_{s_j^*})), \gamma(n_j) \tag{33}$$

6.3.5.2 Parameters

Parameter	Algorithm	Variable	Value(s)	Configurable?
Number of inputs	$N \in \mathbb{N}$	?	[?, 3*96?]	Yes
Fixed-point shift	–	–	–	–
Channels	–	–	96	?

6.3.5.3 Implementation

Input PE	Output
HFREQ (6.3.4)	32-bits

Output PE	Input
NPACK (§6.6.1)	8-bits

```
def hcomp(input: u32) -> u32:
    TODO
```

6.3.6 DCOMP: Decompression

Performs the inverse dictionary encoding, run-length encoding, and Elias-Gamma encoding computed in HCOMP (§6.3.5).

6.3.6.1 Algorithm

The first step is to read in the number of unique hashes U. This is assumed to be a fixed number of bits $B_U \in \mathbb{N}$. The next step is to read U number of hashes (i.e. $\vec{u} \in \mathbb{R}^U$), where each hash has a fixed number of bits $B_H \in \mathbb{N}$. Given this information, the inverse dictionary encoding can be constructed (see Equation 30). This function $d^{-1} : \mathbb{N} \rightarrow \mathbb{R}$ maps an index to the corresponding hash value and can be represented simply with $\vec{u}$.

The next step is to decode the Elias-Gamma coding. Let $\vec{b} \in \{1, 0\}^B$ be the vector of bits corresponding to the compressed data. The decoder will keep track of how many zeros it has read before reaching a one, starting from b_0. Let this be z and let $\vec{b}_z = [b_z, b_{z+1}, \ldots, b_{2z-1}]$ be the next z bits after the one. The encoded number, $x \in \mathbb{Z}$, can be reconstructed with the following:

$$x = 2^z + \sum_{i=0}^{z-1} b_z[i] 2^{z-1-i} \tag{34}$$

After reading $2z + 1$ bits, z resets back to 0 and the count begins for the next encoded number. This process is repeated for all encoded bits in $\vec{b}$. The resulting decoded set of values are the run-length count/value pairs generated from Equation 31.

Given a pair $(d(x_{s_i^*}), n_i)$, the first step is to apply d^{-1} on the value to get $x_{s_i^*}$. This value is then outputted n_i times, recreating the partitioned vectors s.t. $\vec{x_{s_0}} || \vec{x_{s_1}} || \ldots || \vec{x_{s_j}} = \vec{x} \in \mathbb{R}^N$ which is the original uncompressed vector of hashes. The algorithm will keep track of a sum of all n_i counts until it reaches a maximum number of inputs N to determine the stopping point. The output is $\vec{x}$ in the original order.

6.3.6.2 Parameters

Parameter	Algorithm	Variable	Value(s)	Configurable?
Number of expected hashes	$N \in \mathbb{N}$	?	[?, 3*96?]	Yes
Number of bits for unique count	$B_U \in \mathbb{N}$	?	?	?
Number of bits for each hash	$B_H \in \mathbb{N}$	?	?	?
Fixed-point shift	–	–	–	–
Channels	–	–	–	–

6.3.6.3 Implementation

Input PE	Output
UNPACK (6.6.2)	xx-bits

Output PE	Input
CCHECK (§6.3.7)	8-bits

```
def dcomp(input: bool) -> u8:
    TODO
```

6.3.7 CCHECK: Hash Collision Check

Checks if a stream of input hashes are similar to a pre-determined set of hashes. For each of the pre-determined hashes, the output will report if at least one input hash in the stream collided with that pre-determined hash.

6.3.7.1 Algorithm

This algorithm will take as input a template vector $\vec{t} \in \mathbb{R}^T$ and a hash stream $\vec{x} \in \mathbb{R}^N$. The output will be a bit vector $\vec{b} \in \{1, 0\}^T$ s.t. for all $0 \leq i \leq T-1$, if $b_i = 1$, then $\exists x \in \vec{x}$ s.t. $x == t_i$ for $t_i \in \vec{t}$. If $b_i = 0$, then $\forall x \in \vec{x}, x \neq t_i$. The intuition is that b_i indicates whether a hash from the incoming stream of data has collided (i.e. is equal to) with the template t_i.

6.3.7.2 Parameters

Parameter	Algorithm	Variable	Value(s)	Configurable?
Stream size	$N \in \mathbb{N}$	page_length	[?, 3* 96? 8192?]	Yes
Number of templates	$T \in \mathbb{N}$	?	[8, 256?]	Yes
Hash templates?	$\vec{t} \in \mathbb{R}^T$	?	$[-2^{??}, 2^{??} - 1]$	?
Fixed-point shift	–	–	–	–
Channels	–	–	–	–

6.3.7.3 Implementation

Input PE	Output
DCOMP (§6.3.6)	8-bits
GATE (§6.7.1)	8-bits
SC (§6.7.2)	8-bits

Output PE	Input
NPACK (§6.6.1)	8-bits

The following function will store the first num_hashes number of inputs into an array. Given page_length number of additional inputs, these inputs will be compared against the stored inputs to see if they match. If so, the indices of the collisions will be remembered and outputted in a mask after page_length number of inputs. Therefore, this function outputs a mask after page_length + num_hashes number of inputs and 0 on every other input.

```
def ccheck(input: i8) -> i8:
    static hashes[num_hashes]: [i8] = {0}
    static num_inputs: i64 = 0
    static bitmask: i8 = 0

    if num_inputs < num_hashes:
        hashes[num_inputs] = input
    else:
        j: i8 = num_hashes
        for i in 0..num_hashes:
            if hashes[i] == input:
                j = i
                break
        if j < num_hashes:
            bitmask |= (1 << j)

    num_inputs += 1
    if num_inputs == page_length + num_hashes:
        return bitmask
        bitmask = 0
    else:
        return 0
```

6.3.8 CSEL: Channel Selection

Selects, out of the available channels, a subset of those channels based on heuristics.

6.3.8.1 Algorithm

Given an input sequence of channel IDs $\vec{x} \in \mathbb{N}^N$, the algorithm will count the frequency of each channel ID w_i s.t. $0 \leq i \leq C-1$ where C is the total number of channels. Once an end of sequence marker is read or the total number of expected channel IDs have been read (i.e. N), a specified number of channel IDs will be ”released” or outputted and the frequencies w_i for these channels will be reset internally. The maximum releasable channel count is denoted as M which is $\leq C$.

6.3.8.2 Parameters

Parameter	Algorithm	Variable	Value(s)	Configurable?
Number of inputs?	$N \in \mathbb{N}$	?	?	?
EOS Marker	$E \in \mathbb{R}$	eos_marker	255	?
Channel release	$M \in \mathbb{N}$	max_chan_release	16	?
Fixed-point shift	–	–	–	–
Channels	–	num_channels	96	?

6.3.8.3 Implementation

Input PE	Output
UNPACK (§6.6.2)	8-bits

Output PE	Input
SC (§6.7.2)	8-bits

Once an input to this PE is equal to eos_marker, then there will be at most max_chan_release + 1 number of outputs. Otherwise, the number of times this input has been seen will be counted.

```
def csel(
    input: i8,
    num_channels: i32,
    max_chan_release: i32,
    eos_marker: i8
) -> i8:
    static chan_mask[num_channels]: [i32] = {0}

    if input == eos_marker:
        released: i32 = max_chan_release
        for i in 0..num_channels:
            if chan_mask[i] > 0 and released > 0:
                return i
                released -= 1
            chan_mask[i] = 0
        return -1
    else:
        chan_mask[input] += 1
```

6.4 Linear Algebra

These are PEs that perform standard tensor multiply/addition/subtract operations.

6.4.1 ADD: Matrix Adder

Adds two streaming matrices together with additional configurations to the output values such as adding a constant, performing ReLU, and normalization.

6.4.1.1 Algorithm

Given three matrices $A, B, C \in \mathbb{R}^{N\times M}$,

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nm} \end{bmatrix} \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1m} \\ c_{21} & c_{22} & \cdots & c_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nm} \end{bmatrix} \tag{35}$$

The algorithm will compute:

$$A+B+C = \begin{bmatrix} a_{11}+b_{11}+c_{11} & a_{12}+b_{12}+c_{12} & \cdots & a_{1m}+b_{1m}+c_{1m} \\ a_{21}+b_{21}+c_{21} & a_{22}+b_{22}+c_{22} & \cdots & a_{2m}+b_{2m}+c_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1}+b_{n1}+c_{n1} & a_{n2}+b_{n2}+c_{n2} & \cdots & a_{nm}+b_{nm}+c_{nm} \end{bmatrix} \tag{36}$$

In ReLU mode, the ReLU function $r : \mathbb{R} \to \mathbb{R}^+$ is applied on the output sums

$$r(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{37}$$

$$\begin{bmatrix} r(a_{11}+b_{11}+c_{11}) & r(a_{12}+b_{12}+c_{12}) & \cdots & r(a_{1m}+b_{1m}+c_{1m}) \\ r(a_{21}+b_{21}+c_{21}) & r(a_{22}+b_{22}+c_{22}) & \cdots & r(a_{2m}+b_{2m}+c_{2m}) \\ \vdots & \vdots & \ddots & \vdots \\ r(a_{n1}+b_{n1}+c_{n1}) & r(a_{n2}+b_{n2}+c_{n2}) & \cdots & r(a_{nm}+b_{nm}+c_{nm}) \end{bmatrix} \tag{38}$$

In normalization mode, the average is subtracted and the standard deviation is divided from each output. These can be pre-determined values or they can be computed on the fly.

$$\begin{bmatrix} (a_{11}+b_{11}+c_{11}-\mu)/\sigma & (a_{12}+b_{12}+c_{12}-\mu)/\sigma & \cdots & (a_{1m}+b_{1m}+c_{1m}-\mu)/\sigma \\ (a_{21}+b_{21}+c_{21}-\mu)/\sigma & (a_{22}+b_{22}+c_{22}-\mu)/\sigma & \cdots & (a_{2m}+b_{2m}+c_{2m}-\mu)/\sigma \\ \vdots & \vdots & \ddots & \vdots \\ (a_{n1}+b_{n1}+c_{n1}-\mu)/\sigma & (a_{n2}+b_{n2}+c_{n2}-\mu)/\sigma & \cdots & (a_{nm}+b_{nm}+c_{nm}-\mu)/\sigma \end{bmatrix} \tag{39}$$

6.4.1.2 Parameters

Parameter	Algorithm	Variable	Value(s)	Configurable?
Number of rows	$N \in \mathbb{N}$	row_size	[1, 96?]	Yes
Number of columns	$M \in \mathbb{N}$	col_size	[1, 96?]	Yes
ReLU mode	–	relu_mode	$\{0, 1\}$	Yes
Normalization	–	normalize	$\{0, 1\}$	Yes
Average	$\mu \in \mathbb{R}$	?	?	?
Standard deviation	$\sigma \in \mathbb{R}$	?	?	?
Fixed-point shift	–	–	–	–
Channels	–	–	–	–

6.4.1.3 Implementation

Input PE	Output
BMUL (§6.4.2)	32-bits
UNPACK (§6.6.2)	xx-bits

Output PE	Input
BMUL (§6.4.2)	32-bits

The current implementation will read in one row at a time. Specifically, the first input processed will be a matrix entry a_{ij} then the next input will be an entry b_{ij}. On the second input, the value $a_{ij} + b_{ij} + c_{ij}$ will be computed and outputted. The next summation will then be for i and $j + 1$ until $j + 1$ exceeds row_size. Then, the next row $i + 1$ will be processed.

```
def add(input: i32) -> i32:
    TODO
```

6.4.2 BMUL: Block Multiplier

Multiplies two streaming matrices together with additional configurations to the output values such as adding a constant, performing ReLU, and normalization.

6.4.2.1 Algorithm

Given three matrices $A \in \mathbb{R}^{N\times M}$, $B \in \mathbb{R}^{M\times P}$, and $C \in \mathbb{R}^{N\times P}$,

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mp} \end{bmatrix} \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{np} \end{bmatrix} \tag{40}$$

The algorithm will compute $D \in \mathbb{R}^{N\times P}$ s.t. $D = A * B + C =$

$$\begin{bmatrix} \sum_{k=1}^{M} a_{1k}b_{k1} + c_{11} & \sum_{k=1}^{M} a_{1k}b_{k2} + c_{12} & \cdots & \sum_{k=1}^{M} a_{1k}b_{kp} + c_{1p} \\ \sum_{k=1}^{M} a_{2k}b_{k1} + c_{21} & \sum_{k=1}^{M} a_{2k}b_{k2} + c_{22} & \cdots & \sum_{k=1}^{M} a_{2k}b_{kp} + c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{k=1}^{M} a_{nk}b_{k1} + c_{n1} & \sum_{k=1}^{M} a_{nk}b_{k2} + c_{n2} & \cdots & \sum_{k=1}^{M} a_{nk}b_{kp} + c_{np} \end{bmatrix} \tag{41}$$

where the ith row and jth entry for D is equal to

$$d_{ij} = \sum_{k=1}^{M} a_{ik}b_{kj} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{im}b_{mj} \tag{42}$$

for all $1 \leq i \leq N$ and $1 \leq j \leq P$. In ReLU mode, the function r from Equation 37 is applied to each entry, i.e.

$$d_{ij} = r(\sum_{k=1}^{M} a_{ik}b_{kj}) \tag{43}$$

In normalization mode, the average, μ, is subtracted and the standard deviation, σ, is divided from each output, i.e.

$$d_{ij} = (\sum_{k=1}^{M} a_{ik}b_{kj} - \mu)/\sigma \tag{44}$$

6.4.2.2 Parameters

Parameter	**Algorithm**	**Variable**	**Value(s)**	**Configurable?**
Number of rows	$N \in \mathbb{N}$	row_size	[1, 96?]	Yes
Intermediate dimension	$M \in \mathbb{N}$	?	[1, 96?]	Yes
Number of columns	$P \in \mathbb{N}$	col_size	[1, 96?]	Yes
ReLU mode	–	relu_mode	$\{0, 1\}$	Yes
Normalization	–	normalize	$\{0, 1\}$	Yes
Average	$\mu \in \mathbb{R}$	?	?	?
Standard deviation	$\sigma \in \mathbb{R}$	?	?	?
Fixed-point shift	–	–	–	–
Channels	–	–	–	–

6.4.2.3 Implementation

Input PE	**Output**
ADD (§6.4.1)	32-bits
BMUL (§6.4.2)	32-bits
INV (§6.4.3)	32-bits
SUB (§6.4.4)	32-bits
SBP (§6.1.1)	32-bits
SC (§6.7.2)	xx-bits

Output PE	**Input**
ADD (§6.4.1)	32-bits
BMUL (§6.4.2)	32-bits
INV (§6.4.3)	32-bits
SUB (§6.4.4)	32-bits
NPACK (§6.6.1)	8-bits

```
def bmul(input: i32) -> i32:
    TODO
```

6.4.3 INV: Matrix Inversion

Inverts a streaming matrix using Gauss Jordan Elimination described in [13].

6.4.3.1 Algorithm

Given a matrix $A \in \mathbb{R}^{N\times N}$,

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \tag{45}$$

The algorithm will compute

$$A^{-1} \tag{46}$$

using Gauss Jordan Elimination.

6.4.3.2 Parameters

Parameter	Algorithm	Variable	Value(s)	Configurable?
Dimension size	$N \in \mathbb{N}$	?	384?	Yes?
Fixed-point shift	–	–	–	–
Channels	–	–	–	–

6.4.3.3 Implementation

Input PE	Output
BMUL (§6.4.2)	32-bits
SC (§6.7.2)	xx-bits

Output PE	Input
BMUL (§6.4.2)	32-bits
SC (§6.7.2)	xx-bits

The implementation reads data from the NVM (§7) using SC (§6.7.2), performs the matrix inversion, and writes the data back.

```
def inv(input: u32) -> u32:
    TODO
```

6.4.4 SUB: Matrix Subtractor

Computes the difference between two streaming matrices.

6.4.4.1 Algorithm

Given two matrices $A, B \in \mathbb{R}^{N\times M}$,

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nm} \end{bmatrix} \tag{47}$$

The algorithm will compute:

$$A - B = \begin{bmatrix} a_{11} - b_{11} & a_{12} - b_{12} & \cdots & a_{1m} + b_{1m} \\ a_{21} - b_{21} & a_{22} - b_{22} & \cdots & a_{2m} - b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} - b_{n1} & a_{n2} - b_{n2} & \cdots & a_{nm} - b_{nm} \end{bmatrix} \tag{48}$$

6.4.4.2 Parameters

Parameter	Algorithm	Variable	Value(s)	Configurable?
Number of rows	$N \in \mathbb{N}$	row_size	[1, 96?]	Yes
Number of columns	$M \in \mathbb{N}$	col_size	[1, 96?]	Yes
Fixed-point shift	–	–	–	–
Channels	–	–	–	–

6.4.4.3 Implementation

Input PE	Output
BMUL (§6.4.2)	32-bits
SBP (§6.1.1)	32-bits
UNPACK (§6.6.2)	xx-bits

Output PE	Input
BMUL (§6.4.2)	32-bits

```
def sub(input: i32) -> i32:
    TODO
```

6.5 Compression

These are PEs designed to compress raw sensor data directly from the ADCs (§5.1).

6.5.1 LZ: Lempel Ziv

This is a lossless data compression encoder using the LZ77 technique. It contributes to two Lempel-Ziv pipelines in the system: LZ4 and LZMA.

6.5.1.1 Algorithm

In LZ77, an input stream is split into two segments. The first segment is the search buffer, and it contains symbols that have already been seen and processed. The second segment is the look-ahead buffer and contains new symbols that have not been processed yet. The encoder reads a symbol from the look-ahead buffer and tries to match it with symbols in the search buffer by searching backwards. If there is a match, the encoder will read more symbols from the look-ahead buffer to try and find a longer match. When the longest match is settled on, the encoder spits out a token which has a relative offset and a length of match.

This token is replaced in the stream, ideally being shorter than the matched sequence of symbols. If a match does not exist and a symbol is seen for the first time, a null token with the symbol is outputted. The LZ77 algorithm will need to output a literal, a match offset, a match length, and a flush message indicating the end of a stream.

6.5.1.2 Parameters

Parameter	Algorithm	Variable	Value(s)	Configurable?
Dictionary size	–	dict_log_size	[8, 12]	Yes
Number of searches	–	nbsearches	[1, 8]	Yes
Minmatch	–	minmatch	[4,4]	?
Fixed-point shift	–	–	–	–
Channels	–	–	–	–

6.5.1.3 Implementation

Input PE	Output
ADCs (§5.1)	16-bits

Output PE	Input
LIC (§6.5.2)	64-bits
MA (§6.5.4)	64-bits

The first step is to build up a history of inputs to compress. Once dict_log_size number of inputs have been read, the matching process begins. The number of searches that the match does is based on nbsearches.

For the matching process, the current implementation uses a hash table and chain table to efficiently look back into the search buffer. The entries in the hash table are computed by taking a position in the sequence, looking minmatch bytes ahead and hashes those values into dict_log_size bits. The name for this implementation is Morphing Match Chain (see this blog post or this blog post).

The current implementation outputs four types of messages to the next PE. These types can be characterized as CONTEXT_LITERAL, CONTEXT_MATCH, CONTEXT_MATCH_LENGTH, and CONTEXT_FLUSH. These messages are passed using the follow message format. The compression PEs use this format for communication.

Token 2	Token 1	Context 2	Context 1	Value 2	Value 1
0-16	16-32	32-40	40-48	48-56	56-64

Table 2: Compression Message Format.

The flush message sets a 1 in the Value 1 field and CONTEXT_FLUSH value in the Context 1 field. A literal message sets a 1 in the Value 1 field, a CONTEXT_LITERAL value in the Context 1 field, and the literal value in the Token 1 field. When a match occurs, one message is sent such that a 1 is set in both the Value 1 and Value 2 fields, a CONTEXT_MATCH_LENGTH value is set in Context 1, a CONTEXT_MATCH value is set in Context 2, the match length value is set in Token 1, and the match offset is set in Token 2.

```
def lz(
    input: i8,
    dict_log_size: i4,
    nbsearches: i8
) -> i64:
    static state: i32 = -1

    static history[8192]: [i8] = {0}
    static size: i32 = 8192

    static index: i32 = 0
    static tail: i32 = 0
    static src: i32 = 0
    static last_update: i32 = 0

    static hash_table: [4096] = {0} // unique patterns
    static chain_table: [4096] = {0} // collisions of patterns

    if state == -1:
        clear_tables()
        state = 0

    if state == 0:
        state = 1
        history[index] = input
        index += 1
        index = index % size
        return 0x10400000 // CONTEXT_FLUSH
    else:
        history[index] = input
        index += 1
        index = index % size

        if last_update_before_src():
            lz_insert() // TODO
            last_update = (last_update + 1) % size

        if (tail - index) == 1 || (tail == 0 and index == size):
            // no space left
        else if ((size + index - src) % size) >= LZ_MAXD:
            match = lzhc_insert_and_get_match() // TODO

            if match.want_more:
                // wait for more data
            else if match.ml < MINMATCH:
                if (src % size + 1) % size != \
                    ((head - LASTLITERALS) % size):
                    return 0x1010xx00 // CONTEXT_LITERAL
                 else:
                    // wait for more data
            else:
                // output match
                // type 1 and 2
                // CONTEXT_MATCH_LENGTH/CONTEXT_MATCH
                return 0x1132xxyy

            lz_insert() // TODO
            last_update = (src + 1) % size

        update_tail()
```

6.5.2 LIC: Linear Integer Coding

This PE completes the encoding scheme for the LZ4 pipeline (see this blog post).

6.5.2.1 Algorithm

The format of LZ4 compression is composed of sequences. Each sequence is composed of five possible sections as seen in Table 3.

Token	Optional Literal Length	Literals	Offset	Optional Match Length
8-bits	0-N bytes	0-L bytes	16-bits	0-M bytes

Table 3: LZ4 Sequence Format.

The token field is composed of two 4-bit fields. The high 4-bits indicate how many literals (in bytes) there will be in the sequence (i.e. L). The 4-bit field means that L will be between 0 and 15. To set L past 15, the optional literal length field is used. If $L = 15$, then the next byte in the sequence will be an added length to 15. If that byte is set to 255, then another byte is read in for the literal length. This means that there is a potentially unbounded literal length. Once the decoder determines the full literal length, it reads that many literal bytes.

The low 4-bits of the token field indicate the match length (in bytes). There is a base length which is added to this field's value called the minimum match (or minmatch for short) which is at least 4. Therefore, a match length of 0 in the field indicates a match of 4 bytes. This field works just like the literal length field (high 4-bits) where a value of 15 will use the next byte for additional length and so on. The match length is therefore unbounded.

After reading the literals, the decoder will read the offset field which indicates how far back to look in the decompressed stream. This offset will determine what data to copy to replace the match. The length of this copy is determined by low 4-bits of the token field plus any additional match length in the bytes after the offset.

There are a couple of rules to follow to ensure the decoding works. These rules are for speed and to ensure that the buffer length is not exceeded.

- The last sequence stops after the literals field and does not need to read in an offset.
- The last 5 bytes are always literals.
- The last match cannot start within the last 12 bytes.

6.5.2.2 Parameters

Parameter	Algorithm	Variable	Value(s)	Configurable?
Buffer size	–	lit_buff_size	256	?
Minmatch	–	minmatch	4	?
Fixed-point shift	–	–	–	–
Channels	–	–	–	–

6.5.2.3 Implementation

Input PE	Output
LZ (§6.5.1)	64-bits

Output PE	Input
AES (§6.7.3)	xx-bits
GATE (§6.7.1)	8-bits
SC (§6.7.2)	xx-bits

The implementation will decode the message coming from LZ (§6.5.1) and then encode the output in the LZ4 format described above. It will store up to lit_buf_size literals at a time. When there are more literals, it will output the literals in the buffer and flush.

```
def lic(input: u64, lit_buff_size: i32) -> u8:
    // decode
    val1: u8 = (input >> VAL1_POS) & 0xFF;
    val2: u8 = (input >> VAL2_POS) & 0xFF;
    context1: u8 = (input >> CONTEXT1_POS) & 0xFF;
    context2: u8 = (input >> CONTEXT2_POS) & 0xFF;
    token1: u16 = (input >> TOKEN1_POS) & 0xFFFF;
    token2: u16 = (input >> TOKEN2_POS) & 0xFFFF;

    if val1:
        encode_sequence(context1, token1)
    if val2:
        encode_sequence(context2, token2)
```

```

def encode_sequence(context: u8, token: u16):
    static buffer[lit_buff_size]: [u8] = {0}
    static counter: u16 = 0
    static state: u8 = 0
    static match_len: u8 = 0

    if context == CONTEXT_LITERAL:
        // TODO

    if context == CONTEXT_MATCH:
        // TODO

    if context == CONTEXT_MATCH_LENGTH:
        // TODO

    if context == CONTEXT_FLUSH:
        // TODO
```

6.5.3 TOK: Tokenizer

The purpose of this PE is to encode inputs, specifically from DWT (§6.1.3), in a format that MA (§6.5.4) understands, which is also the format that LZ (§6.5.1) outputs.

6.5.3.1 Algorithm

The format of the compression messages in the system are based on Table 2. Given an input, the tokenizer will encode the two bytes in the 64-bit message by placing the input value in the Token 1 field. It will then set the context value to either the CONTEXT_DWT_PRED, CONTEXT_DWT_UP, or CONTEXT_DWT_FLUSH message types. Since the only input connection to TOK is DWT (§6.1.3), these are the only options. However, these message types may be more configurable in the future. The Value 1 field is also set to a 1 as an indication that this message has a valid value.

6.5.3.2 Parameters

Parameter	Algorithm	Variable	Value(s)	Configurable?
Block size	–	blocksize	[1, 20]	Yes
Fixed-point shift	–	–	–	–
Channels	–	–	–	–

6.5.3.3 Implementation

Input PE	Output
DWT (§6.1.3)	16-bits

Output PE	Input
MA (§6.5.4)	64-bits

Given the two outputs from DWT (§6.1.3) which are either predictions or updates, the tokenizer will place the prediction and the update in their own independent 64-bit message. Once the amount of pairs exceeds the blocksize, then a flush message will be sent.

```
def tok(input: i16, block_log_size: i8) -> u64:
    static counter: i32 = 0
    static state: i32 = DWT_PRED_STATE
    static blocksize: i32 = 1 << block_log_size;

    if state == DWT_PRED_STATE:
        state = DWT_UP_STATE
        encode_sequence(CONTEXT_DWT_PRED, input)
    else:
        state = DWT_PRED_STATE
        encode_sequence(CONTEXT_DWT_UP, input)
        counter += 1
        if counter >= blocksize:
            encode_sequence(CONTEXT_DWT_FLUSH, 0xdead)
            counter = 0

def encode_sequence(context: u64, token: u32):
    out: u64 = 1 << VAL1_POS
    out += context << CONTEXT1_POS
    out += token << TOKEN1_POS
    return out
```

6.5.4 MA: Markov Chain

This PE implements part of the LZMA and DWTMA compression pipelines. It receives encoded messages (see Table 2) from LZ (§6.5.1) and TOK (§6.5.3). TOK's purpose is to encode data from DWT (6.1.3). Therefore, this PE runs in two modes: LZ or DWT.

6.5.4.1 Algorithm

The goal of MA is to create a "probability distribution" of the symbols to send to RC (§6.5.5). Given a set of unique symbols, MA starts out by assuming that each of these are equally likely. As symbols come in from an input stream, their probabilities change and this is represented in a range (i.e. low and high values). Connected with this range is a count of the total number of symbols seen.

6.5.4.2 Parameters

Parameter	Algorithm	Variable	Value(s)	Configurable?
Mode	–	mode	[LZ, DWT]	Yes
LZ dictionary size	–	lz_dict_logsize	[8?, 12]	?
Literal symbol size	–	literal_symbol_size	257?	?
Header symbol size	–	header_symbol_size	3?	?
DWT prediction size	–	dwt_pred_size	257?	?
DWT update size	–	dwt_up_size	256?	?
Fixed-point shift	–	–	–	–
Channels	–	–	–	–

6.5.4.3 Implementation

Input PE	Output
LZ (§6.5.1)	64-bits
TOK (§6.5.3)	64-bits

Output PE	Input
RC (§6.5.5)	64-bits

Before any computation, the PE will be set to either LZ (§6.5.1) mode or DWT (§6.1.3) mode using the mode parameter. These input PEs encode messages based on their functionality and this PE needs to take that into account.

The implementation will initialize four tables: one for literals, lengths, offsets, and headers. For the LZ mode, the size of the literals table will be set to literal_symbol_size , the size of the header table will be set to header_symbol_size, and the size of the length/offset table will be set to the maximum LZ dictionary size (i.e. $2^{\text{lz\_dict\_logsize}}$). For the DWT mode, only two tables will be used: length and offset. The size of the length table will be set to dwt_pred_size and the offset table to dwt_up_size. For each table, MA will create different "probability distributions." These distributions are presented as a set of ranges and frequencies for each relevant symbol.

Depending on the message type and the mode, the implementation will do the appropriate encoding step. This PE will output a message to RC (§6.5.5) which includes a context field, symbol low value, symbol high value, and a total value for each byte in the input (i.e. 16-bit values will be split into two bytes each with their own message).

```
def ma(input: u64, mode: bool) -> u64:
    // decode
    val1: u8 = (input >> VAL1_POS) & 0xFF;
    val2: u8 = (input >> VAL2_POS) & 0xFF;
    context1: u8 = (input >> CONTEXT1_POS) & 0xFF;
    context2: u8 = (input >> CONTEXT2_POS) & 0xFF;
    token1: u16 = (input >> TOKEN1_POS) & 0xFFFF;
    token2: u16 = (input >> TOKEN2_POS) & 0xFFFF;

    if val1:
        encode_sequence(context1, token1)
    if val2:
        encode_sequence(context2, token2)

def encode_sequence(context: u8, token: u16):
    if context == CONTEXT_DWT_PRED:
        // TODO

    if context == CONTEXT_DWT_UP:
        // TODO

    if context == CONTEXT_DWT_FLUSH:
        // TODO

    if context == CONTEXT_LITERAL:
        // TODO

    if context == CONTEXT_MATCH:
        // TODO

    if context == CONTEXT_MATCH_LENGTH:
        // TODO

    if context == CONTEXT_FLUSH:
        // TODO

def encode(context: u64, low: u64, high: u64, total: u64):
    out: u64 = 0
    out += context << 48
    out += low << 32
    out += high << 16
    out += total
    return out
```

6.5.5 RC: Range Coding

This is an entropy coding technique based on arithmetic coding. This PE is used for the LZMA and DWTMA compression pipelines (see this blog post and [9]).

6.5.5.1 Algorithm

This algorithm assumes a probability distribution of the underlying symbols and then encodes them in a single number. This technique assigns a subspace on a number line to a symbol. Instead of subdiving equally, subdivision is done by the probability of each symbol occurring in the sequence.

6.5.5.2 Parameters

Parameter	Algorithm	Variable	Value(s)	Configurable?
Number of state bits	–	s_bits	24?	?
Fixed-point shift	–	–	–	–
Channels	–	–	–	–

6.5.5.3 Implementation

Input PE	Output
MA (§6.5.4)	64-bits

Output PE	Input
AES (§6.7.3)	xx-bits
GATE (§6.7.1)	8-bits
SC (§6.7.2)	xx-bits

The implementation assumes messages directly from MA (§6.5.4) for each byte of data. These messages include a context field, symbol low value, symbol high value, and a total value.

```
def rc(input: u64) -> u8:
    context: u8 = (input >> 48) & 0xFF
    low: u16 = (input >> 32) & 0xFFFF
    high: u16 = (input >> 16) & 0xFFFF
    total: u16 = input & 0xFFFF

    range_coder(context, low, high, total)

def range_coder(context: u8, low: u16, high: u16, total: u16):
    static encoder: range_encoder = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ,0}
    static init: bool = false

    if !init:
        init = true
        range_encoder_init() //TODO
    if context == FINISH_CONTEXT:
        finish_encode() // TODO
        range_encoder_init() // TODO
    else:
        write_encode(low, high, total) // TODO
```

6.6 Networking

These are PEs that interact directly with the network through the radios (§8). They involve packing data to send to another device's PE and also unpacking that data on the receiving end.

6.6.1 NPACK: Network Packing

This PE will take a specified number of bytes and warp them into a packet with a header and CRC-32 checksums.

6.6.1.1 Algorithm

For the first input, the algorithm expects the number of data elements that will be sent in the packet up to a specified amount (in this case the maximum is 240 bytes). Then, the header will be constructed as follows:

Source ID	Destination ID	Time	Packet Length	CRC Checksum
8-bits	8-bits	32-bits	8-bits	32-bits

Table 4: Packet header format.

Then, up to 240 bytes of data will be placed in the packet with an additional 32-bit CRC checksum computed on the data. In total, with the header and checksums, each packet can be at most 255 bytes.

Computing the checksums are done using the CRC-32 Sarwate algorithm which includes a pre-computed lookup table. Documentation for this algorithm can be found here.

6.6.1.2 Parameters

Parameter	Algorithm	Variable	Value(s)	Configurable?
Source ID	–	?	?	Yes
Destination ID	–	?	?	Yes
Time format	–	?	?	?
CRC32 Table	?	?	$[2^{-31}, 2^{31} - 1]^{256}$	Yes
Fixed-point shift	–	–	–	–
Channels	–	–	–	–

6.6.1.3 Implementation

Input PE	Output
AES (§6.7.3)	xx-bits
BMUL (§6.4.2)	32-bits
CCHECK (§6.3.7)	8-bits
HCOMP (§6.3.5)	32-bits
SBP (§6.1.1)	32-bits
SC (§6.7.2)	xx-bits
SVM (§6.2.1)	32-bits

Output PE	Input
UNPACK (§6.6.2)	xx-bits

The current implementation only computes a CRC-32 checksum on a specified number of inputs.

```
def npack(input: u8) -> u32:
    TODO
```

6.6.2 UNPACK: Network Unpacking

This PE unpacks a packet from another node's NPACK (§6.6.1) PE and verifies that the data has not been corrupted due to transmission.

6.6.2.1 Algorithm

The algorithm will assume a packet with a header based on Table 4. This means that it will read a source ID, destination ID, time, packet length, and CRC checksum. It will verify the checksum using the CRC-32 check which resembles the original CRC-32 algorithm and requires the same generated table of values. Then, it will expect a specified number of bytes based on the packet length with an additional CRC checksum for the data. This checksum will also be verified to determine if the data has any corruption. The output will of the algorithm will be the data in the packet.

6.6.2.2 Parameters

Parameter	Algorithm	Variable	Value(s)	Configurable?
Time format	–	?	?	?
CRC32 Table	?	?	$[2^{-31}, 2^{31} - 1]^{256}$	Yes
Fixed-point shift	–	–	–	–
Channels	–	–	–	–

6.6.2.3 Implementation

Input PE	Output
NPACK (§6.6.1)	32-bits

Output PE	Input
ADD (§6.4.1)	32-bits
CSEL (§6.3.8)	8-bits
DCOMP (§6.3.6)	8-bits
DTW (§6.1.4)	16-bits
SUB (§6.4.4)	32-bits
SVM (§6.6.2)	32-bits

The HLS code is currently unavailable.

```
def unpack(input: u32) -> xx:
    TODO
```

6.7 Miscellaneous

6.7.1 GATE: Buffer

Buffers data, mainly from compression PEs (§6.5), and can also "drop" or remove buffered data if necessary and prevent it from being passed on to the next PE.

6.7.1.1 Algorithm

This algorithm simply accepts incoming data if its fixed-size buffer has available space. If not, it will stall and slowly open up space in the buffer. If data has been stored in the buffer, then that data should be forwarded when possible and the buffer space freed. Additionally, each element in the buffer has a bit, called a mask, indicating whether data should be dropped or forwarded.

6.7.1.2 Parameters

Parameter	Algorithm	Variable	Value(s)	Configurable?
Buffer size	$N \in \mathbb{N}$	buffer_size	1024?	No
Block size	$M \in \mathbb{N}$	block_size	[2, 1024]	Yes
Fixed-point shift	–	–	–	–
Channels	–	–	–	–

6.7.1.3 Implementation

Input PE	Output
EMDH (§6.3.3)	16-bits
LIC (§6.5.2)	8-bits
NEO (§6.1.2)	32-bits
NGRAM (§6.3.3)	8-bits
RC (§6.5.5)	8-bits
SC (§6.7.2)	xx-bits
THR (§6.2.2)	1-bit

Output PE	Input
AES (§6.7.3)	xx-bits
CCHECK (§6.3.7)	8-bits
HFREQ (§6.3.4)	32-bits
SC (§6.7.2)	xx-bits

There are two input channels which this PE listens to. One is for data to buffer and the other is for a mask which indicates which data to drop or forward to the next PE. A single mask value gets replicated block_size number of times. Certain conditions must be fulfilled before this PE consumes data or a mask value. Data will be outputted based on what is currently in the buffer and what the mask value is.

This PE is unique in its implementation since it accepts input from the FIFO buffer based on a condition as opposed to always accepting inputs and processing them.

```
// Gets called only when the buffer has available space
def gate_input(input: i8) -> i8:
    static buffer[buffer_size]: [i8] = {0}
    static mask_buffer[buffer_size]: [bool] = {0}

    static buffer_head: i32 = 0
    static mask_head: i32 = 0
    static index: i32 = 0
    static init: i32 = 0

    buffer[buffer_head] = input
    buffer_head += 1

    if buffer_head == buffer_size:
        buffer_head = 0

// TODO
def gate_mask(input: bool):
    pass

// TODO
def gate_output():
    pass
```

6.7.2 SC: Storage Controller

This is the PE which manages accesses to the on-chip NVM (§7). It is used for both reading and writing data.

6.7.2.1 Algorithm

For writes, the algorithm is to simply receive inputs up to a fixed-size page, in this case 4KB, and then write that data to an address in the NVM. For reads, the algorithm will need a starting address, and it will read up to a fixed-size page (i.e. 4KB) at that base offset. Depending on how much data is necessary and whether you need hashes or raw signals, different data sizes and counts may be specified.

6.7.2.2 Parameters

Parameter	Algorithm	Variable	Value(s)	Configurable?
Page size	–	?	?	Yes
Fixed-point shift	–	–	–	–
Channels	–	–	–	–

6.7.2.3 Implementation

Input PE	Output
ADCs (§5.1)	16-bits
CSEL (§6.3.8)	8-bits
EMDH (§6.3.3)	16-bits
GATE (§6.7.1)	8-bits
INV (§6.4.3)	32-bits
LIC (§6.5.2)	8-bits
NGRAM (§6.3.2)	32-bits
RC (§6.5.5)	8-bits

Output PE	Input
AES (§6.7.3)	xx-bits
BMUL (§6.4.2)	32-bits
CCHECK (§6.3.7)	8-bits
DTW (§6.1.4)	16-bits
GATE (§6.7.1)	8-bits
INV (§6.4.3)	32-bits
NPACK (§6.6.1)	8-bits

SC has SRAM to buffer writes before sending them to the NVM as 4KB pages. This SRAM is also used to reorganize the data layout to improve reads. There are also registers to store metadata (e.g. the last written page) to speedup access.

```
def sc(input: xx) -> xx:
    TODO
```

6.7.3 AES: Advanced Encryption Standard

Performs AES-128 encryption ECB mode.

6.7.3.1 Algorithm

AES encryption consists of 10 rounds of processing for 128-bit keys. Each round of processing consists of a single byte substitution step, a row-wise permutation step, a column-wise mixing step, and the addition of the round key.

6.7.3.2 Parameters

Parameter	Algorithm	Variable	Value(s)	Configurable?
Encryption key	–	?	$[?]^{128}$	Yes
Fixed-point shift	–	–	–	–
Channels	–	–	–	–

6.7.3.3 Implementation

Input PE	Output
GATE (§6.7.1)	8-bits
LIC (§6.5.2)	8-bits
RC (§6.5.5)	8-bits
SC (§6.7.2)	xx-bits

Output PE	Input
NPACK (§6.6.1)	8-bits

```
def aes(input: xx) -> xx:
    TODO
```

6.7.4 INTLVR: Interleaver

This will buffer data from the ADCs (§5.1) and rearrange the ordering such that certain PEs, like FFT (§6.1.7) or LZ (§6.5.1), can operate on a window of samples per channel.

6.7.4.1 Algorithm

For a given sequence of input data from different channels (up to C) for a window size N (see Figure 1), i.e.

$$x_1[1], x_2[1], \ldots, x_C[1], x_1[2], x_2[2], \ldots, x_C[2], \ldots, x_1[N], x_2[N], \ldots, x_C[N] \quad (49)$$

The interleaver will rearrange the sequence to be

$$x_1[1], x_1[2], \ldots, x_1[N], x_2[1], x_2[2], \ldots, x_2[N], \ldots, x_C[1], x_C[2], \ldots, x_C[N] \quad (50)$$

in fixed-size windows of sample corresponding to a single channel (see Figure 2).

6.7.4.2 Parameters

Parameter	**Algorithm**	**Variable**	**Value(s)**	**Configurable?**
Window size	$N \in \mathbb{N}$	?	?	?
Fixed-point shift	–	–	–	–
Channels	$C \in \mathbb{N}$	?	?	?

6.7.4.3 Implementation

Input PE	**Output**
ADCs (§5.1)	16-bits

Output PE	**Input**
BBF (§6.1.6)	16-bits
DWT (§6.1.3)	16-bits
FFT (§6.1.7)	16-bits

```
def intlvr(input: u16) -> u16:
    TODO
```

7 Storage

Each SCALO had has 128 GB NVM storage. This storage is split into different partitions for signals, hashes, and application data. When full, the oldest partition is overwritten. The storage is co-designed such that signals from a channel are stored continuously to improve access times. Reads are 10x faster but writes are 5x slower. The SC (§6.7.2) PE manages access to the NVM.

The NVM uses 4 KB page sizes and 1 MB block sizes. Operations can read 8 bytes, write a page, or erase a block. NVSim [12] is used to model the NVM.

The SLC NAND erase time is set to 1.5 ms, program time to 360 us, and voltage to 2.7V based on industrial technical reports. A low power transistor type with a temperature of 40°C is selected. NVSim estimates that the leakage power is 0.26 mW and dynamic energies of 918.809 nJ and 1374 nJ per page for reads and writes respectively. These parameters are used to size the SC (§6.7.2) buffers to 24 KB.

8 Network

SCALO nodes communicate with the outside world using the external radio and communicate between other SCALO nodes with the intra-BCI radio. Clock synchronization is also performed to handle clock drift due to the distributed nature of the system.

8.1 Intra-BCI Radio

This is a radio for intra-BCI communication that has a custom protocol with TDMA. The radio can transmit up to 20 cm both for receiving and transmitting at 7 Mbps at 4.12 GHz, consuming 1.721 mW. This radio also has a bit error rate (BER) of $< 10^{-5}$.

Packets in this intra-BCI network have 84-bit headers with a variable data size up to a maximum of 256 bytes. The header and data have 32-bit CRC32 checksums. On an error, the receiver drops the packet if it contains hashes, but uses it if it has signals.

8.2 External Radio

Allows for communication with external devices, both transmitting and receiving, up to 10 meters. This is done with 46 Mbps at 250 MHz, consuming 9.2 mW.

8.3 Clock Synchronization

SCALO synchronizes once a day using SNTP. For SNTP, one SCALO node is designated as the server. All other nodes send synchronization messages to the server which includes previously synchronized clock times and current times. The server also sends its times to the clients. Clocks are adjusted based on the different of these values. This process repeats until all clocks are synchronized to the intended precision (few μs). The intra-BCI radio is paused to all other tasks during clock synchronized but local computation can continue.

9 On-Chip Network

Given the kernel-level decomposition of the applications (§2), there is a well-defined dataflow for possible pipelines. To improve power consumption and provide some degree of flexibility, the interconnect is co-designed with the PEs with NoC route selection which provides a reconfigurable circuit-switched network on an asynchronous communication fabric. All circuit-switched interconnect components, including FIFOs and synchronizers, consume a total of 1.1 mW [16].

These programmable circuit switches between PEs allow for the output of a source PE to flow to the input of a target PE. The MC (§4) can also configure programmable switches in the interconnect to receive outputs from any source PE. This means that there are programmable switches for the output of every PE which can route data to the MC.

For communication, the network uses asynchronous SEND-ACK communication over a 8-bit data bus. The receiver ACKs once it has received the input and is ready to receive new data. There is an interconnect wrapper which provides a FIFO interface for the input and output of each PE. The interconnect sends messages in streams of bytes, bits, and tokens (packets of multiple values).

9.1 FIFO Buffers

There are per-PE FIFO buffers which hold incoming data and transform output data. These buffers act as logical adapters which transform data over the network into a format understandable to the PE. These adapters also modify outputs to match the fixed width interface of the interconnect.

9.2 Synchronizers

There are two flip-flop synchronizers at the sender and receiver PEs to communicate between different clock domains. There is a high-frequency connection (150 MHz) between the PEs with two additional pairs of synchronizers for the sender and receiver [16]. Given this design, sending and receiving data typically takes around two sender and receiver clock cycles.

9.3 PE Connection Graph

The following is a graph representation of the PEs in a single SCALO node and their input/output connections.

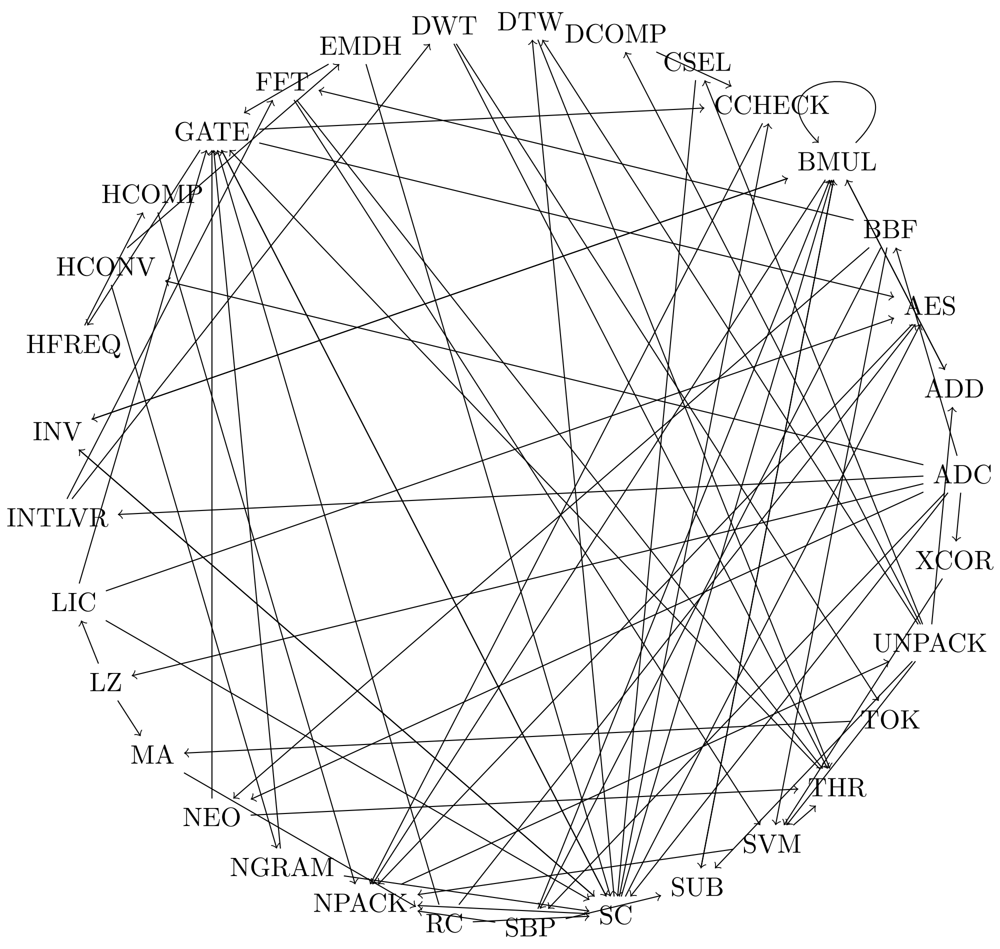

Figure 4: PE connection graph.

10 Scheduler

There exists an ILP-based scheduler to reconfigure the programmable switches on each SCALO node to realize distributed applications. The ILP's objective is to maximize the number of channels processed given power and latency constraints of some application pipeline.

11 Battery

SCALO nodes are assumed to be wirelessly powered. During charging, all pipelines are paused to prevent overheating. Charging frequency and duration varies by algorithm and battery technology.

12 Questions and TODOs

12.1 Constraints

- Where exactly does the 15 mW power budget come from? From previous discussions, it comes from the FDA and IEEE, but this should be double-checked and cited properly. Based on [8], they roughly estimate that the power budget is ∼40mW. Other studies cited in the HALO paper have different budgets.
- Similarly, for seizure detection and stimulation, do we have a citation for the 10 ms latency constraint?
- How does the power overshoot mechanism work? What PEs are able to be interrupted in case there is a power overshoot? How does this impact end-to-end applications?

12.2 Microcontroller

- How is code uploaded to the MC? How does it know what code to run and when?
- How do parameter and pipeline configuration instructions look like? What is their format?
- What are the full set of changes to the MC from the original implementation? How can we document this properly?
- The MC can configure PE outputs to route to itself. Where are those values stored? Is there a limit to how many PE outputs it can reconfigure to itself? Are there different routines for different outputs?
- Each PE supports multiple frequencies which is configurable. This is based on a simple state machine and counter which runs at low power (μWs). How does this configuration work? Does the MC do this?
- Why did the frequency of the microcontroller change from 25 MHz in HALO to 20 MHz in SCALO? Is there a set of steps for determining the proper frequency of the MC?

12.3 Sensors

- 16-bit samples are broken into two 8-bit chunks when communicating over the on-chip network. Why was this done exactly? Should this be a global policy across the entire system?
- A ”window” abstraction is used for the inputs. PEs typically perform their computation one window at a time. Windows are intended to be shifted in time, with overlap. This is such that cross-boundary window analysis is possible since boundaries may miss critical events. However, introducing overlap in this way is too inefficient. Ideally, PEs are designed to give the illusion of overlap even though they process each sample only once. This has been an overlooked aspect of the system and not all PEs are designed in this way. How should shifted windows be handled? How

can data be analyzed across independent windows?

- How do the ADCs scale with channel counts? What is their power consumption as they scale?
- The ADCs require 2.88 mW of power for a single sample across all 96 electrodes. Double-check this.
- Add any additional information on the details of the ADCs. Which ADCs are being used? Do we have citations?
- How are the stimulation parameters configured for the DACs? What does the code look like? How and when is it signaled to run?
- Why are only 16 channels used for stimulation? What are the limitations of the DACs as you scale the stimulation channel count?
- In HALO the DACs consume 0.48 mW whereas in SCALO they cnsume 0.6 mW? Why is that the case?
- Can different sensors be "plugged in" to the chip? Does the silicon need to change to support for channels in the frontend? Do the ADCs need to be designed accordingly to support this?

12.4 Processing Elements

- Add the frequency/latency/power numbers for each of the PEs.
- Canonicalize the input and output bit-widths for all of the PEs given their intended connections.
- What is the interface between the PEs and the MC? The MC is able to read in the outputs from any one of the PEs, or send inputs to any one of the PEs. How does this look exactly?
- How should fixed-point arithmetic be implemented across all PEs? Is there a fixed radix point or is it configurable?
- Should PEs expect raw samples from the ADCs as 16-bits or two bytes? This would influence the logic in each of the PEs unless that logic can be implemented transparently in the on-chip network.
- Should SBP (§6.1.1) include the traditional definition of signal power? i.e. the square of the magnitude instead of the magnitude. This may be an important feature for neuroscientists but requires some additional research/surveying.
- The original paper for SBP assumes that data is filtered through BBF before being passed into SBP. This feature needs to be added since it was not included in the SCALO paper. Similarly, the original NEO (§6.1.2) paper assumes bandpass filtering before performing NEO which was not included in the SCALO paper. This paper states "in order to use the algorithm effectively when the signal consists of several frequency components, it is important to pass the signal through a bank of bandpass filters first." Both of these need to be added and re-evaluated.
- DWT (§6.1.3) in the hardware does not include multiple orders of DWT which was suppose to be included for feature generation in HALO. This needs to be added. Find citations for the maximum order and which wavelet functions would be useful.

- The Sakoe-Chiba window optimization information needs to be added for DTW (§6.1.4).
- What is the intended computation for XCOR (§6.1.5)? There are different implementations that are similar in nature but it is not clear which one the neuroscientists want. Are there citations?
- How should lag be implemented for XCOR (§6.1.5)?
- For BBF (§6.1.6), what is the configuration format for specifying the range of frequencies in the bandpass? Should they be computed in software in the form of coefficients which are sent to the hardware PE through the config bus? Should the frequencies be sent to the PE and the coefficients be computed in hardware?
- For BBF (§6.1.6), what is the maximum order of the filter? Are there citations for this value?
- The BBF hardware implementation needs to be rewritten. The current implementation is incorrect.
- What is the algorithm for the open-source implementation of FFT (§6.1.7)? What is its limit on computation? Is it the Nyquist frequency which is based on the maximum sampling rate?
- What are the parameters for FFT (§6.1.7)? My understanding is that the number of points requires at least that many sample values. Some of the algorithms require frequency components with 1024 points, requiring a large buffer in the PE. Would that be possible given the current window size of 120? How should the output frequencies be chosen? Should it be a range of frequencies, the number of points, or a sequence of frequency bins? Find citations for the parameters of the FFT implementation.
- Should filtering be applied to the signal before computing FFT (§6.1.7)? This would limit the range of frequencies in the signal which could make computing the FFT more efficient.
- SVM (§6.2.1) needs to consider overflow during the summation. How should this be handled?
- For THR (§6.2.2), the lower and upper bounds on the threshold need to be set. Should low > high be possible? This would allow THR to always output 0 if necessary.
- THR's implementation needs to be updated to include an array of inputs. What would that look like exactly? Why was it changed to include an array of inputs?
- The hardware implementation for HCONV (§6.3.1) needs to be updated to include step size for convolution and shifted window computations.
- The EMDH PE's hash is used to approximate other similarity measures such as chi-squared distance, euclidean distance, and pearson's cross correlation. Where is the proof for this? Karthik mentioned it was empirical rather than formal, but it is not clear why the claim can be made.
- HFREQ (§6.3.4) can receive hashes from both EMDH (§6.3.3) and NGRAM (§6.3.2). However, the hashes from NGRAM are 8-bits and for EMDH they are 16-bits. How should HFREQ deal with these two cases?
- HCOMP (§6.3.5) can have a low compression ratio. Should RLE and

EG coding be configurable (i.e. turn them off if necessary) to adjust the compression ratio if necessary?

- When HCOMP (§6.3.5) creates a payload that is greater than 96 bytes, it appears that the system will simply send the data uncompressed since the uncompressed upper bound is 96 bytes. How is this done exactly? The hardware implementation does not include this feature, but it seems to be assumed.
- The hardware implementation for DCOMP (§6.3.6) needs to be updated. Currently, it only does EG docoding, but HCOMP (§6.3.5) does dictionary encoding, then run-length encoding, then EG encoding. Therefore, there needs to be more changes to this PE.
- For CCHECK (§6.3.7), how many hash templates should there be? It seems that the hash templates come from another node? How many hashes are there in a given window? For how long does the local node do the collision check? In the hardware implementation there is a full "page" of hashes that can be used locally (i.e. 8192 hashes). This definitely exceeds a window size of hashes. How does this impact latency?
- The inputs to CCHECK (§6.3.7) come from two different PEs. The hash templates seem to come from UNPACK (§6.6.2) and the stream of hashes that are checked for collision seem to come from SC (§6.7.2). How are these inputs multiplexed? Do they require two different input FIFO buffers like GATE (§6.7.1)?
- The output of CCHECK is a bit vector indicating whether there was a collision with the hash, where the digit in the bit vector correspond to the hash index. This information needs to be sent back to the source node using NPACK (§6.6.1). This connection did not exist previously and needs to be added. The source node needs to UNPACK (§6.6.2) this information and send it CSEL (§6.3.8). CSEL accumulates these collision indices from all the nodes and determines which channel to select based on its own heuristics. Currently, these heuristics do not exist and it simply chooses up to 16 channels to based on whether any node detected them as a collision starting from channel 1 going upwards to 96, meaning that it is biased for the first 16 channels. This release or selection phase is also based on an EOS marker, but no PE currently generates the EOS marker, so it is not clear where it comes from.
- For CSEL, the bit vector conversion to channel ID is not done anywhere in any of the PEs. This needs to be addressed otherwise channels cannot be selected.
- For CSEL (§6.3.8), there is a comment in the hardware code that suggests adding different hueristics for choosing channels. None of these currently exist and need to be added.
- There is a fundamental problem with how the linear algebra PEs are connected. In the SCALO paper, there are multiple "copies" of these PEs and they are all connected to one another. I had assumed that there would be multiple of these copies but after talking with Karthik, he seemed to suggest that there would only be one copy and that multiple copies is not

the intention. This needs to be resolved.

- ADD (§6.4.1) and BMUL (§6.4.2) have the option of subtracting a mean from each value in the output and dividing by a standard deviation. These values are the mean and standard deviation of what exactly? Do these change dynamically over time? Are they fixed? Where do they come from? Maybe they will need to be sent into a second FIFO buffer from another PE similar to how GATE (§6.7.1) has two input buffers.
- I need to add the Gauss Jordan Elimination steps for INV (§6.4.3).
- INV (§6.4.3) uses SC (§6.4.3) to store partial computations of the matrix inversion algorithm. How does this work exactly? The hardware implementation does not have logic which offloads to SC, so it is not clear what partial computations are offloaded. This needs to be addressed. Additionally, how is the final inverted matrix sent to the next PE? How is it used exactly? If its outputs are partial and sent partially, how does the next PE deal with this? Does it need to re-order the inputs in some way?
- For INV (§6.4.3), does the inverse of the matrix always exist? What happens when it does not exist?
- For LZ (§6.5.1), the algorithm details need to be filled in. Specifically, the implementation uses a morphing match chain algorithm which is very obscure and difficult to parse since there is not much documentation about it. The output messages of LZ need to be formalized more as well. The hardware implementation code needs to be cleaned up and clarified since the implementation is very confusing.
- For LIC (§6.5.2), there is a blog post online that includes a LZ4 format. This PE basically takes inputs from LZ and converts it to this format. However, the format online includes some edge cases that need to be double-checked in the hardware implementation. It is not clear if these edge cases are implemented or necessary to implement.
- In general, for the compression PEs there is a 64-bit message format which has the potential to include two different independent values. However, these two independent values are only used when there is a LZ match where one value includes the match and the second includes the match length. Other message types do not utilize the two fields. Is this really necessary? It seems like a waste to have 64-bit messages when the full fields are only used in special situations.
- For MA (§6.5.4) and RC (§6.5.5), the hardware implementation is extremely cryptic and it is not clear how these PEs work together. There is very little documentation about how these are used for the LZMA pipeline. For example, the full name of MA is Markov Chain but the output of the PE is a range of values. It is not clear how a range of values corresponds to a Markov Chain; it is a bit misleading.
- In order to clarify the functionality of MA and RC, it would be nice to have code that decompresses the output of these PEs. Does this code exist? There does not seem to be any code for decompressing and decoding the LZMA and LZ4 pipelines. This seems problematic.
- There are a lot of magic numbers in the hardware implementation for the

compression PEs. There needs to be better documentation overall.

- For NPACK (§6.6.1), the PE simply implements checksumming using CRC-32. First off, the algorithm itself needs to be added into the documentation. It uses a look-up table resembling the Sarwate algorithm. Second off, the actual NPACK implementation needs to create a packet with a header and fields. These details do not exist in the hardware implementation and needs to be added.
- The UNPACK (§6.6.2) PE does not have a hardware implementation. It is nowhere to be found. This needs to be developed.
- The SC (§6.7.2) PE does not have a hardware implementation. It needs to be added and this is extremely critical since it is connected to so many different PEs and enables the NVM storage.
- For SC (§6.7.2), how are the addresses determined. How does a PE specify how much data to read and from where?
- For AES (§6.7.3), the algorithm needs to be documented still and requires some reading.
- The INTLVR (§6.7.4) is a core part of some of the PEs and needs to be implemented in hardware. Apparently, its implementation is scattered throughout the other code but I have not been able to confirm this.
- In addition to all the changes above, the configuration parameters for many of the PEs and their possible ranges need to be determined. The bit-widths for the inputs and outputs need to be determined. This will take time and many studies.
- The window shift assumption underpinning the design of each of the PEs needs to be agreed upon. PE microarchitecture has largely been ignored but will be impacted by the decisions of window shifting.
- The hardware implementation for most of the PEs needs to be filled in.

12.5 Storage

- How much data is stored per channel?
- What is the replacement policy for per-channel data storage?
- How is the NVM addressed? Do all requests need to go through SC (§6.7.2)?

12.6 Network

- On an error, apparently the receiver drops the packet if it contains hashes but uses if it has signals. This logic does not exist in the hardware implementations. This needs to be added.
- Where is the implementation for the clock synchronization code for the microcontroller? Where is the power numbers for this?
- What is the time format for the packets?

12.7 On-Chip Network

- How does the power consumption of the on-chip network scale with more PEs and more connections? What is its overhead? How can this be determined?
- Are there reconfigurable switches between PEs different from the reconfigurable switches to the MC? Can the output of a source PE go to both the MC and a target PE (i.e. broadcast)?
- Where do the programmable switches start? Is there a programmable switch between the ADCs and the MC? Can the raw data from the ADCs go directly to the MC? I am assuming that is the case.
- How will the FIFO buffers sitting between the PEs interact or change with the configuration decisions of the PEs? How do the clock frequencies change with new PE configurations? Do we have a way to easily test these decisions? How do we know they will not overshoot power?

12.8 Scheduler

- The ILP information needs to be added.
- The ILP is designed to maximize the number of channels. How does this impact the fixed channel count of 96? Does this mean that channels can be turned off if necessary? Who decides this.

12.9 Battery

- There needs to be more information about the battery. This requires more research.

13 Future Directions

- Could samples from the ADCs be ordered in different ways? Are there applications that require novel or unusual orderings?
- Look into the Burrows-Wheeler Transform as a PE. Is there any overlap between it and AES/CRC?

References

[1] Gibson, S., Judy, J. W., and Markovic, D. Spike sorting: The first step in decoding the brain: The first step in decoding the brain. *IEEE Signal Processing Magazine 29*, 1 (Jan. 2012), 124–143.

[2] Gorisse, D., Cord, M., and Precioso, F. Locality-sensitive hashing for chi2 distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence 34*, 2 (Feb. 2012), 402–409.

[3] Ioffe, S. Improved consistent sampling, weighted minhash and l1 sketching. In *2010 IEEE International Conference on Data Mining* (Dec. 2010), IEEE.

[4] Kaiser, J. On a simple algorithm to calculate the 'energy' of a signal. In *International Conference on Acoustics, Speech, and Signal Processing*, IEEE.

[5] Karageorgos, I., Sriram, K., Veselỳ, J., Wu, M., Powell, M., Borton, D., Manohar, R., and Bhattacharjee, A. Hardware-software co-design for brain-computer interfaces. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)* (2020), IEEE, pp. 391–404.

[6] lowRISC. Ibex risc-v core. https://github.com/lowRISC/ibex, 2019.

[7] Luo, C., and Shrivastava, A. Ssh (sketch, shingle, hash) for indexing massive-scale time series, 2016.

[8] Marblestone, A. H., Zamft, B. M., Maguire, Y. G., Shapiro, M. G., Cybulski, T. R., Glaser, J. I., Amodei, D., Stranges, P. B., Kalhor, R., Dalrymple, D. A., Seo, D., Alon, E., Maharbiz, M. M., Carmena, J. M., Rabaey, J. M., Boyden, E. S., Church, G. M., and Kording, K. P. Physical principles for scalable neural recording. *Frontiers in Computational Neuroscience 7* (2013).

[9] Moffat, A., Neal, R. M., and Witten, I. H. Arithmetic coding revisited. *ACM Transactions on Information Systems 16*, 3 (July 1998), 256–294.

[10] Mukhopadhyay, S., and Ray, G. A new interpretation of nonlinear energy operator and its efficacy in spike detection. *IEEE Transactions on Biomedical Engineering 45*, 2 (1998), 180–187.

[11] Nason, S. R., Vaskov, A. K., Willsey, M. S., Welle, E. J., An, H., Vu, P. P., Bullard, A. J., Nu, C. S., Kao, J. C., Shenoy, K. V., Jang, T., Kim, H.-S., Blaauw, D., Patil, P. G., and Chestek, C. A. A low-power band of neuronal spiking activity dominated by local single units improves the performance of brain–machine interfaces. *Nature Biomedical Engineering 4*, 10 (July 2020), 973–983.

[12] NVSim. Nvsim - a performance, energy and area estimation tool for non-volatile memory (nvm). https://github.com/SEAL-UCSB/NVSim, 2012.

[13] Quintana, E. S., Quintana, G., Sun, X., and van de Geijn, R. A note on parallel matrix inversion. *SIAM Journal on Scientific Computing 22*, 5 (Jan. 2001), 1762–1771.

[14] Sakoe, H., and Chiba, S. Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing 26*, 1 (1978), 43–49.

[15] Slutzky, M. W. Increasing power efficiency. *Nature Biomedical Engineering 4*, 10 (Oct. 2020), 937–938.

[16] Sriram, K., Karageorgos, I., Wen, X., Veselý, J., Lindsay, N., Wu, M., Khazan, L., Pothukuchi, R. P., Manohar, R., and Bhattacharjee, A. Halo: A hardware–software co-designed processor for brain–computer interfaces. *IEEE Micro 43*, 3 (2023), 64–72.

[17] Sriram, K., Pothukuchi, R. P., Gerasimiuk, M., Ugur, M., Ye, O., Manohar, R., Khandelwal, A., and Bhattacharjee, A. Scalo: An accelerator-rich distributed system for scalable brain-computer interfacing. In *2023 ACM/IEEE 50th Annual International Symposium on Computer Architecture (ISCA)* (2023), IEEE.

[18] Willsey, M. S., Nason-Tomaszewski, S. R., Ensel, S. R., Temmar, H., Mender, M. J., Costello, J. T., Patil, P. G., and Chestek, C. A. Real-time brain-machine interface in non-human primates achieves high-velocity prosthetic finger movements using a shallow feedforward neural network decoder. *Nature Communications 13*, 1 (Nov. 2022).